Adapter pattern

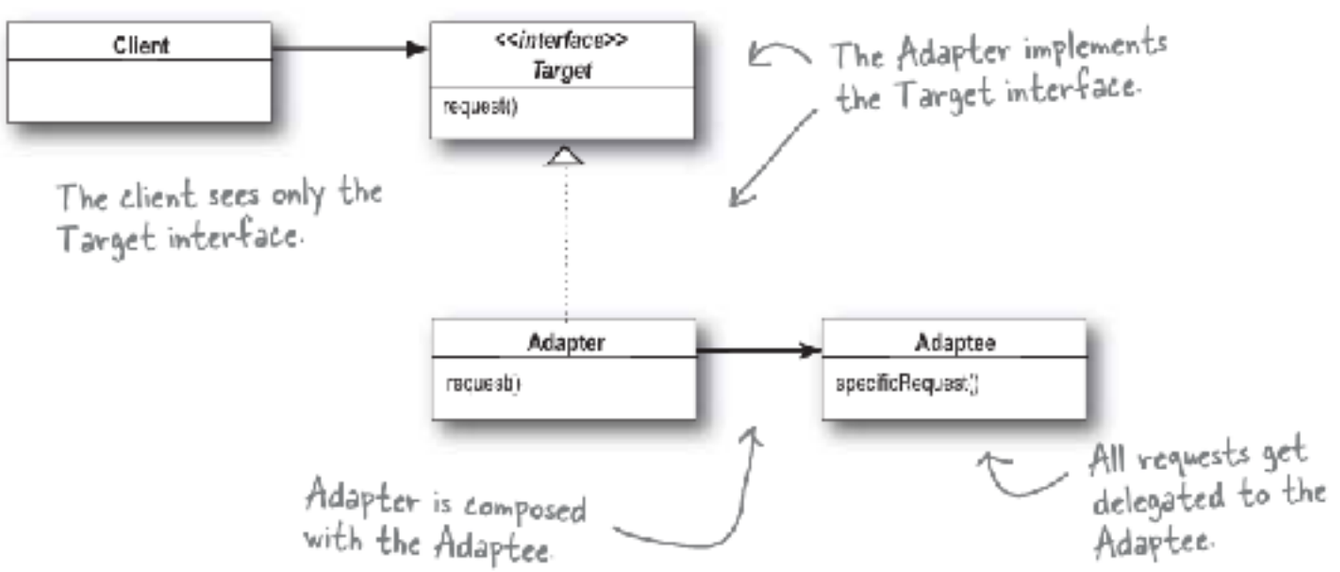
발표자:우연서

Adapter : 의도

재사용하려는 클래스가 제공하는 인터페이스와, 클라이언트의 인터페이스가 다를 때. 인터페이스를 일치시키는 것

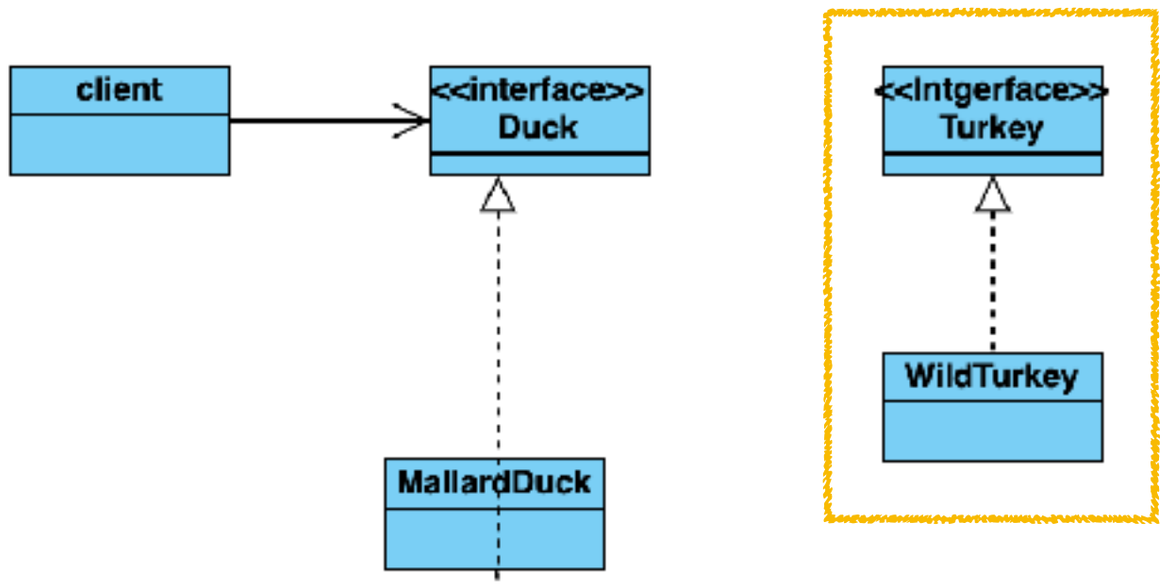
문제상황!!

