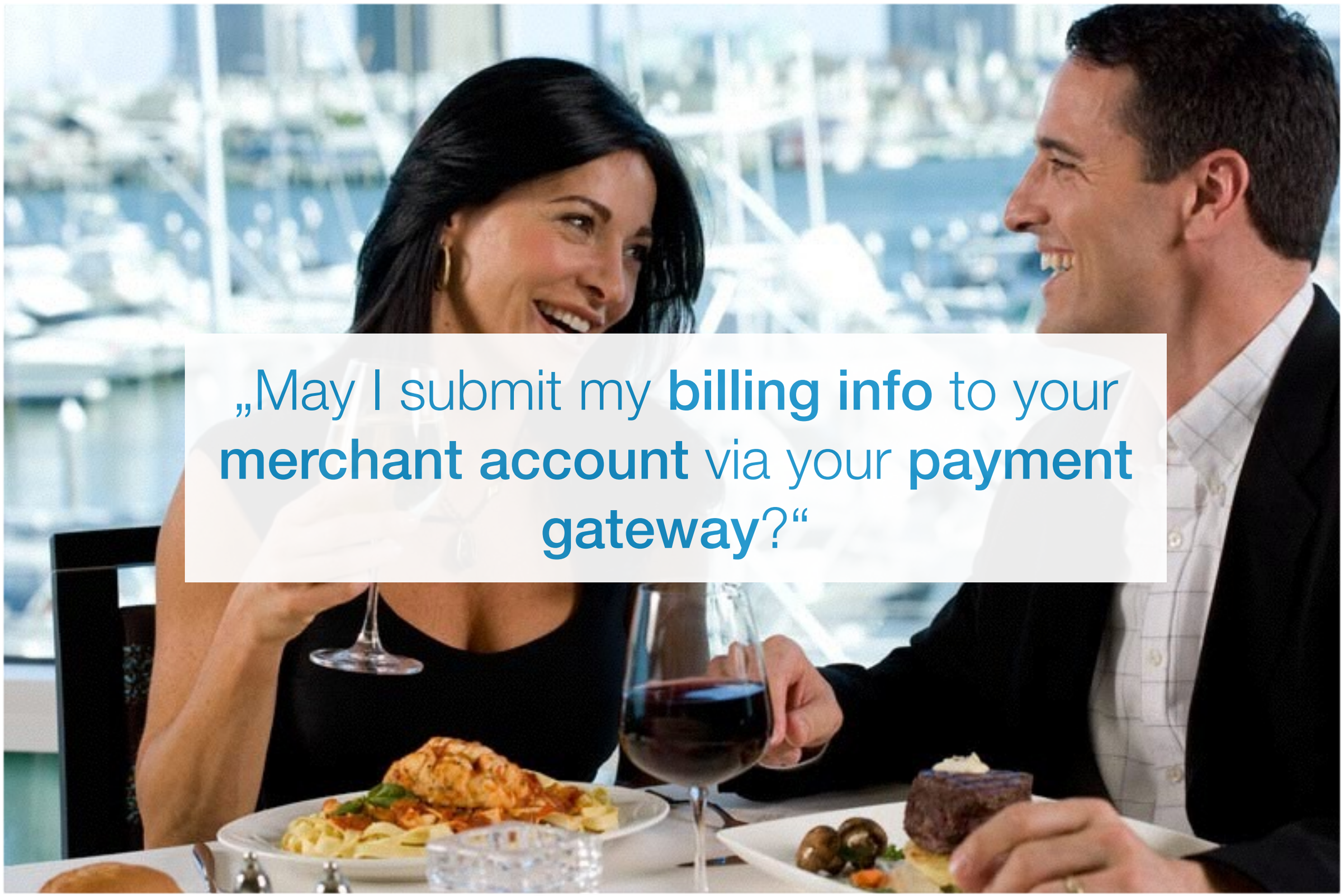
Not your Father's

Java EE

@**mobile**Larson
@_**open**Knowledge

Lars **Röwekamp** | CIO New Technologies

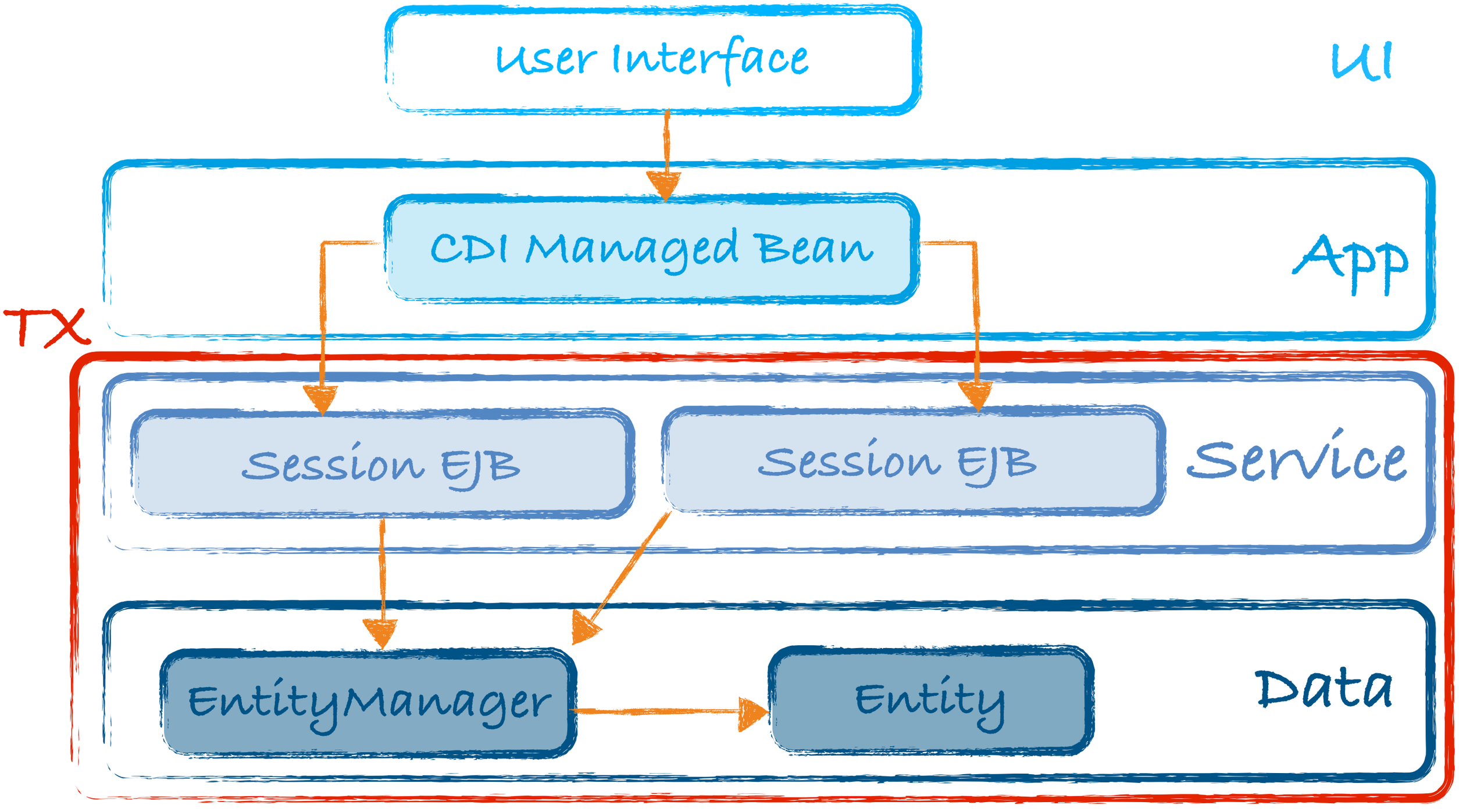

„May I submit my **billing info** to your **merchant account** via your **payment gateway**?“


