Objektorienterad modellering och diskreta strukturer / design

Designmönster och fallstudier

Lennart Andersson

Reviderad 2011–09–08

2011

Misstaget som kostat 1 000 000 000 $

The billion dollar mistake

Den skyldige: Tony Hoare

Tony Hoare

Tony Hoare, Inventor of QuickSort, Turing Award Winner

Sir Charles Antony Richard Hoare (Tony Hoare or C.A.R. Hoare, born January 11, 1934) is a British computer scientist, probably best known for the development in 1960 of Quicksort (or Hoaresort), one of the world's most widely used sorting algorithms.

He also developed Hoare logic for verifying program correctness, and the formal language Communicating Sequential Processes (CSP) used to specify the interactions of concurrent processes (including the Dining philosophers problem) and the inspiration for the Occam programming language.

Abstract:

Presentation: "Null References: The Billion Dollar Mistake"

Track: Historically bad ideas

Time: Friday 13:00 - 14:00

Location: Abbey Room

Abstract: I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years. In recent years, a number of program analysers like PREfix and PREfast in Microsoft have been used to check references, and give warnings if there is a risk they may be non-null. More recent programming languages like Spec# have introduced declarations for non-null references. This is the solution, which I rejected in 1965.

Tony Hoare om Algol:

Here is a language so far ahead of its time that it was not only an improvement on its predecessors but also on nearly all its successors.

Null Object

- Introduktionen av `null` i objektorienterade språk var ett fundamentalt misstag.
- `null` modellerar något som inte finns. Är inte det befängt?
- Länkade listor och träd avslutas ofta med `null`.
- `null` är inget objekt; att använda `null` för att representera "ingenting" är ett exempel på icke-objektorienterad programmering.

Konventionell länkad lista

```
public class Node {
    private Object object;
    private Node next;
}

public class List {
    private Node first;
    public int length() {
        int length = 0;
        Node node = first;
        while (node != null) {
            length++;
            node = node.next();
        return length;
    }
}
```

Null object

- Detta är en maskerad form av `instanceof`.
- next() är en getter.
- Node saknar intelligens.
- Programkoden blir enklare om man istället använder ett riktigt objekt. Detta är ett exempel på mönstret *Null Object*.

Objektorienterad beskrivning

En *lista* är antingen *tom* eller består av en *nod* med ett *element* som är ett Object och en *svans* som är en *lista*.

Objektorienterad representation

```
public interface List {
    public int length();
}
public class Empty implements List {
    public  int length() {
        return 0;
    }
}

public class Node implements List{
    private Object object;
    private List tail;
    public  int length() {
        return 1 + tail.length();
    }
}
```


Objektorienterad listimplementering

- Man ersätter iteration med rekursion; det gör programmet logiskt enklare; testet av det logiska villkoret i `while`-satsen försvinner.
- `ArrayList` är nästan alltid bättre än länkade listor.
- I nästan alla andra sammanhang är det bra design att representera något som är tomt med ett riktigt objekt.

Förbud (nästan)

- instanceof
- static
- getters

Men ...

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
}
```


static

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

public final static double PI

public static double sin(double a)

public class Outer {
  private static class Inner {

  }
}
```

getters

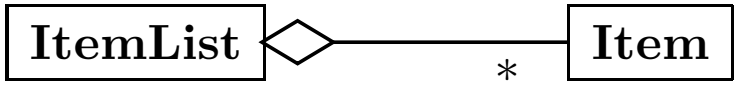
Ge mig ett bra exempel!

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.

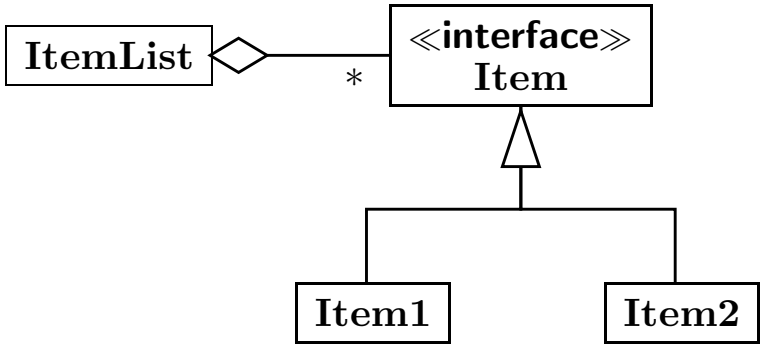
Exempel

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, int> 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);
  }
}
```


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.

Recept för ej muterbarhet

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

Variable

En klass som modellerar en matematisk heltalsvariabel:

