Design patterns

Designmönster/Design patterns

- Vad är det?
  - Beprövade lösningar till återkommande programmeringsproblem
  - Plattformsoberoende
  - Beskrivs ofta med hjälp av UML
- Baseras på en bok
  - Design Patterns, Gamma/Helm/Johnson/Vlissides (Gang of Four)
- Kan/bör återanvändas
- Ca 25 st i orginalboken
- Ger även en terminologi med vilken man kan uttrycka designidéer på ett enklare sätt

Varför använda Design patterns?

- *"Designing object-oriented software is hard and designing reusable object-oriented software is even harder."* - Erich Gamma
- Beprövade lösningar sparar tid
- ”Man ska inte uppfinna hjulet igen”
- Gör det möjligt för oss att dra nytta av andra personers erfarenheter på ett snabbt sätt

Design patterns

- Om jag t.ex. gör ett program som löser problem X och sedan ska göra ett helt annat system där samma problem X uppstår hur gör jag då för att återanvända lösningen?
- Design patterns är just en hjälpmedel för detta.
- Tillåter samtidigt oss att modifiera lösningarna så att de passar ett speciellt problem.

Design patterns

- Ett design pattern:
  - Visar på hur man ska lösa ett problem rent allmänt (dvs inte i ett visst programspråk)
  - Diskuterar olika för och nackdelar lösningen har.
  - Hänvisningar till andra typer av lösningar på samma problem
  - Ger exempel på hur man kan göra och även på "riktiga" användningar av lösningen.

Klassifikation av design patterns

- Creational Patterns
  - Handlar om hur objekt skapas
- Structural Patterns
  - Handlar om hur klasser och objekt är strukturerade
- Behavioral Patterns
  - Handlar om hur klasser och objekt interagerar

Creational Patterns

- Abstract factory
  - Skapar objekt med okänd klass
- Builder
  - Skiljer skapandet från representationen
- Factory method
  - Sparar instantierandet till subklasser
- Prototype
  - Skapar nya objekt från en prototyp
- Singleton
  - Försäkran om enbart ett objekt av viss klass

Structural Patterns

- Adapter - Konverterar klassens gränssnitt
- Bridge - Skiljer abstraktion från impl
- Composite - Aggregat som träd
- Decorator - Lägger till egenskaper
- Facade - Ger enhetligt gränssnitt
- Flyweight - Hanterar många småobjekt
- Proxy - Surrogat för objekt (accesskontroll)

Behavioral Patterns:1

- Chain of responsibility - Överlämnar objekt
- Command - Kommando som objekt
- Interpreter - Språk ger grammatik&tolk
- Iterator - Accessar element sekventiellt
- Mediator - Kapslar in objektsamarbete
- Memento - Sparar objektets tillstånd

Behavioral Patterns:2

- Observer - Sprider ett objekts ändring till andra
- State - Tillståndsmaskin som byter klass
- Strategy - Kapslar in algoritmer
- Template method - Låter subklassen göra delar av algoritmen
- Visitor - Representerar en operation som ska utföras på en objektstruktur

Designprinciper

- ”Programmera mot ett interface, inte en implementation”
  - Handlar om beroenden mellan klasser
  - Lätt att lägga till beroenden, svårt att ta bort
- Avgränsa det som ändras
  - Det ska vara lätt att byta ut/lägga till delar som kan ändras
- Sträva mot ”lösa” beroenden mellan klasser
  - Oftast ska klasser veta så lite som möjligt om varandra
  - Synlighetsmodifierare

Observer/Observable Pattern

- **Basic Ideas:**
  1. The Observer pattern is used to keep consistency between related objects while minimizing their coupling and maximizing reusability (or independence) of the objects.
  2. Creates a one-to-many relationship between a subject and its observers. All observers are notified when there is a change to the subject. The observers then query the subject to synchronize themselves.

Observer/Observable

- **When to use this pattern:**
  1. When a change to one object requires the change of a variable number of other objects (not necessarily known at compile-time).
  2. When an object should be able to talk to another object, but you don't want them essentially dependent on each other.

