TDP024

Enterprise Systems
JPA, Concurrency

Anders Fröberg
anders.froberg@liu.se

- Först lite recap av Programing-to-an-Interface

Program to an interface

```
public interface Sort {
    List sort(List list);
}
```

När jag skriver kod så jobbar jag mot interfacet, jag vet att alla klasser som uppfyller kontraktet kommer ha en funktion som heter sort och tar en lista och returnerar en lista.

```
public class BubbleSort implements Sort {
   public List sort(List list) {
      //Do bubble sort
   }
}
```

```
public class QuickSort implements Sort {
    public List sort(List list) {
       //Do quick sort
    }
}
```

Program to an interface

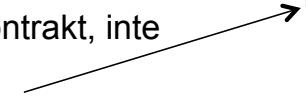
```
public interface Sort {
    List sort(List list);
}
```

```
public class BubbleSort implements Sort {
   public List sort(List list) {
     //Do bubble sort
    }
}
```

```
public class QuickSort implements Sort {
    public List sort(List list) {
       //Do quick sort
    }
}
```

```
public class QuantSort implements Sort {
    public List sort(List list) {
       //Do quick sort
    }
}
```

När någon sedan kommer fram till en snabbare sorteringsalgoritm så behöver jag inte ändra i min kod, för den är skriven mot ett kontrakt, inte mot en implementation.



Program to an interface

```
public class ListUtils {

   private Sort sort;

   public List findSmallest(List list) {
      list = sort.sort(list);
      return list.get(0);

   }

}
```

*Interface, det står **inte**:*
private BubbleSort sort;

Använder en metod som jag vet finns enligt kontraktet, oavsett vilken implementation jag använder.

”That’s a null pointer exception!”

Dependency Injection

```
public class ListUtils {

  private Sort sort;

  public ListUtils(Sort sort) {
    this.sort = sort;
  }

  public List findSmallest(List list) {
    list = sort.sort(list);
    return list.get(0);

  }

}
```

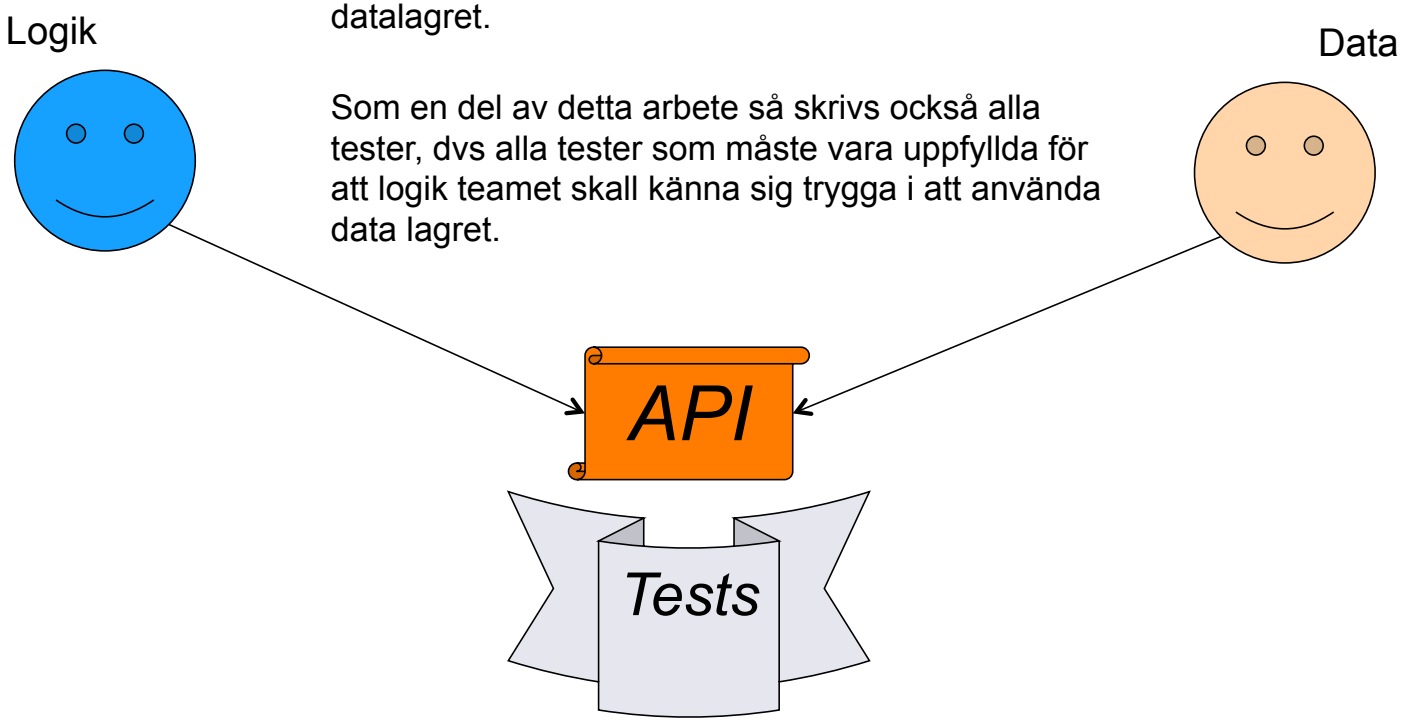
ListUtils väljer inte vilken implementation den skall använda. I sin konstruktor så accepterar den en instans av en implementation. (Notera att vi fortfarande endast använder interfaces).

Detta kallar vi **dependency injection** det består alltså av en klass som har en instansvariabel som blir satt i konstruktorn.

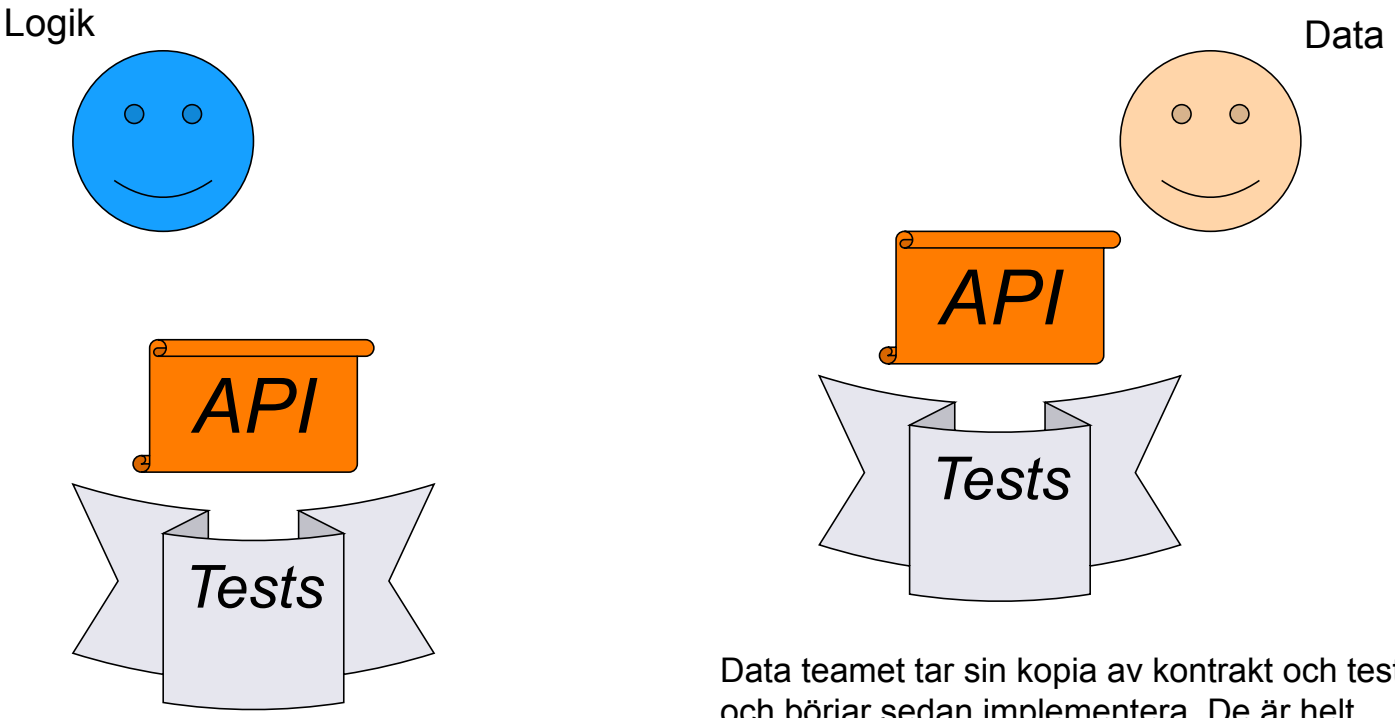
Data – Logic - Web

Data och Logik teamen sitter tillsammans och kommer överens om ett API (dvs ett kontrakt) för vilka funktioner som kommer behövas från datalagret.

Som en del av detta arbete så skrivs också alla tester, dvs alla tester som måste vara uppfyllda för att logik teamet skall känna sig trygga i att använda data lagret.

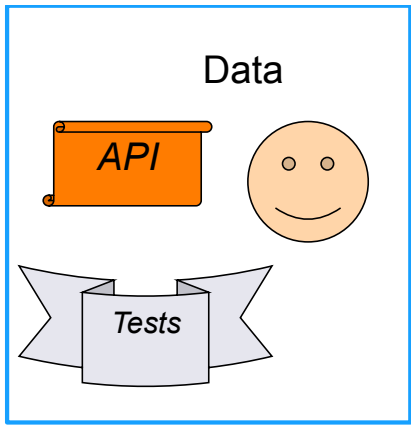


Data – Logic - Web

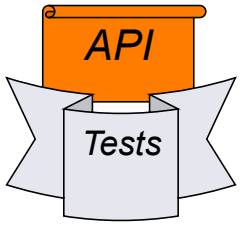


Data teamet tar sin kopia av kontrakt och tester och börjar sedan implementera. De är helt oberoende av alla andra delar av projektet. Och resten av projektet är helt oberoende av deras implementationsdetaljer.

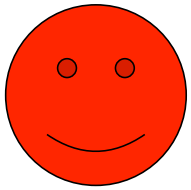
Data – Logic - Web



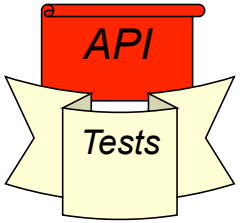
Logik



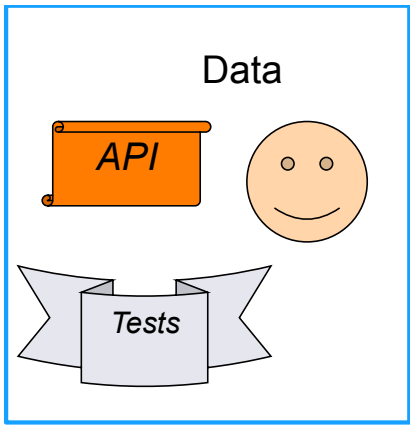
Web

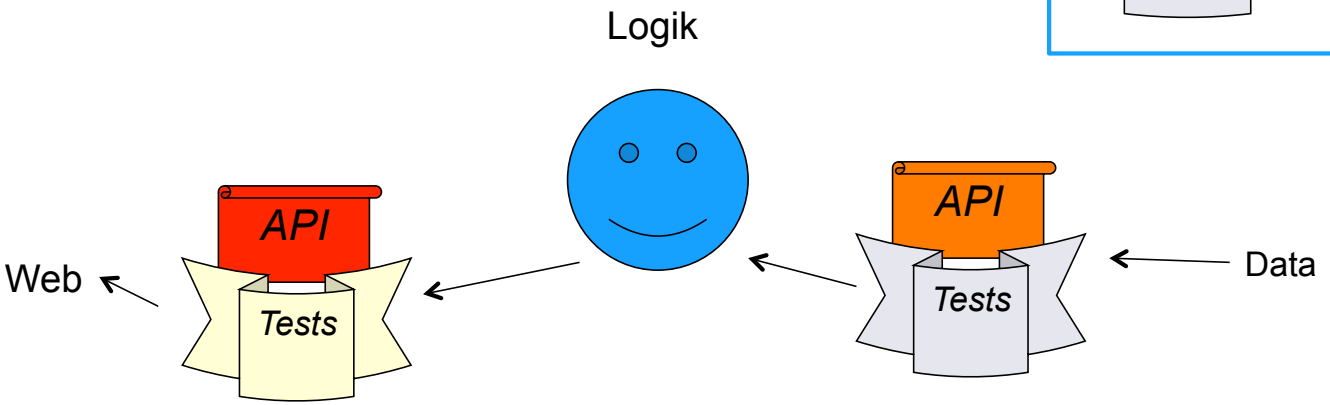


Web och logik teamen gör precis samma sak. De kommer överens om ett kontrakt för kommunikation mellan web och logiklagret. I denna process skrivs även tester.



