Praktikum RPLBO 14

Observer dan Chain of Responsibility Pattern

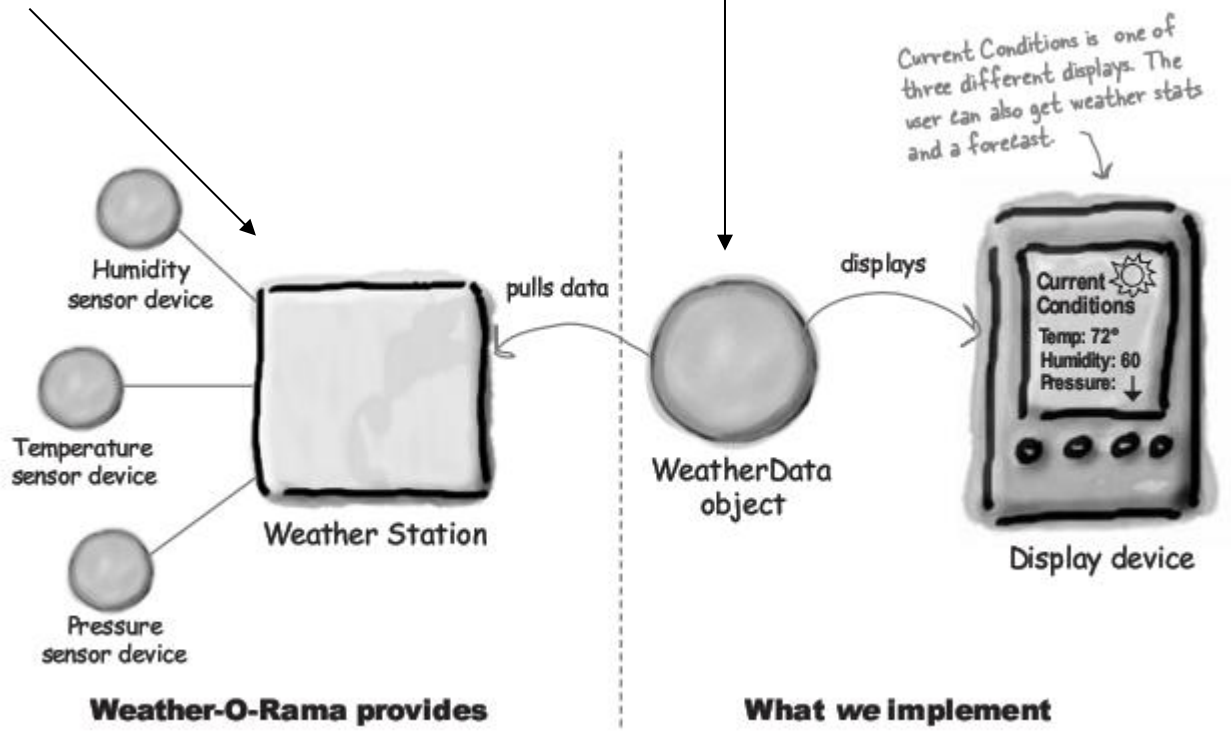
Kasus : The Weather Monitoring Application

- Menggunakan WeatherData object yg bisa mengambil current condition (temperatur, kelembapan, dan tekanan)
- Harus membuat kemampuan menampilkan:
  - Current condition, weather statistics, simple forecast
- Yang harus bisa diexpand:
  - Harus bisa dibuat API nya sehingga para developer bisa menggunakan API (dan bayar pada kita)
- 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 **Design Pattern**