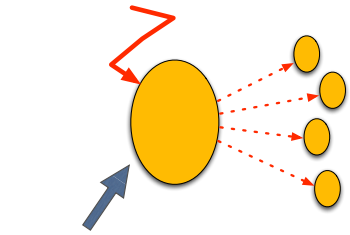
Observer/Observable

- **Benefits:**
  1. Can add new subjects and new observers all independently of all other subjects and observers.
  2. Minimal dependence between subject and observer
  3. Subject doesn't need to know how many observers it has. It simply broadcasts a change in state.

Observer/Observable

- **Pitfalls:**
  1. Because observers don't know about each other, a simple update to an observer might cause a long chain of other updates.
  2. Each observer decides whether it needs to update something when it receives the notification from the subject that something has changed. Therefore, complicated observers have to do a lot of work to figure out what changed when they receive a notification.

Observer - observable

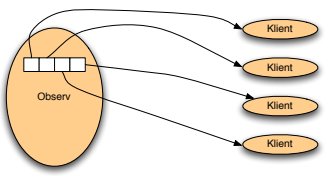


Ska dessa känna av om nåt har hänt?

Ska den veta om de andra objekten?

Observer - Observable

- Observ har en lista där intresserade klienter lagras
- Klienterna anmäler sitt intresse
- Om nåt händer så signalerar Observ till klienterna om att något har hänt



Java

- Basklass - Observable
- Interface - Observer

```
import java.util.Observable;

public class IntrusionDetector
   extends Observable
{
   public void someOneTriesToBreakIn()
   {
      // Vi låtsas att denna metod anropas när någon
      // försöker hacka sig in i datorn. Detta objekt
      // meddelar nu detta till alla som är intresserade
      setChanged();
      notifyObservers();
   }
}
```



```
import java.util.Observer;
import java.util.Observable;

public class SysAdm
   implements Observer
{
   private String name;

   public SysAdm( String name, IntrusionDetector id )
   {
      this.name = name;
      id.addObserver( this );
   }

   public void update( Observable server, Object err )
   {
      System.out.println( name + " förhindrar intrång" );
   }
}
```



```
public class Demo
{
   public static void main( String[] argv )
   {
      // Skapa ett objekt som kollar om någon försöker
      // hacka sig in i systemet
      IntrusionDetector id = new IntrusionDetector();

      // Skapa ett objekt som vill veta om någon
      // försöker hacka sig in. Ge det ett namn
      // och tala om vad det ska vara intresserad av

      SysAdm stric = new SysAdm( "stric", id );

      // Nu låtsas vi att någon försöker ta sig in
      id.someOneTriesToBreakIn();
   }
}
```


Singleton

- Singleton (Creational pattern)
  Se till att en klass bara har en instans, och ge en global access till den
  - T.ex. fönsterhanterare, filsystemhanterare
  - Statiska metoder (t.ex. java.lang.Math) är bättre för enskilda anrop

```
public class MySingletonClass {
  private static MySingletonClass instance
    = new MySingletonClass();

  public static MySingletonClass getInstance() {
      return instance;
  }

  /** There can be only one. */
  private MySingletonClass() {}
}
```


Pizzerian

Ett exempel på några design patterns

Factory Pattern

- Factory Pattern (Creational Pattern)
  Ge ett interface till att skapa relaterade objekt utan att specificera deras konkreta klasser
  - Klienten använder sig av objektet, men vet inte vilken konkret klass det bygger på, det är upp till fabriken att bestämma.
- Pizzerian:
  - Vi vill ha flera olika ugnar, utan att resten av programmet behöver bry sig om vilken som används

Factory Pattern

```
public interface PizzaOven{
    public Pizza bakePizza(Pizza myPizza);
}
```

```
public class StoneOven implements PizzaOven{
    public Pizza bakePizza(Pizza myPizza){
        //Baka på stenugnssätt..
    }
}
```

```
public class MetalOven implements PizzaOven{
    public Pizza bakePizza(Pizza myPizza){
        //Baka på metallugnssätt..
    }
}
```


Factory Pattern

```
public class PizzaOvenFactory{
    private String useOven = "stone";

    public static PizzaOven getOven(){
        if(useOven.equals("stone")
            return new StoneOven();
        else if(useOven.equals("metal")
            return new MetalOven();
    }
}
```

- Man kan också tänka sig att getOven() tar useOven strängen som argument. Vi har ändå fortfarande "skapandet" i fabriken.

