Designing a Simple Compiler

INTERPRETER Design Pattern
- Behavioral

Motivation

- *„Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.”*

  [GoF,243]

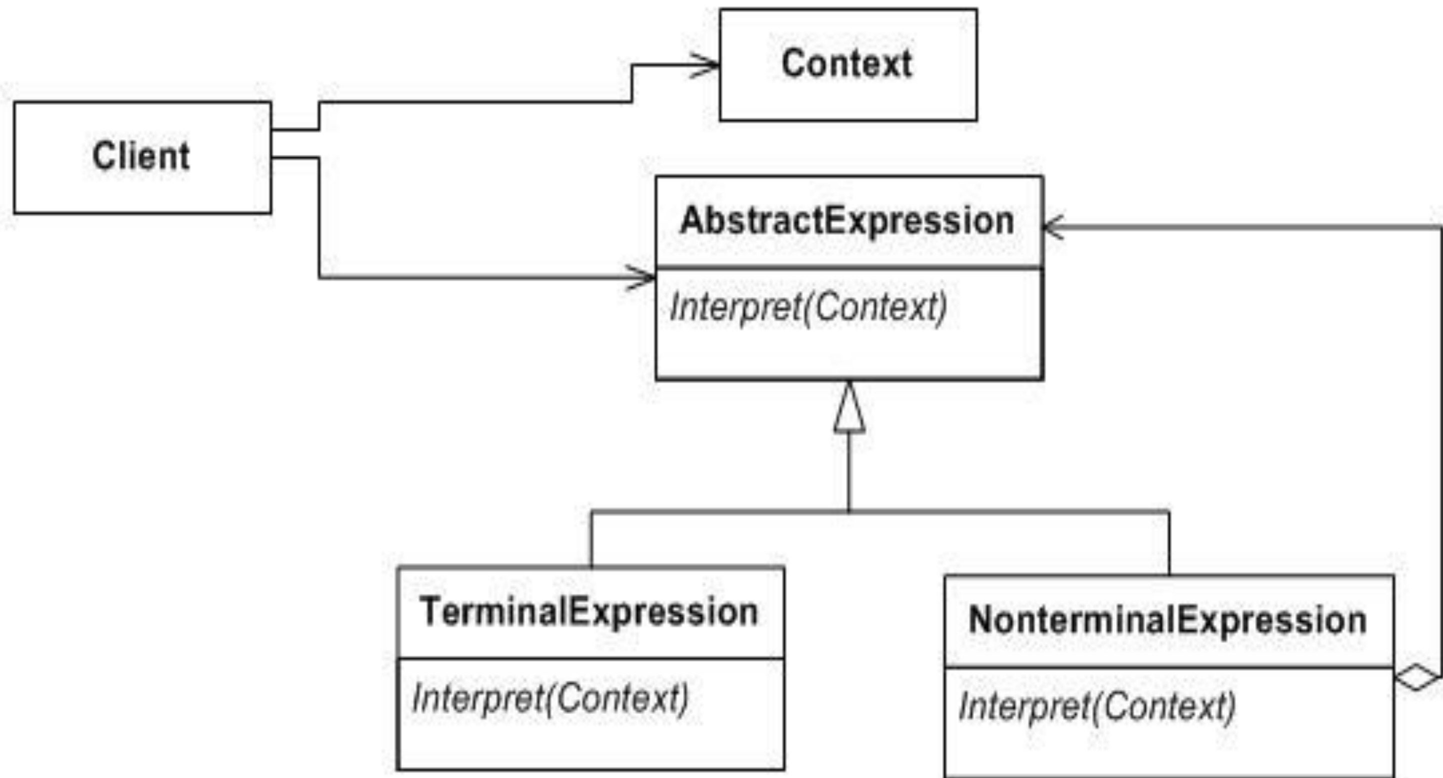
Basic Idea

- A class for each symbol, terminal, or non-terminal.
- The syntax tree of a sentence in the language is an instance of the composite pattern and is used to evaluate (interpret) the sentence.