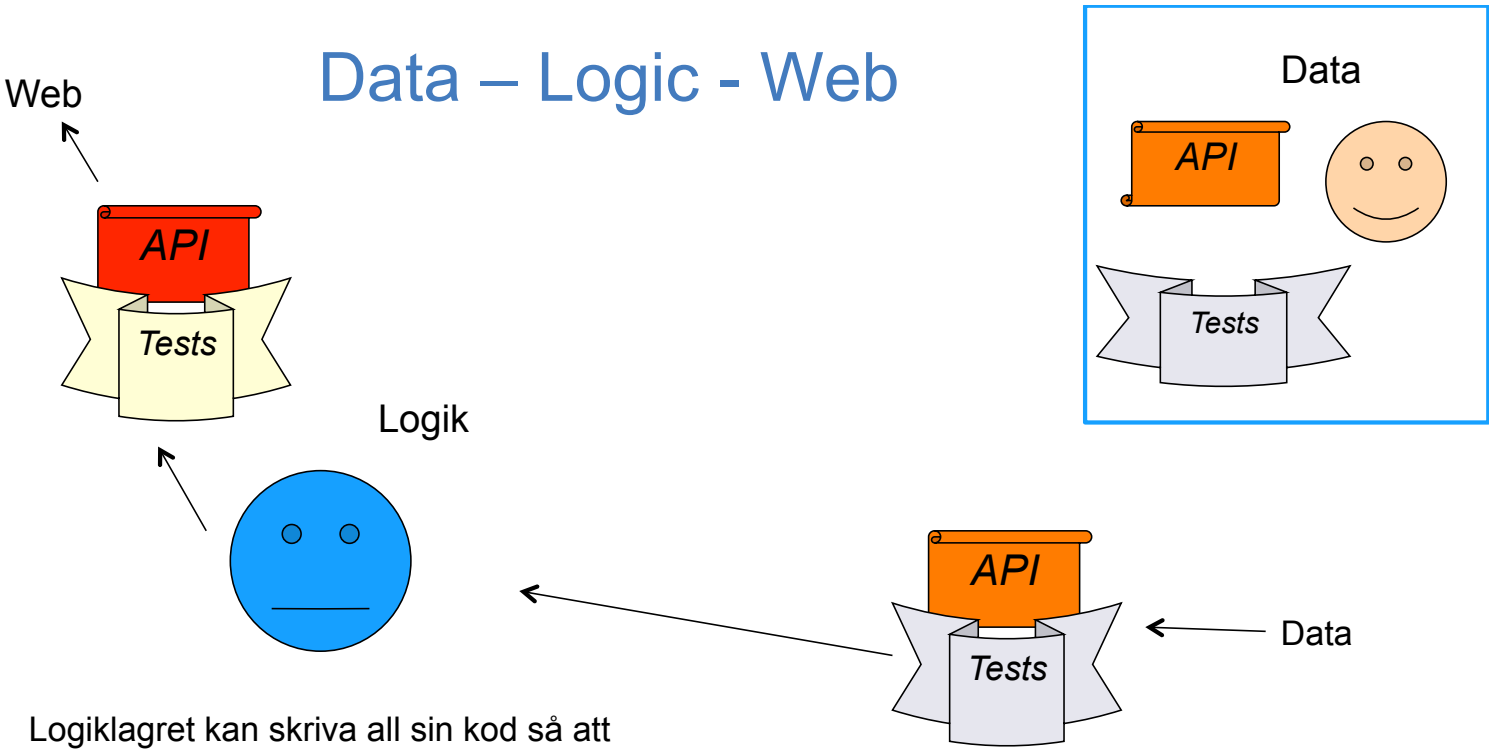
Data – Logic - Web





Nu har logik lagret ett kontrakt som de vet kommer bli uppfyllt (samt tester). Och de har ett kontrakt som de skall uppfylla (samt tester).

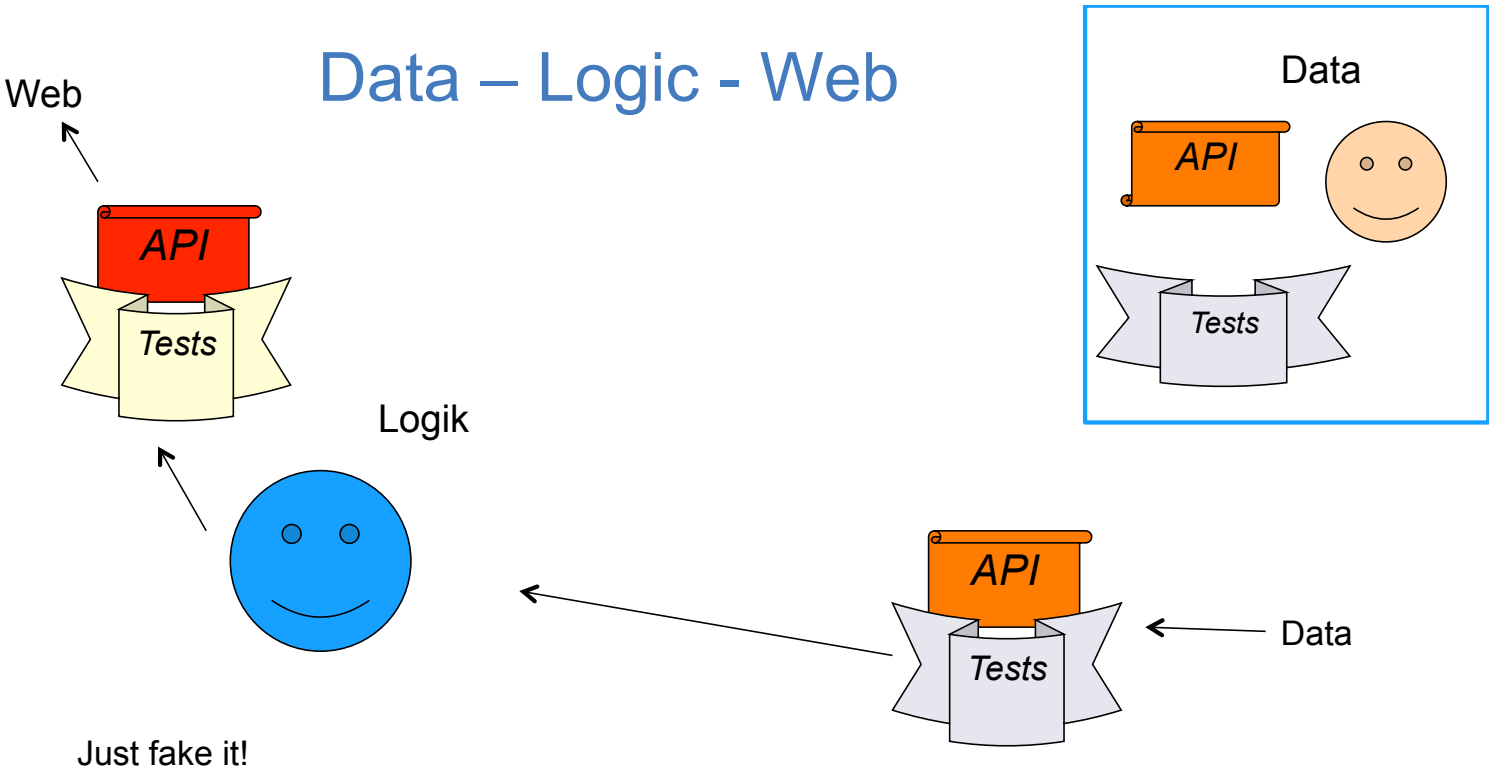
Data – Logic - Web



Logiklagret kan skriva all sin kod så att den uppfyller kontraktet mot web. Eftersom logiklagret **ALLTID** jobbar mot kontraktet så behöver man inte vänta på en implementation från datalagret.

Men hur kan man testa sin logik utan att kunna skapa instanser av de kontrakt som data lagret ska leverera? Måste man vänta på att data lagret blir klara?

Data – Logic - Web



Just fake it!

Logik lagret har ju kontrakten från data lagret, de kan skapa sitt egna lilla fejk datalager som de kan använda för att testa med.

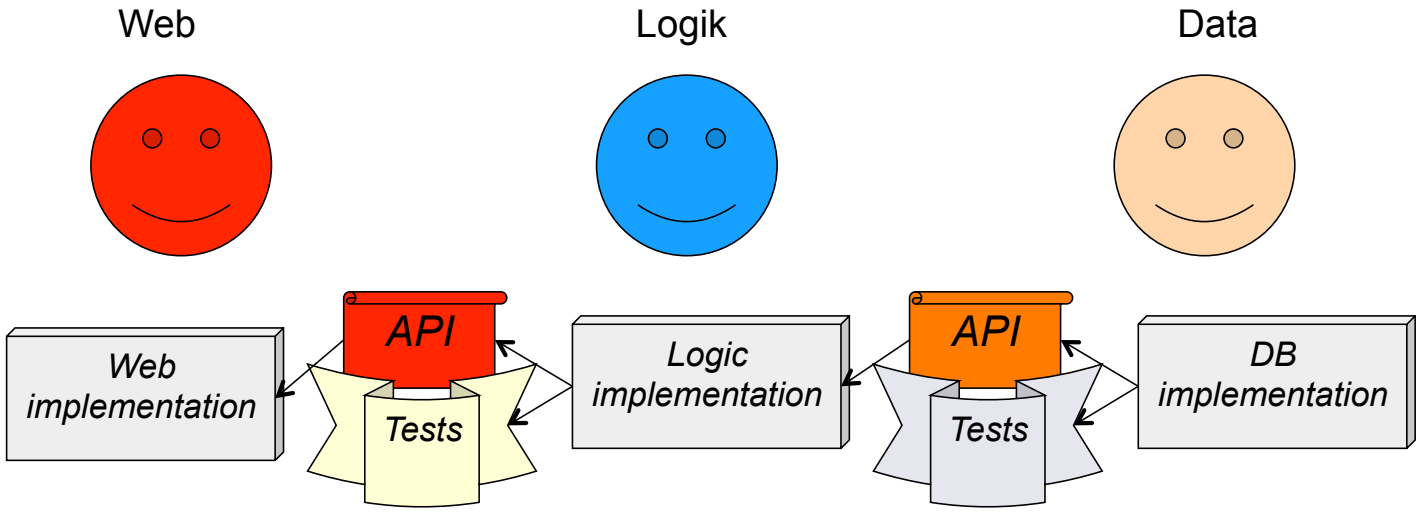
Detta kallas ofta *”mocking”* eller *”creating mock objects”*

Mock objects - Mocking

```
public class Test {

   //--- Unit under test ---//
   private Sort sort = new Sort() {
      public List sort(List sort) {
        return [2];
      }
   }
   //----------------------//

   public void test() {
      List a = [4,3,5,2,6];
      List b = sort.sort(a);
      Assert.assertTrue(b[0] == 2);
   }

}
```

- Så vi skapar en egen implementation av Sort (till dess att data lagret är klara med sin) som vi kan testa med.
- Den behöver inte vara bra, så länge den gör vad den ska.
- När data lagret sedan är klara så stoppar vi in deras implementation istället.