Factory Pattern

- Hur ser klienten (användaren) ut?

```
.. //lite kod
Pizza myUnbakedPizza = new Pizza();
.. //lite kod (lägga på fyllning)

PizzaOven myOven = PizzaOvenFactory.getOven();
myOven.bakePizza(myUnbakedPizza)

.. //lite kod
```

- Vad har vi uppnått?
  - Programmerat mot ett interface
  - Lätt att lägga till/byta ut implementationen av ugnar utan att behöva ändra i klienten

Builder Pattern

- Vi har en bra struktur för pizzaugnen, men vi saknar något…
  - …PIZZA!
  - Vi vill kunna ha flera olika pizzor och alla är på olika sätt (olika såser, ingredienser, osv.)
- Builder Pattern (Creational Pattern)

–Separera konstruktionen av ett komplext objekt från dess representation

  - Till skillnad från factory pattern är vi alltså intresserade av hur ett objekt skapas, inte vilket objekt som skapas.


```
class Pizza {
   private String sauce = "";
   private String topping = "";

   public void setSauce(String sauce) { this.sauce = sauce; }
   public void setTopping(String topping) {
      this.topping = topping;
   }
}
```

```
abstract class PizzaBuilder { //abstrakt "builder"
   protected Pizza pizza;
   public Pizza getPizza() { return pizza; }
   public void createNewPizzaProduct() { pizza = new Pizza(); }
   public abstract void buildSauce();
   public abstract void buildTopping();
}
```



```
//Konkret builder för hawaii
class HawaiianPizzaBuilder extends PizzaBuilder {
    public void buildSauce() {
        pizza.setSauce("mild");
    }
    public void buildTopping() {
        pizza.setTopping("ham+pineapple");
    }
}
```

```
//Konkret builder för salamipizza
class SalamiPizzaBuilder extends PizzaBuilder {
    public void buildSauce() {
        pizza.setSauce("hot");
    }
    public void buildTopping() {
       pizza.setTopping("ham+salami+pepperoni");
    }
}
```

```
//Pizzakocken
class PizzaChef{
    public Pizza constructPizza(PizzaBuilder pb){
        pb.createNewPizzaProduct();
        pb.buildSauce();
        pb.buildTopping();
        return pb.getPizza();
}
```

```
//Ett exempel på beställning av pizza

PizzaChef chef = new PizzaChef();
HawaiiPizzaBuilder hawaiianBuilder = new HawaiiPizzaBuilder();
Pizza hawaii = chef.constructPizza(hawaiianBuilder);
... //lite mer kod
```

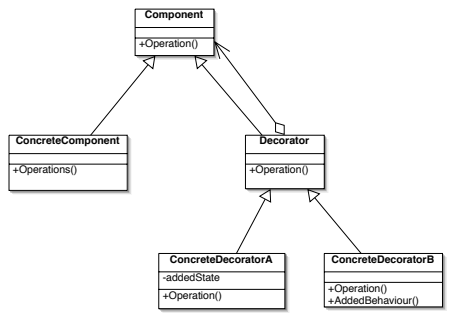

Builder Pattern

- Vad har vi uppnått?
  - Separerat objektet (Pizza) och dess uppbyggnad (konkreta Builders)
  - Lätt att lägga till/byta ut konkreta builders utan att ändra i objektet (Pizza). Man kan t.om. tänka sig arvshierarkier av builders.

