Praktycy dla Praktyków
Szkolenia i doradztwo

JPA and Modeling

Advanced topics

bottega IT SOLUTIONS

Agenda

- Modelling: OO, DDD
  - JPA impl
- Application architecture
  - DAO, Repository, testability, security
- Performance
  - n+1 Select Problem, Lazy Loading, Optimal mapping, cache, SQL,
- Optimistic/pesymistic locking, transactions
- Gotchas, gotchas, gotchas

The real model

You will know:

- Modeling Building Blocks
- BB persistence Implementation

You will learn:

- How to model real objects and design boundaries – consistent units of change
- Basic Building Blocks of Domain Driven Design
  - Entity, Aggregate, Value Objects
- How to impl. OO style code with JPA
- How to impl. identity
- How to deal with inheritance

bottega IT SOLUTIONS

Encapsulation

```
human.getDigestionSystem().
   getPeritoneum().getStomach().
      add(new Sausage(2));
```

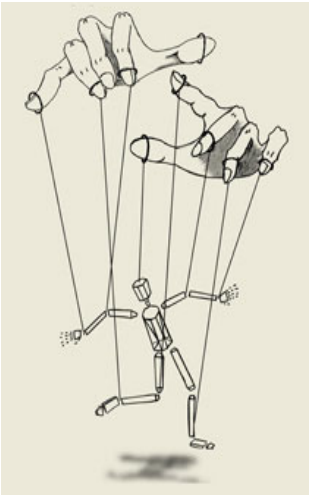
```
human.eat(new Sausage(2));

public void eat(Food f){
  if (! iLike(f))
       thow new IDontLikeItException(f);
  this.digestionSystem.swallow(f);
}
```

Layers of responsibility

User interface

- Presentation

Application logic



- Thin layer – coordination and technical stuff, models use cases

Domain logic

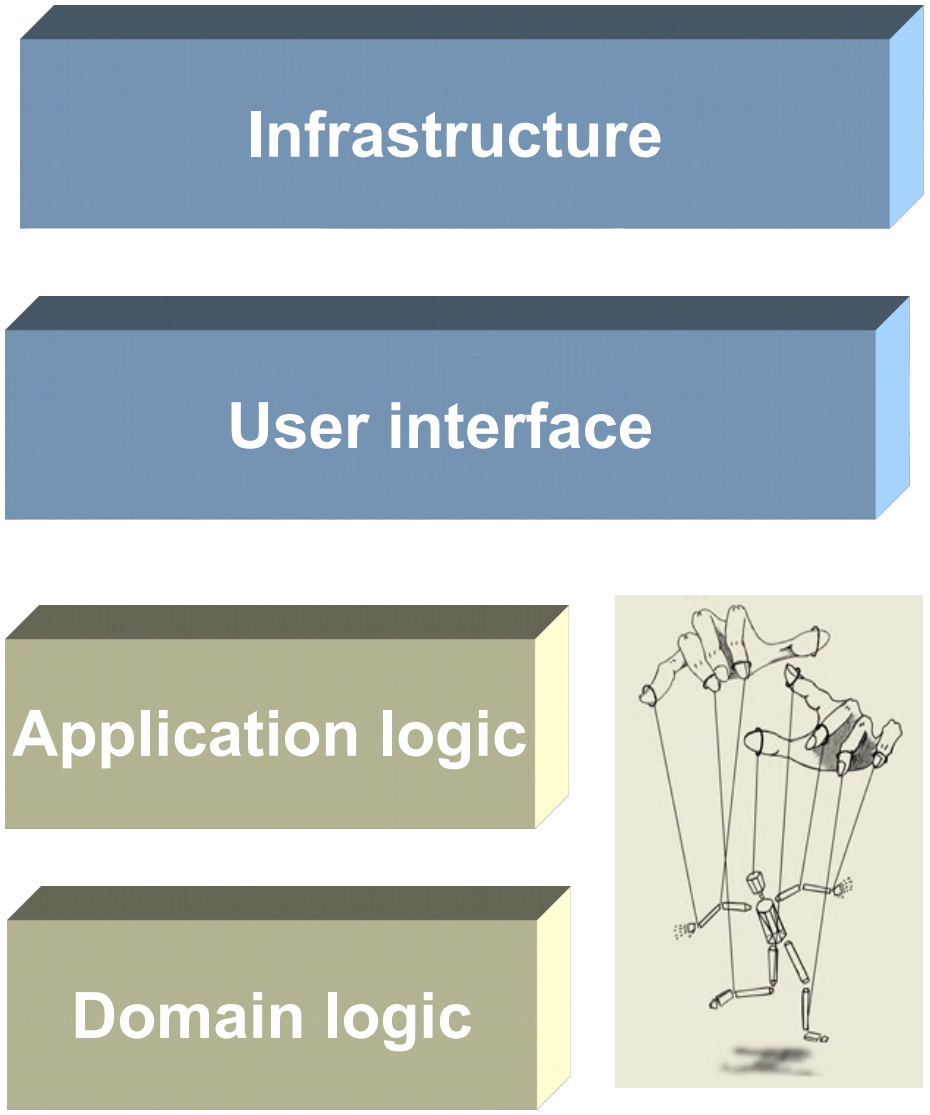
- Domain model (behaviour and rules) – heart of the system

Infrastructure

- Technical capabilities

•http://a.images.blip.tv/Unclephilms-puppetTest1168.jpg

Layers of responsibility - DIP

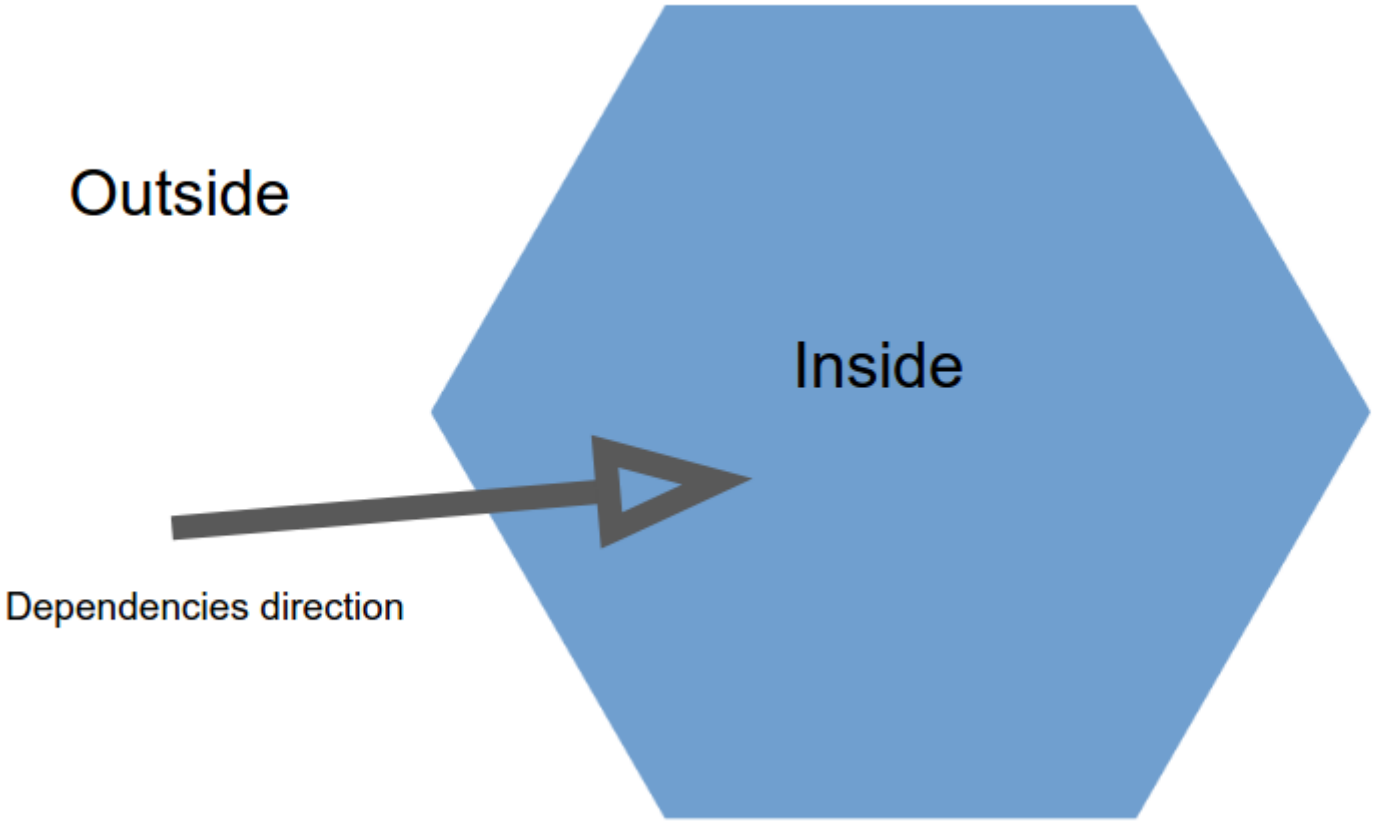


- Infrastructure depends on all other layers
- Infrasturture implements abstractions defined in all other layers
- Other layers define abstractions (e.g. repository)

•http://a.images.blip.tv/Unclephilms-puppetTest1168.jpg

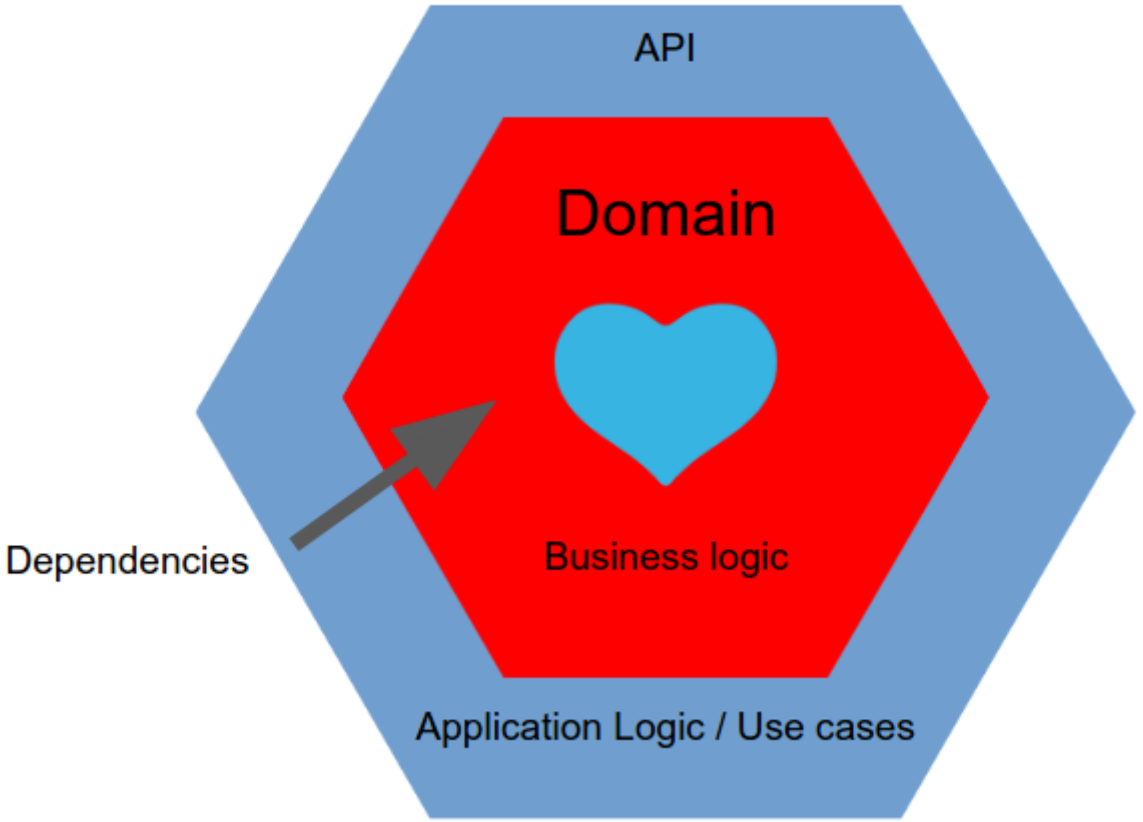
bottega
IT SOLUTIONS

Hexagonal Architecture

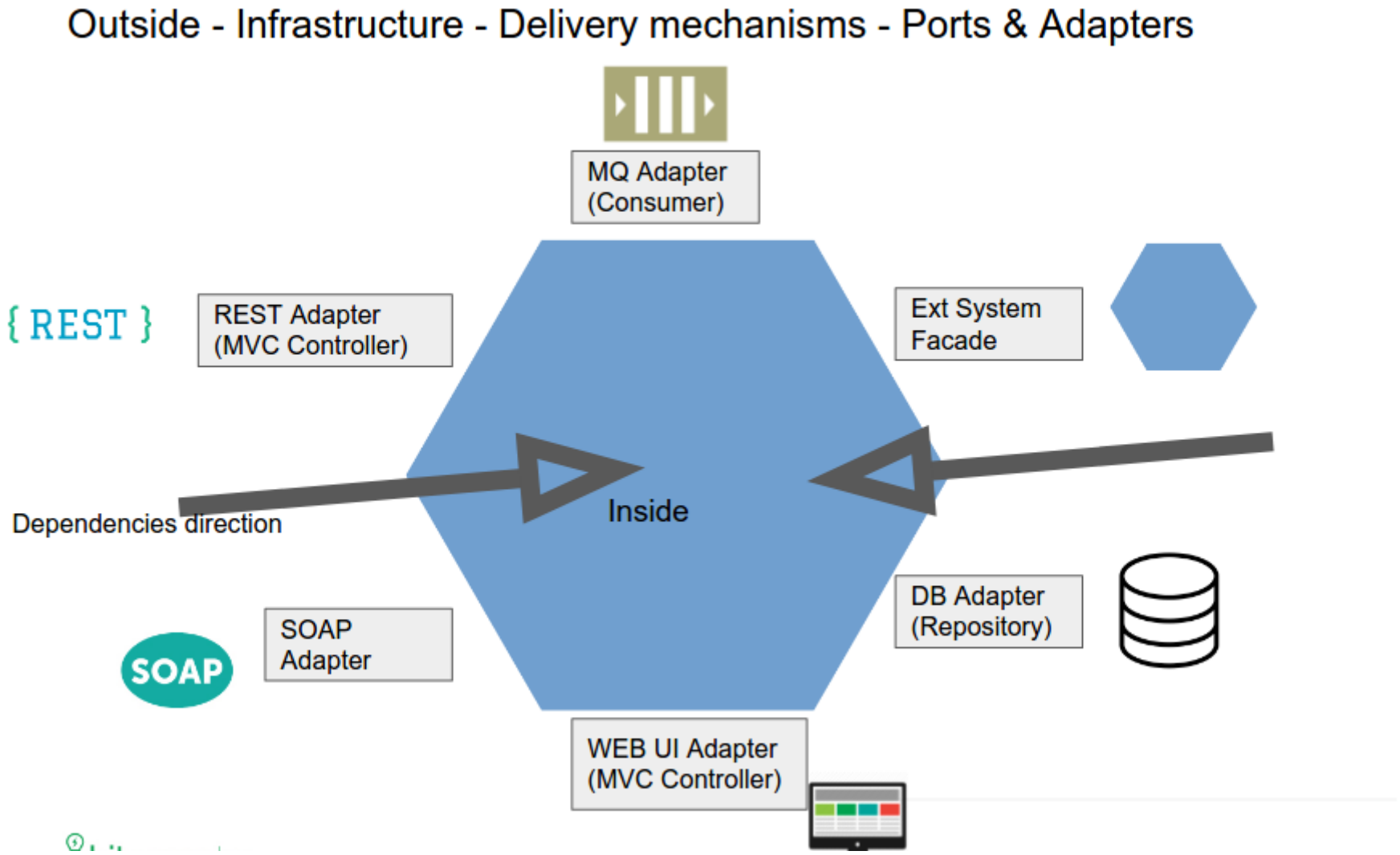


Hexagonal Architecture

The Inside



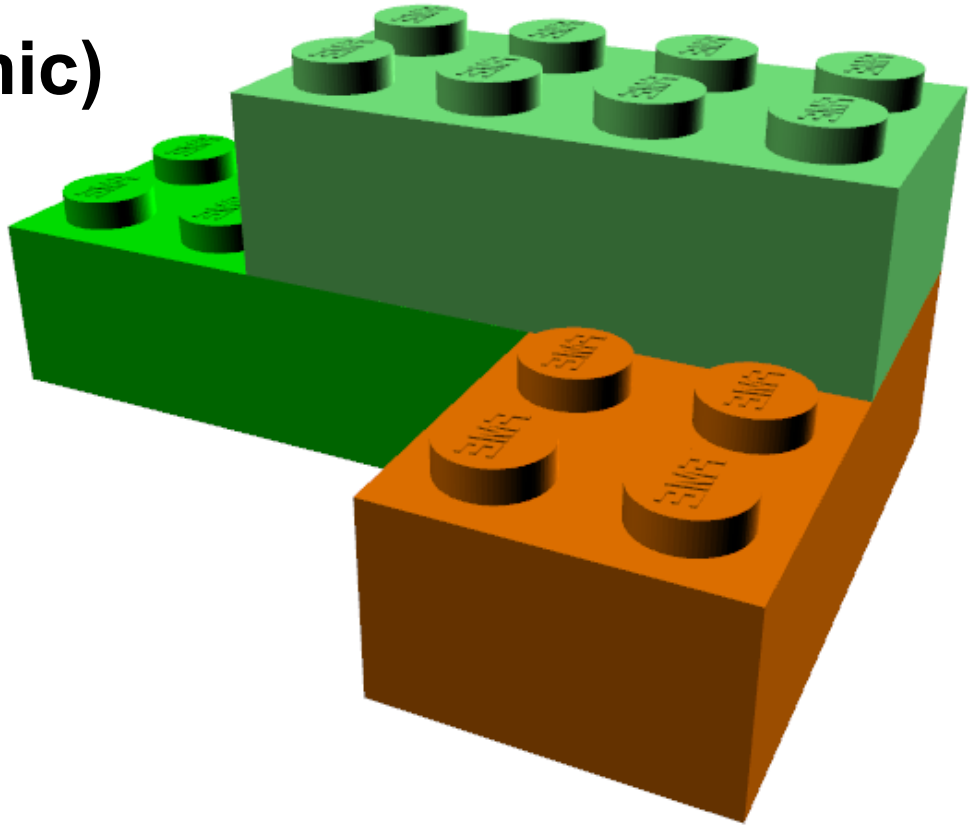
Hexagonal Architecture



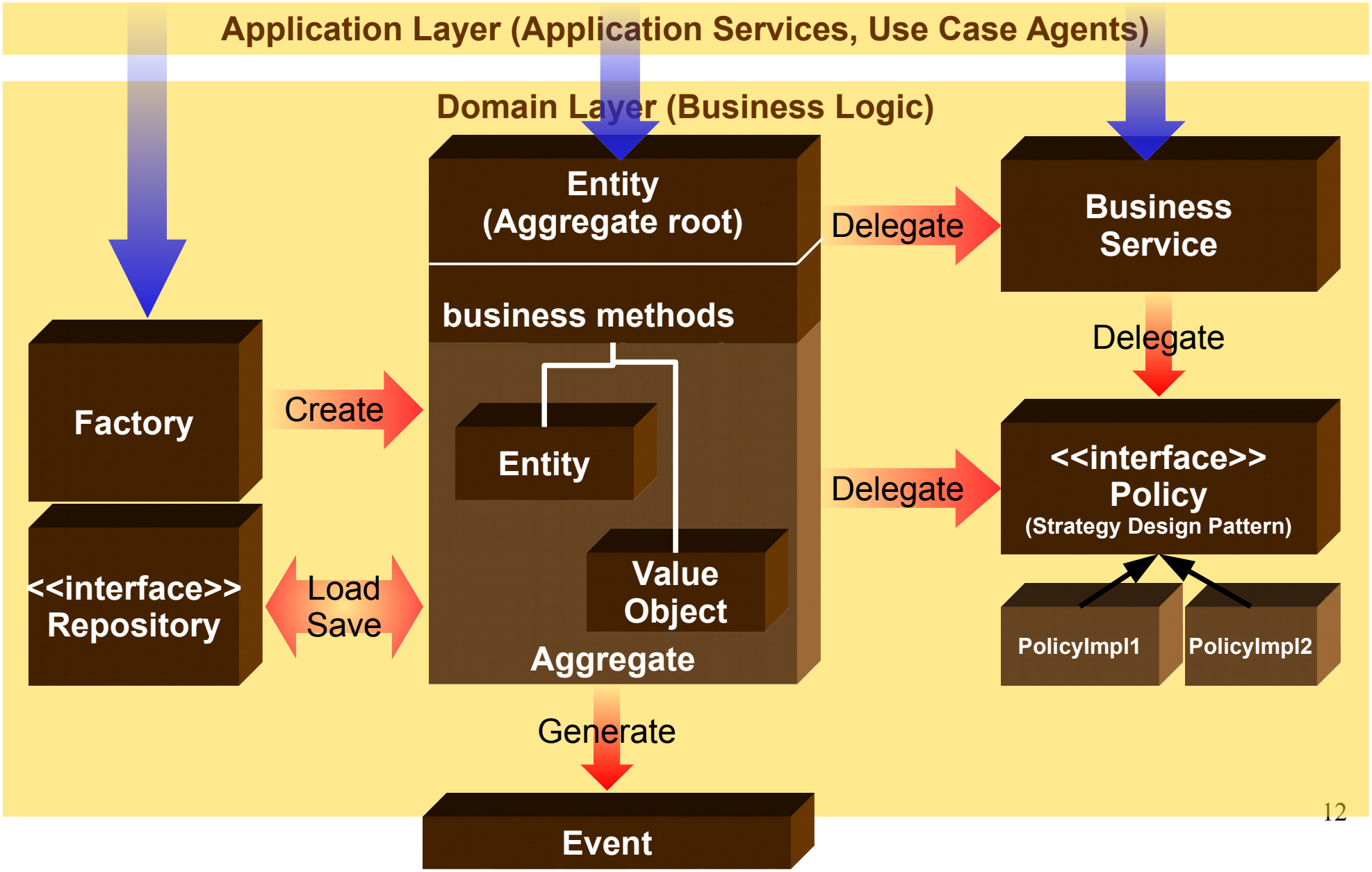
bottega IT SOLUTIONS

Domain Logic Building Blocks

- **Entities (rich, not anaemic)**
- **Value Objects**
- **Aggregates**
- Services (business)
- Policies
- Specifications
- Business Events + Sagas
- Factories
- **Repositories**