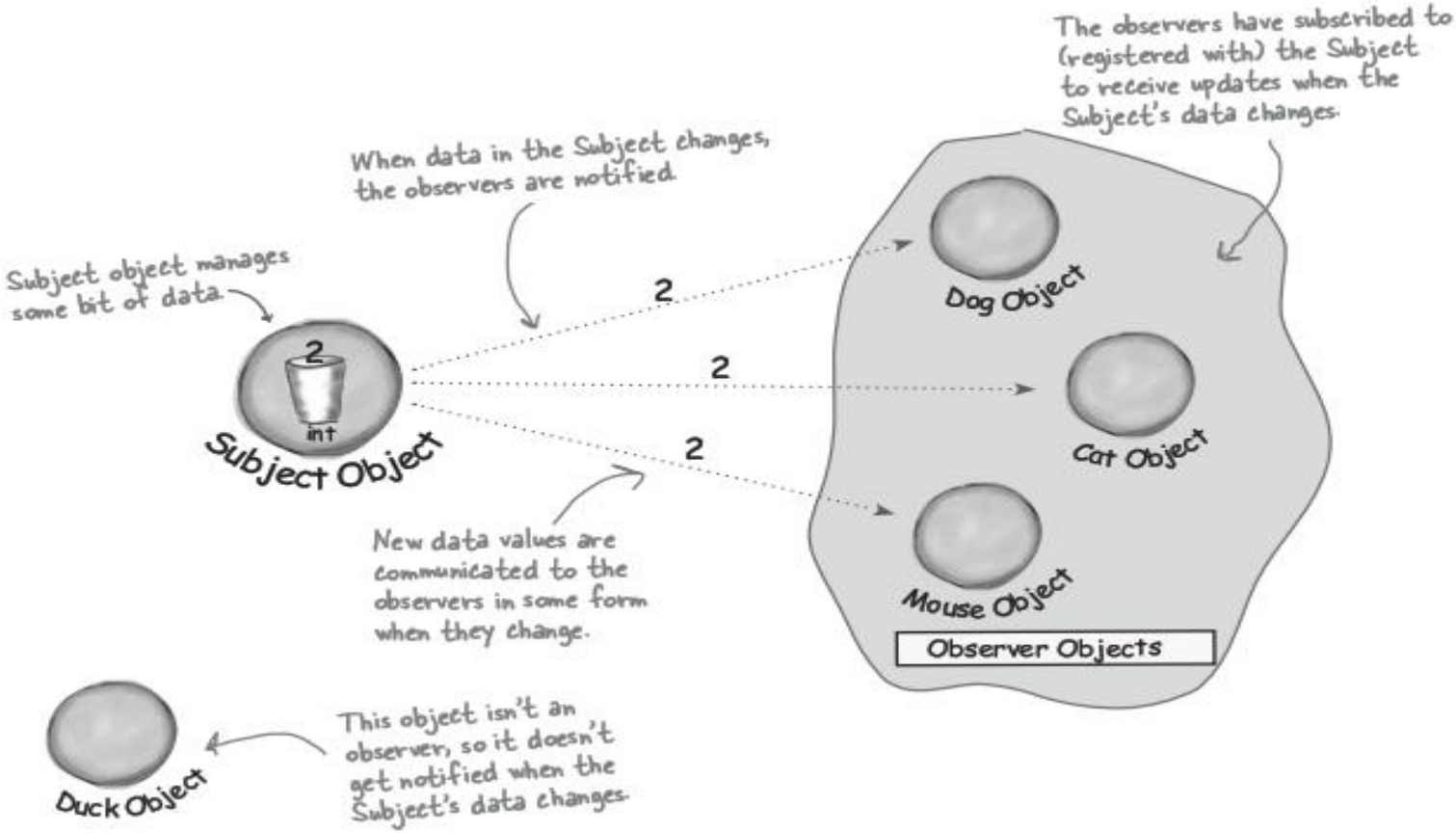
- Kita harus menenkapsulasi fungsi **update** karena selalu **berubah**
- Jika kita membuat langsung **implementasi konkret**, maka kita akan kesulitan untuk mengubah elemen-elemen display lain tanpa mengubah program
- Kita sebaiknya menggunakan **interface umum** yang memiliki method update() yg menerima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

- Kita akan gunakan **OBSERVER PATTERN**

Analogi Observer

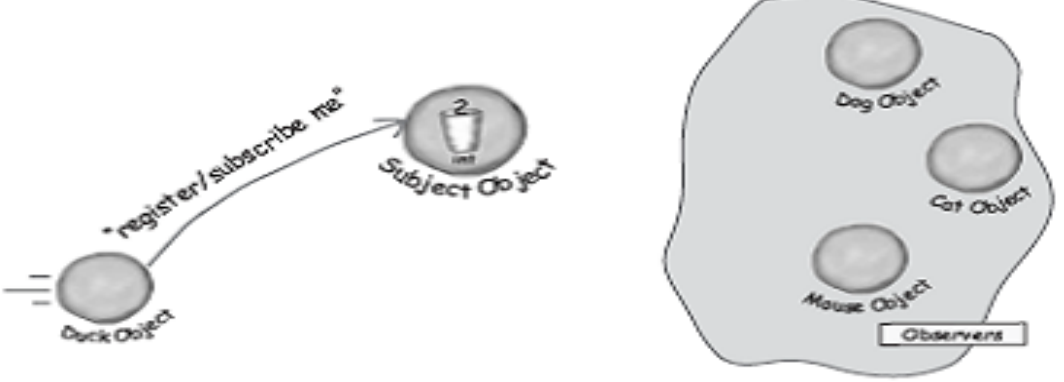
- **Studi kasus**: Langganan Surat Kabar
- Penerbit **menerbitkan** surat kabar
- Kita bisa **mendaftarkan** diri untuk berlangganan
- Selama kita **berlangganan** (dan membayar), kita pasti dapat surat kabar itu
- Kita bisa **berhenti** berlangganan kapan pun
- Pelanggan tidak hanya kita (banyak)

