# Objektorienterad modellering och diskreta strukturer / design

## Designmönster och fallstudier

Lennart Andersson

Reviderad 2012–09–14

2012



## Integritetsprincipen

**Integritetsprincipen**

Gör attribut, metoder och klasser så hemliga de går.

Lämna inte ut representationen i onödan.

En klass skall inte tillgång till klasser som den inte behöver.

Begränsa synligheten genom att använda

- `private` bara klassen kan se.
- paketsynlighet, klasser i paketet kan se
- `protected` subklasser och klasser i paketet kan se.
- `public` hela världen kan se.
- [`object` finns inte i Java.]



## Exponera inte representationen

Exponerad:

```
public class Figure extends ArrayList<Shape> {
  public void paint(Graphics graphics) {
    for (Shape shape : this) {
      shape.paint(graphics);
    }
  }
}
```

Ej exponerad:

```
public class Figure {
  private List<Shape> list = new ArrayList<Shape>();
  public void paint(Graphics graphics) {
    for (Shape shape : list) {
      shape.paint(graphics);
    }
  }
}
```



## Exponera inte representationen

Helt utelämnad representation:

```
public class Figure {
  private List<Shape> list;
  public void paint(Graphics graphics) {
    for (Shape shape : list) {
      shape.paint(graphics);
    }
  }
  public List<Shape> list() {
    return list;
  }
}
```

## Förbud (nästan)

- `instanceof`
- `static`
- getters



## Men det finns undantag

Det finns tillfällen när `instanceof`, `static` och getters är nödvändiga och det enda rätta: ...



## instanceof

```
public final class Integer extends Number
                implements Comparable<Integer> {
  private final int value;
  public boolean equals(Object object) {
    if(object instanceof Integer) {
      Integer other = (Integer) object;
      return value == other.value;
    }
    return false;
  }
  // omissions
}
```

Här finns inget bättre sätt.



## static

```
public static main(String arg[])

public final static double PI

public static double sin(double a)

public class Outer {
  private static class Inner {

  }
}
```

Språkets design kräver `static main`.

Ibland finns det inget tillhörande objekt.

Inre klasser bör helst vara `static`.

## getters

Det finns ett exempel i Computer-projektet där en getter ger en bättre helhetsdesign. Observer-mönstret behöver i regel getters.

Ibland måste man kompromissa när principerna drar åt olika håll.



## Spekulativ design, Martin p. 105

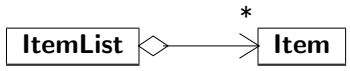
- Fool me once — shame on you
- Fool me twice — shame on me.
- Skriv program som om förutsättningarna inte kommer att förändras. Om detta ändå sker så implementera abstraktioner som skyddar mot framtida förändringar av samma slag.

- Take the first bullet —
- and protect yourself from the second bullet from the same gun.



## Exempel

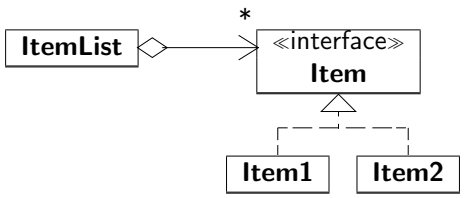
Uppdragsgivaren vill ha en lista av `Item`-objekt:






## Exempel

Uppdragsgivaren vill senare ha en lista som innehåller `Item1`-objekt och `Item2`-objekt. Detta är första skottet; nu garderar vi oss mot ett liknande skott:



## Uttryck med variabler

Vi har modellerat aritmetiska uttryck utan variabler:

$$1 + 2 * 3$$

Nu vill uppdragsgivaren ha aritmetiska uttryck med variabler.

$$1 + x + x * y$$

Vi lägger till en klass för att representera en variabel:

```
public class Variable implements Expr {
  private String name;

  public int value() {
    return ??;
  }
}
```



## Var skall variablernas värden finnas?

I variabeln eller i "minnet"?

```
public class Variable implements Expr {
  private String name;

  public int value(Map<String, Integer> map) {
    return map.get(name);
  }
}

public interface Expr {
    public int value(Map<String, int> map);
}
```



## Uttryck med variabler

```
public class Add implements Expr {
  private Expr epr1, expr2;

  public int value(Map<String, int> map) {
    return expr1.value(map) + expr2.value(map);
  }
}
```

Klassen `Num` behöver inte `map`, men det skulle komplicera designen onödigt mycket om vi försöker förhindra det.



