# Pemrograman Berbasis Komponen 2

Strategy dan Observer Pattern

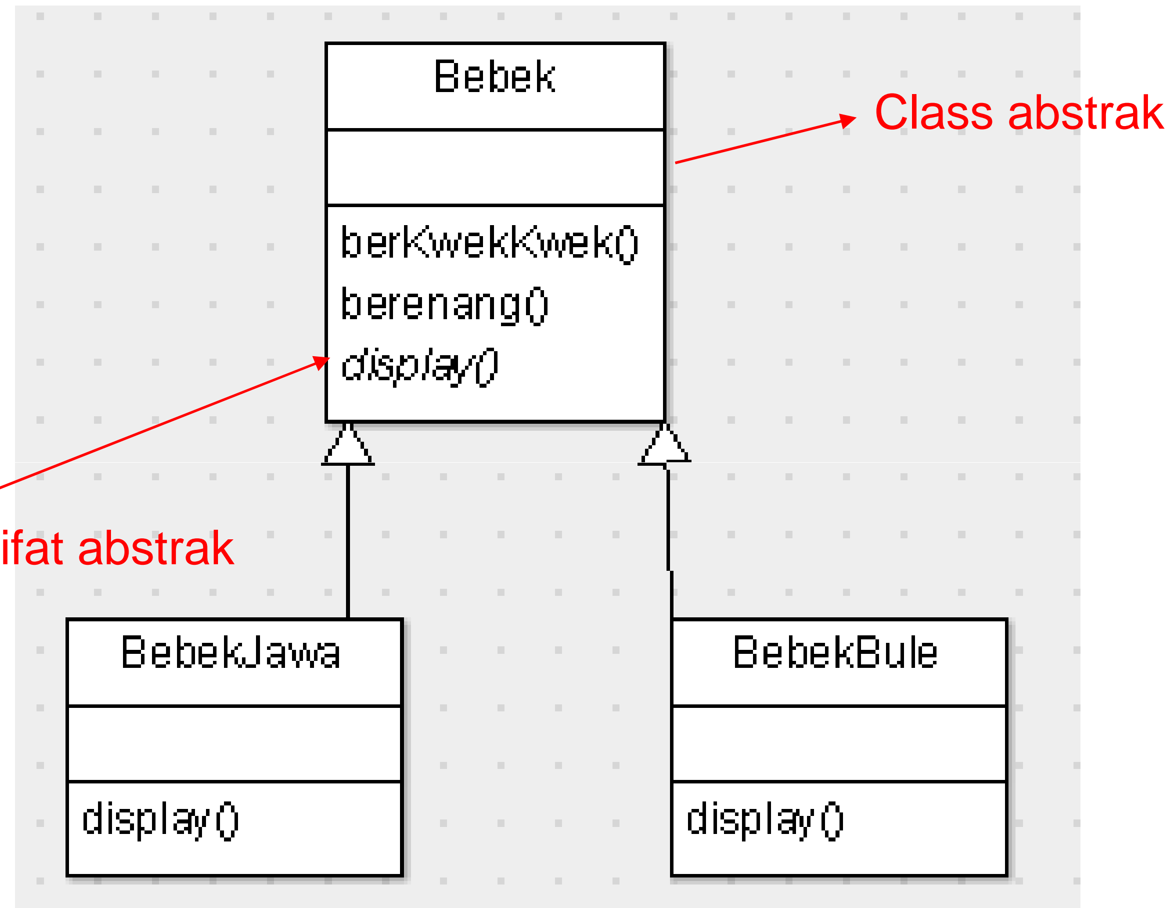
# Background

- Programming is always **change**…
- But this **changes** are sometime the **same**
- So, there will be something - “**a pattern**” to be used in programming problems
- “This pattern” is programming tips from the “**experts**”!
- So we will **use** “the design patterns” and **fit** it to our programming problems!

# Design Pattern

- A design pattern is a **tested solution** to a standard programming problem
- Design patterns are solutions to programming problems that automatically implement **good design techniques**
- A design pattern is the **extension** of object oriented programming (OOP)
- **Beware**: when working on a programming problem, the tendency is to **program** to the problem, **not** in terms of reuse, extensibility, maintainability, or other good design issues.

# Contoh kasus **Pattern**




Class abstrak

Method display() bersifat abstrak

KASUS: **Bebek**

# Kelas Bebek

```
abstract class Bebek {

    public Bebek(){  }

    public void BerKwekKwek(){
            System.out.println("What's Up Doc?");
    }

    public abstract void display();

    public void berenang(){
            System.out.println("semua bebek bisa
mengapung");
    }
}
```

# Perkembangan Kasus

- Ada permintaan untuk menambahkan kemampuan untuk **terbang** pada Bebek
- Dimana kemampuan ini harus diletakkan?

# Tambah **terbang**?

Bebek

berKwekKwek()
berenang()
*display()*
terbang()

BebekJawa

display()

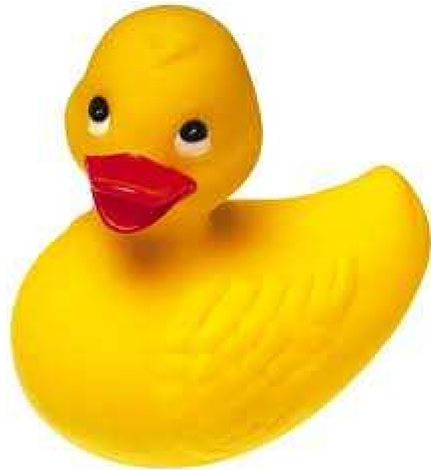
BebekBule

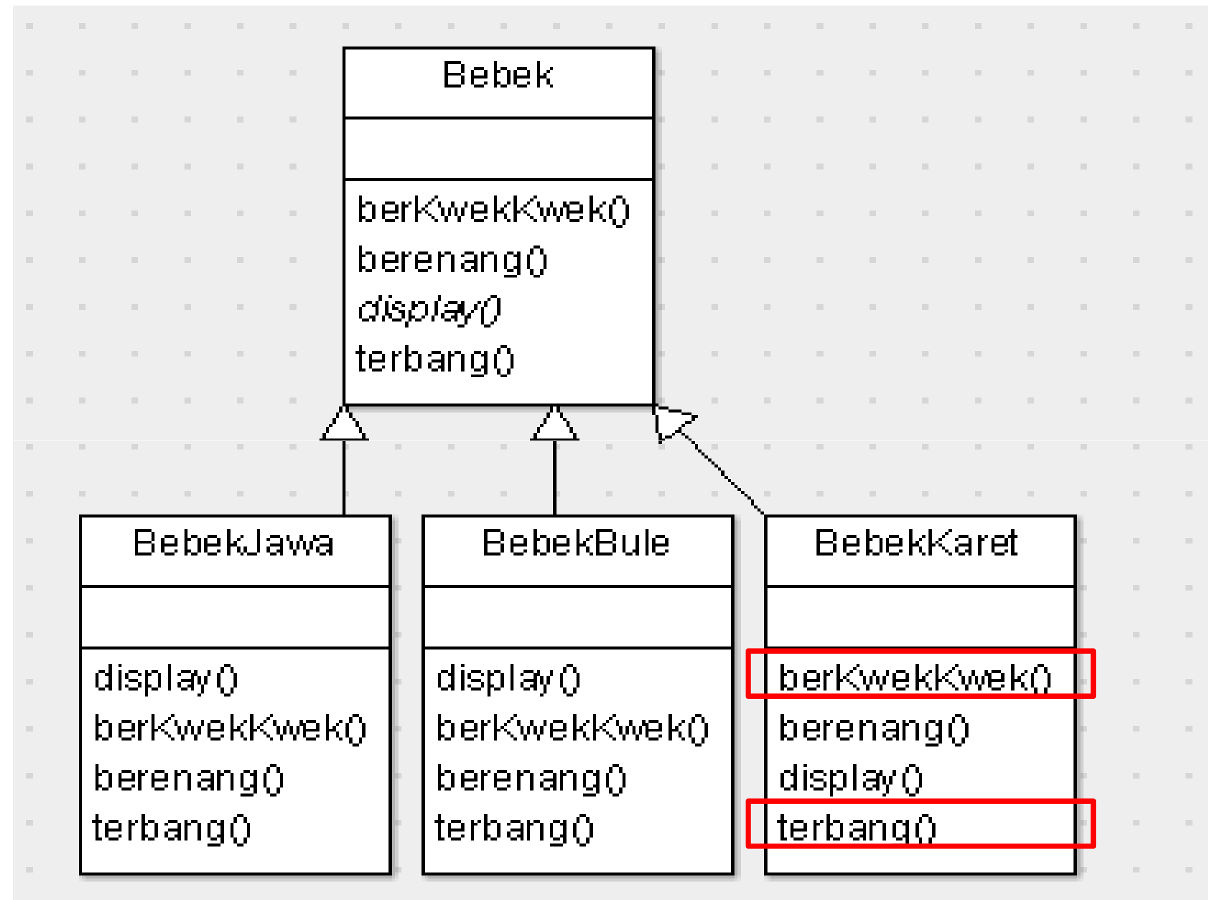
display()

# Kelas bebek (**updated!**)

```
abstract class Bebek {

    public Bebek(){    }

    public void BerKwekKwek(){
        System.out.println("What's Up Doc?");
    }

    public abstract void display();

    public void berenang(){
        System.out.println("semua bebek bisa mengapung");

    }

    public void terbang(){
        System.out.println("Up Up and Away....");
    }
}
```

# Ditambah Bebek baru: **Bebek Karet**



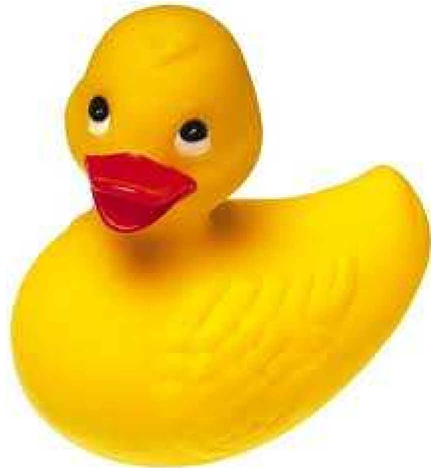



Semua Bebek bisa **terbang**?
Bebek Karet? **TIDAK BISA TERBANG**!
Method **berKwekKwek()** harus **dioverride ->** toet2!