- **기존 클래스를 사용해야 하나 인터페이스가 수정**되어야 하는 경우
- **이미 만들어진 것을 재사용하**고자 하나 이 **재사용 가능한 라이브러리를 수정할 수 없는 경우**
- **기존 클래스를 사용해야 하나 인터페이스가 수정**되어야 하는 경우
- **이미 만들어진 것을 재사용하**고자 하나 이 **재사용 가능한 라이브러리를 수정할 수 없는 경우**
- 이미 존재하는 여러 개의 서브클래스를 사용해야 할 때, 서브클래스 상속을 통해 인터페이스를 모두 변경하는 것이 현실적으로 어렵다. —> 객체 어댑터 방식으로 부모 클래스의 인터페이스를 변형하는 것이 바람직



Adapter : 우리가 다아는 그 예제

재사용하려는 클래스가 제공하는 인터페이스와, 클라이언트의 인터페이스가 다를 때. 인터페이스를 일치시키는 것



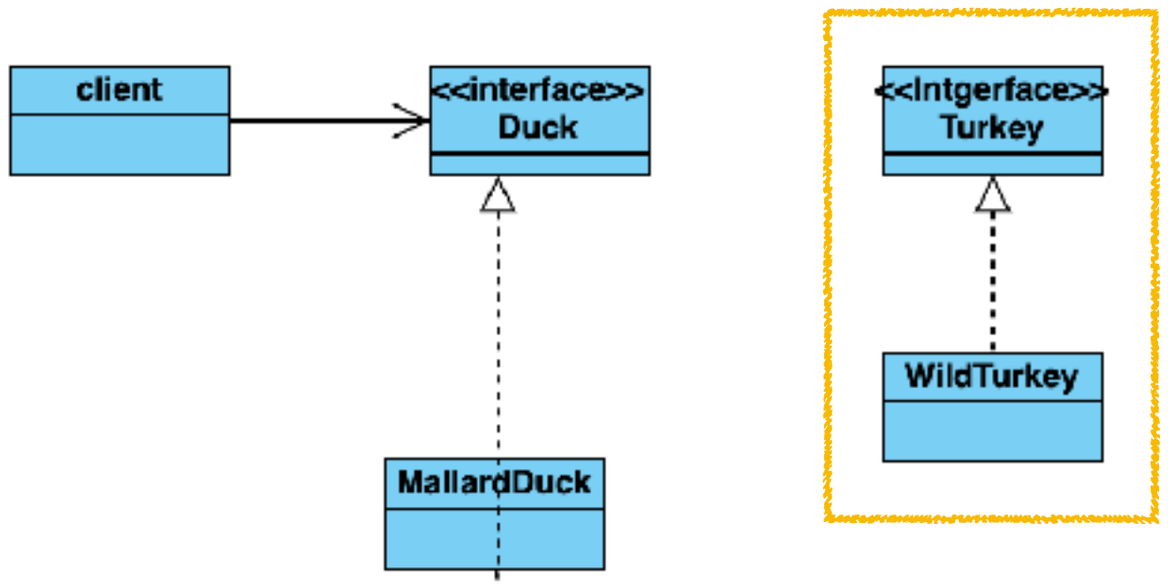
가정

- client가 오리로봇명령어로 칠면조로봇을 움직이고 싶다.
- 오리 인터페이스는 변경이 불가능하다
- MallardDuck빼고도 오리 종류가 이백개라고 가정.