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



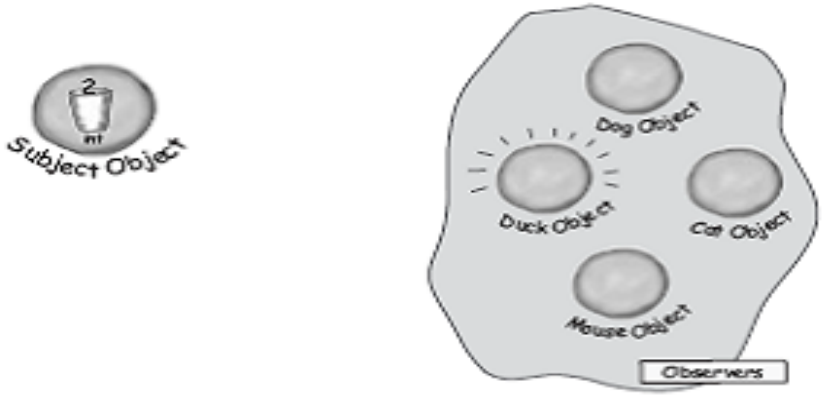
A Duck object comes along and tells the Subject that it wants to become an observer.

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



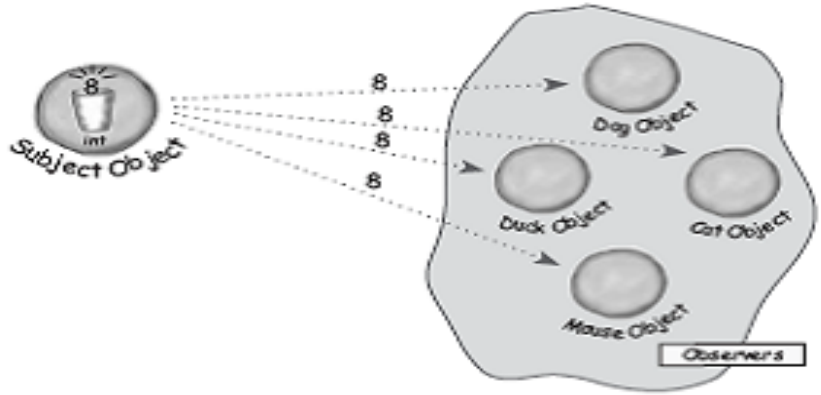
The Duck object is now an official observer.

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



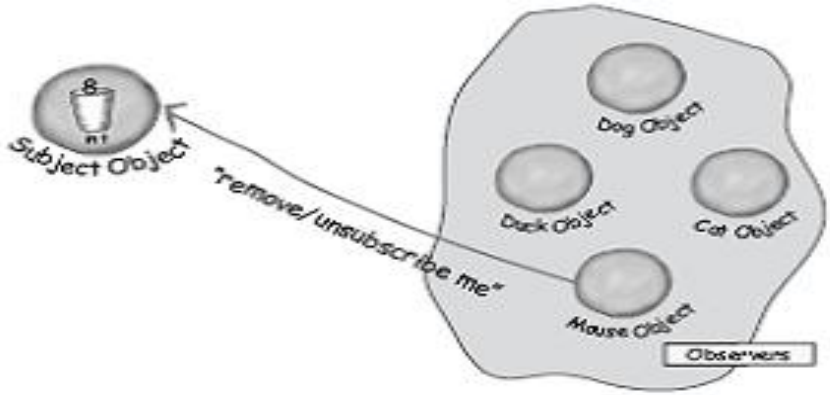
The Subject gets a new data value!

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.

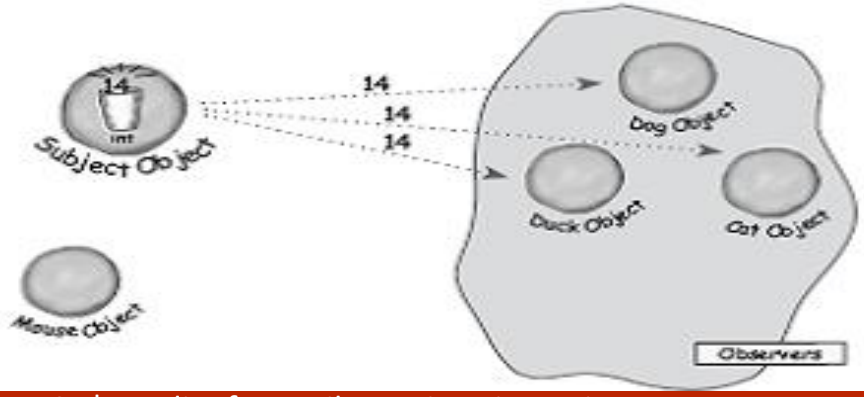


Mouse is outta here!

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.