Building Blocks cooperation



bottega IT SOLUTIONS

Application Service Sample

```
public class PurchaseService{

    public void addProduct(Long orderId, Long productId, int quantity){
        Product product = productsRepository.load(productid);
        Order order = ordersRepository.load(orderId);

        RebatePolicy rebatePolicy = ...
        order.addProduct(product, quantity, rebatePolicy);

        ordersRepository.save(order);
        appEventsManager.fire(new ProductAddedEvent(product.getId()));
    }

    public void submit(Long orderId, Payment payment){
        Order order = ordersRepository.load(orderId);

        order.submit(payment);
         TaxPolicy taxPolicy = ...
        Invoice invoice = bookKeeper.issue(order, taxPolicy);

        ordersRepository.save(order);
        invoicesRepository.save(invoice);
    }
}
```

Repository

- Repository: Data source abstraction
  - manages Aggregates and Entities
    - loads by ID
    - just business queries, no (dozens) search methods
  - decoupling of (potentially many) data sources and assembling
    - also injecting into rich domain model
- Data Access Object
  - originally associated with table
  - CRUD + dozens of search methods

bottega IT SOLUTIONS

Repository

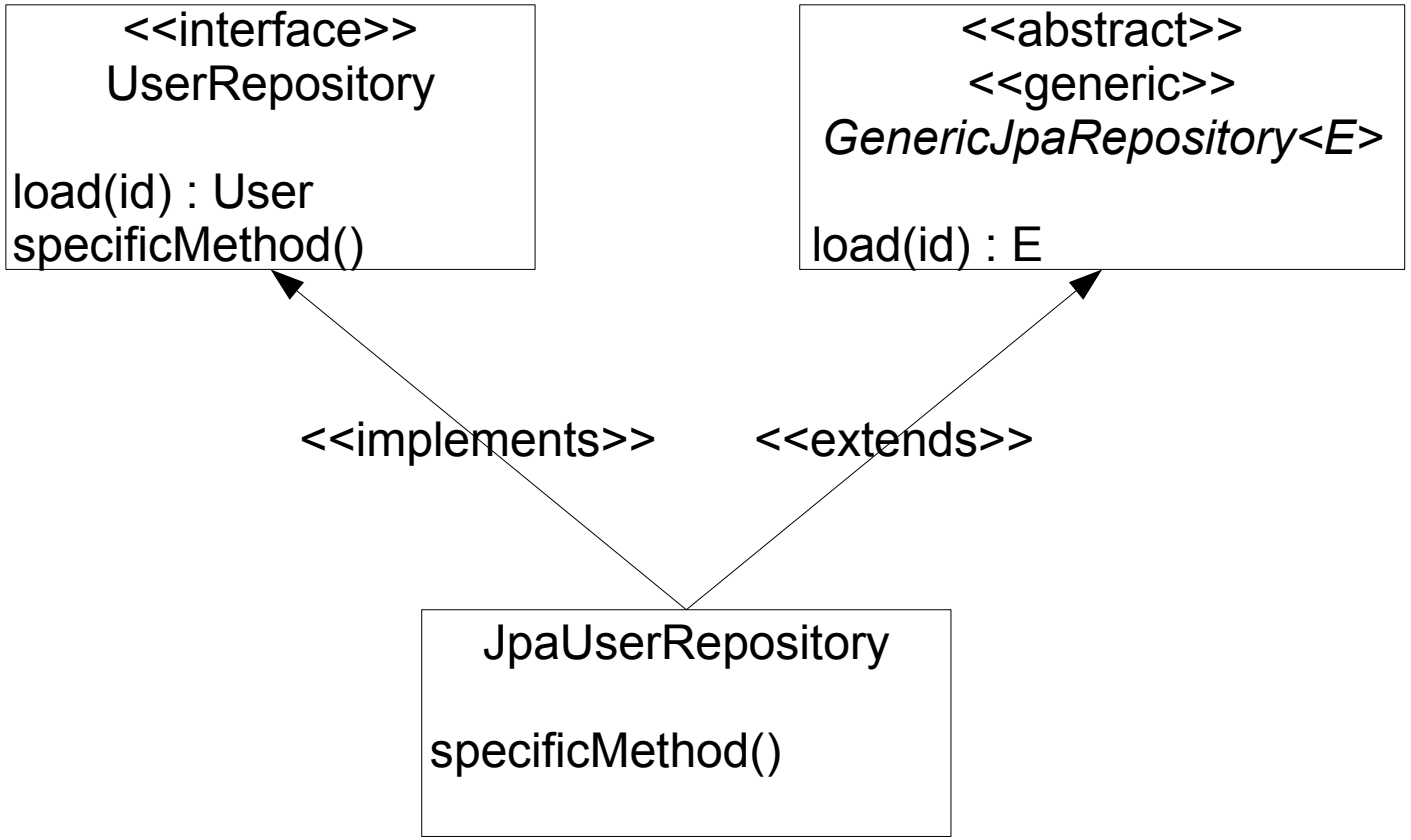
```
public class JpaUserRepository implements UserRepository{
    @PersistenceContext protected EntityManager entityManager;

    public User load(Long id) {
        User user = entityManager.find(User.class, id);
        if (user == null)
         throw new RuntimeException("User " + clazz.getCanonicalName() + " id = " + id
                                        + " does not exist");
        return user;
    }

    public void save(User user) {
         if (! entityManager.contains(user)){
             entityManager.persist(user);
        //else: merge – if we plan do detach objects
          }
    }

    public void delete(Long id){
         User user = load(id);
         entityManager.remove(user)
    }
}
```

bottega IT SOLUTIONS

Generic Repository

```
public abstract class GenericJpaRepository<E> {
    @PersistenceContext protected EntityManager entityManager;
    private Class<E> clazz;
    @Inject private AutowireCapableBeanFactory spring;

    @SuppressWarnings("unchecked")
    public GenericJpaRepository() {
        this.clazz = ((Class<A>)   ((ParameterizedType)getClass()
         .getGenericSuperclass()).getActualTypeArguments()[0]);
    }

    public E load(Long id) {
       E entity = entityManager.find(clazz, id);
        if (entity == null)
         throw new RuntimeException("Entity " + clazz.getCanonicalName() + " id = " + id
                                    + " does not exist");
        return entity;
    }

    public void save(E entity) {
         if (! entityManager.contains(entity)){
             entityManager.persist(entity);
    }

    public void delete(Long id){
         E entity = load(id);
         entityManager.remove(entity)
    }
}
```

Pattern


<<interface>>
UserRepository
load(id) : User
specificMethod()
<<abstract>>
<<generic>>
GenericJpaRepository<E>
load(id) : E
<<implements>>
<<extends>>
JpaUserRepository
specificMethod()

bottega IT SOLUTIONS

Concrete repository

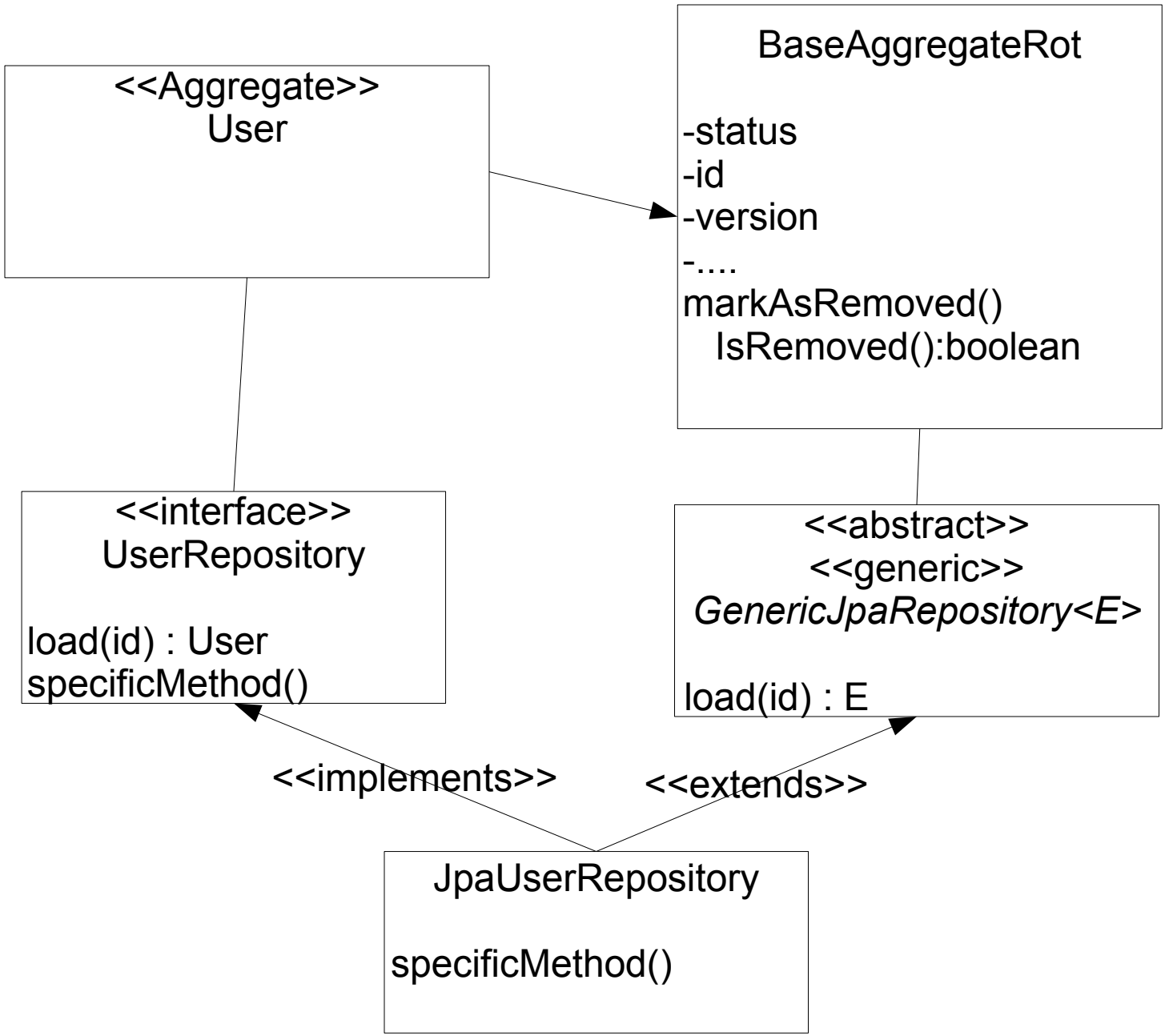
```
public JpaUserRepository extends GenericRepository<User>
    implements UserRepository {

 //additional, specific method

}
```

bottega IT SOLUTIONS

Base Repository + Base Entity

```
public abstract class GenericJpaRepository<A extends BaseAggregateRoot> {
    @PersistenceContext protected EntityManager entityManager;
    private Class<A> clazz;
    @Inject private AutowireCapableBeanFactory spring;
    @SuppressWarnings("unchecked")   public GenericJpaRepository() {
        this.clazz = ((Class<A>) ((ParameterizedType) getClass().getGenericSuperclass()).getActualTypeArguments()[0]);
    }
    public A load(AggregateId id) {
        A aggregate = entityManager.find(clazz, id);
        if (aggregate == null) throw new RuntimeException("Aggregate " + clazz.getCanonicalName() + " id = " + id + " does
not exist");

        if (aggregate.isRemoved())
         throw new RuntimeException("Aggragate + " + id + " is removed.");

        spring.autowireBean(aggregate);
        return aggregate;
    }
    public void save(A aggregate) {
     if (! entityManager.contains(aggregate)){
          entityManager.persist(aggregate);
     }
    }

    public void delete(AggregateId id){
         A entity = load(id);
         entity.markAsRemoved();
     }
}
```

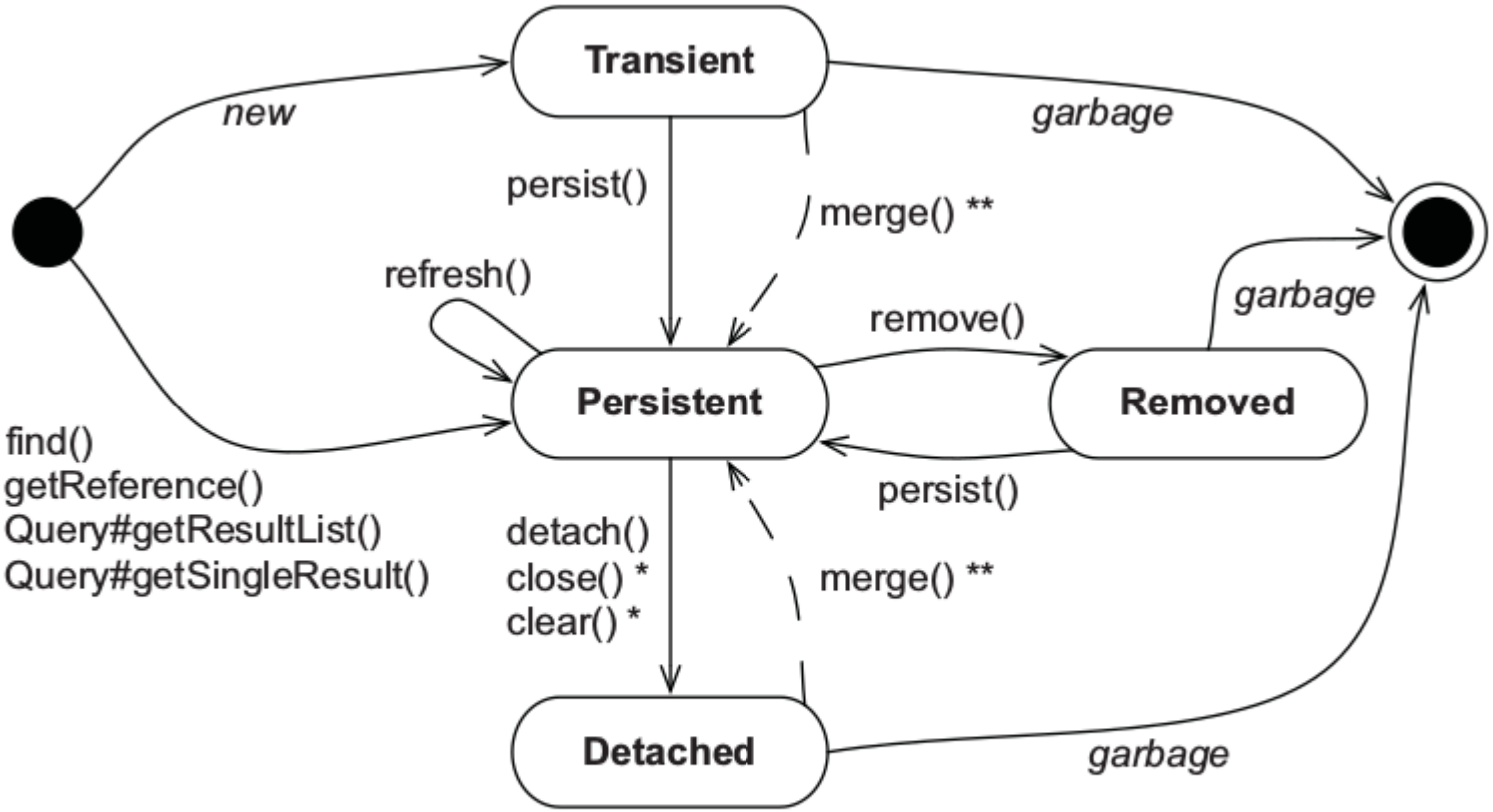
Pattern


<<Aggregate>>
User
BaseAggregateRot
-status
-id
-version
-....
markAsRemoved()
IsRemoved():boolean
<<interface>>
UserRepository
load(id) : User
specificMethod()
<<abstract>>
<<generic>>
GenericJpaRepository<E>
load(id) : E
<<implements>>
<<extends>>
JpaUserRepository
specificMethod()

How to save?

- Session.save()
  - inserts immediately if ID generator needs DB access (e.g. "identity" generator, not "sequence")
  - no matter if in transaction or not
    - not good for long running conversations
- EntityManager.persist()
  - does not guarantee ID assignment immediately
    - can happen during flush (or before any SELECT)
  - will not be executed outside transaction
    - good for long running conversations
- EntityManager.merge() - allows to merge detached entity
  - when entity with the same ID was already loaded

Entity lifecycle



* Affects all instances in the persistence context
** Merging returns a persistent instance, original doesn't change state

bottega IT SOLUTIONS

Merge fail – how to accidentally delete

- Merge loads data from DB and merges to given parameters
  - Intention: attach entities that are detached
- Fail scenario:

1. SELEC User u JOIN FETCH u.addresses a WHERE a.active = true

   As a result we have: users with some (not all) addresses

2. Return Entities from transactional Service to UI

   Entities are **detached**

3. Merge

   Addresses that were not fetched are removed!

bottega IT SOLUTIONS

How do Update smart?

- Dynamic Update
  - updates only changed properties
  - makes difference for large tables (legacy:)

```
@Entity
@org.hibernate.annotations.Entity(dynamicUpdate = true)
public class StockTransaction{
```

bottega IT SOLUTIONS

What to load?

- Repository can load Aggregate optimised per Use Case

@FetchProfiles – for more than one

```
@FetchProfile(name = "customer-with-orders",
    fetchOverrides = {
      @FetchProfile.FetchOverride(entity = Customer.class,
        association = "orders", mode = FetchMode.JOIN)
})
public class Customer {
```

```
public Customer loadCutomerWithOrders(Long customerId){
   session.enableFetchProfile( "customer-with-orders" );
   Customer customer = (Customer) session.
      get( Customer.class, customerId );
   session.disableFetchProfile( "customer-with-orders" );
```

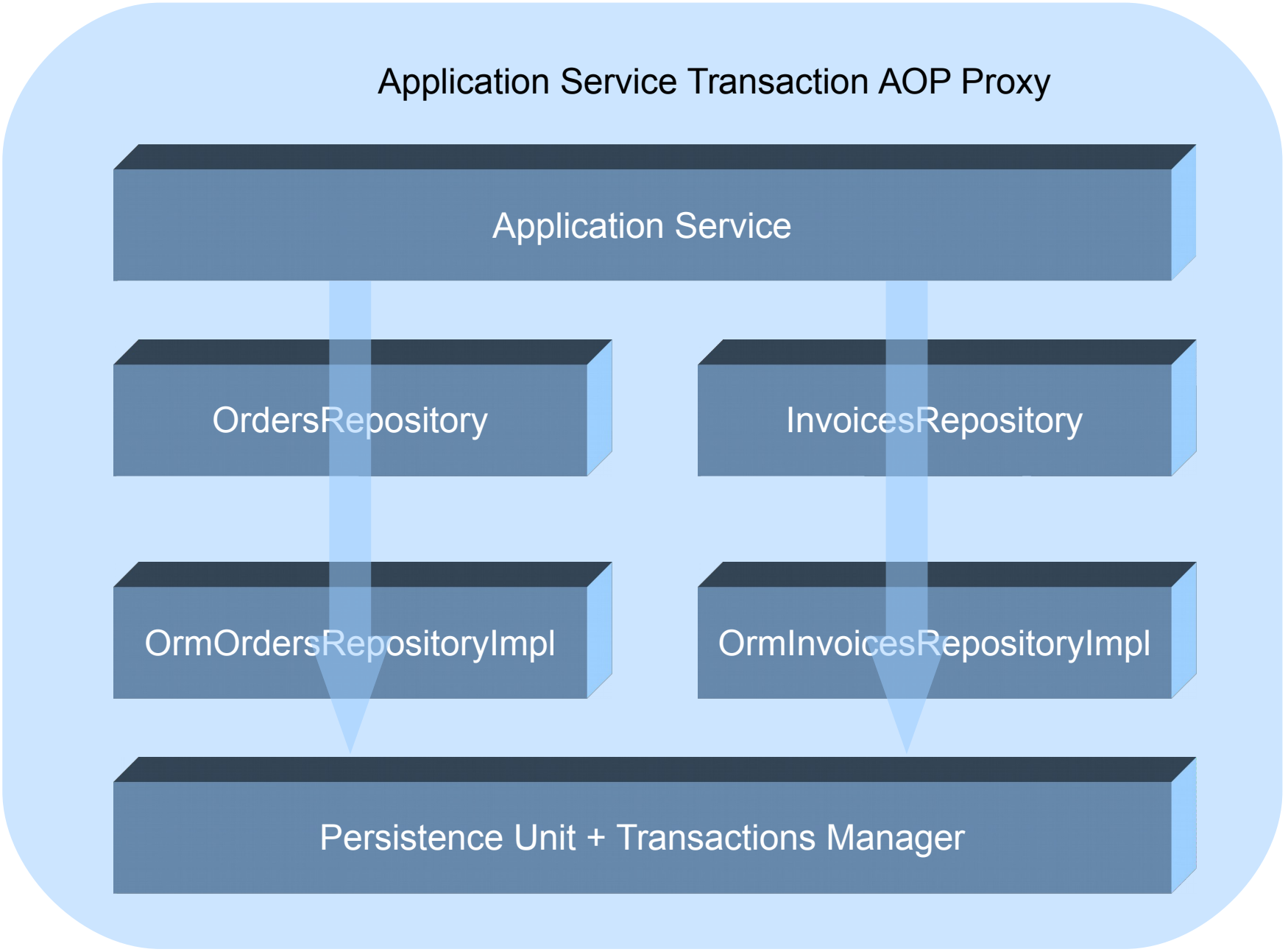
bottega IT SOLUTIONS

Always need to load?