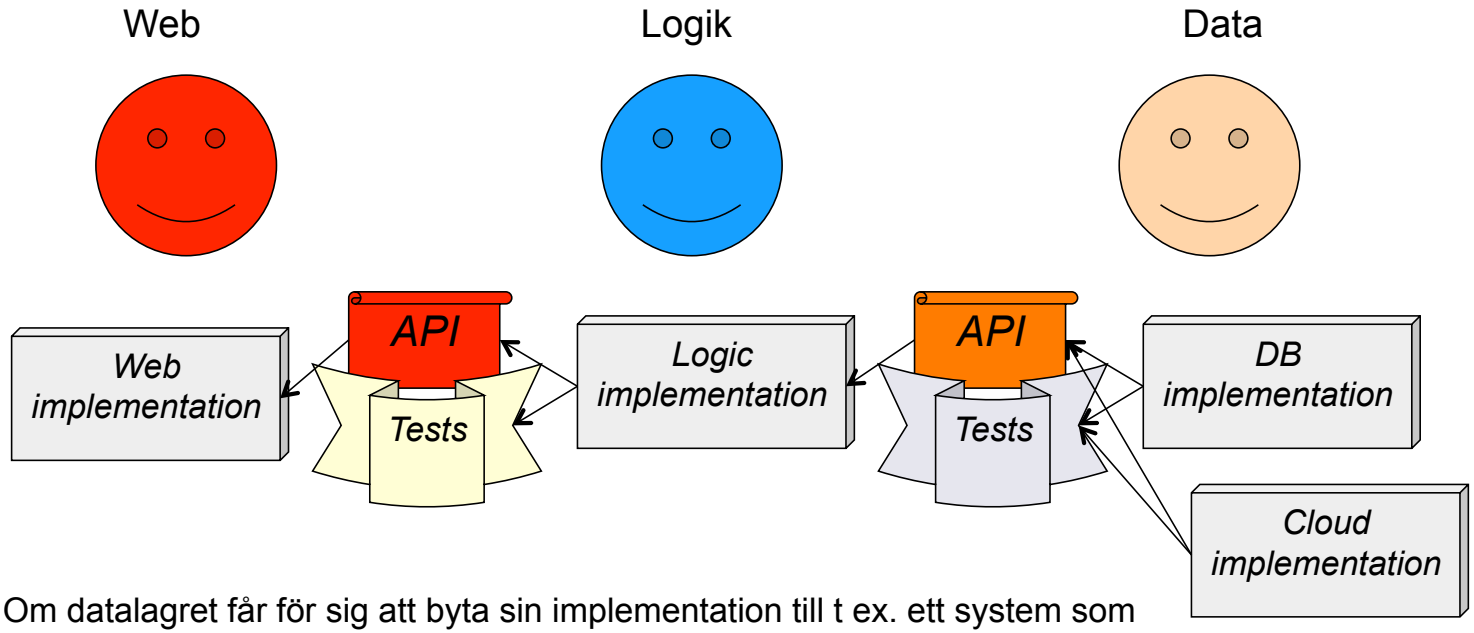
Data – Logic - Web



När alla delar är på plats så har man ett system som är modulärt och hållbart.

De olika implementationerna är endast beroende på kontrakten.

Data – Logic - Web



Om datalagret får för sig att byta sin implementation till t ex. ett system som sparar i en **molntjänst** istället för på lokal databas, eller vill helt plötsligt börja spara i **flat-files**, så spelar det ingen roll för de andra. Så länge datalagret lovar att **upprätthålla kontraktet och testerna** så behöver ingen ändra sin kod.

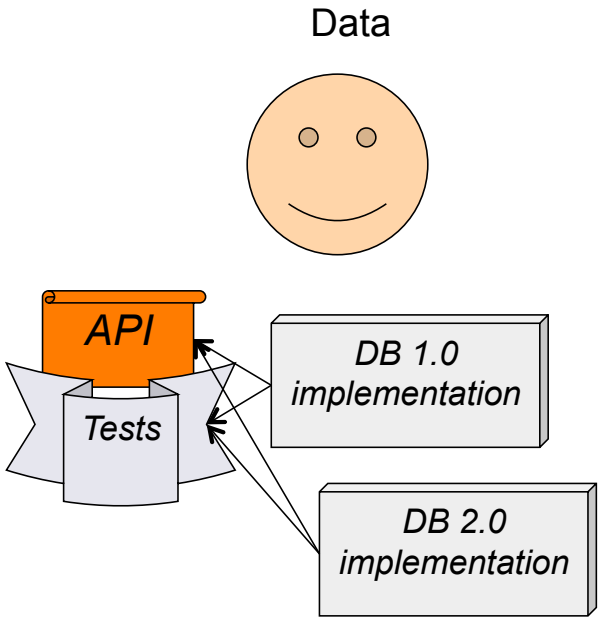
Data – Logic - Web

Även inom data lagret är det smidig att testa nya implementationer (eftersom de redan har tester).

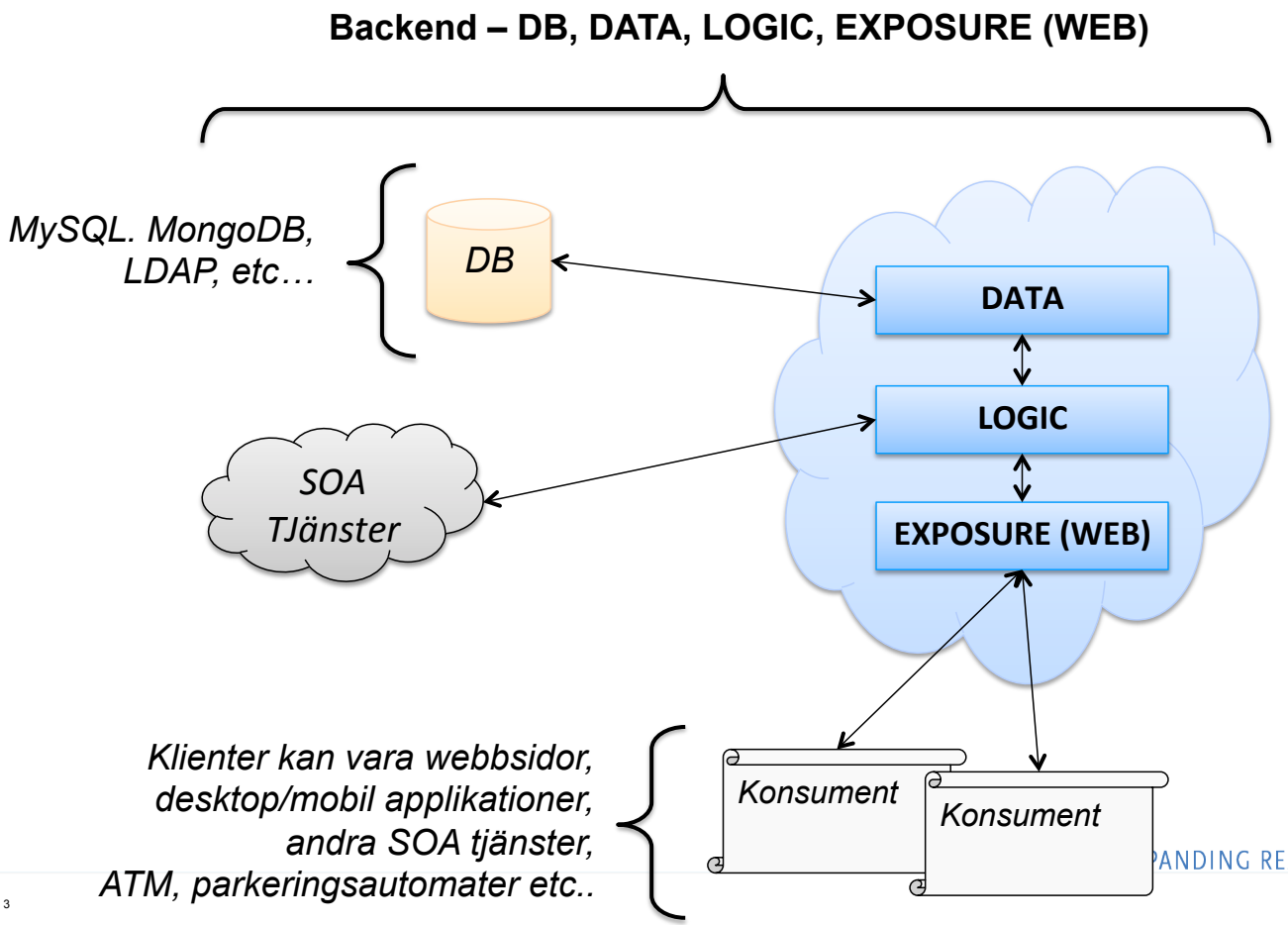
Antag att ett av biblioteken som data lagret använder uppdateras från 1.0 till 2.0, och mycket har ändrats i den nya versionen.

Istället för att börja ändra i sin existerande implementation så skapar man en ny, som är helt baserad på 2.0, men testerna är de samma.

När man väl är färdig med implementationen (och den passerar alla tester) så kan man byta ut den gamla utan att någon behöver ändra på sin kod.

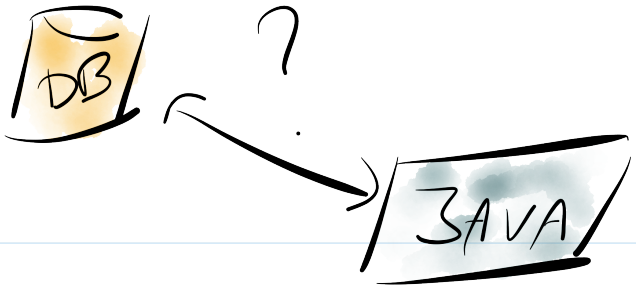


Backend – SOA Tjänst – Egentligen flera lager



Motivation & Problem

- Sparande och förmedling av information är det centrala i våra tjänster
- Databaser gör det möjligt för oss att spara information på ett säkert och stabilt sätt
- Data bör komma in och ut ur applikationen på ett naturligt sätt



- En SQL databas är en samling tabeller med rader och kolumner
- En Java applikation består utav objekt
- JPA är ett Java bibliotek (del utav Java EE) som ”översätter” Java klasser till rader och kolumner - ORM (Objekt-Relational-Mapping)

JPA – POJO & @Entity

- Instanser av klasser annoterade med @Entity kan sparas till databasen
- Som ett minimum behöver man annotera ett fält med @Id för primary-key
- Egentligen är klassen en POJO (Pure Old Java Objekt) utan några speciella krafter
- @Column används för att konfigurera kolumnen för ett fält (nullable, unique, etc)
- JPA kommer skapa tabeller som anses nödvändiga
- Kan hantera komplexa relationer mellan klasser och även datatyper så som HashMap och List (med lite konfiguration)

JPA - Entity Exempel

Interface

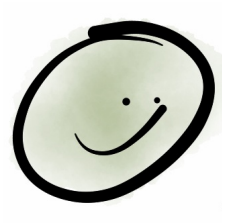
```
public interface Todo {

    long getId();

    void setId(long id);

    String getTitle();

    void setTitle(String title);
}
```

Implementation

```
@Entity
public class TodoDB implements Todo {

    @Id
    private long id;
    private String title;
    private String body;

    @Override
    public long getId() {
        return id;
    }
    ...
}
```

JPA – Table Example

id - bigint(20)	title - varchar(255)	content - varchar(255)
1	Title 1	Content 1
2	Title 2	Content 2
3	Title 3	Content 3
…	…	…
…	…	…



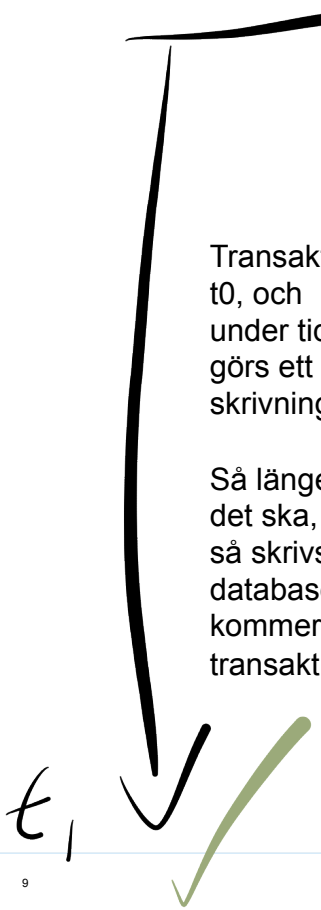
Behöver inte fundera på vilka tabeller eller kolumner som bör skapas, detta görs åt mig.