Applicability

- Simple grammar
- Efficiency is not a critical concern
  - Most efficient way first translating the parse trees into another form [also interpreter pattern] and then interpreting it

Structure



e.g. LiteralExpression

e.g. SequenceExpression

Related Patterns

- *Composite*

  - The AST is an instance of the Composite Pattern

- *Flyweight*

  - How to share terminal symbols within the AST

- *Iterator*

  - For traversing the structure

- *Visitor*

  - For maintaining the behavoir in each node of the AST in one class

Implementation issues

- *Creating the AbstractSyntaxTree:*

- Creating the AST is not in the scope of Interpreter pattern

- *Defining the Interpret operation:*

- If defining new operations is very common, it's better to use the Visitor pattern.

- *Sharing terminal symbols with Flyweight:*

- Grammars whose sentences contain many occurrences of a terminal symbol might benefit from sharing a single copy of that symbol.

Consequences

- Easy to change and extend the grammar using inheritance.
- Implementing is easy too, since classes defining the nodes in the AST have similar implementation.
- Easily to add new ways to interpret expressions, by defining new operations in the expression classes. [Visitor]
- Complex grammars are hard to mantain !

Conclusions

- The Interpreter pattern has a limited area where it can be applied:

  - For parsing light expressions defined in simple grammars

- Interpreter Pattern is useful in terms of formal grammars but in this area there are better solutions, this is why this pattern is not so frequently used.

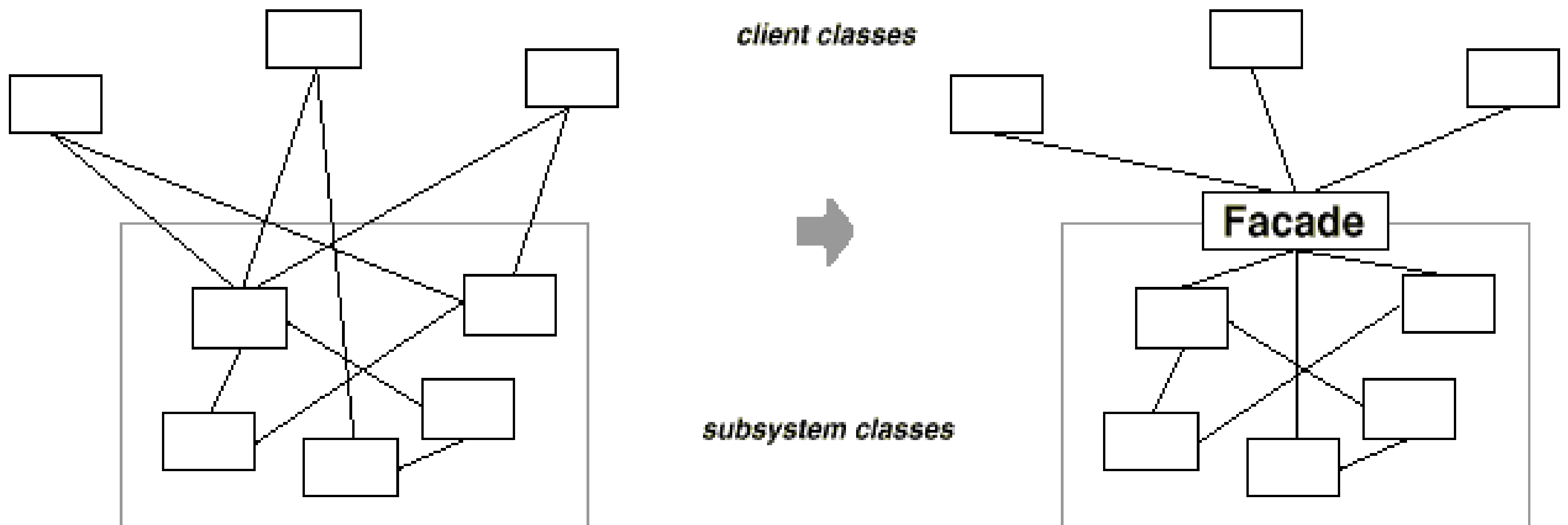
FAÇADE
Structural Pattern

Intent

Provides a unified interface to a set of interfaces of a subsystem.