FAIL

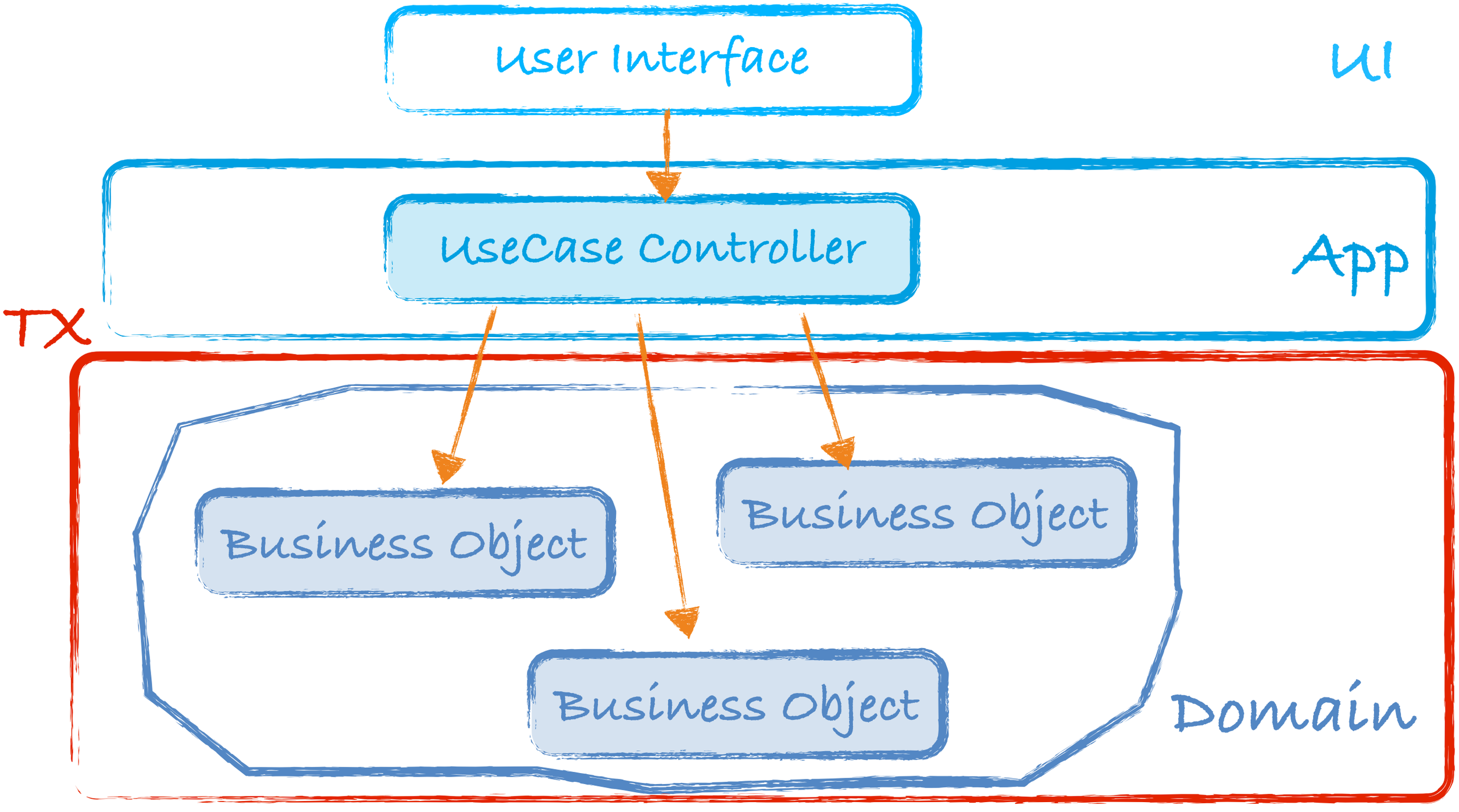
Wo liegt das Problem ...

„Too many times application architects focus on the technical problems instead of designing for the problem domain."

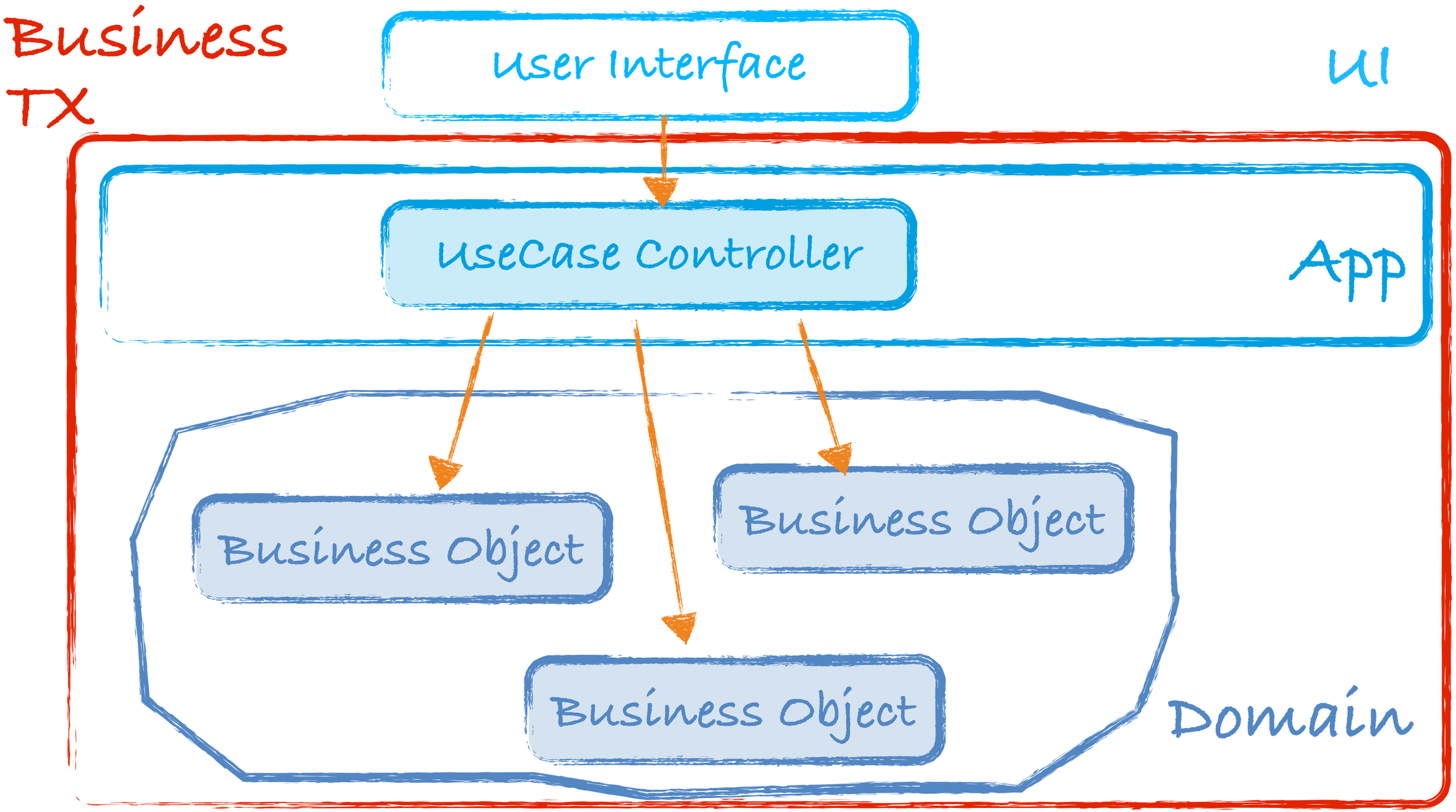
Wo liegt das Problem ...



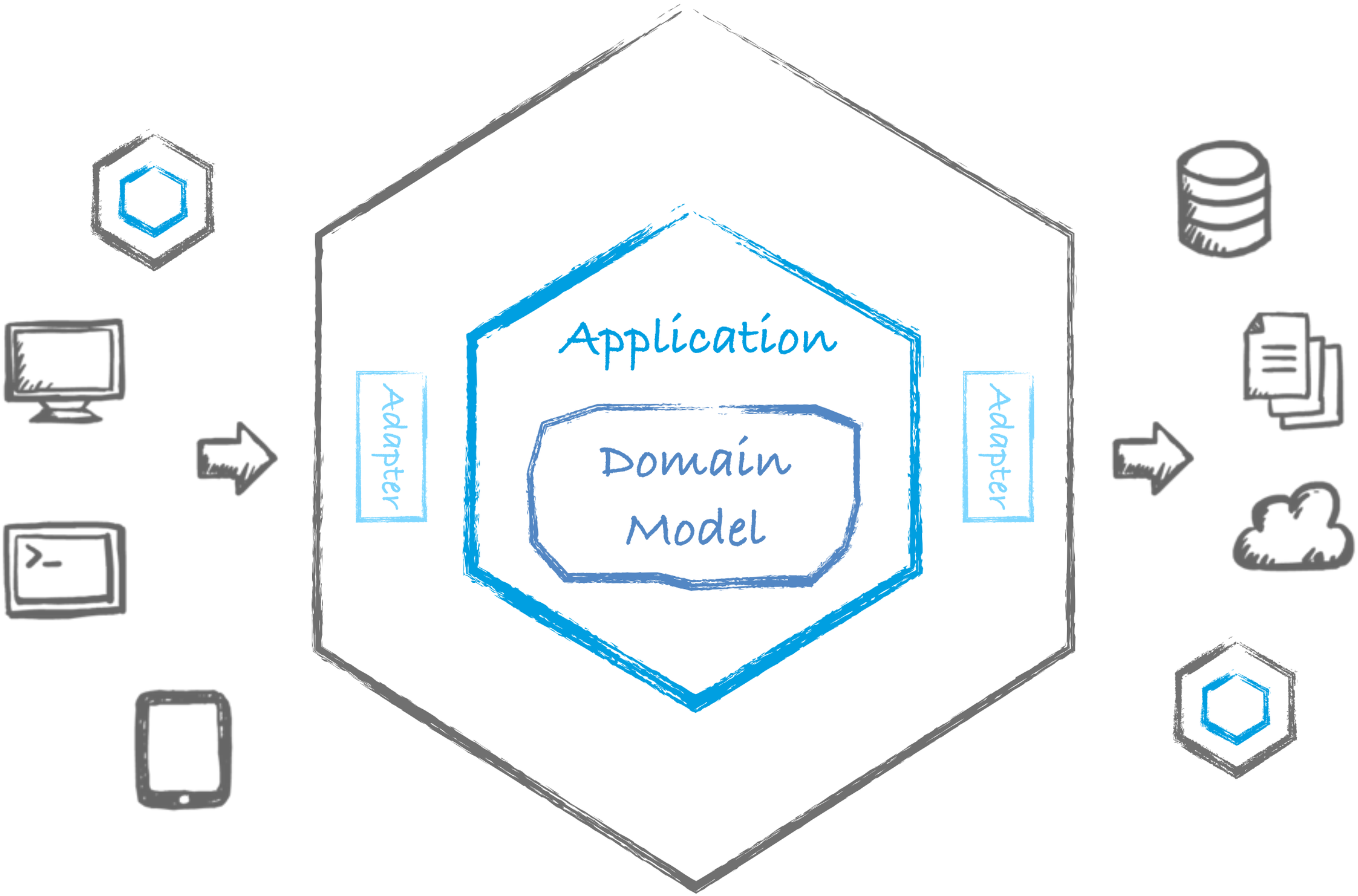
Es wäre doch viel schöner, wenn ...



