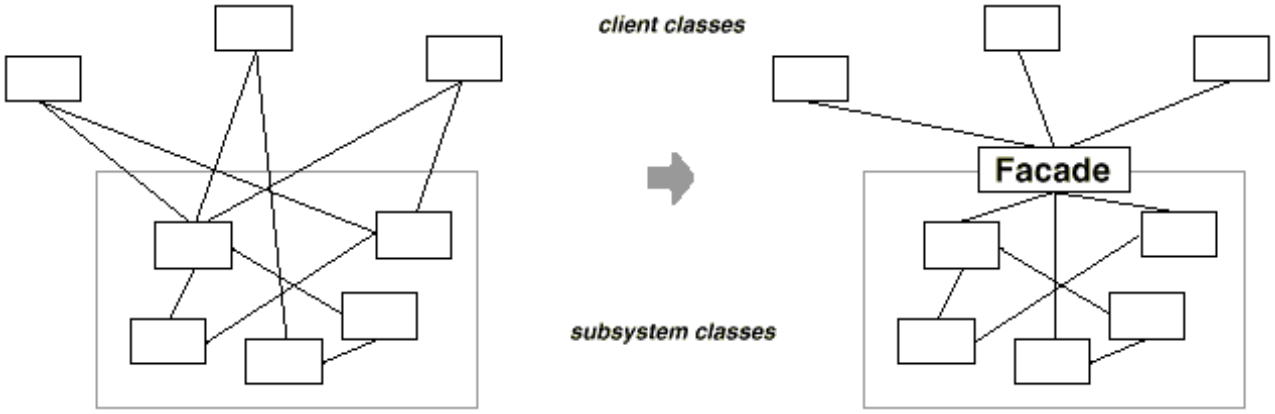
# Intent

Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

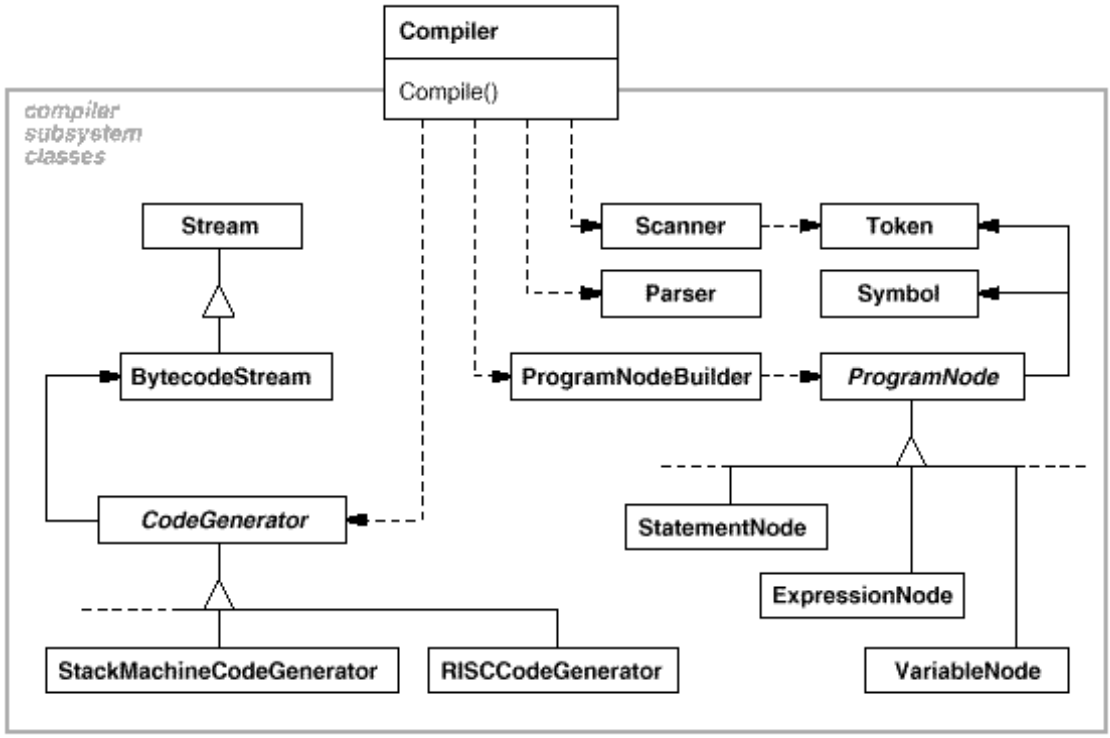
# Motivation

Structuring a system into subsystems helps reduce complexity. A common design goal is to minimize the communication and dependencies between subsystems. One way to achieve this goal is to introduce a **facade** object that provides a single, simplified interface to the more general facilities of a subsystem.