# Ingat bahwa kelas induk mewariskan **semua sifat** ke anaknya!

```
class BebekKaret extends Bebek {
   public BebekKaret() {   }

   public void terbang(){
        System.out.println("Byur....");
     }

   public void BerKwekKwek(){
        System.out.println("Tuit...Tuit..");
   }

   public void display(){
        System.out.println("rubber duck");
   }
}
```

# Jika ditambah lagi **Bebek Kayu**?



Bebek
berKwekKwek()
berenang()
*display()*
terbang()

BebekKayu
berKwekKwek()
berenang()
display()
terbang()

BebekJawa
display()
berKwekKwek()
berenang()
terbang()

BebekBule
display()
berKwekKwek()
berenang()
terbang()

BebekKaret
berKwekKwek()
berenang()
display()
terbang()

Dengan method **terbang()** ada di Superclass,
semua class **memiliki kemampuan terbang(),**
Padahal tidak **semua**!
BebekKayu **tidak bisa:** terbang, berenang dan berKwekKwek()

# Bebek kayu

```
class BebekKayu extends Bebek {
    public BebekKayu()  {    }

    public void terbang(){
          System.out.println(“Gubrak.....”);
     }

    public void BerKwekKwek(){
          System.out.println(".......");
    }

    public void display(){
          System.out.println("wood duck");
    }
}
```

# Kekurangan **Inheritance**

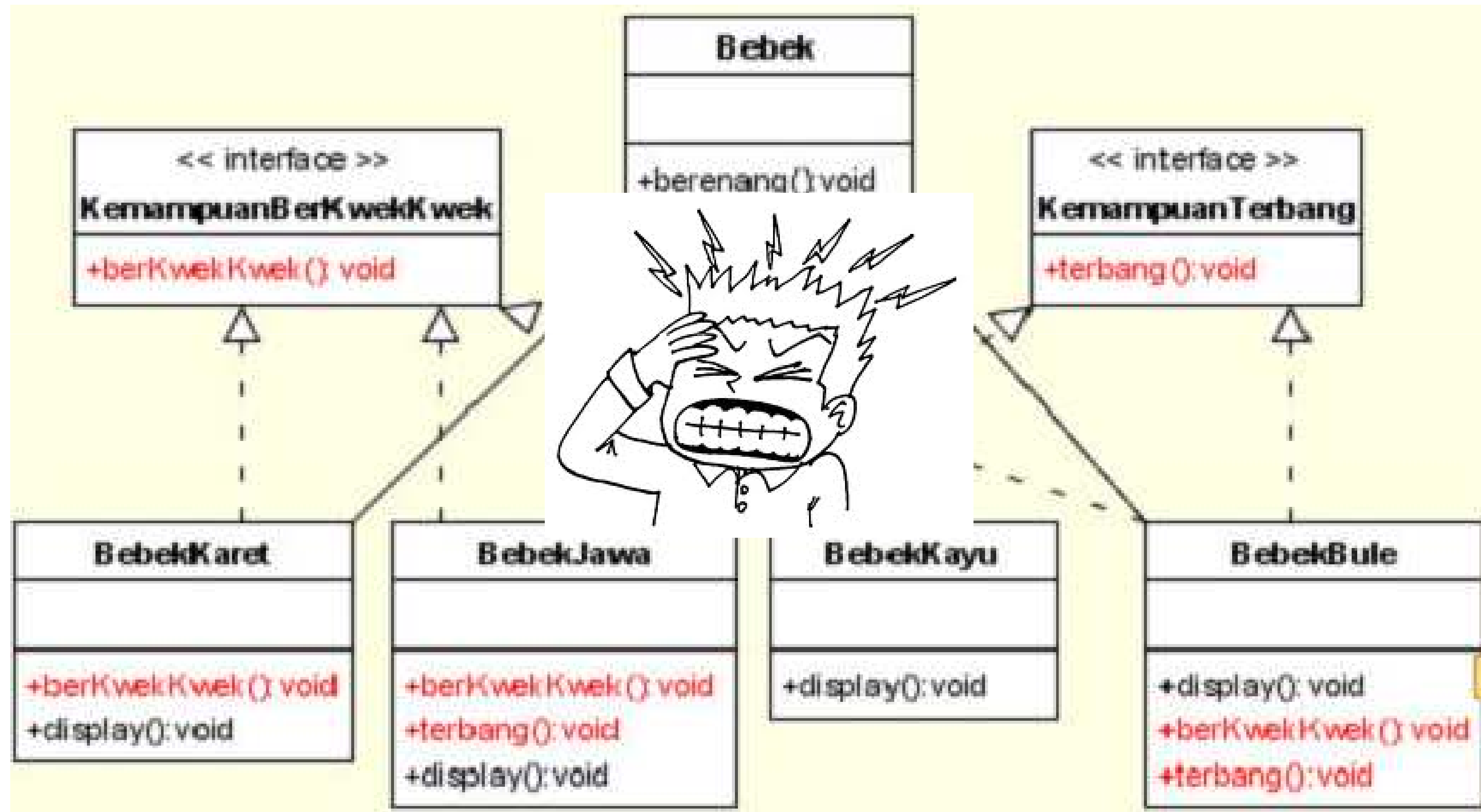
- Sulit dalam maintenance:
  - **Semua** memiliki sifat sesuai dengan **super** classnya
    - Mau tidak mau, **harus**!
  - Perubahan runtime behaviour: **sulit**
  - Kondisi dalam dunia nyata sangat **dinamis**

Solusi?

# Coba gunakan **Interface**

- Agar bisa menambah **bebekKaret** dan **bebekKayu**, coba gunakan interface:
  - Interface KemampuanTerbang
  - Interface KemampuanBerKwekKwek
- Class yang **perlu** kemampuan tambahan, **implementasi** dari salah satu atau kedua interface
- Class yang **tidak perlu** kemampuan tambahan, cukup ambil **turunan** dari super class

# Menggunakan **Interface**?



# Inheritance vs Interface

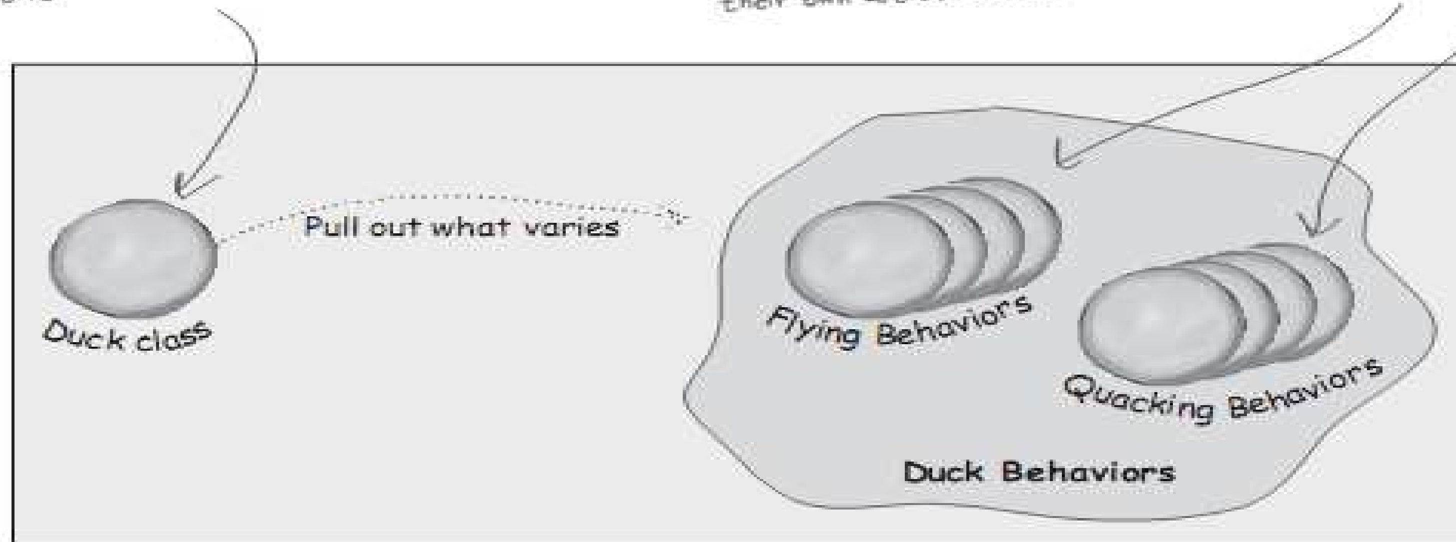
- Kenyataannya tidak semua subclass bisa terbang dan berKwekKwek sehingga **Inheritance** tidak cocok!
- Penggunaan **interface** memang baik, tapi hanya menyelesaikan **sebagian masalah** saja, karena menghilangkan konsep **REUSE**.
  - Create different maintenance nightmare!
  - Harus mengganti semua source class!