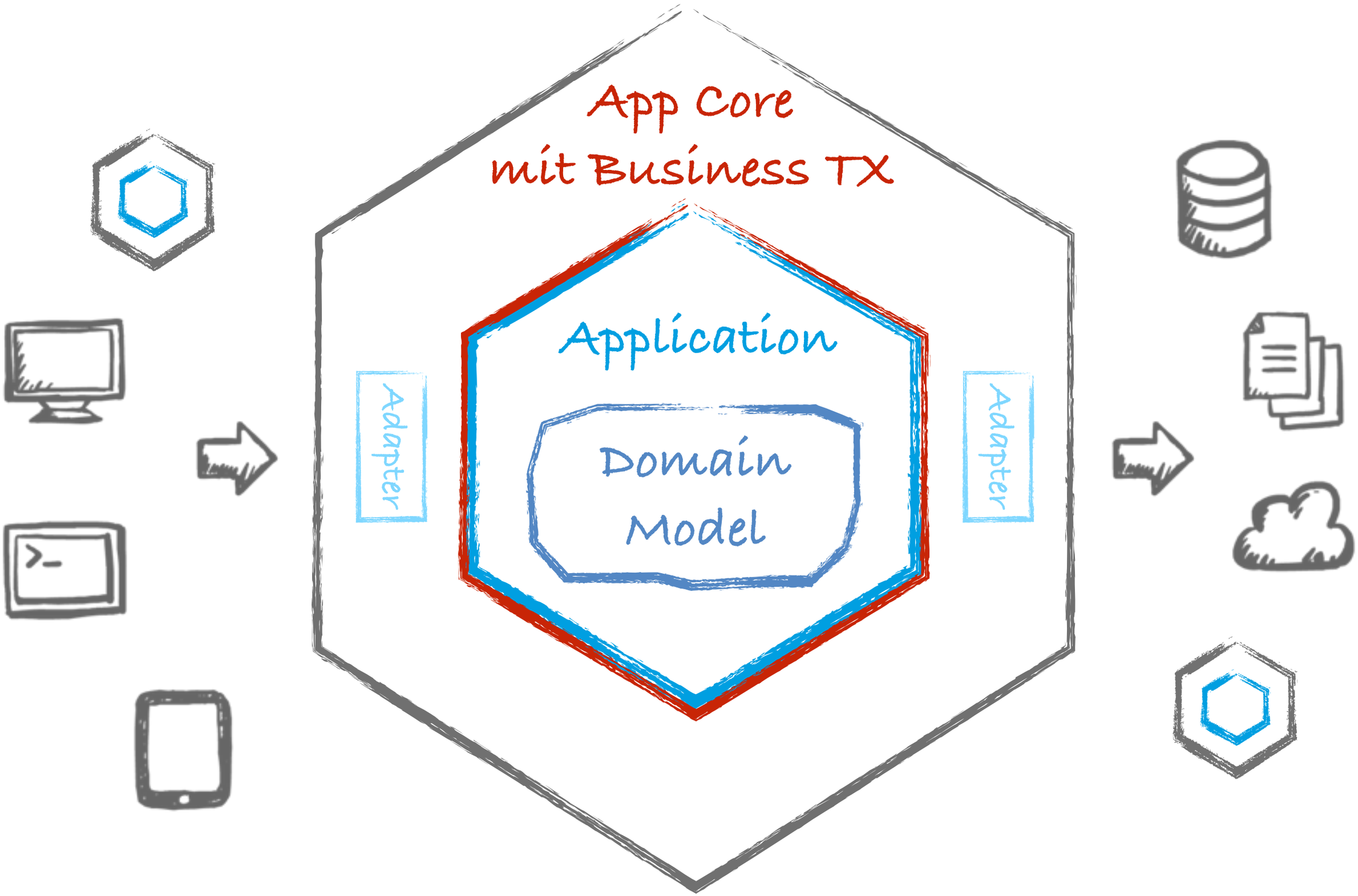
Es wäre doch viel schöner, wenn ...



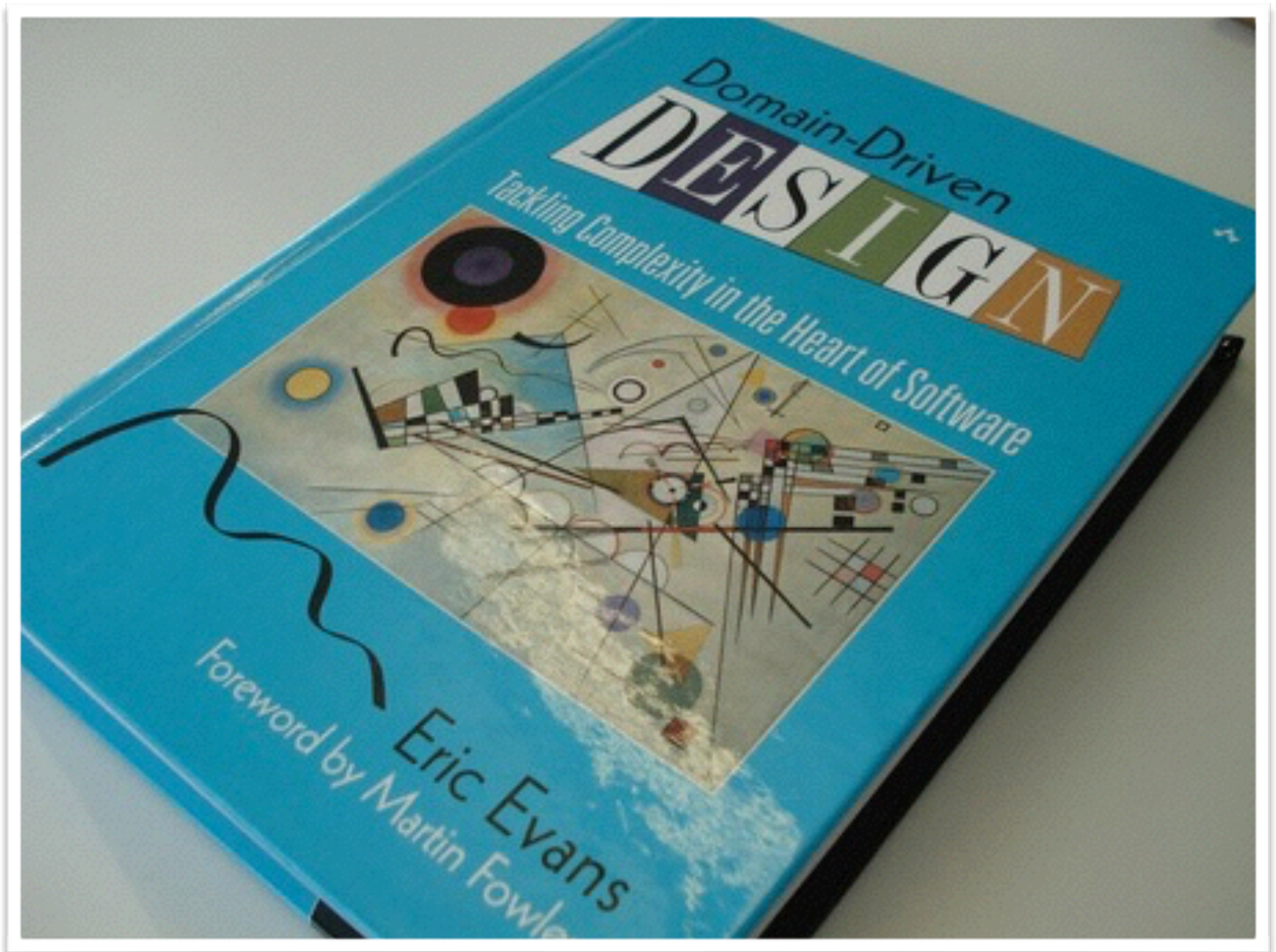
Es wäre doch viel schöner, wenn ...