- Ledningen bestämmer sig för att alla pizzor ska gå att få både inbakade och med ostfyllda kanter.
  - Hur ska vi lösa detta?
    - ★Skapa flera builders? - Dålig ide.
    - ★Arvshierarki av Pizza? – Dålig ide.
  - Vi vill kunna lösa detta utan att ändra för mycket i nuvarande implementation och ändå få en lösning som är utbyggbar och inte skapar för hårda beroenden.
  - Leder oss in på nästa design pattern…

Decorator Pattern

- Decorator Pattern (Structural Pattern)
  Ge ytterligare funktionalitet till ett objekt, utan att ärva från det
  - Används ofta då arv skulle ge alldeles för många barnklasser eller inte är lika applicerbart
- Vi kommer att ”dekorera” Pizza klassen med olika decorators
  - BakedInPizzaDecorator
  - CheeseCrustPizzaDecorator
  - Kanske kommer fler i framtiden



- Vi måste göra om Pizza klassen till en abstrakt klass och lägger till en metod

```
public abstract Pizza {
   public String sauce;
   public String topping;

   public abstract void prepare();

   public void setSauce(String sauce) { this.sauce = sauce; }
   public void setTopping(String topping) {
      this.topping = topping; }
}
```

```
public class SimplePizza extends Pizza{
   public void prepare(){
   //tom implementation
   }
}
```

Decorator Pattern

- Nu behöver vi en basklass för våra decorators (ärver från Pizza)

```
abstract class PizzaDecorator extends Pizza {
    //Vi måste hålla en referens till vår dekorerade pizza
    protected Pizza decoratedPizza;

    public PizzaDecorator(Pizza pizza){
        decoratedPizza = pizza;
    }


    //klassen är abstract eftersom den inte impl. prepare()
}
```


- Konkret decorator för inbakade pizzor

```
class BakedInPizzaDecorator extends PizzaDecorator {
    public BakedInPizzaDecorator(Pizza pizza){
        super(pizza); //sparar undan referensen
    }


    public void prepare(){
        //Först gör vi det vi ska, sen låter vi prepare()
        //anropet "gå vidare"
        bakeIn();
        decoratedPizza.prepare();
    }


    public void bakeIn(){
        //kod för att förbereda en inbakad pizza.
    }
}
```


- Konkret decorator för pizzor med ostfyllda kanter

```
class CheeseCrustPizzaDecorator extends PizzaDecorator {
    public CheeseCrustPizzaDecorator(Pizza pizza){
        super(pizza); //sparar undan referensen
    }


    public void prepare(){
        //Först gör vi det vi ska, sen låter vi prepare()
        //anropet "gå vidare"
        addCheeseCrust();
        decoratedPizza.prepare();
    }


    public void addCheeseCrust(){
        //kod för att lägga till en ostfylld kant
    }
}
```


Decorator Pattern

- Hur har detta hjälpt oss?
  Vi går tillbaka till vår PizzaBuilder

```
abstract class PizzaBuilder { //abstrakt "builder"
   protected Pizza pizza;
   public Pizza getPizza() { return pizza; }
   public void createNewPizzaProduct() {
      //alla dessa är möjliga:
      pizza = new SimplePizza();
      pizza = new BakedInPizzaDecorator(new SimplePizza());
      pizza = new CheeseCrustPizzaDecorator(
                  new SimplePizza());
        //… och tom. denna (vilket vi var ute efter)
        pizza = new BakedInPizzaDecorator(
          new CheeseCrustPizzaDecorator(
                  new SimplePizza()));
       pizza.prepare(); //Olika resultat beroende på typ
    }
   public abstract void buildSauce();
   public abstract void buildTopping();
}
```


Decorator Pattern

- Med hjälp av decorators och builders kan vi nu…
  - konstruera flera olika objekt (pizzor) (builder pattern)
  - ha egenskaper som ska gå att applicera på alla objekt (decorators)
  - fortfarande ha en arkitektur som är skalbar och har lösa beroenden
- Det är fortfarande en aning komplicerat att beställa pizzor
  - Användaren (kunderna) måste veta om vilka builders och decorators som finns
  - Leder oss in på vårt sista design pattern…

