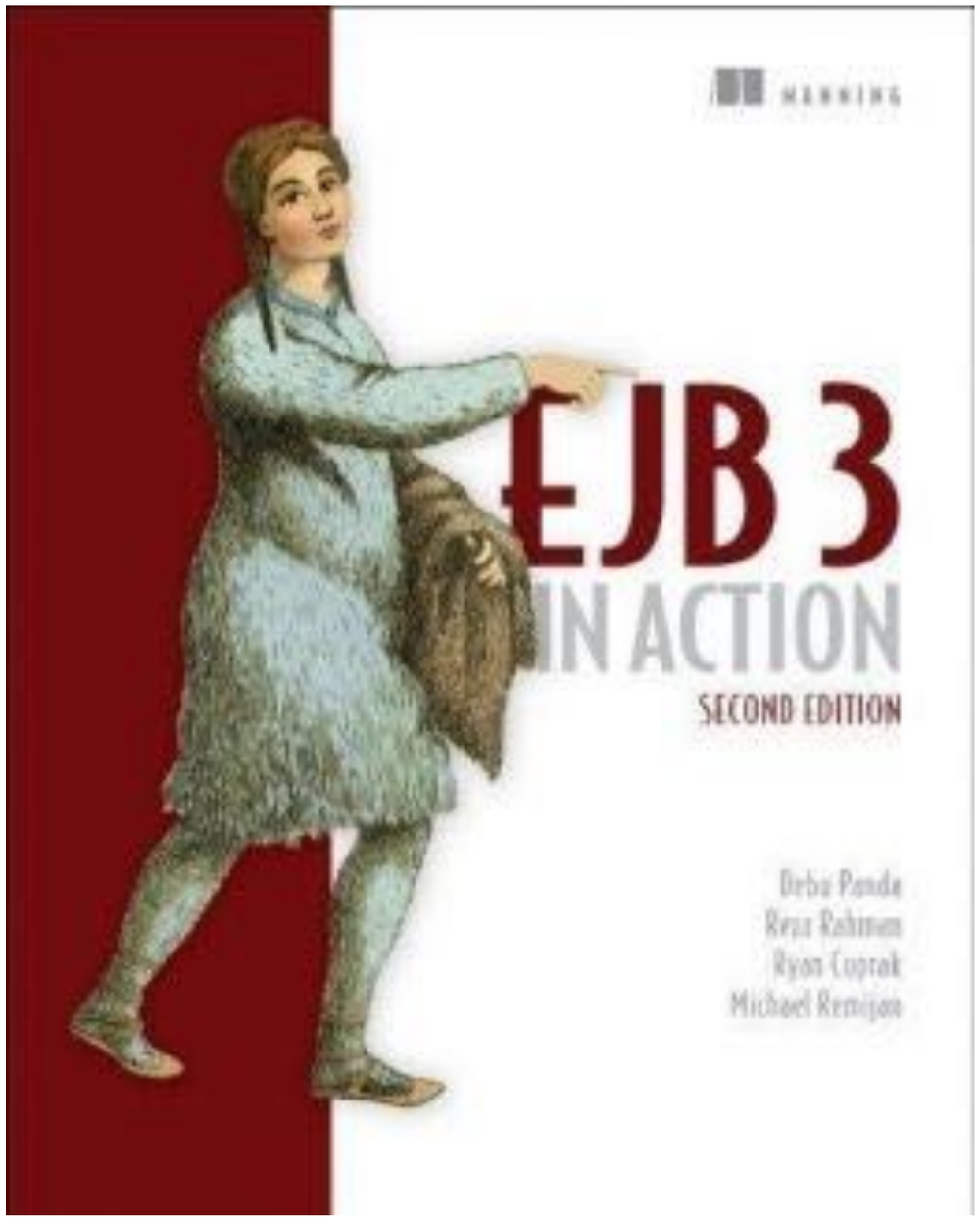
Java EE, EJB Basics, ORM Basics

Philippe Collet, contains 78,3% of slides from Sébastien Mosser

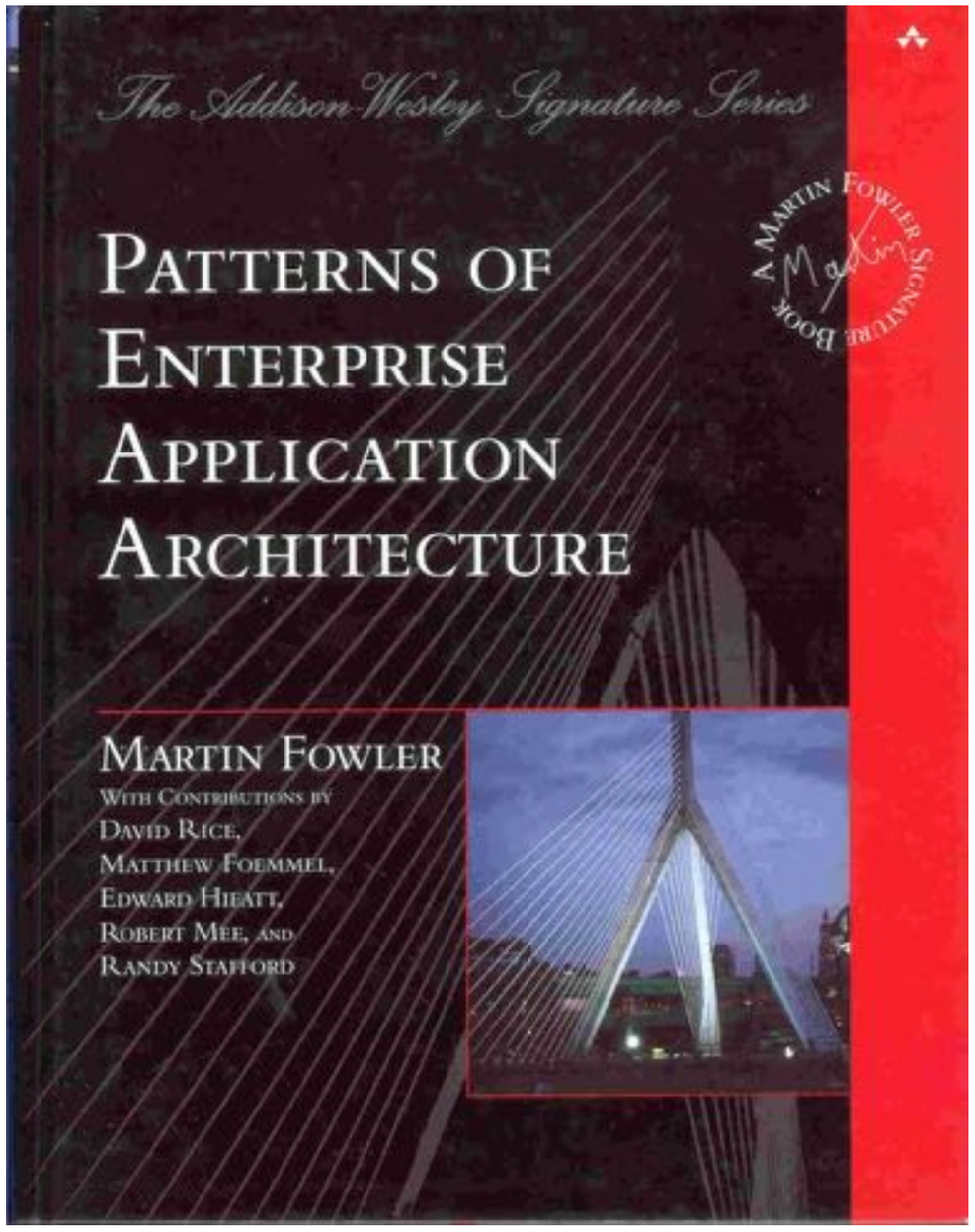
Lecture #2 15.02.2019

pictures: sxc.hu, pixabay

Bibliography

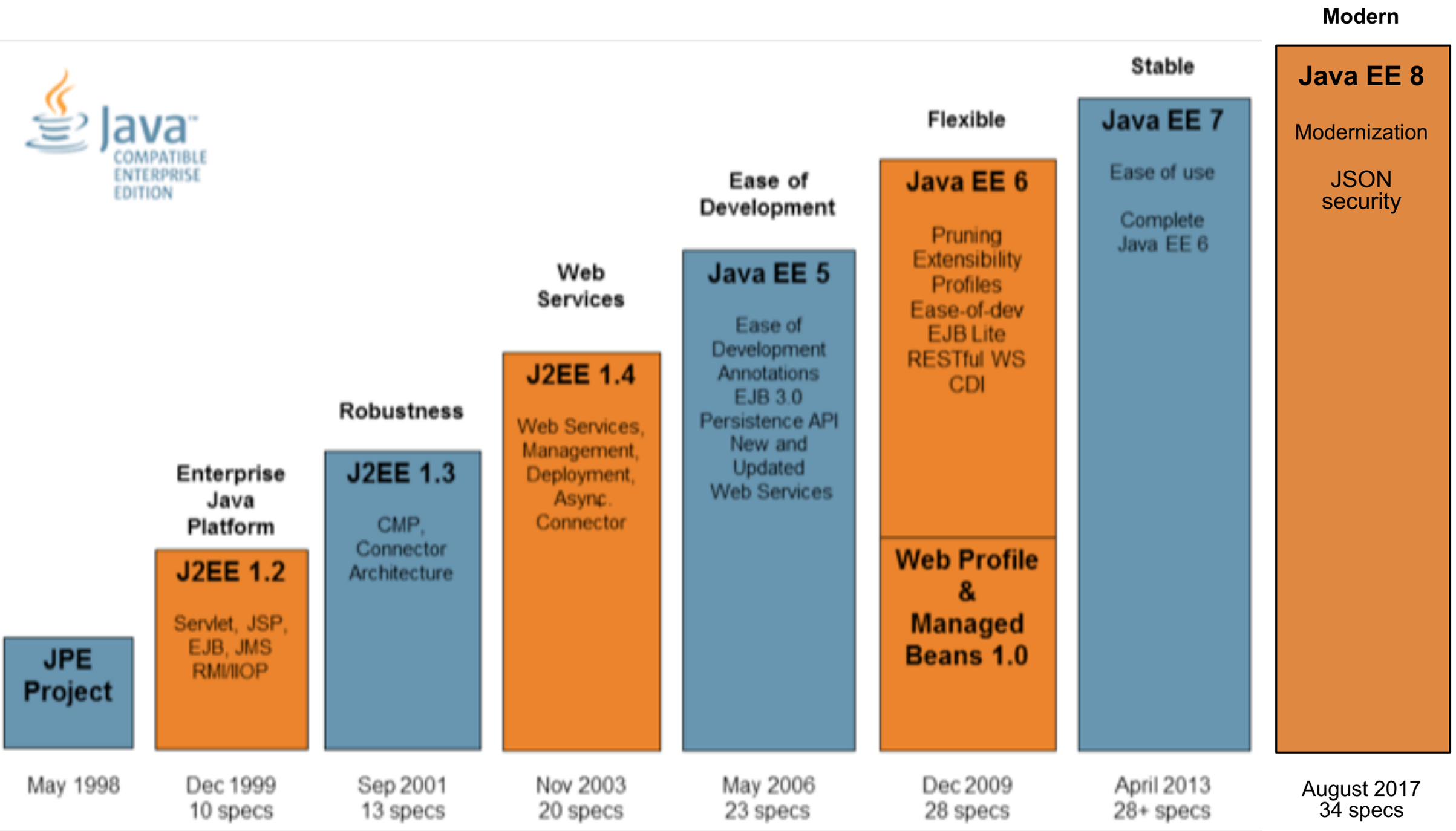


[EJB3 in Action, 2nd ed., 2014]

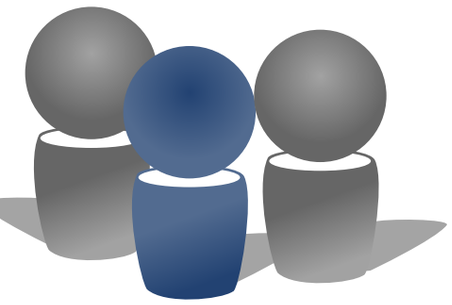


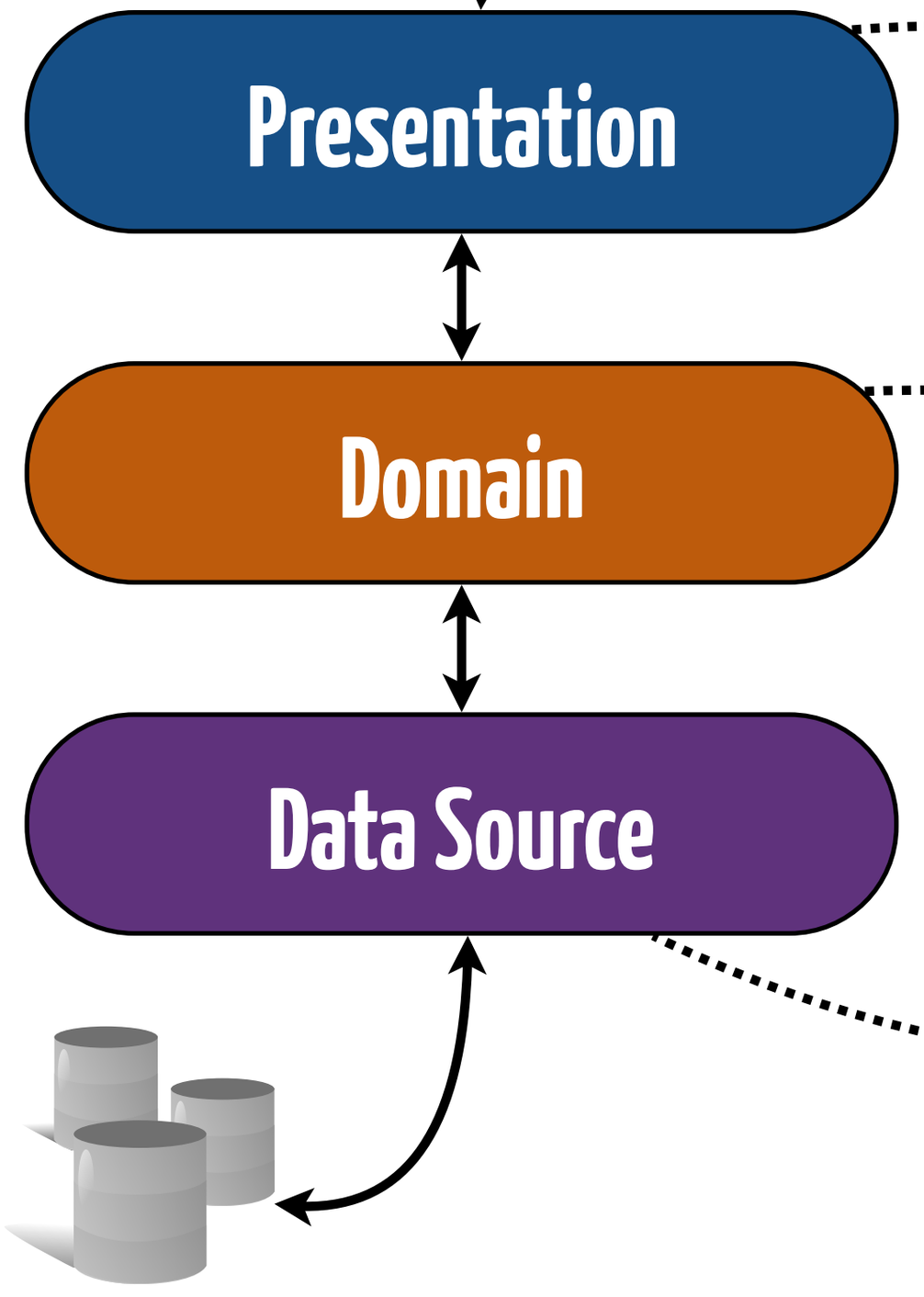
[PoEAA, 2002]