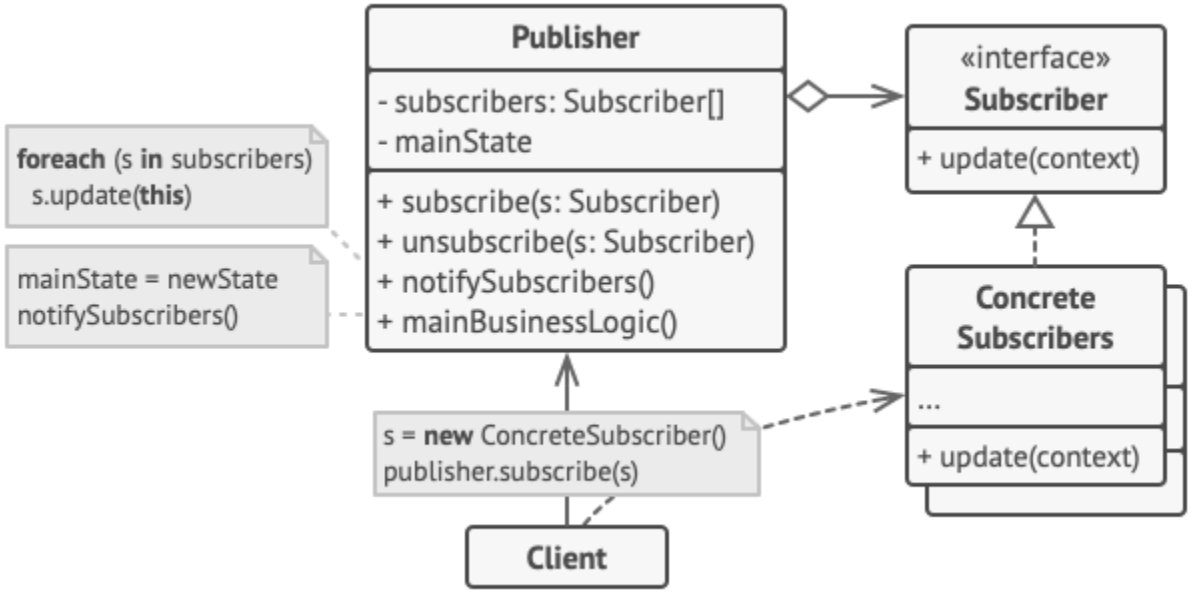


Penjelasan

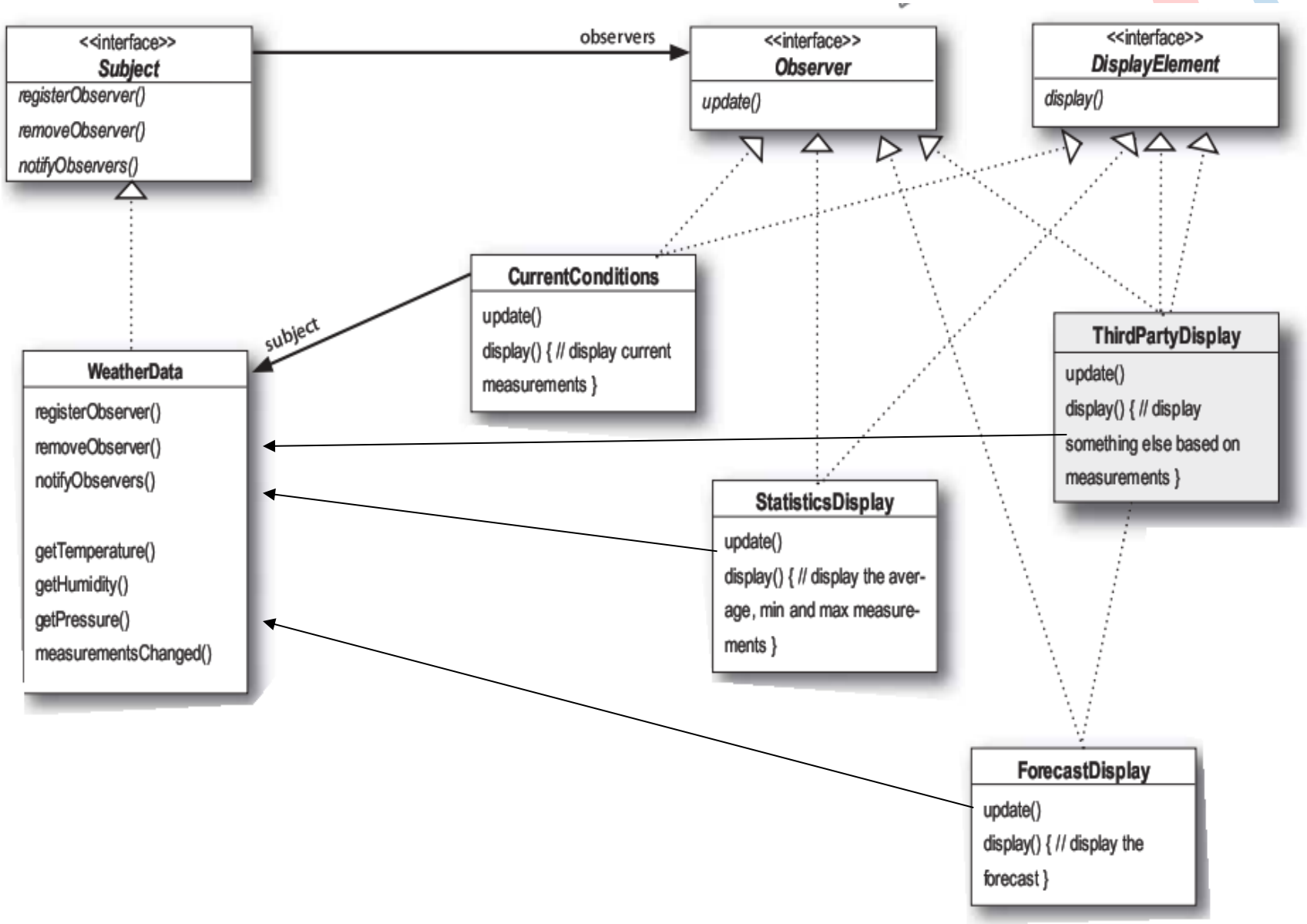
- **Subject Interface**: digunakan untuk mendaftarkan, menghapus, dan memberi tahu Observer
- Subject boleh memiliki **lebih dari satu** Observer
- **Observer Interface**: method update() digunakan jika state Subject **berubah**
- **ConcreteSubject**: implementasi real interface Subject
- **ConcreteObserver**: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