- EntityManager.getReference()
  - Returns Proxy – access to any field (other than ID) will trigger itself to be refreshed from the database
  - coan be used on an insert or update operation
  - does not verify the existence of the object - you may get a foreign key constraint violation

```
Employee manager = em.getReference(managerId);
Employee employee = new Employee();
em.persist(employee);
employee.setManager(manager);
```

Repositories and Transactions



bottega
IT SOLUTIONS

Exercise: Repository

- Introduce Repositories for Orders and Products

bottega
IT SOLUTIONS

Aggregate

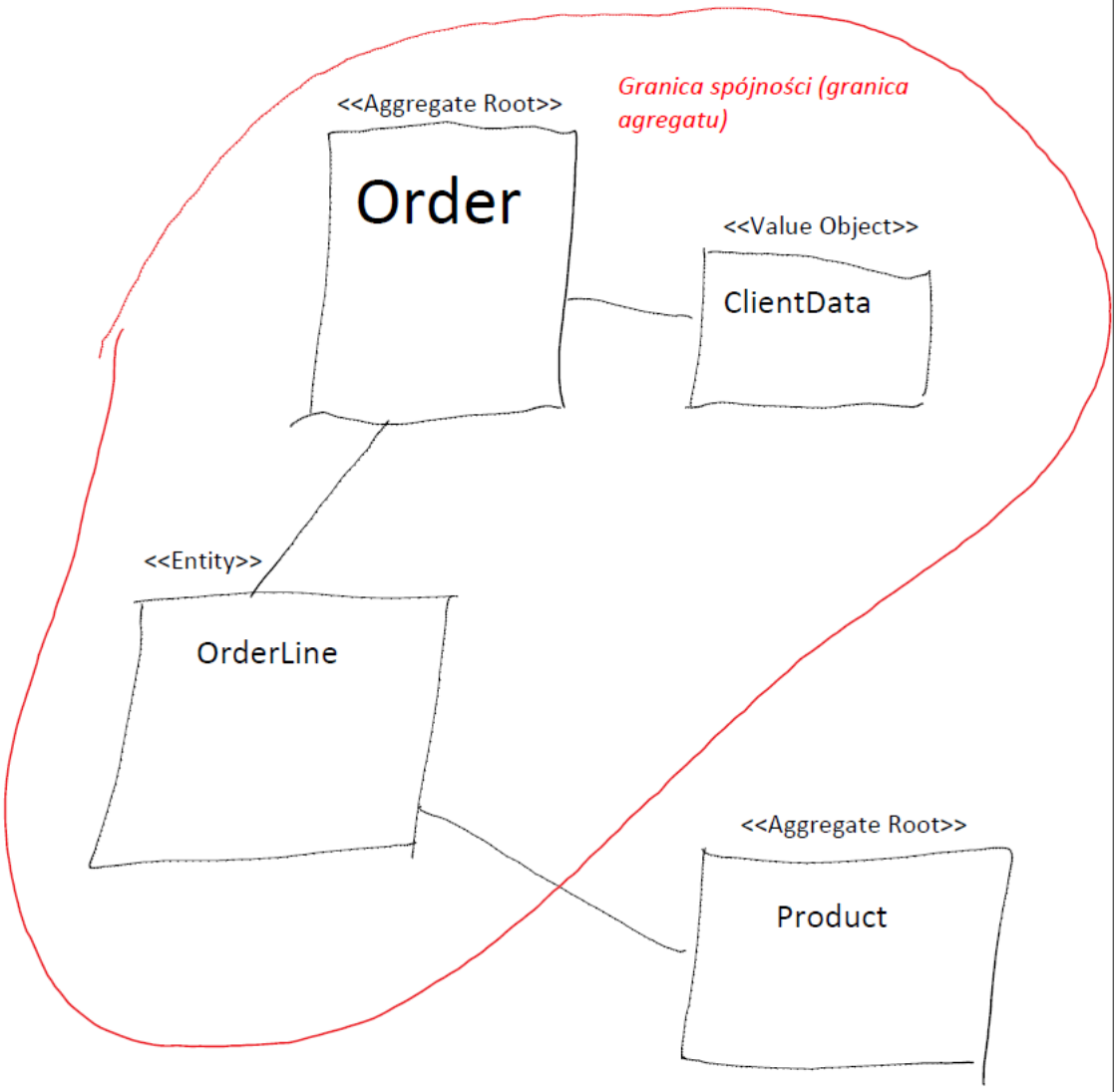
- Technically: object graph (Entity, VO)
- Main Entity - *Aggregate root*
  - Change Boundary
  - Access control - encapsulation
  - Inner objects can see each other
- Basic unit of business work

Train Wreck Code Smell

```
human.getDigestionSystem().
   getPeritoneum().getStomach().
      add(new Sausage(2));
```

```
human.eat(new Sausage(2));

public void eat(Food f){
  if (! iLike(f))
      thow new IDontLikeItException(f);
  this.digestionSystem.swallow(f);
}
```

Object with clear boundary



- Order is the root of whole structure
- It contains: OrderLines and Client Data
- But does not contain: Client and Product
  - They are different Aggregates

bottega
IT SOLUTIONS

Aggregate boundary

- Boundary
  - bad: bags of attributes
  - better: consistent unit of change per UC/US
- Bounded Context
  - IDs (business IDs) of aggregates for different BC

bottega
IT SOLUTIONS

ORM challenges

- State Encapsulation
- Lazy Loading of the aggregated objects
- Embedded Value Objects
- Association mapping
- Cascades

bottega IT SOLUTIONS

Encapsulation

- Fields annotations
  - Hibernate uses reflection
  - faster!
- Getters/setters only if makes sense
  - access – sure?
  - change – rare, use business methods

bottega IT SOLUTIONS

Encapsulation and Lazy Loading

- **Well shaped Aggregates can be Eager loaded**
- Private getters to achieve Lazy Loading
  - proxy works only on method calls
- btw: LL does not work on nullable @OneToOne
  - if null is possible than Proxy can not be set
  - solution: one element collection
    - still hermetical inside Aggregate
    - can be useful someday:)

bottega IT SOLUTIONS

Lazy Loading – time trap

What if...

- Load *root* (but *inner* objects are Lazy)
  - In the “meantime” *inner* objects are changes by another user
- Lazy load *inner* objects, but *root* is **outdated**!

Solution:

- Eager loading
- Functional approach

bottega IT SOLUTIONS

We need access to the inside...

- Aggregate transformation
  - ex. export
- Solution:
  - Builder Design Pattern
  - Read Model*

bottega IT SOLUTIONS

Value Objects

- Description of the Domain Concept
- Undistinguishable
  - VOs are the same if their attributes are the same
- *Immutable* – because no identity
  - reusable, no side effects
- Clean Code, „code smell” reduction:
  - Primitive Obsession, Data Clumps, Long Parameter List

bottega IT SOLUTIONS

Value Objects as Embeddable classes

- Usage of separate class does not force JOINs in DB

```
@Entity
public class Order{
  @Embedded
  private Money totalCost;

}
```

```
@Embeddable
public class Money{
  private BigDecimal value;
  private String currencyCode;

  //domain methods
}
```

Many VOs in the same Entity

```
@Embedded
@AttributeOverrides({
            @AttributeOverride(name = "value", column = @Column(name = "net_value")),
            @AttributeOverride(name = "currencyCode",
               column = @Column(name = "net_currencyCode")) })
private Money net;


@Embedded
@AttributeOverrides({
      @AttributeOverride(name = "value", column = @Column(name = "gros_value")),
      @AttributeOverride(name = "currencyCode",
          column = @Column(name = "gros_currencyCode")) })
private Money gros;
```

bottega IT SOLUTIONS

Power of Value Objects

- Increase expression of code – enrich domain vocabulary (Ubiquitous Language)
  - solution for Primitive Obsession code smell
  - wrappers for technical primitives
  - right level of abstraction
- Useful methods (no more Utils:)
  - validation (constructor, factory method)

bottega IT SOLUTIONS

Candidates

- Strings with restrictions
  - zip code
  - name
- Numbers with restrictions
  - percent (fraction? int?),
  - units - calculation
- Complex structures
  - money (currency, date), address (time period), time period (operations: intersection, duration)
- Projections of the Aggregate's inner state

Exercise: Aggregate

Implement Order Aggregate

- encapsulated
  - uses business methods
  - consider which setters makes domain sense
  - consider which getters makes sense (and what should be returned – value objects?)
  - consider exporting
- containing:
  - client
  - list of items
  - regular total cost
  - total cost after applying rebate
  - status (enum)

bottega IT SOLUTIONS

Cascade Operations

- Cascade operations on aggregated objects
  - PERSIST
  - MERGE – also adds aggregated
  - REMOVE
  - REFRESH – cost!
  - **ALL – good for well shaped Aggegate**
- This feature makes sense when modelling Aggregates
- Security warning!
  - When objects are sent from remote client (should be?)
    - persisting security sensitive data

```
@Entity
public class Customer{
 @ManyToOne(cascade={
   CascadeType.PERSIST,
   CascadeType.REMOVE})
 private Address addr;
}
```

Hibernate specific cascades

JPA 1 (Hibernate): DELETE_ORPHAN

- applicable for @OneToMany
- deletes entities that were removed form collection

```
@Cascade({
org.hibernate.annotations.CascadeType.SAVE_UPDATE,
org.hibernate.annotations.CascadeType.DELETE_ORPHAN})
```

JPA 2: orphanRemoval

```
@OneToMany(orphanRemoval=true, cascade={CascadeType.ALL})
private List<PhoneNumbers> phones;
```

Collection Mapping

- Hibernate semantics
  - Set: no duplicates, no order
  - SortedSet: no duplicates, order
  - Bag: duplicates, no order
    - Java: List, Collection
    - just one Bag per class – JPA can not determine doubles
  - List: duplicates, order
    - Java: List + @OrderColumn
      - name
      - base (default 0)
    - update: delete + insert
      - Hibernate does not know which entity is duplicated

bottega IT SOLUTIONS

How does it work...

	add	remove	update
bag	re-recrate: 1 x Delete + N inserts	re-create: 1 x delete + n x inserts	1 x udpate
set	load to check if qunique 1 x insert	1 x delete	1 x udpate
list	1 x insert + M x updates	1 x delete + M x updates	1 x udpate

Collection mapping

- Hibernate can not fetch=”join” on two (or even more) parallel collections (bag)
  - Hibernate can’t know which rows contain duplicates that are valid (bags allow duplicates) and which aren’t.
  - If using bag collections (they are the default @OneToMany collection in Java Persistence), don’t enable a fetching strategy that results in cartesian products.
  - Use subselects or immediate secondary-select fetching for parallel eager fetching of bag collections.

bottega IT SOLUTIONS

Reversed associations

- Order.items.add(item)
  - items must be loaded
- Item.setOrder(order)
  - reversed association – Item is the owner
  - items does **not** nave to be loaded
    - **but: Index column does not work on the owning side!**

```
@OneToMany(mappedBy="order")
private List<OrderLine> items;
```

bottega
IT SOLUTIONS

Exercise: collections

Identity

- Problem: how to implement equals() and hashCode()
  - id changes during lifetime of the object
  - new entity (null id) once put into hash-structure will be inaccessible after persisting
  - can not put more than one new entity into hash-structure
- Solution: just don't, default equals is ok
  - L1 (Session/EntityManager) cache guarantees single instance
  - What if Entity outlives Session (cross session UC)?

Identity

- Real problem: what doest it mean to be identifiable, whose responsibility is to manage ids, should JPA (DB) rule the identity of the object?
- Solution: generate ID during construction
  - UUID
    - common in asynch. systems (CqRS)

bottega IT SOLUTIONS

Self identifiable Entity

```
public abstract class AbstractPersistentObject  implements PersistentObject {
    private String id = IdGenerator.createId(); //java.util.UUID.randomUUID();
    private Integer version;

    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || !(o instanceof PersistentObject)) {
            return false;
        }
        PersistentObject other = (PersistentObject)o;
        if (id == null) return false;
        return id.equals(other.getId());
    }

    public int hashCode() {
        (id != null) ? ( return id.hashCode(); ) : ( return super.hashCode() );
    }
}
```

http://onjava.com/pub/a/onjava/2006/09/13/dont-let-hibernate-steal-your-identity.html

Hibernate: natural id

```
@Entity
public class Employee {
    @Id @GeneratedValue     Long id;

    @NaturalId
    String regon;

    @NaturalId
    String nip;
}

return session.byNaturalId( Employee.class )
        .using( "regon", "12345" )
        .using( "nip", "6789" )
        .load();
```

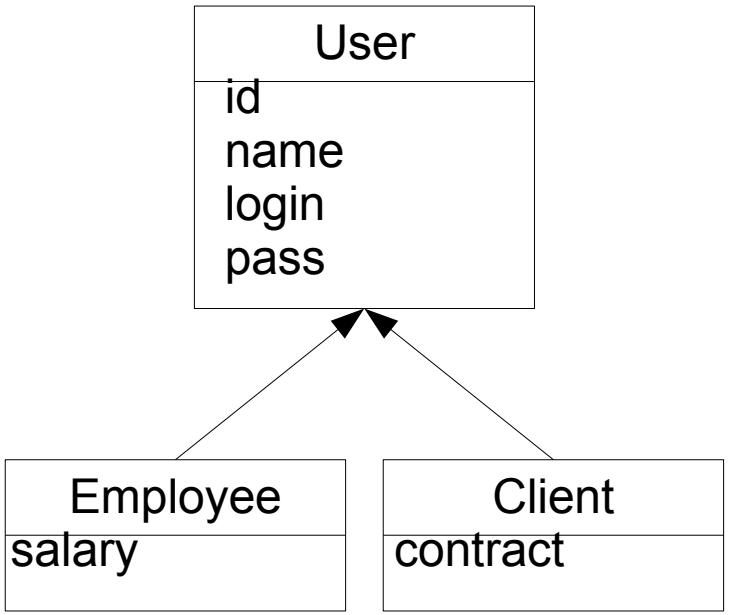
Natural id:

- Not null
- Unique
- Immutable (by default)

You will know:

- How to implement inheritance
- When not to use it
- How to write polymorphic queries

Domain model inheritance



Polymorphic queries:

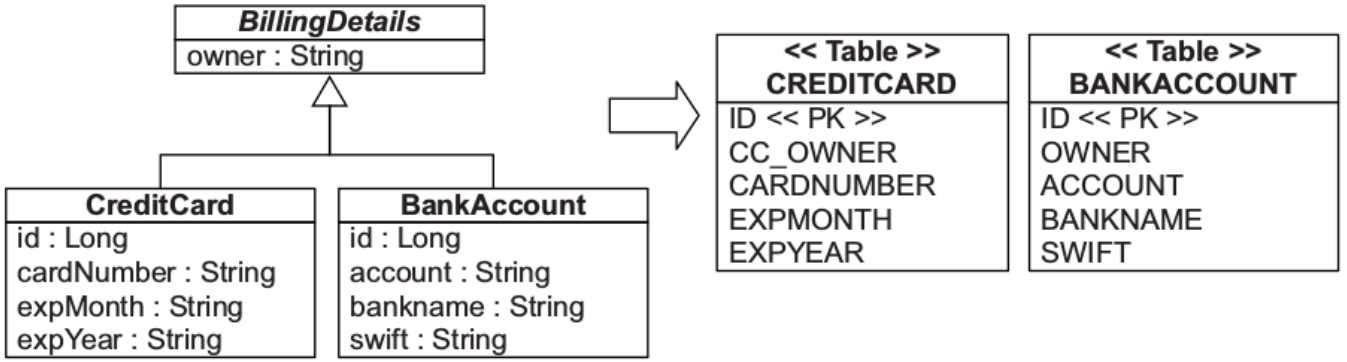
```
SELECT u FROM User u
```

„instanceof" - JPA 2.0

```
SELECT u
FROM User u
WHERE TYPE(u) = Client OR TYPE(u) = Employee
```

```
SELECT u.name,
        CASE TYPE(u)
            WHEN Employee THEN 'E'
            WHEN Client THEN 'C'
            ELSE 'x'
        END
FROM User u
WHERE u.name IS NOT EMPTY
```

Inheritance @MappedSuperclass

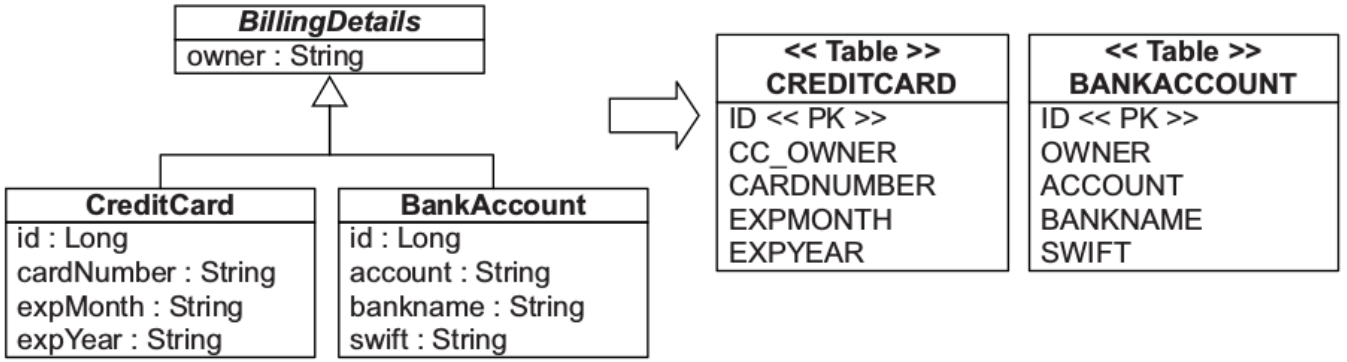


- Base classes are not Entities
  - Just common attributes

```
@MappedSuperclass
public abstract class BillingDetails {
    @NotNull
    protected String owner;
    // ...
}
```

```
@Entity
public class CreditCard extends BillingDetails {
    @Id
    @GeneratedValue(generator = Constants.ID_GENERATOR)
    protected Long id;
    @NotNull
    protected String cardNumber;
    @NotNull
    protected String expMonth;
    @NotNull
    protected String expYear;
    // ...
}
```

Inheritance
Table per concrete class

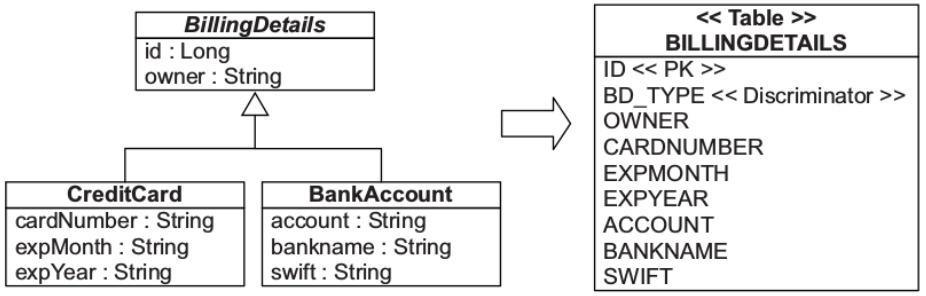



```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class BillingDetails {
    @NotNull
    protected String owner;
    // ...
}
```

```
@Entity
public class CreditCard extends BillingDetails {
    @Id
    @GeneratedValue(generator = Constants.ID_GENERATOR)
    protected Long id;
    @NotNull
    protected String cardNumber;
    @NotNull
    protected String expMonth;
    @NotNull
    protected String expYear;
    // ...
}
```

+Constrains
- Redundancy
- Performance (Unions or Many Selects when polymorphic query)

Inheritance Single table

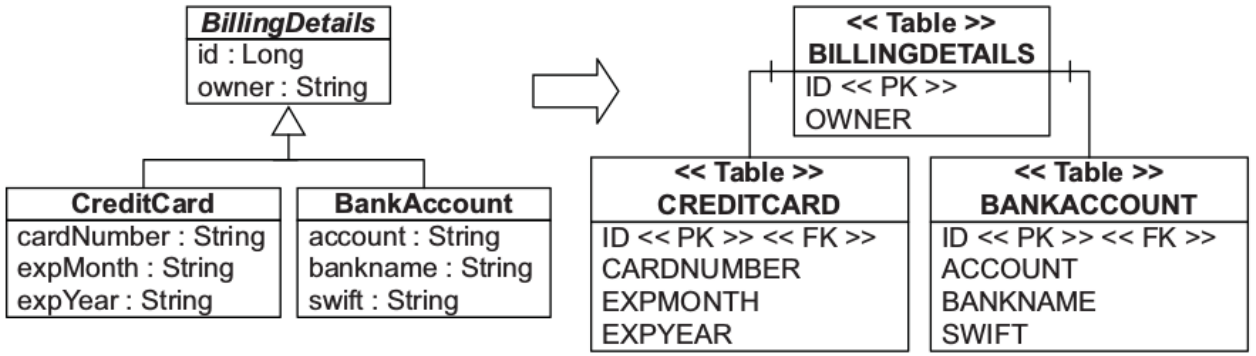



```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "BD_TYPE")
public abstract class BillingDetails {
     @Id
     @GeneratedValue(generator = Constants.ID_GENERATOR)
     protected Long id;
     @NotNull
     @Column(nullable = false)
     protected String owner;
     // ...
}
```

```
@Entity
@DiscriminatorValue("CC")
public class CreditCard extends BillingDetails {
     @NotNull
     protected String cardNumber;
     @NotNull
     protected String expMonth;
     @NotNull
     protected String expYear;
     // ...
}
```