## Muterbara objekt

Ett objekt vars tillstånd kan förändras kallas *muterbart* (mutable).

När objektet modellerar någonting i verkligheten som har ett tillstånd som kan förändras så är det rimligt att modellen har samma egenskap.

## Ej muterbara objekt

Ett objekt vars tillstånd inte kan förändras kallas *omuterbart* (immutable).

När ett objektet modellerar någonting i verkligheten vars tillstånd inte kan förändras så är det rimligt att modellen har samma egenskap.

Integer-objekt och String-objekt är omuterbara.



## Variable

En omuterbar klass:

```
public final class Num implements Expr {
  private int value;
  public value() {
    return value;
  }
}
```

En muterbar klass:

```
public class Counter {
  private int counter;
  public increment() {
    counter++;
  }
}
```



## Recept för omuterbarhet

Klassen måste vara final.

Attributen måste vara privata.

Attributen får ej vara muterbara.

Inga metoder får förända attribut.



## Fördelar med omuterbara objekt

Säkrare

Behöver ej kopieras, kan delas

Enklare att resonera om

## Aktivitet

```
public final class SafeContainer {
  private final Object object;
  public SafeContainer(Object object) {
    this.object = object;
  }
  public String toString() {
    return object.toString();
  }
}
```

Är klassen muterbar?



## Payroll: ... Systembeskrivning

- Vissa anställda tillhör fackföreningen. Deras anställningskort innehåller veckoavgift. Föreningen kan debitera serviceavgifter för enskilda medlemmar. Avgifter dras från nästa lön.
- Transaktioner till systemet behandlas en gång om dagen och uppdaterar en databas. Löneutbetalningar genereras varje avlöningsdag.



## Payroll: Systembeskrivning ...

### Martin: Fallstudien Payroll

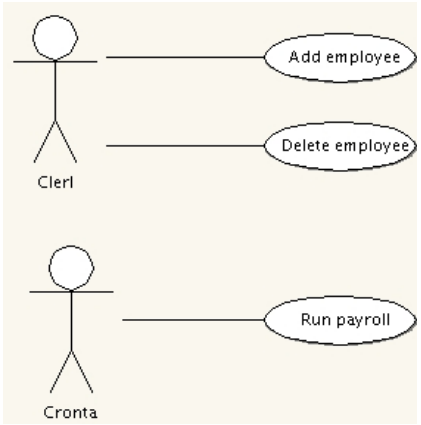
- Vissa anställda arbetar på timbasis. Deras timlön finns i anställningsposten. De lämnar in tidkort med datum och antal timmar. Lönen utbetalas fredagar.
- Vissa anställda har fast lön som utbetalas sista vardagen varje månad.
- Vissa anställda med fast lön får också provision baserad på kvitton med datum och belopp med löneutbetalning varannan fredag.
- Lön utbetalas antingen via postanvisning till angiven adress, direkt till konto på bank, eller avhämtas på lönekontoret.