- **기존 클래스를 사용해야 하나 인터페이스가 수정**되어야 하는 경우 -> 칠면조로봇은 다르게 운다
- **이미 만들어진 것을 재사용하**고자 하나 이 **재사용 가능한 라이브러리를 수정할 수 없는 경우 ->오리명령어를 사용하지만 오리 인터페이스는 고칠 수 없다**
- **기존 클래스를 사용해야 하나 인터페이스가 수정**되어야 하는 경우 -> 오리명령어를 사용해야함.
- 이미 존재하는 여러 개의 서브클래스를 사용해야 할 때, 서브클래스 상속을 통해 인터페이스를 모두 변경하는 것이 현실적으로 어렵다. —> 객체 어댑터 방식으로 부모 클래스의 인터페이스를 변형하는 것이 바람직

Adapter : 우리가 다아는 그 예제

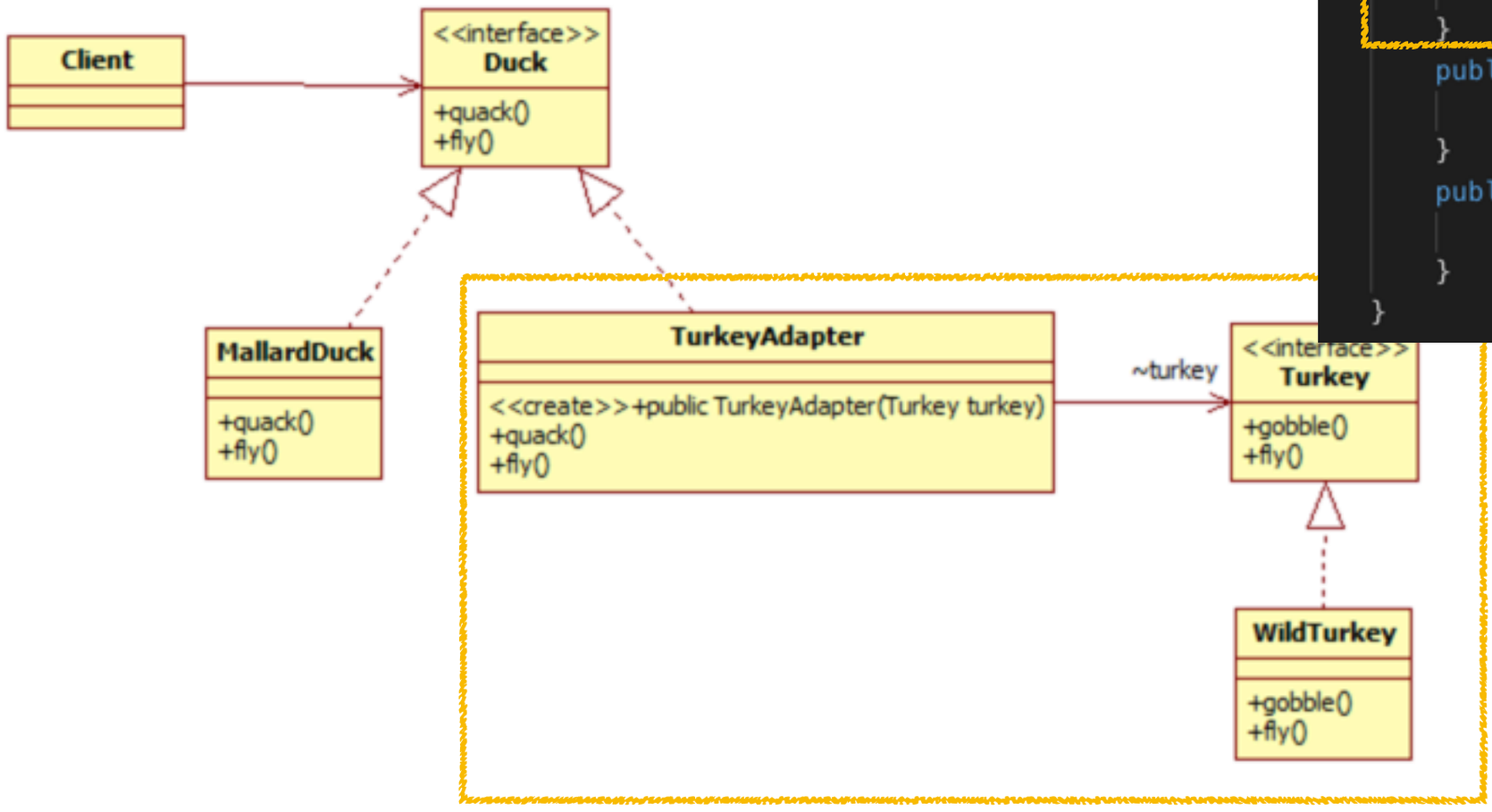
재사용하려는 클래스가 제공하는 인터페이스와, 클라이언트의 인터페이스가 다를 때. 인터페이스를 일치시키는 것