+ Fast
- Redundancy
- Constrains

Inheritance Joined




```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class BillingDetails {
     @Id
     @GeneratedValue(generator = Constants.ID_GENERATOR)
     protected Long id;
     @NotNull
     protected String owner;
     // ...
}
```

```
@Entity
public class BankAccount extends BillingDetails {
     @NotNull
     protected String account;
     @NotNull
     protected String bankname;
     @NotNull
     protected String swift;
     // ...
}
```

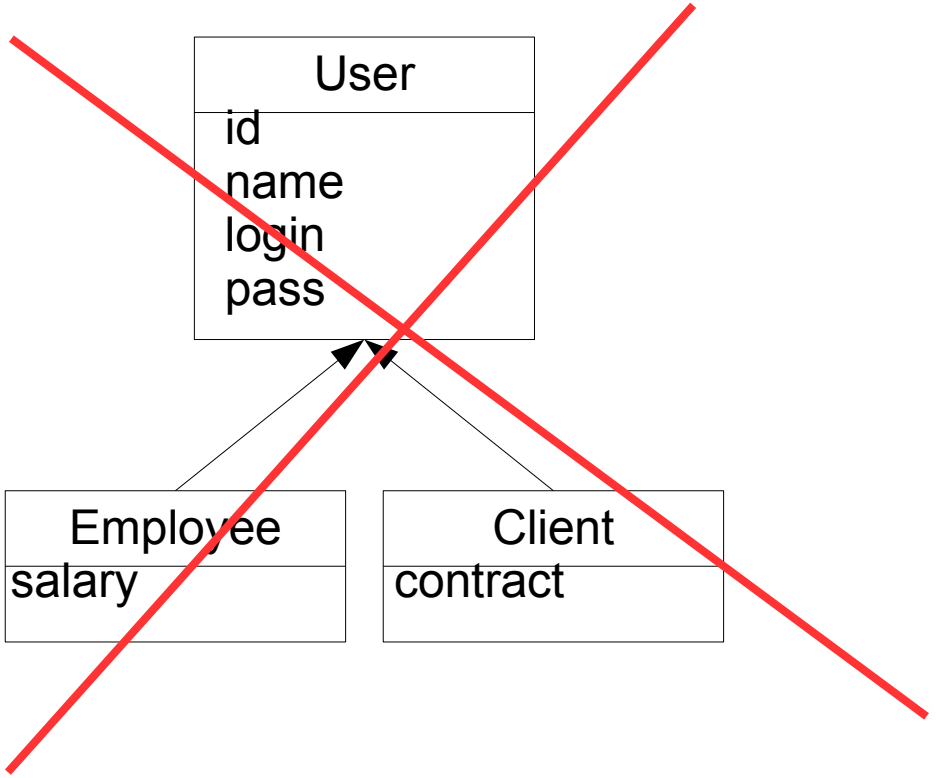
+ No redundancy (normalized)

~Constrains (common columns are still problematic)

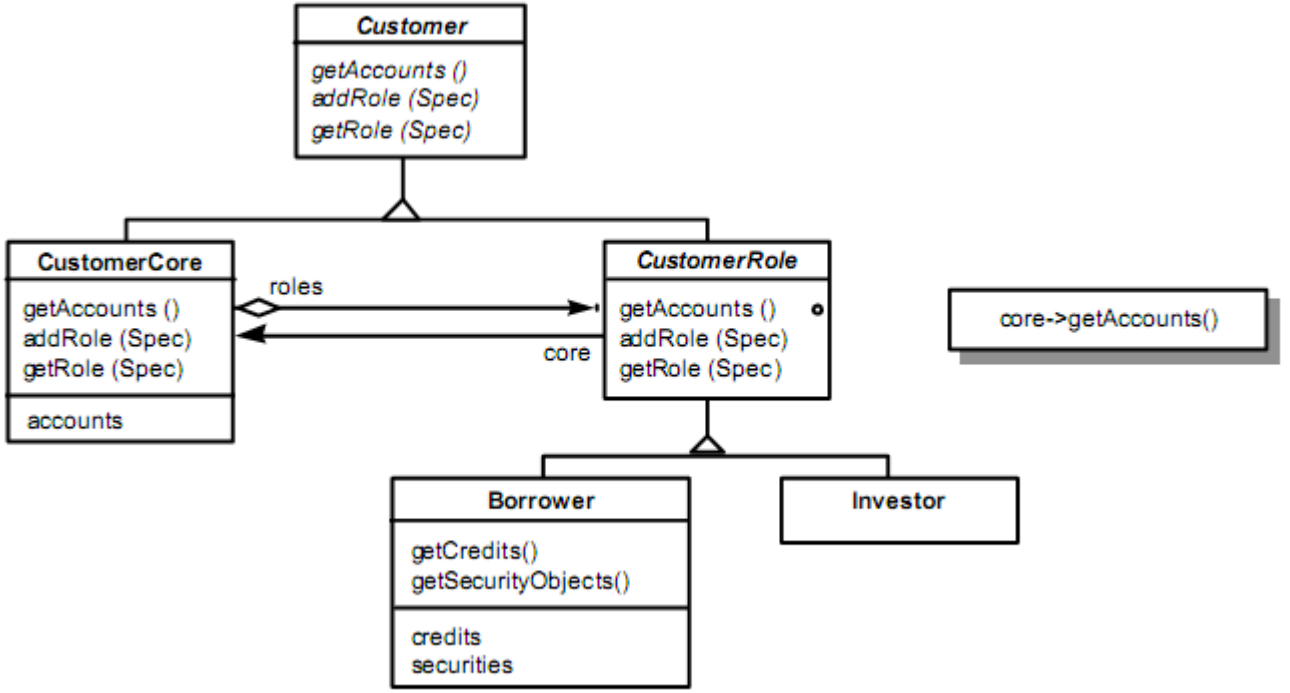
- Performance (Joins)

Domain model inheritance

- Inheritance is the worst approach to model roles
- Use instead: Role Object Design pattern

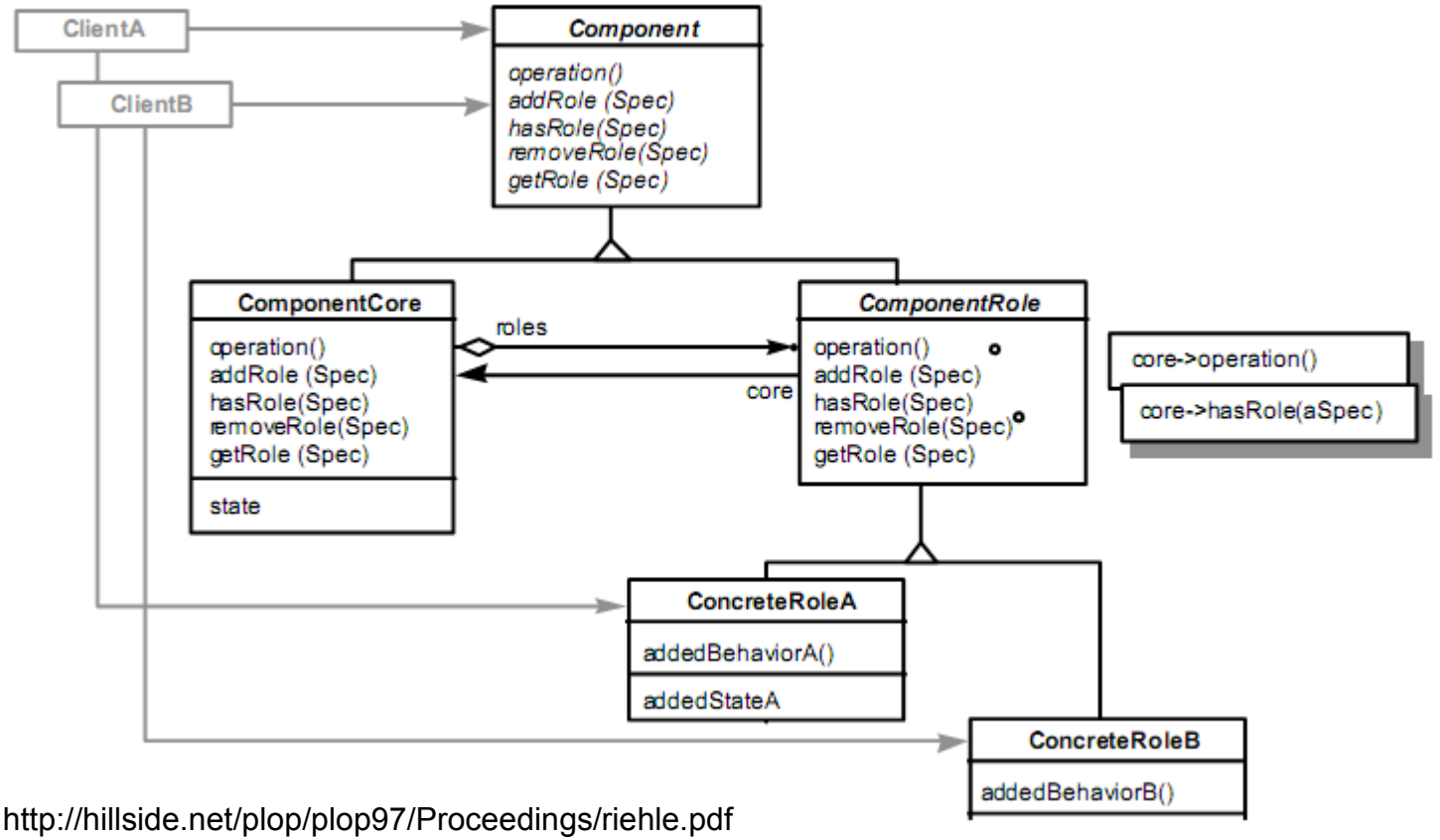


Role Object Design Pattern Sample



source: http://hillside.net/plop/plop97/Proceedings/riehle.pdf

Role Object Design Pattern
General Structure



source: http://hillside.net/plop/plop97/Proceedings/riehle.pdf

bottega IT SOLUTIONS

Exercise: Inheritance and roles

- Introduce user model: Admin, Supervisor and Standard
  - Introduce roles: Invoice Issuer, Order Corrector
    - each user type has specific impl of the role (Standard user can not correct orders)

bottega IT SOLUTIONS

Common operation performed on entities

- Base Repository
- Callbacks
  - code in the entity
- Listeners
  - multiple
- Interceptor
  - just one
  - per SessionFactory or just single Session
  - allows data changing

bottega IT SOLUTIONS

Injecting Repos

```
@Repository
public class JpaOrderRepository implements OrderRepository
  private TaxPolicy taxPolicy;
  private RebatePolicy rebatePolicy;


  public Order loadOrder(Long id){
    Order o = session...
    o.setTaxPolicy(taxPolicy);
    o.setRebatePolicy(rebatePolicy);
    return o;
  }
}
```

bottega IT SOLUTIONS

Callbacks

```
@MappedSuperclass
public class AbstractTimestampEntity {

    @Temporal(TemporalType.TIMESTAMP)
    @Column(name = "created", nullable = false)
    private Date created;

    @Temporal(TemporalType.TIMESTAMP)
    @Column(name = "updated", nullable = false)
    private Date updated;

    @PrePersist
    protected void onCreate() {
        updated = created = new Date();
    }

    @PreUpdate
    protected void onUpdate() {
       updated = new Date();
    }
}
```

Events

```
@Entity
@EntityListeners({LastUpdateListener.class})
public class AbstractTimestampEntity {
```

```
public class LastUpdateListener {
    @PreUpdate
    @PrePersist
    public void setLastUpdate(AbstractTimestampEntity o) {
        o.setLastUpdate( new Date() );
    }
}
```

Interceptors

```
public class TimeStampInterceptor extends EmptyInterceptor {

    public boolean onFlushDirty(Object entity, Serializable id, Object[] currentState,
            Object[] previousState, String[] propertyNames, Type[] types) {
        if (entity instanceof TimeStamped) {
            int indexOf = ArrayUtils.indexOf(propertyNames, "lastUpdated");
            currentState[indexOf] = new Date();
            return true;
        }
        return false;
    }

    public boolean onSave(Object entity, Serializable id, Object[] state,
            String[] propertyNames, Type[] types) {
            if (entity instanceof TimeStamped) {
                int indexOf = ArrayUtils.indexOf(propertyNames, "createdDate");
                state[indexOf] = new Date();
                return true;
            }
            return false;
    }
}
```

bottega IT SOLUTIONS

Exercise: Dependency Injection

- Inject into aggregates:
  - time service
  - events engine

Performance

You will learn:

- n+1 Select Problem
  - idea
  - detection
  - possible solutions
- Effective data loading
- Effective mapping tips
- Batch processing
- Quering tips

n+1 Select Problem

List<User>
- user1
  - List<Address>
    - address1
    - address2
- user2
  - List<Address>
    - address3
    - address4
    - address5
- user3
  - List<Address>
    - address6

```
@Entity
public class User{
    @OneToMany
    private List<Address> addresses;
}
```

```
List<User> users = entityManager.
    createQuery("SELECT u FROM User u").getResultList();

for (User u : users){
    for (Address a : u.getAddresses()){
        //...
    }
}
```

Eventually: iteration in some graphic component (table) on GUI using Open Session in View

bottega IT SOLUTIONS

Some trick and traps

List<User>
- user1
  - List<Address>
    - address1
    - address2
- user2
  - List<Address>
    - address3
    - address4
    - address5
- user3
  - List<Address>
    - address6

```
@Entity
public class User{
    @OneToMany(fetch=FetchType.EAGER)
    private List<Address> addresses;
}
```

- fetch=FetchType.EAGER
  - HQL respects that intention and guarantees data being loaded – but how?
  - **HQL do not respect any @Fetch(FetchMode.JOIN)**
  - **HQL respect @Fetch(FetchMode.SUBSELECT)**
    - loads Address data for all users in subselect
- @LazyCollection
  - FALSE
  - EXTRA – tries to avoid loading all elements but only needed: size(), contains(), get(), etc. do not trigger collection initialization

Weak points: hardcoded for all use cases

n+1 SP Quick Fix Batch Size

List<User>
- user1
  - List<Address>
    - address1
    - address2
- user2
  - List<Address>
    - address3
    - address4
    - address5
- user3
  - List<Address>
    - address6

- @org.hibernate.annotations.BatchSize
- loads x next **collections** – not elements
  - **Addresess for x Users**
- „blind” mechanism

```
@Entity
public class User{
    @OneToMany
    @BatchSize(size=10)
    private List<Address> addresses;
}
```

select … from User
select … from Address where user_id in (?, ?, ?, ?, ?, ?, ?, ?, ?, ?)

bottega
IT SOLUTIONS

Batch Size

BatchSize can be a solution for multiple Joins

```
SELECT emp FROM Employee emp
    LEFT JOIN FETCH emp.address
        LEFT JOIN FETCH emp.phoneNumbers
```

Each join causes cartesian product – result multiplication

n+1 SP Real Solution
Query per Use Case

List<User>
- user1
  - List<Address>
    - address1
    - address2
- user2
  - List<Address>
    - address3
    - address4
    - address5
- user3
  - List<Address>
    - address6

```
SELECT DISTINCT u FROM User u
   LEFT JOIN FETCH u.addresses
```

```
Criteria criteria = hibernateSession
    .createCriteria(User.class);

if (…){
    criteria.setFetchMode(„addresses”, FetchModel.JOIN);
}
```

bottega IT SOLUTIONS

n+1 SP Detection

- Manual: console
  - convenient parameters: use P6Spy Driver
  - **works for n>1:)**

```
<property name="show_sql">true</property>
<property name="format_sql">true</property>
<property name="use_sql_comments">true</property>
```

```
log4j.logger.org.hibernate.type=DEBUG
```

- Automatic
  - end2end tests measuring query count

```
Statistics stats = sessionFactory.getStatistics()
```

n+1 SP Automatic Detection in EJB

```
public class NPlusOneSelectProblemDetectingInterceptor {

 @PersistenceUnit
 private EntityManagerFactory entityManagerFactory;

 @AroundInvoke
 public Object countStatements(InvocationContext invContext)
                                                    throws Exception {
  InjectedEntityManagerFactory iemf = (InjectedEntityManagerFactory)
                                                        entityManagerFactory;
  EntityManagerFactoryImpl hemf = (EntityManagerFactoryImpl)
                                                    iemf.getDelegate();

  SessionFactory sessionFactory = hemf.getSessionFactory();
  Statistics statistics = sessionFactory.getStatistics();
  statistics.setStatisticsEnabled(true);

  long before = statistics.getPrepareStatementCount();

  Object result = invContext.proceed();

  long count = statistics.getPrepareStatementCount() - before;
  if (count > 30){
    String message = invContext.getTarget().getClass()
           + "->" + invContext.getMethod().getName() + " statements: " + count;
    //TODO wysłać maila do db-nazi
  }
  return result;
 }
}
```

bottega
IT SOLUTIONS

Exercise: n+1 SP

- Export all Orders of the given User to the XML

bottega IT SOLUTIONS

Additional optimization

You will learn:

- Efficient data loading
- Batch operations traps
- Data Tranfer Objects usage

bottega IT SOLUTIONS

Naive data loading

- Cross table queries (grids)
  - usually needs few columns form each table
- DB communication overhead
- Remote Service overhead
  - repacking entities to Dto solves only this problem
- Memory bursts

bottega IT SOLUTIONS

Dedicated mapping classes

- No caching
- Types explosion

```
@MappedSuperClass
public class DocumentBase{
    @Id
    private Long id;
}
```

```
@Entity
public class DocumentLite extends DocumentBase{
    private String title;
}
```

```
@Entity
public class DocumentBig extends DocumentBase{
    private String content;
}
```

bottega IT SOLUTIONS

Load just what is needed

```
SELECT NEW mypackage.UserDTO(u.id, u.name, u.address)
     FROM User u JOIN FETCH u.address
```

```
sess.createSQLQuery("SELECT id, title FROM Documents").list();
```

```
sess.createSQLQuery("SELECT id, title FROM Documents")
    .addEntity(„d”, Document.class)
    .addJoin(„d.author”);
```

```
sess.createSQLQuery("SELECT id, title FROM Documents")
   .setResultTransformer(Transformers.aliasToBean(DocumentDTO.class))
```

bottega IT SOLUTIONS

Optimisation Tips

- @Immutable for collections - if set specifies that the elements of the collection never change (a minor performance optimization in some cases)
- find the size of a collection without initializing it?

```
Integer size = (Integer) s.createFilter( collection, "select count(*)" ).uniqueResult();
```

Query Tips
Criteria Corelated Subqueries

```
private DetachedCriteria coutSellersSubquery(){
  DetachedCriteria subquery = DetachedCriteria.forClass(Item.class, „i”);

  //no join, seller_id is a column in item
  subquery.add(Restrictions.eqProperty(„i.seller.id”, „u.id”));
  subquery.setProjection(Property.forName(„i.id”).count())

  return subquery;
}
```

```
Criteria criteria = session.createCriteria(User.class, „u”);
if (…)
  criteria.add(Subqueries.lt(10, coutSellersSubquery()));
```

Query Tips
Dynamic Fetching

- Criteria does not ignore *fetch* mapping property
  - as HQL does (excluding subselect)
- Performance: many joins may be slower than few queries; when eager than limiting result in memory

```
session.createCriteria(Item.class)
  .createAlias(„bids”, „b”, CriteriaSpecification.INNER_JOIN)
  .setFetchMode(„b”, FetchMode.JOIN) //outer join
```

Problem: cartesian product

- Solution: repack to LinkedHashSet (maintain order), use Result transformer or batch size

```
criteria.setResultTransformer(Criteria.DISTINCT_ROOT_ENTITY)
```

Query tips

- order by the size of a collection?

```
select user from User user
left join user.messages msg
group by user
order by count(msg)
```

- place a condition upon a collection size

```
from User user where size(user.messages) >= 1
from User user where exists elements(user.messages)
```

```
select user from User user
join user.messages msg
group by user  having count(msg) >= 1
```

```
select user from User as user
left join user.messages as msg
group by user having count(msg) = 0
```

Row Counting Smart Projection

```
int count = sess.createQuery("FROM Document")
                .list().size();
```

```
public int countDocuments(...){
    Criteria crit = createCriteria(...);
    return (Integer) crit.setProjection(Projections.rowCount())
                         .list().get(0);
}

public List<Document> searchDocuments(...){
    Criteria crit = createCriteria(...);
    return crit.list();
}

private Criteria createCriteria(...){
    Criteria criteria = hibernateSession.createCriteria(Document.class);
    …
    return criteria;
}
```

bottega IT SOLUTIONS

Exercise: searching

- Create Finder that search for all users' Orders that matches criteria
  - provide 2 methods:
    - searching
    - counting
  - prepare relevant model

Batch operations

```
Query q = session.createQuery(
   "update [versioned] Account set balance=
      (balance + (balance*interestRate))
   where accountType='SAVINGS' ");

int updatedItems = q.executeUpdate();
```

by default @version is not updated

```
Query q = session.createQuery(
"insert into ArchivedAccount(
   accountId, creationDate, balance)
select a.accountId, a.creationDate, a.balance
   from Account a");

int createdObjects = q.executeUpdate();
```

bottega IT SOLUTIONS

Batch operations trap

Batch is just translated to SQL, therefore:

- **Cache** is **outdated**
- **Cascade** operations are **ignored**

Good practices

- batch operations first, entity manager operations after batch
- Batch operations in separate TX (REQURES_NEW propagation)

Batch Conf

- Ordering statements – operations are batched only statements are the same – therefore we must sort them
  - Sample problem: large loop: update customer and add address

```
<property name="hibernate.order_updates" value="true" />
<property name="hibernate.order_inserts" value="true" />
```

- Hibernate can perform batch operations
  - But if insertion required database depended ID generation (ex. Identity) that batching is impossible

```
<property name="hibernate.jdbc.batch_size" value="50" />
```

Batch traps

```
for (int i=0; i<100000; i++ ) {
   Customer customer = new Customer(.....);
   session.save(customer);
   if (i % 20 == 0) { //20, same as the JDBC batch size
      //flush a batch of inserts and release memory:
      session.flush();
      session.clear();
   }
}
```

- 20-50 is recommended
- clear()
  - Saves memory
  - But causes GC run – may slowdown dramatically
- Solution: “it depends, you must measure”

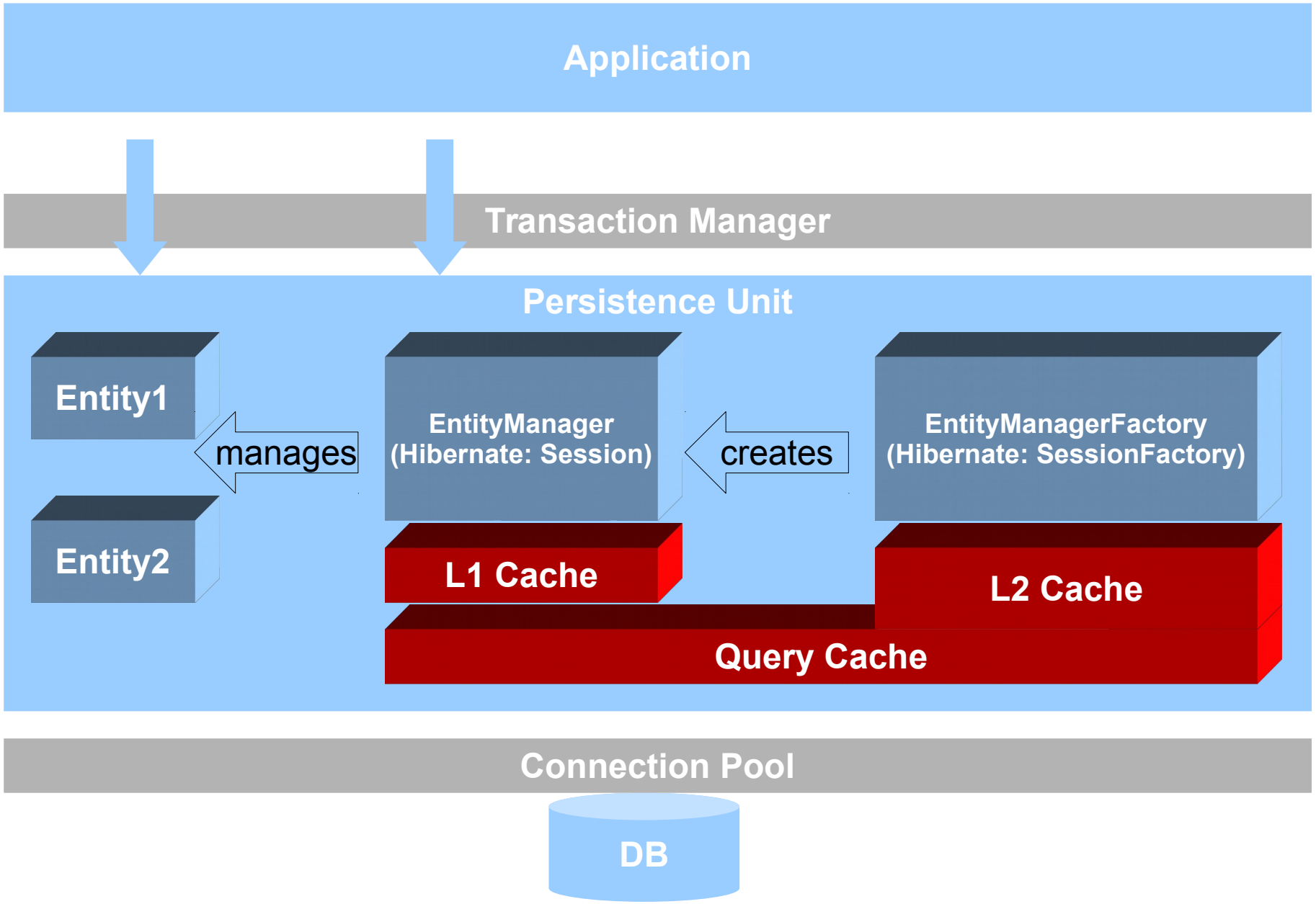
Hibernate Cache

You will know:

- Cache levels: L1, L2, query
- How to configure cache
- How to work effectively witch cache
- API: Hibernate and JPA 2

bottega IT SOLUTIONS

Hibernate architecture



L1 Cache

- L1 – entity cache Level 1
  - by default
  - associated with Session/EntityManager
  - Unit of Work – optimisation:
    - caching objects – load once
    - executing SQL later – when needed (before SELECT)
      - if Exception some statements never hit DB
      - keeps DB lock as shot as possible (from first update to commit)
    - Dirty checking – never update unnecessary objects
    - Rolling 2 updates into 1

L1 Cache Gotchas

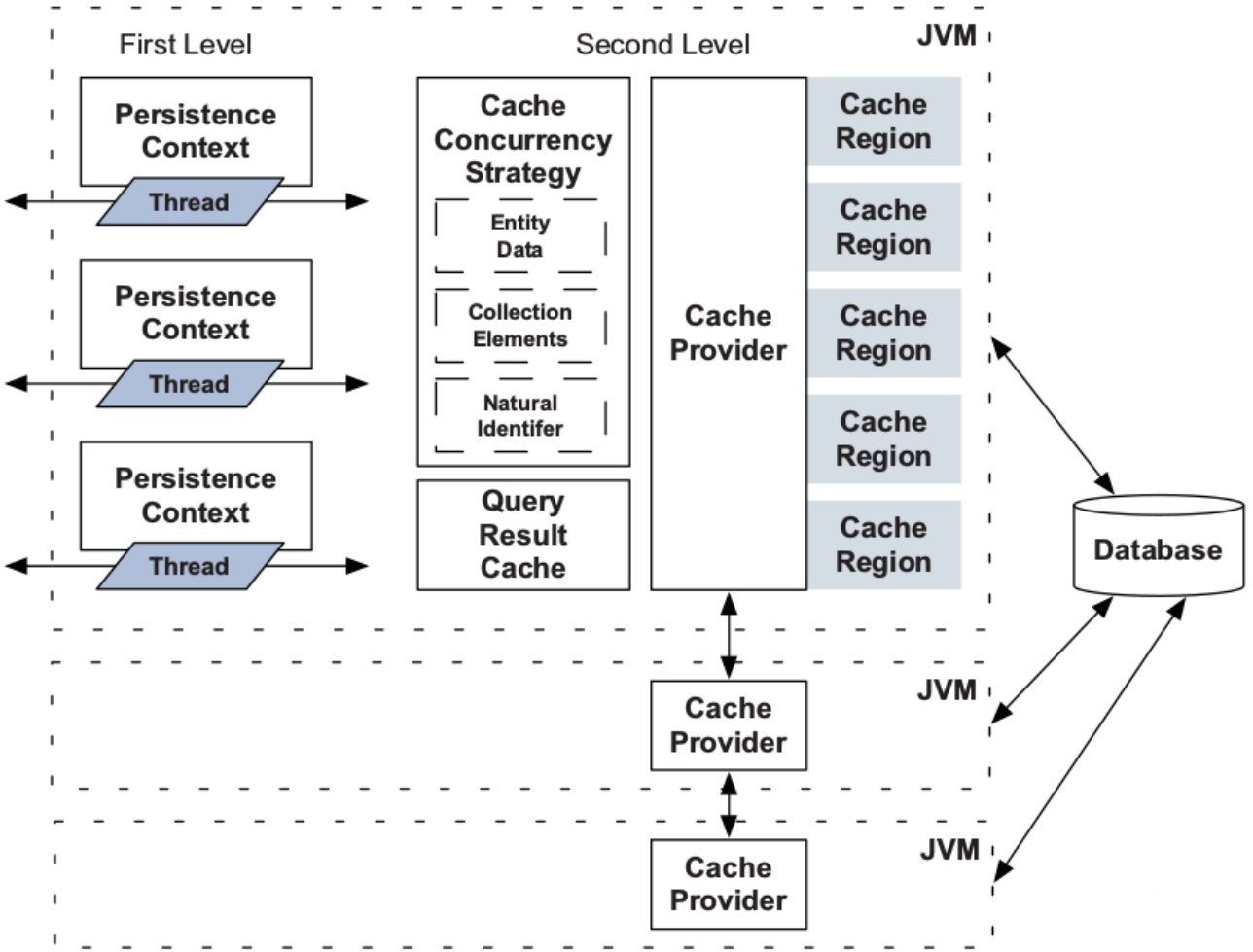
- Large memory consumption by L1
- Evict entity from L1 cache
  - When reading huge amount of data
  - or just don't want to update during flush

```
ScrollableResult users = sess.createQuery("from Users as user").scroll();

while ( users.next() ) {
    User user = (User) users.get(0);
    //.....
    session.evict(user);
}
```

L2 Cache
JPA 2: Shared Cache

- L2 – entity (or collection) cache Level 2
  - need to be turned on
  - associated with SessionFactory (EntityManagerfactory)
  - application scope
    - get() hits DB only once
  - accessed in step 2 (engine first checks L1)

L2 Cache
JPA 2: Shared Cache



L2 Cache configuration

```
<property
   name="hibernate.cache.provider_class">
      org.hibernate.cache.EHCacheProvider
</property>

<property
   name="hibernate.cache.use_second_level_cache"
   value="true"/>
```

```
@Entity
@Cache(
   usage = CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
public class User {
    @OneToMany()
    @Cache(
          usage = CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
    public List<Address> addresses;
}
```

bottega IT SOLUTIONS

L2 Cache strategy

- Read-only
  - most efficient
  - entities that are never modified (dictionaries)
- Read-write
  - more overhead
  - entities can be modified
- Transactional
- Nonstrict Read-write
  - rare modification
  - unlikely that **two transactions modify** the same Entity
    - If concurrent access to an item is possible, this concurrency strategy makes no guarantee that the item returned from the cache is the latest version available in the database.

```
@Entity
@Cacheable
public class User {

}
```

bottega IT SOLUTIONS

Evicting Cache in case of modification by side systems

```
sessionFactory.evict(User.class, userId);
sessionFactory.evict(User.class);
sessionFactory.evictCollection("User.addresses",   userId);
sessionFactory.evictCollection("User.adresses");
```

JPA 2:

```
Cache cache = factory.getCache();
cache.evict(Employee.class, id);
```

bottega IT SOLUTIONS

Query Cache

- Caches query result – only IDs
  - works only with L2
  - but scalar result are stored directly in QC
- Makes sense for queries that are:
  - the same
  - has the same parameters
- Introduces overhead – checking if **tables** were changed
  - **measure before using!**

```
<property name="hibernate.cache.use_query_cache" value="true"/>
```

bottega IT SOLUTIONS

Query Cache Samples

```
@NamedQuery(
    name="allusers",
    query="FROM User",
    hints={
        @QueryHint(
            name="org.hibernate.cacheable",
            value="true")})
@Entity
public class User{..}
```

```
hibernateSession.createQuery(„FROM User”)
   .setCacheable(true).list();
```

```
Criteria criteria = hibernateSession.createCriteria(Document.class);
criteria.setCacheable(true);
```

Query Cache Sample JPA 2

```
Query query = em.createQuery("Select e from Employee e");
query.setHint("javax.persistence.cache.storeMode", "REFRESH");
```

- javax.persistence.cache.retrieveMode : CacheRetrieveMode
  - BYPASS : Ignore the cache, and build the object directly from the database result.
  - USE : Allow the query to use the cache. If the object/data is already in the cache, the cached object/data will be used.
- javax.persistence.cache.storeMode : CacheStoreMode
  - BYPASS : Do not cache the database results.
  - REFRESH : If the object/data is already in the cache, then refresh/replace it with the database results.
  - USE : Cache the objects/data returned from the query.

Query Cache Gotchas

- Memory “leaks”
  - parameters are also stored in cache

```
public List<User> findUsersByAddress(Address a){
   return hibernateSession
    .createQuery(„FROM User u WHERE u.address = ?”)
       .setParameter(0, a)
       .setCacheable(true)
       .list();
}
```

- Overlapping queries
  - timeout of one query cache evicts entities of other query cache (that has some time left)

Query Cache Tip

Named query (which is precompiled) can be **faster** than **dynamic** query with **cache**

Makes sense for rarely modified entities (trap: audit logging in entities ex: last access imet)

Useful for Natural id queries – L2 cache works only for primary id.

bottega IT SOLUTIONS

Exercise: Cache

- Configure cache
  - L2
    - Users, Products
  - Query
    - Special offer products query

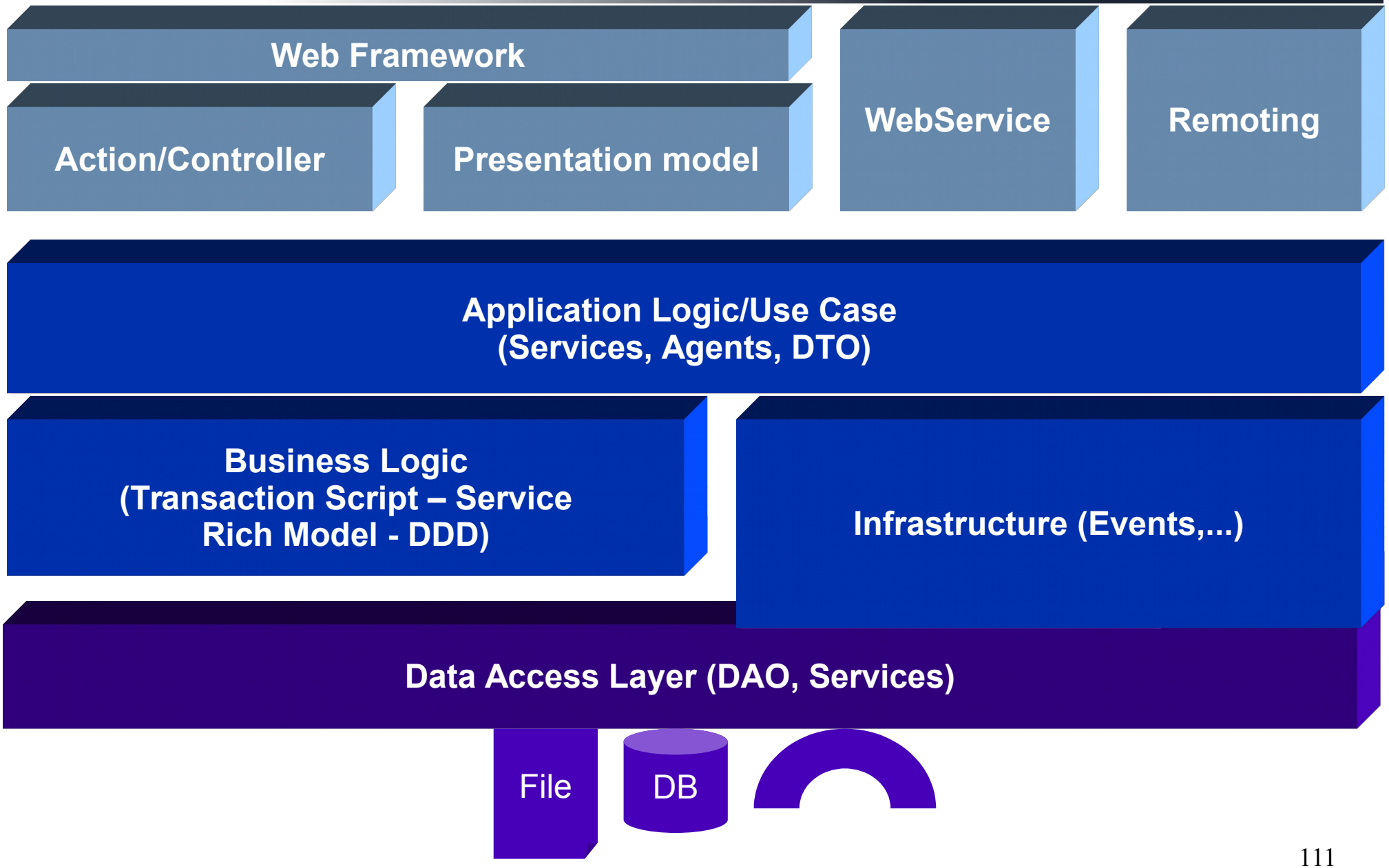
bottega IT SOLUTIONS

Command-query Responsibility Segregation

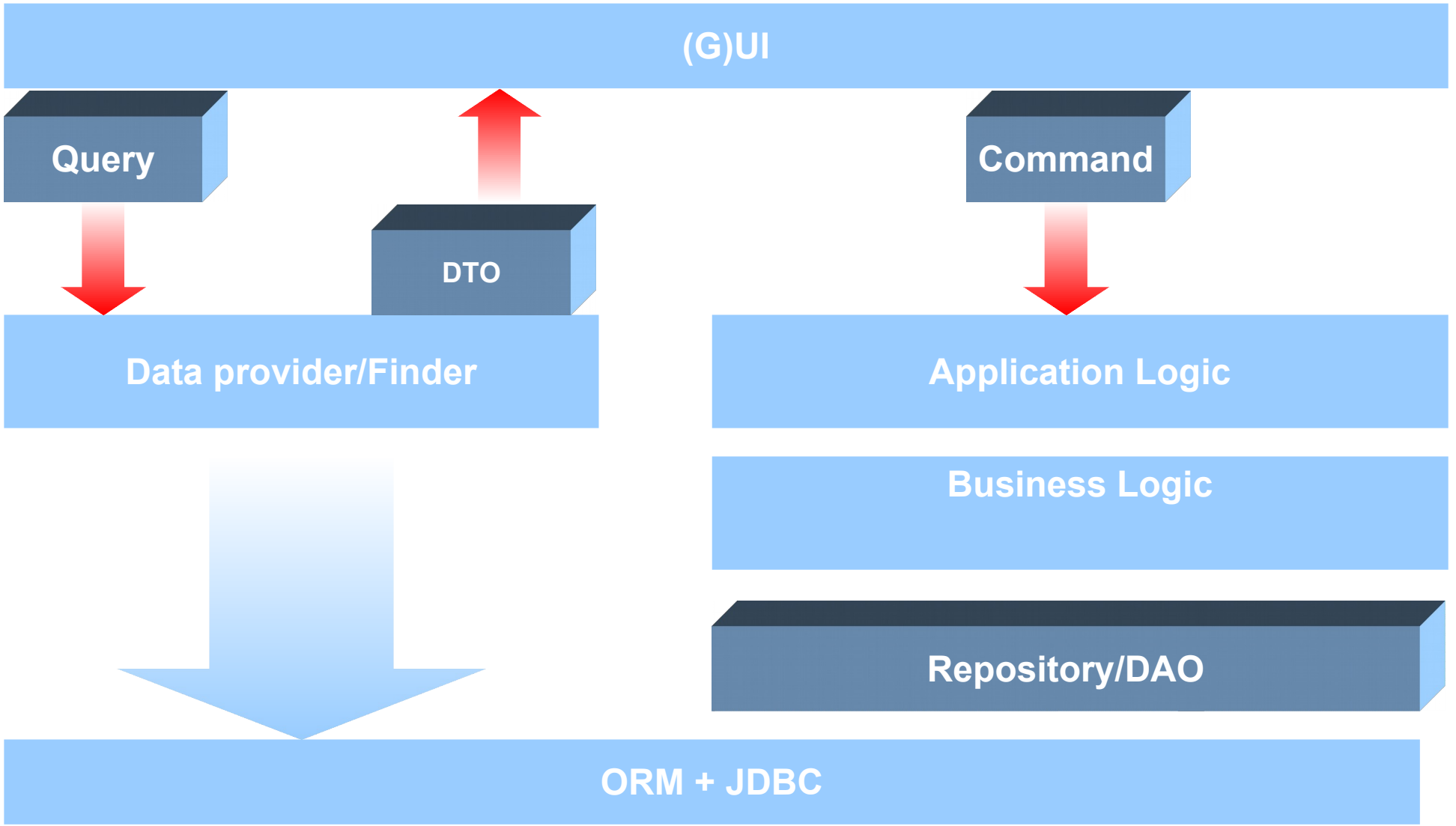
You will learn:

- CqRS - general idea
- 3 implementations
- myBatis

Classic Architecture



Command-query Responsibility Segregation



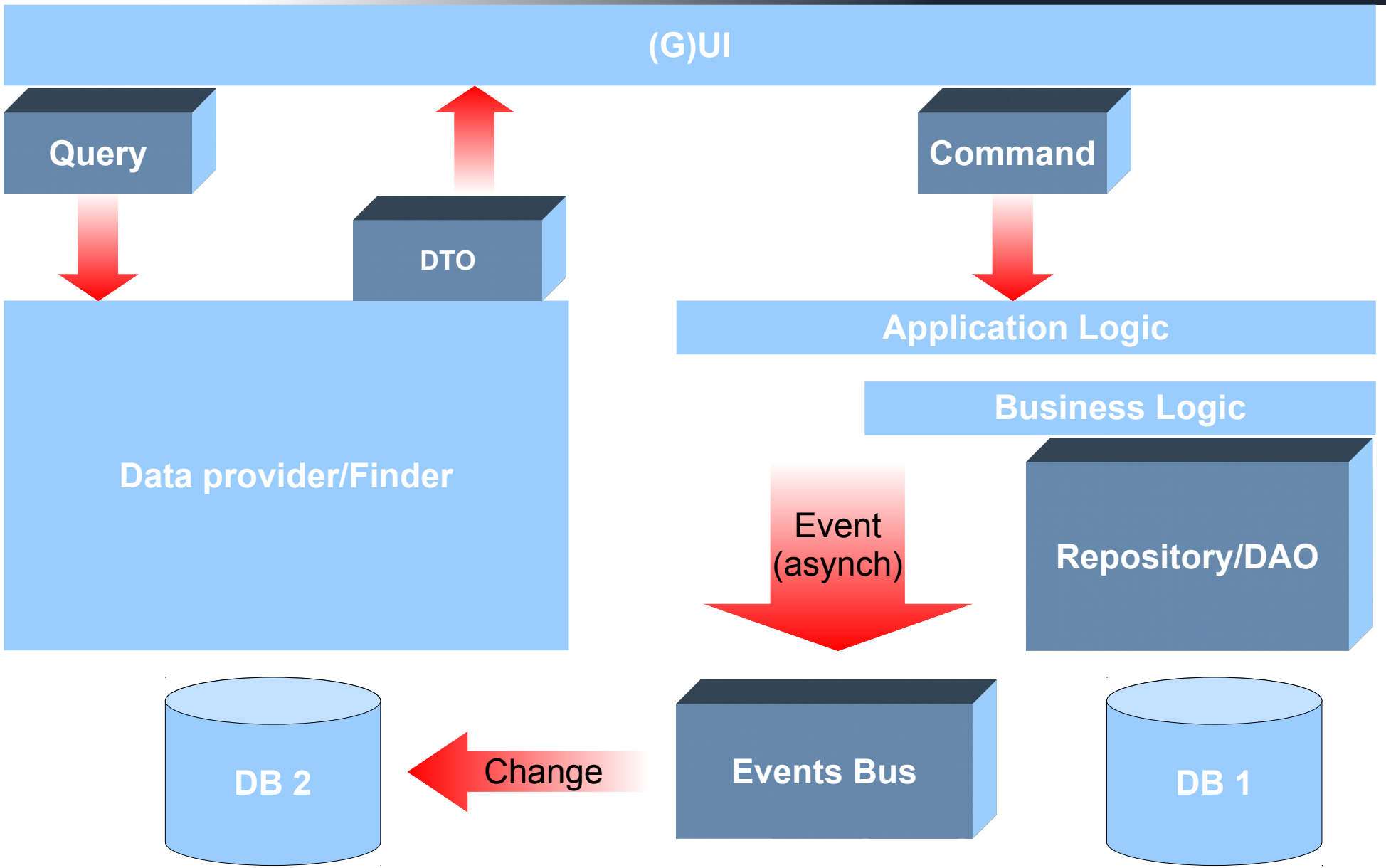
bottega IT SOLUTIONS

CqRS postulates

- Intention revealing GUI – not just CRUD
  - taskbased, inductive
  - user working with screens builds a Command
  - bad: SaveAddress, ChangeAddress
  - good: Relocate, Correct
- Commands can be stored
  - interaction history, users' habbits

Read and Write models

- Synchronizing Read Model
  - SQL Select for specific fields only
    - **return DTOs relevant for Use Case**
  - Materialized Views
  - Separate, dedicated model
    - read oriented: fast, simple query (no joins)
    - nosql?
    - How to replicate...?


(G)UI
Query
DTO
Command
Application Logic
Business Logic
Data provider/Finder
Event
(asynch)
Repository/DAO
DB 2
Change
Events Bus
DB 1

Command stack

```
public class AddroductCommand
```

```
public interface Handler<T> {
    Class<T> getMessageType();
    void handleMessage(T message) throws Exception;
}
```

```
public class AddProductHandler
                        extends AbstractHandler<AddProductCommand> {

    private ProductRepository repository;

    public void handleMessage(AddProductCommand message) {
        //...
    }
}
```

bottega IT SOLUTIONS

Sample Finder

```
public class SearchDocumentsQuery implements Serializable{
    private Status status;
    private Date epiryDate;
    private String[] titleWords;
    private String[] contentWords;

    //Gettery i settery/konstruktor

}
```

```
public class DocumentFinder /*implements DocumentFinderLocal*/{
    public List<Document> search(SearchDocumentsQuery query){
        //ORM
    }

    public List<DocumentDTO> search(SearchDocumentsQuery query){
        //JDBC
    }
}
```

bottega IT SOLUTIONS

Query a'la DSL

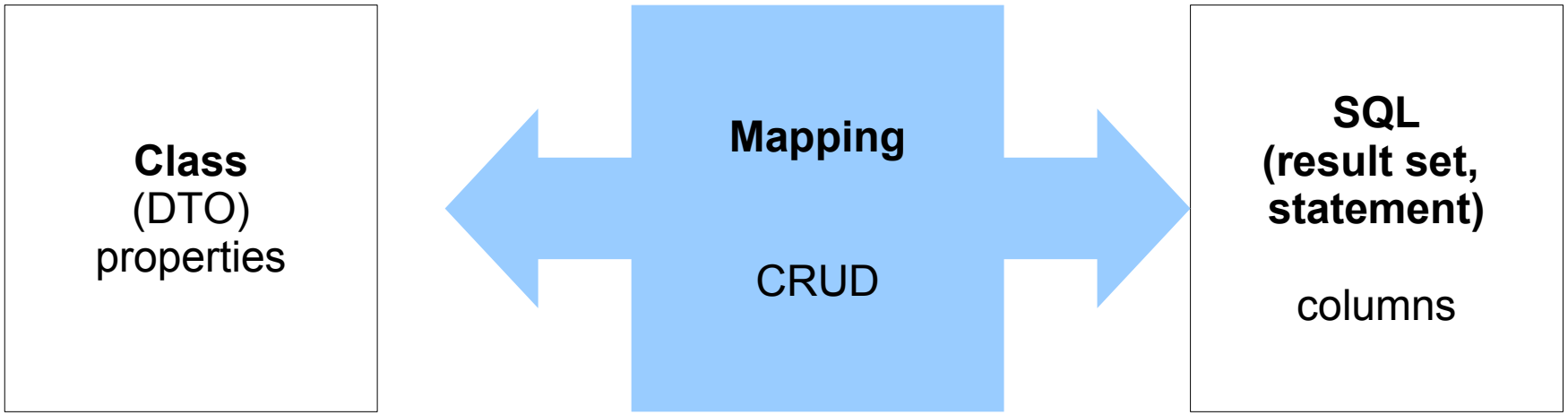
```
public class SearchDocumentsQuery implements Serializable{
    private Status status;
    private Date epiryDate;
    private String[] titleWords;
    private String[] contentWords;

    //ONLY getters

    public SearchDocumentsQuery current(){
        status = Status.ACTIVE;
        expiryDate = //tommorow
        return this;
    }

    public SearchDocumentsQuery contains(String phrase){
        String[] words = phrase.split(" ");
        titleWords = words;
        contentWords = words;
        return this;
    }
}
```

bottega IT SOLUTIONS

Paginated Finder

```
public class PaginatedResult<T> implements Serializable {
    private final List<T> items;
    private final int pageSize;
    private final int pageNumber;
    private final int pagesCount;
    private final int totalItemsCount;
```

```
public class ProductSearchCriteria implements Serializable {
    public enum ProductSearchOrder {NAME, PRICE; }
    // constraints
    private String containsText;
    private Double maxPrice;
    //
    private ProductSearchOrder orderBy = ProductSearchOrder.NAME;
    private boolean ascending = true;
    // pagination support
    private int pageNumber = 1;
    private int itemsPerPage = 50;
```


Class
(DTO)
properties
Mapping
CRUD
SQL
(result set,
statement)
columns

myBatis
sample: mapper and query builder

```
public interface BlogMapper {
  @Select("SELECT * FROM blog WHERE id = #{id}")
  Blog selectBlog(int id);
}

BlogMapper mapper = session.getMapper(BlogMapper.class);
Blog blog = mapper.selectBlog(101);
```

```
private String selectPersonLike(Person p){
  BEGIN(); // Clears ThreadLocal variable
  SELECT("P.ID, P.USERNAME, P.PASSWORD, P.FIRST_NAME, P.LAST_NAME");
  FROM("PERSON P");
  if (p.id != null) {
  WHERE("P.ID like #{id}");
  }
  if (p.firstName != null) {
  WHERE("P.FIRST_NAME like #{firstName}");
  }
  if (p.lastName != null) {
  WHERE("P.LAST_NAME like #{lastName}");
  }
  ORDER_BY("P.LAST_NAME");
  return SQL();
}
```

bottega IT SOLUTIONS

Exercise: Read Model

- Implement Read Model using myBatis

You will learn:

- ACID, Anomalies, Isolation
- TX management
- TX propagation
- traps

bottega
IT SOLUTIONS

ACID

- **Atomic** – all or nothing

- **Consistent** – transaction will bring the database from one valid state to another; consistent, no integrity violation

- **Isolated** – transaction should be able to interfere with another transaction (depends on isolation level)

- **Durable** - means that once a transaction has been committed, it will remain so, even after crash

bottega
IT SOLUTIONS

Read anomalies

- **dirty read** – TX1 read data changed by TX2 even if later TX2 is rolled back. TX1 red non-existing data

- **unrepeatable read** – TX1 performs the same query but receives different data (row attributes). TX2 modifies attributes

- **phantoms -** TX1 performs the same query but receives different data sets (amount of rows). TX2 modifies rows.

Dirty Read is crucial problem for data integrity.
Unrepeatable reads and Phantoms can be tolerated.

bottega IT SOLUTIONS

Isolation levels – performance/consistency

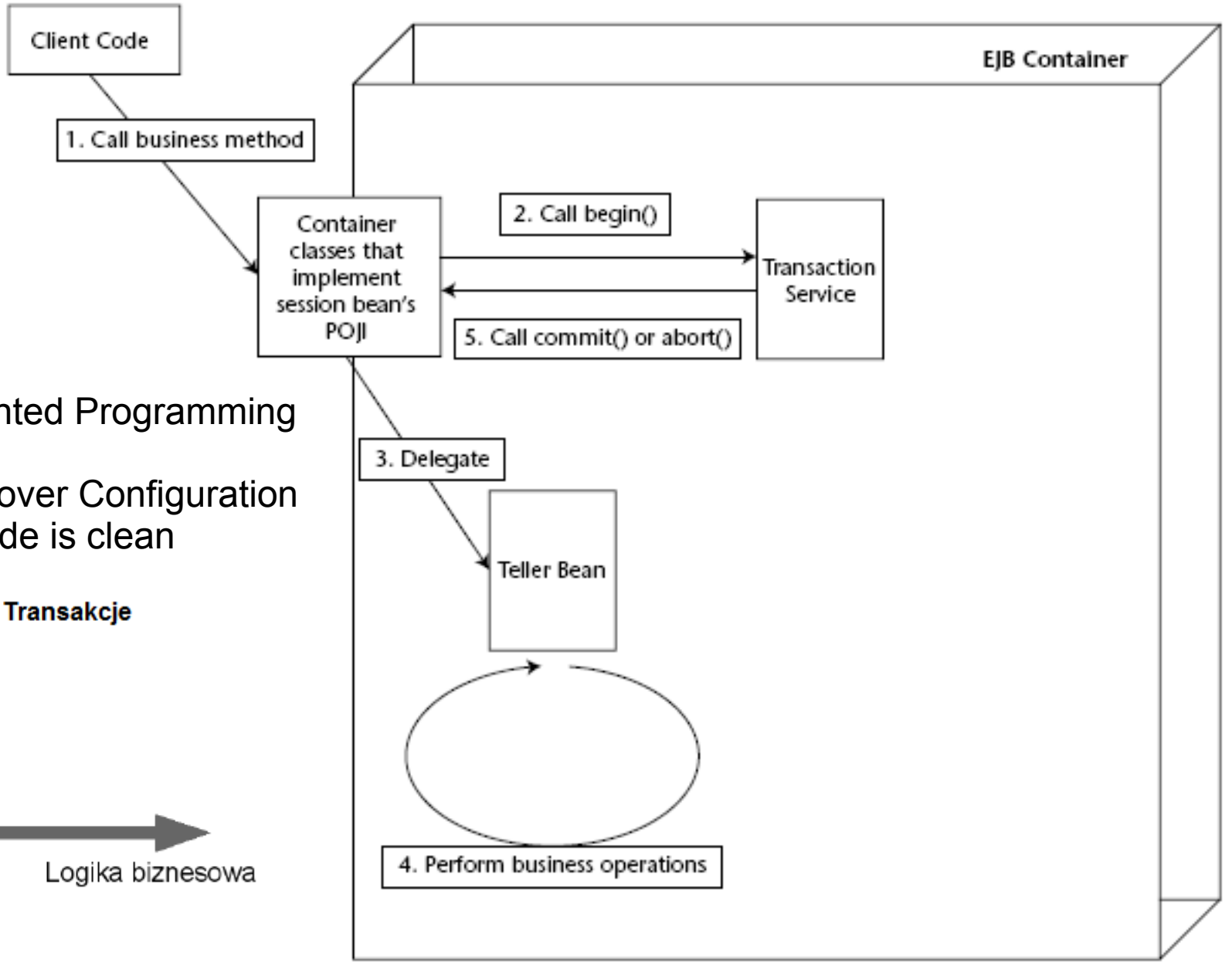
- **read_uncommited** – uncommitted changes made by TX1 are visible for TX2 (dirty-reads, non-repetable-reads, phantoms)

- **read_commited** – changes made by TX1 are visible only after commit (non-repetable-reads, phantoms)

- **repetable_read** – loaded data are blocked, so repeated reads returns the same data (phantoms)

- **serializable** – transactions are serialized (no anomalies)

Container Managed TX



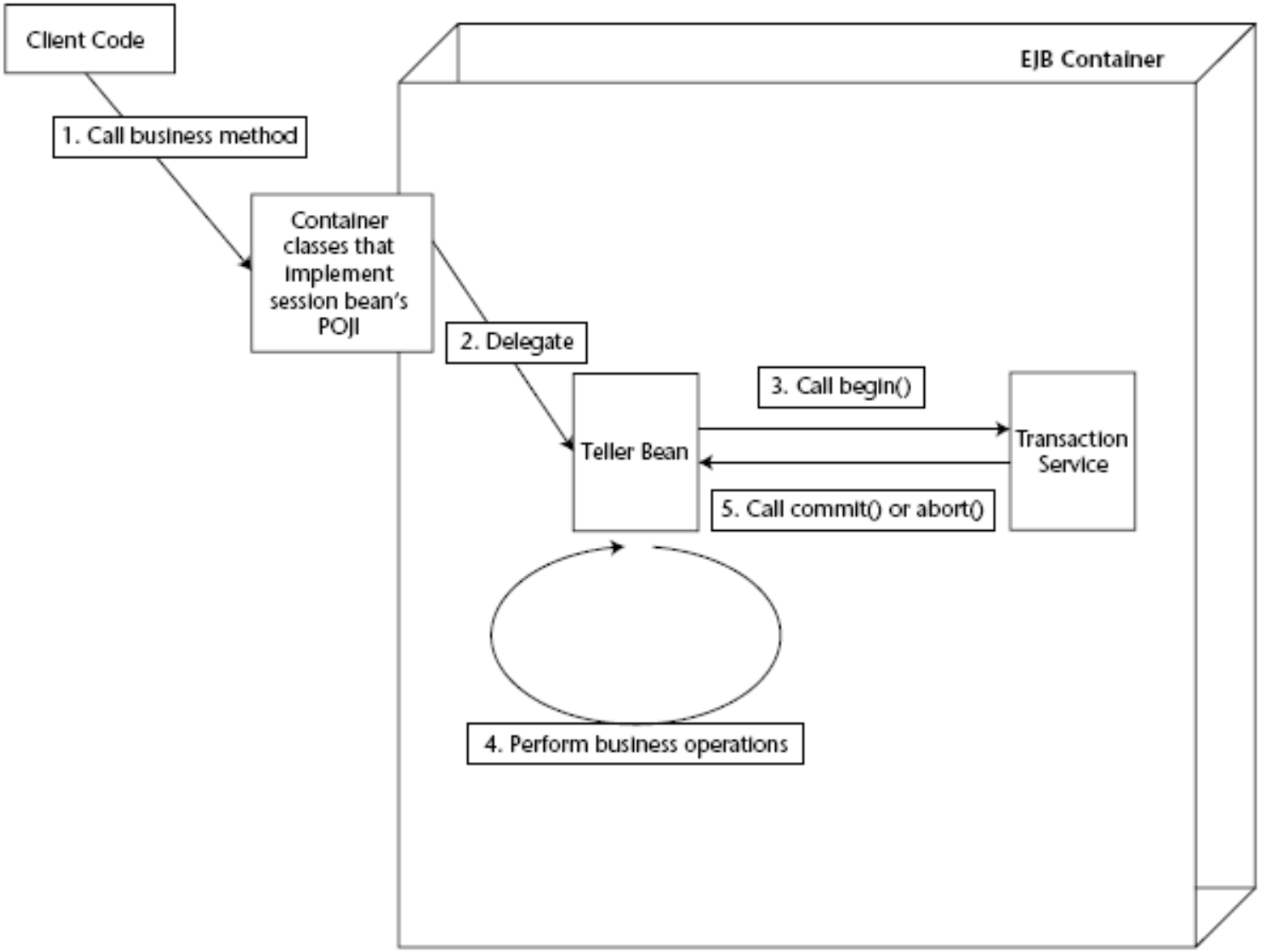
- Aspect Oriented Programming style
- Convention over Configuration
- Business code is clean

Container Managed TX

- Dafault propagation: REQUIRED

```
@TransactionAttribute(TransactionAttributeType.REQUIRED)
@Stateless
public class BiznesBean implements BiznesBeanLocal{
   @Resource
   private EJBContext ctx;

   @TransactionAttribute(TransactionAttributeType.REQUIRED)
   public void doBiznes(){
      ctx.setRollbackOnly();//!!!
   }
}
```

Bean Managed TX



Bean Managed TX

- SLSB – propagation only per single method (because: pooled objects)
- MDB only on *onMessage()*

```
@Stateless
@TransactionManagment(TransactionManagmentType.BEAN)
public class BiznesBean implements BiznesLocal{
    @Resource SessionContext ejbContext
    @Resource UserTransaction ut

    public void doBiznes(){
        ut.begin();
        //...
        ut.rollback();
        ut.setRollbackOnly(); //(!!!)
        ut.commit();
    }
}
```

Bean Management - template

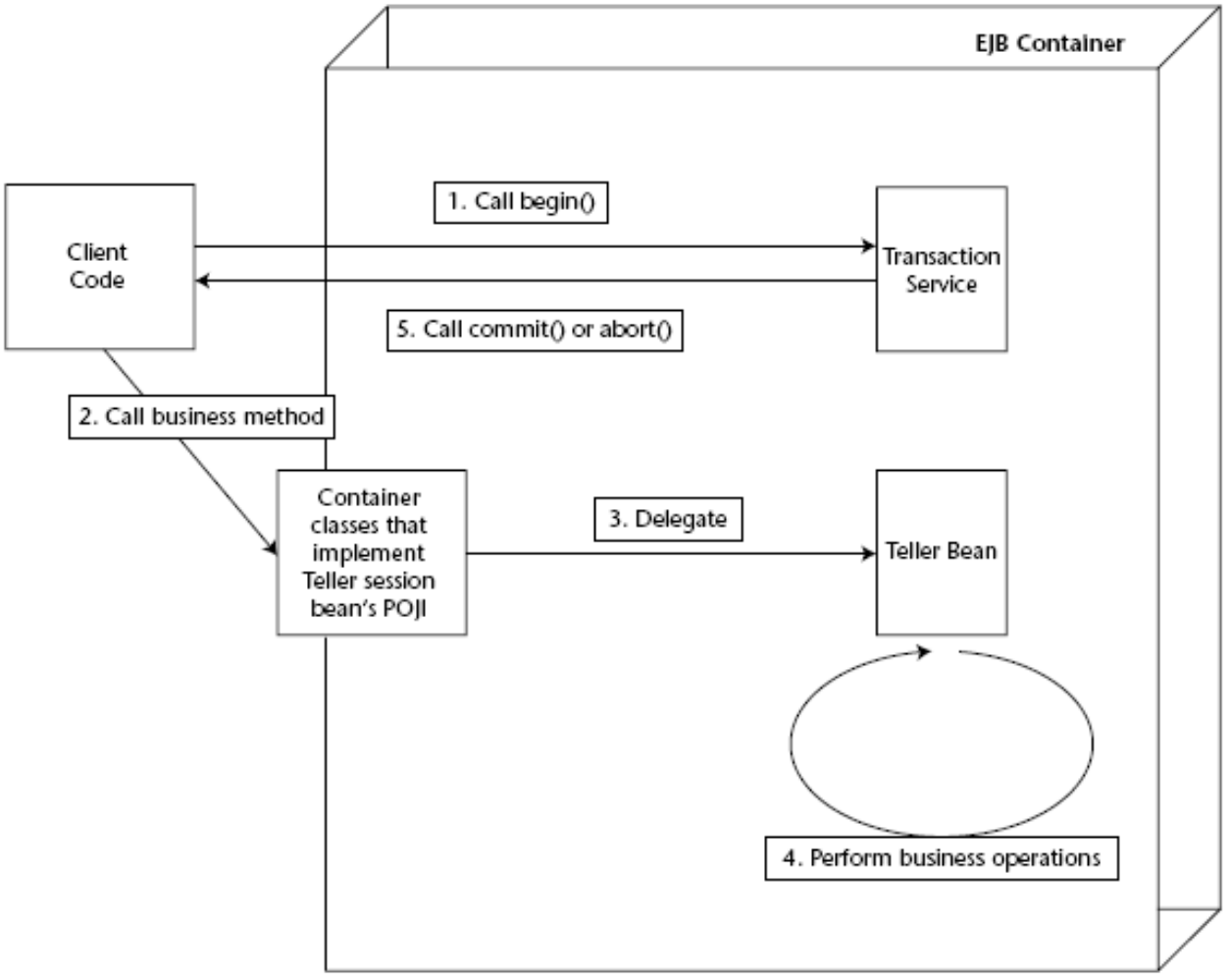
```
public class SimpleService {
    private final TransactionTemplate transactionTemplate;

    public SimpleService(PlatformTransactionManager transactionManager) {
        this.transactionTemplate = new TransactionTemplate(
                                                transactionManager);
    }

    public Object someServiceMethod() {
        transactionTemplate.execute(new TransactionCallbackWithoutResult() {

            protected void doInTransactionWithoutResult(
                                                TransactionStatus status) {
                try {
                   //...
                } catch (SomeBusinessExeption ex) {
                      status.setRollbackOnly();
                }
            }
        });
}
```