## Användningsfall

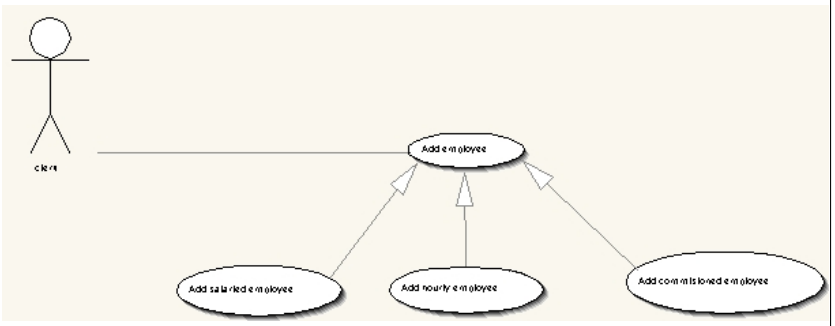
1. Lägg till en ny anställd.
2. Ta bort en anställd
3. Registrera ett tidkort
4. Registrera ett försäljningskvitto
5. Registrera en föreningsavgift
6. Ändra anställningsinformation
7. Generera löneutbetalningar

## Use case diagram



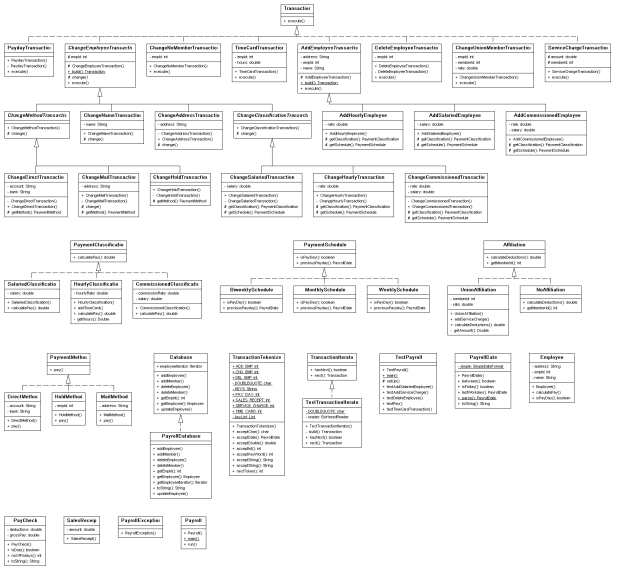



## Arv i användningsfall






## Oanvändbart klassdiagram





## Substantiv

| | | | |
|---|---|---|---|
| anställd | timbasis | timlön | anställningspost |
| tidkort | lön | fredag | vardag |
| vecka | månad | provision | kvitto |
| datum | belopp | utbetalning | postanvisning |
| adress | konto | bank | lönekontoret |
| fackförening | veckoavgift | anställningskort | serviceavgift |
| medlem | avgift | transaktion | system |
| databas | avlöningsdag | | |

Vilka substantiv är centrala?

## Payroll

- Övergripande design – kapitel 18 i Martin
- Implementering – kapitel 19 i Martin med C++.
- Implementering med Java i Payroll.zip via föreläsningssidan.



## Databasen enligt Martin 1

- Databasen är en implementeringsdetalj.
- Moduler skall bero på abstraktioner (DIP).
- Definiera ett gränssnitt (API) !



## Databas – API

```
public interface Database {
    public Employee put(int key, Employee employee);
    public Employee get(int key);
    public Employee remove(int key);
}
```



## Databas – Implementering för testning

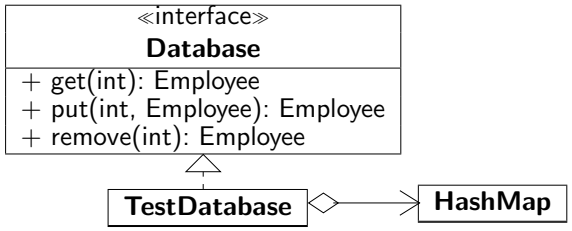
```
public class TestDatabase implements Database {
    private Map<Integer, Employee>
            employees = new HashMap<Integer, Employee>();

    public Employee get(int empId) {
        return employees.get(empId);
    }

    public Employee remove(int empId) {
        return employees.remove(empId);
    }

    public Employee put(int empId, Employee employee) {
        return employees.put(empId, employee);
    }
}
```

## Designmönster – Façade






## Databasen enligt Martin 2

- Databasen används överallt. Den måste vara lätt åtkomlig.
- Det skall bara finnas en databas.

