

컴파운드 패턴

이관우

kwlee@hansung.ac.kr

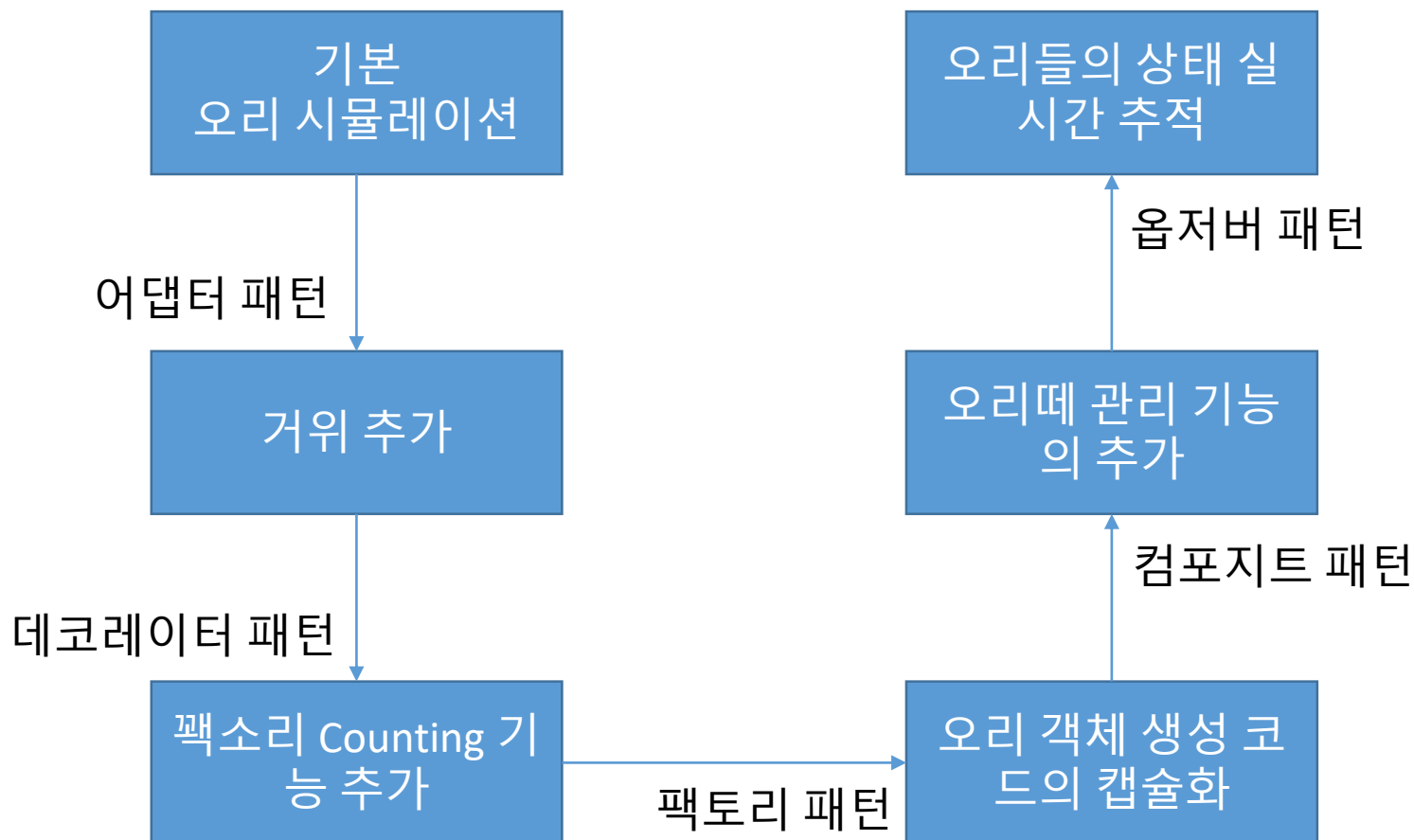
학습 목표

- 패턴 섞어 쓰기 개요
 - 오리 시뮬레이터
- 다양한 패턴 적용
 - 어댑터 패턴 적용
 - 데코레이터 패턴 적용
 - 컴포지트 패턴 적용
 - 팩토리 패턴 적용
 - 옵저버 패턴 적용