From J2EE to JavaEE



JavaEE: a 3-tiers architecture





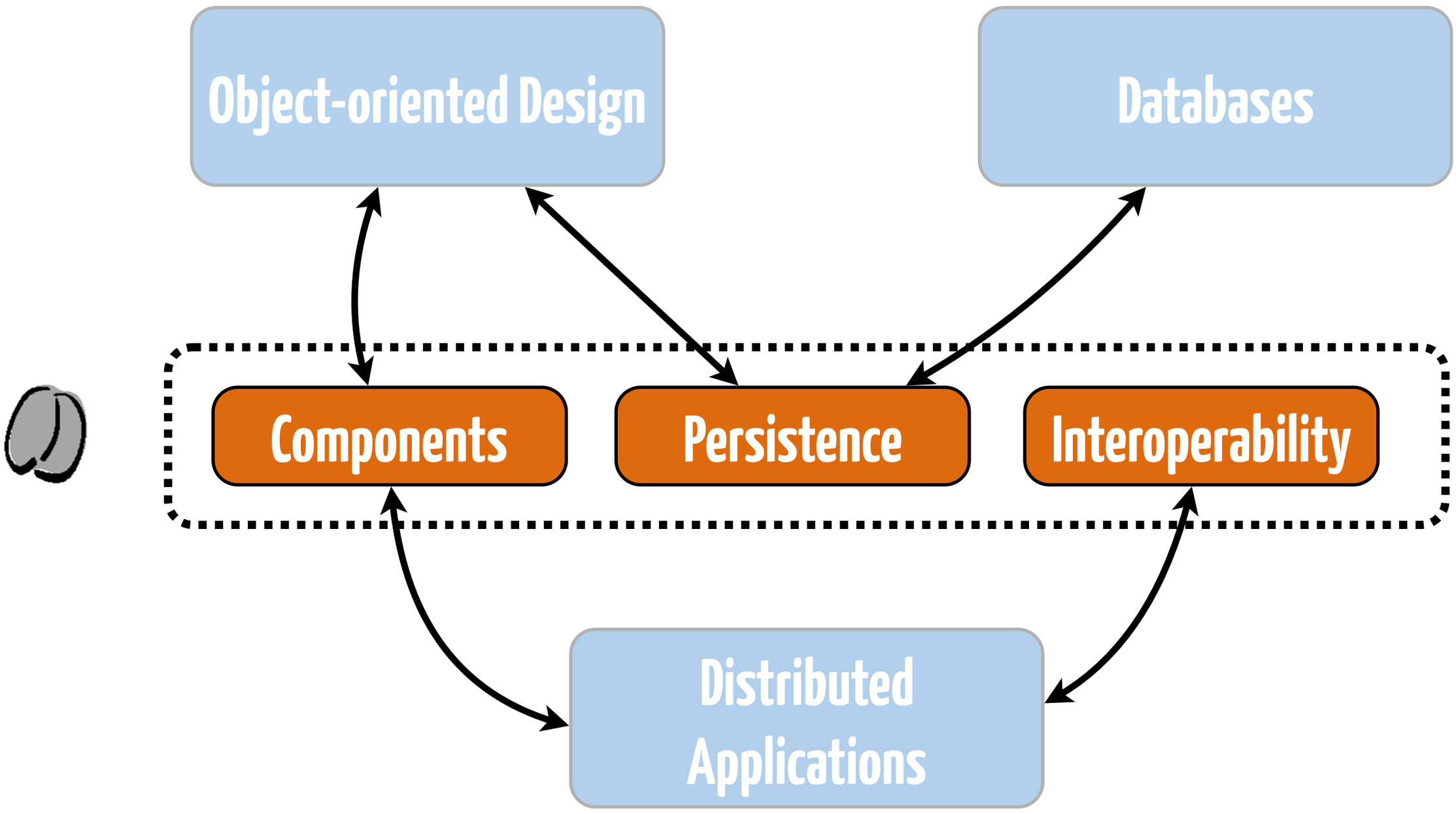
Handle users

Business functionality

Handle data storage

Enterprise Java Beans 101

Applications Server: Dependencies



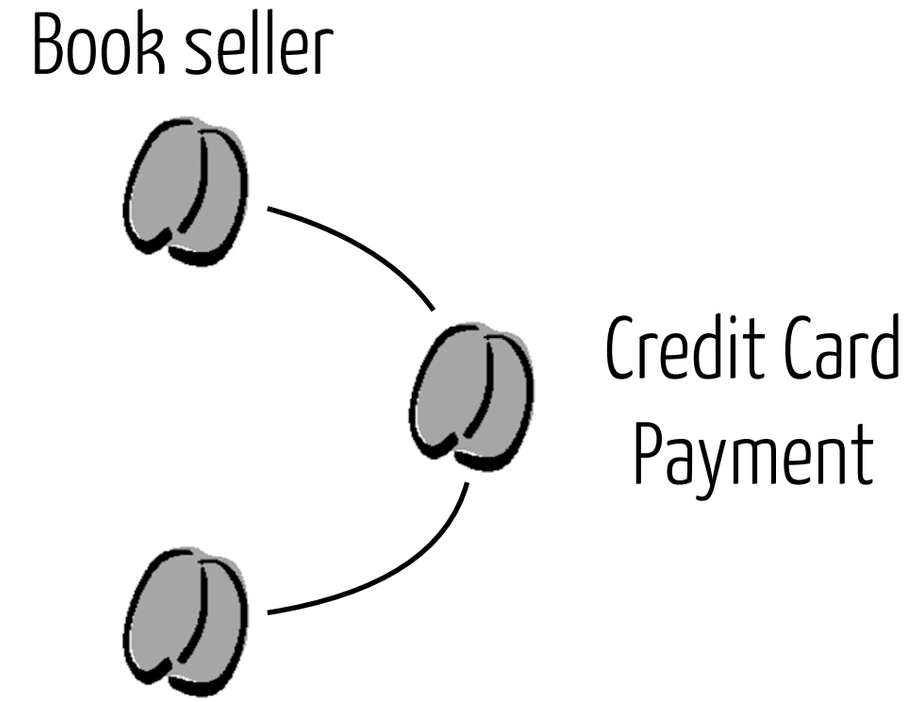
EJB as **Components**

Interface **reuse**

Encapsulate

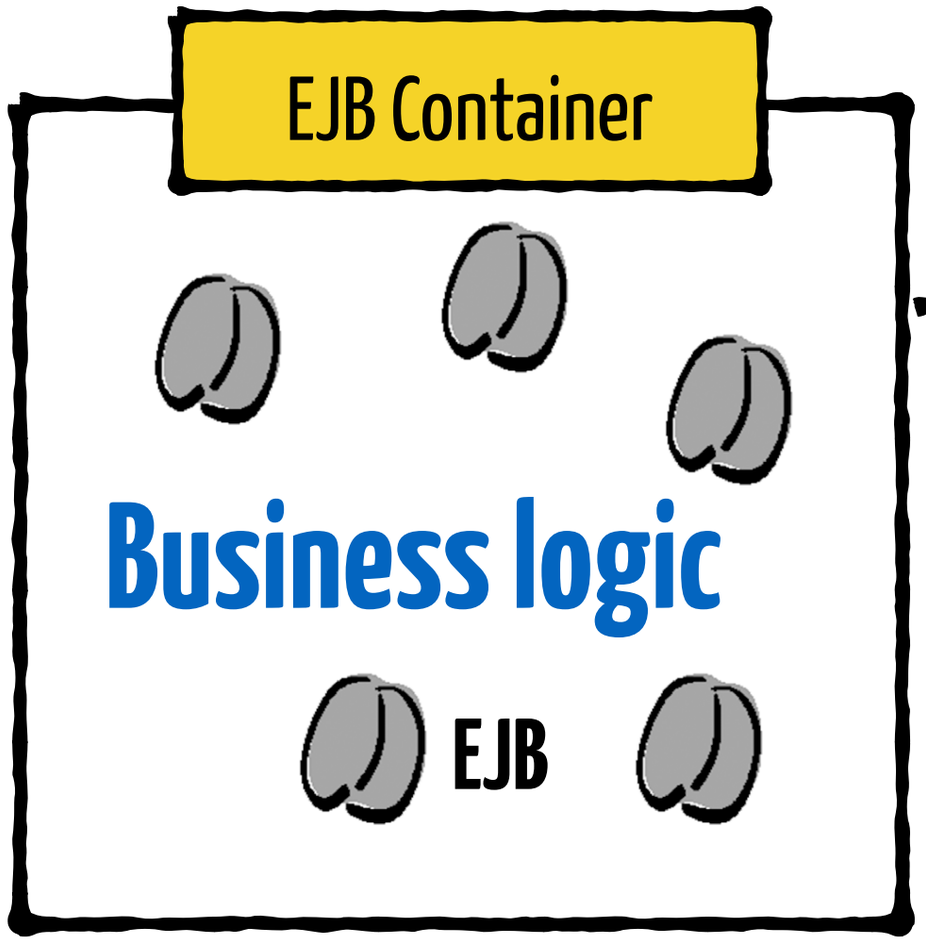
application

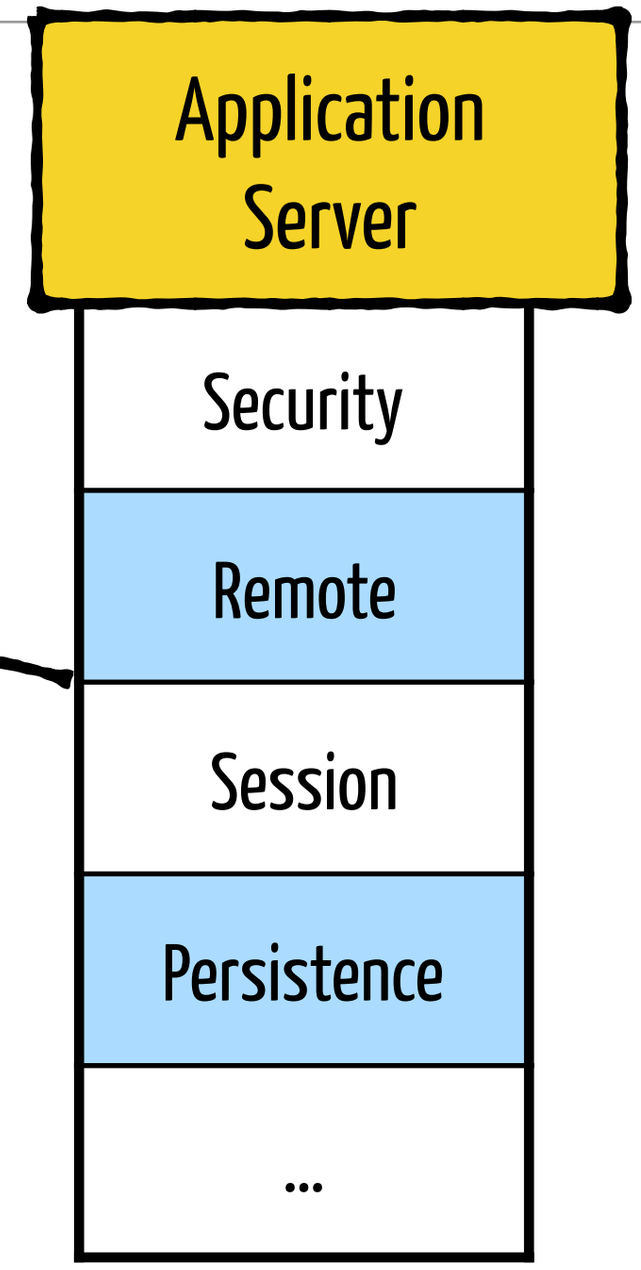
Behavior



JavaEE as the **specification** for

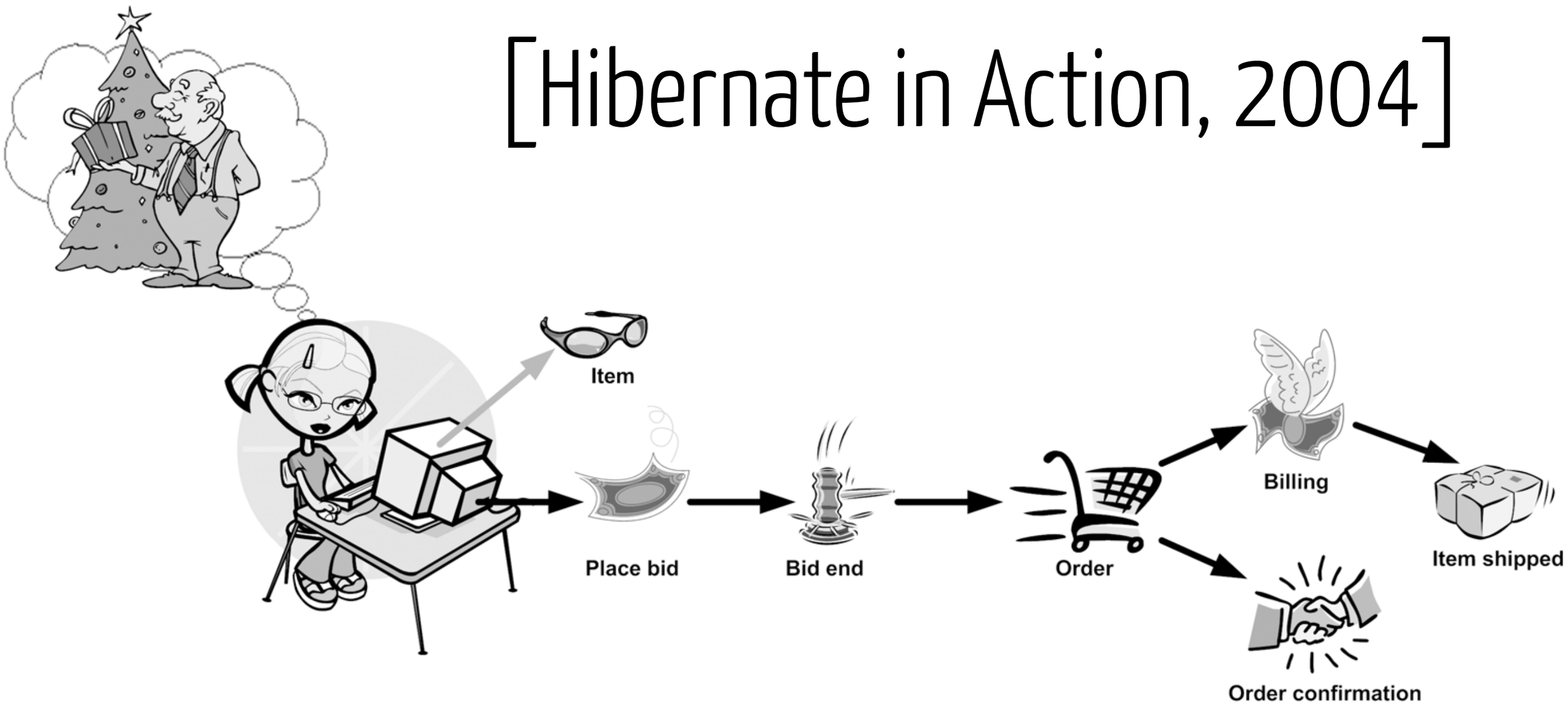
Life cycle management





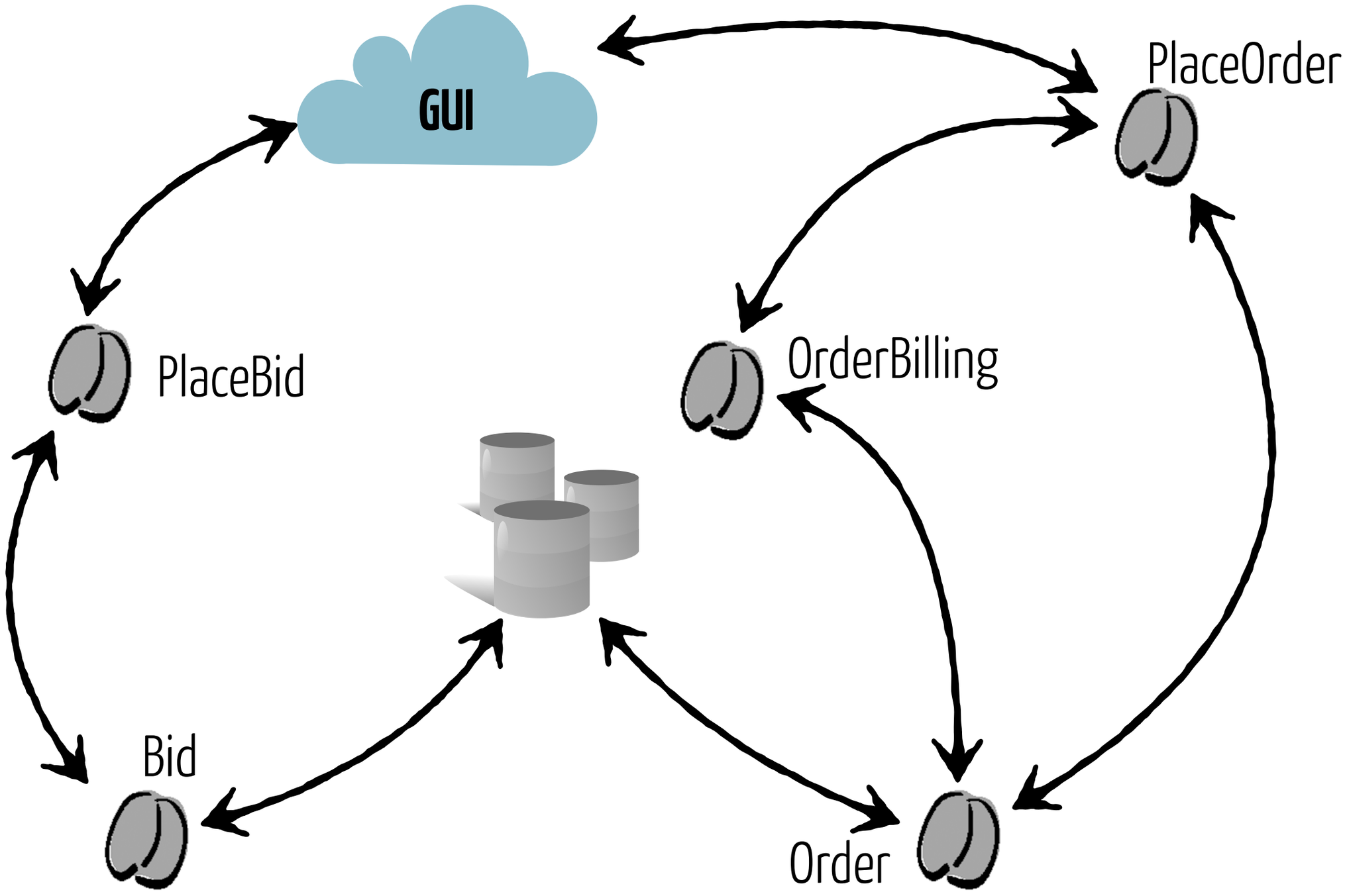
System level services

The ActionBazaar example

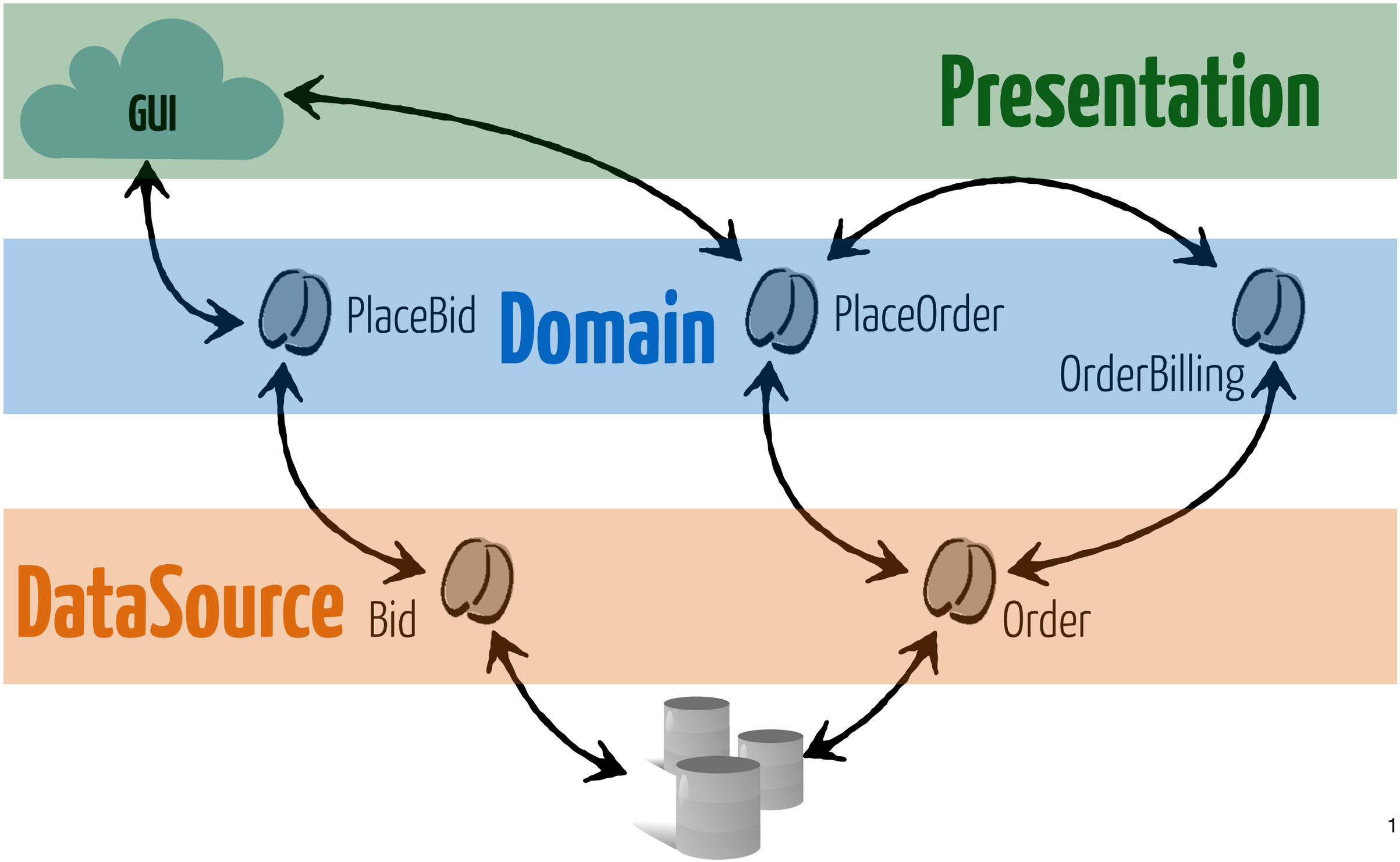


[Hibernate in Action, 2004]

Identifying **Components**



3-tiers Architecture



Rule of Thumb

Domain Beans interfaces as **Verbs**

DataSource Beans as **Nouns**

What is a **POJO**?

An ordinary Java Object

Not bound to any restriction such as

Extending some classes
Implementing some interfaces
Containing some annotations

What is a **Bean**?

A POJO that

is **serializable**

has **no-argument constructor**

allows access to properties using **getters / setters** with a **simple naming convention**

What is a **Entreprise Java Bean**?

POJO **Annotation** **EJB**

EJB v3 = "**Plain Old Java Object** on **Steroids**"

[EiA]

POJO on Steroids

```
public interface HelloUser {
   public void sayHello(String n);
}
```

```
@javax.ejb.Stateless
public class HelloUserBean implements HelloUser{
   public void sayHello(String n) {
      System.out.println("Hello, " + n "!");
   }
}
```