# Design Pattern

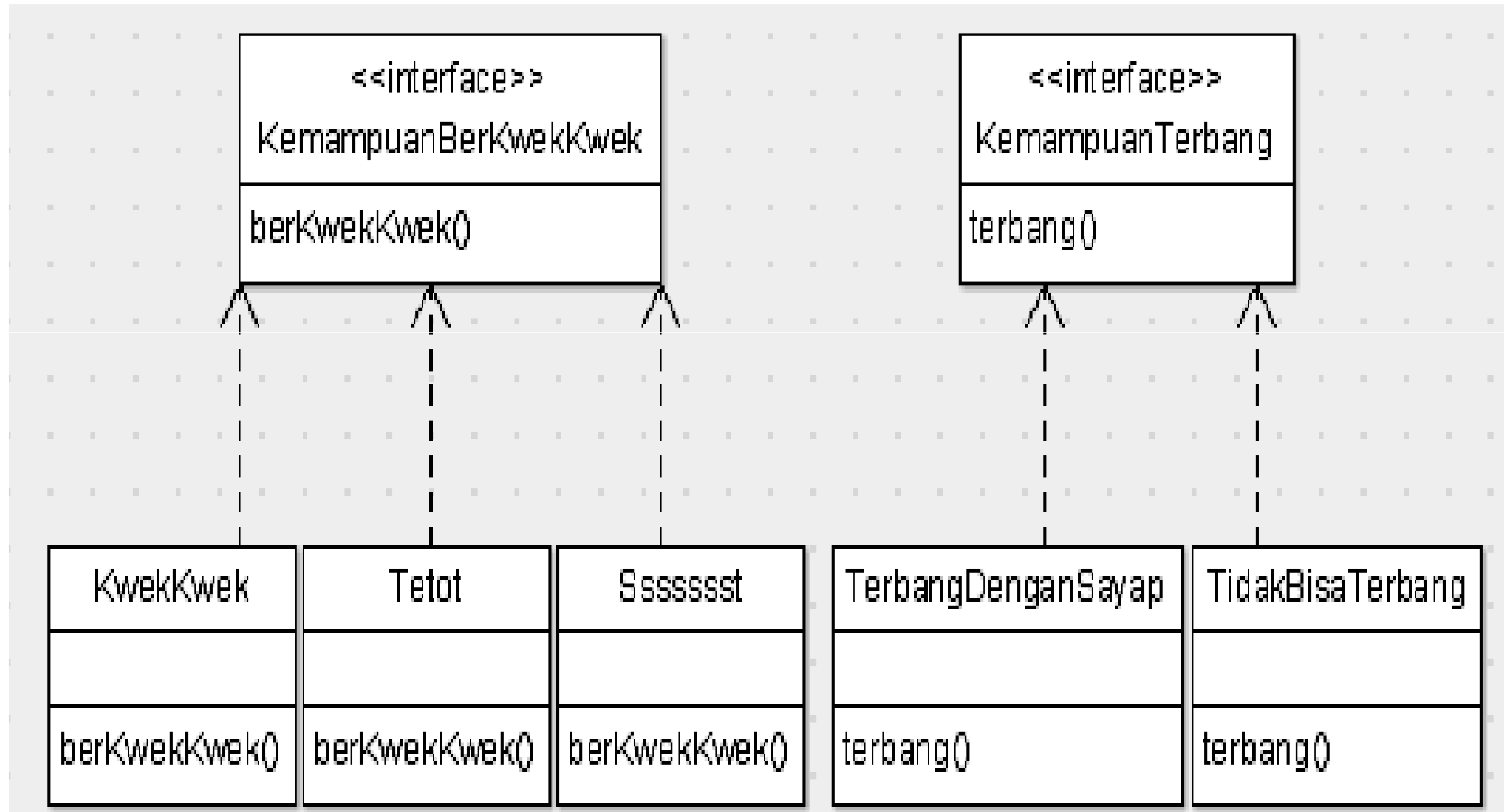
- Satu hal yang konstan pada software development: CHANGE
- **Design Principle**:
  – Identifikasikan aspek dari aplikasi yang paling sering **berubah** dan **pisahkan** dari yang bersifat **statik**
  – Lakukan **enkapsulasi** sehingga bisa dilakukan modifikasi atau pengembangan tanpa mempengaruhi bagian yang lain
  – Koding pada **interface**, dan bukan pada **implementasi**

# Pemisahan

- Pada kasus Bebek, bagian yang sering berubah adalah **terbang()** dan **berKwekKwek()**
- Kita pisah menjadi class baru yang **terpisah**



# Sehingga…



```
interface KemampuanTerbang{
    public void terbang();
}
```

```
class TerbangDenganSayap implements
KemampuanTerbang{
    public void terbang(){
            System.out.println("terbang");
    }
}

class TidakBisaTerbang implements
KemampuanTerbang{
    public void terbang(){
            System.out.println("tidak bisa
terbang");
    }
}
```

```java
interface KemampuanBerKwekKwek{
   public void berKwekKwek();
}

class KwekKwek implements KemampuanBerKwekKwek{
   public void berKwekKwek(){
      System.out.println("Kwek kwek");
   }
}

class Tetot implements KemampuanBerKwekKwek{
   public void berKwekKwek(){
      System.out.println("Tetot");
   }
}

class Ssst implements KemampuanBerKwekKwek{
   public void berKwekKwek(){
      System.out.println("Ssst");
   }
}
```

# Kelas Bebek

```
abstract class Bebek {
    KemampuanBerKwekKwek kemampuanBerKwekKwek;
    KemampuanTerbang kemampuanTerbang;

    public Bebek(){    }

    public void BerKwekKwek(){
        kemampuanBerKwekKwek.berKwekKwek();
    }

    public abstract void display();

    public void berenang(){
        System.out.println("semua bebek bisa 
mengapung");
    }

      public void terbang(){
        kemampuanTerbang.terbang();
      }
}
```

# BebekKayu dan BebekKaret

```
class BebekKayu extends Bebek {
   public BebekKayu() {
      kemampuanBerKwekKwek = new Ssst();
      kemampuanTerbang = new
TidakBisaTerbang();
   }

   public void display(){
      System.out.println("wood duck");
   }
}
```

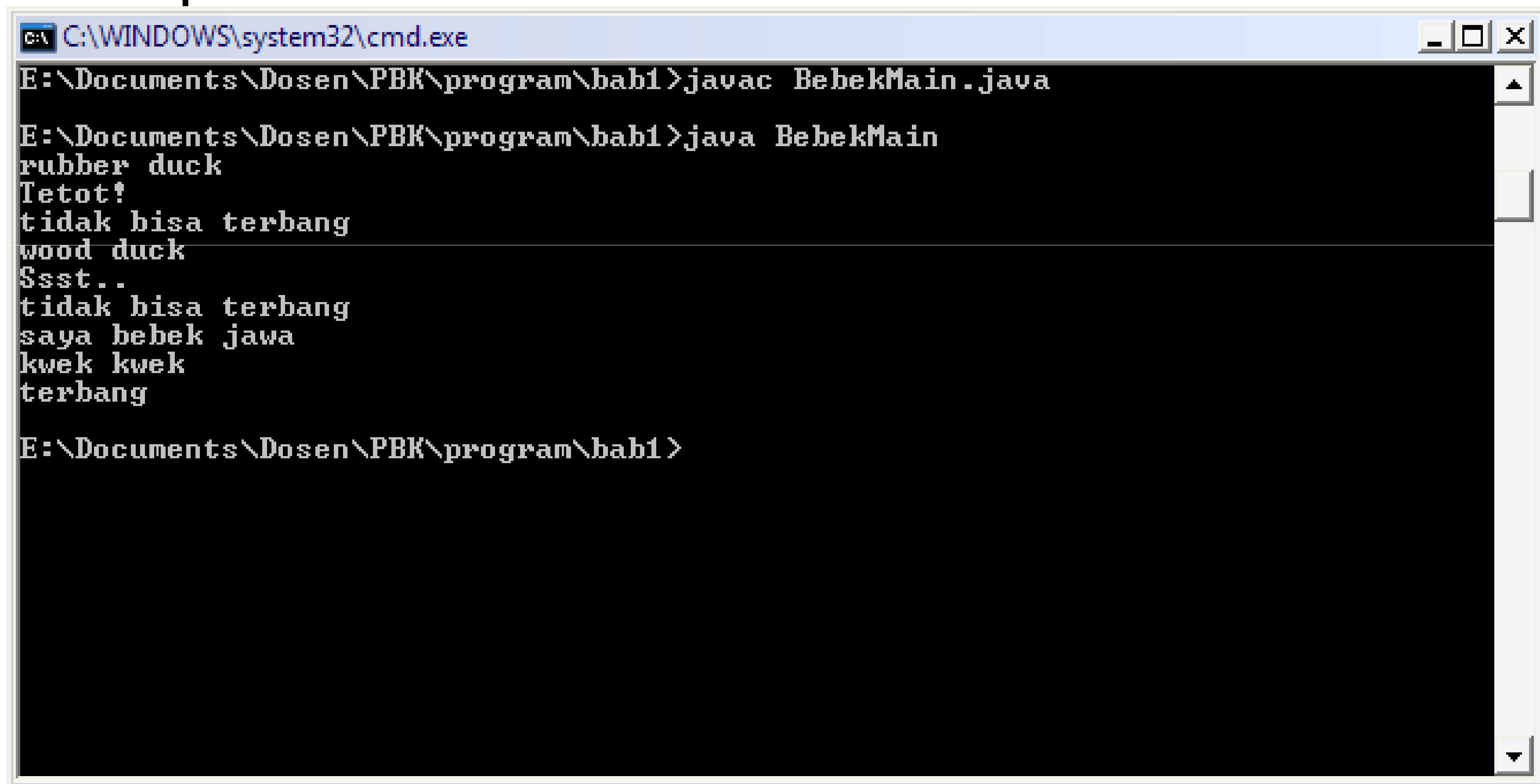
```
class BebekKaret extends Bebek {
   public BebekKaret() {
      kemampuanBerKwekKwek = new Tetot();
      kemampuanTerbang = new
TidakBisaTerbang();
   }

   public void display(){
      System.out.println("rubber duck");
   }
}
```

```
class BebekJawa extends Bebek {
   public BebekJawa() {
      kemampuanBerKwekKwek = new KwekKwek();
      kemampuanTerbang = new
TerbangDenganSayap();
   }

   public void display(){
      System.out.println("saya bebek jawa");
   }
}
```

```
public class BebekMain{
   public static void main(String args[]){
   Bebek B = new BebekKaret();
   B.display();
   B.BerKwekKwek();
   B.terbang();

   Bebek C = new BebekKayu();
   C.display();
   C.BerKwekKwek();
   C.terbang();

   Bebek D = new BebekJawa();
   D.display();
   D.BerKwekKwek();
   D.terbang();
   }
}
```