컴파운드 패턴

- 컴파운드 패턴이란
 - 반복적으로 생길 수 있는 일반적인 문제를 해결하기 위한 용도로 두 개 이상의 패턴을 결합해서 사용하는 것
 - 단순한 패턴의 결합이 모두 컴파운드 패턴은 아님
- 컴파운드 패턴의 예
 - 모델-뷰-컨트롤러
 - 옵저버+컴포지트+스트래티지 패턴
 - “사용자 인터페이스로부터 비즈니스 로직을 분리하여 애플리케이션의 시각적 요소나 그 이면에서 실행되는 비즈니스 로직을 서로 영향 없이 쉽게 고칠 수 있는 애플리케이션을 만들 수 있다”
[<https://ko.wikipedia.org/wiki/모델-뷰-컨트롤러>]
 - 다양한 아키텍처 패턴
 - Buschmann F.; Meunier R.; Rohnert H.; Sommerlad P.; Stal M. (1996). *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons.

오리 시뮬레이션 예제



기본 오리 시뮬레이션

```
public interface Quackable {  
    public void quack();  
}
```

```
public class MallardDuck implements Quackable {  
    public void quack() {  
        System.out.println("Quack");  
    }  
}
```

```
public class RedheadDuck implements Quackable {  
    public void quack() {  
        System.out.println("Quack");  
    }  
}
```

```
public class DuckCall implements Quackable {  
    public void quack() {  
        System.out.println("Kwak");  
    }  
}
```

```
public class RubberDuck implements Quackable {  
    public void quack() {  
        System.out.println("Squeak");  
    }  
}
```

기본 오리 시뮬레이션

```
public class DuckSimulator {
    public static void main(String[] args) {
        DuckSimulator simulator = new DuckSimulator();
        simulator.simulate();
    }

    void simulate() {
        // mallardDuck, redheadDuck, duckCall, rubberDuck 생성

        System.out.println("\nDuck Simulator");
        simulate(mallardDuck);
        simulate(redheadDuck);
        simulate(duckCall);
        simulate(rubberDuck);
    }

    void simulate(Quackable duck) {
        duck.quack();
    }
}
```

실행결과

```
Duck Simulator
Quack
Quack
Kwak
Squeak
```

거위 추가

- 시뮬레이션에 거위도 추가해 보죠

```
public class Goose {  
    public void honk() {  
        System.out.println("Honk");  
    }  
}
```



거위 추가

- GooseAdapter

```
public class GooseAdapter implements Quackable {
    Goose goose;

    public GooseAdapter(Goose goose) {
        this.goose = goose;
    }

    public void quack() {
        goose.honk();
    }

    public String toString() {
        return "Goose pretending to be a Duck";
    }
}
```


거위 추가

```
public class DuckSimulator {  
    ...  
  
    void simulate() {  
        // mallardDuck, redheadDuck, duckCall, rubberDuck 생성  
        Quackable gooseDuck = new GooseAdapter(new Goose());  
  
        System.out.println("\nDuck Simulator: With Goose Adapter");  
  
        simulate(mallardDuck);  
        simulate(redheadDuck);  
        simulate(duckCall);  
        simulate(rubberDuck);  
        simulate(gooseDuck);  
    }  
    ...  
}
```