Es wäre doch viel schöner, wenn ...



Es könnte alles so einfach sein ...



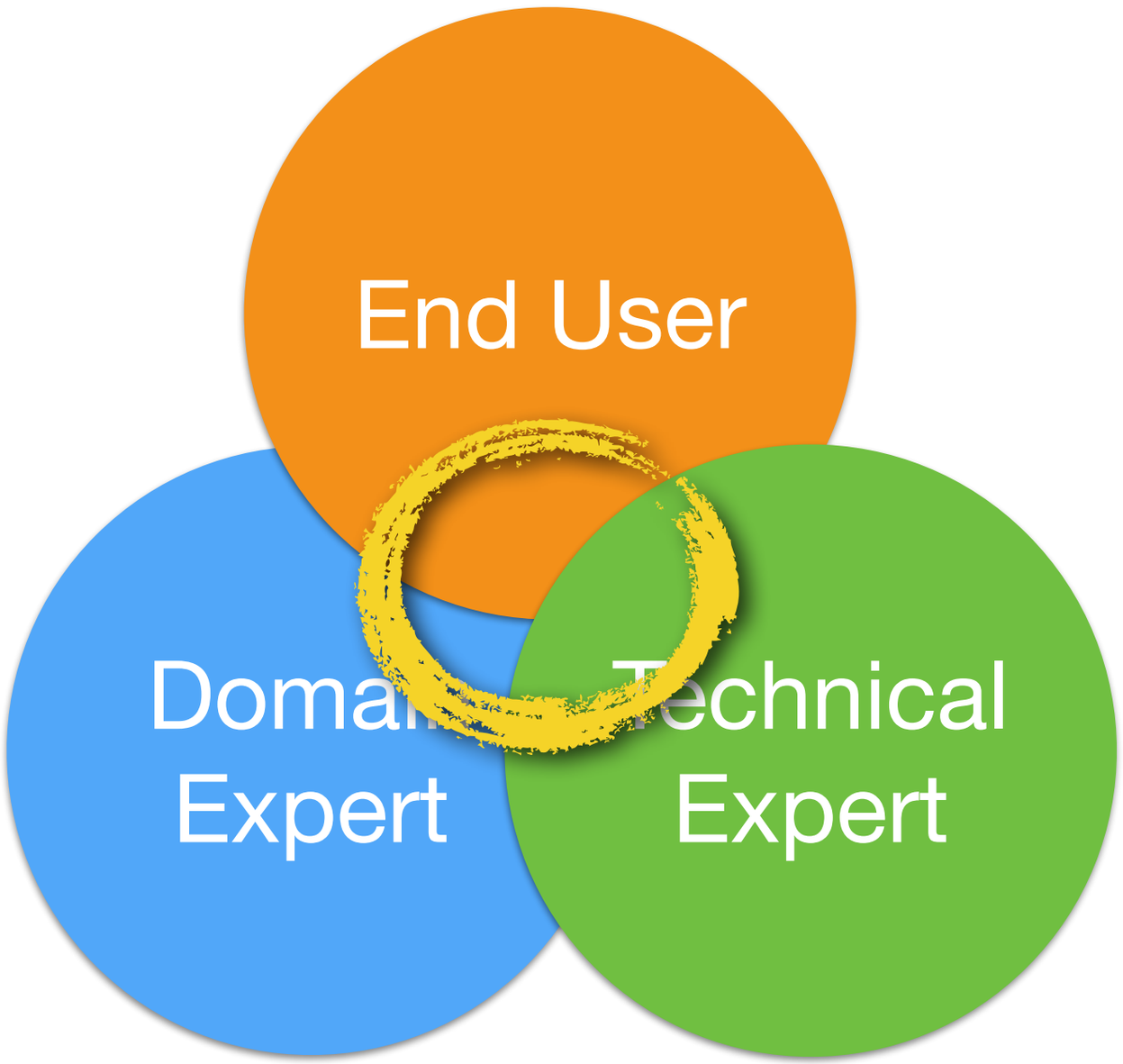
Das kleine 1x1 des DDD …

Eric Evans says

„For most software projects, the **primary focus** should be on the **domain** and **domain logic**.“

„Complex domain design should be **based on a model**.“

Das kleine 1x1 des DDD …



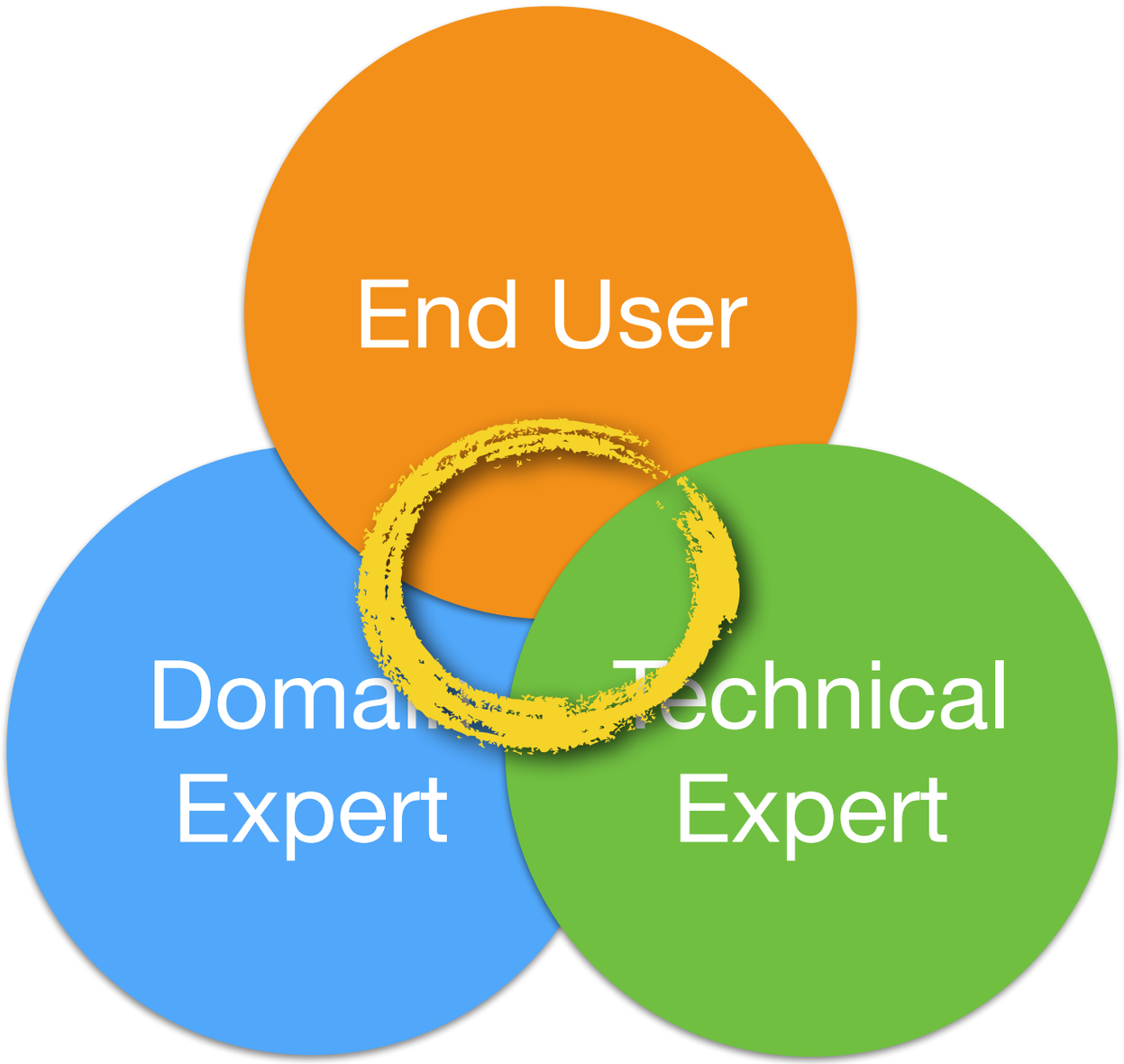
fachliche **Domäne**

Die gemeinsame

fachliche Welt

in der wir „leben“.

