Компоновщик: декоратор можно считать вырожденным случаем составного объекта, у которого есть только один компонент. Однако декоратор добавляет новые обязанности, агрегирование объектов не является его целью.

Стратегия: декоратор позволяет изменить внешний облик объекта, стратегия – его внутреннее содержание. Это два взаимодополняющих способа изменения объекта.

Паттерн Facade

Название и классификация паттерна

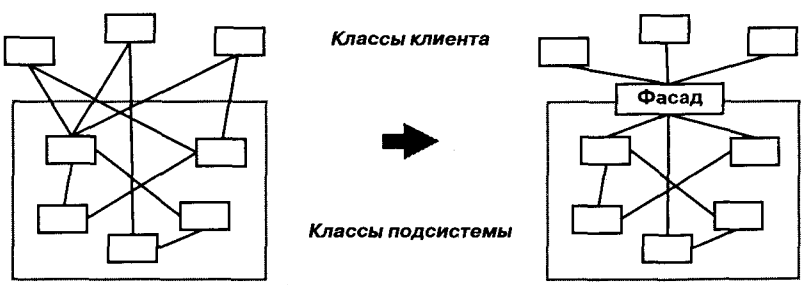
Фасад – паттерн, структурирующий объекты.

Назначение

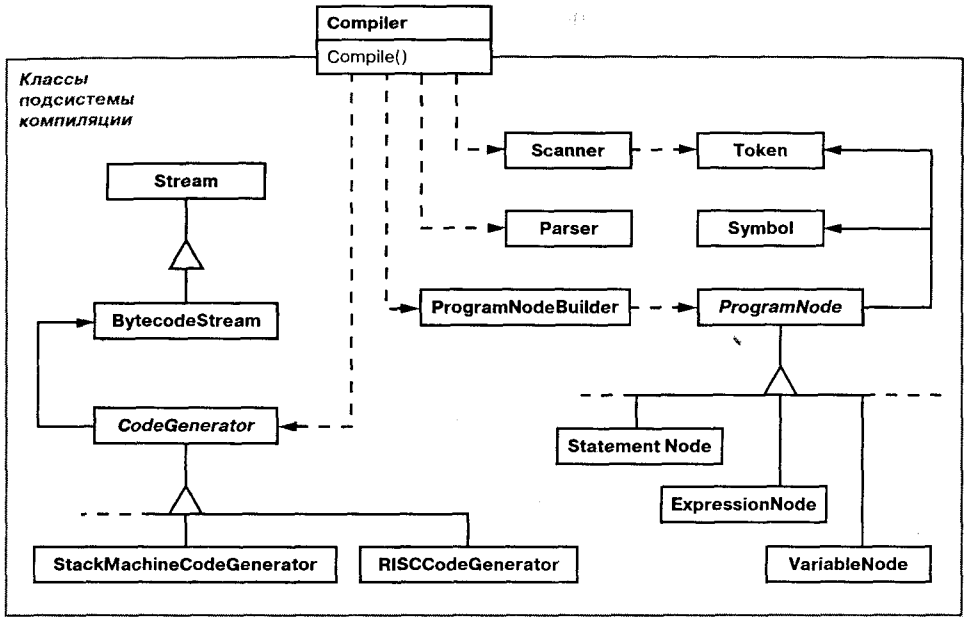
Предоставляет унифицированный интерфейс вместо набора интерфейсов некоторой подсистемы. Фасад определяет интерфейс более высокого уровня, который упрощает использование подсистемы.

Мотивация

Разбиение на подсистемы облегчает проектирование сложной системы в целом. Общая цель всякого проектирования – свести к минимуму зависимость подсистем друг от друга и обмен информацией между ними. Один из способов решения этой задачи – введение объекта фасад, предоставляющий единый упрощенный интерфейс к более сложным системным средствам.