Façade

- The Façade Pattern (Structural pattern)*Erbjud ett enhetligt gränssnitt till en mängd klasser och objekt i ett sub-system*
  - En ”fasad” till sub-systemet skulle göra det enklare att använda, då man inte behöver veta alla ingående komponenter
  - Detta betyder dock inte att det inte går att komma åt komponenterna
  - Låt oss titta på hur en fasad till ”pizza-systemet” skulle kunna se ut

```
public class Waitress {

    public Pizza orderHawaii(boolean bakedIn, boolean wCheese){
        PizzaChef chef = new PizzaChef();
        HawaiiPizzaBuilder hawaiianBuilder = new
    HawaiiPizzaBuilder();
        Pizza hawaii = chef.constructPizza(hawaiianBuilder
            bakedIn, wCheese);
        return hawaii;
    }

    public Pizza orderSalami(boolean bakedIn, boolean wCheese){
        PizzaChef chef = new PizzaChef();
        SalamiPizzaBuilder salamiBuilder = new
    SalamiPizzaBuilder();
        Pizza salami = chef.constructPizza(salamiBuilder
            bakedIn, wCheese);
        return salami;
    }


    // Här lägger vi till om vi har fler pizzor
}
```


Façade

- Hur ser det då ut från kundens perspektiv?

```
public class TestClass {

    public static void main(String[] args){
        Waitress waitress = new Waitress();

        Pizza myPizza = waitress.orderHawaii(true, false);

        Pizza myPizza2 = waitress.orderSalami(false, true);
    }
}
```

- Detta är alltså allt som användaren behöver känna till

Databaser, SQL och JDBC

Databaser - kortfattat

- Förenklad beskrivning - om ni vill veta mer bör ni gå den ordinarie databaskursen
- Vilken ide finns bakom databaser
  - Lagra information i en strikt struktur
    - Fritext sökning
  - Göra det möjligt att snabbt hämta information

Representera information

Lena Andersson	Kurt Olssons Väg 12 123 45 Långbortistan	090 123 45 67

Lena	Andersson	Kurt Olssons Väg 12 123 45 Långbortistan	090 123 45 67

Lena	Andersson	Kurt Olssons Väg	12	123 45	Långbortistan	090	123 45 67

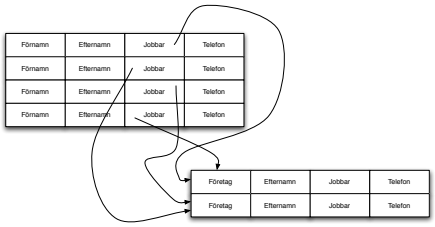
Men matchar allting detta?

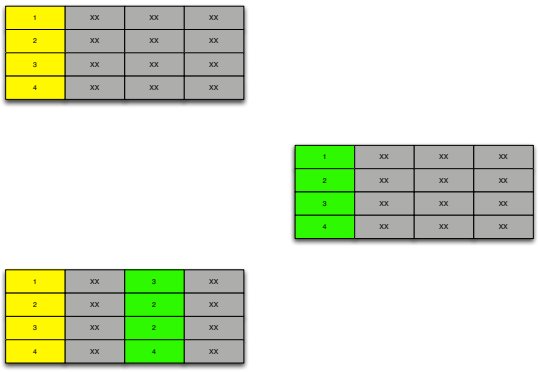
Kalle Svensson Karlsson
Box 21
234 56 Tomby

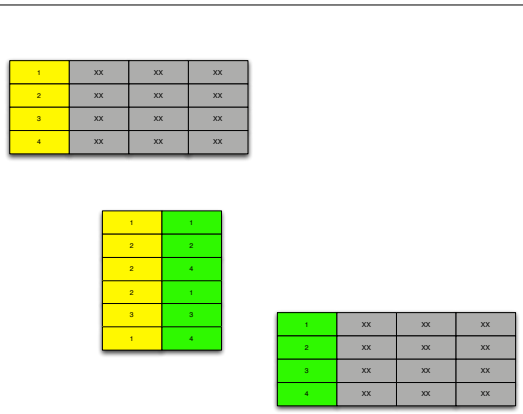
Svea Grahn
Västgötagatan 45, 3tr
345 78 Gamby

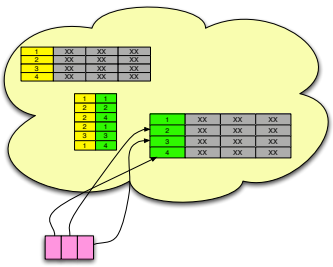
Clark Kent
21th Super Street
Hero City 123-2389