Observer Pattern

- **Observer Pattern** adalah salah satu pola desain perilaku (behavioral design pattern) yang memungkinkan sebuah objek (subject) secara otomatis memberi tahu objek-objek lainnya (observers) saat terjadi perubahan keadaan (state)



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 Observer GUI

Dalam Kasus Aplikasi Mahasiswa

- Dalam konteks aplikasi pencatatan data mahasiswa, objek yang bertindak sebagai subject adalah model data mahasiswa, sementara observer adalah komponen yang mencatat aktivitas (logger)
- Ketika terjadi perubahan data, observer akan secara otomatis menerima notifikasi dan mencatat peristiwa tersebut tanpa perlu dipanggil secara eksplisit oleh bagian lain dalam sistem.

Kode Observer Mahasiswa

```
public interface MahasiswaObserver {
    void onMahasiswaChanged(String message);
}

public class ConsoleLogger implements MahasiswaObserver {
    @Override
    public void onMahasiswaChanged(String message) {
        System.out.println("[LOG] " + message);
    }
}
```

```
public class MahasiswaModel {
    private List<MahasiswaObserver> observers = new ArrayList<>();
    private List<Mahasiswa> dataMahasiswa = new ArrayList<>();

    public void addObserver(MahasiswaObserver observer) {
        observers.add(observer);
    }

    private void notifyObservers(String message) {
        for (MahasiswaObserver observer : observers) {
            observer.onMahasiswaChanged(message);
        }
    }
                                        public void addMahasiswa(Mahasiswa m) {
                                              dataMahasiswa.add(m);
                                              notifyObservers("Mahasiswa ditambahkan: " + m.getNama());
                                          }

                                          public void deleteMahasiswa(Mahasiswa m) {
                                              dataMahasiswa.remove(m);
                                              notifyObservers("Mahasiswa dihapus: " + m.getNama());
                                          }

                                          public void updateMahasiswa(Mahasiswa m) {
                                              notifyObservers("Mahasiswa diupdate: " + m.getNama());
                                          }
                                    }
```

Kode Observer Mahasiswa

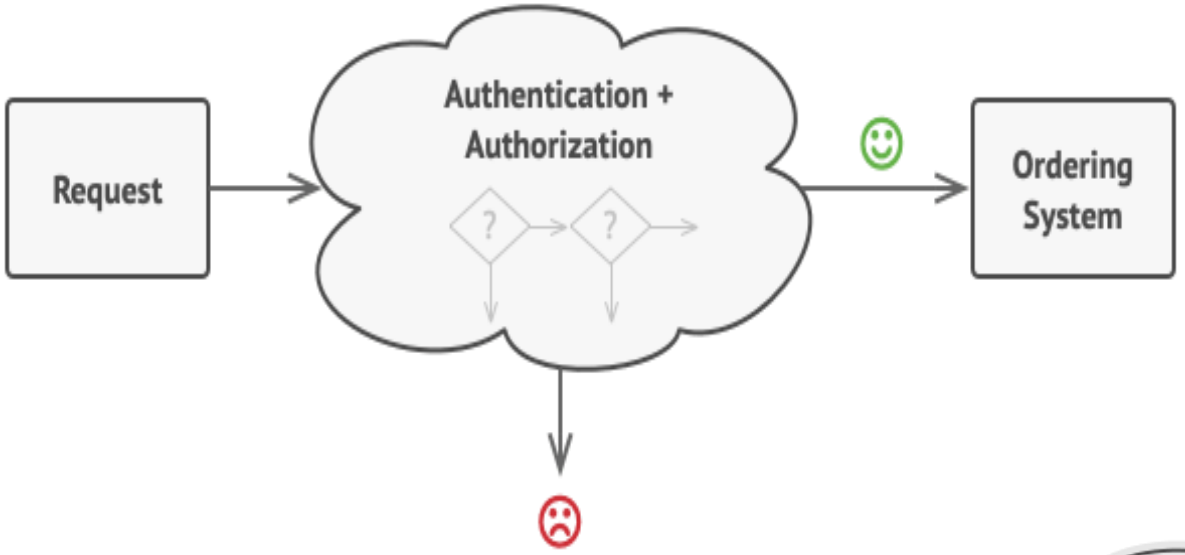
```
public class MainApp {
    public static void main(String[] args) {
        MahasiswaModel model = new MahasiswaModel();
        ConsoleLogger logger = new ConsoleLogger();

        model.addObserver(logger);  // Logger mendaftar ke model

        Mahasiswa mhs = new Mahasiswa("Budi", "12345", 3.5);
        model.addMahasiswa(mhs);  // Akan memicu pencatatan
  otomatis
    }
}
```