Рассмотрим, например, среду программирования, которая дает приложениям доступ к подсистеме компиляции. В этой подсистеме имеются такие классы, как `Scanner` (лексический анализатор), `Parser` (синтаксический анализатор), `ProgramNode` (узел программы), `BytecodeStream` (поток байтовых кодов) и `ProgramNodeBuilder` (строитель узла программы). Все вместе они составляют компилятор. Некоторым специализированным приложениям, возможно, понадобится прямой доступ к этим классам. Но для большинства клиентов компилятора такие детали, как синтаксический разбор и генерация кода, обычно не нужны; им просто требуется откомпилировать некоторую программу. Для таких клиентов применение мощного, но низкоуровневого интерфейса подсистемы компиляции только усложняет задачу.



Чтобы предоставить интерфейс более высокого уровня, изолирующий клиента от этих классов, в подсистему компиляции включен также класс `Compiler` (компилятор). Он определяет унифицированный интерфейс ко всем возможностям компилятора. Класс `Compiler` выступает в роли фасада: предлагает простой интерфейс к более сложной подсистеме. Он «склеивает» классы, реализующие функциональность компилятора, но не скрывает их полностью. Благодаря фасаду компилятора работа большинства программистов облегчается. При этом те, кому нужен доступ к средствам низкого уровня, не лишаются его.

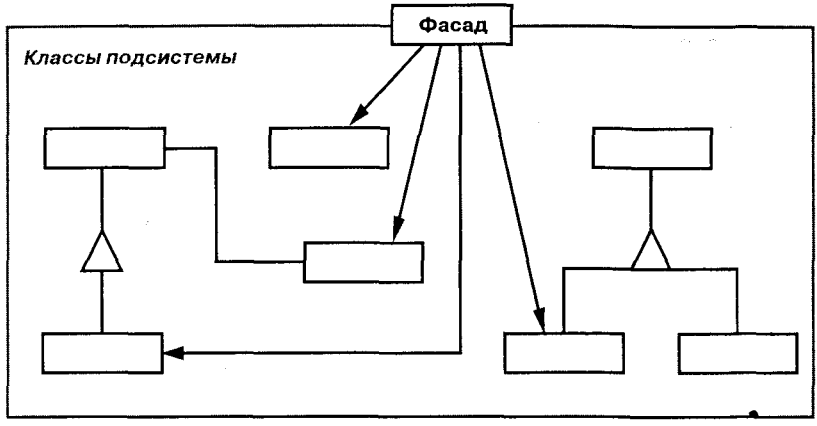
Применимость

Используйте паттерн **фасад**, когда:

- хотите предоставить простой интерфейс к сложной подсистеме. Часто подсистемы усложняются по мере развития. Применение большинства паттернов приводит к появлению меньших классов, но в большем количестве. Такую подсистему проще повторно использовать и настраивать под конкретные нужды, но вместе с тем применять подсистему без настройки становится труднее. **Фасад** предлагает некоторый вид системы по умолчанию, устраивающий большинство клиентов. И лишь те объекты, которым нужны более широкие возможности настройки, могут обратиться напрямую к тому, что находится за фасадом;
- между клиентами и классами реализации абстракции существует много зависимостей. **Фасад** позволит отделить подсистему как от клиентов, так и от других подсистем, что, в свою очередь, способствует повышению степени независимости и переносимости;

- вы хотите разложить подсистему на отдельные слои. Используйте фасад для определения точки входа на каждый уровень подсистемы. Если подсистемы зависят друг от друга, то зависимость можно упростить, разрешив подсистемам обмениваться информацией только через фасады.

Структура



Участники

- **Facade** (`Compiler`) – фасад:
  - «знает», каким классам подсистемы адресовать запрос;
  - делегирует запросы клиентов подходящим объектам внутри подсистемы;
- **Классы подсистемы** (`Scanner`, `Parser`, `ProgramNode` и т.д.):
  - реализуют функциональность подсистемы;
  - выполняют работу, порученную объектом `Facade`;
  - ничего не «знают» о существовании фасада, то есть не хранят ссылок на него.