Consider for example a programming environment that gives applications access to its compiler subsystem. This subsystem contains classes such as Scanner, Parser, ProgramNode, BytecodeStream, and ProgramNodeBuilder that implement the compiler. Some specialized applications might need to access these classes directly. But most clients of a compiler generally don't care about details like parsing and code generation; they merely want to compile some code. For them, the powerful but low-level interfaces in the compiler subsystem only complicate their task.

To provide a higher-level interface that can shield clients from these classes, the compiler subsystem also includes a Compiler class. This class defines a unified interface to the compiler's functionality. The Compiler class acts as a facade: It offers clients a single, simple interface to the compiler subsystem. It glues together the classes that implement compiler functionality without hiding them completely. The compiler facade makes life easier for most programmers without hiding the lower-level functionality from the few that need it.

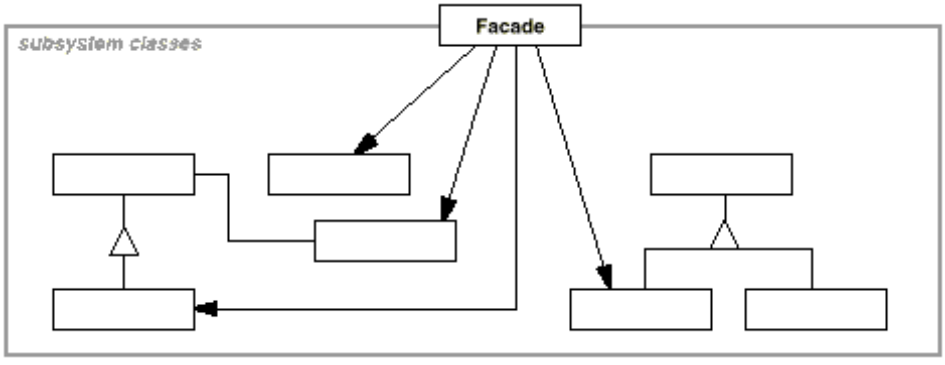



# Applicability

Use the Facade pattern when

- you want to provide a simple interface to a complex subsystem. Subsystems often get more complex as they evolve. Most patterns, when applied, result in more and smaller classes. This makes the subsystem more reusable and easier to customize, but it also becomes harder to use for clients that don't need to customize it. A facade can provide a simple default view of the subsystem that is good enough for most clients. Only clients needing more customizability will need to look beyond the facade.
- there are many dependencies between clients and the implementation classes of an abstraction. Introduce a facade to decouple the subsystem from clients and other subsystems, thereby promoting subsystem independence and portability.
- you want to layer your subsystems. Use a facade to define an entry point to each subsystem level. If subsystems are dependent, then you can simplify the dependencies between them by making them communicate with each other solely through their facades.

# Structure




# Participants

- **Facade** (Compiler)

- knows which subsystem classes are responsible for a request.
- delegates client requests to appropriate subsystem objects.

- **subsystem classes** (Scanner, Parser, ProgramNode, etc.)
  - implement subsystem functionality.
  - handle work assigned by the Facade object.
  - have no knowledge of the facade; that is, they keep no references to it.

# Collaborations

- Clients communicate with the subsystem by sending requests to Facade, which forwards them to the appropriate subsystem object(s). Although the subsystem objects perform the actual work, the facade may have to do work of its own to translate its interface to subsystem interfaces.
- Clients that use the facade don't have to access its subsystem objects directly.

# Consequences

The Facade pattern offers the following benefits:

1. It shields clients from subsystem components, thereby reducing the number of objects that clients deal with and making the subsystem easier to use.
2. It promotes weak coupling between the subsystem and its clients. Often the components in a subsystem are strongly coupled. Weak coupling lets you vary the components of the subsystem without affecting its clients. Facades help layer a system and the dependencies between objects. They can eliminate complex or circular dependencies. This can be an important consequence when the client and the subsystem are implemented independently.

   Reducing compilation dependencies is vital in large software systems. You want to save time by minimizing recompilation when subsystem classes change. Reducing compilation dependencies with facades can limit the recompilation needed for a small change in an important subsystem. A facade can also simplify porting systems to other platforms, because it's less likely that building one subsystem requires building all others.
3. It doesn't prevent applications from using subsystem classes if they need to. Thus you can choose between ease of use and generality.