Bean Management - manager

org.springframework.transaction.PlatformTransactionManager

```
DefaultTransactionDefinition def = new DefaultTransactionDefinition();
def.setName("SomeTxName");
def.setPropagationBehavior(TransactionDefinition.PROPAGATION_REQUIRED);

TransactionStatus status = txManager.getTransaction(def);
try {
    // …
}
catch (MyException ex) {
    txManager.rollback(status);
    throw ex;
}
txManager.commit(status);
```

Client Managed TX



Client Managed TX

- Client works on remote stubs
- Client is responsible for consistency!!!

```
javax.transaction.UserTransaction tr = //lookup
Bean bean = //lookup
tr.begin();
bean.doSth();
tr.commit();
```

bottega IT SOLUTIONS

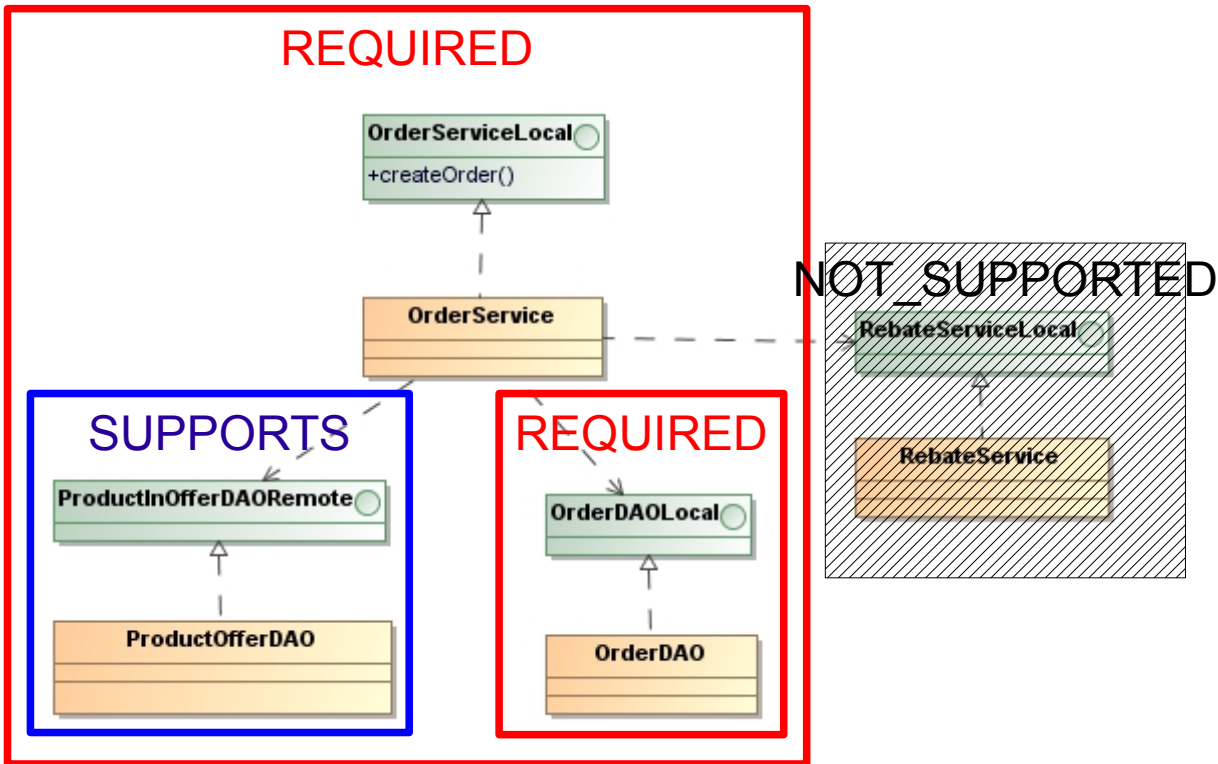
TX propagation

- Cross component method calls – Container decides
  - CMT->CMT – depends on policy (required by default)
  - BMT-CMT – always propagated
  - CMT-BMT – never propagated

- Transakcje na warstwie API (serwisów)
  - Serwisy REQUIRED
  - Odczyt SUPPORTS
- Transakcje zorientowane na współbieżność
  - Skracanie czasu w transakcji
  - Odczyty najpierw (SUPPORTS)
  - Zapis później (REQIRED)
- Transakcje zorientowane na wydajność
  - Zarządzane przez bazę (local) – każda operacja jest zapisywane
  - Mechanizm kompensacyjny
- Transakcje sterowane przez kod kliencki
  -

Propagation policies

- **NotSupported –** existing TX is suspended
- **Supports –** attaching to existing TX
- **Never –** if TX exists than exception
- **Required** – attaching to existing or creating new
- **RequiresNew –** Creating new TX. Eventually suspending existing one.
- **Mandatory –** Attaching to existing, if not exists than exception



- REQUIRED - **domyślnie**
  - Jeżeli transakcja nie istnieje, to wówczas jest tworzona
  - Jeżeli metoda rozpoczyna transakcję to musi również ją zakończyć (commit/rollback)
  - Jedna transakcja fizyczna
  - Podczas propagacji są tworzone osobne logiczne transakcje
    - Marker rolbackOnly per metoda
    - Zewnętrzna metoda może obsłużyć wyjątek wewnętrznej
- REQUIRES_NEW
  - Jeżeli transakcje istnieje to jest zawieszana
  - Osobne transakcje fizyczne
  - Zewnętrzna metoda może kontynuować pomimo rollback wewnętrznej – jeżeli przechwyci wyjątek
- NESTED: http://forum.springsource.org/archive/index.php/t-16594.html
  - Jedna transakcja fizyczna + Savepoints
  - Commit na końcu całości
  - Zewnętrzna metoda może kontynuować pomimo rollback wewnętrznej

REQUIRED

```
@Autowired
private MyDAO myDAO;

@Autowired
private InnerBean innerBean;

@Override
@Transactional(propagation=Propagation.REQUIRED)
public void testRequired(User user) {
  myDAO.insertUser(user);
  try{
    innerBean.doRequired();
  } catch(RuntimeException e){
    // handle exception
  }
}

@Override
@Transactional(propagation=Propagation.REQUIRED)
public void doRequired() {
  throw new RuntimeException("Rollback this transaction!");
}
```

RuntimeException **oznacza transakcję logiczną do wycofania (mimo catch) – żadne dane nie będą zapisane**

REQUIRES_NEW

```
@Autowired
private MyDAO myDAO;

@Autowired
private InnerBean innerBean;

@Override
@Transactional(propagation=Propagation.REQUIRED)
public void testRequiresNew(User user) {
  myDAO.insertUser(user);
  try{
    innerBean.doRequiresNew();
  } catch(RuntimeException e){
    // handle exception
  }
}

@Override
@Transactional(propagation=Propagation.REQUIRES_NEW)
public void doRequiresNew() {
  throw new RuntimeException("Rollback this transaction!");
}
```

RuntimeException **wycofuje transakcję fizyczną, ale nie wpływa na transakcję metody zewnętrznej**

NESTED

```
@Autowired
private MyDAO myDAO;

@Autowired
private InnerBean innerBean;

@Override
@Transactional(propagation=Propagation.REQUIRED)
public void testNested(User user) {
  myDAO.insertUser(user);
  try{
    innerBean.doNested();
  } catch(RuntimeException e){
    // handle exception
  }
}

@Override
@Transactional(propagation=Propagation.NESTED)
public void donested() {
  throw new RuntimeException("Rollback this transaction!");
}
```

RuntimeException wycofuje do savePoint.
Gdyby nie został przechwycony, wówczas wycofa całość.

- SUPPORTS
  - Może działać bez kontekstu transakcji
  - Jeżeli kontekst istnieje to jest do niego dołączana (widoczność Session/Entitymanager)
  - Odpowiednia propagacja do odczytów
    - Metoda odczytująca „widzi” dane zapisane (jeszcze bez commit) przez metodę nadrzędną – dane z transaction log
    - Przy braku transakcji odczyt nastąpi z danych z tabeli
- MANDATORY
  - Dołącza się do istniejącego kontekstu transakcji
  - Jeżeli takowy nie istnieje to wyjątek
  - Odpowiednia propagacja dla transakcji zarządzanych przez kod kliencki

- NOT_SUPPORTED
  - Zawiesza istniejącą transakcję (brak widoczności zasobów taki jak Session)
  - Odpowiednie dla wywołania procedur, które same zarządzają transakcjami i łącznie (chaining) nie jest wspierane
- NEVER
  - Wyjątek gdy transakcja istnieje
  - Zastosowanie: ???
    - Testowanie: jeżeli metoda zwraca wyjątek, oznacza to, że kontekst transakcji istniał

Tips

- REQUIRES_NEW
  - Zawsze jest tworzona nowa transakcja
    - Jeżeli używamy JPA to pracujemy z nową instancją EntityManager/Session (osobny L1 cache)
  - Uwaga na rekursje!
  - Możemy niezależnie sterować izolacją takiej transakcji
    - Pozwala nakładać silną izolację na krótsze jednostki czasu (np. generator kluczy)
- NESTED
  - Przetwarzanie dużych ilość „pod-ścieżek”
- Warto używać NOT_SUPPORTED i NEVER
- XA tylko gdy naprawdę ich potrzebujemy
  - Warto używać obu managerów transakcji: XA i local

@Transactional - attributes

- **value** – (opcjonalnie) wsazanie Tx Managera (jeżeli zdefiniowano kilka w systemie)
- **propagation** – polityka propagacji (domyślnie REQUIRED)
  - **required**, **requires_new** (niezależne transakcje, zewnętrzna jest zawieszana), **nested** (jdbc savepoints)
- **isolation** – poziom izolacji (domyślny ba źródła)
- **readOnly** – true w celu optymalizacji zasobów DB
- **timeout**
- **rollbackFor/noRollbackFor** – klasy wyjątków (nie)wycofujących transakcję

Traps

http://www.ibm.com/developerworks/java/library/j-ts1/index.html?ca=drs-

- ReadOnly dla JDBC
  - Nie ma sensu z SUPPORTS
    - supports podłącza się do istniejącej transakcji, która domyślnie istnieje w tym kontekście
  - Działa tylko dla REQUIRED
    - Zatem readOnly wymusza stosowanie transakcji w ogóle
    - Co powoduje zbędny narzut
- RadOnly dla JPA
  - Ma sens jedynie z SUPPORTS – domyślnie mamy REQUIRED
  - Generalnie: zależy od implementacji JPA

bottega IT SOLUTIONS

TX performance tips

- REQUIRES_NEW – potentially danger when multiple calls (loop, recursion!)
- Use NOT_SUPPORTED i NEVER
  - Useful for reading (ex. reporting query)
  - Some mapping steps can be omitted
- Use Container Managed Transactions – optimized by server
- Use XA only if needed
  - Use both: XA and local

TX traps

JVM memory model is not transactional, therefore if **rollback**, than remember:

- Generated ID will be **recycled**, but **remains** in your objects
- @Version will be **recycled**, but **remains** in your objects

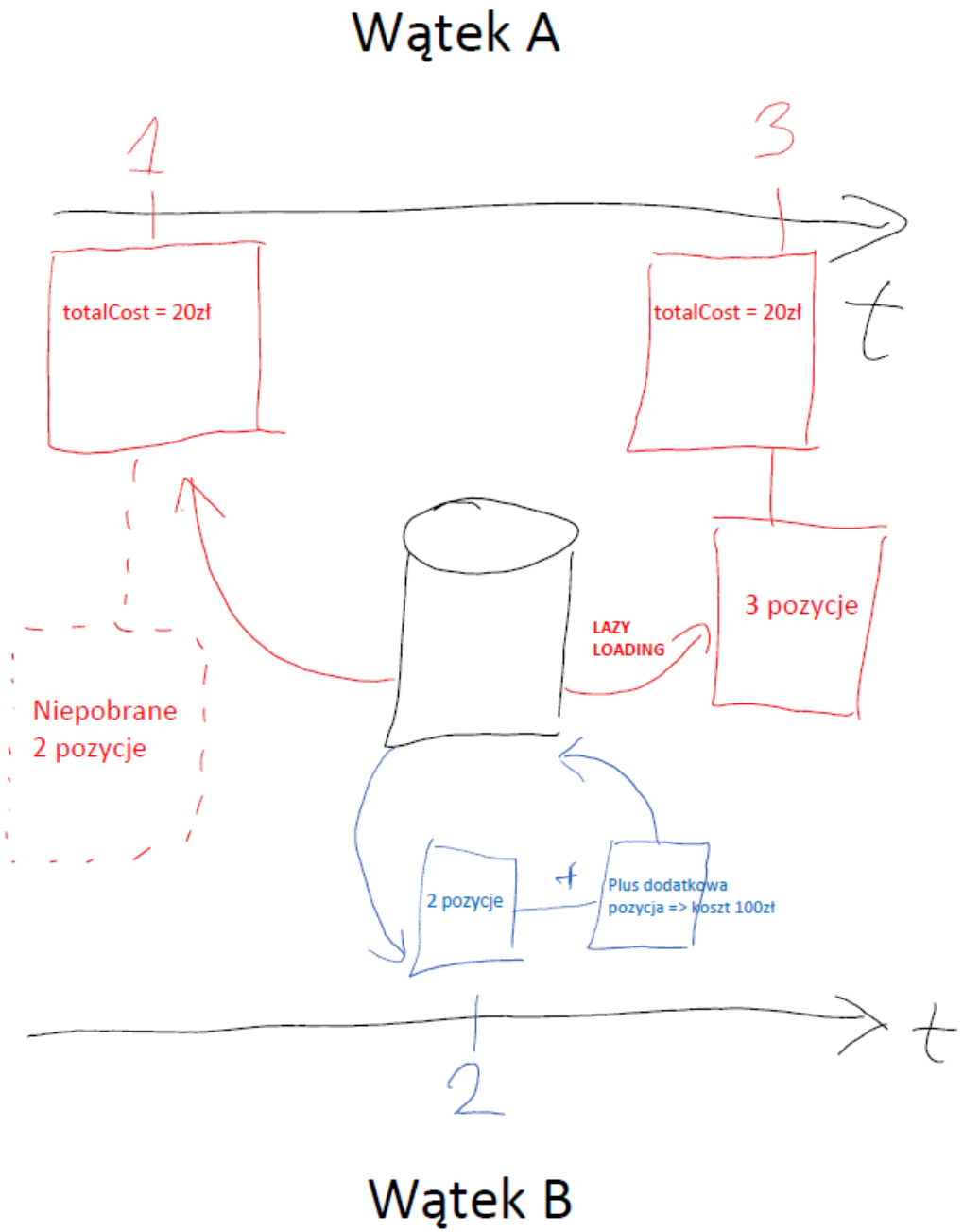
Its application job to handle objects.

Locking

You will learn:

- Types of locks: optimistic, pessimistic
- When to use
- API

Fail Scenario



- One thread loads data
  - Data won't be changed
  - Just load an Order to issue an Invoice
  - Therefore @Version won' help
- Another thread changes data that are aggregated inside First one
- Problems
  - Root (total cost) is not relevant to inner structures (order items)
  - Invoice is based on Order Header and Items that comes from different moments in time
- Solutions:
  - EAGER load (join, subselect, second select?)
  - Recalculate total cost every time (what if calculation takes time?)
  - ...

Locking

- @Version field incremented on UPDATE
  - Trap: Timestamp – db dependent, problems in cluster environment
- Optimistic and Pessimistic via API
  - EntityManager.lock()
  - passing a LockModeType to an EntityManager find() or refresh()
  - setting the lockMode of a Query or NamedQuery.

bottega IT SOLUTIONS

Locking - philosophy

- Optimistic (@Version, em.lock(READ/WRITE))
  - By application
  - Concurrency is exceptional
    - We assume that it wont happen
    - But if, than Exception
- Pessimistic (em.lock(PESIMISTIC...))
  - By DB
  - Concurrency is expected but not allowed

Optimistic Locking

- Lock on commit
- It is sometimes desirable to lock something that you did not change
  - when making a change to one object, that **is based on the state of another object**
  - ensure that the other object represents the current state of the database at the point of the commit

```
Employee employee = entityManager.find(Employee.class, id);
employee.setSalary(employee.getManager().getSalary() / 2);

entityManager.lock(employee.getManager(), LockModeType.READ);
```

bottega IT SOLUTIONS

Another fail scenario

- Order contains:
  - Client Data
  - Shipment Data
- There is a rule: some clients can't receive order in some places
- Problem:
  - Thread 1: changes Client Data
  - Thread 2: changes Shipment Data
- Solutions:
  - let's make Order dirty (temp field) - @Version will protect us
  - ...

Optimistic Locking Cascade Locks

- *WRITE* lock provides object-level protection
  - a **change** to a **dependent** object will conflict with any change to the **parent** object, or any other of its dependent object
  - can also be used to lock relationships (OneToMany or ManyToMany) - force the parent's version to be incremented

```
Employee employee = entityManager.find(Employee.class, id);
employee.getAddress().setCity("Ottawa");
entityManager.lock(employee, LockModeType.WRITE);
```

Optimistic Locking Semantics

- *READ* (JPA 2: OPTIMISTIC) lock will ensure that the **state** of the object **does not change on commit**
  - check the optimistic version field
  - Prevents Repeatable Read anomaly
- *WRITE* (JPA 2: OPTIMISTIC_FORCE_INCREMENT) lock will ensure that this transaction conflicts with any **other transaction changing** or **locking** the object
  - check **and increments** the optimistic version field

bottega IT SOLUTIONS

More general problem

- What the Object is?
  - Mesh of data structures
- That the REAL Object is
  - Hermetic
  - Clear boundary
    - No traversing through whole universe
    - Boundary should protect business invariants (rules)
      - Data that changes in consistent way based on domain rules

bottega
IT SOLUTIONS

Object with clear boundary

•Order is the root of whole structure
•It contains: OrderLines and Client Data
•But does not contain: Client and Product
- They are different Aggregates

bottega IT SOLUTIONS

Simple Rules for well designed objects

- Load:
  - EAGER
    - If boundary is well designed than it's ok – You will need all data anyway
    - Root and inner structure will come from the same moment in time (will be consistent)
      - One DB shoot: FETCH using JOIN not another select
      - Problem: can't join more that one Bag – use Set or List
  - OPTIMISTIC (READ) Lock
    - To be sure You relay on fresh data
- Save
  - Cascade=ALL, orphanRemoval = true
  - OPTIMISTIC_FORCE_INCREMENT (WRITE) Lock
    - To be sure that whole aggregate will be consistent

bottega IT SOLUTIONS

(almost) all You need to know about JPA (can be hidden in one abstract class:)

```
public abstract class GenericJpaRepository<A extends BaseAggregateRoot> {
    @PersistenceContext protected EntityManager entityManager;
    private Class<A> clazz;
    @Inject private AutowireCapableBeanFactory spring;
    @SuppressWarnings("unchecked") public GenericJpaRepository() {
        this.clazz = ((Class<A>) ((ParameterizedType) getClass().getGenericSuperclass()).getActualTypeArguments()
[0]);
    }
    public A load(AggregateId id) {
        A aggregate = entityManager.find(clazz, id, LockModeType.OPTIMISTIC);
        if (aggregate == null)
          throw new RuntimeException("Aggregate " + clazz.getCanonicalName() + " id = " + id + " does not
exist");
        if (aggregate.isRemoved())
          throw new RuntimeException("Aggragate + " + id + " is removed.");
        spring.autowireBean(aggregate);
        return aggregate;
    }

    public void save(A aggregate) {
     if (entityManager.contains(aggregate)){
          entityManager.lock(aggregate, LockModeType.OPTIMISTIC_FORCE_INCREMENT);
     }
     else{
          entityManager.persist(aggregate);
     }
    }

    public void delete(AggregateId id){
     A entity = load(id);
     entity.markAsRemoved();
    }
}
```

bottega
IT SOLUTIONS

Exercise: Locking

Pessimistic Locking

- Lock before you begin to edit the object
  - Row lock: SELECT ... FOR UPDATE SQL syntax
  - use database resources: connection and TX
  - typically not desirable for interactive web applications
  - can also have concurrency issues and cause deadlocks

Pessimistic Locking

- PESSIMISTIC_READ - The Entity is locked on the database, prevents any other transaction from acquiring a PESSIMISTIC_WRITE lock.
- PESSIMISTIC_WRITE - The Entity is locked on the database, prevents any other transaction from acquiring a PESSIMISTIC_READ or PESSIMISTIC_WRITE lock.
- PESSIMISTIC_FORCE_INCREMENT - The Entity is locked on the database, prevents any other transaction from acquiring a PESSIMISTIC_READ or PESSIMISTIC_WRITE lock, and the Entity will have its **optimistic lock version incremented** on commit. This is unusual as it does both an optimistic and pessimistic lock, normally an application would only use one locking model.