Innebär att det är lättare att byta databas den dagen företaget väljer att köpa in t ex Oracle, eller helt byta till t ex MongoDB (som inte består utav rader och kolumner alls)

EntityManager & Transactions

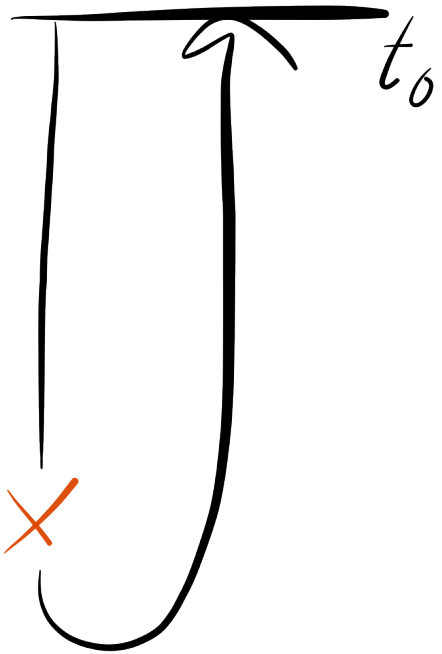
- Förutom att använda *annotations* (@) på klasser som skall sparas i databasen så behöver vi **kommunicera** med databasen.
- En **EntityManager** skapas genom klassen **EntityManagerFactory**
- EntityManager ansvarar för att skapa, starta och avsluta **transactions**
- All kommunikation som skriver till databasen måste ske inom en transaktion
- Antingen så lyckas alla operationer inom en transaktion eller så misslyckas alla

Transactions



Transaktionen öppnar vid t0, och
under tiden den är öppen görs ett antal skrivningar till databasen.

Så länge allt fungerar som det ska,
så skrivs allt data till databasen och vi kommer till t1, där transaktionen stängs



Om något går fel inom transaktionen, så görs först en ”**rollback**” på alla operationer, och transaktionen stängs.
Allt är som det var vid t0.

Glöm inte bort EntityManager, EntityManagerFactory och Transactions, vi återkommer till dessa strax!

Ett **design pattern** är ett namn på en ”konstruktion” eller uppsättning klasser som jobbar tillsammans på ett sätt som är användbart för många applikationer.

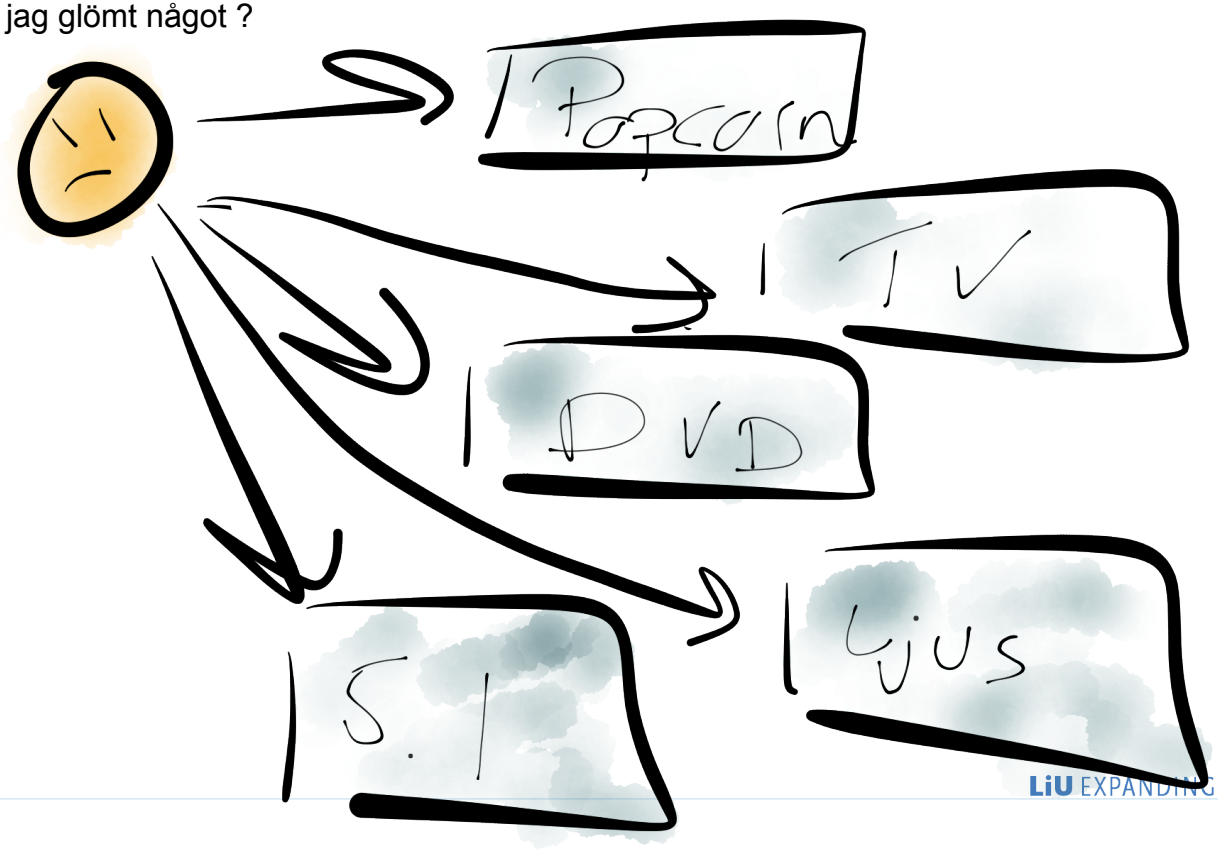
Det ger utvecklare ett vokabulär att kommunicera med istället för att upprepa sig själva hela tiden.

Intermezzo

FAÇADE DESIGN PATTERN

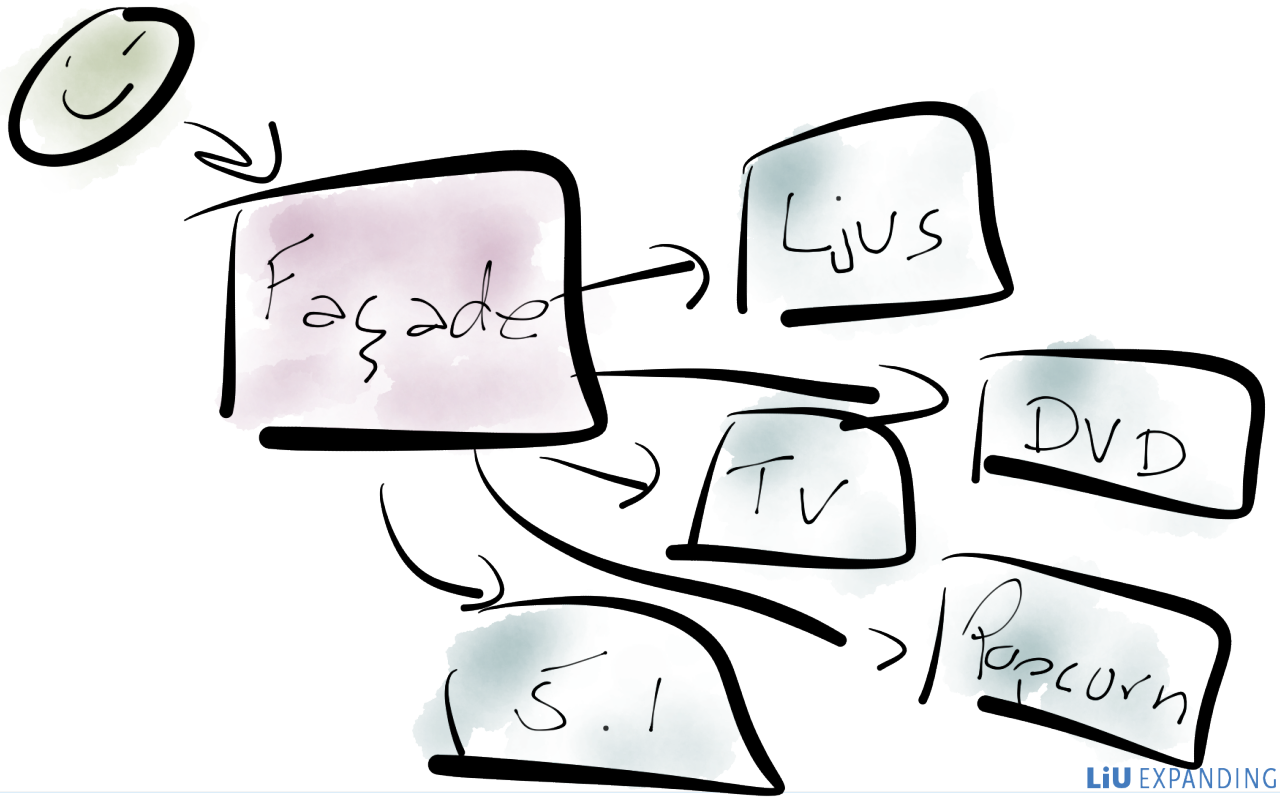
Intermezzo – Façade Design Pattern

Har jag glömt något ?



Intermezzo – Façade Design Pattern

Klart!



Intermezzo - Facade Design Pattern

- En klass vars uppgift det är att abstrahera komplexa delar, och på så vis förenkla interaktionen med systemet
- I **datalagret** använder vi façades för att erbjuda funktioner för ”verben” ***create, remove, update, find***
- Det är viktigt att tänka på att de façades som vi skapar i datalagret skall hantera **generiska** ”verb” som är lika mellan applikationer, inte saker som är unika för just din tjänst.
- Vi har även façades i **logiklagret** som kan hantera andra typer av ”verb”, så som ”checkIn” och ”checkOut” … mer om detta på senare föreläsningar

JPA – Façade, EMF, Transactions

```
public class TodoEntityFacadeDB implements TodoEntityFacade {

    @Overrides
    public long create(String title, String body) {
          EntityManager em = EMF.createEntityManager();

          em.getTransaction().begin();

          Todo todo = new TodoDB();
          todo.setTitle(title);
          todo.setBody(body);

          em.persist(todo);
          em.getTransaction().commit();

          return todo.getId();
    }
}
```

EMF är en klass som man måste skapa själv, det finns en sådan i kodskelettet .