Das kleine 1x1 des DDD …



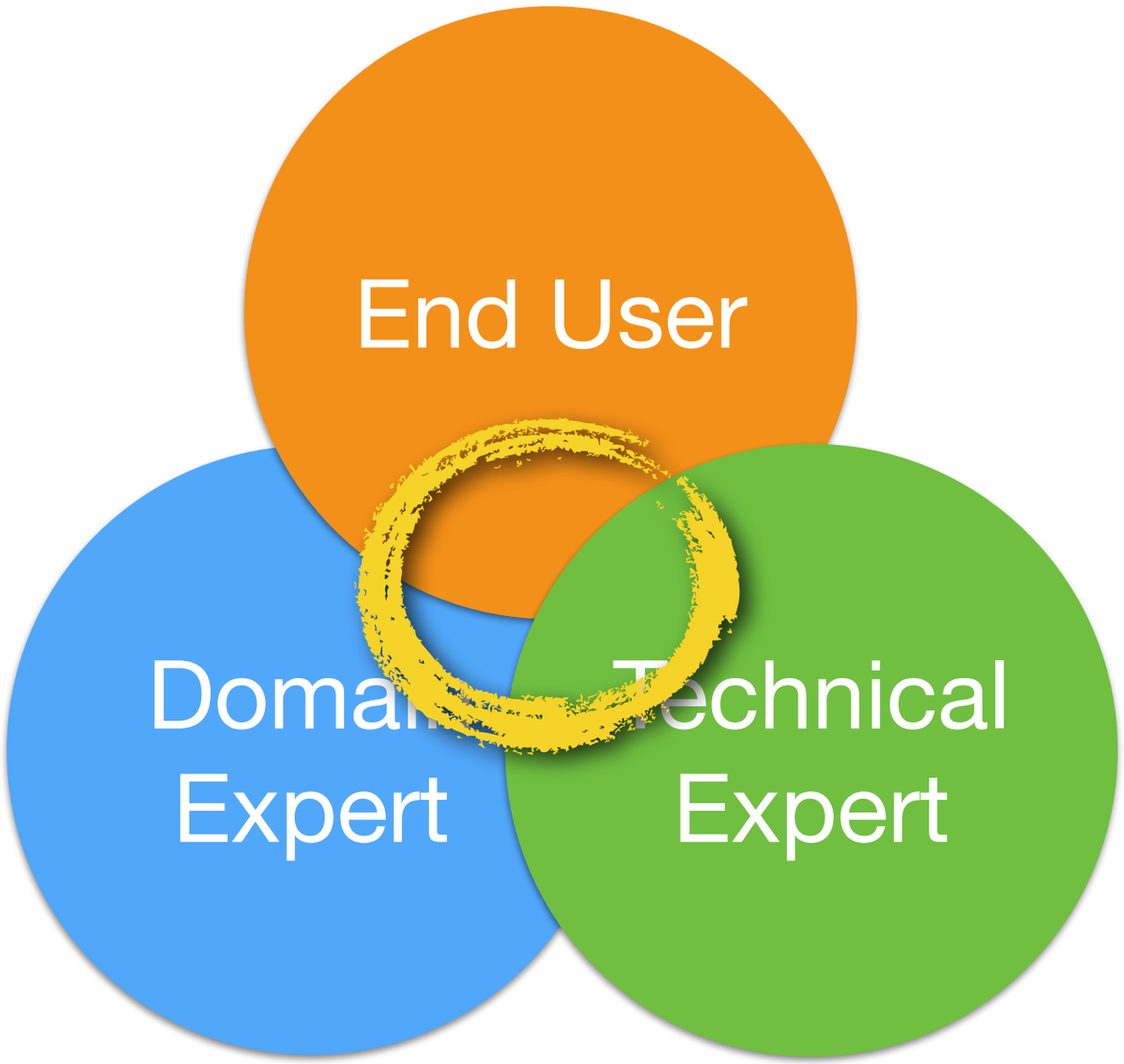
Ubiquitous Language

Fachsprache als Basis

für ein gemeinsames

Domain Model

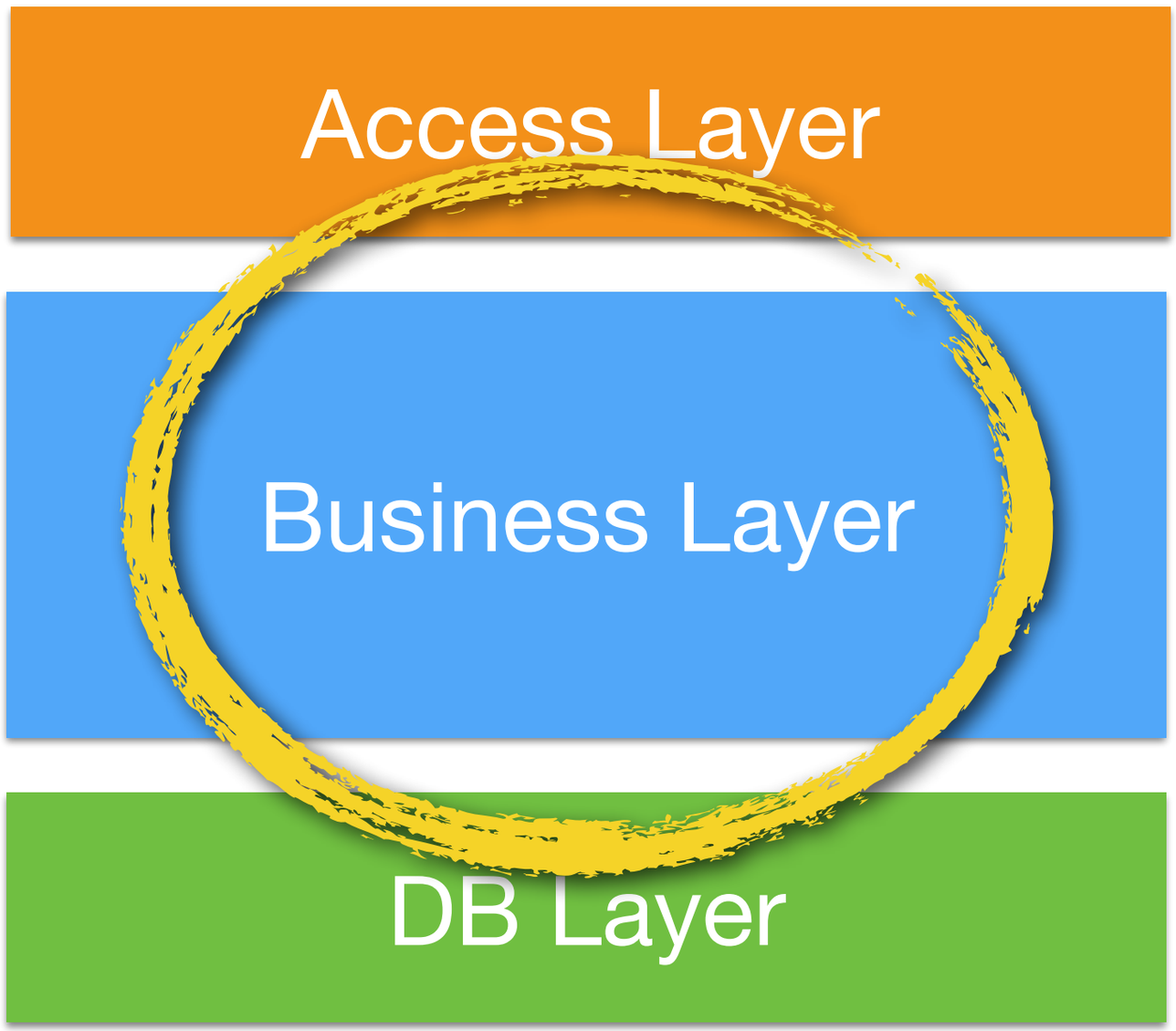
Das kleine 1x1 des DDD …



Rich Domain Model