적용

- **기존 클래스를 사용해야 하나 인터페이스가 수정**되어야 하는 경우 -> 칠면조로봇은 다르게 운다
- **이미 만들어진 것을 재사용하**고자 하나 이 **재사용 가능한 라이브러리를 수정할 수 없는 경우 ->오리명령어를 사용하지만 오리 인터페이스는 고칠 수 없다**
- **기존 클래스를 사용해야 하나 인터페이스가 수정**되어야 하는 경우 -> 오리명령어를 사용해야함.
- 이미 존재하는 여러 개의 서브클래스를 사용해야 할 때, 서브클래스 상속을 통해 인터페이스를 모두 변경하는 것이 현실적으로 어렵다. —> 객체 어댑터 방식으로 부모 클래스의 인터페이스를 변형하는 것이 바람직

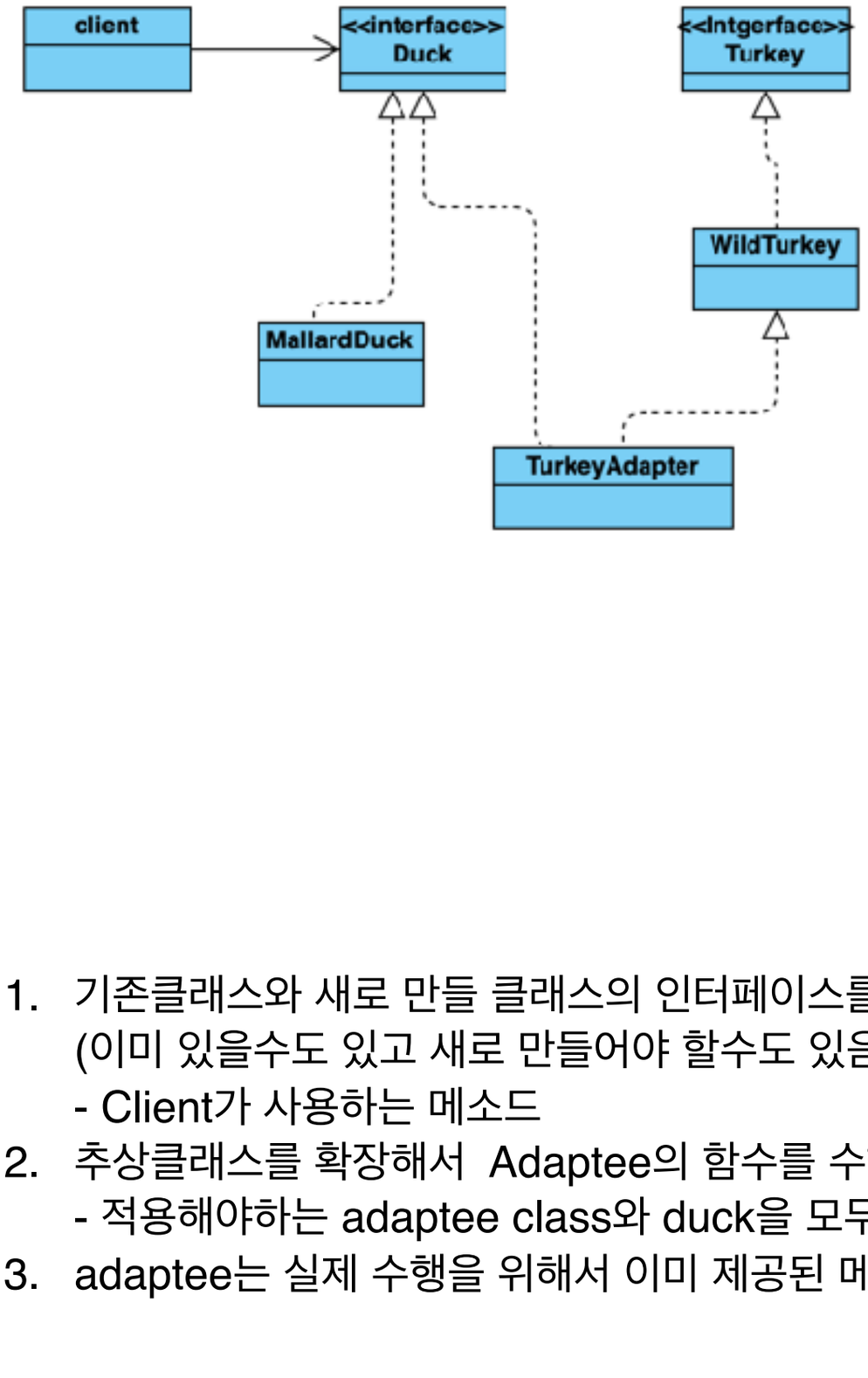
Object Adapter : 우리가 다아는 그 예제




```
class TurkeyAdapter implements Duck {
    public turkey:Turkey;
    constructor(turkey:Turkey) {
        this.turkey = turkey;
    }
    public  quack():void{
        this.turkey.gobble();
    }
    public  fly():void{
        this.turkey.fly();
    }
}
```