Exemplet är inte komplett då vi inte hanterar eventuella fel på ett bra sätt. Mer om detta senare.

Interface på vänstersidan, implementation på höger!

JPA - Konfiguration

- JPA behöver **konfigureras** för att veta vilka klasser som skall sparas
- JPA behöver information om var databasen är och hur vi kopplar oss mot den

En grundläggande persistence.xml finns i kodskelettet.

- src/main/resources/persistence.xml
- Vi kommer använda en databas som bara existerar i RAM, så inga ändringar behövs för detta.
- Finns en hel del andra inställningar som är viktiga, t ex ”drop-and-create”

persistence.xml

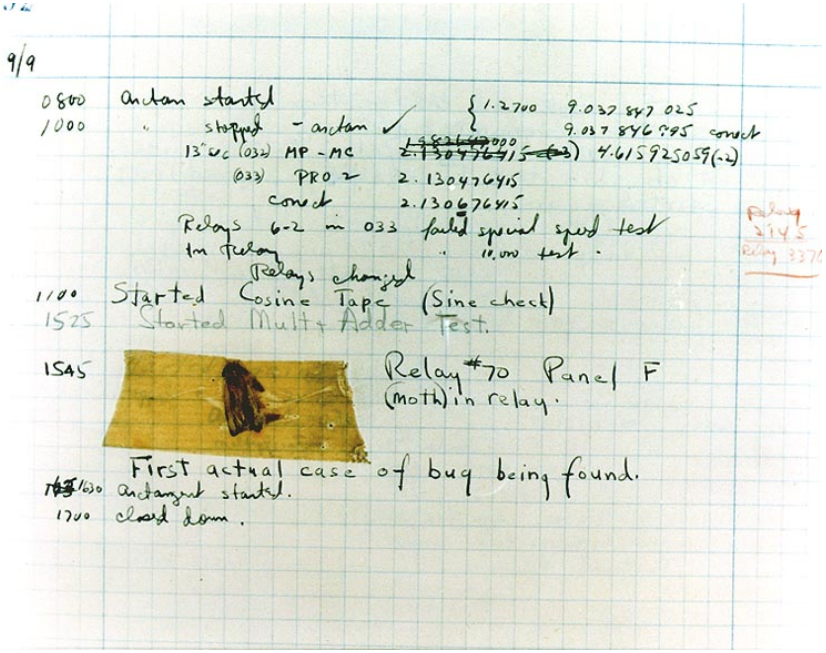
Lägg till en sådan rad för varje entity

….

```
<persistence-unit name="todo-dataPU" transaction-type="RESOURCE_LOCAL">
   <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
   <class>tdp024.todo.data.db.entity.TodoDB</class>
   <shared-cache-mode>NONE</shared-cache-mode>
   <validation-mode>NONE</validation-mode>
   <properties>
     <property name="javax.persistence.jdbc.url” value="jdbc:derby:memory:dataPU_embedded;create=
     <property name="javax.persistence.jdbc.password" value=""/>
     <property name="javax.persistence.jdbc.driver" value="org.apache.derby.jdbc.EmbeddedDriver"/
     <property name="javax.persistence.jdbc.user" value=""/>
     <property name="eclipselink.ddl-generation" value="drop-and-create-tables"/>
   </properties>
 </persistence-unit></persistence>
```

Motivation & Problem

- Hur vet man att façaden är klar och kan användas av logiklagret ?
- Om man ändrar i sin façade i efterhand, hur vet man att något inte gått sönder ?
- Hur vet man att det inte finns några buggar i sin façade ?



JPA - JUnit

- Viktigt att enhetstesta sin façade med Junit
- ”Test your interface, not your implementation”
- Under utvecklingen skriver man först test för de funktioner man tror sin façade kommer behöva, sedan skriver man själva funktionerna
- När alla testen kör så vet man att façaden är klar och fungerar
- Test Driven Development (TDD) är en del av eXtreme Programming
- De viktigaste fördelarna är:
  - Systemet är testat och fungerar som tänkt.
  - Endast de funktioner man behöver har utvecklats, dvs lösningen är inte ***over-engineered***.
  - Det går snabbt att kontrollera att ändringar i systemet inte har ändrat funktionaliteten (**recession testing**)

JPA - TDD

Realizing quality improvement through test driven development: results and experiences of four industrial teams - Nachiappan Nagappan & E. Michael Maximilien & Thirumalesh Bhat & Laurie Williams - (2008)

(From abstract of paper)

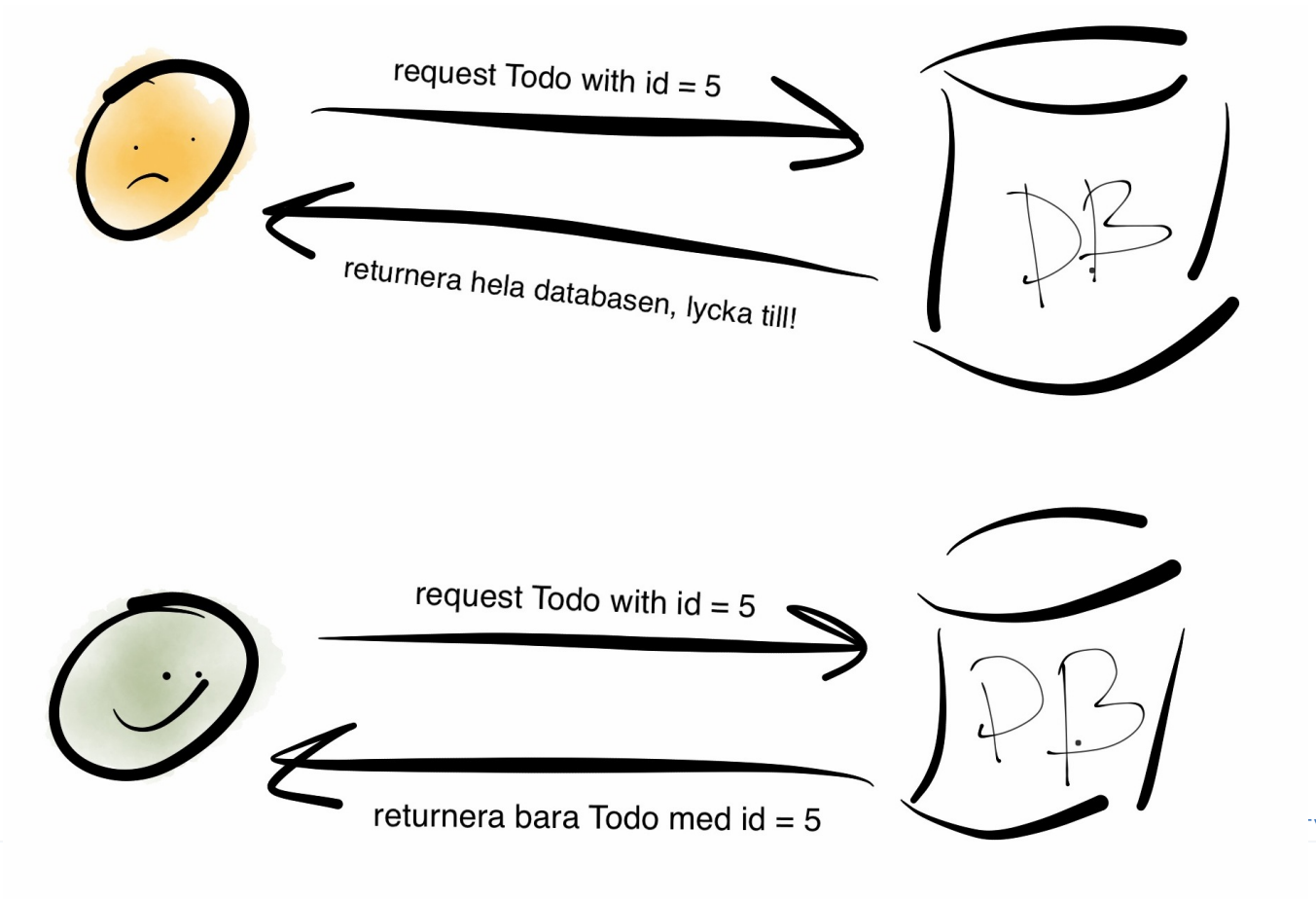
Case studies were conducted with three development teams at Microsoft and one at IBM that have adopted TDD. The results of the case studies indicate that the pre-release defect density of the four products decreased between 40% and 90% relative to similar projects that did not use the TDD practice. Subjectively, the teams experienced a 15–35% increase in initial development time after adopting TDD.

JPA – JUnit exempel

```
@Test
public void test() {

      TodoFacade todoFacade = new TodoFacade();
      long id = todoFacade.create(”Todo 1”, ”Don’t forget …”);
      Assert.assertFalse(id == 0);

}
```

Inte ett komplett exempel, men sådana finns i kodskelet och i föreläsnings kod. Det fina med att skriva testfall på detta sätt är att man kan få ut rapporter på hur många av sina test som gått igenom och att man kan få ***code-coverage***.

Motivation & Problem



JPA – Find & JPQL

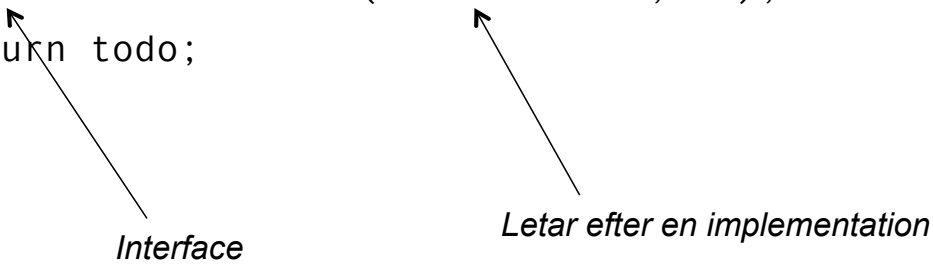
- Vi vill kunna hämta enskilda objekt ur databasen
- Vi vill hitta dessa via valfri column, t ex *id* eller *title*
- Vi vill också kunna hämta alla objekt tillhörande en klass, eller en filtrerad lista

- **JQPL** har ett SQL liknande syntax
- Skapa ett **Query** objekt genom **EntityManager**
- Sätt **parametrar** i Query objekt
- Hämta **ett objekt** eller en **lista**

JPA – Find & JPQL - Exempel

Find by id (@Id)

```
public Todo find(long id) {

    EntityManager em = EMF.createEntityManager();
    Todo todo = em.find(TodoDB.class, id);
    return todo;

}
```



Vi skriver inte till databasen, så vi behöver ingen transaction

JPA – Find & JPQL - Exempel

Find by title

```
public Todo findByTitle(String title) {

    EntityManager em = EMF.createEntityManager();
    Query query = em.createQuery(
         ”SELECT t FROM TodoDB t WHERE t.title = :titleParam”
    );
    query.setParameter(”titleParam”, title);
    return (Todo) query.getSingleResult();
}
```

Vi skriver inte till databasen, så vi behöver ingen transaction.
Exemplet är inte komplett då vi inte hanterar eventuella fel.