Different **Flavors**

Domain

Session Bean

Message-driven Bean

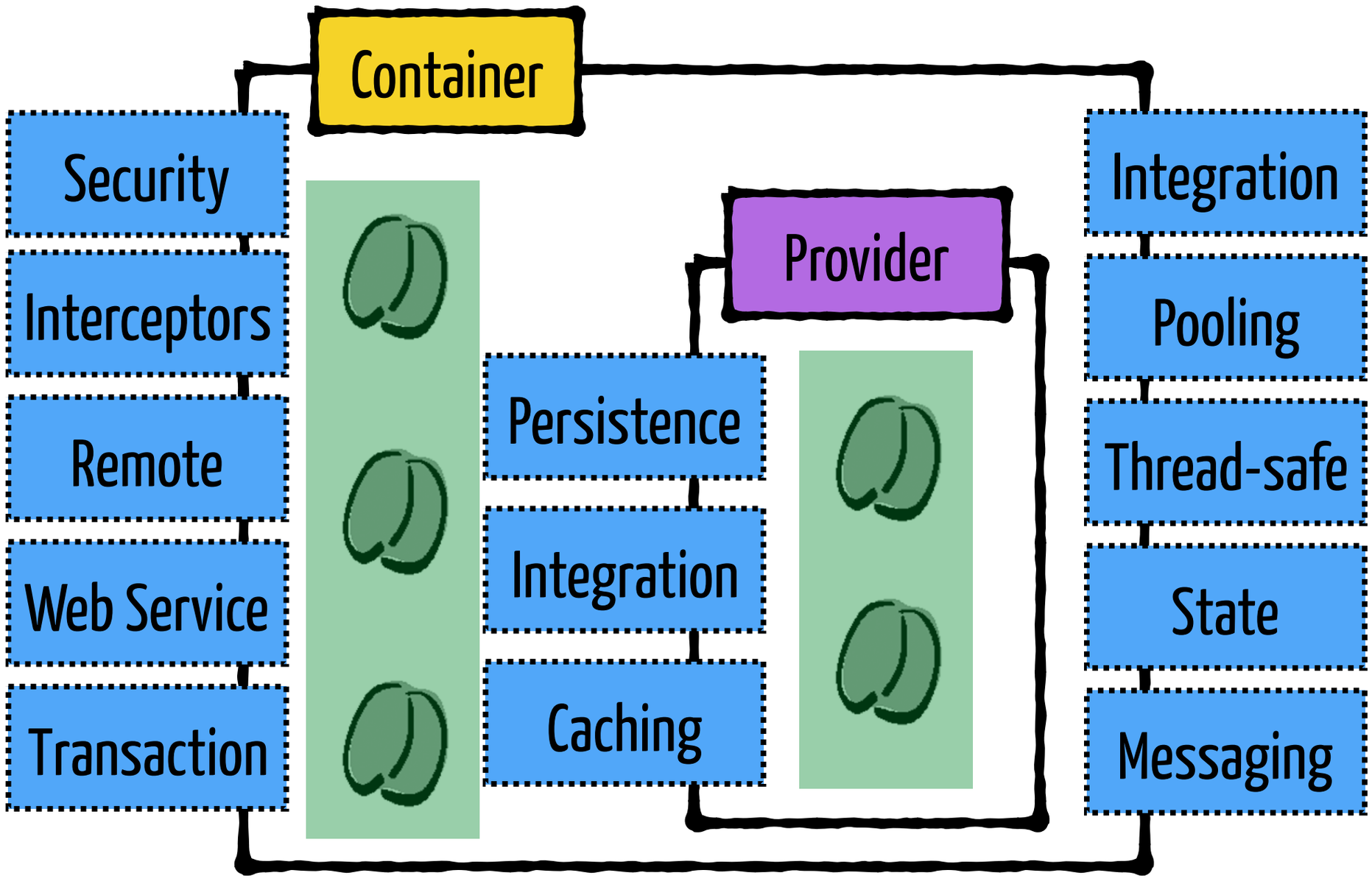
DataSource

Entity Bean

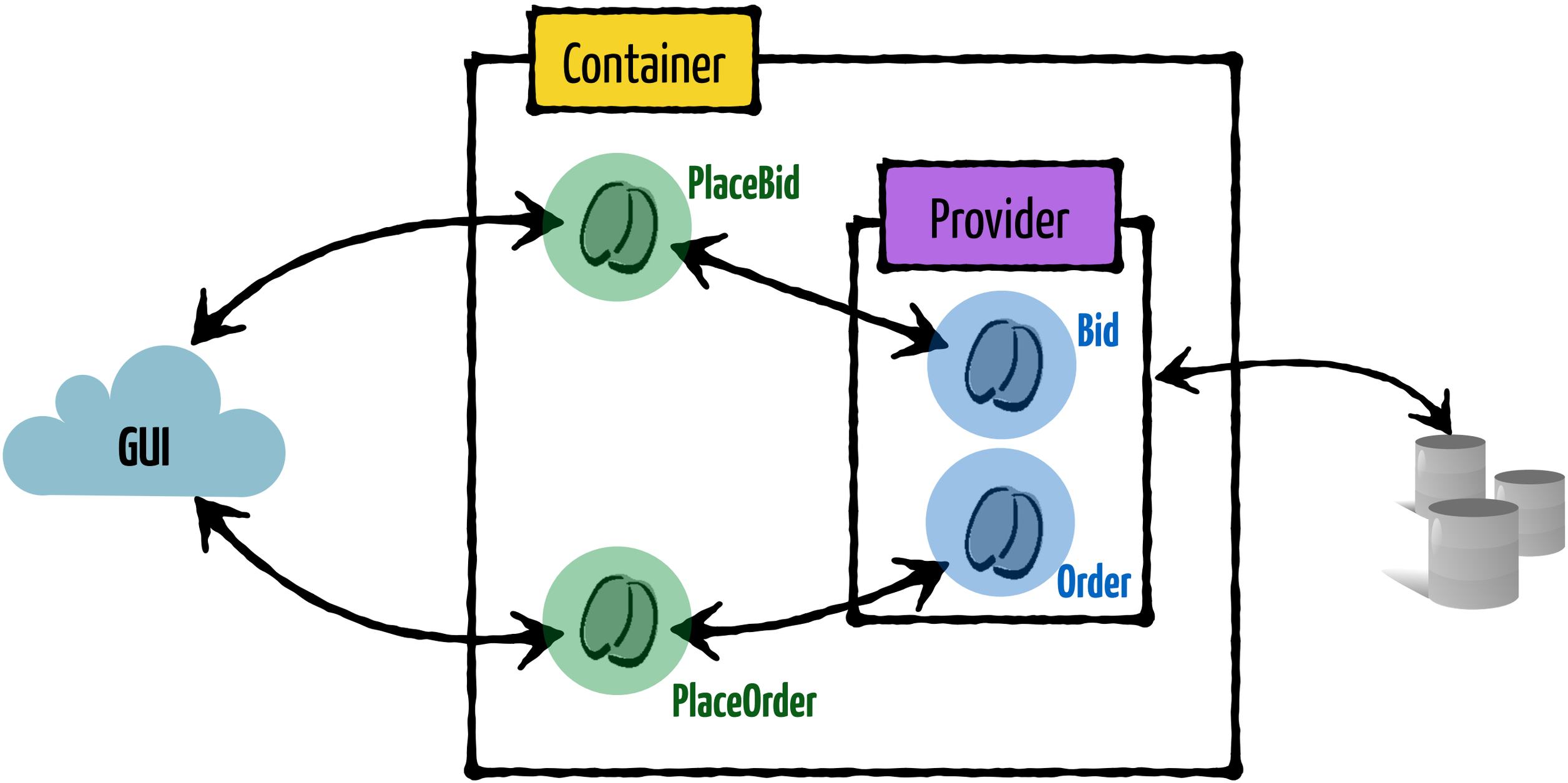
Container

Provider

EJBs **Services**: focus on **Business**!

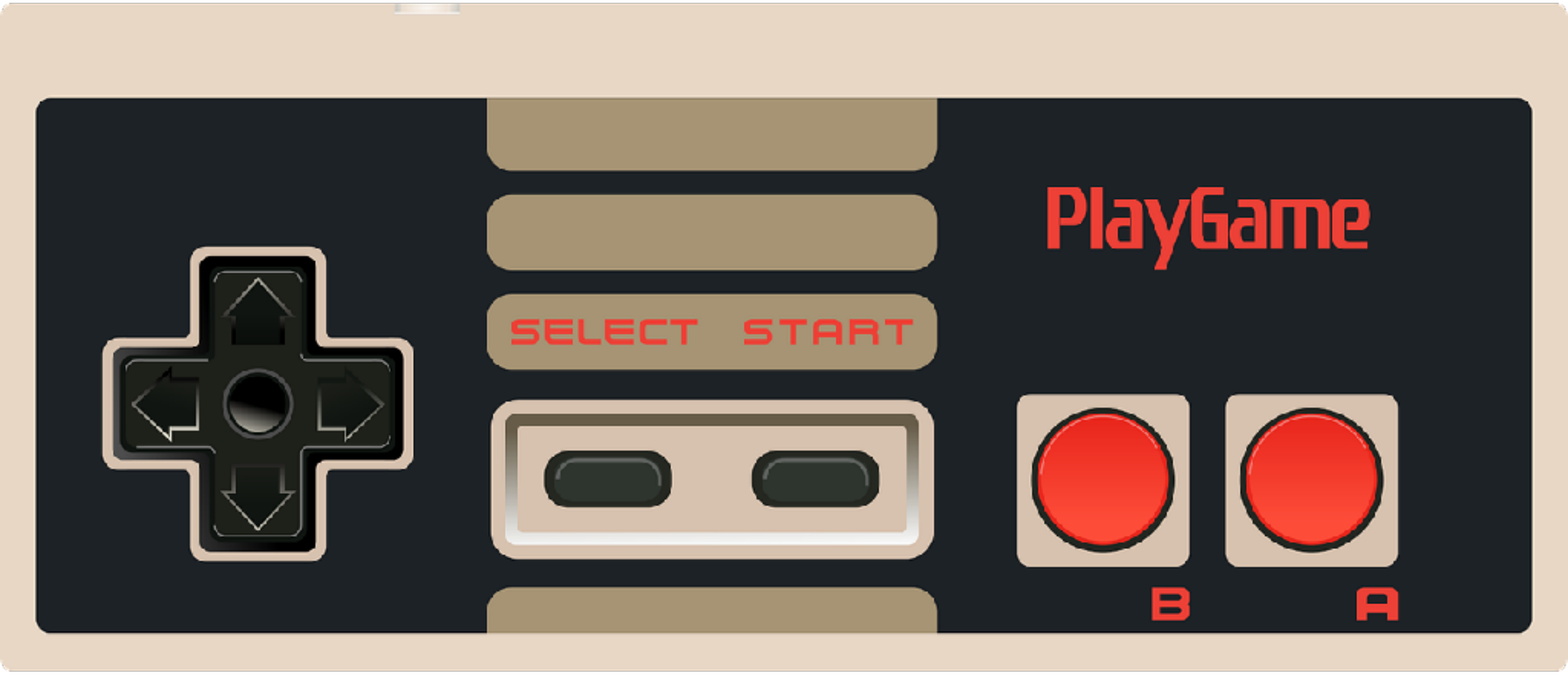


Session & Entity

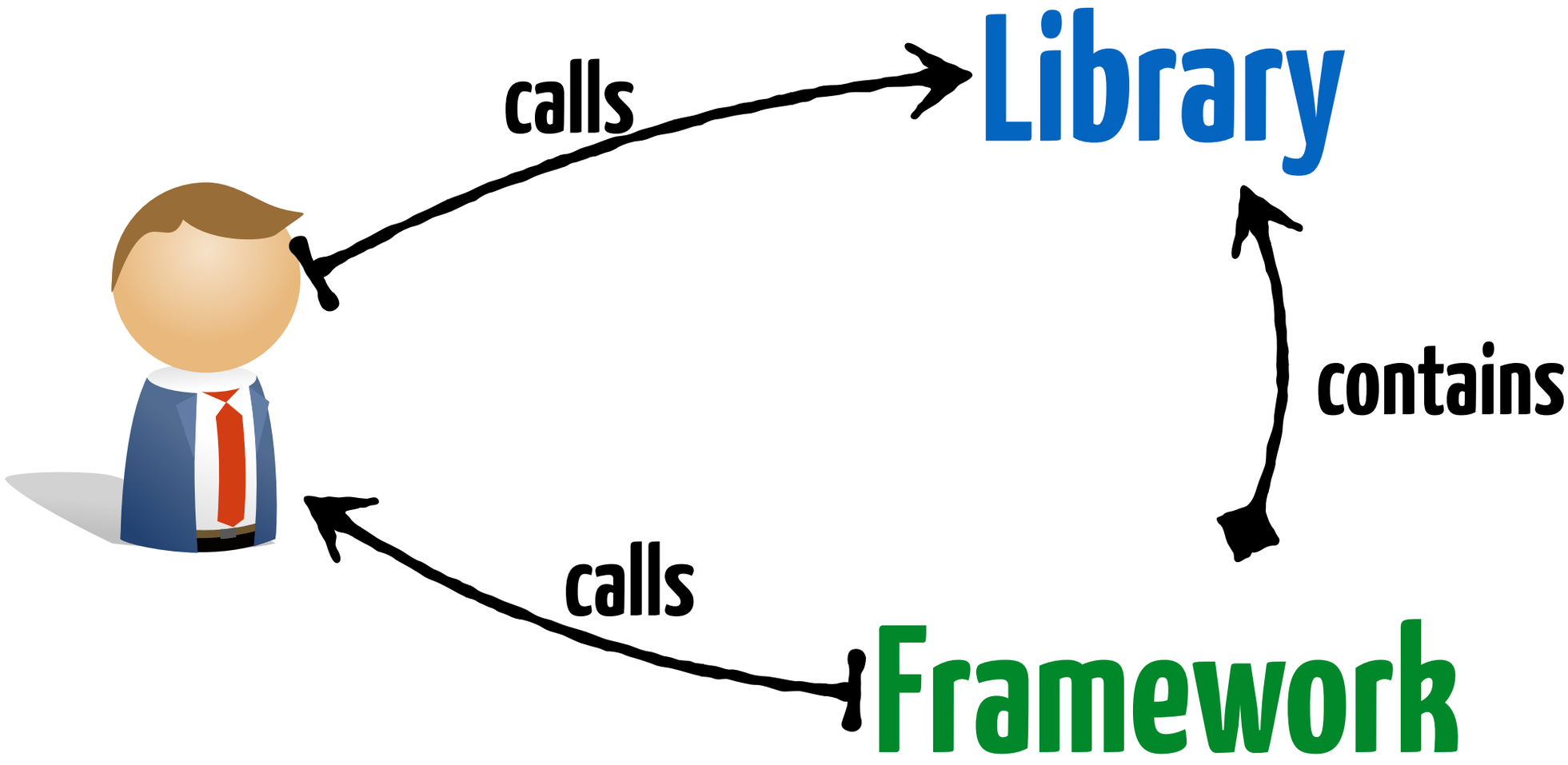


Inversion of Control

101



Library versus Framework?



Inversion of Control

Your **code** reuses a **library**

A **framework** reuses your **code**


HOTEL

Dependency Injection

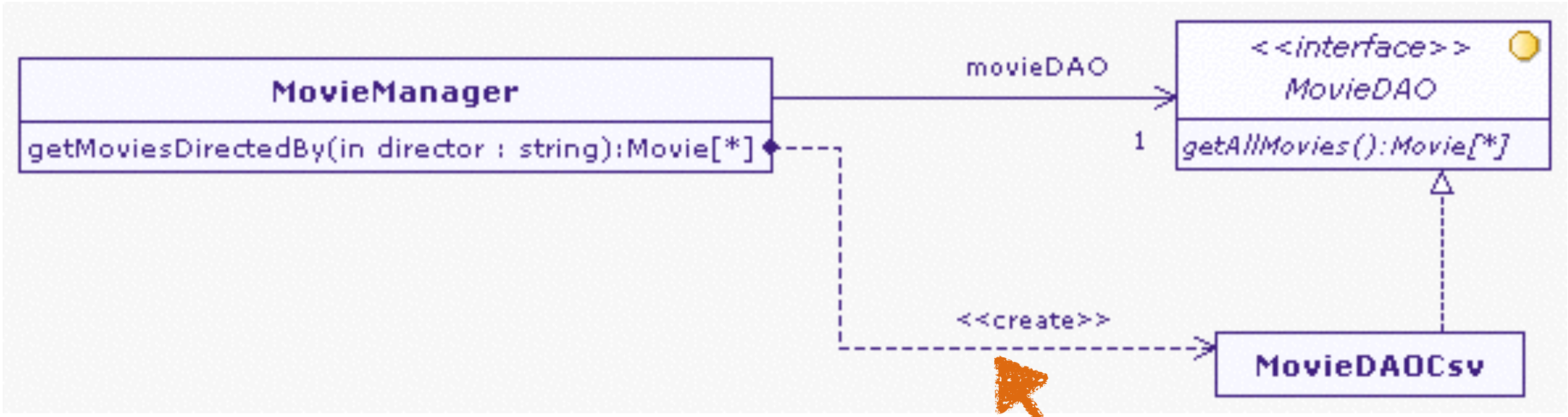
Illustration

Based in part on Martin Fowler's reference article on dependency injection


```java
public class MovieManager {

    private MovieDAOCsv movieDAOCsv;

    public MovieManager() {
        movieDAOCsv = new MovieDAOCsv("mymovies.txt");
    }

    public List<Movie> getMoviesDirectedBy(String director) {
        List<Movie> allMovies = movieDAOCsv.getAllMovies();
        // ...
```

strong coupling

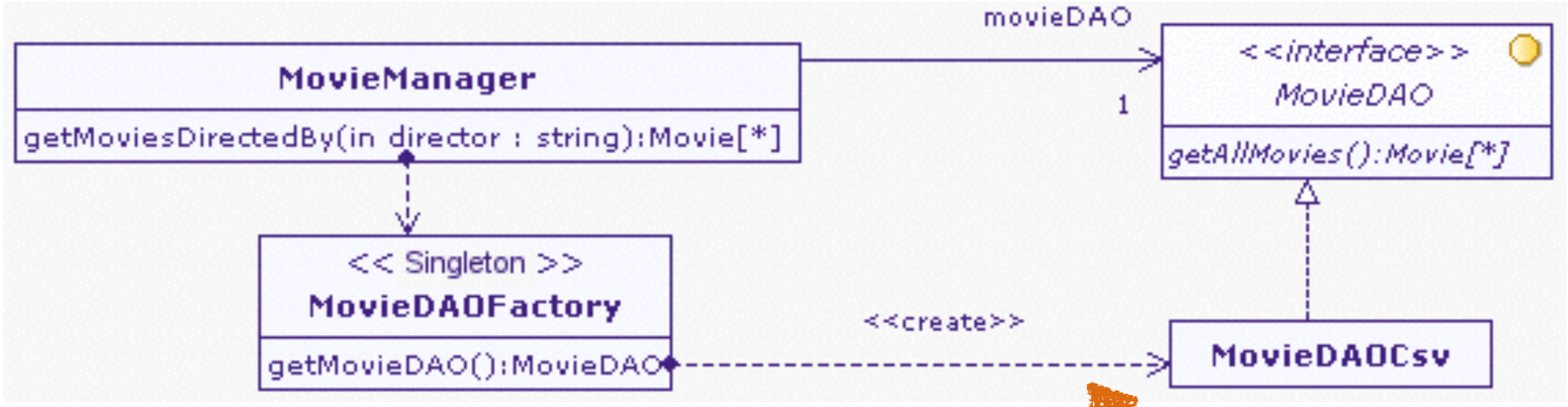
Using an interface




```java
public class MovieManager {

    private MovieDAO movieDAO;

    public MovieManager() {
        movieDAO = new MovieDAOCsv("mymovies.txt");
    }


    public List<Movie> getMoviesDirectedBy(String director) {
        List<Movie> allMovies = movieDAO.getAllMovies();
        // ...
```

weaker coupling

Using a factory




```
public class MovieManager {

    private MovieDAO movieDAO;

    public MovieManager() {
        movieDAO = MovieDAOFactory.getInstance().getMovieDAO();
    }

    public List<Movie> getMoviesDirectedBy(String director) {
        List<Movie> allMovies = movieDAO.getAllMovies();
```

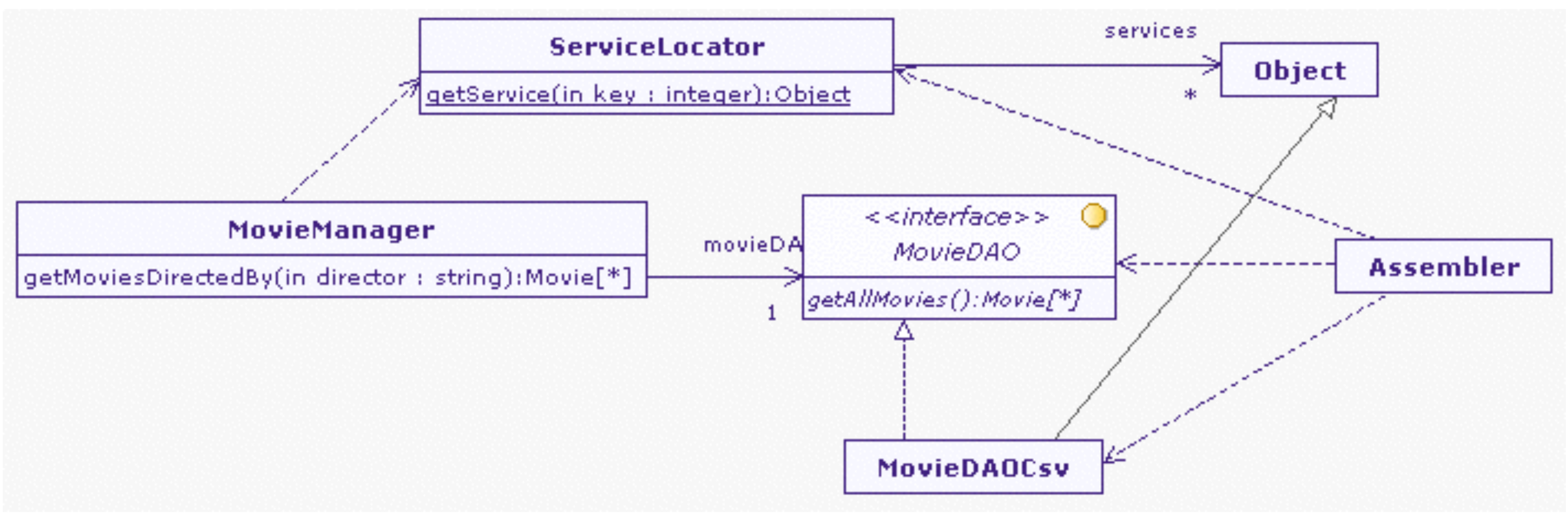
decoupling

But the factory is a class in our application...