실행결과

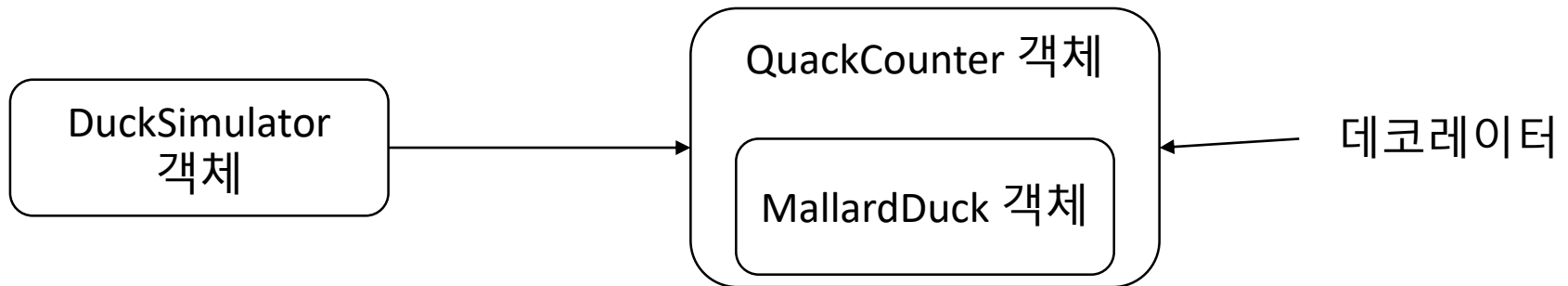
```
Duck Simulator: With Goose Adapter  
Quack  
Quack  
Kwak  
Squeak  
Honk
```

팩소리 Counting 기능 추가

- 오리 클래스는 그대로 두면서 오리가 팩소리를 낸 회수를 세려면 어떻게 해야 할까요?



새로운 기능의 추가



팩소리 Counting 기능 추가

```
public class QuackCounter implements Quackable {  
    Quackable duck;  
    static int numberOfQuacks;  
  
    public QuackCounter (Quackable duck) {  
        this.duck = duck;  
    }  
  
    public void quack() {  
        duck.quack();  
        numberOfQuacks++;  
    }  
  
    public static int getQuacks() {  
        return numberOfQuacks;  
    }  
    ...  
}
```

데코레이터는 감싸는
객체와 동일한 타입이
어야 함

팩소리 Counting 기능 추가

```
public class DuckSimulator {  
    ...  
  
    void simulate() {  
        Quackable mallardDuck = new QuackCounter(new MallardDuck());  
        Quackable redheadDuck = new QuackCounter(new RedheadDuck());  
        Quackable duckCall = new QuackCounter(new DuckCall());  
        Quackable rubberDuck = new QuackCounter(new RubberDuck());  
        Quackable gooseDuck = new GooseAdapter(new Goose());  
  
        System.out.println("\nDuck Simulator: With Decorator");  
  
        simulate(mallardDuck);  
        simulate(redheadDuck);  
        ...  
  
        System.out.println("The ducks quacked " +  
                           QuackCounter.getQuacks() + " times");  
    }  
}
```

오리 객체 생성 코드의 캡슐화

- 오리 객체 생성의 이슈
 - QuackCounter 데코레이터를 쓸 때, 객체들을 제대로 감싸지 않으면 원하는 행동이 제대로 추가되지 않음
 - QuackCounter 데코레이터가 필요없는 경우



팩토리를 사용하여 객체 생성 코드의 캡슐화

서로 다른 오리 **제품 군** (Counting 기능이 없는 오리 객체들, Counting 기능이 있는 오리 객체들)을 생성시키기 위해 **추상 팩토리 패턴**

오리 객체 생성 코드의 캡슐화


```
public class DuckFactory extends AbstractDuckFactory {  
  
    public Quackable createMallardDuck() {  
        return new MallardDuck();  
    }  
  
    public Quackable createRedheadDuck() {  
        return new RedheadDuck();  
    }  
  
    public Quackable createDuckCall() {  
        return new DuckCall();  
    }  
  
    public Quackable createRubberDuck() {  
        return new RubberDuck();  
    }  
}
```

오리 객체 생성 코드의 캡슐화

```
public class CountingDuckFactory extends AbstractDuckFactory {  
  
    public Quackable createMallardDuck() {  
        return new QuackCounter(new MallardDuck());  
    }  
  
    public Quackable createRedheadDuck() {  
        return new QuackCounter(new RedheadDuck());  
    }  
  
    public Quackable createDuckCall() {  
        return new QuackCounter(new DuckCall());  
    }  
  
    public Quackable createRubberDuck() {  
        return new QuackCounter(new RubberDuck());  
    }  
}
```