# Selamat!

- Kita sudah menerapkan **STRATEGY PATTERN**!
- Output:



# Setting Behaviour **Dynamically**

- Tambahkan **setter** method untuk mengubah:
  - KemampuanBerKwekKwek
  - KemampuanTerbang pada class **Bebek**

```
/* Abstract Class Bebebk */
abstract class Bebek {
  KemampuanBerKwekKwek kemampuanBerKwekKwek;
  KemampuanTerbang kemampuanTerbang;

  public Bebek(){

  }

  public void BerKwekKwek(){
    kemampuanBerKwekKwek.berKwekKwek();
  }

  public abstract void display();

  public void berenang(){
    System.out.println("semua bebek bisa mengapung");
  }

  public void terbang(){
    kemampuanTerbang.terbang();
  }

  //setting behaviour dynamically
  public void setTerbang(KemampuanTerbang kt) {
      kemampuanTerbang = kt;
  }

  public void setKwekKwek(KemampuanBerKwekKwek kw) {
    kemampuanBerKwekKwek = kw;
  }
}
```

# Buat class **BebekModel** dan **TerbangDenganRoket**

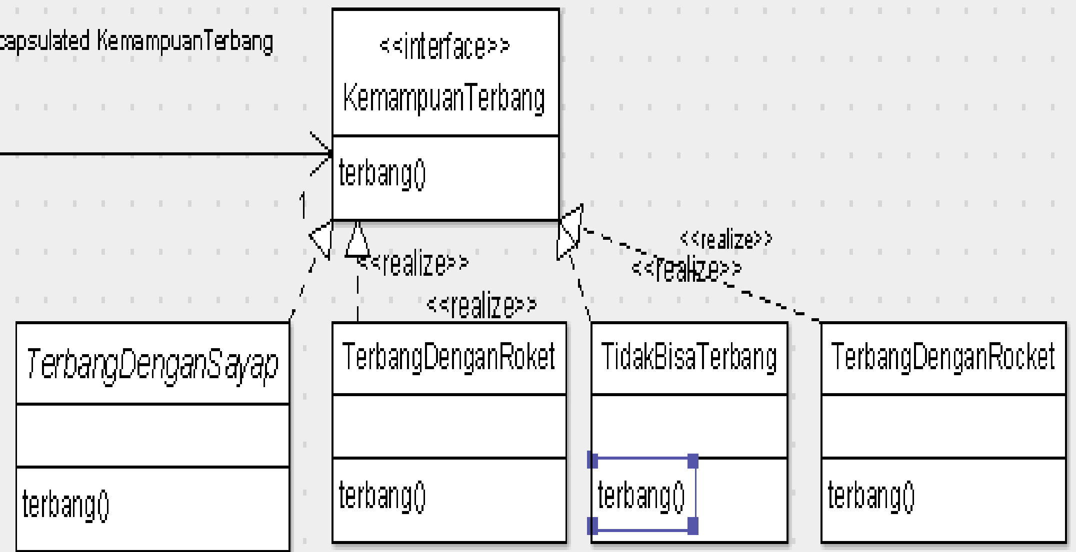
```
/* Class BebekModel */

class BebekModel extends Bebek {
  public BebekModel() {
    kemampuanTerbang = new TidakBisaTerbang();
    kemampuanBerKwekKwek = new Ssst();
  }

  public void display() {
    System.out.println("saya hanya Model sebuah Bebek");
  }
}

class TerbangDenganRoket implements KemampuanTerbang {
  public void terbang() {
    System.out.println("Terbang dengan roket");
  }
}
```

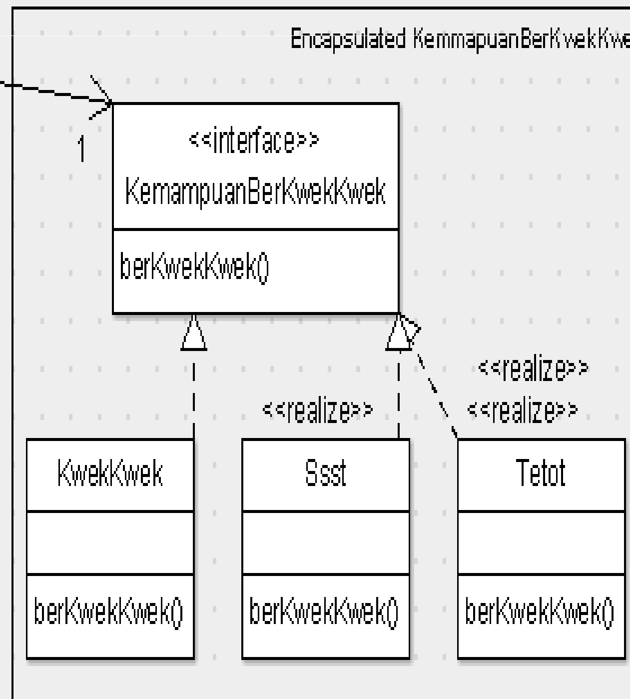
# Tambahkan pada **BebekMain**

```
Bebek model = new BebekModel();
model.terbang();
model.setTerbang(new TerbangDenganRoket());
model.terbang();
```

```
E:\Documents\Dosen\PBK\program\bab1>java BebekMain
rubber duck
Tetot!
tidak bisa terbang
wood duck
Ssst..
tidak bisa terbang
saya bebek jawa
kwek kwek
terbang
tidak bisa terbang
Terbang dengan roket
```

Client Bebek
Bebek
kemampuanTerbang : KemampuanTerbang
kemampuanBerKwekKwek : KemampuanBerKwekKwek
berKwekKwek()
Bebek()
display()
berenang()
terbang()
setTerbang(kt : KemampuanTerbang)
setKwekKwek(kw : KemampuanBerKwekKwek)
BebekKayu
BebekKayu()
display()
BebekKaret
BebekKaret()
display()
BebekKaret
BebekKaret()
display()
BebekModel
BebekModel()
display()
BebekMain
main(args[] : String)


Encapsulated KemampuanTerbang
<<interface>>
KemampuanTerbang
terbang()
<<realize>>
TerbangDenganSayap
terbang()
TerbangDenganRoket
terbang()
TidakBisaTerbang
terbang()
TerbangDenganRocket
terbang()