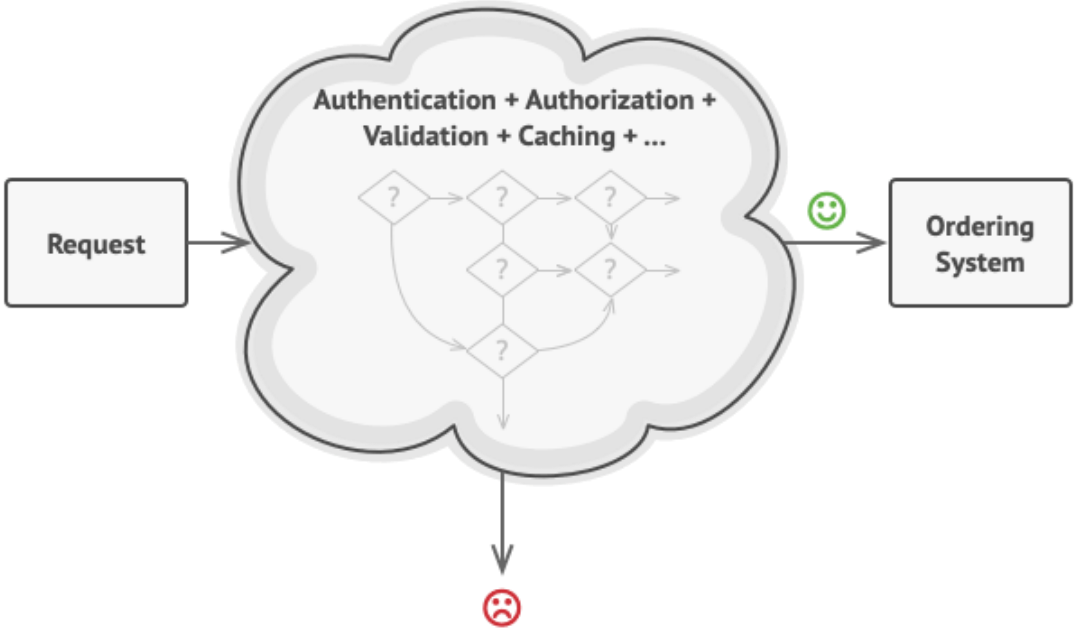
Chain of Resposibility

- **Chain of Responsibility** adalah pola desain perilaku yang memungkinkan sejumlah objek untuk **menangani permintaan secara berantai**
- Tujuan utama pola ini adalah untuk melewati permintaan di sepanjang rantai objek sampai salah satu dari mereka menangani permintaan itu



Ilustrasi bagaimana setiap permintaan (request) harus melewati serangkaian pemeriksaan sebelum diproses

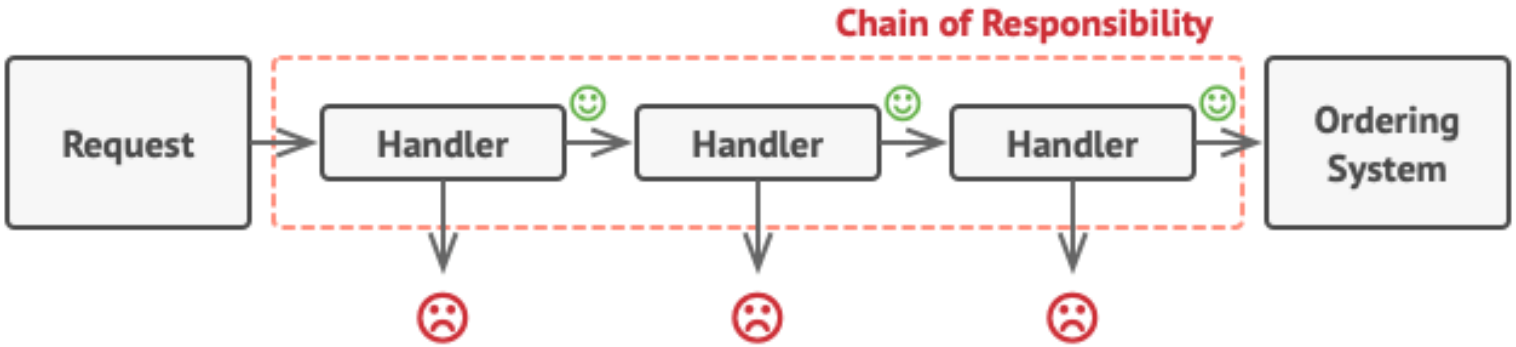
Mekanisme validasi bertambah seiring berkembangnya sistem, membuat kode program semakin kompleks