오리 객체 생성 코드의 캡슐화

```
public class DuckSimulator {  
    public static void main(String[] args) {  
        DuckSimulator simulator = new DuckSimulator();  
        AbstractDuckFactory duckFactory = new CountingDuckFactory();  
  
        simulator.simulate(duckFactory);  
    }  
  
    void simulate(AbstractDuckFactory duckFactory) {  
        Quackable mallardDuck = duckFactory.createMallardDuck();  
        Quackable redheadDuck = duckFactory.createRedheadDuck();  
        Quackable duckCall = duckFactory.createDuckCall();  
        Quackable rubberDuck = duckFactory.createRubberDuck();  
        Quackable gooseDuck = new GooseAdapter(new Goose());  
  
        System.out.println("\nDuck Simulator: With Abstract Factory");  
  
        simulate(mallardDuck);  
        ...  
    }  
}
```



new DuckFactory() 로 교체

오리떼 관리 기능의 추가

- 오리를 하나씩 일일이 관리하는 대신에, 모든 오리를 일괄적으로 관리하던가, 종별로 관리할 수 있으려면...

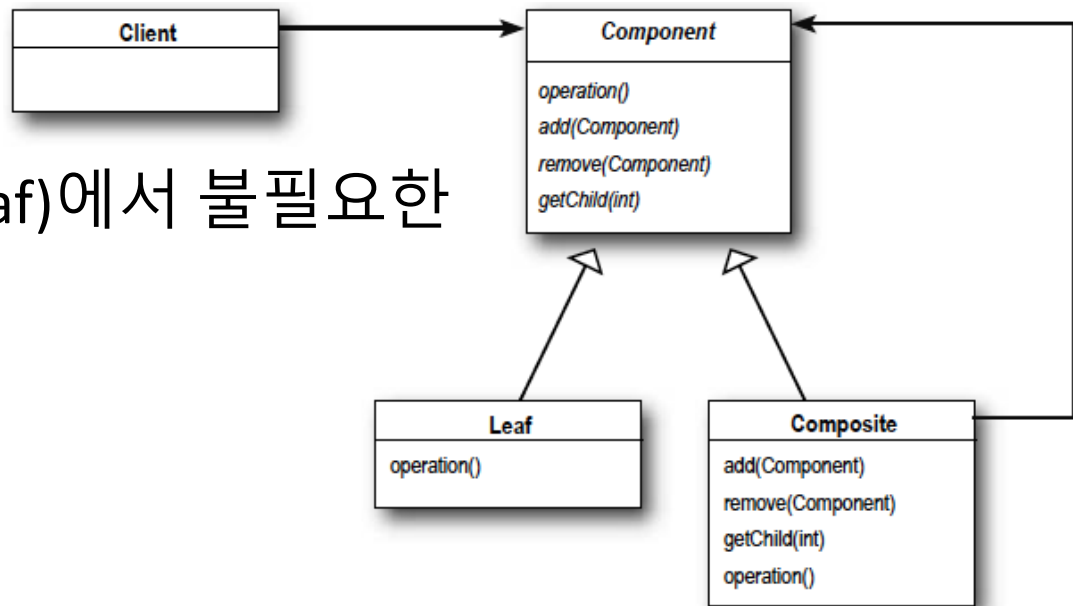


컴포지트 패턴을 적용하여,
개별 오리나 **오리 컬렉션**(예, 전체 오리 집합, 종별 오리 집합)을 동일한 방법으로 관리

컴포지트 패턴 (revisited)

- 투명성
 - 클라이언트는 개별 객체 (Leaf)나 복합객체 (Composite)를 동일한 방법으로 다룸

- 안전성
 - 개별 객체 (Leaf)에서 불필요한 요소 포함



오리떼 관리 기능의 추가

```
public class Flock implements Quackable {
    ArrayList<Quackable> quackers = new ArrayList<Quackable>();

    public void add(Quackable quacker) {
        quackers.add(quacker);
    }

    public void quack() {
        Iterator<Quackable> iterator = quackers.iterator();
        while (iterator.hasNext()) {
            Quackable quacker = iterator.next();
            quacker.quack();
        }
    }

    public String toString() {
        return "Flock of Quackers";
    }
}
```