Lena	Andersson	Kurt Olssons Väg	12	123 45	Långtbortistan	090	123 45 67
Erik	Karlsson	Videvägen	18	987 65	Långtbortistan	090	890 123
Mia	Grahn	Videvägen	1289	876 54	Långtbortistan	090	222 22 22
Fia	Talltopp	Storgatan	1	129 21	Långbortistan	090	123 4857







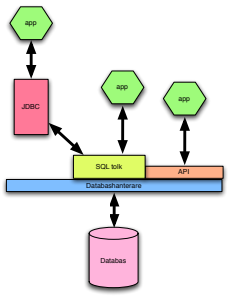


Hur gör man rent praktiskt??

- Olika implementationer, t.ex:
  - MySQL
  - Postgresql
  - Oracle
  - SQLite
  - SQLServer
  - ...

Förenklad bild av hur det fungerar

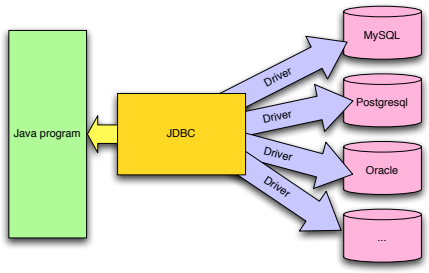
Inte sann men ger en översikt



SQL

- Standard Query Language
- Select
  SELECT XXX FROM YYY WHERE ZZZ
- Insert
  INSERT INTO XXX VALUES YYY
- Delete
  DELETE FROM XXX WHERE YYY
- Update
  UPDATE XXX SET YYY WHERE ZZZ

JDBC - princip



Fyra begrepp

- Statement
- Query
- ResultSet
- Transaction

- Ladda drivrutinen
  Class.forName(driverPath)
- Öppna förbindelse
  con = DriverManager.getConnection(url, user, password)
- Fixa till ett statement
  ms = con.createStatement()
- Gör nåt
  res = ms.executeQuery(sqlStuff)
  nr = ms.executeUpdate(sqlStuff)


```
final private static String driver = "org.postgresql.Driver";
final private static String url    = "jdbc:postgresql://postgres.cs.umu.se/X";
final private static String user   = "X";
final private static String passwd = "X";

public static void main( String argv[] )
    throws java.sql.SQLException
{
    Connection connection;

    try{
        Class.forName( driver );
    } catch ( java.lang.ClassNotFoundException  e ) {
        System.err.println( "Problem - Kunde inte hitta drivrutinen" );
        return;
    }
```

```
try{
     connection = DriverManager.getConnection( url, user, passwd );
} catch ( java.sql.SQLException e ) {
     System.err.println( "Problem - Kunde inte ansluta till databasen" );
     return;
}
```

```
Statement statement = connection.createStatement( );

statement.execute( "create table names (first char(20), last char(20))" );

statement.execute( "insert into names (first, last) values ('Kalle', 'Anka')" );
statement.execute( "insert into names (first, last) values ('Kajsa', 'Anka')" );
statement.execute( "insert into names (first, last) values ('Knatte', 'Anka')" );
statement.execute( "insert into names (first, last) values ('Fnatte', 'Anka')" );
statement.execute( "insert into names (first, last) values ('Tjatte', 'Anka')" );
statement.execute( "insert into names (first, last) values ('Joakim', 'von Anka')" );
statement.execute( "insert into names (first, last) values ('Kalle', 'Johansson')" );
statement.execute( "insert into names (first, last) values ('Anna', 'Andersson')" );
```



```
ResultSet result = statement.executeQuery( "select last, first from names" );
System.out.println( "\n\nTest 1" );
while( result.next() ){
    System.out.println( result.getString(1).trim() + ", " + result.getString(2).trim() );
}
```

```
Test 1
Anka, Kalle
Anka, Kajsa
Anka, Knatte
Anka, Fnatte
Anka, Tjatte
von Anka, Joakim
Johansson, Kalle
Andersson, Anna
```



```
result = statement.executeQuery( "select last, first from names order by last, first" );
System.out.println( "\n\nTest 2" );
while( result.next() ){
    System.out.println( result.getString(1).trim() + ", " + result.getString(2).trim() );
}
```

```
Test 2
Andersson, Anna
Anka, Fnatte
Anka, Kajsa
Anka, Kalle
Anka, Knatte
Anka, Tjatte
Johansson, Kalle
von Anka, Joakim
```



```
result = statement.executeQuery( "select last, first from names where last = 'Anka' order by last, first" );
System.out.println( "\n\nTest 3" );
while( result.next() ){
    System.out.println( result.getString(1).trim() + ", " + result.getString(2).trim() );
}
```

```
Test 3
Anka, Fnatte
Anka, Kajsa
Anka, Kalle
Anka, Knatte
Anka, Tjatte
```



```
statement.execute( "update names set first = 'Karl' where first = 'Kalle'" );
result = statement.executeQuery( "select last, first from names order by last, first" );
System.out.println( "\n\nTest 4" );
while( result.next() ){
    System.out.println( result.getString(1).trim() + ", " + result.getString(2).trim() );
}
```

```
Test 4
Andersson, Anna
Anka, Fnatte
Anka, Kajsa
Anka, Karl
Anka, Knatte
Anka, Tjatte
Johansson, Karl
von Anka, Joakim
```



```
statement.execute( "delete from names where last = 'Anka'" );
result = statement.executeQuery( "select last, first from names order by last, first" );
System.out.println( "\n\nTest 5" );
while( result.next() ){
    System.out.println( result.getString(1).trim() + ", " + result.getString(2).trim() );
}
```

```
Test 5
Andersson, Anna
Johansson, Karl
von Anka, Joakim
```