JPA – Find & JPQL - Exempel

Find all

```
public List<Todo> findAll() {

    EntityManager em = EMF.createEntityManager();
    Query query = em.createQuery(
         ”SELECT t FROM TodoDB t”
    );
    return query.getResultList();
}
```

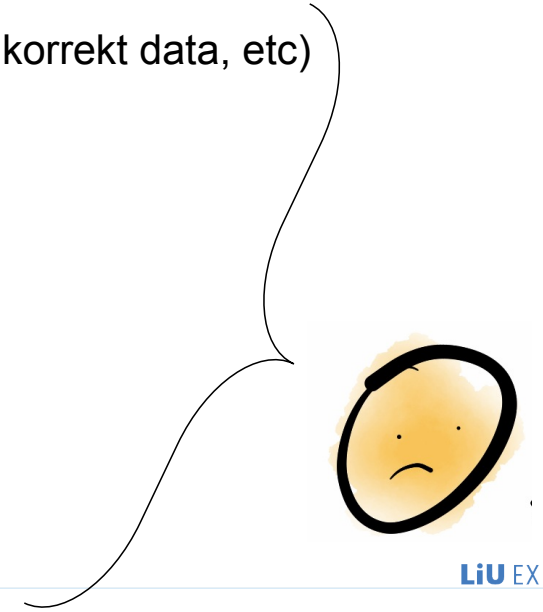
Vi skriver inte till databasen, så vi behöver ingen transaction.
Exemplet är inte komplett då vi inte hanterar eventuella fel.

Motivation & Problem

”Anything that can go wrong will go wrong”

- Murphy’s Law

- Data -> Databas (Nätverk, inkorrekt data, etc)
- Databas -> Data (Nätverk)
- Jordbävning i serverhall ?
- Strömavbrott ?
- ”Moth in relay” ?



JPA - Exceptions

- Ibland avbryts exekveringen pga ett fel
- Servrar kan vara nere, nätverk kan vara otillgängliga
- Ett logiskt fel, t ex samma email för flera konton
- Viktigt att hantera olika typer av fel på olika sätt
- Att bara hantera felet ”tyst”, utan att tala om att något har gått fel är oftast inte så användbart
- Vi vill tala om för anropet att något gått fel, men inte krascha systemet

JPA – Exceptions - Exempel

```
public class TodoEntityFacadeDB implements TodoEntityFacade {

public long create(String title, String body)
        throws Exception {

        if(title == null || title.isEmpty()) {
            throw new IllegalArgumentException(
                "Title can't be null or empty"
            );
        }

        …

    }
}
```

Återblick – Façade, EMF, Transactions

```
public class TodoEntityFacadeDB implements TodoEntityFacade {

    public long create(String title, String body) {
          EntityManager em = EMF.createEntityManager();

          em.getTransaction().begin();

          Todo todo = new TodoDB();
          todo.setTitle(title);
          todo.setBody(body);

          em.persist(todo);
          em.getTransaction().commit();

          return todo.getId();
    }
}
```

JPA – Exceptions - Exempel

```
… samma kod som tidigare exempel …
EntityManager em = EMF.createEntityManager();
try {
  //samma kod som i tidigare men vi stänger inte EntityManager.
} catch (Exception e) {
    … gör något med felet …
    return 0;
} finally {
    if(em.getTransaction().isActive()) {
        em.getTransaction().rollback();
    }
    em.close();
}
```

Returnera 0, då vet användaren att objektet inte har sparats i databasen och att något gått fel.

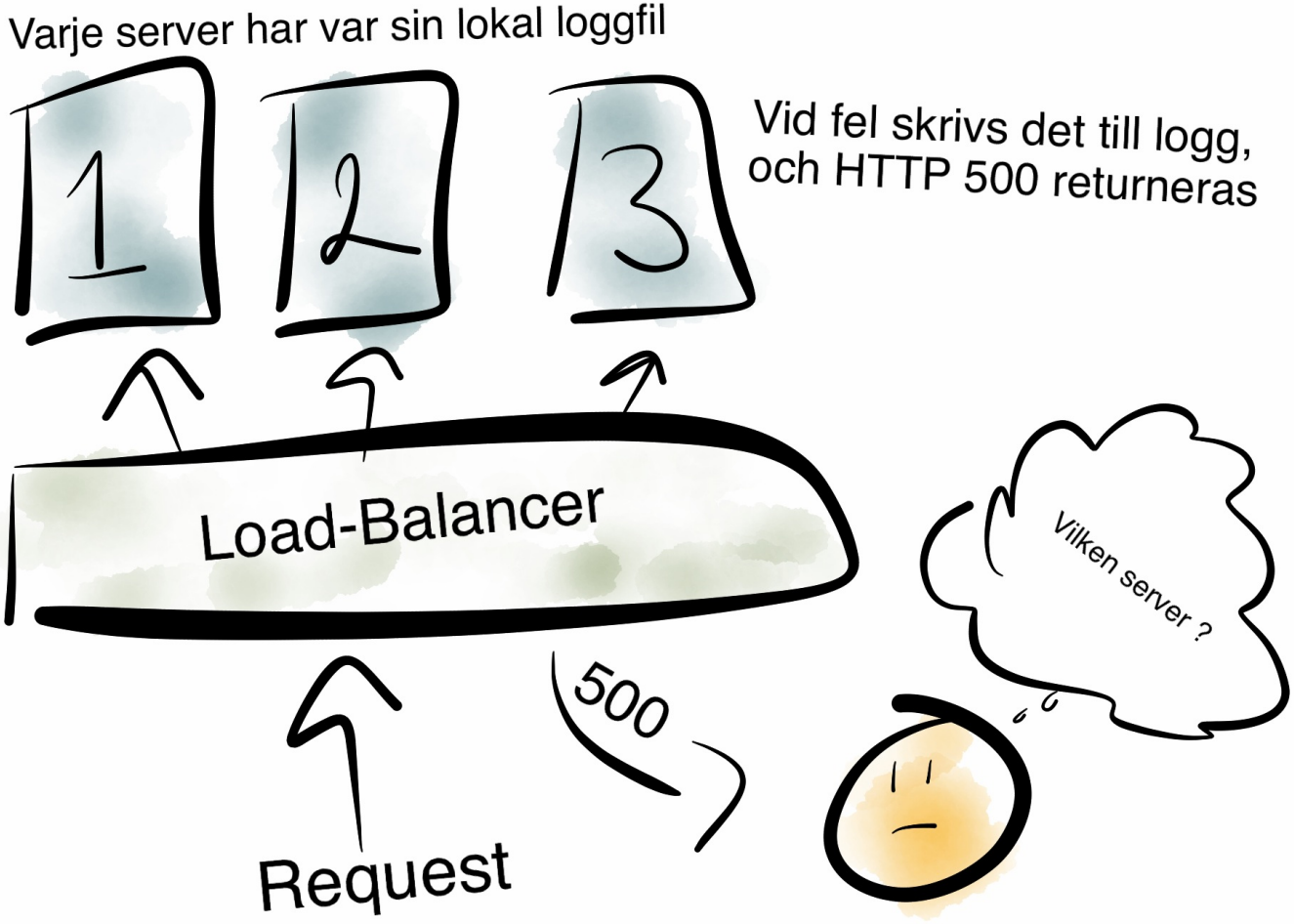
”finally” körs alltid, oavsett om fel har uppstått eller inte, det ger oss möjlighet att göra rollbacks på transaktionen och stänga vår EntityManager innan vi gör något annat.

Vi tittar på detta i ett tydligare exempel senare…

JPA - Logging

- Att logga alla anrop och fel gör att det blir enklare att hitta buggar när systemet väl är igång
- Genom att skriva till något persistent, istället för bara System.out, så kan vi gå tillbaka i tiden och även visualisera med HTML eller annat format
- Loggning kan också ske i form av mejl
- Men det finns ett problem med den modell vi visat här
- Vad händer om vår tjänst ligger bakom en load-balancer och är skalad till 20 servrar ?
- Alla anrop hamnar på olika servrar, så då har vi 20 filer som alla har loggat delar av en användares väg genom systemet

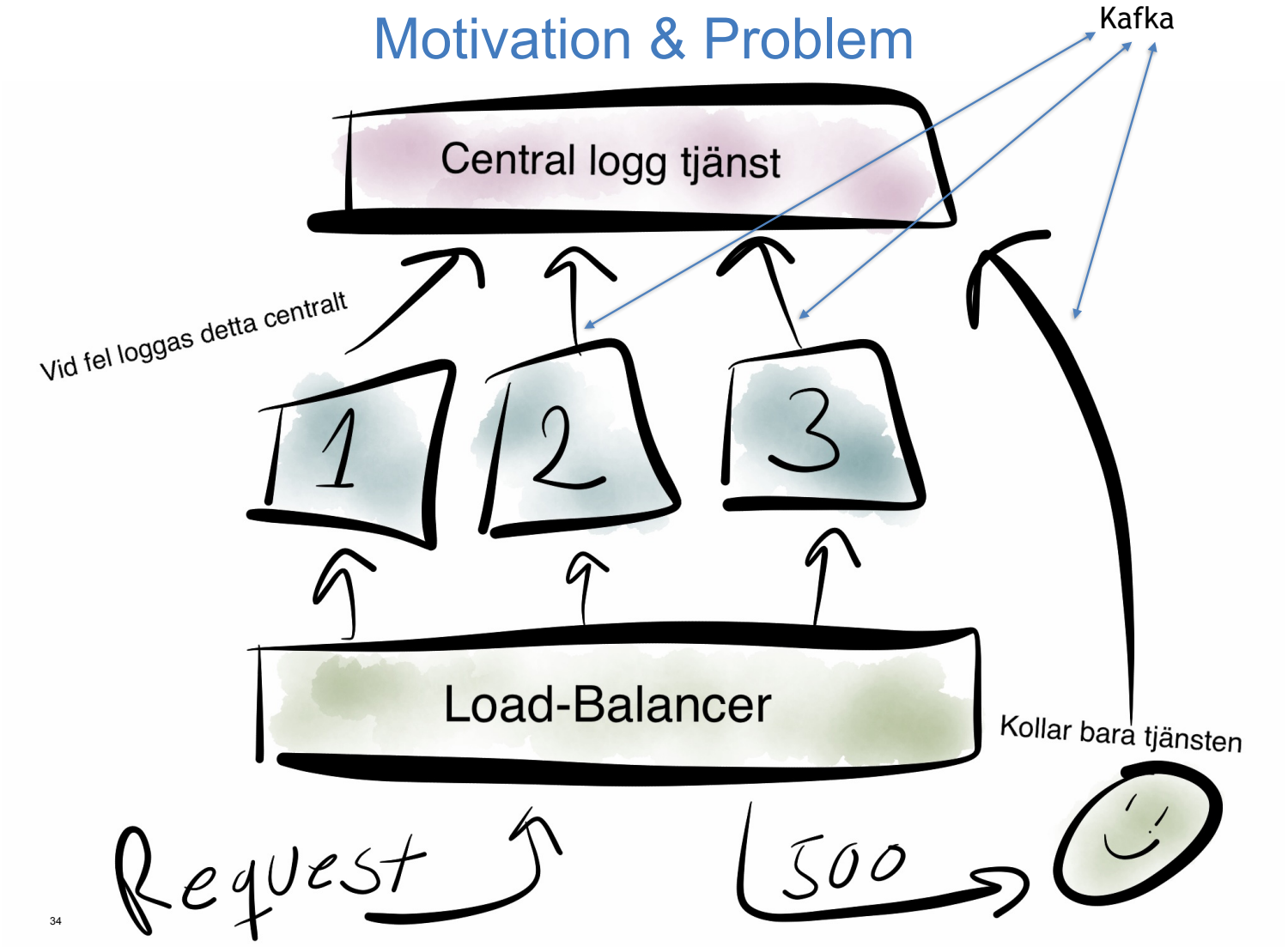
Motivation & Problem



Distribuerad Loggning

- Istället bör man komplettera sin vanliga loggning med distribuerad loggning
- T ex en tjänst som tar emot loggmeddelanden och ordnar dessa centralt