Slutsats: Använd Singleton!



## Databasen enligt Andersson

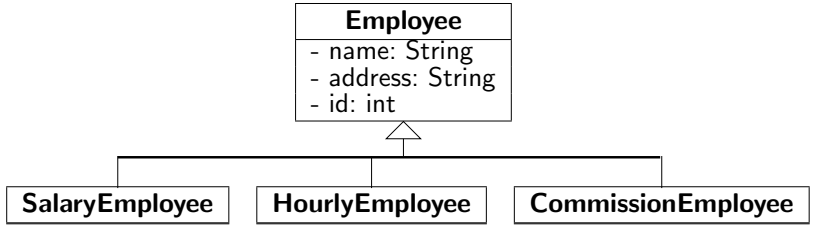
- Databasen används bara i en metod. Det är lätt att ge metoden tillgång till databasen.
- Det behövs olika databaser när man testar och använder i verkligheten.

Slutsats: Använd inte Singleton!



## Det finns tre sorters anställda

Design utan eftertanke:



## Användningfall: en timanställd blir löneanställd

Vad är problemet?

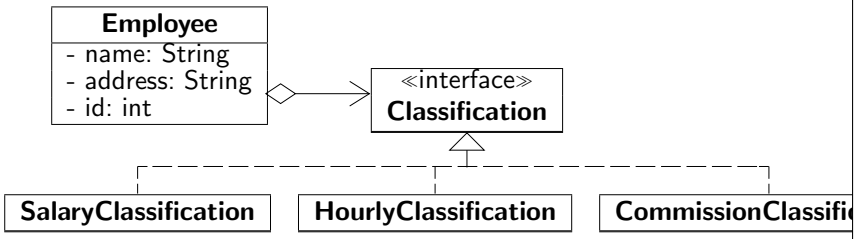
Om en timanställd blir löneanställd så måste man skapa ett nytt objekt och kopiera data från det gamla.

- Bättre design?
- Designmönster?



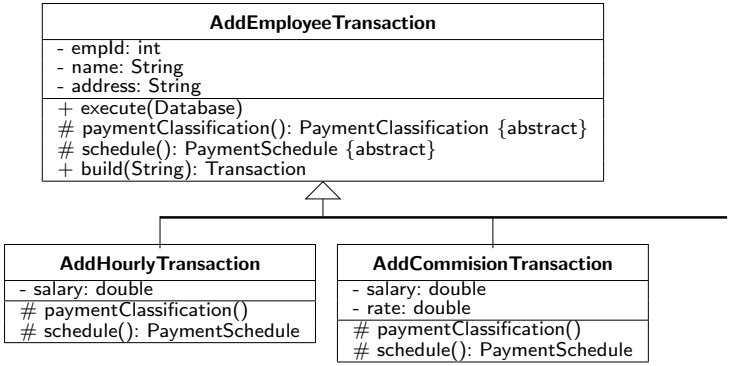
## Strategy

Anställningsformen är en strategi.