# Implementation

Consider the following issues when implementing a facade:

1. *Reducing client-subsystem coupling.* The coupling between clients and the subsystem can be reduced even further by making Facade an abstract class with concrete subclasses for different implementations of a subsystem. Then clients can communicate with the subsystem through the interface of the abstract Facade class. This abstract coupling keeps clients from knowing which implementation of a subsystem is used.

   An alternative to subclassing is to configure a Facade object with different subsystem objects. To customize the facade, simply replace one or more of its subsystem objects.
2. *Public versus private subsystem classes.* A subsystem is analogous to a class in that both have interfaces, and both encapsulate something—a class encapsulates state and operations, while a subsystem encapsulates classes. And just as it's useful to think of the public and private interface of a class, we can think of the public and private interface of a subsystem.

The public interface to a subsystem consists of classes that all clients can access; the private interface is just for subsystem extenders. The Facade class is part of the public interface, of course, but it's not the only part. Other subsystem classes are usually public as well. For example, the classes Parser and Scanner in the compiler subsystem are part of the public interface.

Making subsystem classes private would be useful, but few object-oriented languages support it. Both C++ and Smalltalk traditionally have had a global name space for classes. Recently, however, the C++ standardization committee added name spaces to the language [Str94], which will let you expose just the public subsystem classes.

## Sample Code

Let's take a closer look at how to put a facade on a compiler subsystem.

The compiler subsystem defines a {BytecodeStream} class that implements a stream of `Bytecode` objects. A `Bytecode` object encapsulates a bytecode, which can specify machine instructions. The subsystem also defines a `Token` class for objects that encapsulate tokens in the programming language.

The `Scanner` class takes a stream of characters and produces a stream of tokens, one token at a time.

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

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

The class `Parser` uses a `ProgramNodeBuilder` to construct a parse tree from a `Scanner`'s tokens.

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

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

`Parser` calls back on `ProgramNodeBuilder` to build the parse tree incrementally. These classes interact according to the Builder (97) pattern.

```
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;
    };
```

The parse tree is made up of instances of `ProgramNode` subclasses such as `StatementNode`, `ExpressionNode`, and so forth. The `ProgramNode` hierarchy is an example of the Composite (163) pattern. `ProgramNode` defines an interface for manipulating the program node and its children, if any.

```
    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();
    };
```

The `Traverse` operation takes a `CodeGenerator` object. `ProgramNode` subclasses use this object to generate machine code in the form of `Bytecode` objects on a `BytecodeStream`. The class `CodeGenerator` is a visitor (see Visitor (331)).

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

`CodeGenerator` has subclasses, for example, `StackMachineCodeGenerator` and `RISCCodeGenerator`, that generate machine code for different hardware architectures.

Each subclass of `ProgramNode` implements `Traverse` to call `Traverse` on its child `ProgramNode` objects. In turn, each child does the same for its children, and so on recursively. For example, `ExpressionNode` defines `Traverse` as follows:

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

        ListIterator i(_children);

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

The classes we've discussed so far make up the compiler subsystem. Now we'll introduce a `Compiler` class, a facade that puts all these pieces together. `Compiler` provides a simple interface for compiling source and generating code for a particular machine.

```
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);
}
```

This implementation hard-codes the type of code generator to use so that programmers aren't required to specify the target architecture. That might be reasonable if there's only ever one target architecture. If that's not the case, then we might want to change the `Compiler` constructor to take a `CodeGenerator` parameter. Then programmers can specify the generator to use when they instantiate `Compiler`. The compiler facade can parameterize other participants such as `Scanner` and `ProgramNodeBuilder` as well, which adds flexibility, but it also detracts from the Facade pattern's mission, which is to simplify the interface for the common case.