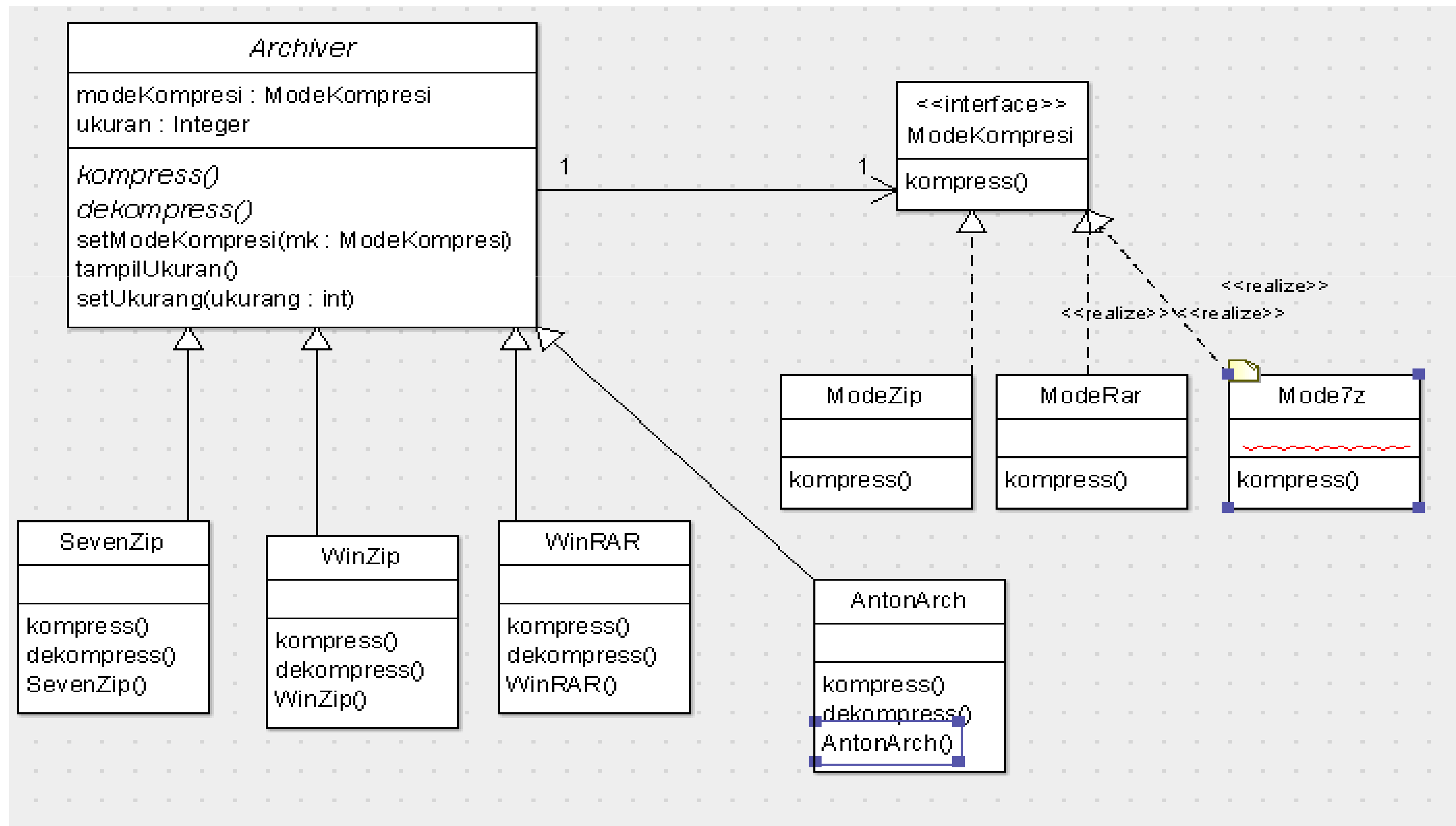

Encapsulated KemmapuanBerKwekKwek
<<interface>>
KemampuanBerKwekKwek
berKwekKwek()
<<realize>>
KwekKwek
berKwekKwek()
Ssst
berKwekKwek()
Tetot
berKwekKwek()

# Strategy Pattern

- Defines **family of algorithms**, **encapsulate** each one, and makes them **interchangable**.
- Strategy lets the algorithm vary **independently** from client that use it

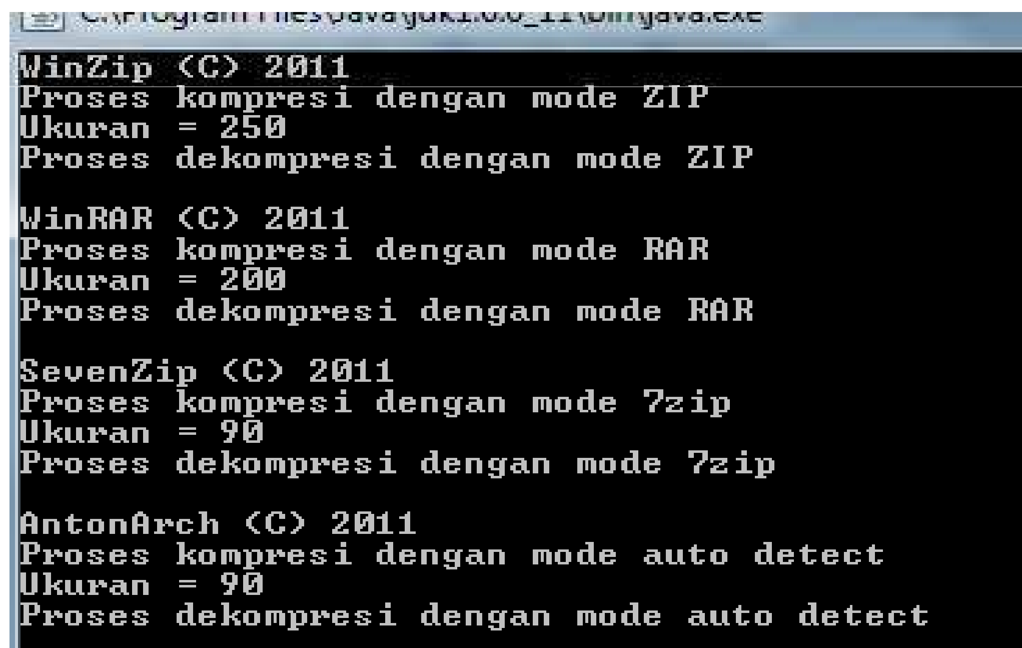
# Contoh Lain

- Archiver



# Main

```
ModeRar rar = new ModeRar();
ModeZip zip = new ModeZip();
Mode7z sevenzip = new Mode7z();

WinZip winzip = new WinZip();
winzip.kompress();
winzip.tampilUkuran();
winzip.dekompress();
System.out.println ();

WinRAR winrar = new WinRAR();
winrar.kompress();
winrar.tampilUkuran();
winrar.dekompress();
System.out.println ();

SevenZip win7zip = new SevenZip();
win7zip.kompress();
win7zip.tampilUkuran();
win7zip.dekompress();
System.out.println ();

AntonArch arc = new AntonArch();
arc.kompress();
arc.tampilUkuran();
arc.dekompress();
System.out.println ();
```



# OBSERVER PATTERN

# Project : The Weather Monitoring Application

- Menggunakan WeatherData object yg bisa mengambil current condition (temperatur, kelembapan, dan tekanan)

- Harus membuat kemampuan menampilkan:
  - Current condition, weather statistics, simple forecast

# Project : The Weather Monitoring Application

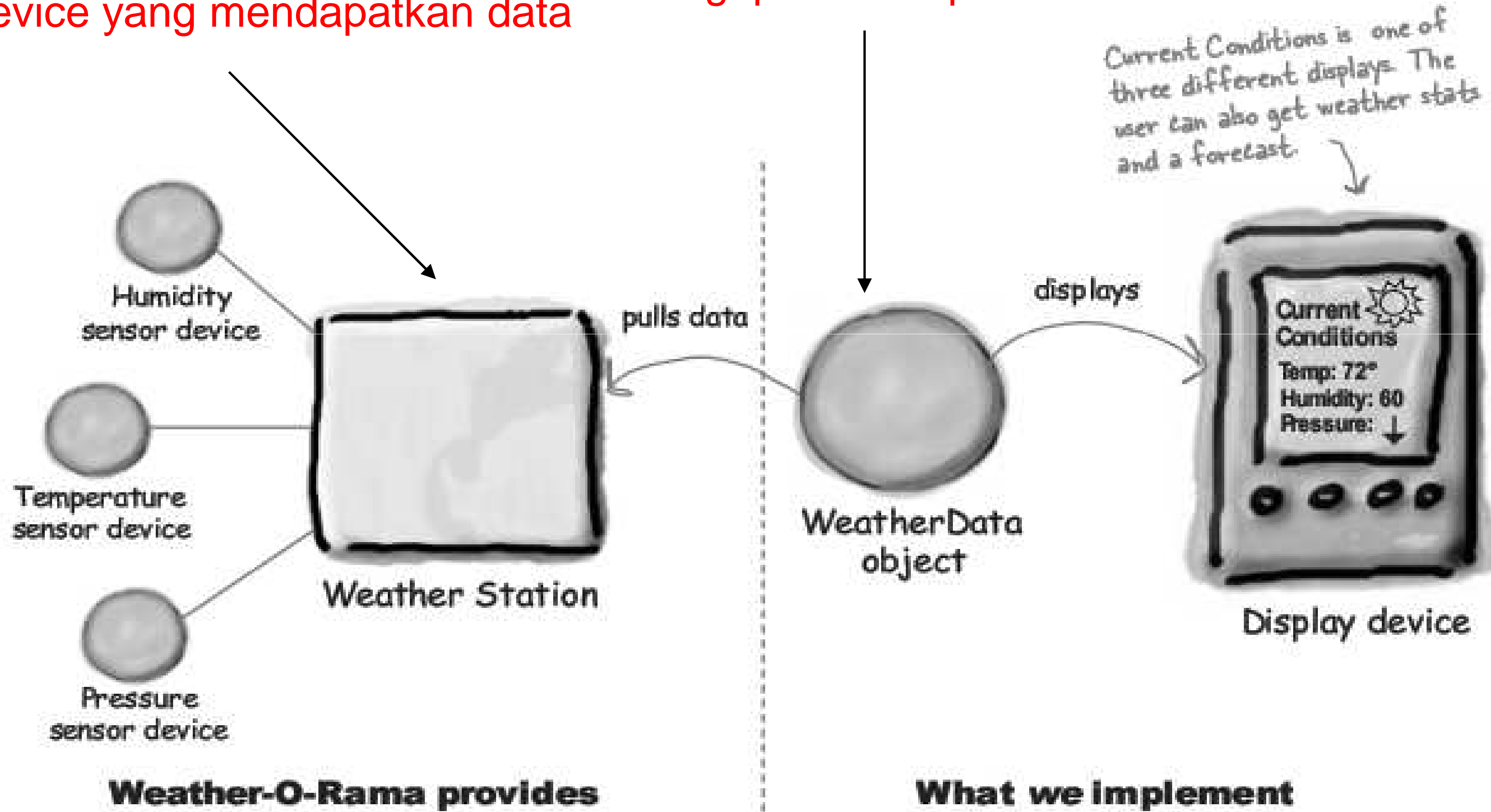
- Yang harus bisa diexpand:
  - Harus bisa dibuat API nya sehingga para developer bisa menggunakan API (dan bayar pada Weather-O-Rama)

- Kita hanya disediakan WeatherData source code!

