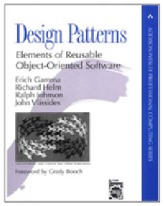


Design Pattern Definitions from the GoF Book

The Adapter Pattern

Converts the interface of a class into another interface the clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Creational Patterns

- **The Factory Method Pattern**
  Defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- **The Abstract Factory Pattern**
  Provides an interface for creating families of related or dependent objects without specifying their concrete classes.
- **The Singleton Pattern**
  Ensures a class has only one instance, and provides a global point of access to it.
- **The Builder Pattern**
- **The Prototype Pattern**

Structural Patterns

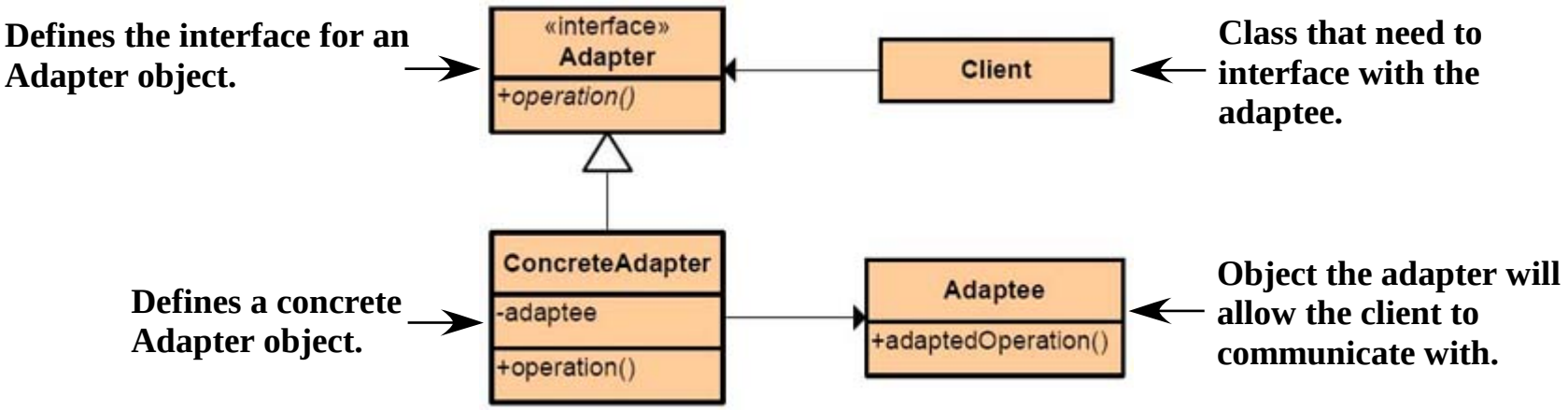
- **The Decorator Pattern**
  Attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
- **The Adapter Pattern**
- **The Facade Pattern**
- **The Composite Pattern**
- **The Proxy Pattern**
- **The Bridge Pattern**
- **The Flyweight Pattern**

Behavioral Patterns

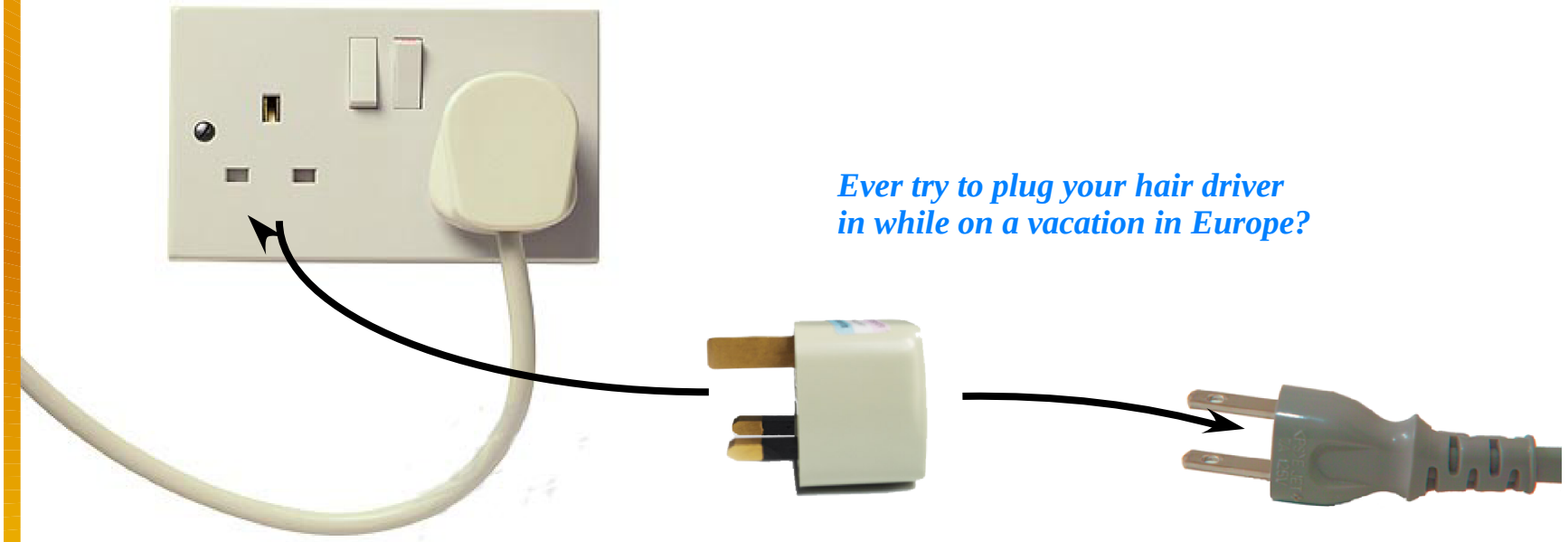
- **The Strategy Pattern**
  Defines a family of algorithms, encapsulates each one, and makes them interchangeable.
- **The Observer Pattern**
  Defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatially.
- **The Command Pattern**
  Encapsulates a request as an object, thereby letting you parameterize other objects with different requests, queue or log requests, and support undoable operations.
- **The Template Method Pattern**
- **The Iterator Pattern**
- **The State Pattern**
- **The Chain of Responsibility Pattern**
- **The Interpreter Pattern**
- **The Mediator Pattern**
- **The Memento Pattern**
- **The Visitor Pattern**