Looking for dependencies

An external object is responsible for creating instances and add them to a list of services

These services are managed by a provider by associating them with key (string)



Looking for dependencies

```
public class MovieManager {

    private MovieDAO movieDAO;

    public MovieManager() {
        movieDAO = (MovieDAO) ServiceLocator.getService("movieDao");
    }
```

Only the interface declaration

Lookup

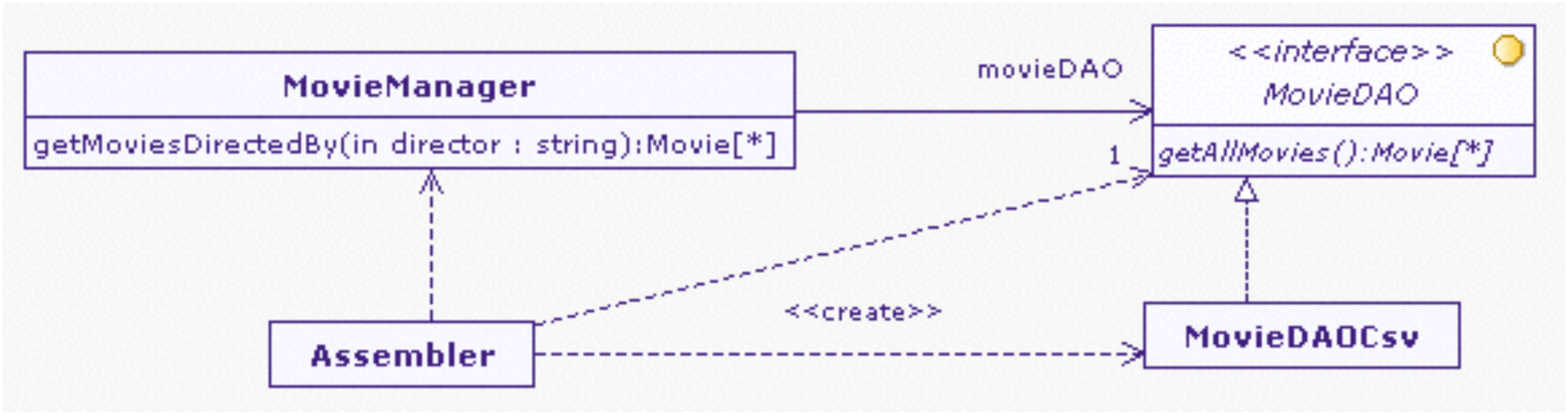
ServiceLocators is known as an Architectural Pattern

Dependency Injection

An external object is responsible for creating the application

It creates instances

It injects them in the classes it uses



Dependency Injection

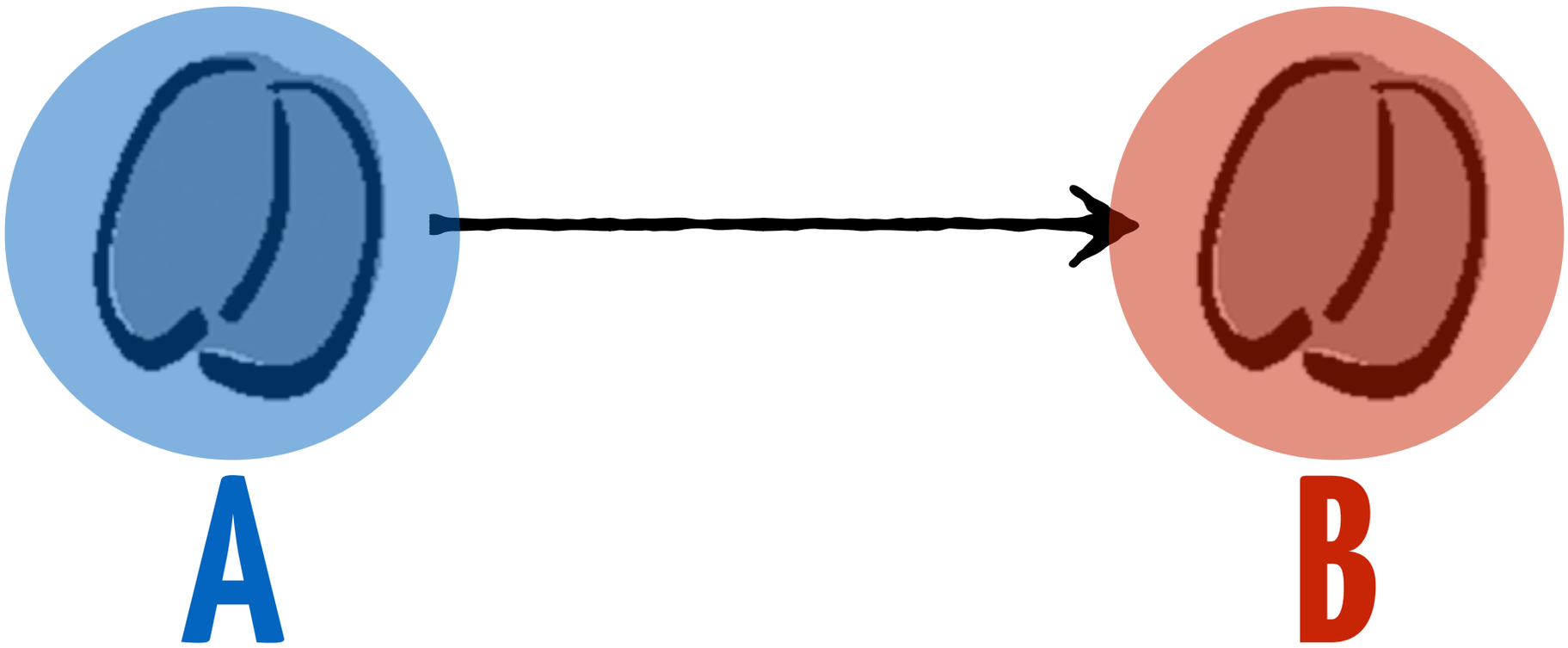
```
public class MovieManager {

    private MovieDAO movieDAO;

    public MovieManager() {
    }

    public void setMovieDAO(MovieDAO movieDAO) {
        this.movieDAO = movieDAO;
    }
```

Only the interface declaration

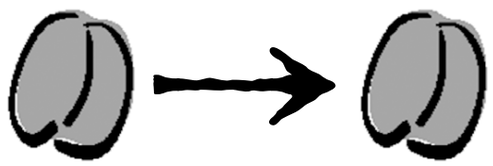
creates movieDAO

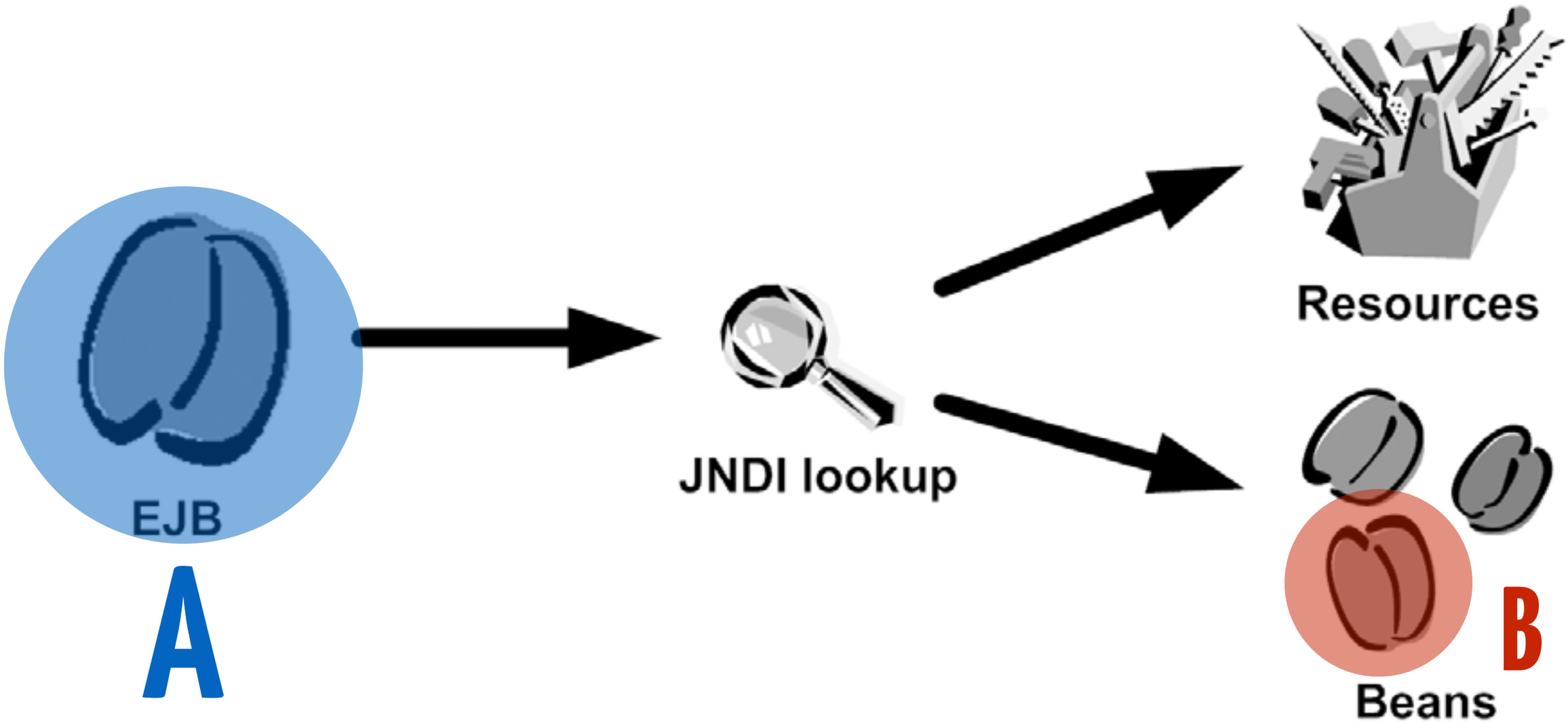
calls setMovieDAO

Problem: Bean Dependencies



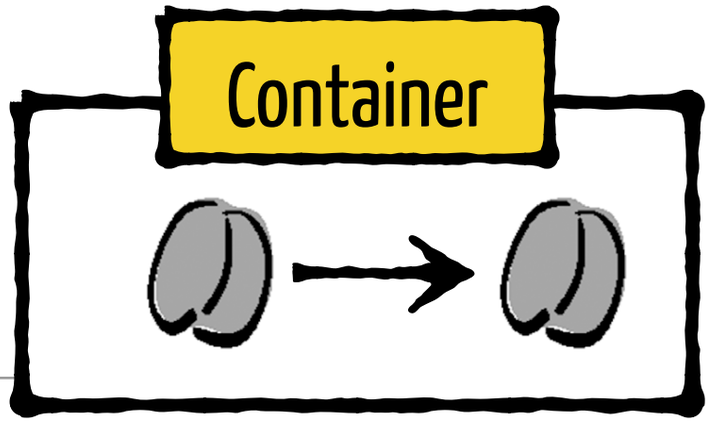
The good-old method: Lookup

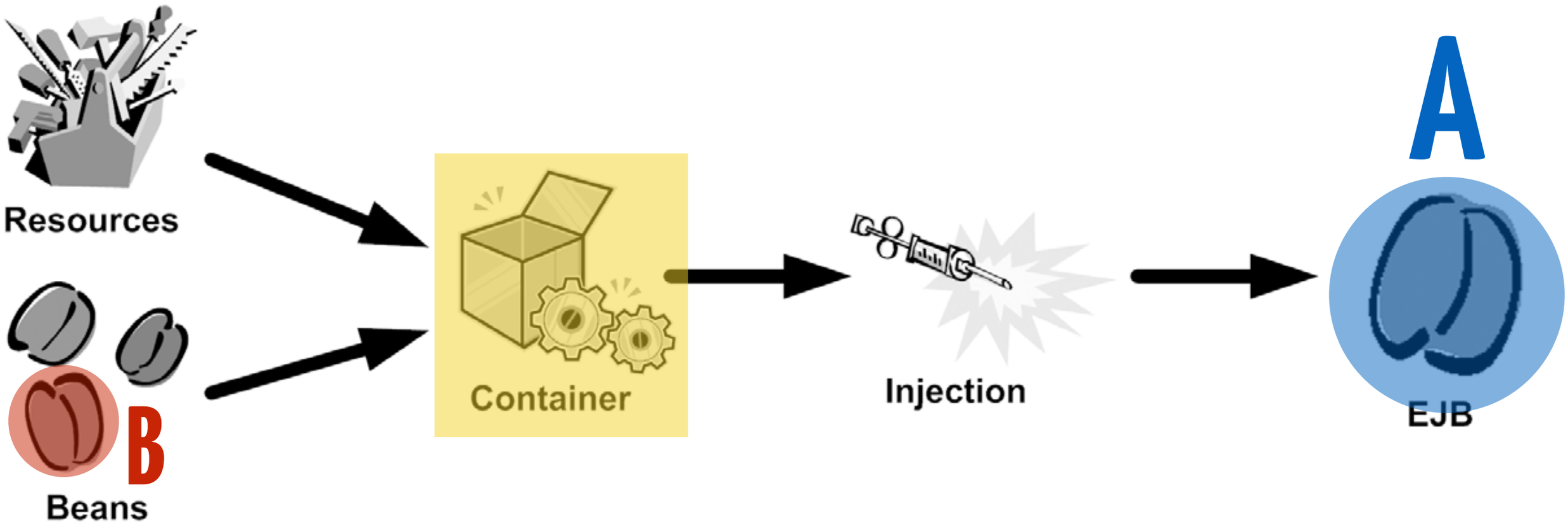





```
Object ejbHome = new InitialContext().lookup("java:comp/env/PlaceBid");
PlaceBidHome placeBidHome = (PlaceBidHome)
    PortableRemoteObject.narrow(ejbHome, PlaceBidHome.class);
PlaceBid placeBid = placeBidHome.create();
```

Steroids: Dependency Injection



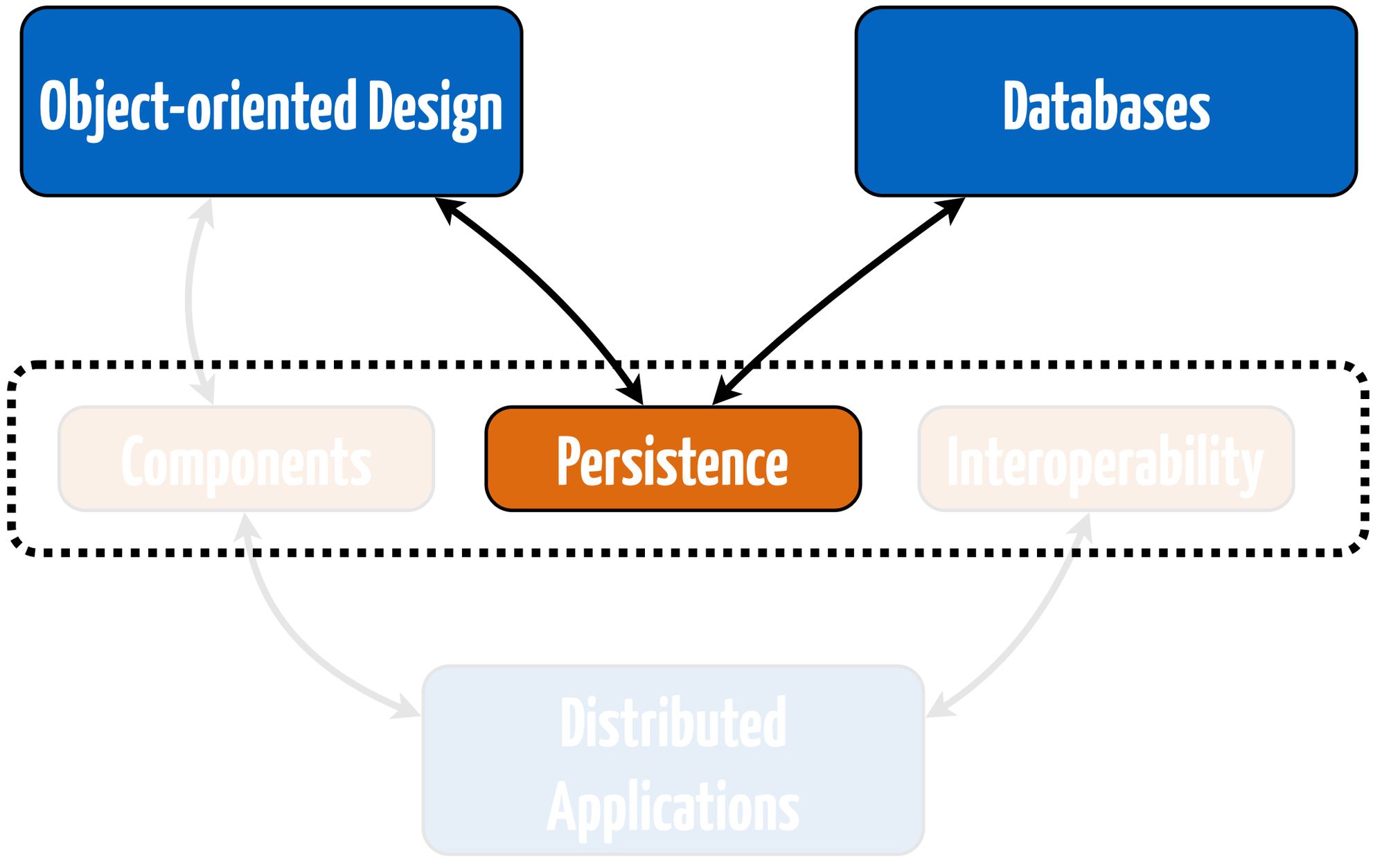



```
public class PlaceOrderTestClient {
    @EJB
    private static PlaceOrder placeOrder;
```

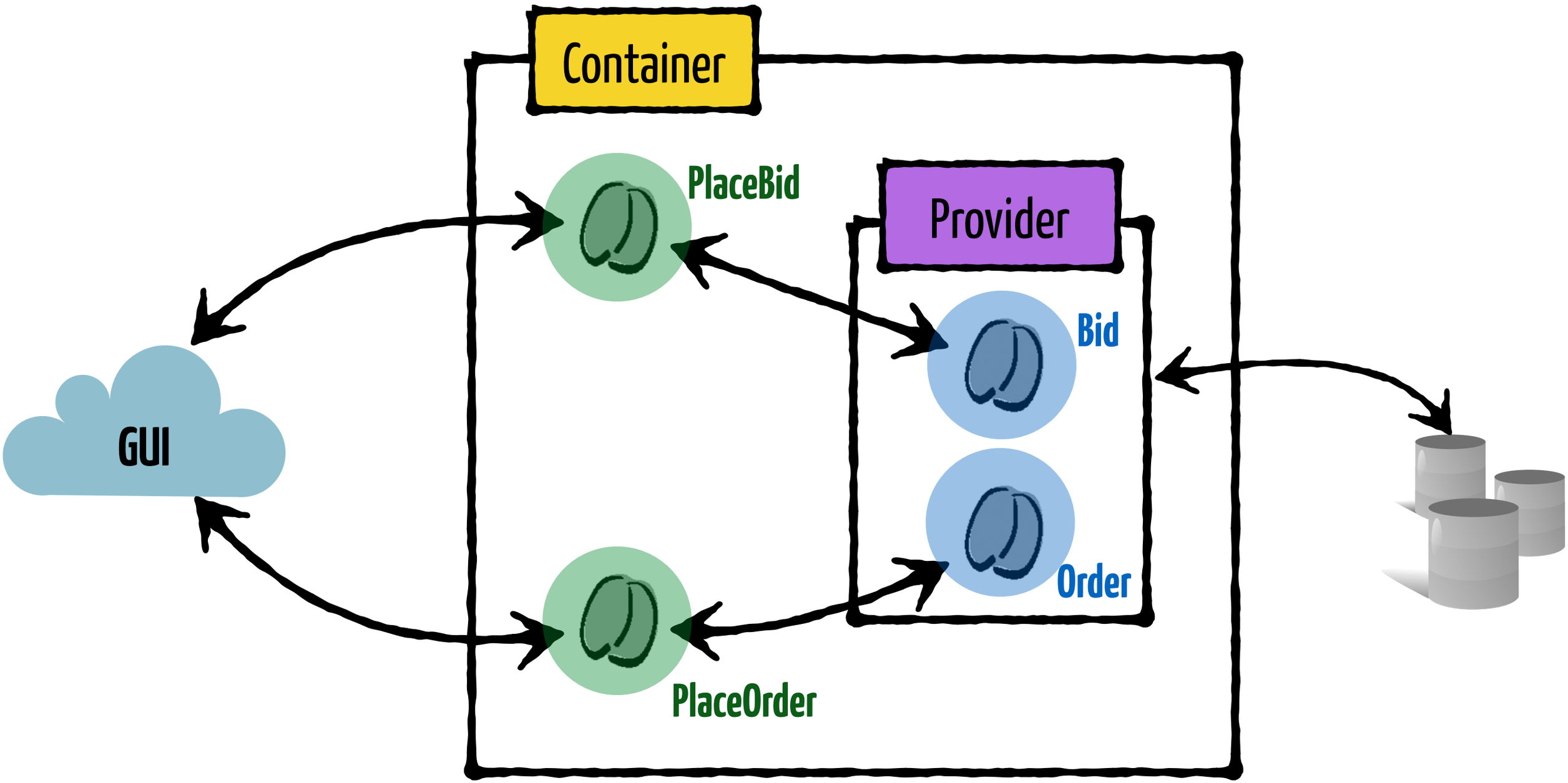
Injects an instance of EJB

Object-Relational Mapping 101

Applications Server: Dependencies



Session & Entity

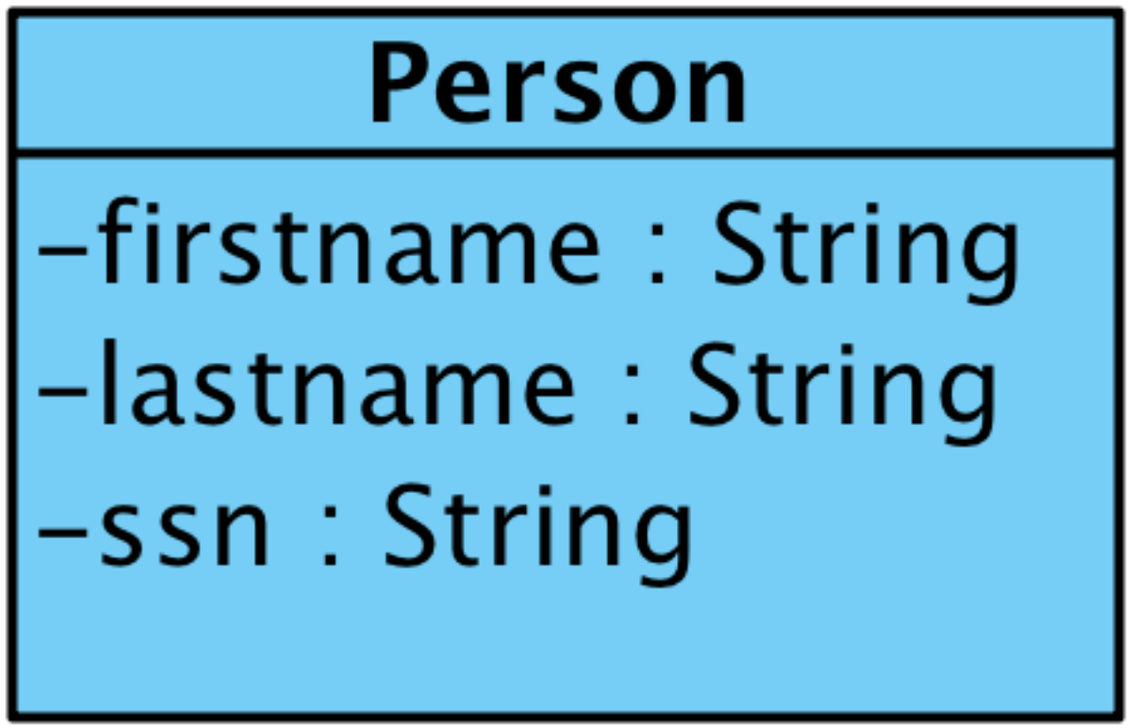


Impedance Mismatch

Object-Oriented	Relational
Classes	Relation (table)
Object	Tuple (row)
Attribute	Attribute (column)
Identity	Primary Key
Reference	Foreign Key
Inheritance	N/A
Methods	~ Stored Procedure

[EJB3 in Action]

Example of **Domain Model**



first_name	last_name	ssn
Sébastien	MOSSER	16118325358
...		