DANGER ALERT

DANGER ALERT

DANGER ALERT

DANGER ALERT

DANGER ALERT

DANGER ALERT

DANGER ALERT

DANGER ALERT

DANGER ALERT

DANGER ALERT

DANGER ALERT

DANGER ALERT

DANGER ALERT

DANGER ALERT

DANGER ALERT


```
String first = "Arne";
String last  = "Anka";
statement.execute( "insert into names (first, last) values ('" + first + "', '" + last + "')" );
result = statement.executeQuery( "select last, first from names order by last, first" );
System.out.println( "\n\nTest 6" );
while( result.next() ){
    System.out.println( result.getString(1).trim() + ", " + result.getString(2).trim() );
}
```



```
first = "Arne', 'Anka'); drop table names; insert into funny ( woop, miff) values ('Kalle";
last  = "Anka";
```

```
statement.execute( "insert into names (first, last) values ('" + first + "', '" + last + "')" );
```


Precompiled statements

```
PreparedStatement prep =
    con.prepareStatement(
        "INSERT INTO XXX (a,b,c) VALUES (?,?,?)"
    )
```

```
prep.setXXX( index, value )
```

```
prep.executeUpdate ... etc
```


Datatyper i databaser

- Olika databaser stödjer olika datatyper
- **postgresql** - smallint, integer, bigint, decimal, real, serial, money, varchar, text, timestamp, date, time, byte, boolean, etc.
- **mysql** - char, varchar, tinytext, text, blob, longtext, tinyint, int, bigint, date, datetime, timestamp, time, etc.

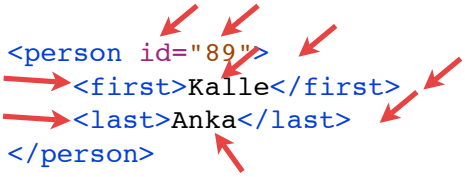
Vad ska man tänka på?

- Tänk några steg framåt
  - Förändringar i framtiden
  - Överdesigna inte
- Lagra information på **ett** ställe
- Ofta bra att ha ett unikt id på (nästan) varje post
- Gör inte en databashanterare i ditt program !!!
  - Utnyttja databashanteraren och låt den göra jobbet

Validera XML dokument

Validera att dokument är riktiga

- Varför validera?
- Jo, för att få enklare kod



Alternativ

- DTD - Document Type Definition
- XML Schema

```
<!ELEMENT font (name,size)>
```

```
<xsd:element name="font">
    <xsd:sequence>
        <xsd:element name="name" type="xsd:string" />
        <xsd:element name="name" type="xsd:int" />
    </xsd:sequence>
</xsd:element>
```