Design Patterns: The Adapter
Quick Overview

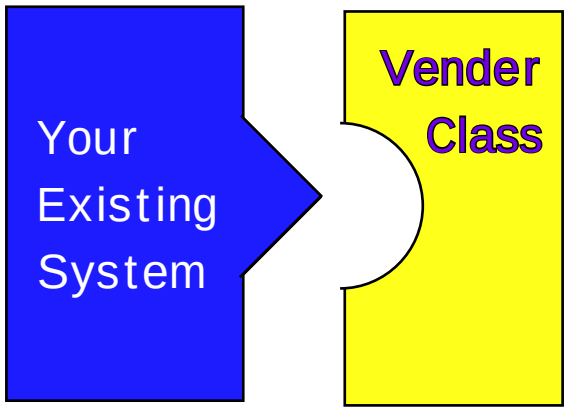
Converts the interface of a class into another interface the clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.



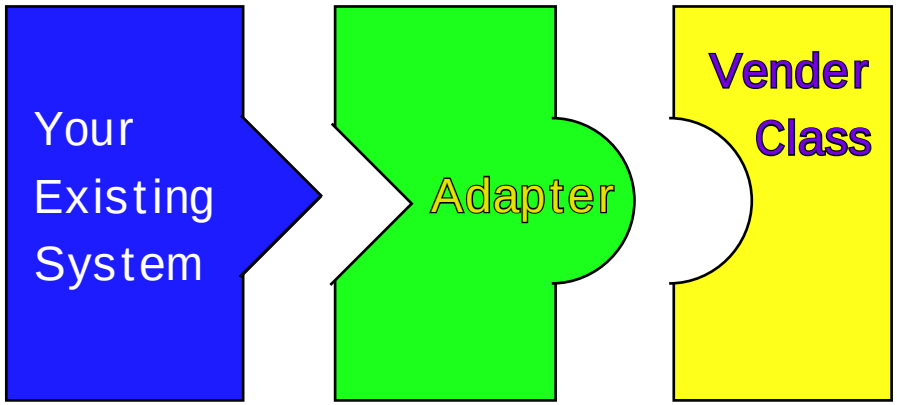
Design Patterns: The Adapter



Ever try to plug your hair driver in while on a vacation in Europe?



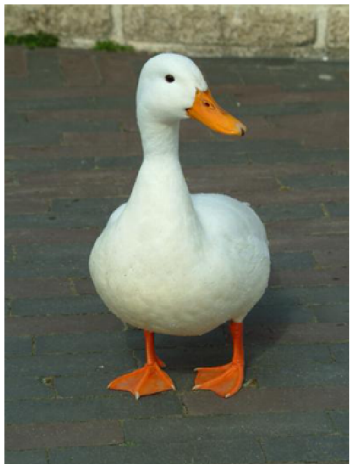
Without Adapter



With Adapter

Design Patterns: The Adapter

Remember the Duck interface?



Meet the newest fowl on the block, the Turkey interface.

Duck

void quack()
void fly()

Turkey

void gobble()
void fly()

Turkey's don't quack, they gobble.