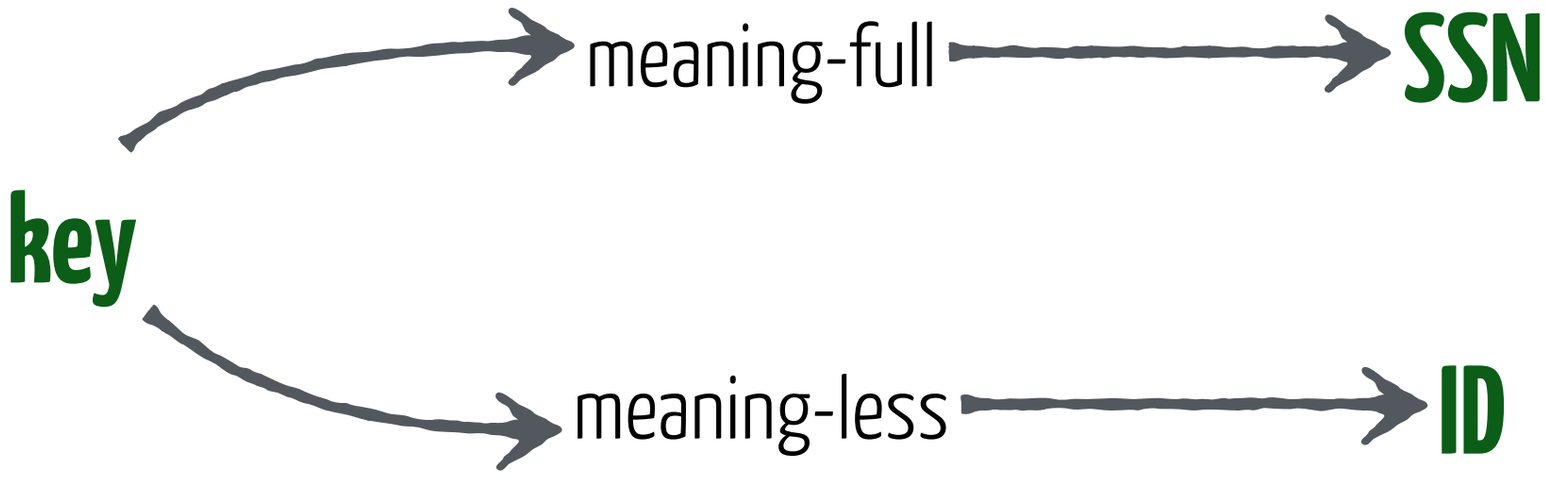
[PoEAA]

Problem: Domain Object Identity

```
Person p1 = new Person("X","Y","DDMMYYXXXXX")
Person p2 = new Person("X","Y","DDMMYYXXXXX")
```

How to support persons **uniqueness**?

Candidates for **Key** role



- Necessary condition:
  - **Key must be unique**
- Nice-to-have condition:
  - **Key should be immutable**
- **Compound** versus **Simple**

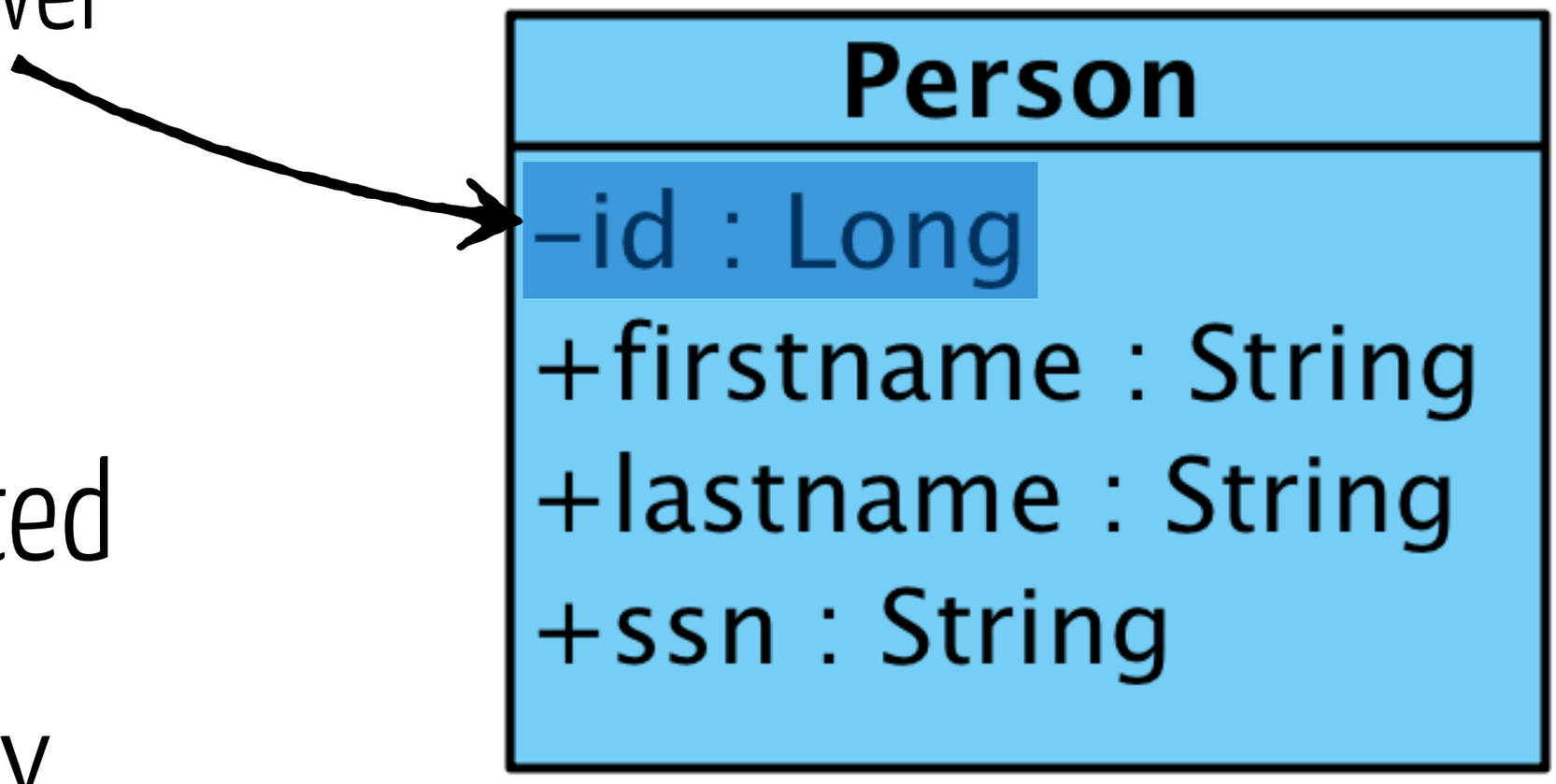
[PoEAA]

Pattern: **Identity Field**

Not visible at
the business level

auto-generated

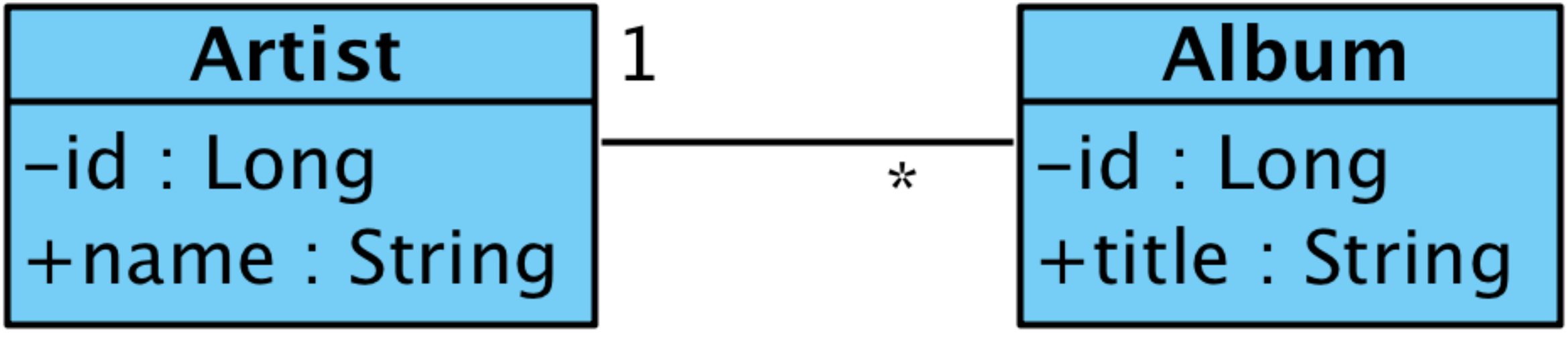
GUID DIY



[PoEAA]

When are 2 customers equals?

Problem: Representing associations



artists

id	name
1	Linkin Park
	…

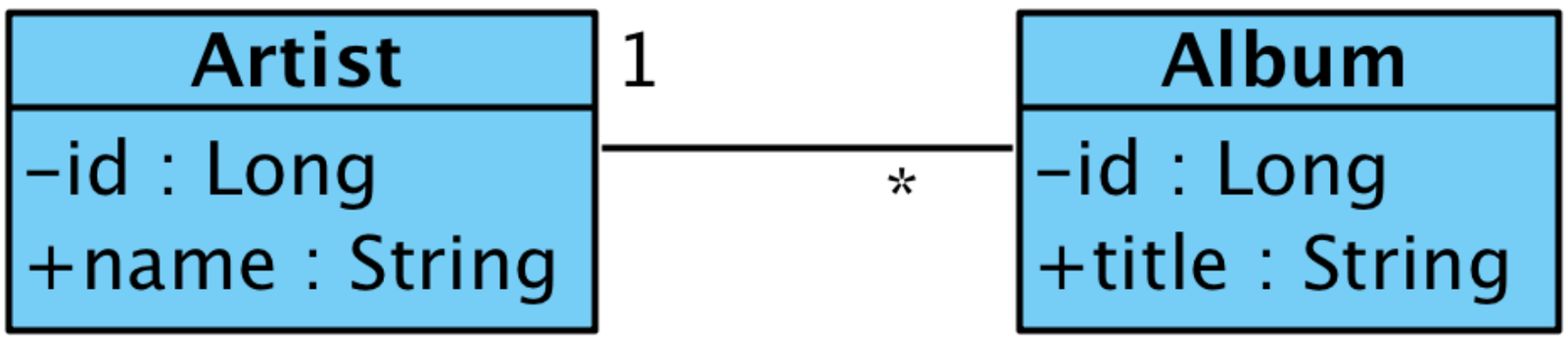
albums

id	title
1	A Thousand Suns
2	Minutes to Midnight
	…

[PoEAA]

How to bind customers to orders?

Solution #1: Association Table [M-N]



artists

id	name
1	Linkin Park
...	

albums

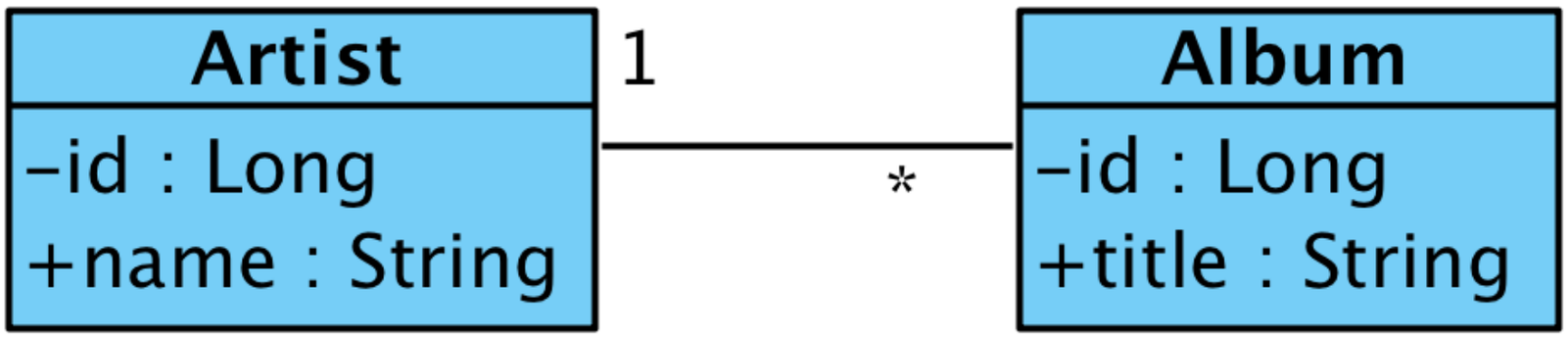
id	title
1	A Thousand Suns
2	Minutes to Midnight
...	

artists_to_albums

artist_id	album_id
1	1
1	2
...	

[PoEAA]

Solution #2: **Foreign Key** [1-N]



artists

id	name
1	Linkin Park
...	

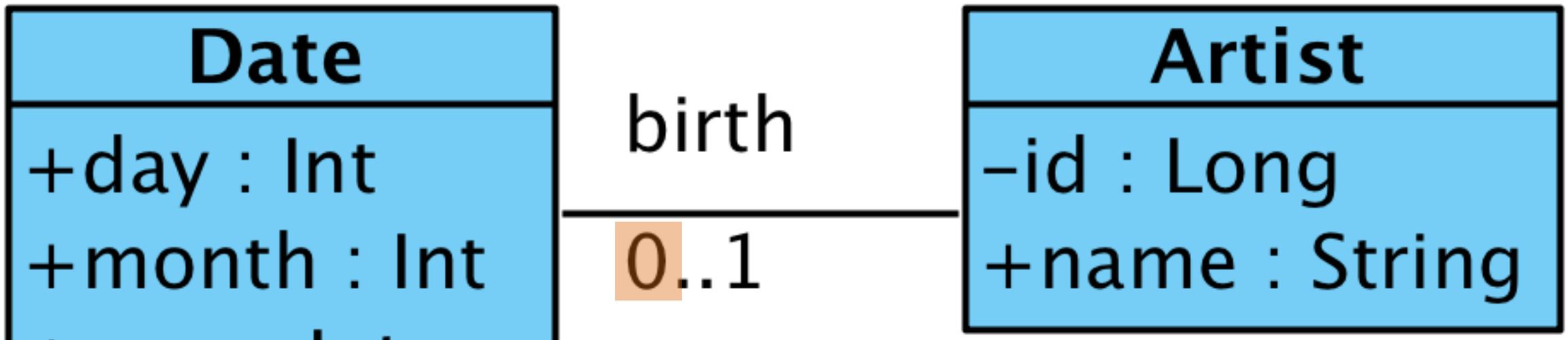
albums

id	title	artist_id
1	A Thousand Suns	1
2	Minutes to Midnight	1
...		

or **[1-N]** ≡ **[M-N]** when N =1

[PoEAA]

Solution #3: **Relation Merge** [1-1]



artists

id	name	birth_day	birth_month	birth_year
1	Linkin Park	-1	-1	-1
...				

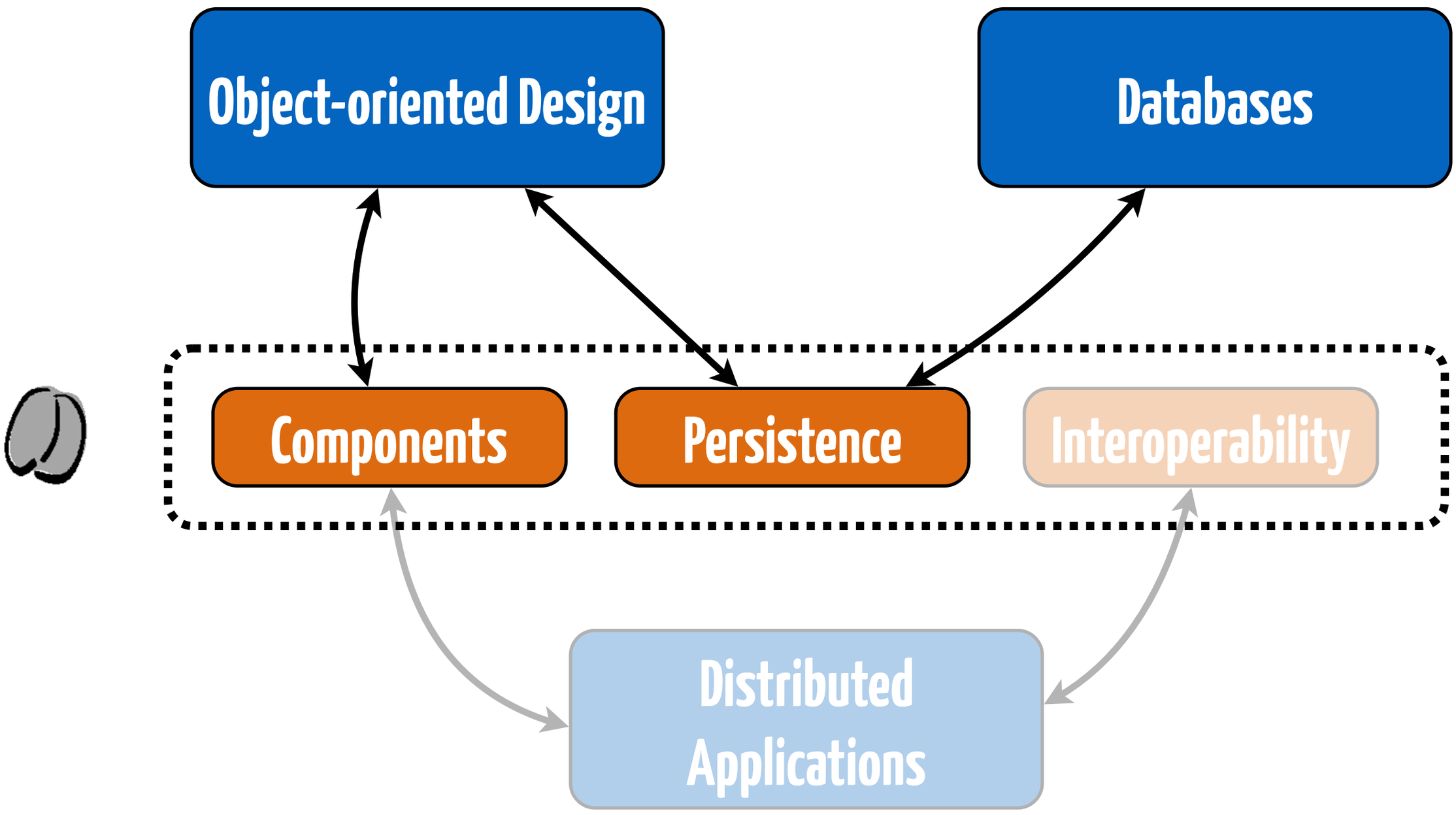
or **[1-1]** ≡ **[1-N]** when N =1

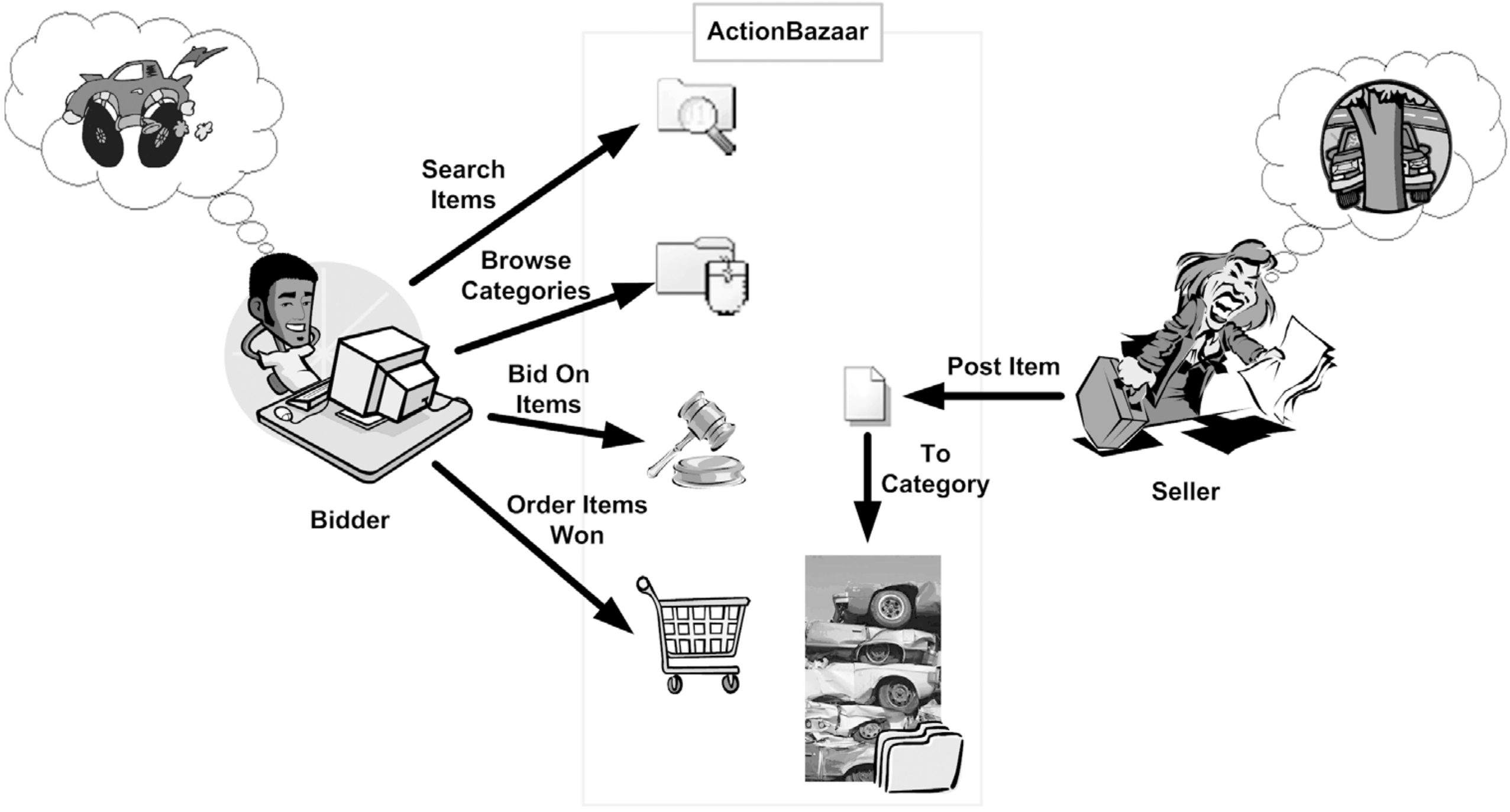
[PoEAA] or **[1-1]** ≡ **[M-N]** when M = 1 and N =1