1. 기존클래스와 새로 만들 클래스의 인터페이스를 동일하게 맞추는 추상클래스를 만든다.
   (이미 있을수도 있고 새로 만들어야 할수도 있음)
   - Client가 사용하는 메소드
2. 추상클래스를 확장해서 Adaptee의 함수를 수행하는 Adapter를 만든다.
   - 여기서 위임의 방식인지, 상속의 방법인지가 갈린다. **일단 위임!**
3. adaptee는 실제 수행을 위해서 이미 제공된 메소드를 갖는다

Class Adapter




```
class TurkeyAdapter extends WildTurkey implements Duck {
    constructor() {
        super();
    }
    public  quack():void{
        this.gobble();
    }
    public  fly():void{
        this.fly();
    }
}
```

1. 기존클래스와 새로 만들 클래스의 인터페이스를 동일하게 맞추는 추상클래스를 만든다.
   (이미 있을수도 있고 새로 만들어야 할수도 있음)
   - Client가 사용하는 메소드
2. 추상클래스를 확장해서 Adaptee의 함수를 수행하는 Adapter를 만든다.
   - 적용해야하는 adaptee class와 duck을 모두 구현한다.
3. adaptee는 실제 수행을 위해서 이미 제공된 메소드를 갖는다

Object Adapter	Class Adapter
- 모든 서브클래스에 적용됨 - 어뎁터 하나로 사용 불가능	- 오버라이딩으로 기능 추가 가능 전체를 재정의할 필요 없음 - 어뎁터를 하나로 사용 가능 - 특정클래스에만 적용됨

언제 Object Adapter를 사용할까

1. 상속을 받을 수 없게 설정된 경우 - java final 설정된 경우
2. **client가 interface가 아닌 abstract class와 연결된 경우 다중상속을 구현할 수 없으므로 object Adapter를 사용한다.**

   - <<interface>> Duck이 아니라 <<abstract>>Duck인 경우
3. 여러 adaptee를 adapter 에 적용하고, 그 집합체에 대한 작업이 필요할 때.

2.

```
abstract class duck {
    abstract void display();
}

class turkeyAdapter extends duck {
    Turkey turkey;
    void fly() {
        turkey.fly();
    }
}
```

https://stackoverflow.com/questions/9978477/difference-between-object-adapter-pattern-and-class-adapter-pattern

언제 Object Adapter를 사용할까

1. 상속을 받을 수 없게 설정된 경우 - java final 설정된 경우
2. client가 interface가 아닌 abstract class와 연결된 경우 다중상속을 구현할 수 없으므로 object Adapter를 사용한다.

   - <<interface>> Duck이 아니라 <<abstract>>Duck인 경우
3. **여러 adaptee를 adapter 에 적용하고, 그 집합체에 대한 작업이 필요할 때.**

3.

```
class MyTableModel extends AbstractTableModel {

  MyDomainObject[] existingDomainObjects[];

  public int getColumnCount() {
      return 4;
  }

  public int getRowCount() {
      return existingDomainObjects.length();
  }

  public MyDomainObject getValueAt(int i) {
      return existingDomainObjects[i];
  }
}
```

책 예시 : Adapting an Enumeration to an Iterator

다른거 못찾겠어요 ㅠㅠ

그래서1 : 지금 어떻게 쓰이는데

```java
//Adapter
public class EnumerationIterator implements Iterator{
   Enumeration enumeration;

   public EnumerationIterator(Enumeration enumeration) {
     this.enumeration= enumeration;
   }

   @Override
   public boolean hasNext(){
     return enumeration.hasMoreElements();
   }

   @Override
   public Object next() {
     return enumeration.nextElement();
   }

   @Override
   public void remove() {
     // 예외 : UnsupportedOperationException
     throw new UnsupportedOperationException();
   }
}
```

생각1 : 새로운 스펙의 변경이 적용되어야 할 때. Like 서드파티 라이브러리
생각2 : A->B로 포맷의 변경이 필요한 경우 ??(같은소리..)