Turkey's can fly, but only short distances.

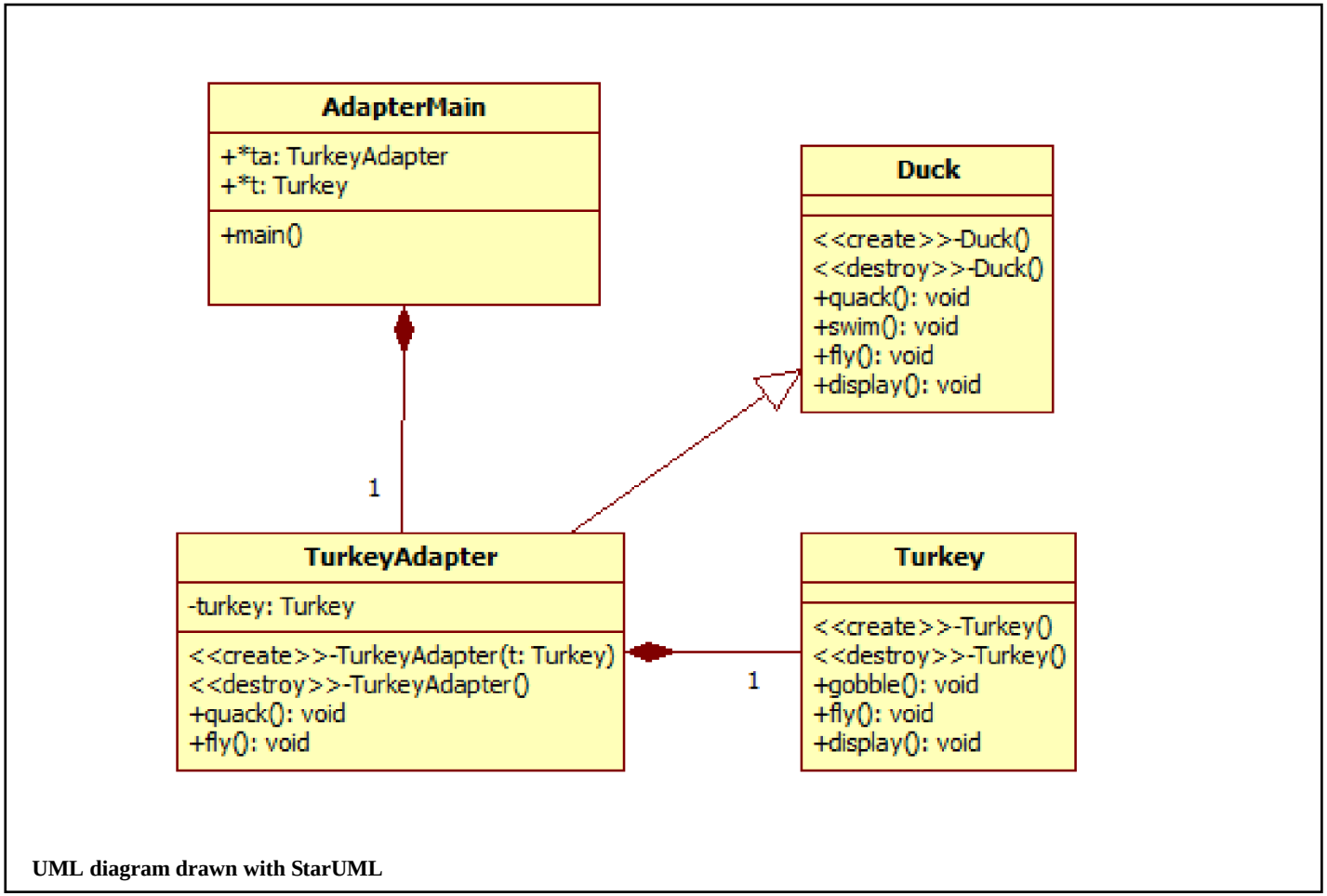
TurkeyAdapter

```
class TurkeyAdapter:Duck
{
   private:
        Turkey *turkey;
    public:
        TurkeyAdapter(
                Turkey *turkey)
        {
           this->turkey = turkey;
        }
        void quack()
        {
            turkey->gobble();
         }
         void fly()
         {
             for(int i=0; i<5; i++)
                 turkey->fly();
         }
}
```

Turkey's can fly, but only in short spurts. They can't fly long distances like ducks.

Design Patterns: The Adapter

Code Sample



UML diagram drawn with StarUML

AdapterMain

Creates an instance of TurkeyAdapter and connects it to a Turkey

Makes Duck interface calls which the TurkeyAdapter translates for the Turkey.

Let's look at the code and run the demonstration.