Make your beans **persistent**

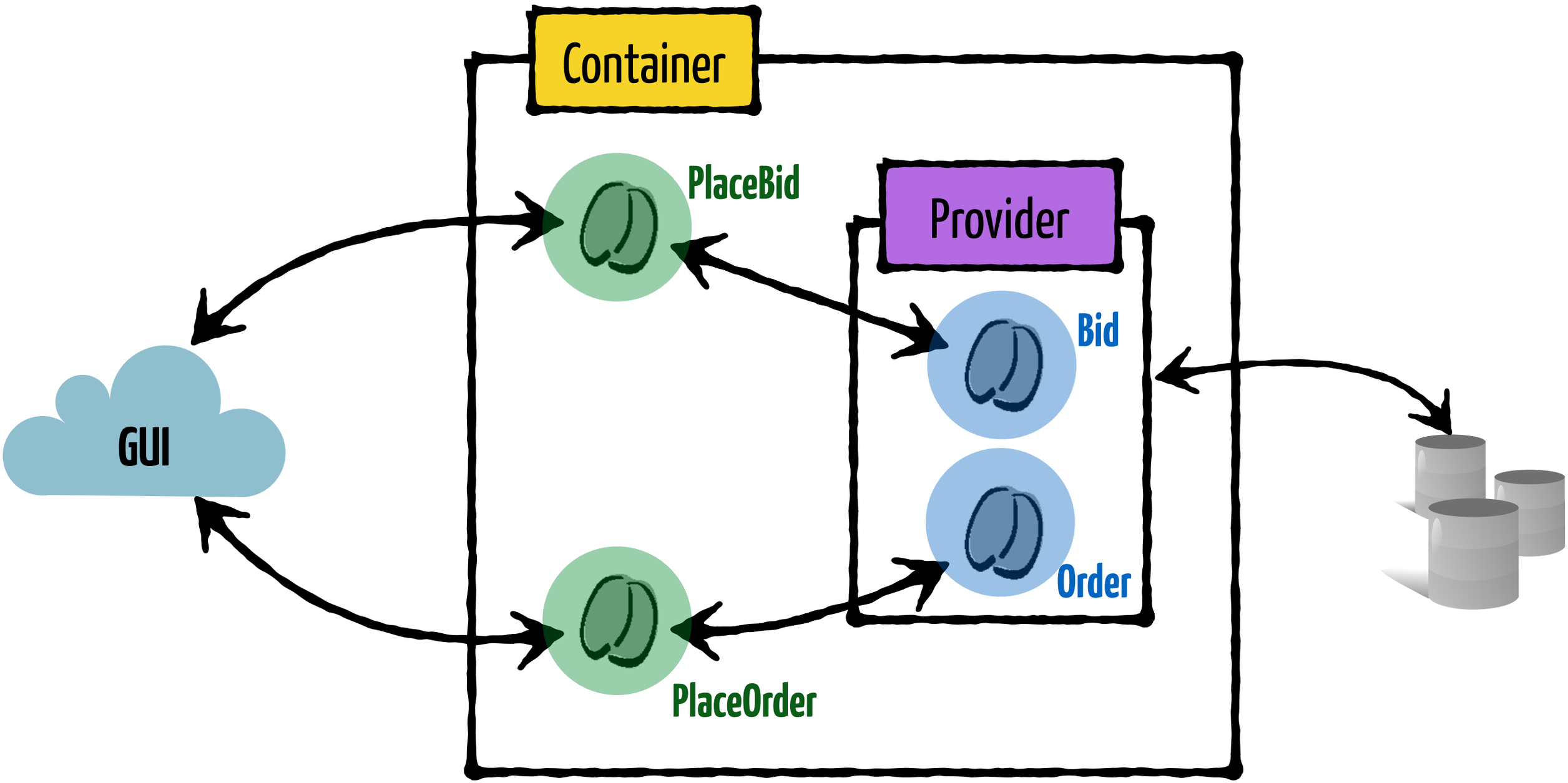
101

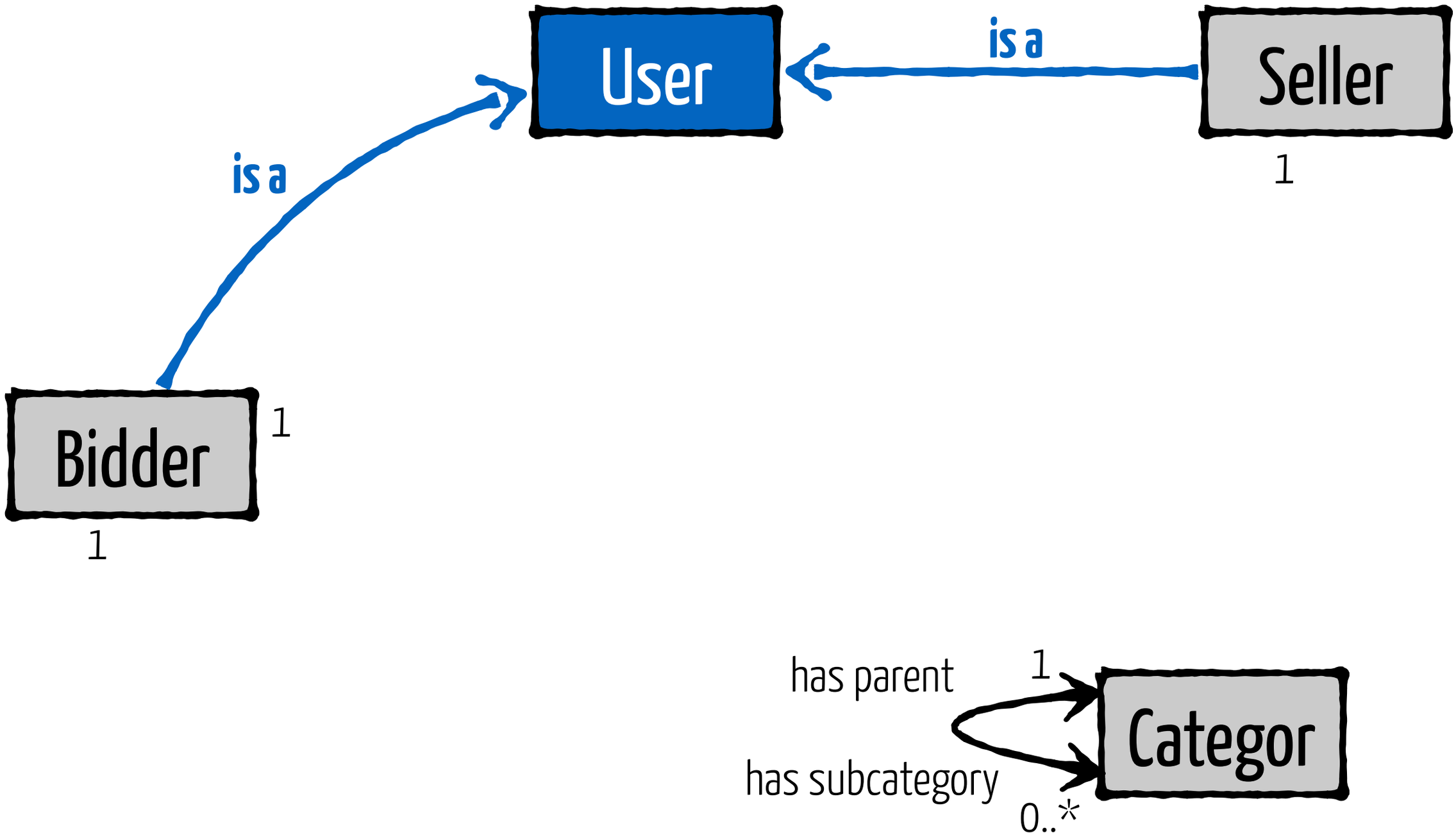
Applications Server: Dependencies




ActionBazaar
Search Items
Browse Categories
Bid On Items
Order Items Won
Bidder
Post Item
To Category
Seller

Session & Entity




User
is a
Seller
1
is a
Bidder
1
1

has parent
1
Categor
has subcategory
0..*

[EiA]


```
@Entity
public class Category {

   public Category() { … }

   protected String name;

   public String getName() {
      return this.name;
   }

   public void setName(String n) {
      this.name = n.toUpperCase();
   }
}
```

property-based access

(JPA)

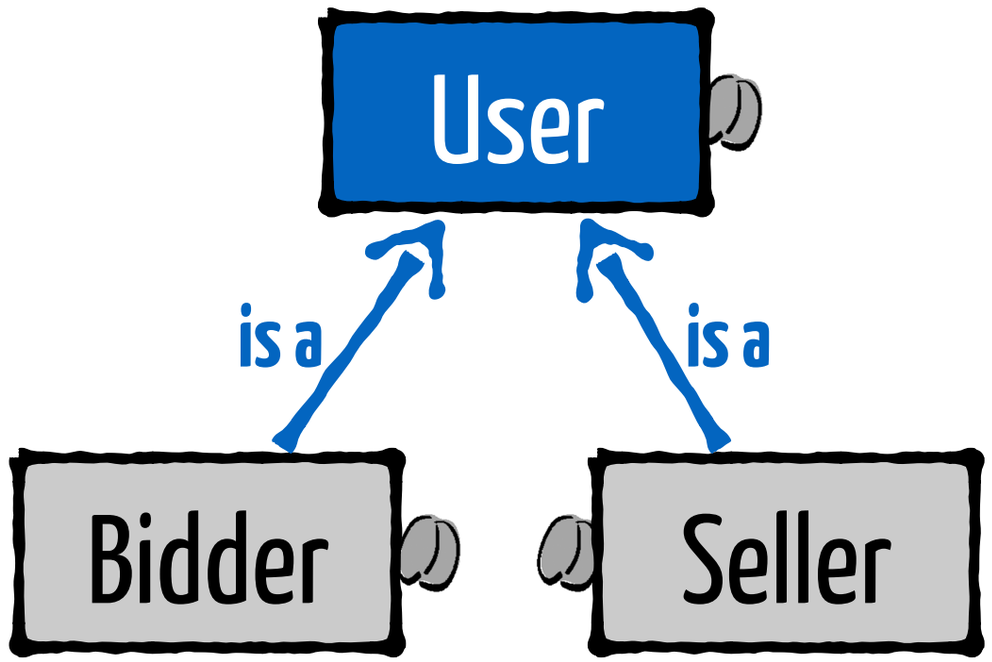
Category

```
@Entity
public class Category {

   public Category() { … }

   public String name;
}
```

field-based access

Fields are simple but forbid encapsulation

```java
@Entity
public abstract class User {
   // …
}
```




```java
@Entity
public class Bidder extends User {
   // …
}

@Entity
public class Seller extends User {
   // …
}
```

[EiA]

Simple Primary Key: @Id

```
@Entity
public class Category {
   // …

   @Id
   @GeneratedValue(strategy = GenerationType.AUTO)
   public Long id;

}
```

Identifiers must define an "equals" method

Composite Key: @IdClass

```
public class CategoryPK extends Serializable {
   String name;
   Date createDate;
}
```

```
@Entity
@IdClass(CategoryPK.class)
public class Category {

   @Id
   protected String name;

   @Id
   protected Date createDate;
}
```

Identifiers must define an "equals" method

```
public class CategoryPK extends Serializable {

   public boolean equals(Object other) {

      if (other instanceOf CategoryPK) {
         final CategoryPK that = (CategoryPK) other;
         return that.name.equals(name) &&
                that.createDate.equals(createDate);
      return false;

   }

   public int hashCode() {
      return super.hashCode();
   }
}
```

[EiA]

Serializable requirements

Equality Relation definition

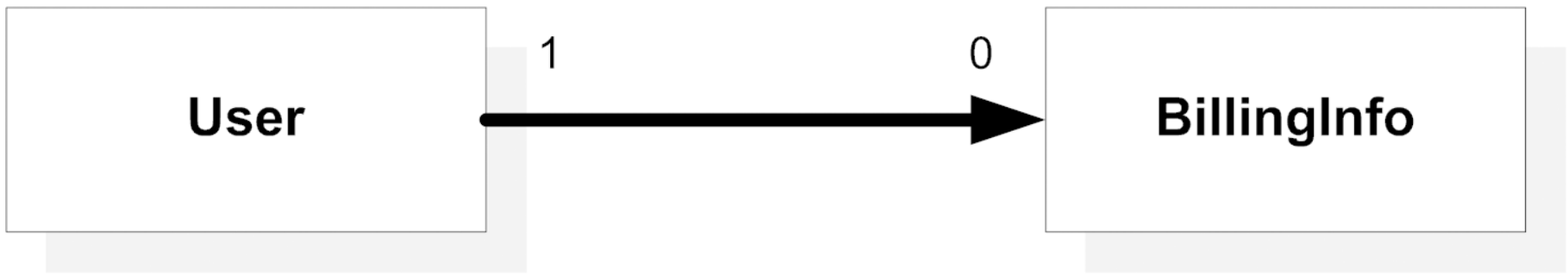
- equals is reflexive
- equals is symmetric
- equals is transitive
- equals is consistent
- equals uses null as absorbing element

It's complicated!

Relationships

Type of Relationship	Annotation
1-1	`@OneToOne`
1-n	`@OneToMany`
n-1	`@ManyToOne`
n-m	`@ManyToMany`

[EiA]



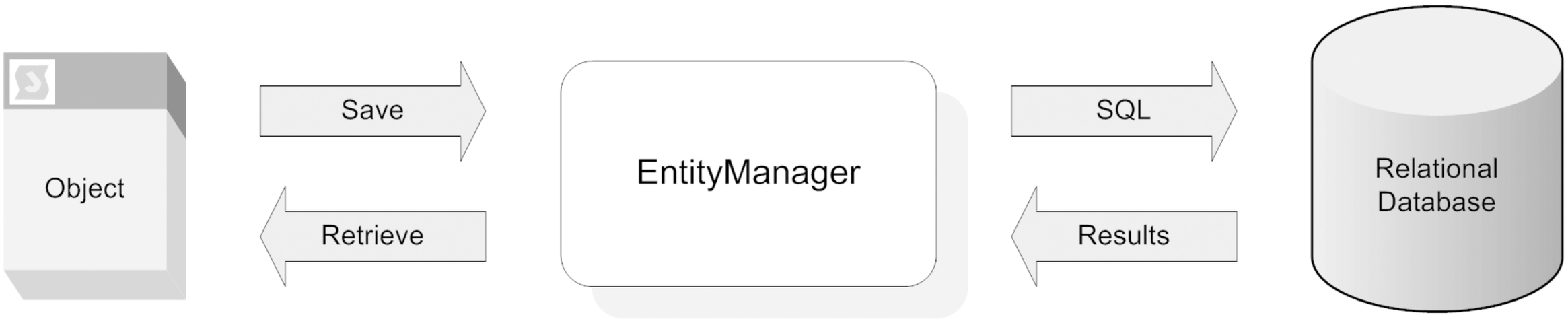
Unidirectional 1-1 mapping

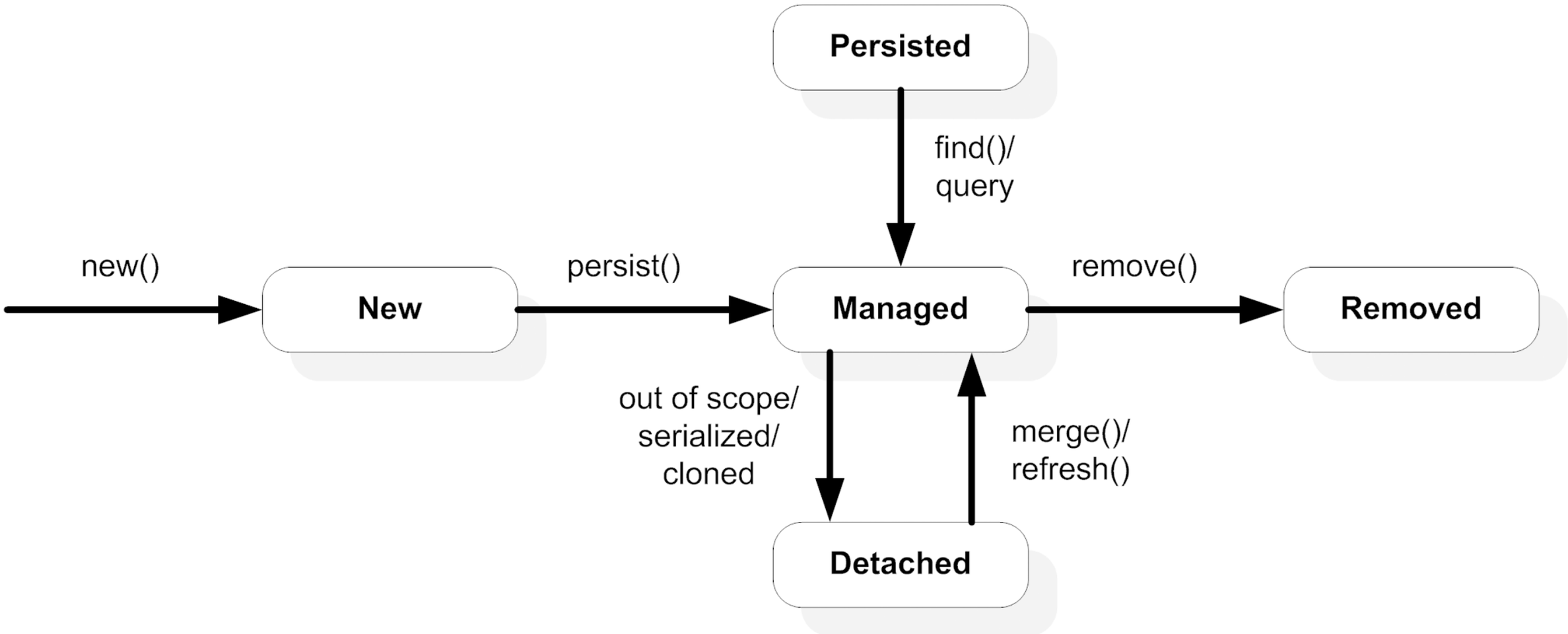
```
@Entity
public class User {
   @Id
   protected String userId;
   protected String email;

   @OneToOne
   protected BillingInfo billingInfo;
}
```

[EiA]

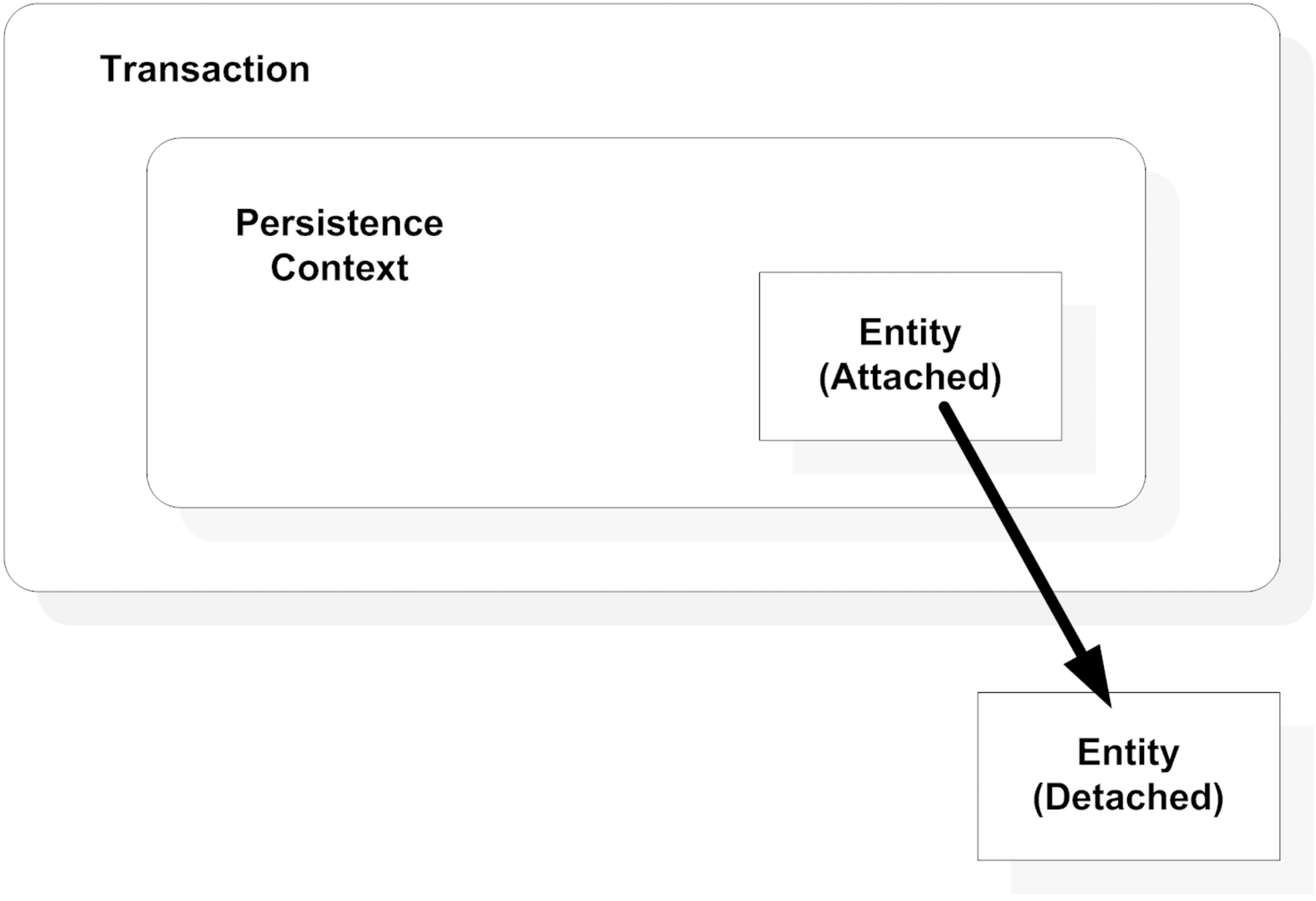
For property-based beans, annotate the getter.