Façade defines a higher-level interface that makes the subsystem easier to use. This can be used to simplify a number of complicated object interactions into a single interface.

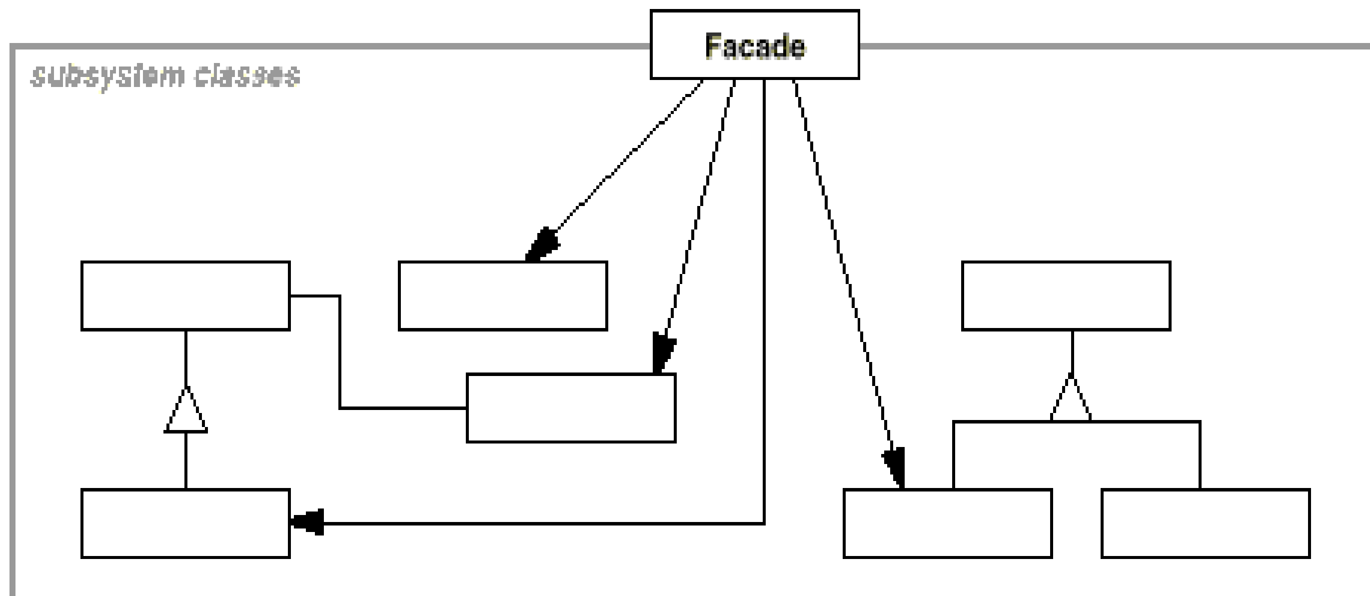
Motivation



Applicability

- you want to provide a simple interface to a complex subsystem
- there are many dependencies between clients and the implementation classes of an abstraction.
- you want to layer your subsystems.

Structure



Consequences

Benefits of Façade:

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.