„Produce Software that makes perfectly **sense to business**, not only to coder“

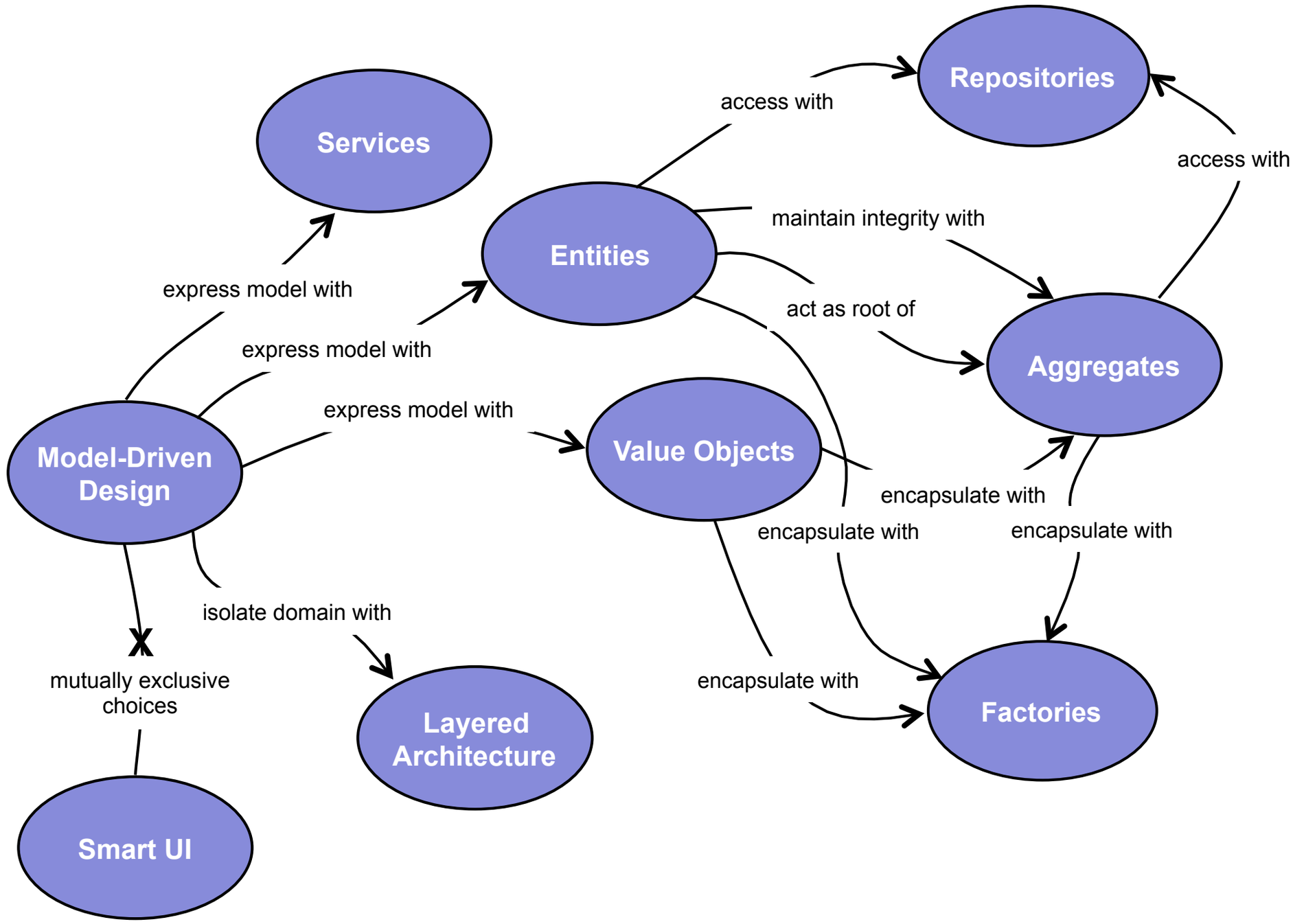
Das kleine 1x1 des DDD …



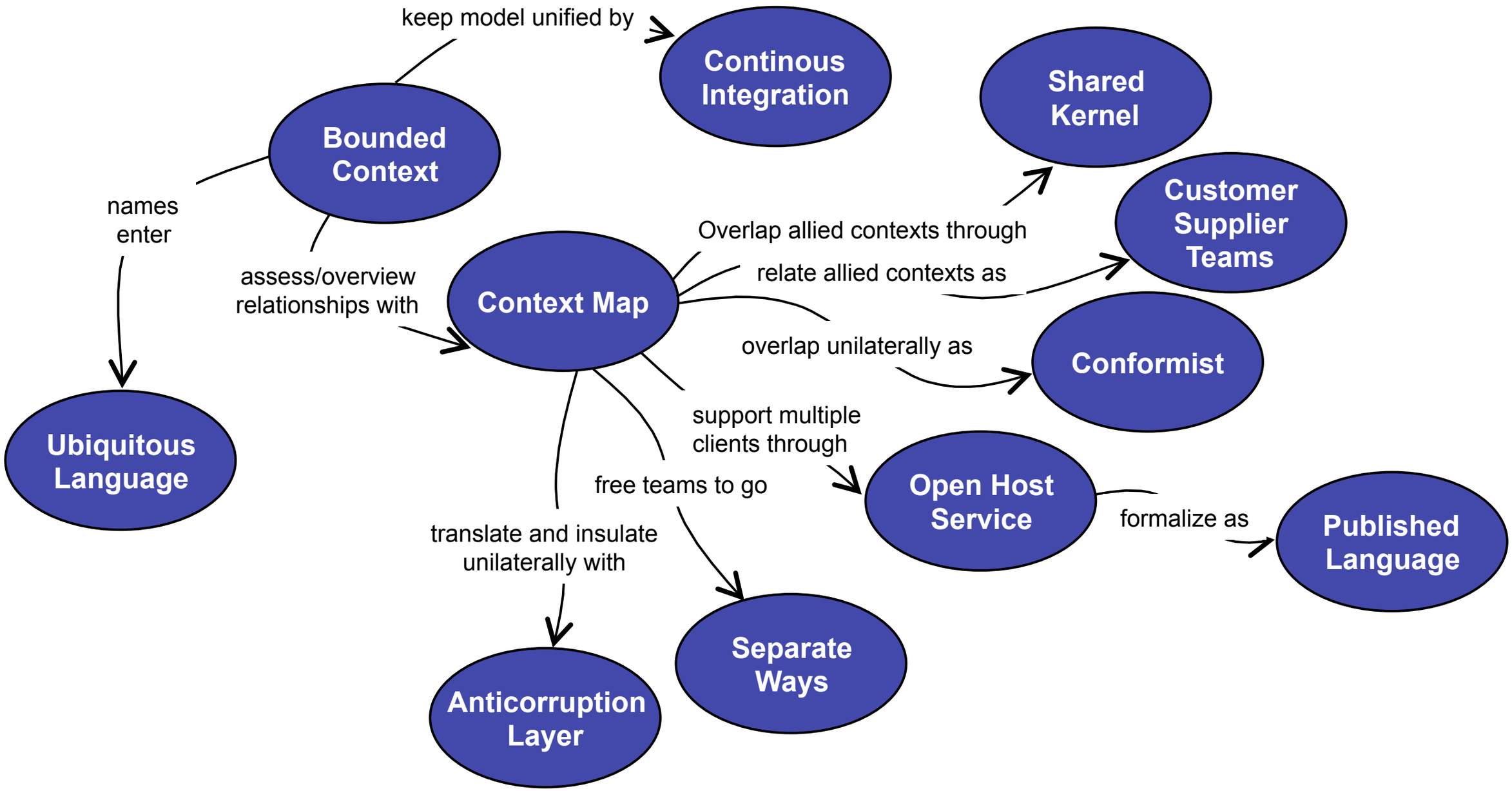
Rich Domain Model