Entity Manager




Persisted
find()/
query
new()
persist()
remove()
New
Managed
Removed
out of scope/
serialized/
cloned
merge()/
refresh()
Detached

[EiA]

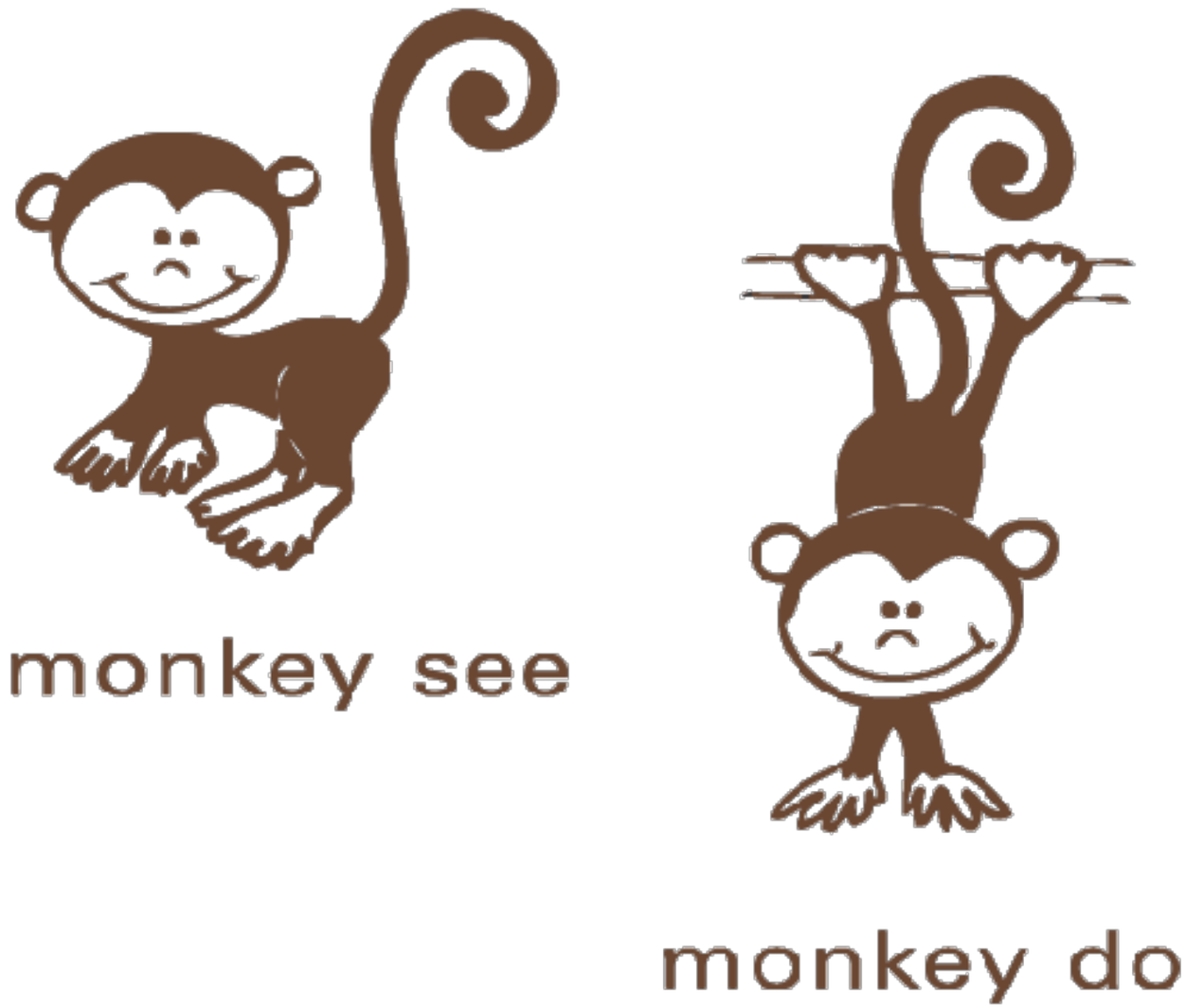

Transaction
Persistence Context
Entity (Attached)
Entity (Detached)

Persistence context is **Injected**

```
@PersistenceContext(unitName="admin")
EntityManager manager

@Resource
private UserTransaction transaction;

public void createAndStore() {

  AnEntityBean b = new AnEntityBean("Parameters");
  transaction.begin();
  try {
    manager.persist(b);
  } finally {
    transaction.commit();
  }

}
```

See [EiA], chapter 9

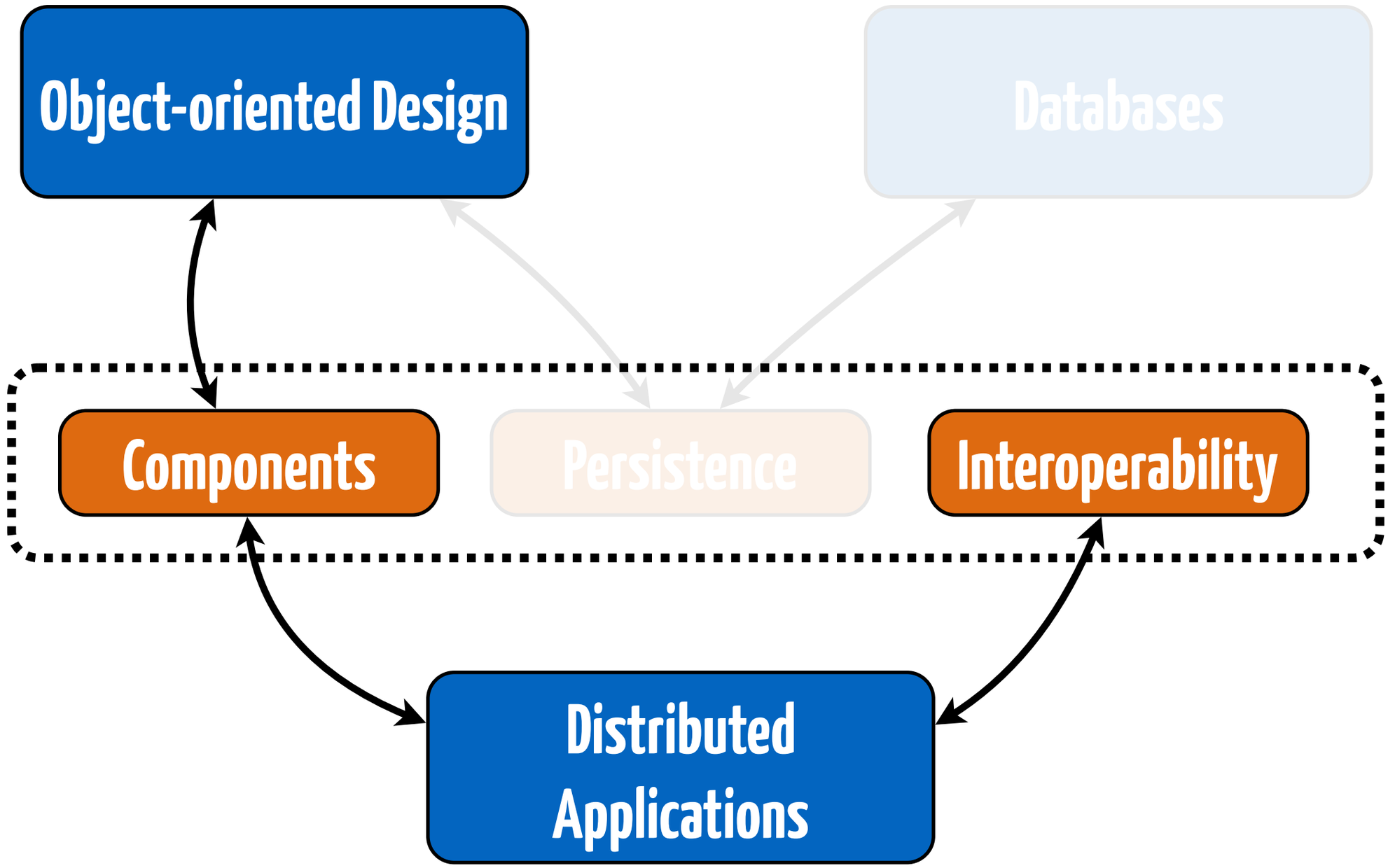
EJB are **standard**: Learn by **Example**!



http://infiltrated.net/index.php?option=com_content&view=article&id=43&Itemid=49

Session Beans 101

Applications Server: Dependencies



3-tiers architecture

Presentation

Handle users

Domain

Business functionality

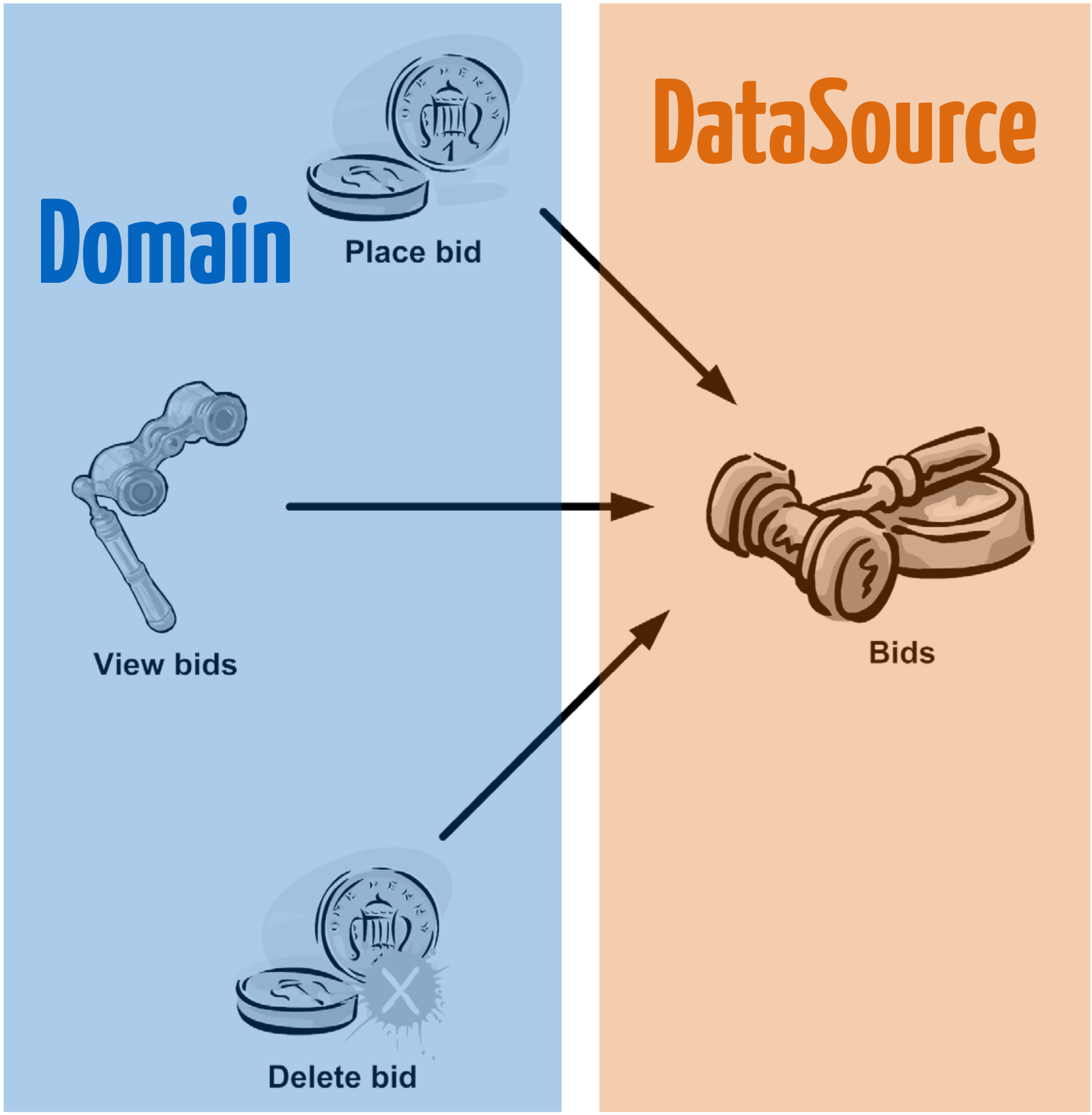
Data Source

Handle data storage

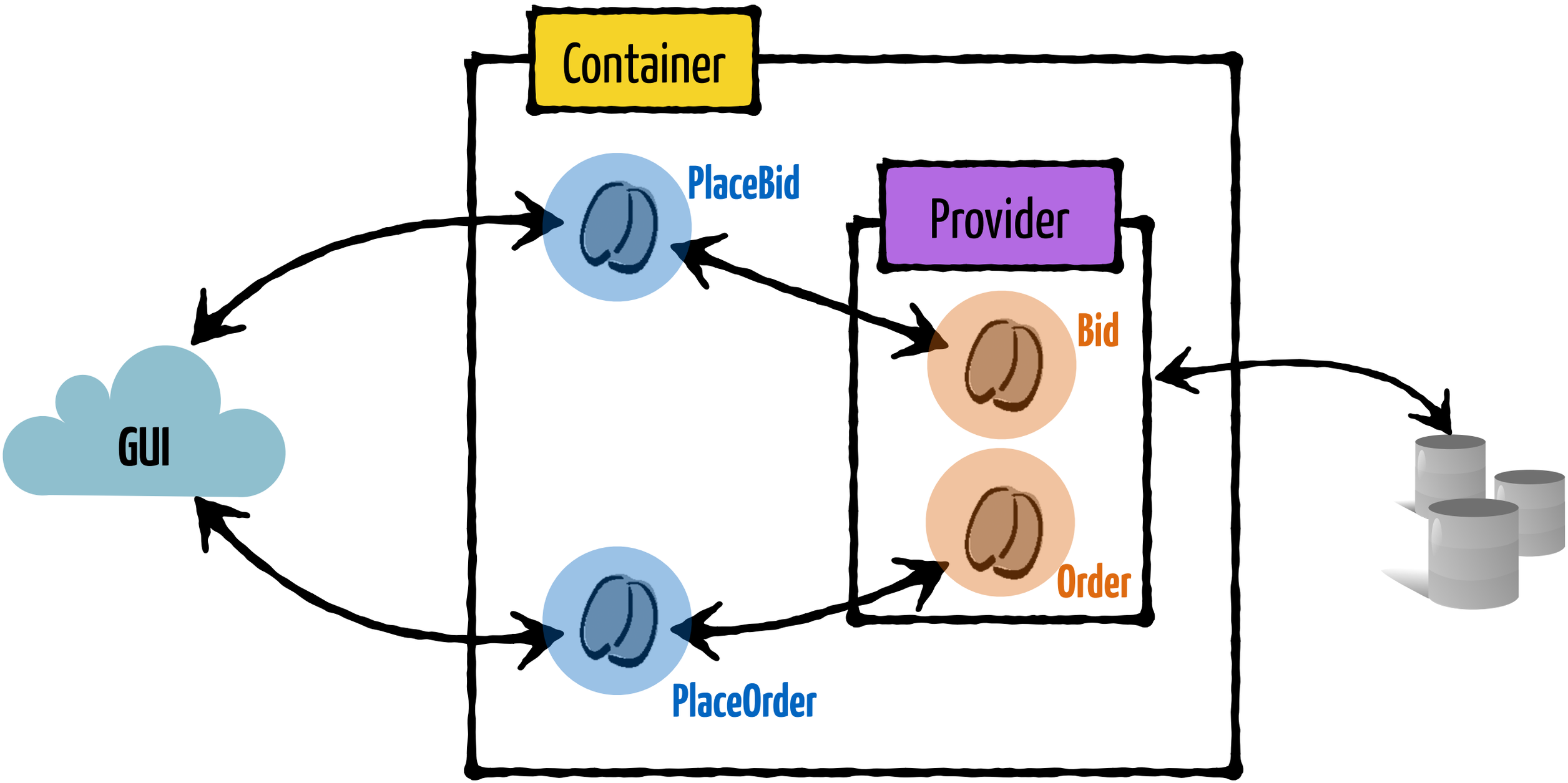
Rule of Thumb

Domain Bean interfaces as **Verbs**

DataSource Beans as **Nouns**


Domain
DataSource
Place bid
View bids
Delete bid
Bids

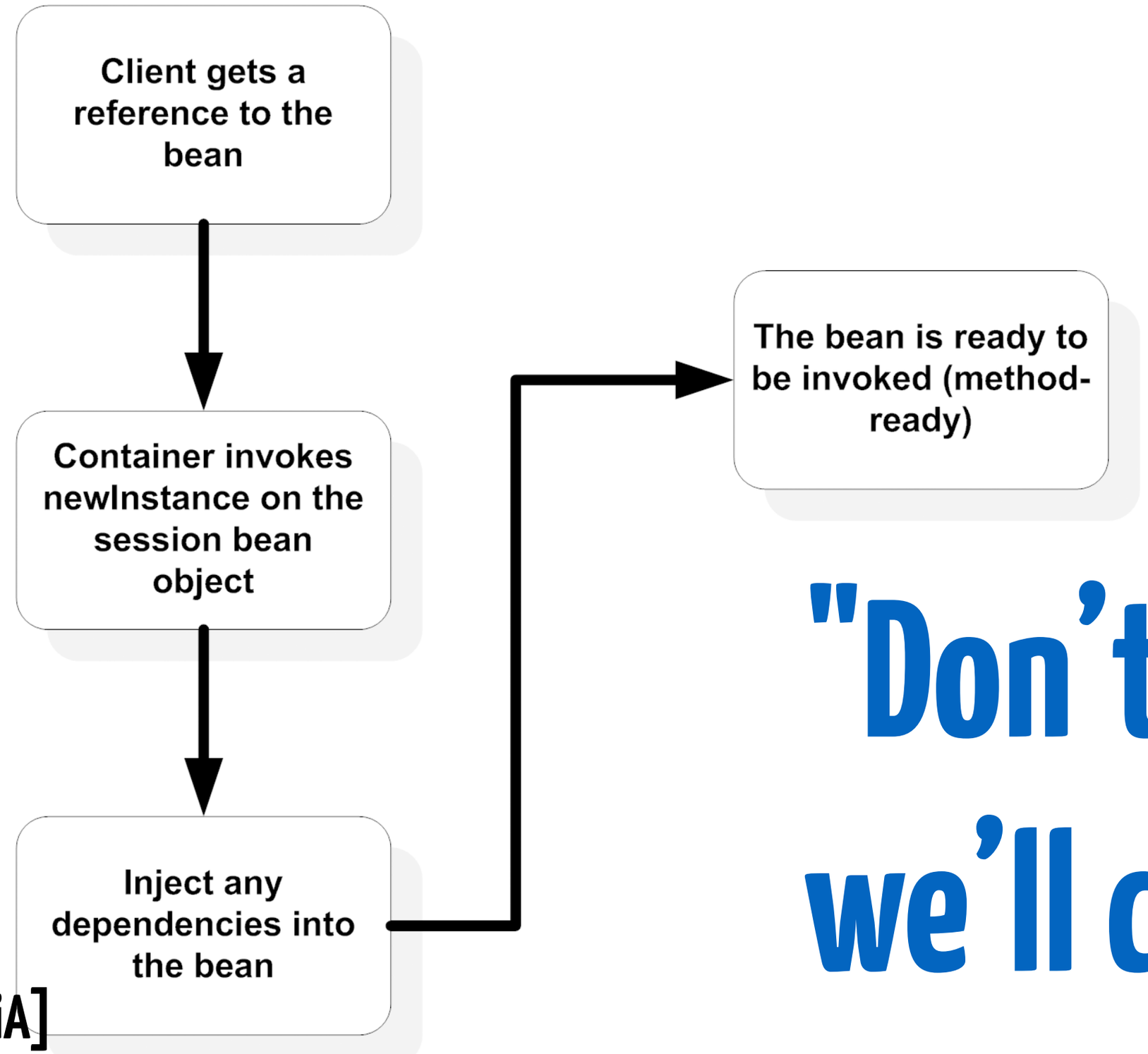
Client **never calls** a datasource directly



[EiA]

You'll **never** instantiate a domain bean.