CoR

- **Chain of Responsibility (CoR)** Pattern merupakan salah satu pola desain perilaku (**behavioral design pattern**) yang dapat digunakan untuk menyusun **validasi** secara modular dan fleksibel
- Pola ini memungkinkan serangkaian objek (**handlers**) untuk memproses permintaan satu per satu dalam sebuah rantai, hingga permintaan tersebut diproses atau ditolak

Ilustrasi bagaimana Chain of Responsibility diterapkan pada Ordering System



Karakteristik CoR

- Setiap objek (handler) memiliki kesempatan untuk **menangani** permintaan **atau meneruskannya**.
- Digunakan untuk **menghindari** struktur if-else bertingkat yang kompleks.
- Cocok untuk validasi berlapis, middleware, atau filter.
- Komponen utama:
  - **Handler**: Interface/kelas dasar yang mendefinisikan setNext() dan handle().
  - **ConcreteHandler**: Implementasi logika pemeriksaan

Kelebihan dan Kekurangan CoR

- **Kelebihan:**
  - Memisahkan tanggung jawab ke dalam unit yang mandiri.
  - Mudah menambah, mengubah, atau menyusun ulang handler tanpa mengubah client.
- **Kekurangan:**
  - Tidak ada jaminan bahwa permintaan akan ditangani.
  - Sulit dilacak saat rantai terlalu panjang atau dinamis

CoR dalam Sistem Mahasiswa

- Pada aplikasi pencatatan data mahasiswa, validasi input merupakan aspek penting sebelum data dimasukkan ke dalam database.
  - Data seperti nama mahasiswa, NIM, dan IPK harus diperiksa terlebih dahulu agar sesuai dengan ketentuan yang berlaku
- Dalam ruang lingkup validasi data mahasiswa, setiap handler dalam rantai bertanggung jawab terhadap satu jenis validasi, seperti memeriksa apakah nama kosong, apakah NIM sudah digunakan, atau apakah IPK berada dalam batas yang diperbolehkan.
- Selama ini hal ini dilakukan menggunakan if the else pada controller

Implementasi InputValidator

```
public abstract class InputValidator {
    protected InputValidator next;

    public InputValidator setNext(InputValidator nextValidator) {
        this.next = nextValidator;
        return nextValidator;
    }

    public boolean validate(Mahasiswa mhs) {
        if (!doValidate(mhs)) return false;
        return next == null || next.validate(mhs);
    }

    protected abstract boolean doValidate(Mahasiswa mhs);
}
```

```
public class EmptyFieldValidator extends InputValidator {
    @Override
    protected boolean doValidate(Mahasiswa mhs) {
        if (mhs.getNama().isEmpty() || mhs.getNim().isEmpty()) {
            System.out.println("Validasi gagal: Nama atau NIM
 tidak boleh kosong.");
            return false;
        }
        return true;
    }
}
```

```
public class IPKRangeValidator extends InputValidator
{
    @Override
    protected boolean doValidate(Mahasiswa mhs) {
        double ipk = mhs.getIpk();
        if (ipk < 0.0 || ipk > 4.0) {
            System.out.println("Validasi gagal: IPK
harus antara 0.0 dan 4.0");
            return false;
        }
        return true;
    }
}
```

InputValidator

```
public class NIMUniqueValidator extends InputValidator {
    private List<String> existingNIMs;

    public NIMUniqueValidator(List<String> existingNIMs) {
        this.existingNIMs = existingNIMs;
    }

    @Override
    protected boolean doValidate(Mahasiswa mhs) {
        if (existingNIMs.contains(mhs.getNim())) {
            System.out.println("Validasi gagal: NIM sudah
  terdaftar.");
            return false;
        }
        return true;
    }
}
```

MahasiswaController

```
public class MahasiswaController {
    private MahasiswaModel model;

    public MahasiswaController(MahasiswaModel model) {
        this.model = model;
    }

    public void prosesTambahMahasiswa(String nama, String nim, double ipk) {
        Mahasiswa mhs = new Mahasiswa(nama, nim, ipk);

        List<String> nimSudahAda = model.getDaftarNIM();  // ambil dari database

        InputValidator validator = new EmptyFieldValidator();
        validator.setNext(new NIMUniqueValidator(nimSudahAda))
                 .setNext(new IPKRangeValidator());

        if (validator.validate(mhs)) {
            model.addMahasiswa(mhs);
            System.out.println("Mahasiswa berhasil ditambahkan.");
        } else {
            System.out.println("Mahasiswa gagal ditambahkan.");
        }
    }
}
```