„Produce Software
that is always
in a correct state,
not only partially“

Das kleine 1x1 des DDD …

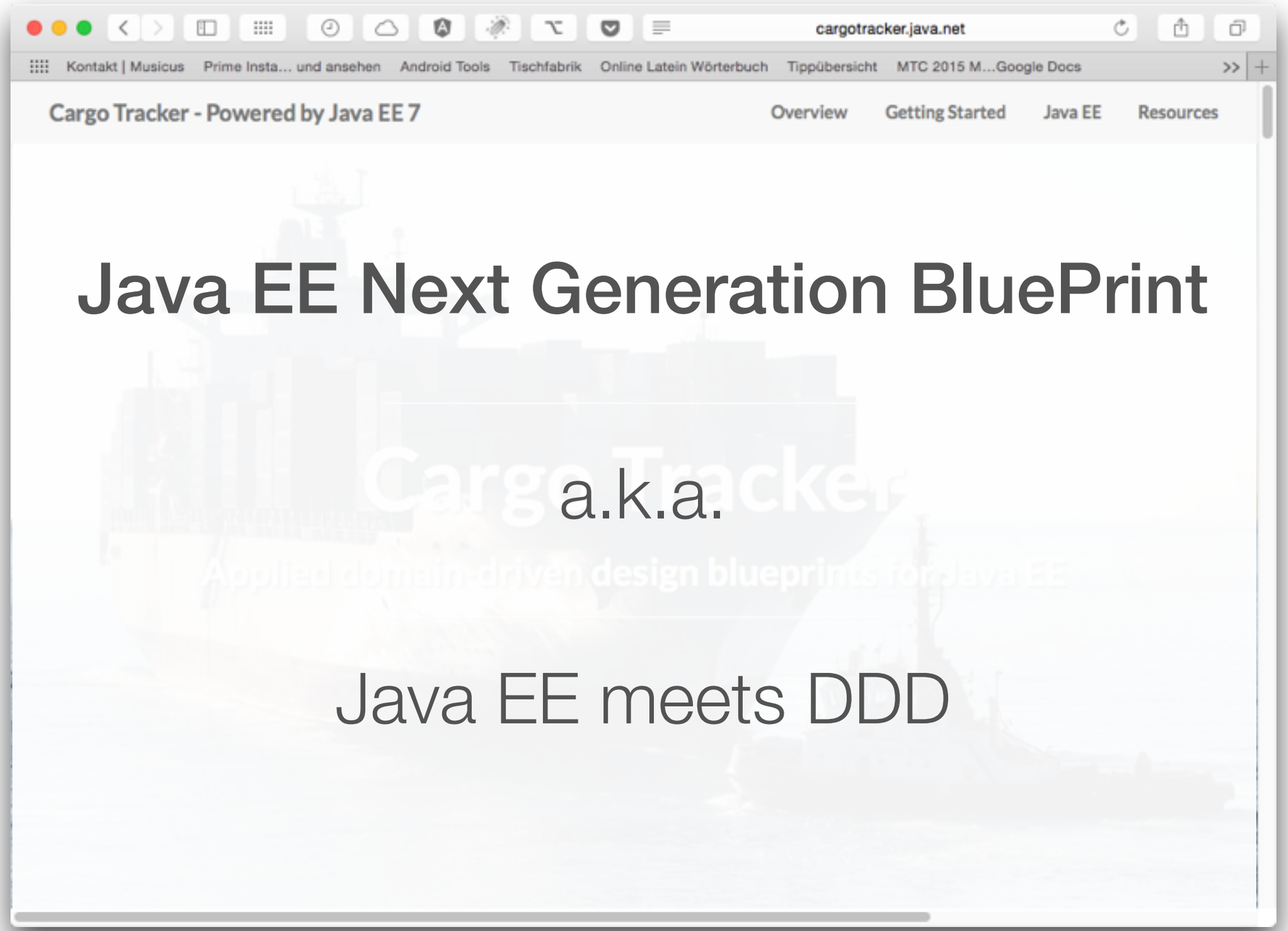


Das kleine 1x1 des DDD …

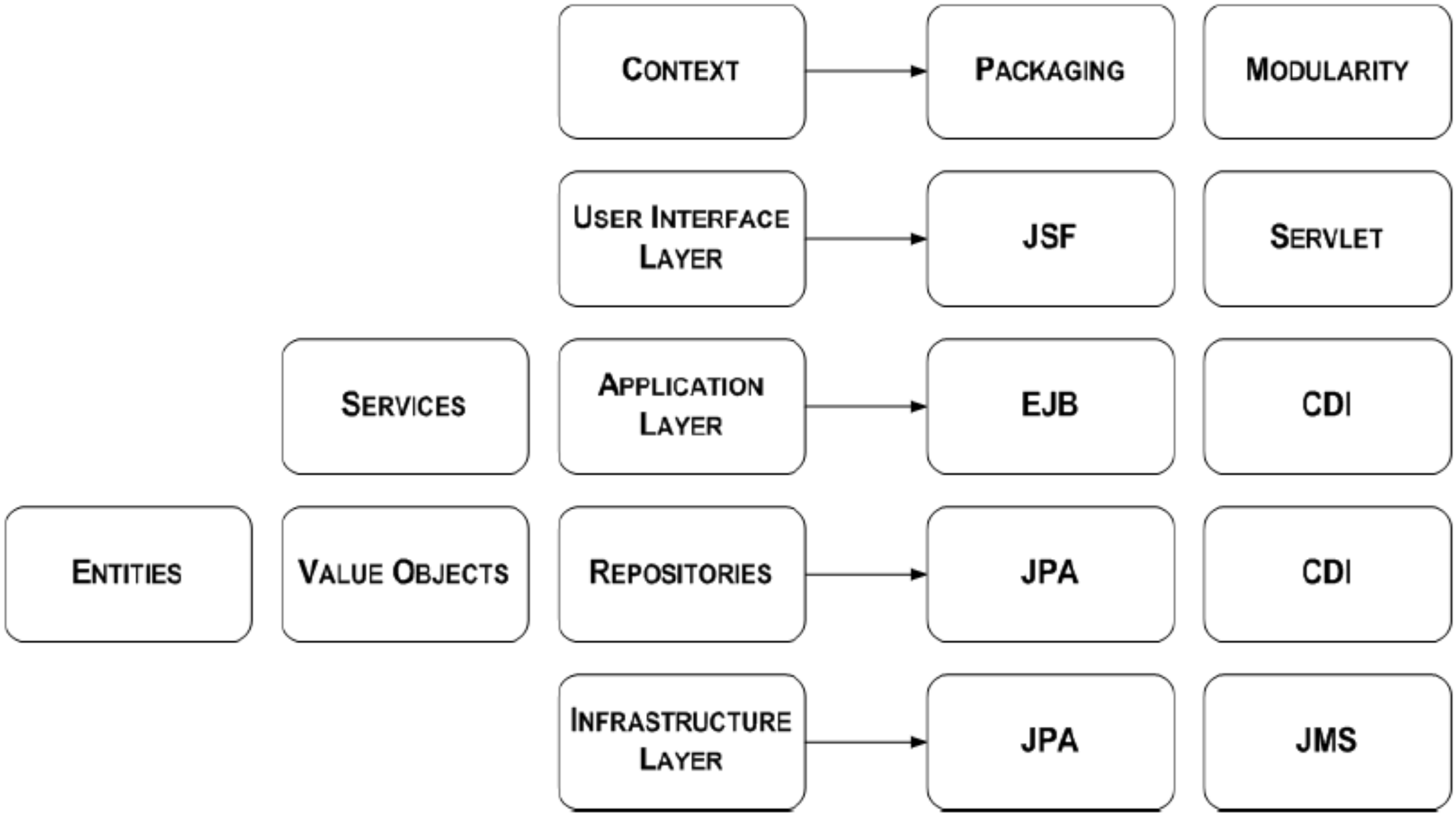


Passt das zu Java EE?