오리떼 관리 기능의 추가

```
void simulate(AbstractDuckFactory duckFactory) {  
    Quackable redheadDuck = duckFactory.createRedheadDuck();  
    Quackable duckCall = duckFactory.createDuckCall();  
    Quackable rubberDuck = duckFactory.createRubberDuck();  
    Quackable gooseDuck = new GooseAdapter(new Goose());  
  
    System.out.println("\nDuck Simulator: With Composite - Flocks");  
  
    Flock flockOfDucks = new Flock(); ← 오리 객체를 포함할 Flock 객체  
  
    flockOfDucks.add(redheadDuck);  
    flockOfDucks.add(duckCall);  
    flockOfDucks.add(rubberDuck);  
    flockOfDucks.add(gooseDuck);  
  
    System.out.println("\nDuck Simulator: Whole Flock Simulation");  
    simulate(flockOfDucks);  
  
    ...  
}
```

오리떼 관리 기능의 추가

```
void simulate(AbstractDuckFactory duckFactory) {
```

```
...
```

```
Flock flockOfMallards = new Flock();
```

MallardDuck만 포함하는 Flock 객체

```
Quackable mallardOne = duckFactory.createMallardDuck();
```

```
Quackable mallardTwo = duckFactory.createMallardDuck();
```

```
Quackable mallardThree = duckFactory.createMallardDuck();
```

```
Quackable mallardFour = duckFactory.createMallardDuck();
```

```
flockOfMallards.add(mallardOne);
```

```
flockOfMallards.add(mallardTwo);
```

```
flockOfMallards.add(mallardThree);
```

```
flockOfMallards.add(mallardFour);
```

```
...
```

```
System.out.println("\nDuck Simulator: Mallard Flock Simulation");
```

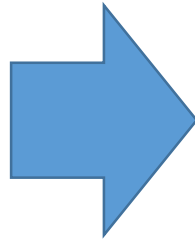
```
simulate(flockOfMallards);
```

```
...
```

```
}
```