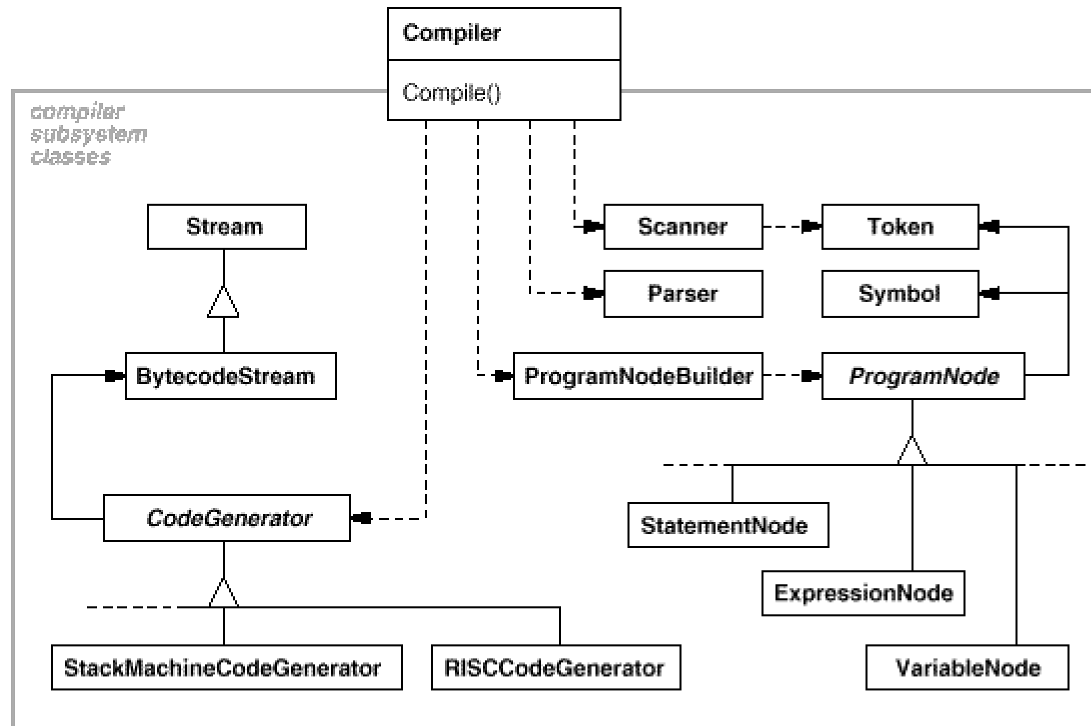
3. It doesn't prevent applications from using subsystem classes if they need to. Thus you can choose between ease of use and generality.

Implementation

1. Reducing client-subsystem coupling.

2. Public versus private subsystem classes.

Examples



The Façade defines a unified, higher level interface to a subsystem that makes it easier to use. Clients encounter a Façade when compiling the source code. The Compiler service acts as a Façade, providing an interface to the complex orchestration of different objects working together to achieve a compilation task.

Sample Code

```
public class Cube {

public int doTheCube(int x) {
return x * x * x;
}

}
```

```
public class DoubleCube {

public int doDoubleCube(Cube cube, int x) {
return 2 * cube.doTheCube(x);
}
}
```

```
public class CubeAndDoubleCube {

public int doCubeAndDoubleCube(Cube cube,
DoubleCube doubleCube, int x) {
return cube.doTheCube(x) *
doubleCube.doDoubleCube(cube, x);
}
}
```

```
public class Façade {

public int cubeX(int x) {
Cube cube = new Cube();
return cube.doTheCube(x);
}

public int cubeXTimes2(int x) {
Cube cube = new Cube();
DoubleCube doubleCube= new DoubleCube();
return doubleCube.doDoubleCube(cube, x);
}

public int cubeAndDoubleCubeOfx(int x) {
Cube cube = new Cube();
DoubleCube doubleCube= new DoubleCube();
CubeAndDoubleCube cubeAndDoubleCube= new CubeAndDoubleCube();
return cubeAndDoubleCube.doCubeAndDoubleCube(cube, doubleCube, x);
}

}
```

```
public class FaçadeDemo {

public static void main(String[] args) {

Façade Façade = new Façade();

int x = 3;
System.out.println("Cube of " + x + " is: " + Façade.cubeX(x));
System.out.println("Cube of " + x + " multiply with 2 is: " +
Façade.cubeXTimes2(x));
System.out.println("Cube X Double cube of " + x + " is: " +
Façade.cubeAndDoubleCubeOfx(x));

}

}
```

Related Patterns

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

- Mediator is similar to Façade 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.
  In contrast, a Façade 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 Façade object is required. Thus Façade objects are often Singletons.