# Desain awal

Melacak data dari weather station dan mengupdate tampilan

Device yang mendapatkan data



# Yang kita tahu

- WeatherData punya getter method untuk mengambil temperatur, kelembapan, dan tekanan
- Method measureChanged() dipanggil setiap saat ketika data tersedia dan ada perubahan
- Kita harus membuat display untuk current condition, statistic, dan forecast
- System harus bisa diexpand
  - Developer lain boleh membuat elemen lain
  - Pengguna boleh tambah/menghapus elemen yang diinginkan
  - Jenis yang diketahui baru 3 (kondisi aktual, statistik, dan perkiraan)

# Implementasi Awal

```
public class WeatherData {

  public void measurementsChanged() {
    float temp = getTemperature();
    float humidity = getHumidity();
    float preasure = getPreasure();

    currentConditionDisplay.update(temp,humidity,preasure);
    statisticDisplay.update(temp,humidity,preasure);
    forecastDisplay.update(temp,humidity,preasure);
  }

  public void getTemperature() {
  }

  public void getHumidity() {
  }

  public void getPreasure() {
  }
}
```

# berdasarkan **Strategy Pattern**?

- Kita harus menenkapsulasi fungsi **update** karena selalu **berubah**
- Jika kita membuat langsung **implementasi konkret**, maka kita akan kesulitan untuk mengubah elemen2 display lain tanpa mengubah program
- Kita sebaiknya menggunakan **interface umum** yang memiliki method update() yg menerima parameter temp, humidity, dan presure

- Kita akan gunakan **OBSERVER PATTERN**

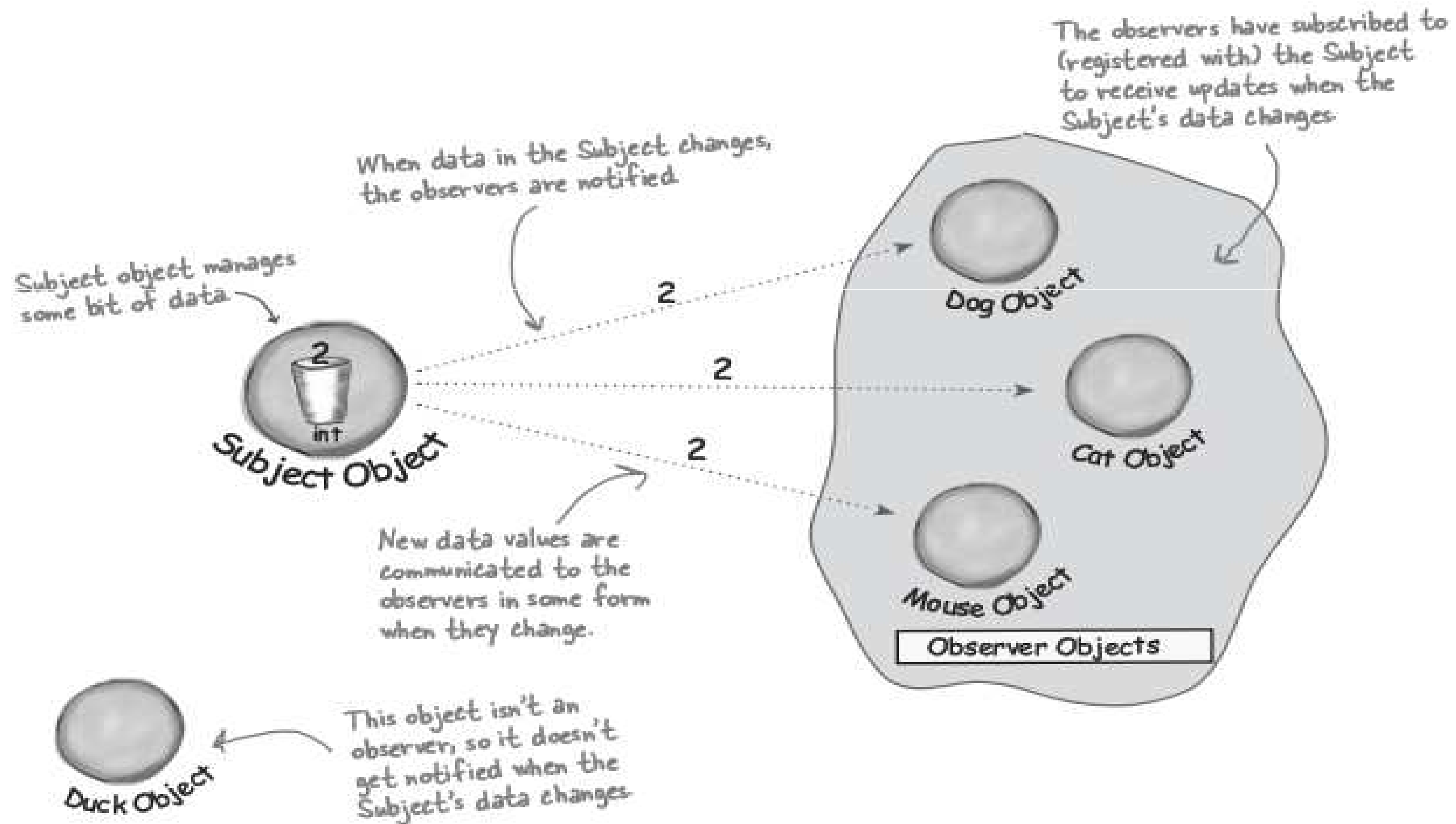
# Timeout

**Kita pelajari pola observer dahulu**

# Studi Kasus

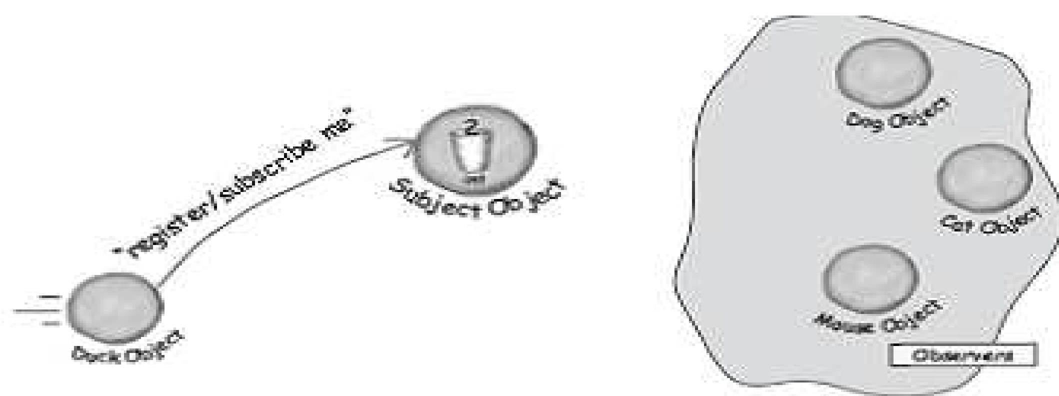
- Studi kasus: Langganan Surat Kabar
- Penerbit menerbitkan surat kabar
- Kita bisa mendaftarkan diri untuk berlangganan
- Selama kita berlangganan (dan membayar), kita pasti dapat surat kabar itu
- Kita bisa berhenti berlangganan kapan pun
- Pelanggan tidak hanya kita

- Publisher + subscriber = Observer Pattern
- Publisher = SUBJECT
- Subscriber = OBSERVERS



**A Duck object comes along and tells the Subject that it wants to become an observer.**

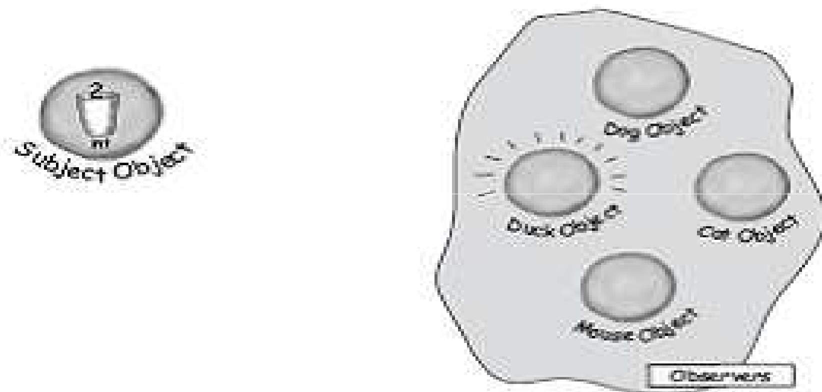
Duck really wants in on the action; those ints Subject is sending out whenever its state changes look pretty interesting...




---

**The Duck object is now an official observer.**

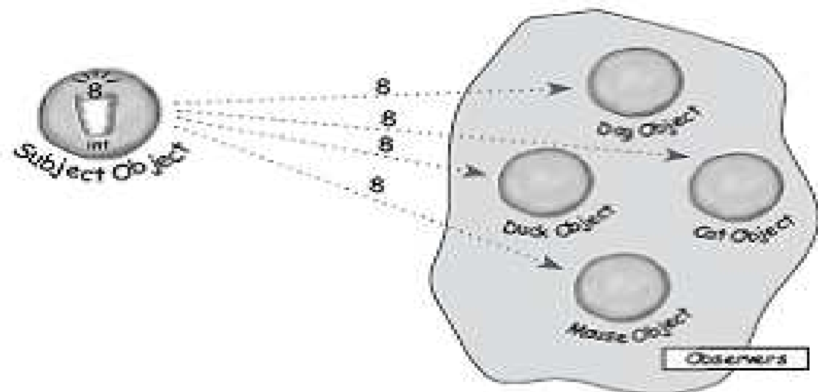
Duck is psyched... he's on the list and is waiting with great anticipation for the next notification so he can get an int.