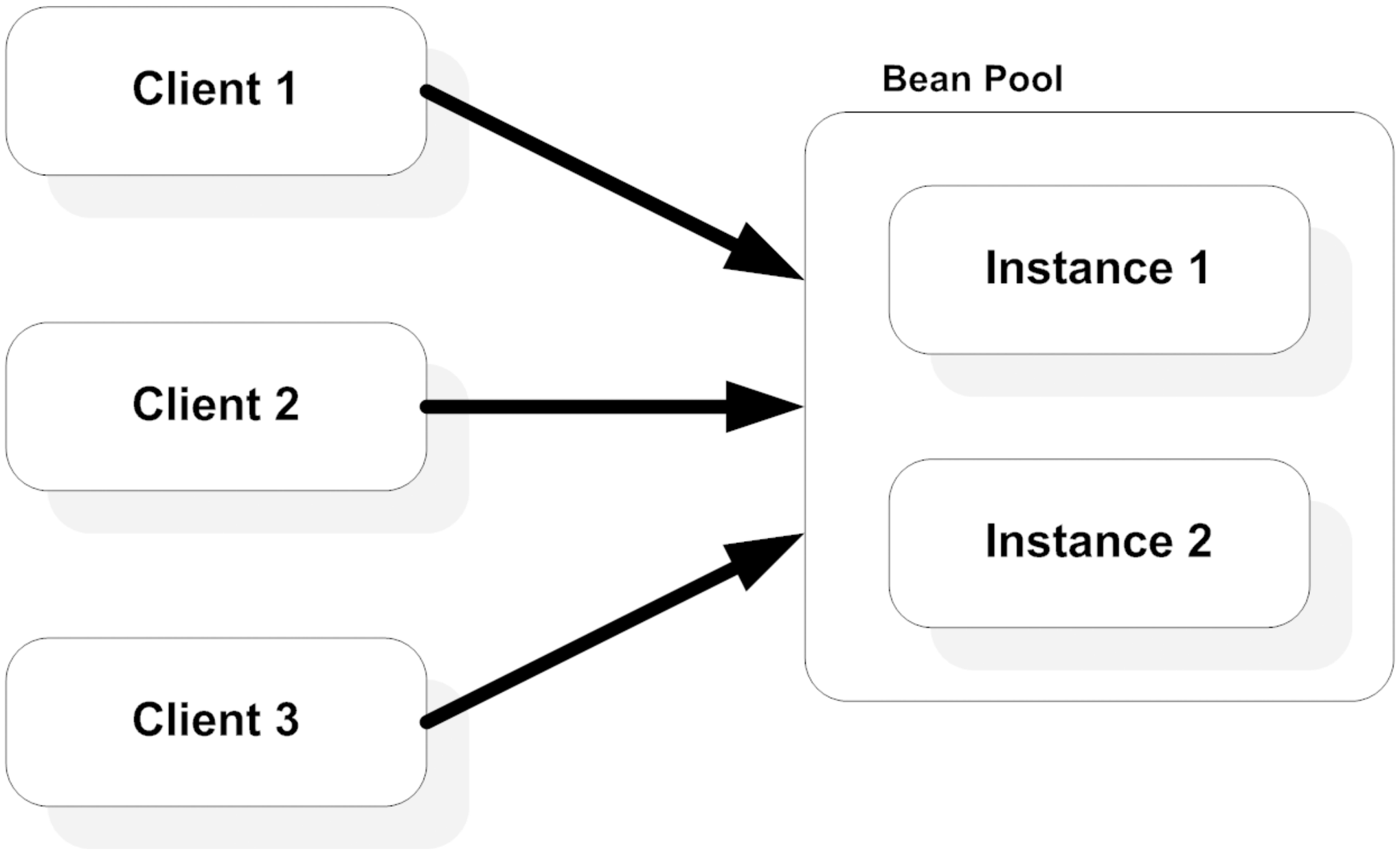
EJB's Lifecycle: **Inversion of Control**



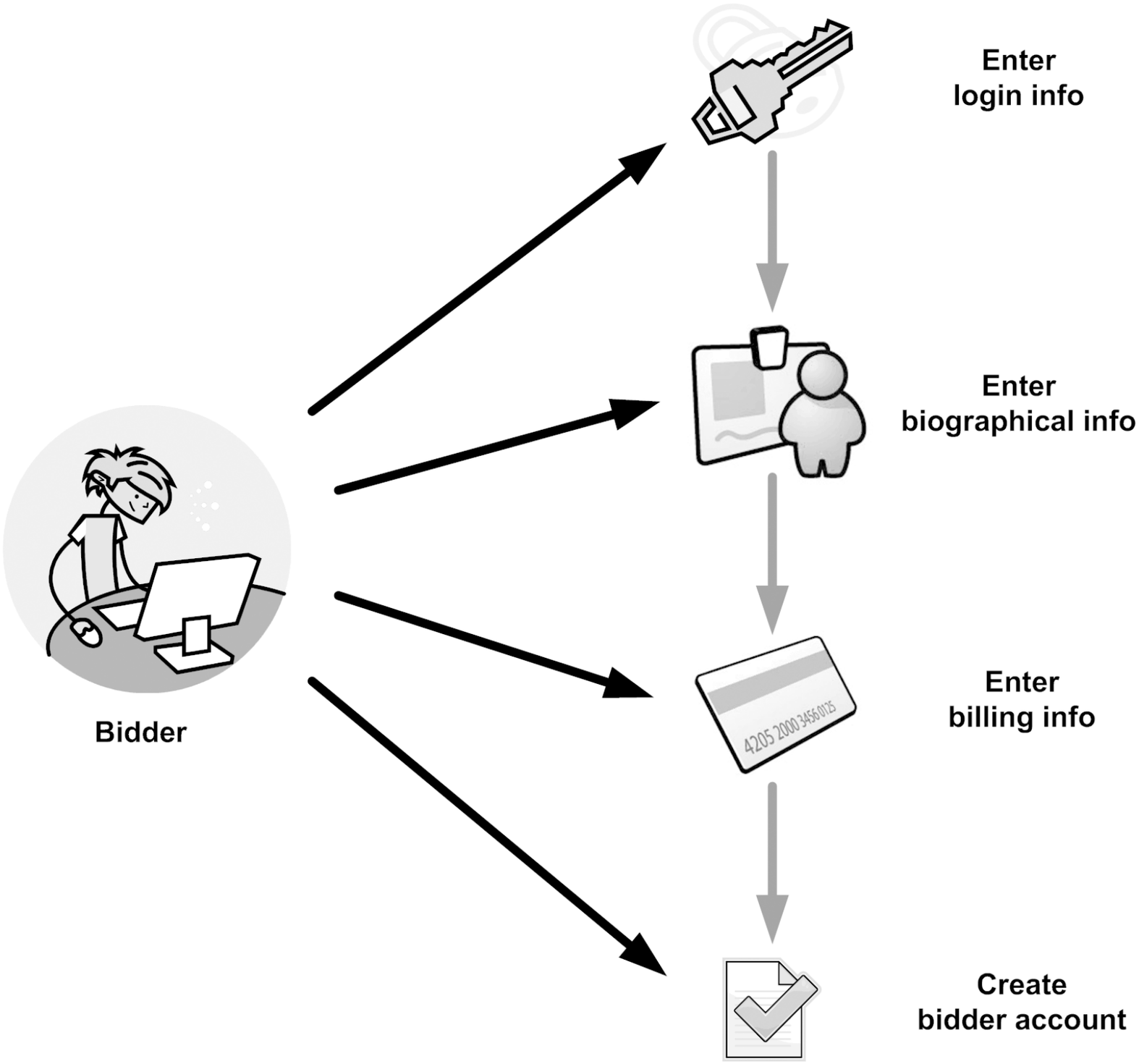
"Don't call us, we'll call you"

[EiA]

EJBs live in a "**pool**"

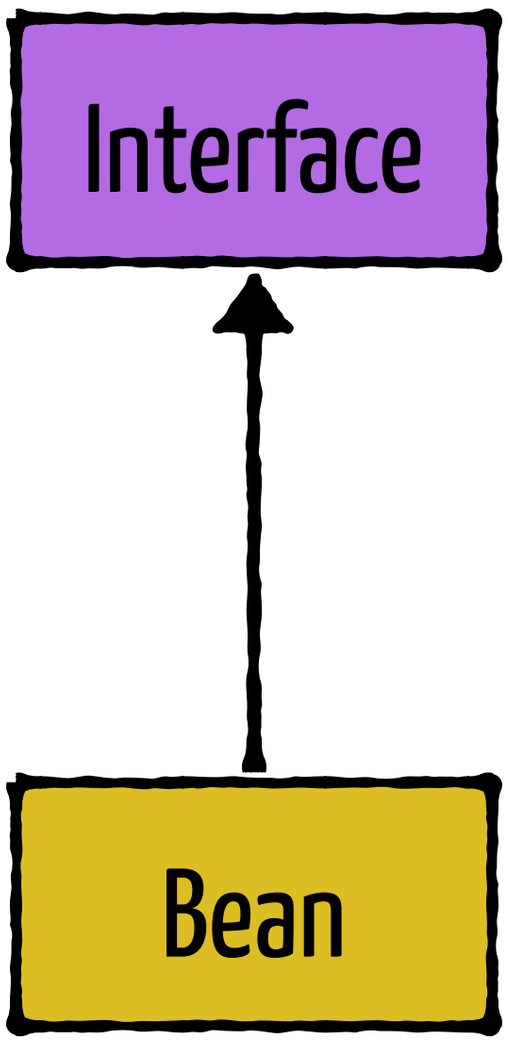


Domain beans live during a **Session**



State(less|ful) beans | 101

Stateless beans : POJO + Annotations

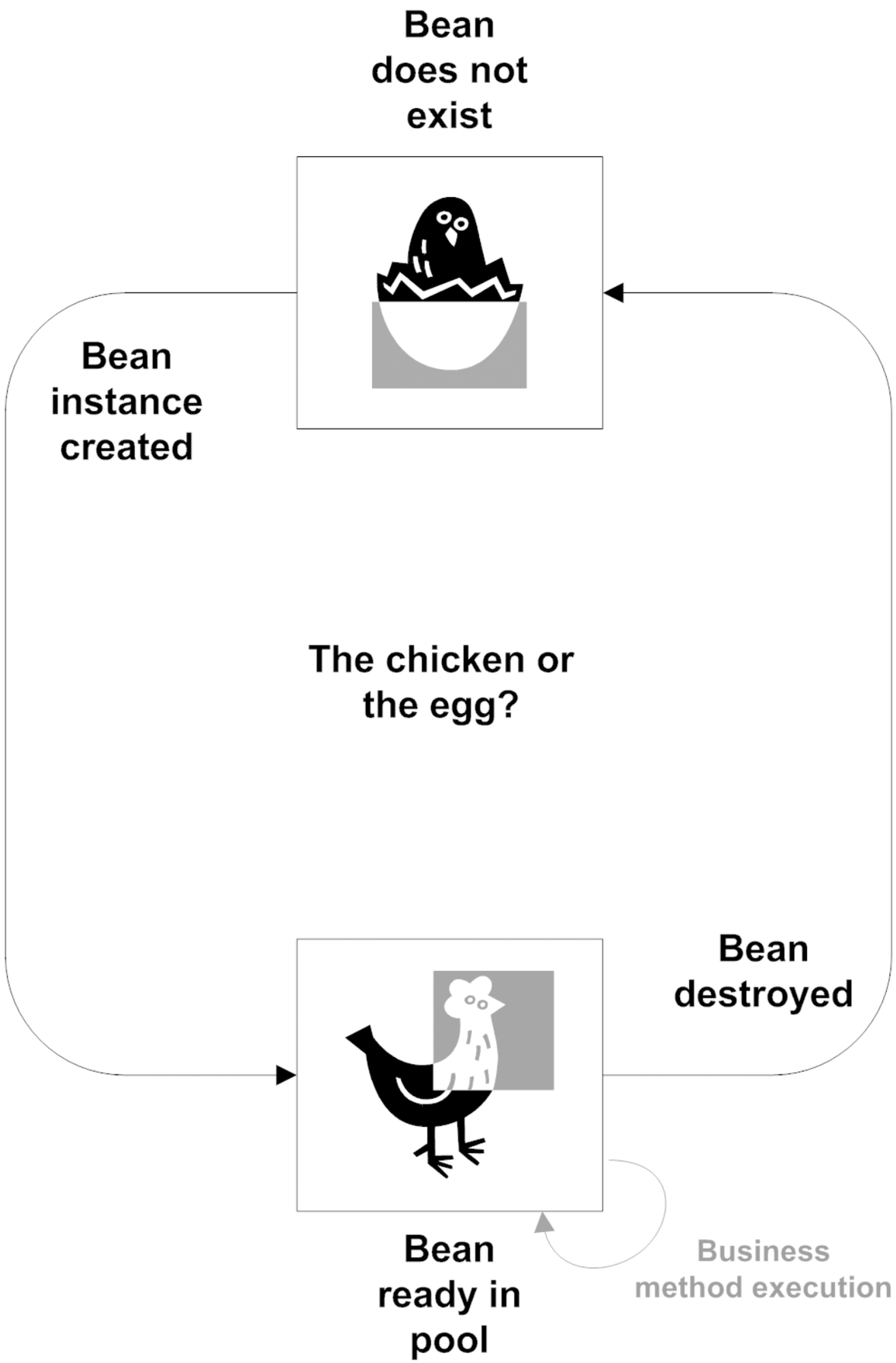



```
public interface PetManager {
    public Pet create(String name);
}
```

```
@Stateless
public class PetManagerBean implements PetManager {

    @PersistenceContext
    EntityManager entityManager;

    @Override
    public Pet create(String name){
        Pet p = new Pet(name);
        entityManager.persist(p);
        return p;
    }

}
```

Lifecycle

Handled by the container



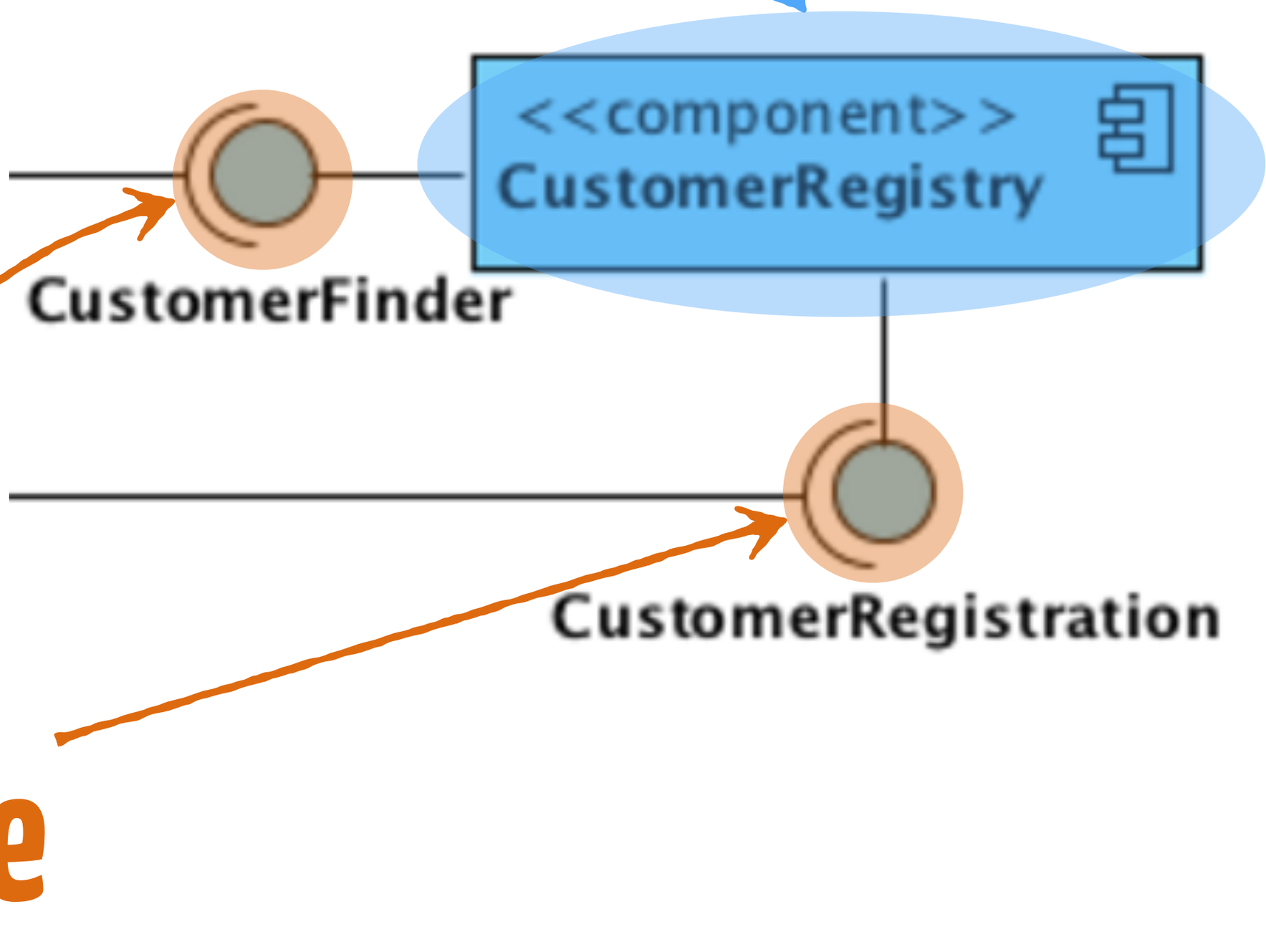
Consuming a Bean: Inversion of Control

```
@EJB
private PetManager manager;
```

Interface

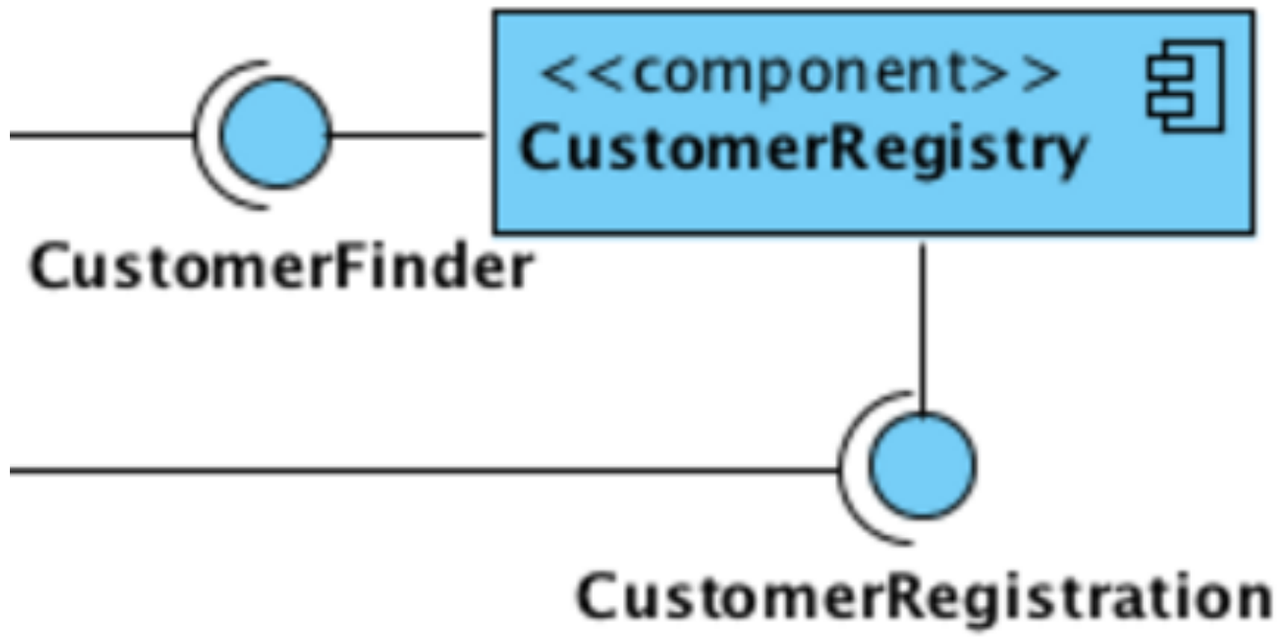
```
@Test
public void testCreation() throws Exception {
   Pet jinx = manager.create("Jinx");
   assertEquals(jinx.name, "Jinx");
}
```

Stateless Bean



Interface

```java
@Local
public interface CustomerFinder {

    Optional<Customer> findByName(String name);

}
```




```java
@Local
public interface CustomerRegistration {

    void register(String name, String creditCard)
            throws AlreadyExistingCustomerException;

}
```

```java
@Stateless
public class CustomerRegistryBean
        implements CustomerRegistration, CustomerFinder {

    @EJB
    private Database memory;

    /**********************************************
     ** Customer Registration implementation **
     **********************************************/

    @Override
    public void register(String name, String creditCard)
            throws AlreadyExistingCustomerException {
        if(findByName(name).isPresent())
            throw new AlreadyExistingCustomerException(name);
        memory.getCustomers().put(name, new Customer(name, creditCard));
    }


    /**********************************************
     ** Customer Finder implementation **
     **********************************************/

    @Override
    public Optional<Customer> findByName(String name) {
        if (memory.getCustomers().containsKey(name))
            return Optional.of(memory.getCustomers().get(name));
        else
            return Optional.empty();
    }

}
```

Persistence mock

First Conclusions

Architecture is a creative process

The same choice will not win each time

Tradeoffs are the key

“If you only have an **hammer**, everything looks like a **nail**

Labs

1h: From the CookieFactory code to diagrams

- Look at interfaces, annotations, dependencies
- Make explicit components and their interfaces

1h: Project work on Poly'Diploma

- Find components, interfaces, annotations, dependencies