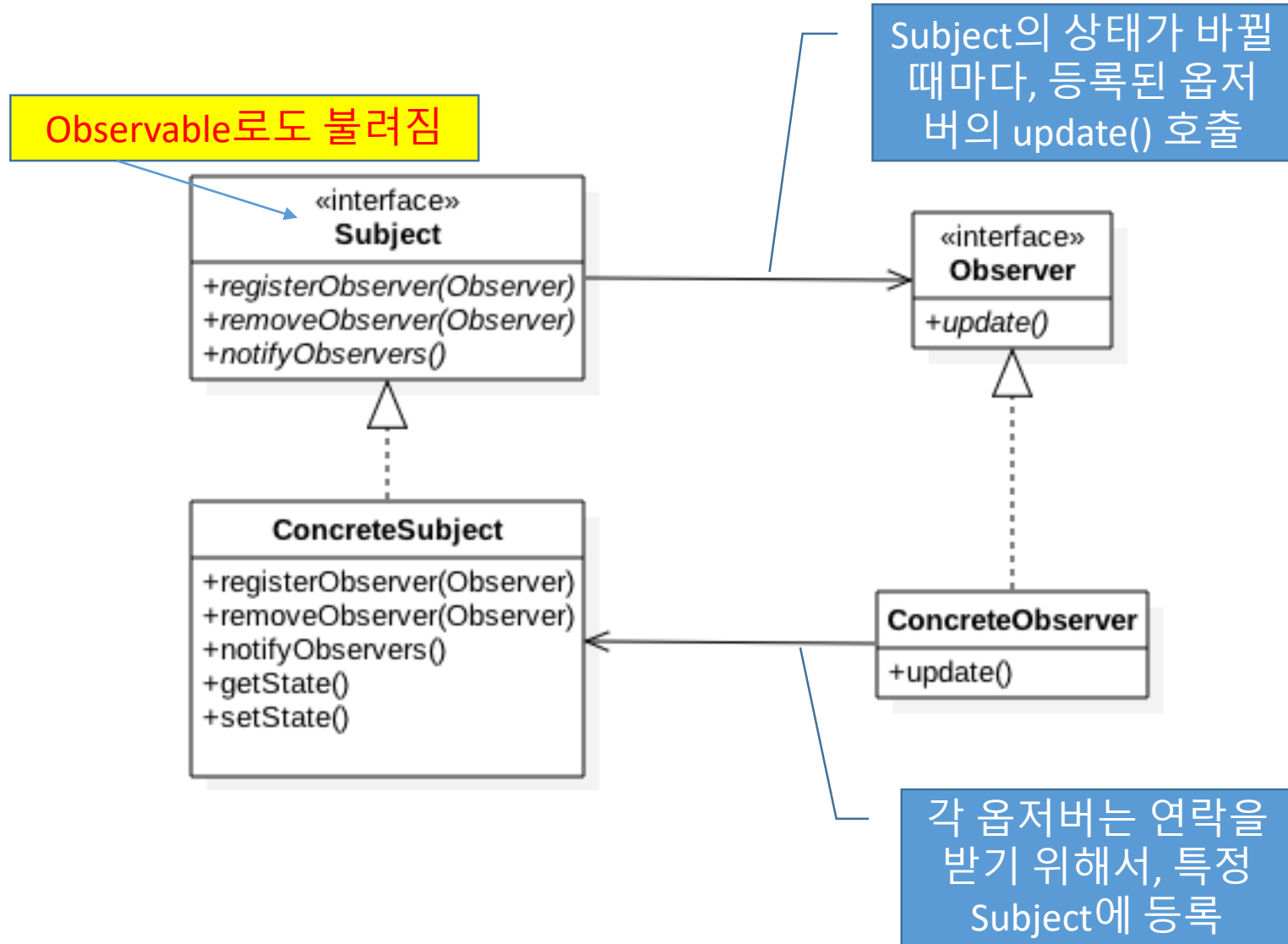
오리들의 상태 실시간 추적

- 오리들을 각각 하나씩 실시간으로 추적할 수 있는 기능을 만들어 주세요..



옵저버 패턴을 이용하여,
오리들의 상태 변화를
통지 받을 수 있도록 구현

옵저버 패턴: 클래스 다이어그램



오리들의 상태 실시간 추적

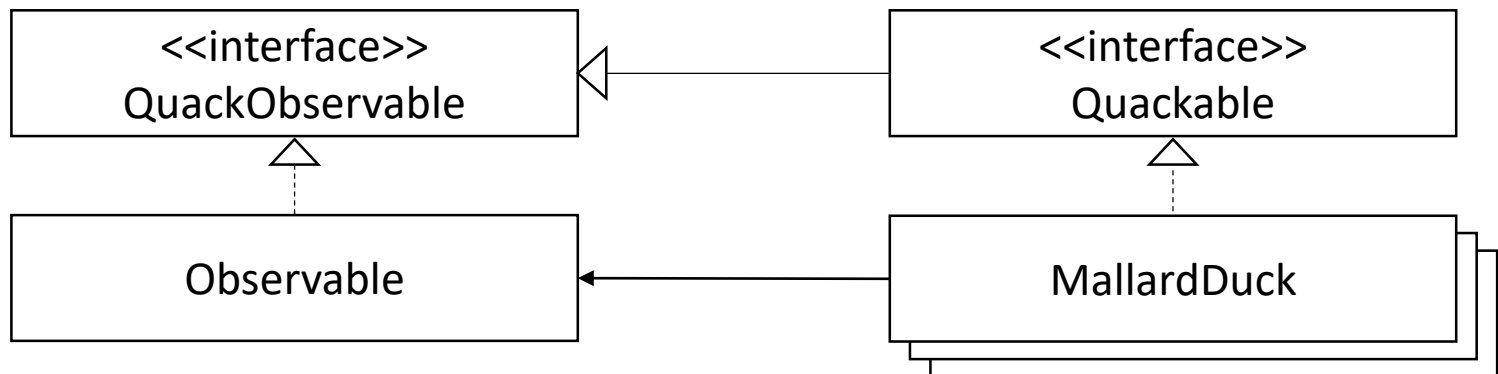
- Observable (Subject) 인터페이스

```
public interface QuackObservable {  
    public void registerObserver(Observer observer);  
    public void notifyObservers();  
}
```

```
public interface Quackable extends QuackObservable {  
    public void quack();  
}
```