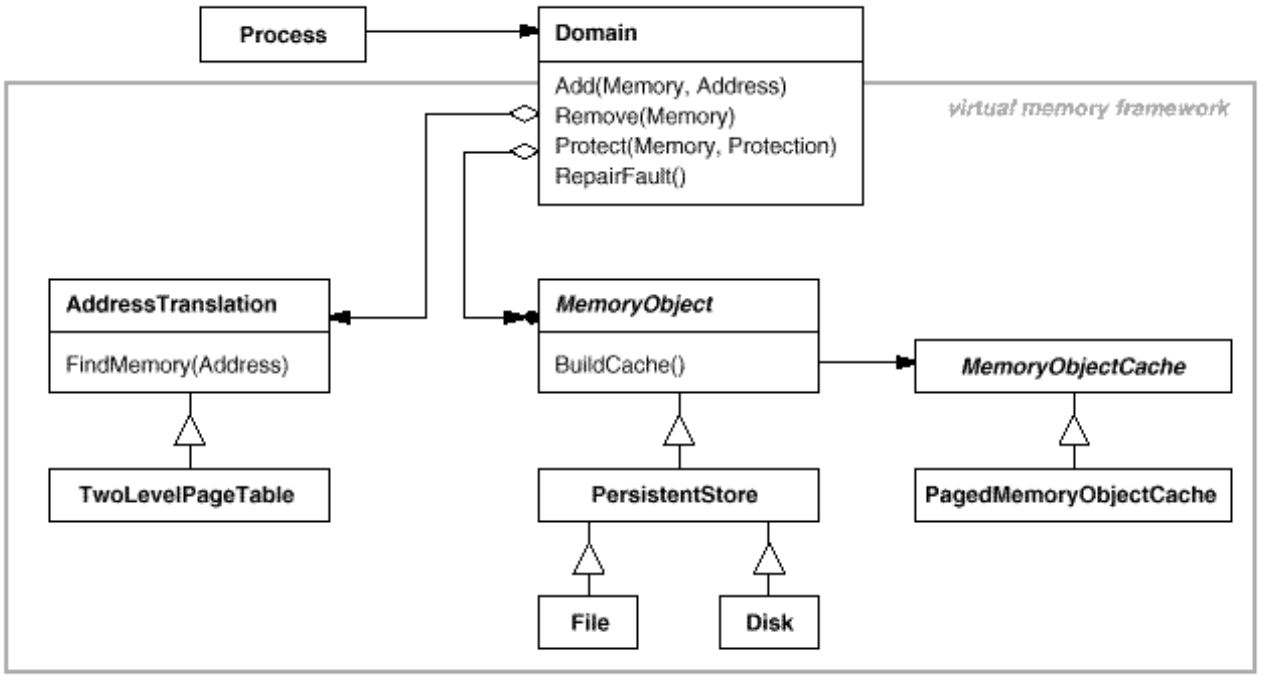
## Known Uses

The compiler example in the Sample Code section was inspired by the ObjectWorks\Smalltalk compiler system [Par90].

In the ET++ application framework [WGM88], an application can have built-in browsing tools for inspecting its objects at run-time. These browsing tools are implemented in a separate subsystem that includes a Facade class called "ProgrammingEnvironment." This facade defines operations such as InspectObject and InspectClass for accessing the browsers.

An ET++ application can also forgo built-in browsing support. In that case, ProgrammingEnvironment implements these requests as null operations; that is, they do nothing. Only the ETProgrammingEnvironment subclass implements these requests with operations that display the corresponding browsers. The application has no knowledge of whether a browsing environment is available or not; there's abstract coupling between the application and the browsing subsystem.

The Choices operating system [CIRM93] uses facades to compose many frameworks into one. The key abstractions in Choices are processes, storage, and address spaces. For each of these abstractions there is a corresponding subsystem, implemented as a framework, that supports porting Choices to a variety of different hardware platforms. Two of these subsystems have a "representative" (i.e., facade). These representatives are FileSystemInterface (storage) and Domain (address spaces).




For example, the virtual memory framework has Domain as its facade. A Domain represents an address space. It provides a mapping between virtual addresses and offsets into memory objects, files, or backing store. The main operations on Domain support adding a memory object at a particular address, removing a memory object, and handling a page fault.

As the preceding diagram shows, the virtual memory subsystem uses the following components internally:

- MemoryObject represents a data store.
- MemoryObjectCache caches the data of MemoryObjects in physical memory. MemoryObjectCache is actually a Strategy (315) that localizes the caching policy.
- AddressTranslation encapsulates the address translation hardware.

The RepairFault operation is called whenever a page fault interrupt occurs. The Domain finds the memory object at the address causing the fault and delegates the RepairFault operation to the cache associated with that memory object. Domains can be customized by changing their components.

# Related Patterns

Abstract Factory (87) can be used with Facade to provide an interface for creating subsystem objects in a subsystem-independent way. Abstract Factory can also be used as an alternative to Facade to hide platform-specific classes.

Mediator (273) is similar to Facade in that it abstracts functionality of existing classes. However, Mediator's purpose is to abstract arbitrary communication between colleague objects, often centralizing functionality that doesn't belong in any one of them. A mediator's colleagues are aware of and communicate with the mediator instead of communicating with each other directly. In contrast, a facade merely abstracts the interface to subsystem objects to make them easier to use; it doesn't define new functionality, and subsystem classes don't know about it.

Usually only one Facade object is required. Thus Facade objects are often Singletons (127).

Flyweight
Decorator

ÿ