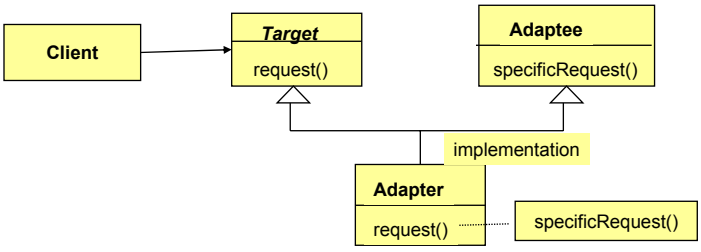
Adapter(GoF)

- Intent
  - Convert the interface of a class into another interface *clients expect*.
  - Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- Applicability - use when
  - you want to use an existing class and its interface does not match the one you need
  - you want to create a reusable class that cooperates with unrelated or unforeseen classes, that is classes that don't necessarily have compatible interfaces
  - you need to use several existing subclasses, but it's impractical to adapt their interface by subclassing everyone. An object adapter can adapt the interface of its parent class
- Also Known As: Wrapper
- Motivation
  - Sometimes a toolkit or class library can not be used because its interface is incompatible with the interface required by an application
  - We can not change the library interface, since we may not have its source code
  - Even if we did have the source code, we probably should not change the library for each domain-specific application
  - provide a new interface to existing legacy components (Interface engineering, reengineering).

Structure: Two adapter patterns (two forms)

Class adapter: Uses multiple inheritance to adapt one interface to another

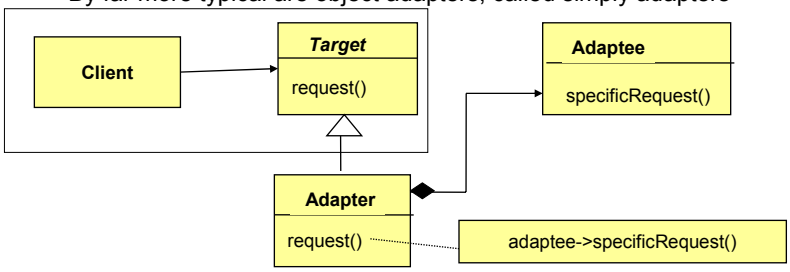
Object adapter: Uses single inheritance and delegation



Structure of the class adapter pattern

… structure cont.

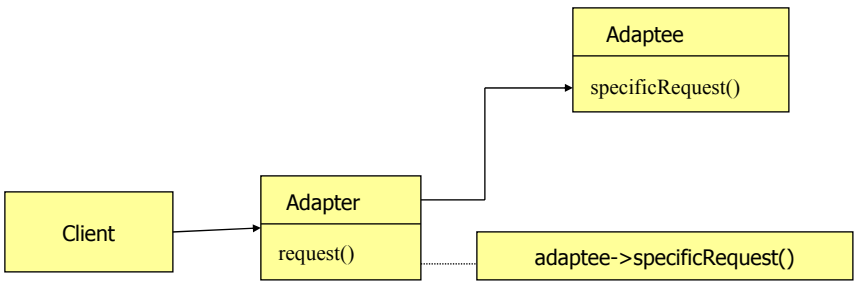
By far more typical are object adapters, called simply adapters



Structure of the object adapter pattern

- Delegation is used to bind an **Adapter** and an **Adaptee**
- Interface inheritance is use to specify the interface of the **Adapter** class.
- ***Target*** and **Adaptee** (usually called legacy system) pre-exist the **Adapter.**
- **Target** may be realized as an interface in Java.

Structure of a simple wrapper
(for comparision)



Stack interface and an implementation (c++)

```
class Stack {  // interface for clients of Stack
protected:
   Stack() {}
public:
   virtual void push(int x)   {}
   virtual void pop()         {}
   virtual int top()    const { return 0; }
   virtual bool empty() const { return true; }
};

class FixedArray {  // implementation, no client wants to see this
public:
   FixedArray() : index(EMPTY)        {}
   void insert(int x)  { items[++index] = x; }
   void removeLast()   { --index; }
   int getLast() const { return items[index]; }
   bool empty()  const { return index == EMPTY; }
private:
   int index;
   enum { MAX = 10 }; // the "enum hack"!
   enum { EMPTY = -1 };
   int items[MAX];
};
```

The real Stack

A class adapter: using multiple inheritance, inherited interface is Implemented using inherited implementation

```
class MyStack : public Stack, private FixedArray {
public:
   virtual void push(int x)   { insert(x); }
   virtual void pop()         { removeLast(); }
   virtual int top()    const { return getLast(); }
   virtual bool empty() const {
       return FixedArray::empty();
   }
};
```