MahasiswaModel

```
public class MahasiswaModel {
    private List<Mahasiswa> data = new ArrayList<>();

    public List<String> getDaftarNIM() {
        return data.stream().map(Mahasiswa::getNim).toList();
    }

    public void addMahasiswa(Mahasiswa mhs) {
        data.add(mhs);
        // Proses insert ke SQLite di sini
    }
}
```

Implementasi

DB: Tabel Log

	id	event	timestamp
1	1	Mahasiswa diubah:…	2025-05-26 01:30:…
2	2	Mahasiswa ditamba…	2025-05-26 08:31:…

```
create table log
(
   id        INTEGER
      primary key autoincrement,
   event     TEXT not null,
   timestamp DATETIME default
CURRENT_TIMESTAMP
);
```

Manager: MahasiswaLogManager

```
public class MahasiswaLogManager implements MahasiswaObserver {
    LogRepository logRepository;  3 usages

    public MahasiswaLogManager() { this.logRepository = new LogRepository(DBConnectionMana

    @Override  1 usage
    public void onMahasiswaChanged(String message) { this.logRepository.save(new LogEvent(


    public List<LogEvent> getAllLogs() { return this.logRepository.findAll(); }
}
```

Manager: ManagerMahasiswa

```
// CREATE: Menambahkan Mahasiswa
public boolean tambahMahasiswa(Mahasiswa mahasiswa) {  1 usage
    if (mahasiswaRepository.save(mahasiswa)) {
        notifyObservers("Mahasiswa ditambahkan: " + mahasiswa.getNama() + " (NIM: " +
        return true;
    } else {
        return false;
    }
}

public ArrayList<Mahasiswa> getAllMahasiswa() { return (ArrayList<Mahasiswa>) mahasis

// UPDATE: Memperbarui Data Mahasiswa
public boolean updateMahasiswa(Mahasiswa mahasiswa) {  1 usage
    if (mahasiswaRepository.update(mahasiswa)) {
        notifyObservers("Mahasiswa diubah: " + mahasiswa.getNama() + " (NIM: " + maha
        return true;
    } else {

        return false;
    }
```

Model: LogEvent

```java
public class LogEvent {                                                    ⚠ 3
    private int id;  3 usages
    private String event;  5 usages
    private LocalDateTime timestamp;  5 usages

    // Konstruktor tanpa parameter (penting untuk ORM/framework tertentu)
    public LogEvent() {}

    // Konstruktor lengkap
    public LogEvent(int id, String event, LocalDateTime timestamp) {  no usages
        this.id = id;
        this.event = event;
        this.timestamp = timestamp;
    }

    // Konstruktor tanpa ID (untuk insert baru)
    public LogEvent(String event) {  1 usage
        this.event = event;
        this.timestamp = LocalDateTime.now(); // default waktu sekarang
    }

    // Getter dan Setter
    public int getId() { return id; }

    public void setId(int id) { this.id = id; }
```

InputMahasiswaValidator

```
public abstract class InputMahasiswaValidator {
    protected InputMahasiswaValidator next;
    protected String errorMessage;

    public InputMahasiswaValidator setNext(InputMahasiswaValidator nextValidator) {
        this.next = nextValidator;
        return nextValidator;
    }

    public boolean validate(Mahasiswa mhs) {
        if (!doValidate(mhs)) return false;
        return next == null || next.validate(mhs);
    }

    protected abstract boolean doValidate(Mahasiswa mhs);
```

```
public String getLastErrorMessage() {
        if (this.errorMessage != null) {
            return this.errorMessage;
        } else if (this.next != null) {
            return
this.next.getLastErrorMessage();
        }
        return null;
    }
}
```

CoR : EmptyField dan IPKRangeValidator

```java
public class EmptyFieldMahasiswaValidator extends InputMahasiswaValidator {  3 usages
    @Override  1 usage
    protected boolean doValidate(Mahasiswa mhs) {
        if (mhs.getNama().isEmpty() || mhs.getNim().isEmpty()) {
            errorMessage = "Validasi gagal: Nama atau NIM tidak boleh kosong.";
            return false;
        }
        return true;
    }
}
```

```java
public class IPKRangeMahasiswaValidator extends InputMahasiswaValidator {  3 usages
    @Override  1 usage
    protected boolean doValidate(Mahasiswa mhs) {
        double nilai = mhs.getNilai();
        if (nilai < 0.0 || nilai > 4.0) {
            errorMessage = "Validasi gagal: IPK harus antara 0.0 hingga 4.0.";
            return false;
        }
        return true;
    }
}
```

CoR: UniqueMahasiswaValidator

```java
public class NIMUniqueMahasiswaValidator extends InputMahasiswaValidator {  2 usages
    private final List<String> existingNims;  2 usages

    public NIMUniqueMahasiswaValidator(List<String> existingNims) { this.existingNim

    @Override  1 usage
    protected boolean doValidate(Mahasiswa mhs) {
        if (existingNims.contains(mhs.getNim())) {
            errorMessage = "Validasi gagal: NIM sudah digunakan.";
            return false;
        }
        return true;
    }
}
```

Next

- MVC dan MVP Pattern