Motivation & Problem

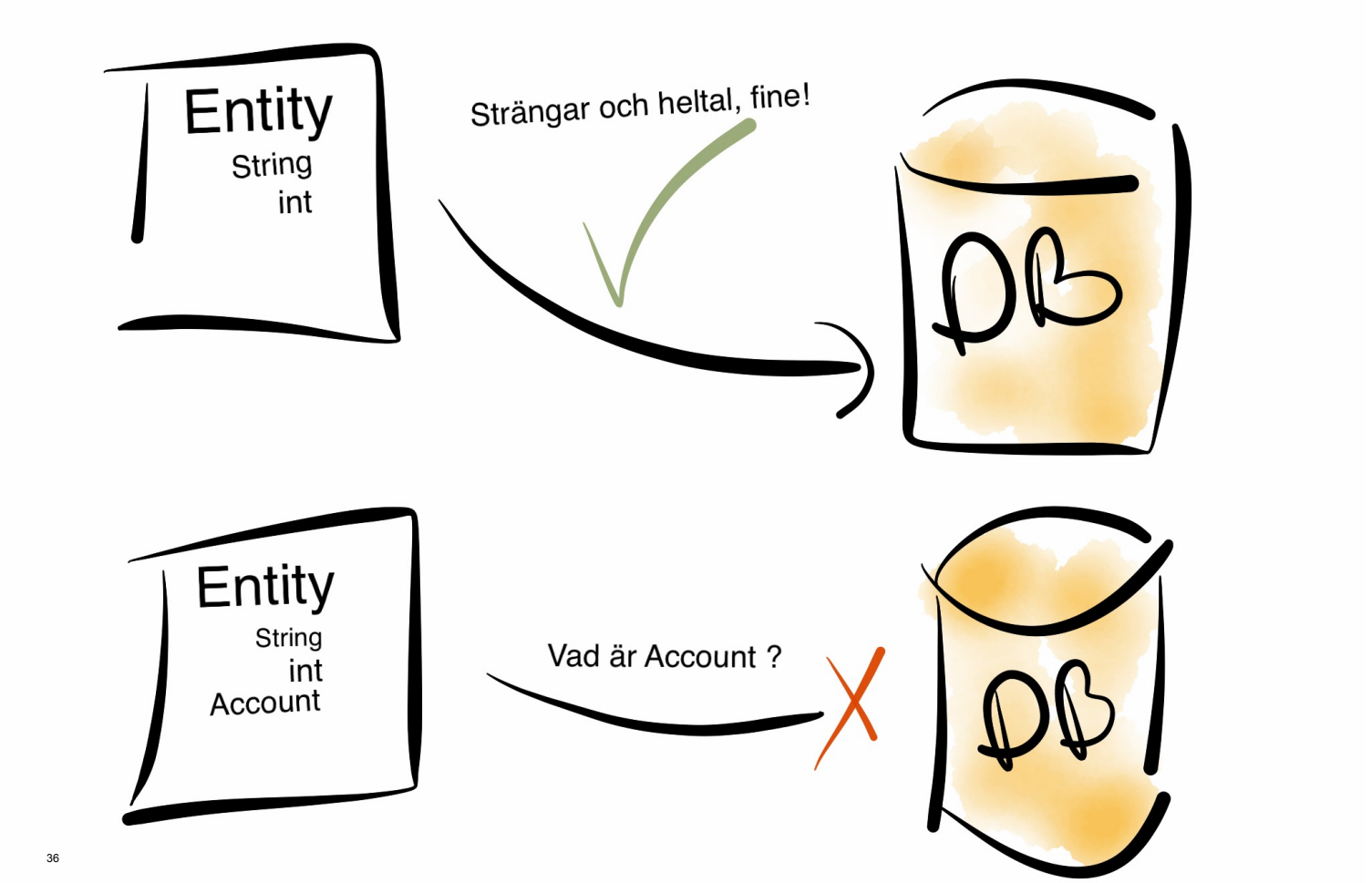


JPA - Mappings

id - bigint(20)	title - varchar(255)	content - varchar(255)
1	Title 1	Content 1
2	Title 2	Content 2
3	Title 3	Content 3
…	…	…
…	…	…

Vår första Entity skapade en tabell som ovan, den innehåll bara strängar och heltal …

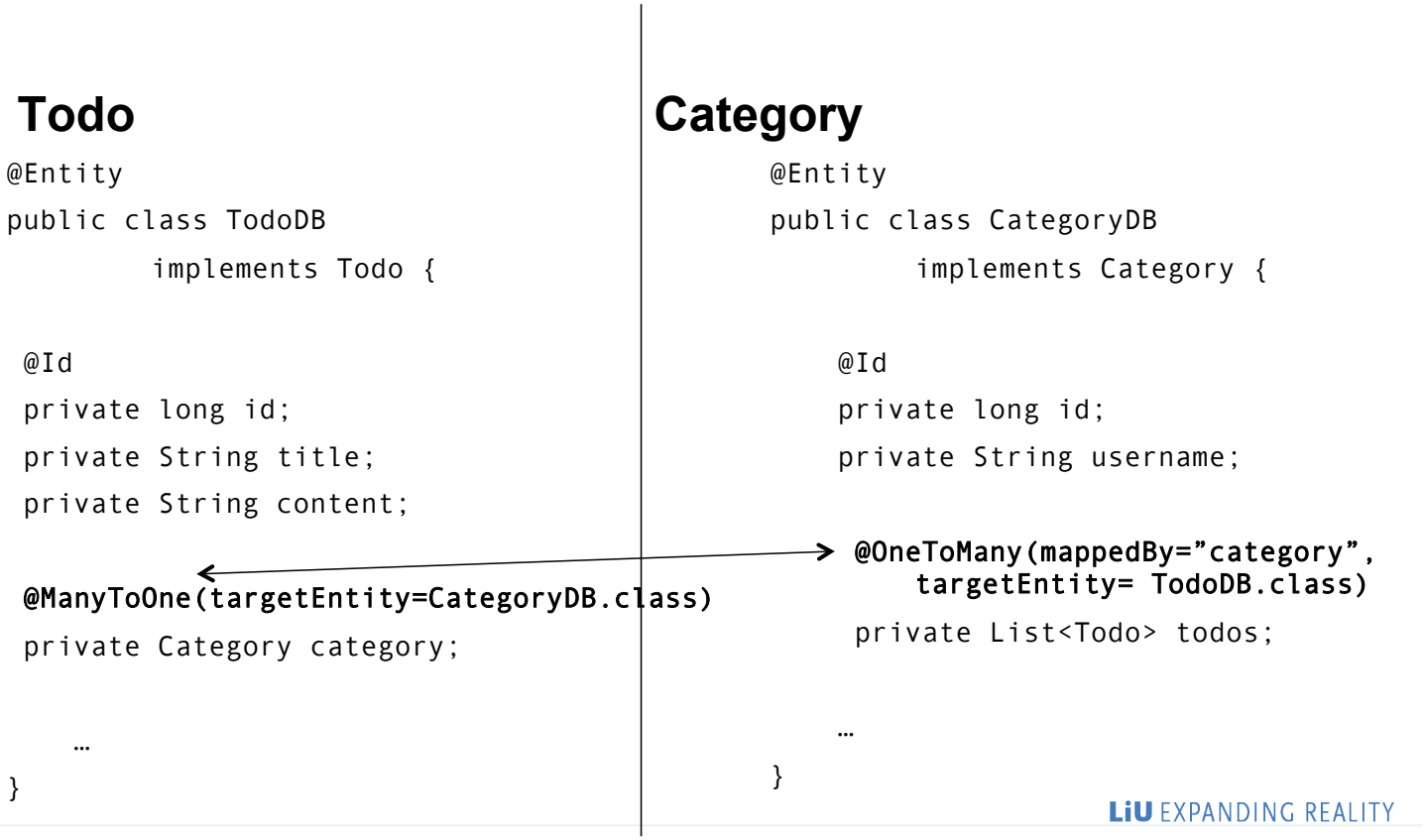
Motivation & Problem

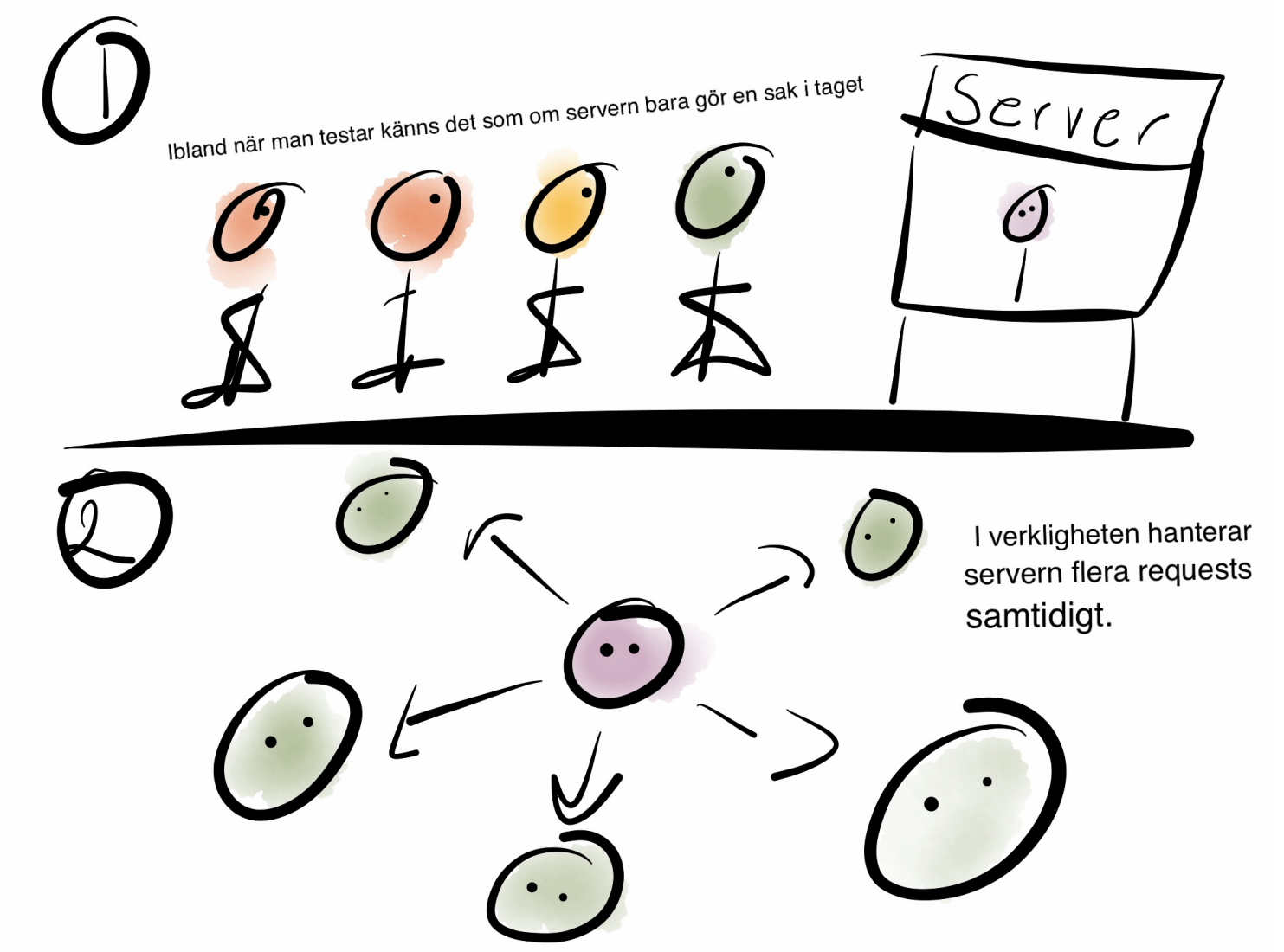


JPA - Mappings

- Komplexa förbindelser, Todo <-> Account
- @OneToMany
  @OneToOne
  @ManyToOne
  @ManyToMany
- Väldigt viktigt med utförliga testfall för att förstå hur dessa ”mappings” påverkar varandra och databasens uppbyggnad
- Det finns många inställningsmöjligheter med mappings, och det är ofta viktigt att testa en inställning, titta på hur databasen skapas för detta och ta ett beslut om man vill använda den strukturen
- Skriv testfall som testar vad som händer om man tar bort, lägger till, uppdaterar objekten i en relation

JPA – Mappings exempel




1
Ibland när man testar känns det som om servern bara gör en sak i taget
Server
2
I verkligheten hanterar servern flera requests samtidigt.