```
public class Variable implements Expr {
  private int value;
  public setValue{int value} {
    this.value = value;
  }
}
```

Martin:

Payroll project

Payroll: Systembeskrivning ...

- 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.

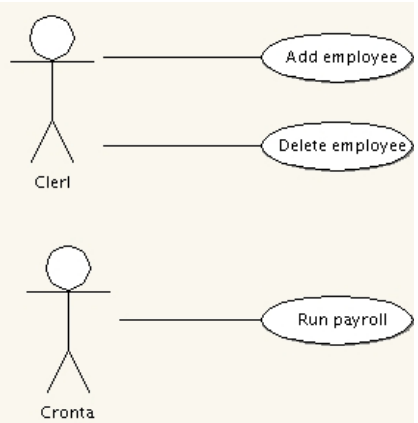
Payroll: ... Systembeskrivning

- 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.

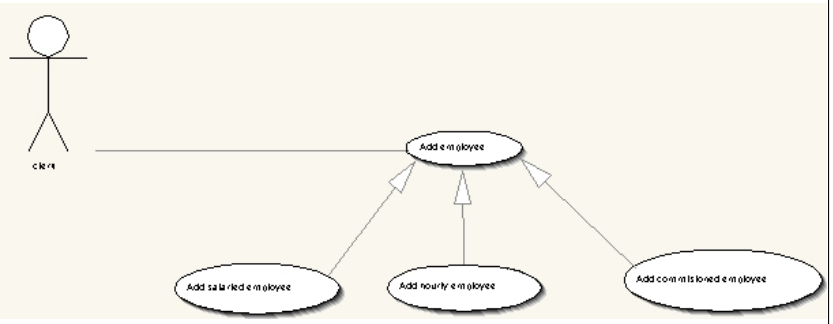
Användningsfall

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

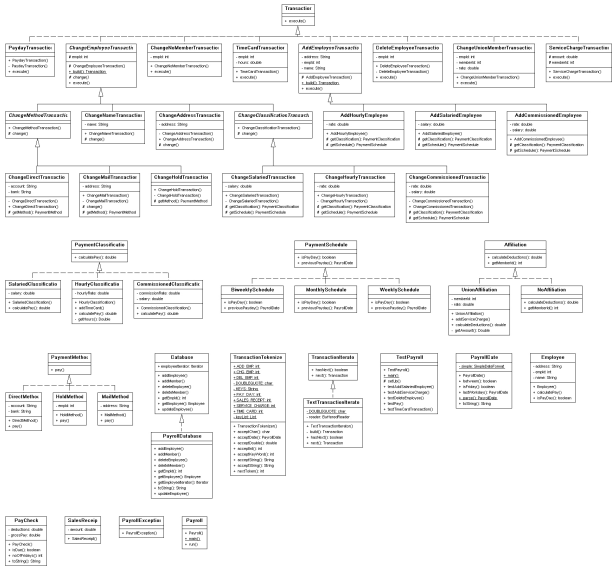
Use case diagram



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

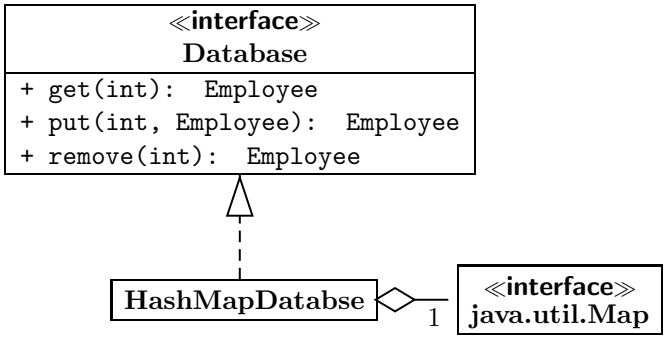
```
public class SimpleDatabase implements Database {
    private Map<Integer, Employee>
            employees = new HashMap<Integer, Employee>();

    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

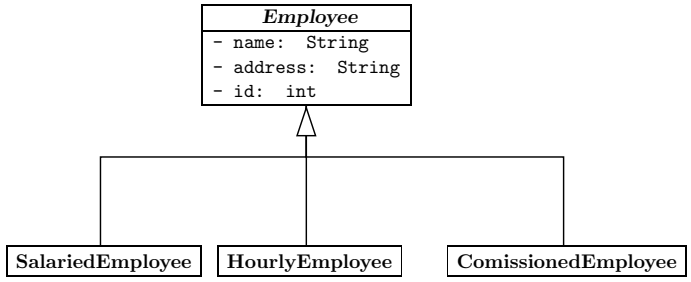
- Databasen används överallt. Den måste vara lätt åtkomlig.
- Det skall bara finnas en databas.
- Använd Singelton!