---

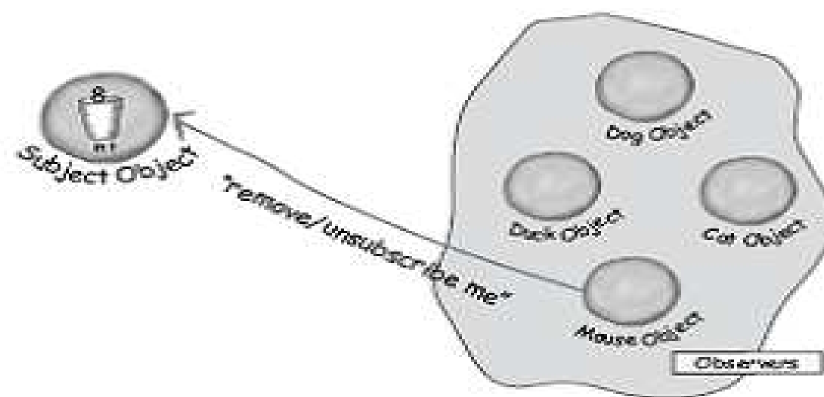
**The Subject gets a new data value!**

Now Duck and all the rest of the observers get a notification that the Subject has changed.



**The Mouse object asks to be removed as an observer.**

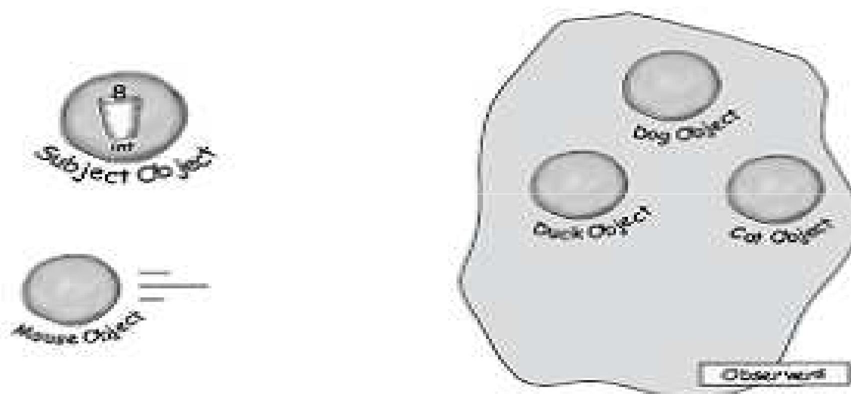
The Mouse object has been getting ints for ages and is tired of it, so it decides it's time to stop being an observer.




---

**Mouse is outta here!**

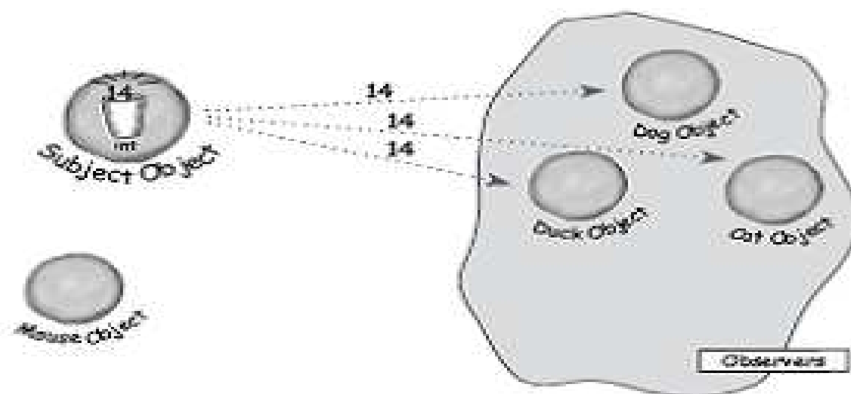
The Subject acknowledges the Mouse's request and removes it from the set of observers.




---

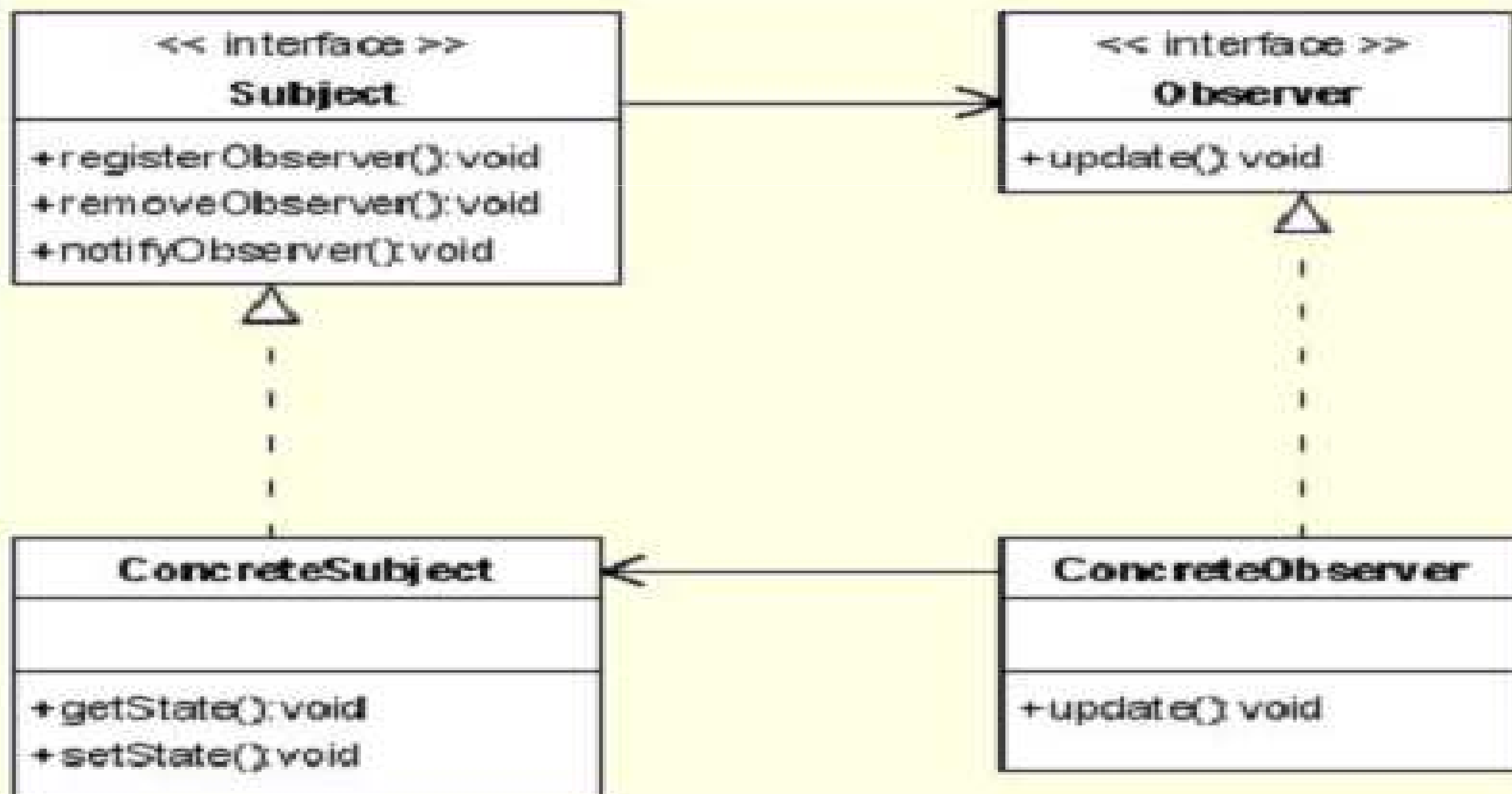
**The Subject has another new int.**

All the observers get another notification, except for the Mouse who is no longer included. Don't tell anyone, but the Mouse secretly misses those ints... maybe it'll ask to be an observer again some day.



# Definisi

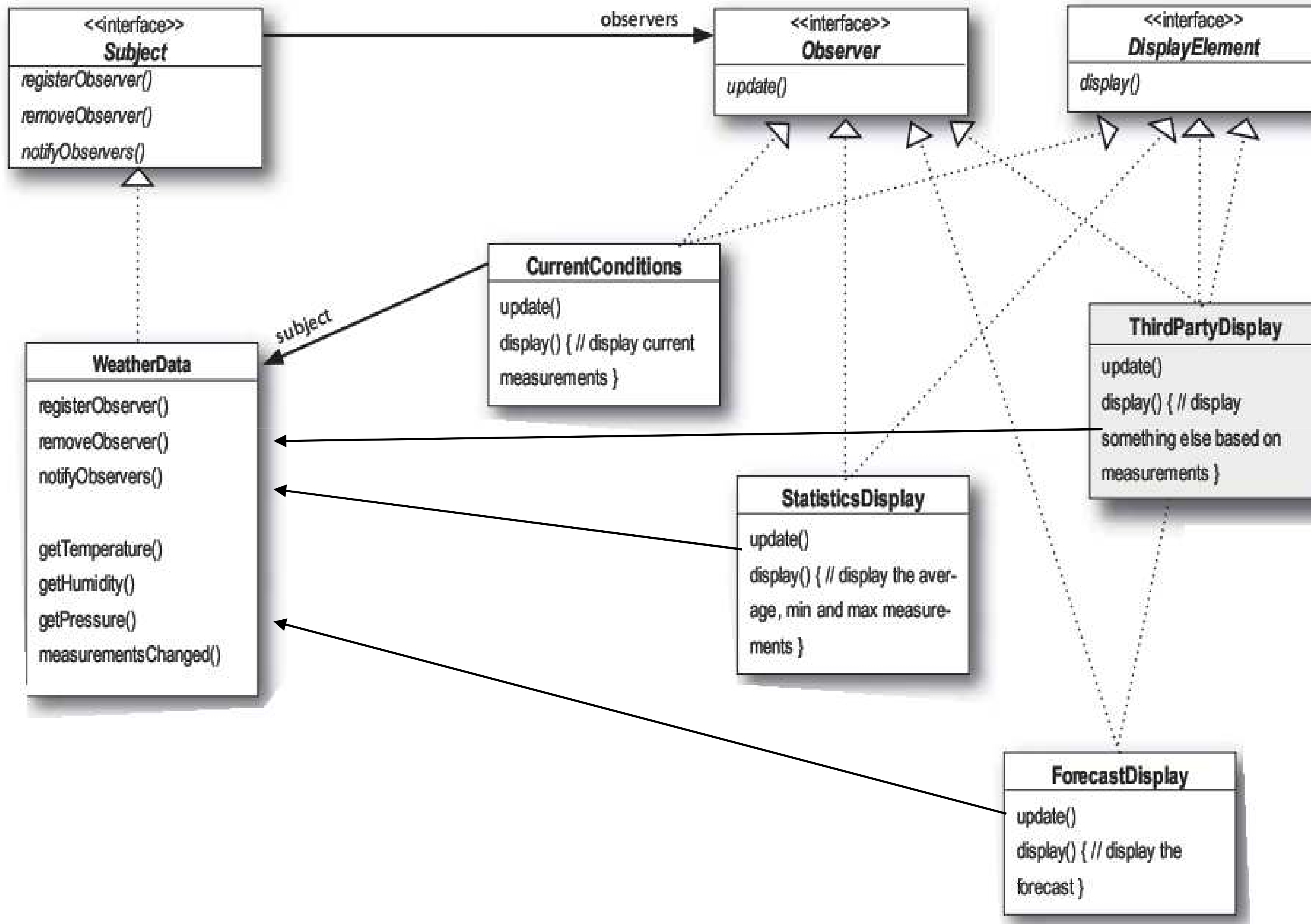
"Defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically"