Отношения

Клиенты общаются с подсистемой, посылая запросы фасаду. Он переадресует их подходящим объектам внутри подсистемы. Хотя основную работу выполняют именно объекты подсистемы, фасаду, возможно, придется преобразовать свой интерфейс в интерфейсы подсистемы.

Клиенты, пользующиеся фасадом, не имеют прямого доступа к объектам подсистемы.

Результаты

У паттерна фасад есть следующие преимущества:

- изолирует клиентов от компонентов подсистемы, уменьшая тем самым число объектов, с которыми клиентам приходится иметь дело, и упрощая работу с подсистемой;

- позволяет ослабить связанность между подсистемой и ее клиентами. Зачастую компоненты подсистемы сильно связаны. Слабая связанность позволяет видоизменять компоненты, не затрагивая при этом клиентов. Фасады помогают разложить систему на слои и структурировать зависимости между объектами, а также избежать сложных и циклических зависимостей. Это может оказаться важным, если клиент и подсистема реализуются независимо. Уменьшение числа зависимостей на стадии компиляции чрезвычайно важно в больших системах. Хочется, конечно, чтобы время, уходящее на перекомпиляцию после изменения классов подсистемы, было минимальным. Сокращение числа зависимостей за счет фасадов может уменьшить количество нуждающихся в повторной компиляции файлов после небольшой модификации какой-нибудь важной подсистемы. Фасад может также упростить процесс переноса системы на другие платформы, поскольку уменьшается вероятность того, что в результате изменения одной подсистемы понадобится изменять и все остальные;
- фасад не препятствует приложениям напрямую обращаться к классам подсистемы, если это необходимо. Таким образом, у вас есть выбор между простотой и общностью.

Реализация

При реализации фасада следует обратить внимание на следующие вопросы:

- *уменьшение степени связанности клиента с подсистемой.* Степень связанности можно значительно уменьшить, если сделать класс `Facade` абстрактным. Его конкретные подклассы будут соответствовать различным реализациям подсистемы. Тогда клиенты смогут взаимодействовать с подсистемой через интерфейс абстрактного класса `Facade`. Это изолирует клиентов от информации о том, какая реализация подсистемы используется.
  Вместо порождения подклассов можно сконфигурировать объект `Facade` различными объектами подсистем. Для настройки фасада достаточно заменить один или несколько таких объектов;
- *открытые и закрытые классы подсистем.* Подсистема похожа на класс в том отношении, что у обоих есть интерфейсы и оба что-то инкапсулируют. Класс инкапсулирует состояние и операции, а подсистема – классы. И если полезно различать открытый и закрытый интерфейсы класса, то не менее разумно говорить об открытом и закрытом интерфейсах подсистемы.
  Открытый интерфейс подсистемы состоит из классов, к которым имеют доступ все клиенты; закрытый интерфейс доступен только для расширения подсистемы. Класс `Facade`, конечно же, является частью открытого интерфейса, но это не единственная часть. Другие классы подсистемы также могут быть открытыми. Например, в системе компиляции классы `Parser` и `Scanner` – часть открытого интерфейса.
  Делать классы подсистемы закрытыми иногда полезно, но это поддерживается немногими объектно-ориентированными языками. И в C++, и в Smalltalk для классов традиционно использовалось глобальное пространство имен.

Однако комитет по стандартизации C++ добавил к языку пространства имен [Str94], и это позволило разрешать доступ только к открытым классам подсистемы.

Пример кода

Рассмотрим более подробно, как возвести фасад вокруг подсистемы компиляции. В подсистеме компиляции определен класс `BytecodeStream`, который реализует поток объектов `Bytecode`. Объект `Bytecode` инкапсулирует байтовый код, с помощью которого описываются машинные команды. В этой же подсистеме определен еще класс `Token` для объектов, инкапсулирующих лексемы языка программирования.