Pessimistic Locking Query hitns

Query, NamedQuery, or EntityManager find(), lock() or refresh() operation.

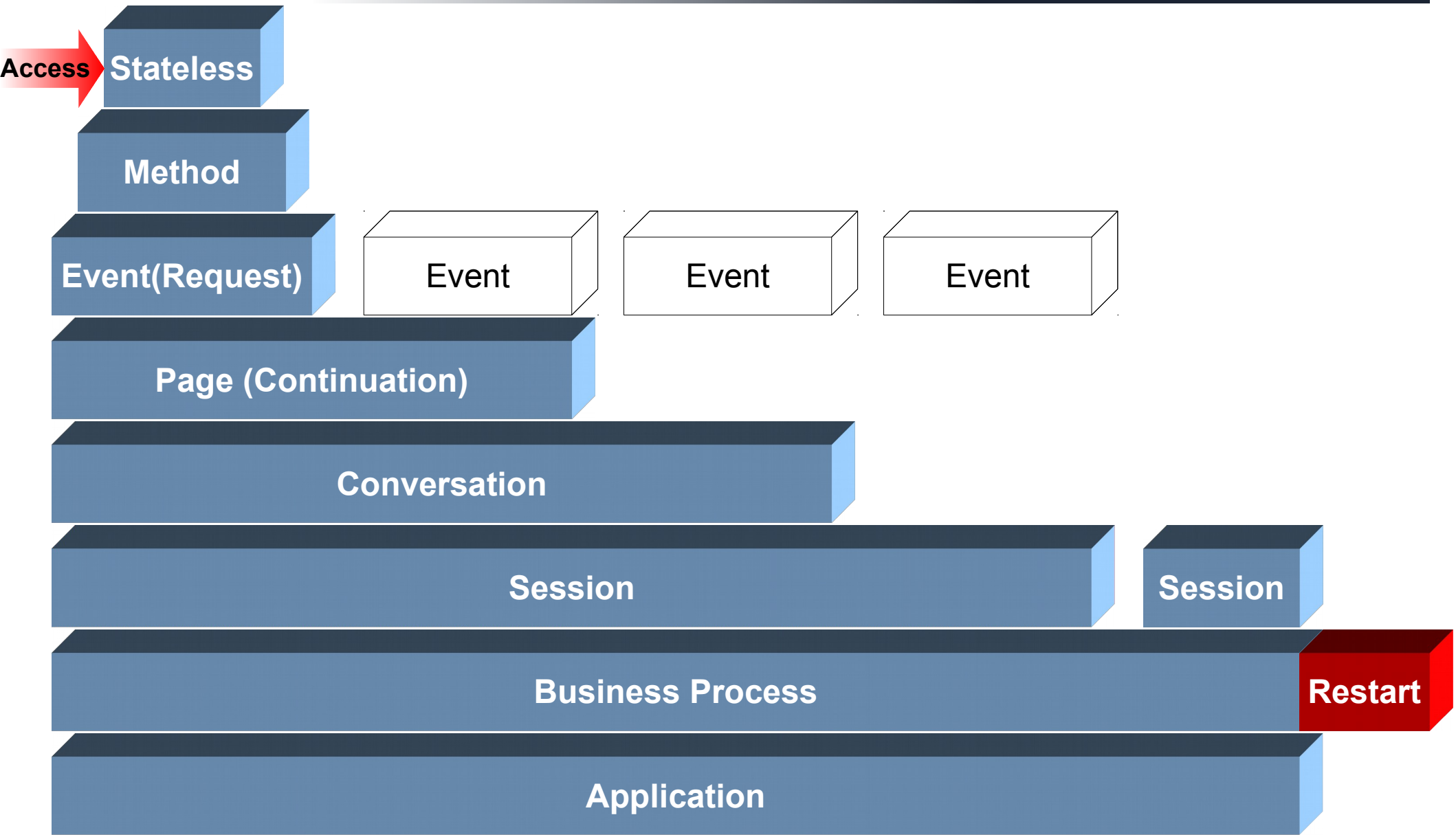
- "javax.persistence.lock.timeout" - milliseconds to wait on the lock before throwing a PessimisticLockException.
- "javax.persistence.lock.scope" – PessimisticLockScope: NORMAL or EXTENDED - will also lock the object's owned join tables and element collection tables.

Application Transactions

You will know:

- Modes of the Persistence Context
  - Transactional
  - Extended
- Conversation scope concept
- How to configure Extended Persistence Context
- How to use it as an Application Transaction

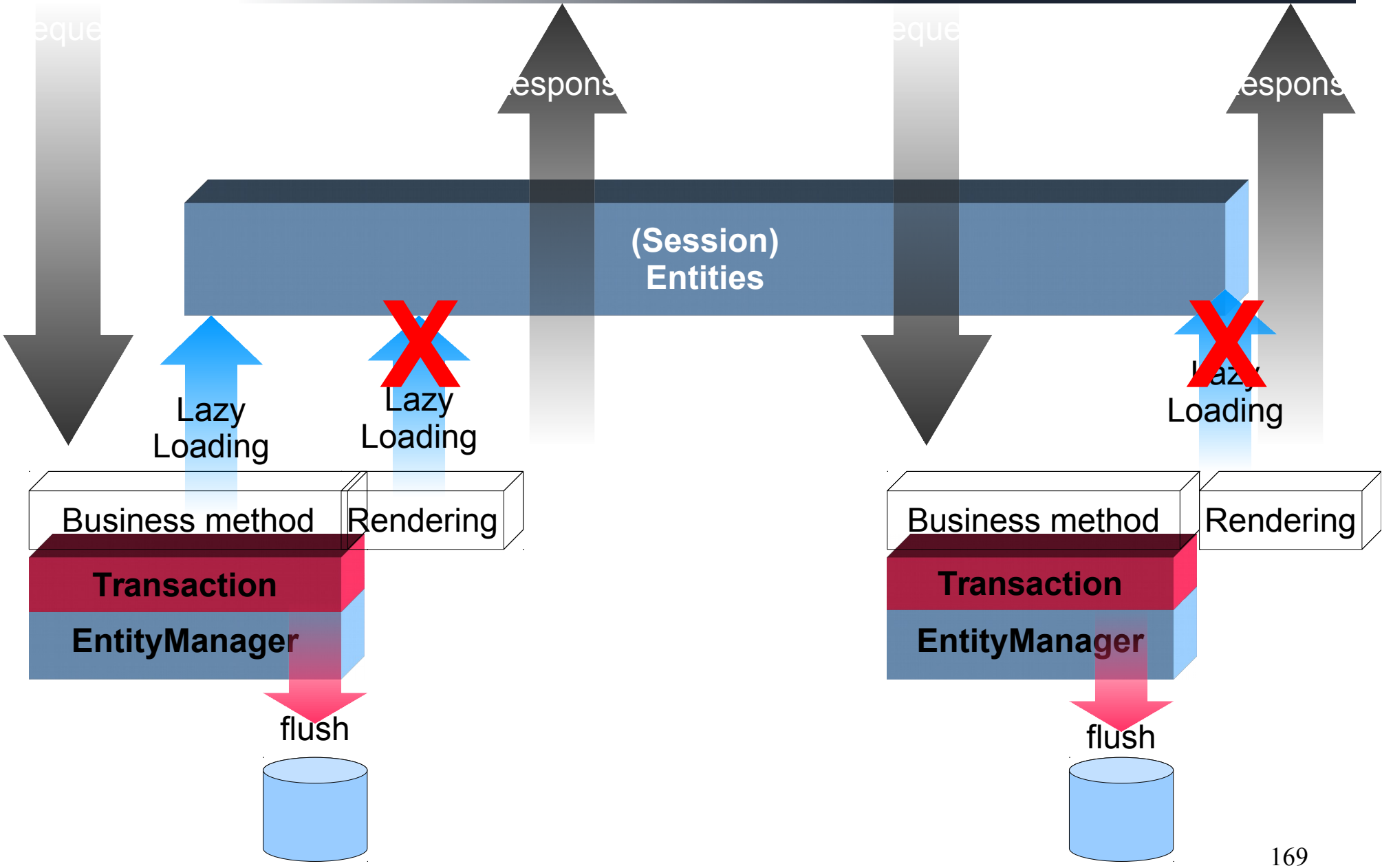
Sample Scopes (Seam)



Unit of Work

- EntityManager was designed for long life (longer than transaction)
  - should not be seen as instantaneous DB connection
  - but is not threadsafe
- EM and Conversation makes perfect match as Unit of Work
  - according to Hibenrate authors:)

Transactional Mode

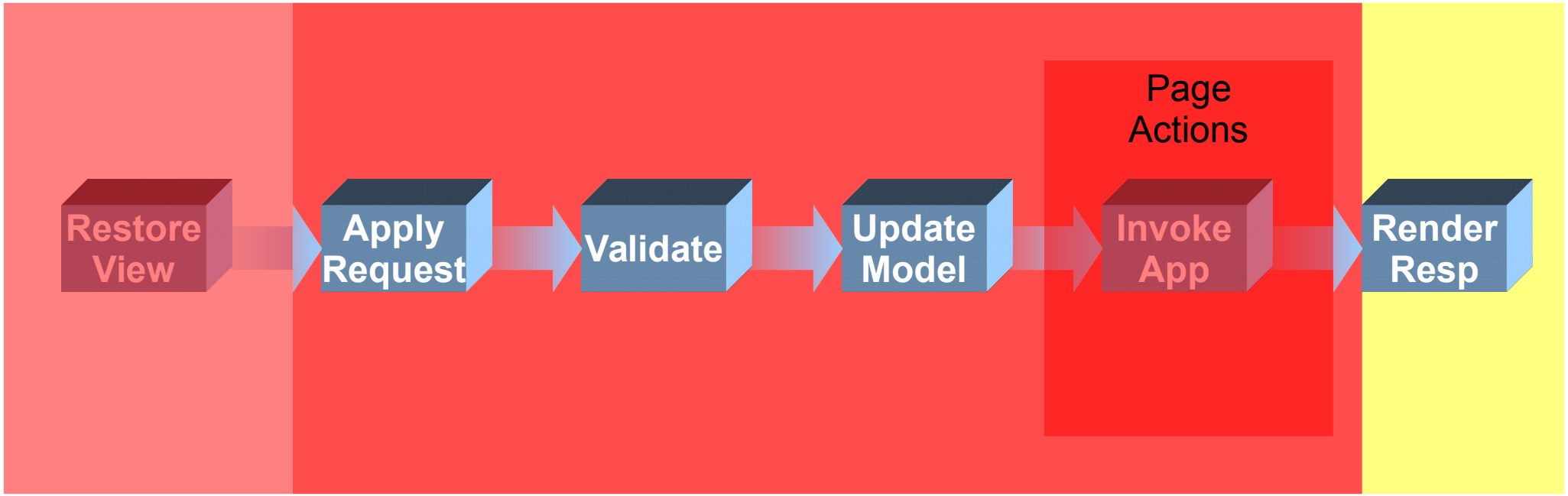


Extended Mode

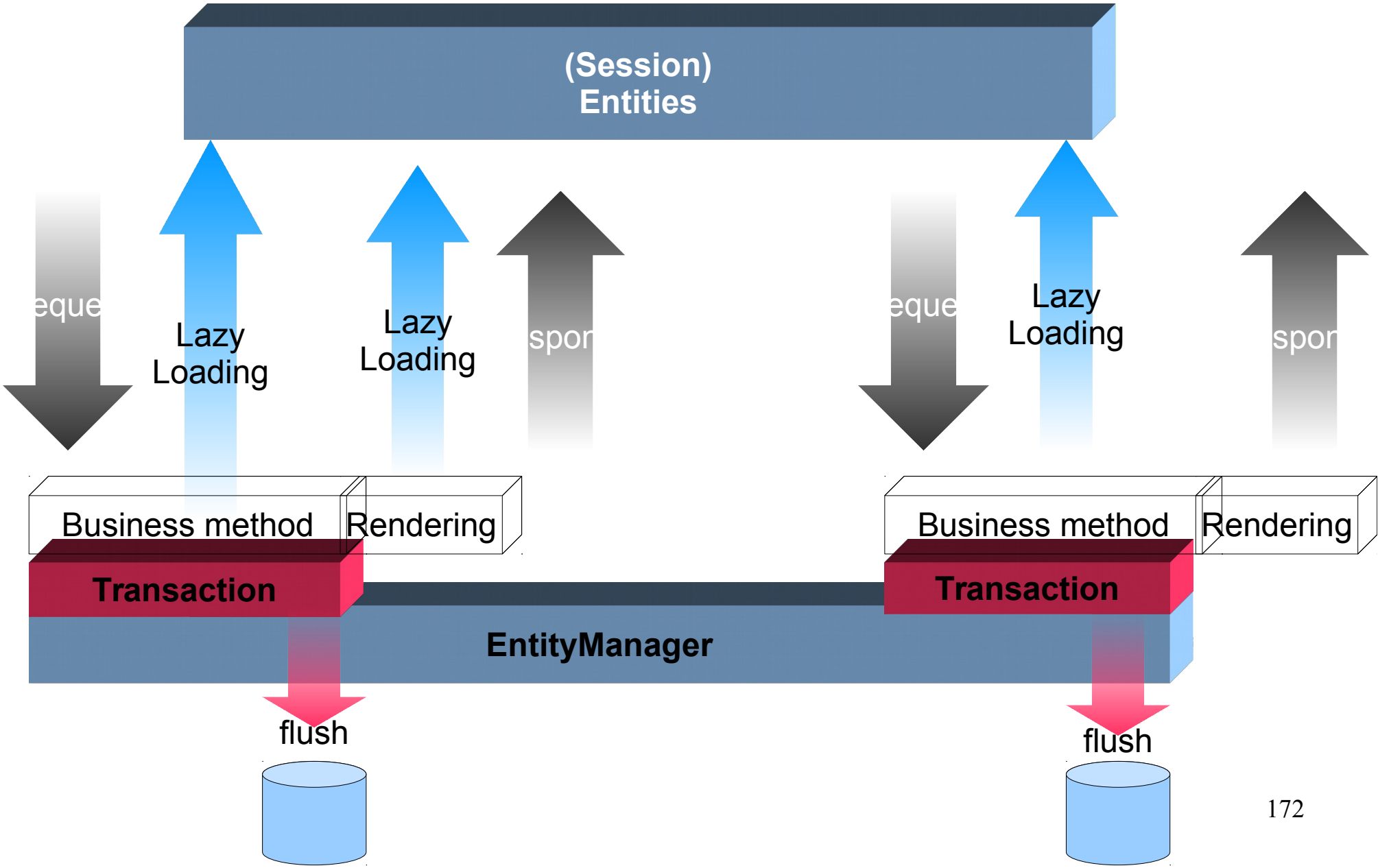
```
@Stateful
public class UserAgentBean implements UserAgent {

   @PersistenceContext(
      type = PersistenceContextType.EXTENDED)
   private EntityManager em;

}
```

- Entities are in Managed State
  - Lazy Loading works after reattachment
  - Seam supports special readonly TX for rendering
- DB communication reduction – no merges
  - **@Version is must have!**
- Utilisation of the L1 cache

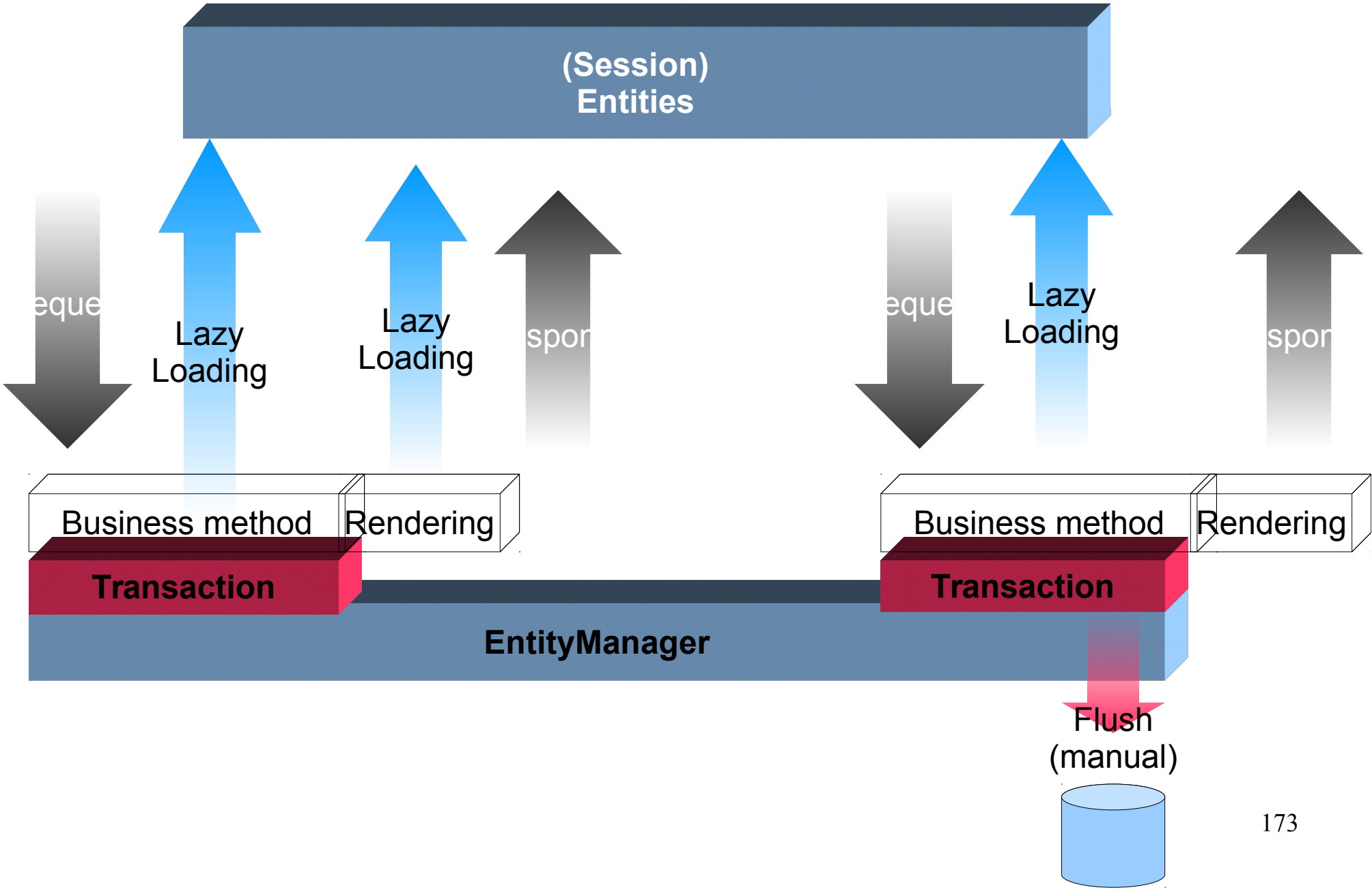
Open Session in View – Rendering TX
JSF lifecycle sample



Extended Mode
Unwanted flushing



Extended Mode and Manual Flush



Sample “conversation”

```
@Stateful
public class UserWizard implements Serializable {
   @PersistenceContext(type = PersistenceContextType.EXTENDED,
   properties=@PersistenceProperty(
      name="org.hibernate.flushMode", value="MANUAL"))
   private EntityManager entityManager;

   private User user;

   public void addUser() {
      user = ...
      entityManager.persist(user);
   }
   public void addBasicData() {
      user.addAddress(entityManager.find(Address.class, 1L));
   }

   @Remove public void save() {
      entityManager.flush();
   }
}
```

bottega IT SOLUTIONS

Exercise: Application Transactions

- Implement SFSB that models conversation with Cart
  - using conversation scoped Persistence Context

JPA 2: FlushMode on queries

Flushing is performed before query – in order to synchronize DB before quering. Sometimes it's unnecessary and slow.

```
public void archiveConversations(Date minAge) {
    List<Conversation> active =
    em.createNamedQuery("findActiveConversations",
                Conversation.class).getResultList();

    TypedQuery<Date> maxAge = em.createNamedQuery(
                "findLastMessageDate", Date.class);
    maxAge.setFlushMode(FlushModeType.COMMIT);

    for (Conversation c : active) {
        maxAge.setParameter("conversation", c);

        //calling this query
        Date lastMessageDate = maxAge.getSingleResult();

        //would flush this entity!
        if (lastMessageDate.before(minAge)) {
            c.setStatus("INACTIVE");
        }
    }
}
```

Valuable links

- http://what-when-how.com/hibernate/mapping-collections-and-entity-associations-hibernate/
- http://what-when-how.com/hibernate/advanced-query-options-hibernate/
- http://www.infoq.com/articles/hibernate_tuning
- http://www.mkyong.com/hibernate/hibernate-fetching-strategies-examples/
- https://community.jboss.org/wiki/HibernateFAQ-TipsAndTricks