Databasen enligt Andersson

- Databasen används bara i en metod. Det är lätt att ge metoden tillgång till databasen.
- Det behövs en annan databas redan när man skall testa programmet.
- Använd inte Singelton!

Det finns tre sorters anställda

Design utan eftertanke:



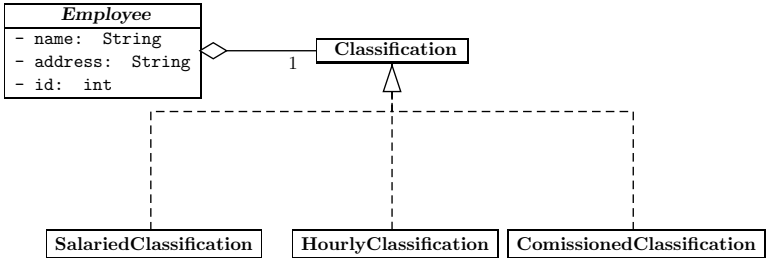
Användningfall: en timanställd blir löneanställd

Vad är problemet?

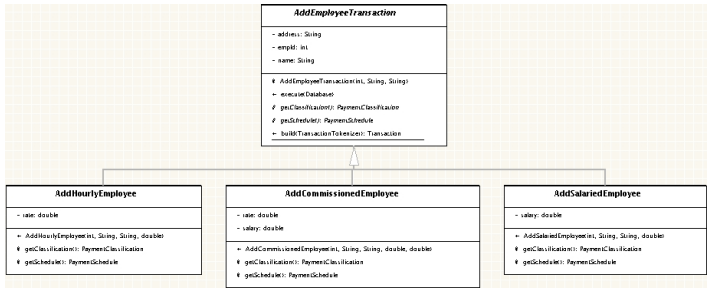
- Man måste skapa ett nytt objekt och kopiera data från det gamla.
- Bättre design?
- Designmönster?

Strategy

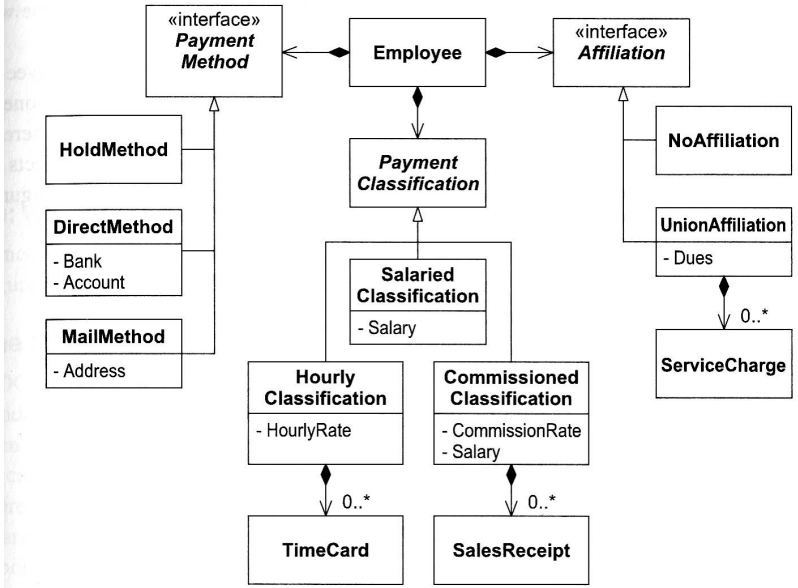
Anställningsformen är en strategi.



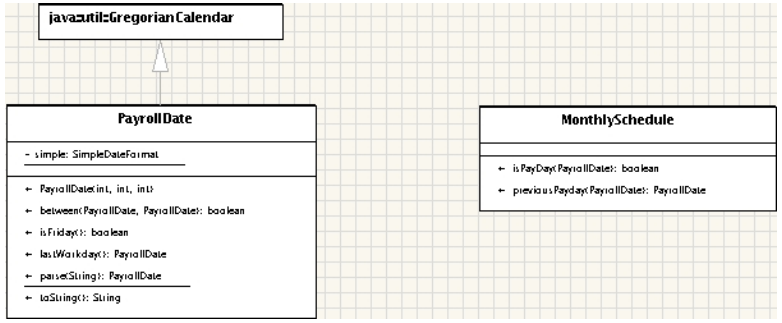
AddEmployee – Template method



Employee



PayrollDate



Klassdiagram: Associationer

—— ren association

⟶ riktad association

◇⟶ aggregering

◆⟶ komposition