오리들의 상태 실시간 추적

- **QuackObservable의 메소드 (registerObserver(), notifyObservers())를 구현하는 방법**
 1. Quackable의 인터페이스를 구현하는 모든 오리 클래스에서 구현하는 방법
 2. java.util.Observable 클래스를 이용하는 방법
 3. **QuackObservable의 메소드를 Observable 이라는 한 클래스 캡슐화 해 놓은 다음 구성을 통해서 오리 클래스에 포함시킴**



오리들의 상태 실시간 추적

```
public class Observable implements QuackObservable {  
    ArrayList<Observer> observers = new ArrayList<Observer>();  
    QuackObservable duck;  
  
    public Observable(QuackObservable duck) {  
        this.duck = duck;  
    }  
  
    public void registerObserver(Observer observer) {  
        observers.add(observer);  
    }  
  
    public void notifyObservers() {  
        Iterator<Observer> iterator = observers.iterator();  
        while (iterator.hasNext()) {  
            Observer observer = iterator.next();  
            observer.update(duck);  
        }  
    }  
}
```



오리들의 상태 실시간 추적

- 오리 클래스에서 Observable 객체 포함

```
public class MallardDuck implements Quackable {  
    Observable observable;  
  
    public MallardDuck() {  
        observable = new Observable(this);  
    }  
    public void quack() {  
        System.out.println("Quack");  
        notifyObservers();  
    }  
    public void registerObserver(Observer observer) {  
        observable.registerObserver(observer);  
    }  
    public void notifyObservers() {  
        observable.notifyObservers();  
    }  
}
```

옵저버에게 상태
변화 공지

observable에 위임

오리들의 상태 실시간 추적

- QuackCounter 데코레이터에서도 **Quackable**을 구현하므로, 변경 사항을 반영 합니다.

```
public class QuackCounter implements Quackable {  
    Quackable duck;  
    static int numberOfQuacks;  
  
    ...  
    public void quack() {  
        duck.quack();  
        numberOfQuacks++;  
    }  
  
    public static int getQuacks() {  
        return numberOfQuacks;  
    }  
    public void registerObserver(Observer observer) { ...}  
    public void notifyObservers() { ... }  
}
```

오리들의 상태 실시간 추적

- 복합객체인 Flock도 **Quackable**을 구현하므로, 변경 사항을 반영 합니다.

```
public class Flock implements Quackable {  
    ArrayList ducks = new ArrayList();  
    ...  
    public void add(Quackable duck) {  
        ducks.add(duck);  
    }  
  
    public void quack() {  
        Iterator<Quackable> iterator = ducks.iterator();  
        while (iterator.hasNext()) {  
            Quackable duck = (Quackable)iterator.next();  
            duck.quack();  
        }  
    }  
    public void registerObserver(Observer observer) { ...}  
    public void notifyObservers() { ... }  
}
```

오리들의 상태 실시간 추적

- Observer 인터페이스 및 구현

```
public interface Observer {  
    public void update(QuackObservable duck);  
}
```

```
public class Quackologist implements Observer {  
  
    public void update(QuackObservable duck) {  
        System.out.println("Quackologist: " + duck + " just quacked.");  
    }  
  
    public String toString() {  
        return "Quackologist";  
    }  
}
```

오리들의 상태 실시간 추적

- 테스트

```
void simulate(AbstractDuckFactory duckFactory) {  
  
    // 오리 및 오리떼 생성  
  
    System.out.println("\nDuck Simulator: With Observer");  
  
    Quackologist quackologist = new Quackologist();  
    flockOfDucks.registerObserver(quackologist);  
  
    simulate(flockOfDucks);  
  
    System.out.println("\nThe ducks quacked " +  
        QuackCounter.getQuacks() +  
        " times");  
}
```

지금까지 했던 일

- 거위도 Quackable로 만들고 싶었고
 - 어댑터 패턴 적용
- 팩소리 회수를 세고 싶었고
 - 데코레이터 패턴 적용
- 오리 객체 생성 코드를 캡슐화하고 싶었고
 - 팩토리 패턴 적용
- 오리떼를 개별 오리와 동일한 방법으로 관리하고 싶었고
 - 컴포지트 패턴 적용
- 개별 오리 및 오리 떼의 상태를 관찰하고 싶었습니다.
 - 옵저버 패턴 적용