# Penjelasan

- **Subject Interface**: digunakan untuk mendaftarkan, menghapus, dan memberi tahu Observer
- Subject boleh memiliki **lebih dari satu** Observer
- **Observer Interface**: method update() digunakan jika state Subject **berubah**
- ConcreteSubject: implementasi real interface Subject
- ConcreteObserver: implementasi real interface Observer

# BACK: Weather Monitoring Application



# Implementasi Subject, Observer, & DisplayElement

```
public interface Subject {
  public void registerObserver(Observer
  o);
  public void removeObserver(Observer o);
  public void notifyObservers();
}

public interface Observer {
  public void update(float temp, float
  humidity, float pressure);
}

public interface DisplayElement {
  public void display();
}
```

# WeatherData

```
import java.util.*;

public class WeatherData implements Subject {
  private ArrayList observers;
  private float temperature;
  private float humidity;
  private float pressure;

  public WeatherData() {
      observers = new ArrayList();
  }

  public void registerObserver(Observer o) {
      observers.add(o);
  }

  public void removeObserver(Observer o) {
      int i = observers.indexOf(o);
      if (i >= 0) {
          observers.remove(i);
      }
  }
```

# WeatherData

```
public void notifyObservers() {
    for (int i = 0; i < observers.size(); i++) {
            Observer observer =
(Observer)observers.get(i);
            observer.update(temperature, humidity,
pressure);
    }
}

public void measurementsChanged() {
    notifyObservers();
}

public void setMeasurements(float temperature,
float humidity, float pressure) {
    this.temperature = temperature;
    this.humidity = humidity;
    this.pressure = pressure;
    measurementsChanged();
}
```

# WeatherData

```
  public float getTemperature() {
      return temperature;
  }

  public float getHumidity() {
      return humidity;
  }

  public float getPressure() {
      return pressure;
  }
}
```

# CurrentConditionsDisplay

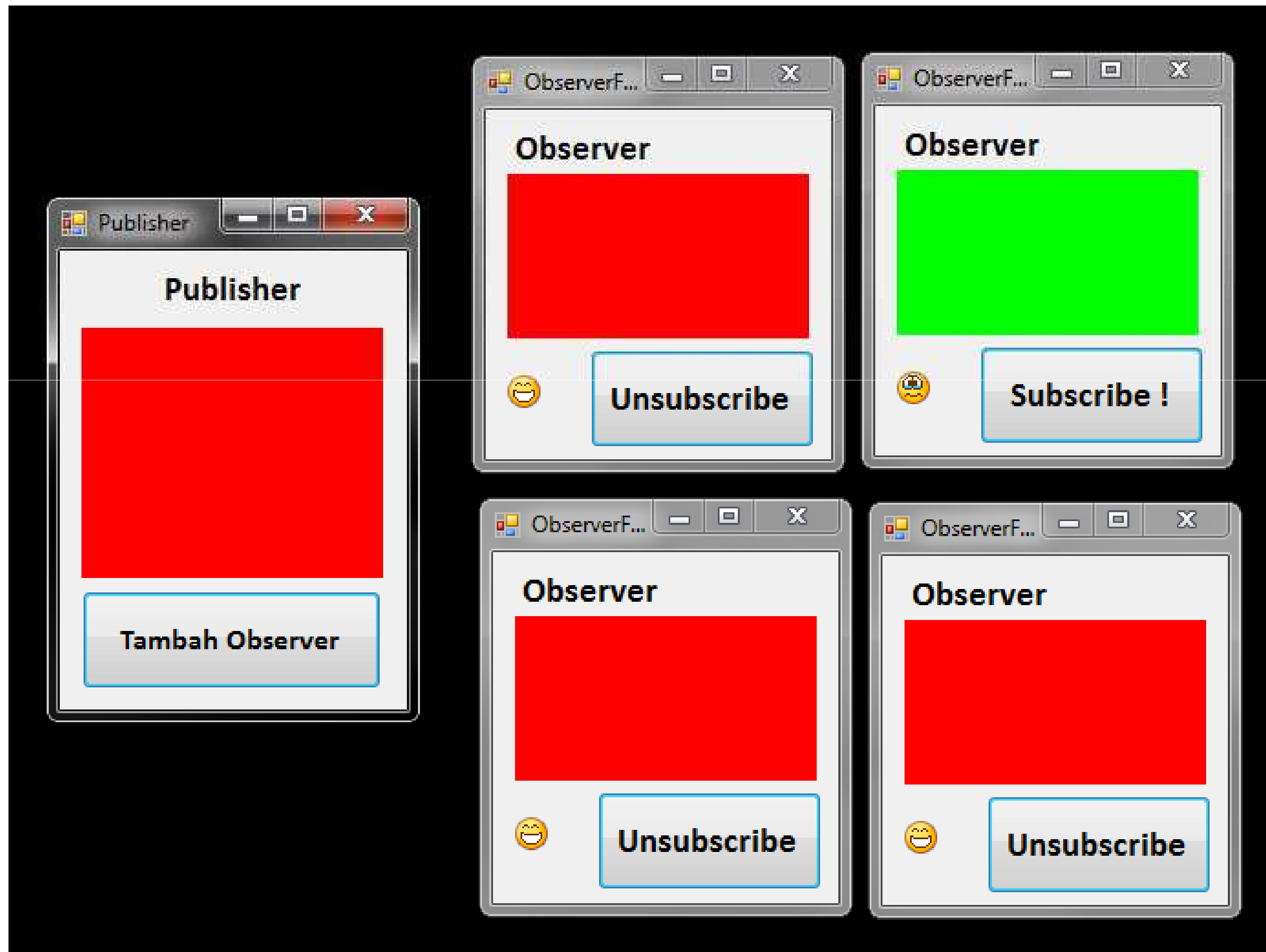
```
public class CurrentConditionsDisplay implements Observer,
   DisplayElement {
   private float temperature;
   private float humidity;
   private Subject weatherData;

   public CurrentConditionsDisplay(Subject weatherData) {
       this.weatherData = weatherData;
       weatherData.registerObserver(this);
   }

   public void update(float temperature, float humidity,
   float pressure) {
       this.temperature = temperature;
       this.humidity = humidity;
       display();
   }

   public void display() {
       System.out.println("Current conditions: " +
   temperature                  + "F     degrees and " +
   humidity + "% humidity");
   }
}
```

# Hasil

```
import java.util.*;

public class WeatherStation {
  public static void main(String[] args) {
      WeatherData weatherData = new WeatherData();
      CurrentConditionsDisplay currentDisplay =
            new CurrentConditionsDisplay(weatherData);
      StatisticsDisplay statisticsDisplay = new
  StatisticsDisplay(weatherData);
      ForecastDisplay forecastDisplay = new
  ForecastDisplay(weatherData);
      weatherData.setMeasurements(80, 65, 30.4f);
      weatherData.setMeasurements(82, 70, 29.2f);
      weatherData.setMeasurements(78, 90, 29.2f);
  }
}
```

```
E:\Documents\Dosen\PBK\program\bab2\observer\weather>java WeatherStation
Current conditions: 80.0F degrees and 65.0% humidity
Avg/Max/Min temperature = 80.0/80.0/80.0
Forecast: Improving weather on the way!
Current conditions: 82.0F degrees and 70.0% humidity
Avg/Max/Min temperature = 81.0/82.0/80.0
Forecast: Watch out for cooler, rainy weather
Current conditions: 78.0F degrees and 90.0% humidity
Avg/Max/Min temperature = 80.0/82.0/78.0
Forecast: More of the same
```

# Contoh Kasus Lain



# Studi Kasus Lain

- Swing pada Java juga menggunakan Observer Pattern
- Bisa sebutkan kasus-kasus apalagi yang bisa dipecahkan dengan Strategy Pattern atau Observer Pattern ?

# Design Pattern

- Design pattern merupakan best practices yang sudah teruji
- Hanya cocok untuk kasus tertentu saja
- Pahami permasalahan, pilih pattern yang sesuai

# Next

- Factory Pattern
  - Produksi object yang bisa dicustomize
- Singleton
  - Cukup satu saja (one of a kind)