얻어가야 할 아이디어
- 감싸서 전환을 지원할 수 있다.
- 의문 : class 자체를 넘기는 게 옳은 일인가.

// Enumeration는 remove() 가 없다.
// 런타임 예외를 던진다

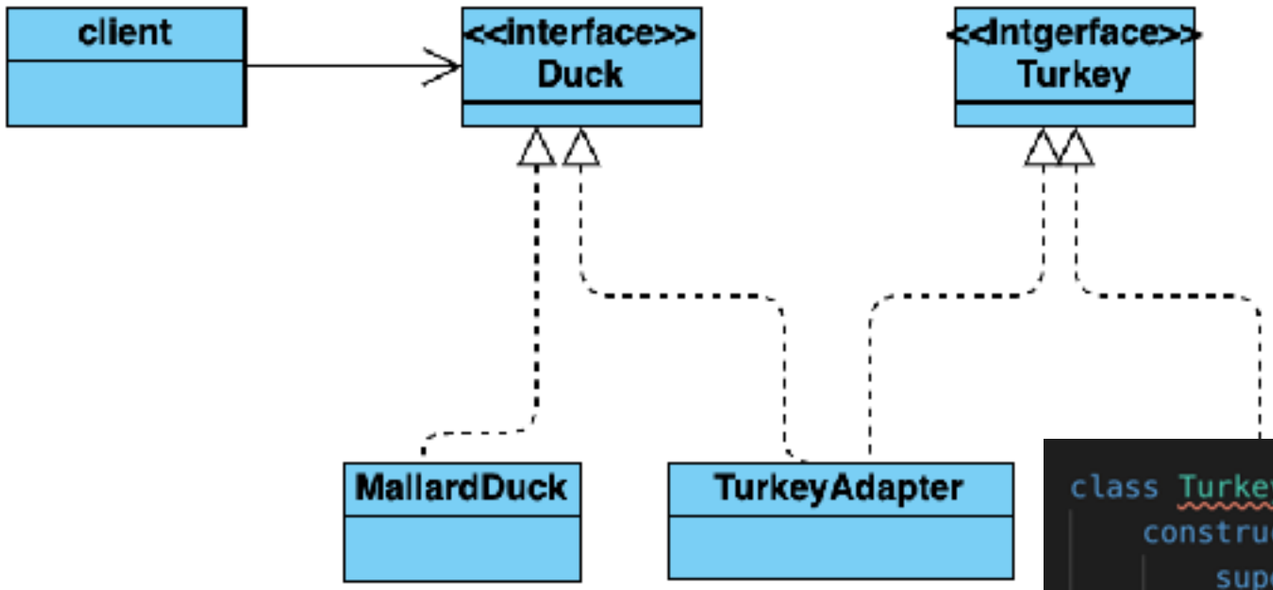
그래서2 : 뭐가 다른데 : FACADE

결국 구분하자면

FACADE	공통적인 작업에 대해 간편한 메소드들을 제공
ADAPTER	인터페이스를 필요한 다른 인터페이스로 변환
DECORATOR	인터페이스의 수정 없이 기능을 추가

בב

Adapter : 우리가 다아는 그 예제




```
class TurkeyAdapter extends WildTurkey implements Duck {
    constructor() {
        super();
    }
    public  quack():void{
        this.gobble();
    }
    public  fly():void{
        this.fly();
    }
}
```

1. 기존클래스와 새로 만들 클래스의 인터페이스를 동일하게 맞추는 추상클래스를 만든다.
   (이미 있을수도 있고 새로 만들어야 할수도 있음)
   - Client가 사용하는 메소드
2. 추상클래스를 확장해서 Adaptee의 함수를 수행하는 Adapter를 만든다.
   - 여기서 위임의 방식인지, 상속의 방법인지가 갈린다. **일단 위임!**
3. adaptee는 실제 수행하는 이미 제공된 메소드