JPA - Concurrency

- Våra tjänster har inte bara en tråd
- Räkna med att för varje ny användare skapas en ny tråd, och då ofta en ny EntityManager
- Vad händer om man försöker köra samma funktion samtidigt, t ex uppdatera en Todo ?

- ***Pessimistisk**:* Lås hela raden som skall ändras, ingen annan för röra den tills tråden är klar, lås upp när vi är klara
- ***Optimistisk**:* Hämta objekt och ändra som vanligt, men om objektet har ändrats i databasen när vi försöker skriva så fallerar funktionen

JPA – Concurrency Exempel

```
…
public long update(long id, String title, String body) {

    …
      Todo todo = em.find(TodoDB.class, id, LockModeType.PESSIMISTIC_WRITE);
    …
}
```

För andra varianter av LockModeType kolla länkar under kurslitteratur på hemsidan

JPA – Concurrency – Klassiskt exempel

```
(OBS: Pseudo kod)

function debit(amount) {

   Account account = findAccount();

   if(account.credit >= amount) {
      account.credit = account.credit – amount;
      persistToDatabase(account);
      return true;
   } else {
      return false;
   }

}
```

Antag att Account bara har 100kr på sitt konto, och
vi får in två requests samtidigt med en debitering på 100kr

JPA – Concurrency – Klassiskt exempel

(OBS: Pseudo kod)

Tråd 2 väntar….

```
function debit(amount) {

   Account account = findAccount();

   if(account.credit >= amount) {
      account.credit = account.credit - amount;
      persistToDatabase(account);
      return true;
   } else {
      return false;
   }

}
```

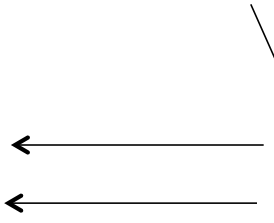

Tråd 1

Antag att Account bara har 100kr på sitt konto, och
vi får in två requests samtidigt med en debitering på 100kr

JPA – Concurrency – Klassiskt exempel

```
(OBS: Pseudo kod)

function debit(amount) {

   Account account = findAccount();

   if(account.credit >= amount) {
      account.credit = account.credit - amount;
      persistToDatabase(account);
      return true;
   } else {
      return false;
   }

}
```

CPU'n har valt att arbeta med tråd 2, tråd 1 står därmed still.



Tråd 2 debiterar och skriver till, databasen. Account har nu 0kr på sitt konto.

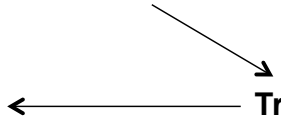
```
Antag att Account bara har 100kr på sitt konto, och
vi får in två requests samtidigt med en debitering på 100kr
```

JPA – Concurrency – Klassiskt exempel

```
(OBS: Pseudo kod)

function debit(amount) {

   Account account = findAccount();

   if(account.credit >= amount) {
      account.credit = account.credit - amount;
      persistToDatabase(account);
      return true;
   } else {
      return false;
   }

}
```

Tråd 2 är klar…

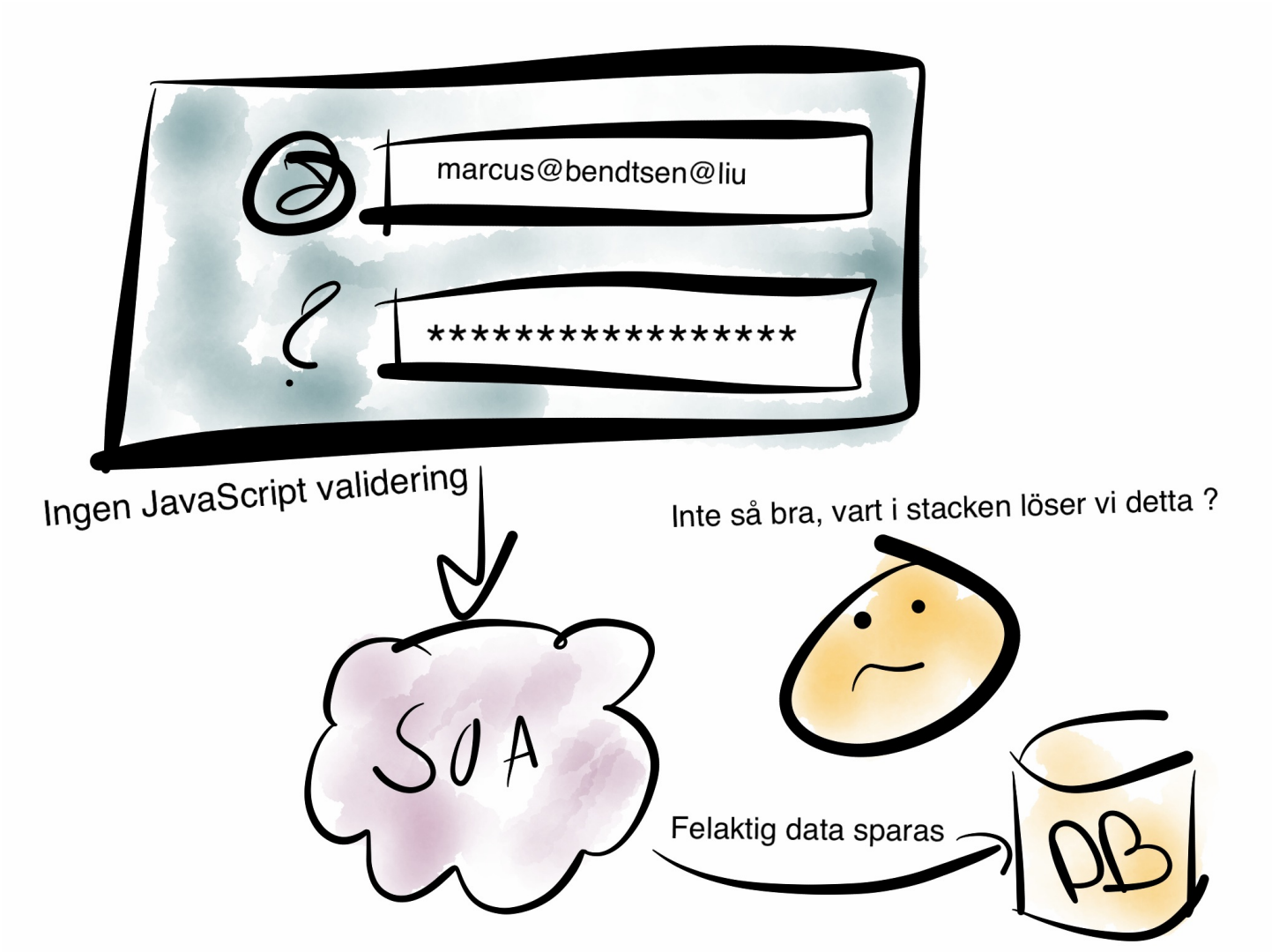
Tråd 1 har redan kollat att account hade >= 100kr, så den kör på.



Tråd 1

När Tråd 1 är klar har vi debiterat, 200 kr från account (som bara hade 100kr) och account.credit == 0

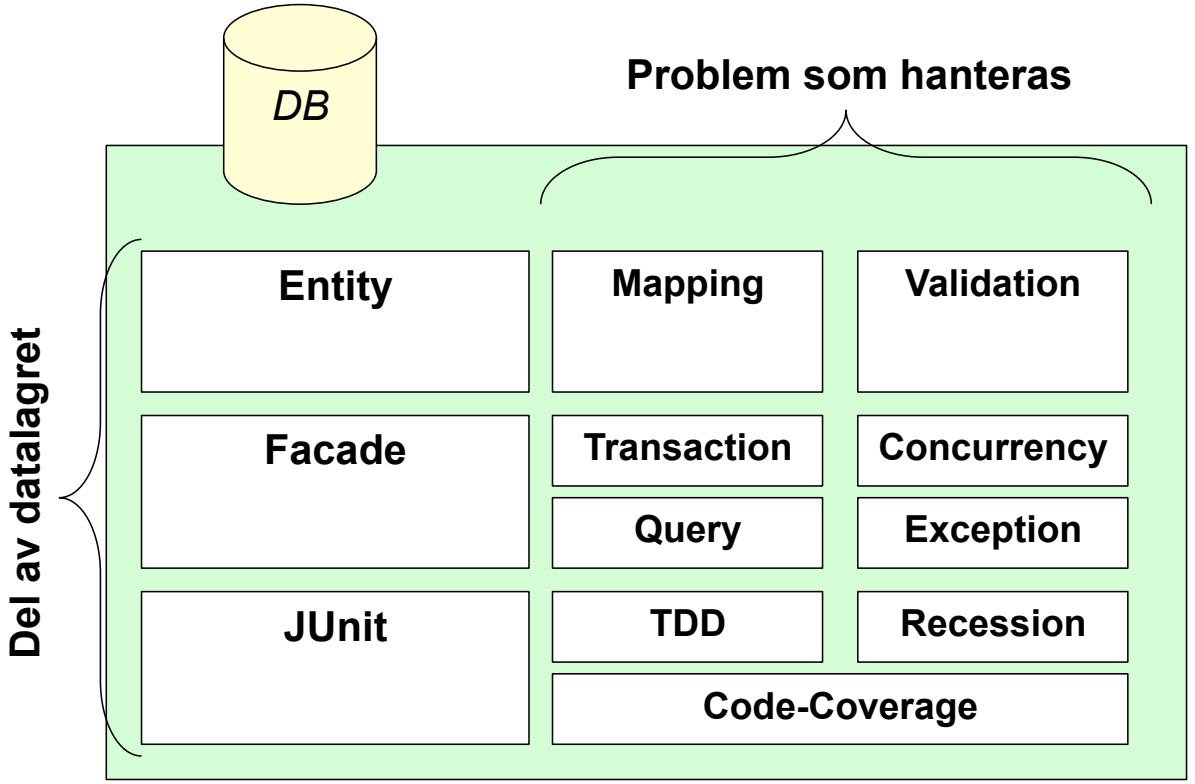
```
Antag att Account bara har 100kr på sitt konto, och
vi får in två requests samtidigt med en debitering på 100kr
```


marcus@bendtsen@liu
****
Ingen JavaScript validering
SOA
Inte så bra, vart i stacken löser vi detta ?
Felaktig data sparas
DB

JPA - Validering

- Det finns bibliotek som del av Java EE som löser detta problem.
- Man kan automatiskt validera email, siffor, max o min, godtyckliga strängar etc.
- Biblioteket kallas Bean Validation.
- Vi kommer inte att ta på oss problemet med validering i denna kurs, men det är naturligtvis ett viktigt problem i större system.

Datalayer - DONE




LINKÖPINGS UNIVERSITET
LINKÖPINGS UNIVERSITET
Linköpings universitet
expanding reality
www.liu.se