Класс `Scanner` принимает на входе поток символов и генерирует поток лексем, по одной каждый раз:

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

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

Класс `Parser` использует класс `ProgramNodeBuilder` для построения дерева разбора из лексем, возвращенных классом `Scanner`:

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

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

`Parser` вызывает `ProgramNodeBuilder` для инкрементного построения дерева. Взаимодействие этих классов описывается паттерном строитель:

```
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;
};
```

Дерево разбора состоит из экземпляров подклассов класса `ProgramNode`, таких как `StatementNode`, `ExpressionNode` и т.д. Иерархия классов `ProgramNode` – это пример паттерна **компоновщик**. Класс `ProgramNode` определяет интерфейс для манипулирования узлом программы и его потомками, если таковые имеются:

```
class ProgramNode {
public:
    // манипулирование узлом программы
    virtual void GetSourcePosition(int& line, int& index);
    // ...

    // манипулирование потомками
    virtual void Add(ProgramNode*);
    virtual void Remove(ProgramNode*);
    // ...

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

Операция `Traverse` (обход) принимает объект `CodeGenerator` (кодогенератор) в качестве параметра. Подклассы `ProgramNode` используют этот объект для генерации машинного кода в форме объектов `Bytecode`, которые помещаются в поток `BytecodeStream`. Класс `CodeGenerator` описывается паттерном **посетитель**:

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

У CodeGenerator есть подклассы, например StackMachineCodeGenerator и RISCCodeGenerator, генерирующие машинный код для различных аппаратных архитектур.

Каждый подкласс ProgramNode реализует операцию Traverse и обращается к ней для обхода своих потомков. Каждый потомок рекурсивно делает то же самое для своих потомков. Например, в подклассе ExpressionNode (узел выражения) операция Traverse определена так:

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

    ListIterator<ProgramNode*> i(_children);

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

Классы, о которых мы говорили до сих пор, составляют подсистему компиляции. А теперь введем класс Compiler, который будет служить фасадом, позволяющим собрать все эти фрагменты воедино. Класс Compiler предоставляет простой интерфейс для компилирования исходного текста и генерации кода для конкретной машины:

```
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);
}
```

В этой реализации жестко «зашит» тип кодогенератора, поэтому программисту не нужно явно задавать целевую архитектуру. Это может быть вполне разумно, когда есть всего одна такая архитектура. Если же это не так, можно было бы изменить конструктор класса Compiler, чтобы он принимал объект CodeGenerator в качестве параметра. Тогда программист указывал бы, каким генератором пользоваться при

инстанцировании объекта `Compiler`. Фасад компилятора можно параметризовать и другими участниками, скажем, объектами `Scanner` и `ProgramNodeBuilder`, что повышает гибкость, но в то же время сводит на нет основную цель фасада – предоставление упрощенного интерфейса для наиболее распространенного случая.

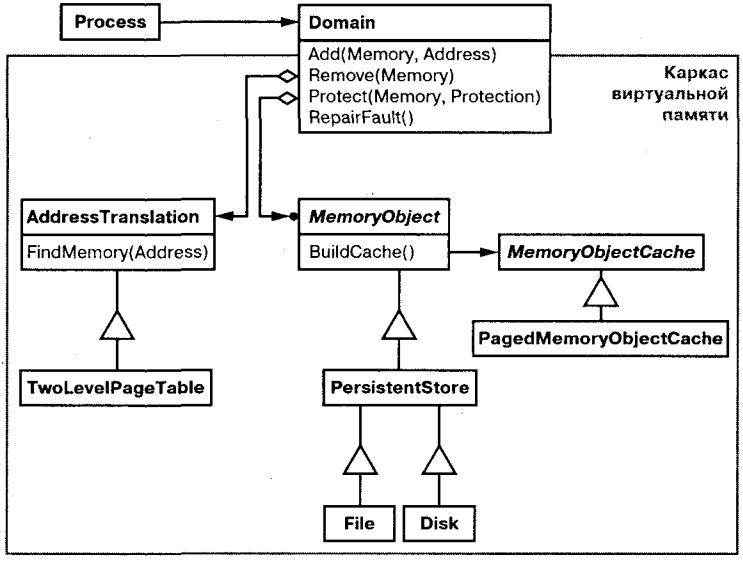
Известные применения

Пример компилятора в разделе «Пример кода» навеян идеями из системы компиляции языка ObjectWorks\Smalltalk [Par90].

В каркасе ET++ [WGM88] приложение может иметь встроенные средства инспектирования объектов во время выполнения. Они реализуются в отдельной подсистеме, включающей класс фасада с именем `ProgrammingEnvironment`. Этот фасад определяет такие операции, как `InspectObject` и `InspectClass` для доступа к инспекторам.

Приложение, написанное в среде ET++, может также запретить поддержку инспектирования. В таком случае класс `ProgrammingEnvironment` реализует соответствующие запросы как пустые операции, не делающие ничего. Только подкласс `ETProgrammingEnvironment` реализует эти операции так, что они отображают окна соответствующих инспекторов. Приложению неизвестно, доступно инспектирование или нет. Здесь мы встречаем пример абстрактной связанности между приложением и подсистемой инспектирования.

В операционной системе Choices [CIRM93] фасады используются для составления одного каркаса из нескольких. Ключевыми абстракциями в системе Choices являются процессы, память и адресные пространства. Для каждой из них есть соответствующая подсистема, реализованная в виде каркаса. Это обеспечивает поддержку переноса Choices на разные аппаратные платформы. У двух таких подсистем есть «представители», то есть фасады. Они называются `FileSystemInterface` (память) и `Domain` (адресные пространства).



Например, для каркаса виртуальной памяти фасадом служит Domain. Класс Domain представляет адресное пространство. Он обеспечивает отображение между виртуальными адресами и смещениями объектов в памяти, файле или на устройстве длительного хранения. Базовые операции класса Domain поддерживают добавление объекта в память по указанному адресу, удаление объекта из памяти и обработку ошибок отсутствия страниц.

Как видно из вышеприведенной диаграммы, внутри подсистемы виртуальной памяти используются следующие компоненты:

- MemoryObject представляет объекты данных;
- MemoryObjectCache кэширует данные из объектов MemoryObjects в физической памяти. MemoryObjectCache – это не что иное, как объект Стратегия, в котором локализована политика кэширования;
- AddressTranslation инкапсулирует особенности оборудования трансляции адресов.

Операция RepairFault вызывается при возникновении ошибки из-за отсутствия страницы. Domain находит объект в памяти по адресу, где произошла ошибка и делегирует операцию RepairFault кэшу, ассоциированному с этим объектом. Поведение объектов Domain можно настроить, заменив их компоненты.

Родственные паттерны

Паттерн абстрактная фабрика допустимо использовать вместе с фасадом, чтобы предоставить интерфейс для создания объектов подсистем способом, не зависимым от этих подсистем. Абстрактная фабрика может выступать и как альтернатива фасаду, чтобы скрыть платформенно-зависимые классы.

Паттерн посредник аналогичен фасаду в том смысле, что абстрагирует функциональность существующих классов. Однако назначение посредника – абстрагировать произвольное взаимодействие между «сотрудничающими» объектами. Часто он централизует функциональность, не присущую ни одному из них. Коллеги посредника обмениваются информацией именно с ним, а не напрямую между собой. Напротив, фасад просто абстрагирует интерфейс объектов подсистемы, чтобы ими было проще пользоваться. Он не определяет новой функциональности, и классам подсистемы ничего неизвестно о его существовании.

Обычно требуется только один фасад. Поэтому объекты фасадов часто бывают одиночками.

Паттерн Flyweight

Название и классификация паттерна

Приспособленец – паттерн, структурирующий объекты.

Назначение

Использует разделение для эффективной поддержки множества мелких объектов.