## AddEmployee – Template method






## Employee

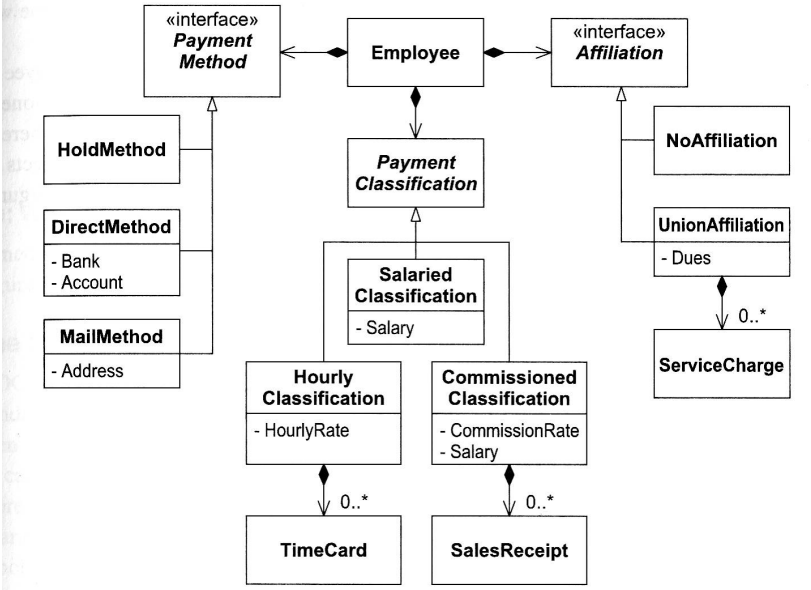
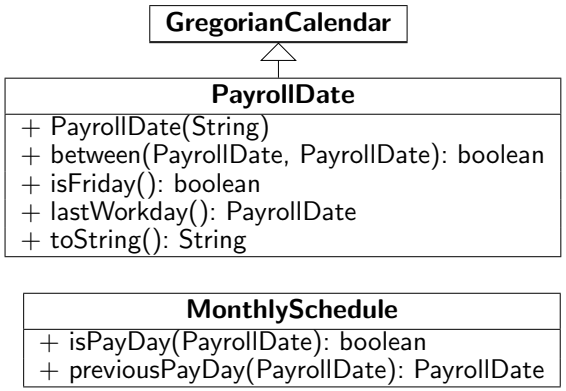



**Figure 18.6** Revised Class Diagram for Payroll

## PayrollDate






## *Decorator*-mönstret

Skånetrafiken vill få en lista på alla köp av biljetter under oktober.
Standardautomaten:

```
public interface PaySation {
  public double pay(TravelCard travelCard, int zones);
}

public StandardPayStation implements PayStation {
  private double sum;
  public double pay(TravelCard travelCard, int zones) {
    double price = travelCard.price(zones);
    sum += price;
    return return price;
  }
}
```



## *Decorator*-mönstret

Den nya funktionaliteten:

```
public LoggingPayStation implements PayStation {
  private PayStation payStation;
  private List<String> log;
  public LoggingPayStation(PayStation payStation, List<String> log) {
    this.payStation = payStation;
    this.log = log;
  }
  public double pay(TravelCard travelCard, int zones) {
    log.add(travelCard.toString() + zones);
    return payStation.pay(travelCard, zones);
  }
}
```



## Loggningen kan göras dynamiskt

```
PayStation standardPayStation = new StandardPayStation();
PayStation payStation = standardPayStation;
```

I början av oktober lägger vi till loggningen:

```
payStation = new LoggingPayStation(standardPayStation);
```

I slutet av oktober tar vi bort den:

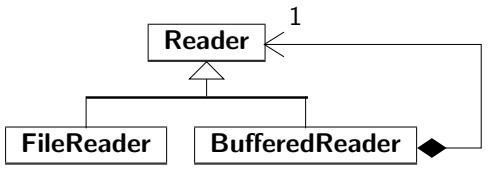
```
payStation = standardPaystation;
```

Tillståndet i standardPayStation bevaras.

## *Decorator*-mönstret i Reader-klasserna

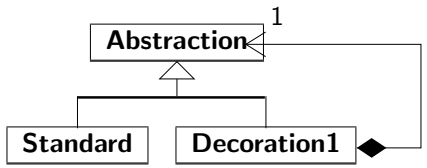
```
reader = new BufferedReader(new FileReader(fileName));
```

```
public class BufferedReader extends Reader {
    private Reader in;
    // omissions
}
```






## Decorator






## Aktivitet 23

```
List<String> list = new ArrayList<String>();
SafeContainer safeContainer = new SafeContainer(list);
list.add("string");
```

Klassen Unkown är muterbar.