XML Schema

- Beskriver hur XML dokumentet ska se ut
- Är själv ett XML dokument
- Vad gör ett XML Schema
  - definierar element
  - definierar attribut
  - definierar datatyper


```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

</xsd:schema>
```


Vad innehåller ett schema

- Innehåll
  - Type defintion - simple eller complex
  - Elementdefinitioner
  - Attributdefinitioner
- Simple type
  - Inga sub-element
  - Inga attribut
- Complex type
  - Har sub-element, element eller attribut
- Globala definitioner
  - Dokumentens rotelement
  - Kan refereras

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="purchaseorder" type="PurchaseOrderType">
    <xsd:element name="comment" type="xsd:string" />
    <xsd:complexType name="purchaseOrderType">

    </xsd:complexType>
</xsd:schema>
```



```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="purchaseorder" type="PurchaseOrderType">
    <xsd:element name="comment" type="xsd:string" />
    <xsd:complexType name="purchaseOrderType">
        <xsd:sequence>
            <xsd:element name="shipTo" type="USAddress" />
            <xsd:element name="billTo" type="USAddress" />
            <xsd:element ref="comment" minOccurs="0" />
        </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="USAddress">

    </xsd:complexType>
</xsd:schema>
```


Finns fördefinerade typer (simple)

- xsd:string
- xsd:QName
- xsd:int
- xsd:long
- xsd:decimal
- xsd:double
- xsd:boolean
- xsd:date
- xsd:time
- xsd:dateTime


```
<xsd:element name="lastname"  type="xsd:string" />
<xsd:element name="age"       type="xsd:int" />
<xsd:element name="birthdate" type="xsd:date" />
```

```
<lastname>Andersson</lastname>
<age>34</age>
<birthdate>2006-12-18</birthdate>
```

```
<xsd:attribute name="nationality"
   type="xsd:string" />
```

```
<person nationality="Sweden">Kalle Anka</person>
```



```
<xsd:simpleType name="CarType">
    <xsd:restriction base="xsd:string">
        <xsd:enumeration value="Audi" />
        <xsd:enumeration value="Ford" />
        <xsd:enumeration value="Volvo" />
    </xsd:restriction>
</xsd:simpleType>
<xsd:element name="car" type="CarType">
```


Begränsningar

- minOccurs, maxOccurs
  - default = 1, max kan vara "unbounded"
- Sequence - Choice - All

Validera vid parsning

```
String schemaLang = "http://www.w3.org/2001/XMLSchema";
SchemaFactory schemaFactory = SchemaFactory.newInstance(schemaLang);
//create schema by reading it from an XSD file
Schema schema = schemaFactory.newSchema(
                    new StreamSource("classes.xsd"));
factory.setSchema(schema); //add validation using the schema
DocumentBuilder parser = factory.newDocumentBuilder();
Document document = parser.parse( new InputSource( r ) );
```


Validera separat

```
import javax.xml.transform.stream.StreamSource;
import javax.xml.validation.Schema;
import javax.xml.validation.SchemaFactory;
import javax.xml.validation.Validator;
import org.xml.sax.SAXException;

public class Test {
  public static void main(String[] args) {
    try {
      // define the type of schema - we use W3C:
      String schemaLang = "http://www.w3.org/2001/XMLSchema";
      // get validation driver:
      SchemaFactory factory = SchemaFactory.newInstance(schemaLang);
      // create schema by reading it from an XSD file:
      Schema schema = factory.newSchema(new StreamSource("sample.xsd"));
      Validator validator = schema.newValidator();
      // at last perform validation:
      validator.validate(new StreamSource("sample.xml"));
    }catch (SAXException ex) {
      // we are here if the document is not valid:
      // ... process validation error...
    } catch (Exception ex) {
      ex.printStackTrace();
    }
  }
}
```


Validera mha DTD

```
// Create a new DOM Document Builder factory
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();

// Turn on validation
factory.setValidating(true);

// Create a validating DOM parser
DocumentBuilder builder = factory.newDocumentBuilder();
```