IN1010 Gruppetime – Uke 11

Gruppe 7

Dagens tema: Designmønstre

Designmønstre

- Designmønstre er gjenbrukbare måter å strukturere programmer på
- Forskjellige mønstre har ulike ulemper/fordeler, basert på hva vi ønsker å oppnå i programmet.
- Ikke no generell fasit på hvordan disse mønstrene implementeres, idéene er mer abstrakte

Iterator-mønsteret

- Allerede kjent for dere
- Iterator-mønsteret kan oppnås i Java med grensesnittene Iterable<> og Iterator<>
- Nyttig når vi skal implementere lenkelister

Observer-mønsteret

- Observer-mønsteret handler om å ha to *typer* klasser, én kalt en Observatør og den andre kalt Subjekt
- Observatøren observerer Subjektet, og får beskjed om endringer i Subjektet.
  - Når en endring skjer i Subjektet, kan Observatøren også gjøre en endring
  - Slik oppnår man en slags sammenkobling mellom objektene
- Eksempel: MVC
  - View er observatør, Model er subjekt
  - Når en endring skjer i Model, så får View beskjed og kan oppdatere brukergrensesnittet for å reflektere endringen
  - Mer om dette i temaet GUI

Decorator-mønsteret

- Decorator-mønsteret handler om å kunne legge til, endre eller fjerne egenskaper til et objekt under kjøring
- Dette kan gjøres med egne klasser vi kan implementere, som tar inn et objekt og *dekorerer* objektet
  - Disse klassene er slags *wrappers* for objektet, men med lik funksjonalitet
- Et alternativ til vanlig arv

```java
// Interface for the main product - ConcertTicket
interface ConcertTicket {
    String getDescription();
    double getCost();
}


// A ConcertTicket implementation
class BasicConcertTicket implements ConcertTicket {
    @Override
    public String getDescription() {
        return "Concert ticket";
    }


    @Override
    public double getCost() {
        return 450.0;
    }
}
```

```java
// The ConcertTicket decorator
abstract class ConcertTicketDecorator implements ConcertTicket {
    private final ConcertTicket decoratedTicket;


    public ConcertTicketDecorator(ConcertTicket decoratedTicket){
        this.decoratedTicket = decoratedTicket;
    }


    public String getDescription() {
        return decoratedTicket.getDescription();
    }


    public double getCost() {
        return decoratedTicket.getCost();
    }
}
```

```java
// A specific ConcertTicket decorator - VIPSpot
class VIPSpot extends ConcertTicketDecorator {
    public VIPSpot(ConcertTicket decoratedTicket) {
        super(decoratedTicket);
    }


    @Override
    public String getDescription() {
        return super.getDescription() + ", VIP spot";
    }


    @Override
    public double getCost() {
        return super.getCost() + 300.0;
    }
}
```

Singleton-mønsteret

- Singleton-mønsteret handler om at det kun skal være én instans av et objekt i hele programmet.
- Hvis et objekt er veldig ressurskrevende, så kan det være lurt å begrense det til én instans
- Medfører at det blir en global tilstand i programmet
  - Kan gjøre at graden av kobling i programmet økes
- Ofte en forutsetning med trådsikkerhet i Singleton-mønsteret
  - Må låse når instansen opprettes, ellers risikerer vi at trådene oppretter instansen samtidig, og lager flere instanser

```java
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public final class LoggerSingleton {
    // Single instance
    private static volatile LoggerSingleton instance = null;
    private static Lock lock = new ReentrantLock();

    // Prevent instantiation from other classes by making constructor private
    private LoggerSingleton() {}


    // Method to get the single instance
    public static LoggerSingleton getInstance() {
        if (instance == null) {
            lock.lock();
            try {
                instance = new LoggerSingleton();
            } finally {
                lock.unlock();
            }
        }
        return instance;
    }
    // Method to demonstrate Singleton usage logging messages to the terminal
    public void logMessage(String message) {
        System.out.println("Log Message: " + message);
    }
}
```

Livekode

Platesjappa