Using the Stack

```
void empty_and_print(Stack * stk) {
   while ( !stk->empty() ) {
      cout << stk->top() << endl;
      stk->pop();
   }
}
int main() {
   Stack * stk = new MyStack;
   stk->push(99);
   stk->push(3);
   stk->push(17);
   empty_and_print(stk);
   if ( stk->empty() ) cout << "yes\n";
   return 0;
}
```

The Java Interface and implementation

```
public interface Stack {
   public void push(int x);
   public void pop();
   public int top();
   boolean empty();
}

public class FixedArray {
   public FixedArray() {
      index = -1;
      items = new int[10];
   }
   public void insert(int x)  { items[++index] = x; }
   public void removeLast()   { --index; }
   public boolean empty()     { return index == -1; }
   public int top()           { return items[index]; }

   private int index;
   private int [] items;
}
```

The real Java Stack

An object adapter: inherited interface is
Implemented using aggregate implementation object

```
public class MyStack implements Stack {
   public MyStack() { array = new FixedArray(); }
   public void push(int x) { array.insert(x); }
   public void pop()       { array.removeLast(); }
   public int top()        { return array.top(); }
   public boolean empty()  { return array.empty(); }

   private FixedArray array;
}
```

Consequences: tradeoffs between class and object adapter

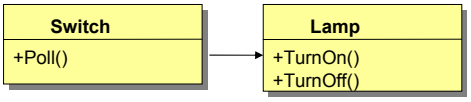
- A class adapter
  - adapts Adaptee to target by committing to a concrete Adapter class. Thus, a class adapter won't work when we want to adapt a classand all it's subclasses
  - lets Adapter override some of Adaptee's behavior, since Adapter is a subclass of Adaptee
  - Introduces only one object, and no pointer indirection to get to Adaptee
- An object adapter
  - let's a single Adapter work with many Adaptees: the adaptee and all of its subclasses.
  - The Adapter can add functionality to all Adaptees at once
  - Makes it harder to override Adaptee behavior; requires subclassing Adaptee and making Adapter refer to the subclass rather than the Adaptee itself.

Notes on Adapter

- The adapter pattern is most commonly used to allow for polymorphism
  - The adapted class should be handled through an abstract base class (i.e. as a *shape*, not as a *circle* or *square*). In this case the abstract base class is the *target* interface.
  - The adapter is very often used to allow for polymorphism required by other design patterns
- In case the object being adapted does not do all the needed things, the adapter class can
  - Adapt the behavior that is implemented in the adapted object
  - Implement the rest of the behavior
- During analysis or conceptual design, adapter frees from worrying about interfaces
  - As long as you have a class that does what you need, at least conceptually, you know that you can always use the adapter
  - This will become important also as you start designing with patterns: many patterns require that some classes derive from a same class or interface, a requirement you know you can achieve with an adapter

Abstract Server, a related pattern (not from GoF)

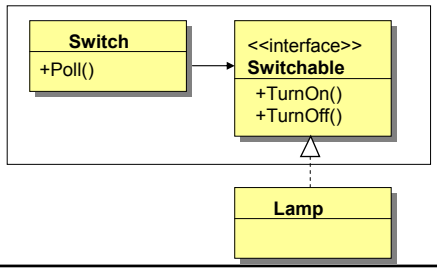
- The problem of a switch controlling a lamp can be solved in many ways
- The simple design below violates both the DIP and OCP
  - DIP: Switch depend on a concrete class
  - OCP: we can not easily extend Switch to control other devices since the design forces us to drag a Lamp along wherever we need a Switch



- Inheriting Switch to control something else, e.g. a fan, does not fix the problem because the dependency to lamp is also inherited to say FanSwitch.

Abstract Server, a related pattern (not from GoF)

- The problem is solved by applying one of the simplest design patterns, 'Abstract Server' that introduces an interface between the two classes
- This immediately resolves the violations of DIP and OCP
- An interesting point that Abstract Server makes clear is the owner of the interface
  - Traditionally in OO design it was thought that the physical tight inheritance dependency between Switchable and Lamp is more important, and thus inheritance hierarchies should be packaged together
  - However it is often the case that the logical dependence between the client (Switch) and the interface the client controls (Switchable) is more relevant with regard of reusability and flexibility of changes, thus the client and the interface it depends on should be packaged together



Abstract Server leads to Adapter

- The problem in previous 'Abstract Server' solution is a potential violation on SRP
  - Lamp and Switchable are bound together, but these two may have different reasons for changing.
- Other problems
  - What if we can not add the required inheritance to Lamp?
  - What if the Lamp is a third party component and we do not even have the source code?
  - What if there is another class that Switch should be able to control, but we can not make it inherit from Switchable?
- Solution: extend this to Adapter