Passt das zu Java EE? Klar …

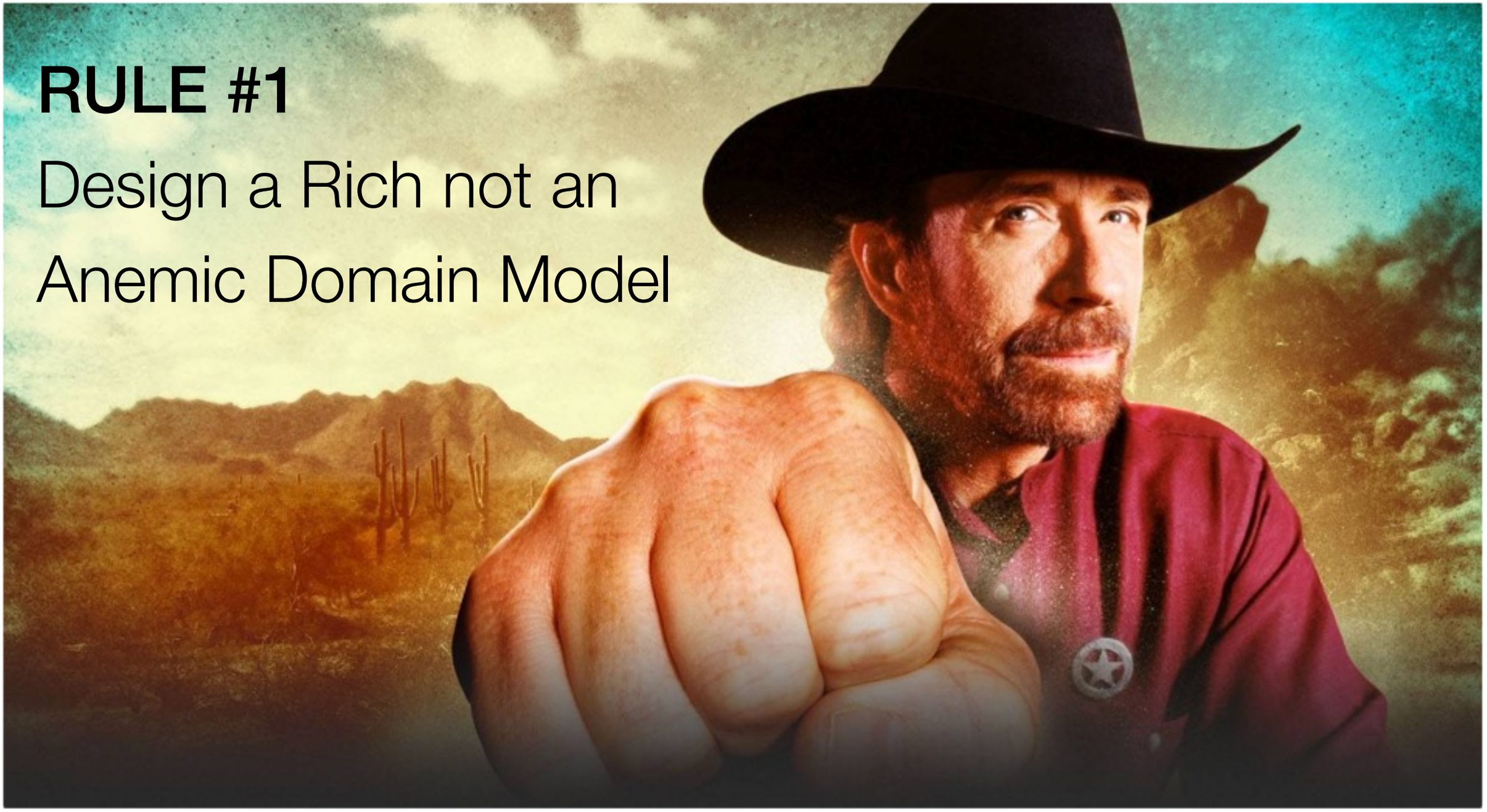


Passt das zu Java EE? Klar …



Passt das zu Java EE? Klar, aber …

Always follow the Rules …



Rule #1: Rich not Anemic Domain Model

```
/**
  * Represents a business customer with a
  * full qualified name, a date of birth indicating
  * if the customer is „sui juris“ and a valid address
  * that may change after a relocation.
  */
public class Customer {

    private String firstName;
    private String lastName;

    private Date birthDate;
    private Address address;

    public void setFirstName(String aFirstName) { ...  }
    public String getFirstName() { return firstName;  }

    public void setLastName(String aLastName) { ... }
    public String getLastName() { return lastName; }

    public void setBirthDate(Date birthDate) { ... }
    public Date getBirthDate() { return birthDate; }

    public void setAddress(Address address) { ... }
    public Address getAddress() { return address; }

}
```

Fachlichkeit?

Rule #1: Rich not Anemic Domain Model

```
/**
   * Represents a customer data access object to
   * encapsulate all db relevant activities
   */
public class CustomerDao {

    ...

    public List<Customer> findByZipCodeOrCity(
        String zipCode,
        String city) {
          return ...;
    }

    public List<Customer> findByZipCodeOrCityOrCityLimit(
        String zipCode,
        String city,
        int order) {
          return ...;
    }
}
```

sprechend?

Rule #1: Rich not Anemic Domain Model

Zur **Erinnerung**

„Wir möchten ein **sprechendes Domain Model**, mit einem **sprechenden API**, welches wir in sich **konsistent erzeugen** und im weiteren Verlauf konsequent **konsistent erhalten**.“

Rule #1: Rich not Anemic Domain Model

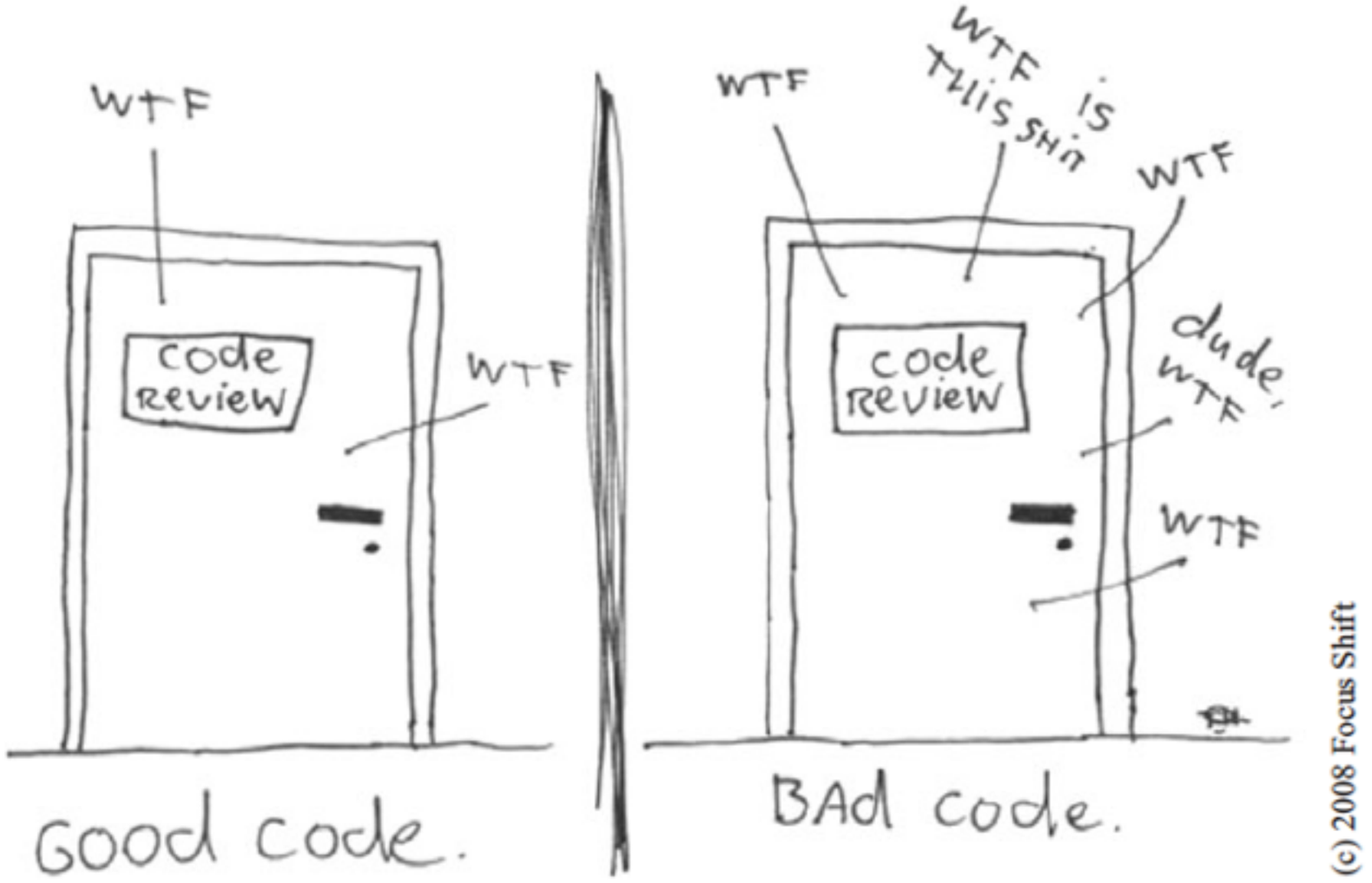
Anforderungen

- sprechendes **Modell**
- sprechendes **API**

- konsistent **erzeugen**
- konsistent **erhalten**

- **Fachlichkeit, Validierung, Konvertierung, Normalisierung** und **Konsistenzprüfung!**

The ONLY VALID MEASUREMENT OF CODE QUALITY: WTFs/MINUTE



Rule #1: Rich not Anemic Domain Model

Anforderungen

- sprechendes **Modell**
- sprechendes **API**
- konsistent **erzeugen**
- konsistent **erhalten**

Rule #1: Rich not Anemic Domain Model

sprechend

```
// (JPA) Entity with Value Objects
public class Customer extends HumanBeing {

   private FirstName firstName;
   private LastName lastName;
   private FamilyStatus familyStatus;
   private Customer partner;
   private Address postalAddress;
   private Map<PhoneNumberType, PhoneNumber> phoneNumbers;
   ...
   public void marry(Partner partner,LastName newLastName) {
      lastName = notNull(newLastName)
      familyStatus = FamilyStatus.MARRIED;
      // Ensure bidirectional integrity for partner entity!
      ...
   }

   public boolean isMarried() {
      return (familyStatus == FamilyStatus.MARRIED);
   }
}
```

Rule #1: Rich not Anemic Domain Model

```
// (JPA) Entity with Value Objects
public class Customer extends HumanBeing {

   ...
   private Map<PhoneNumberType, PhoneNumber> phoneNumbers;
   ...

   public void changeHomePhoneNumber(
      PhoneNumber changedHomePhoneNumber) { ... }
   public void disconnectHomePhone() { ...}

   public void changeMobilePhoneNumber(
      PhoneNumber changedHomePhoneNumber) { ... }
   public void disconnectMobilePhone() { ... }

   ...

}
```

sprechend

Rule #1: Rich not Anemic Domain Model

```
/**
   * Represents a customer data access object to
   * encapsulate all db relevant activities
   */
public class CustomerDao {

    ...
    public List<Customer> findByZipCodeOrCityOrStreet(
        String street,
        String city,
        String zipCode) {
          return ...;
    }

    public List<Customer> findByAnyOf(
        ZipCode zipCode,
        City city,
        Street order) {
          return ...;
    }
}
```

sprechend

Hmm, passt das **wirklich** zu Java EE?

Rule #1: Rich not Anemic Domain Model

```
/**
  * Simple value object for first name using the
  * common base <code>SimpleValueObject</code>
  */
public class FirstName extends SimpleValueObject<String> {

    public FirstName(String aValue) {
        super(aValue);
    }

    public String getText() {
        return super.getValue();
    }

}
```

Common Base

- Passt das wirklich: **Value Objects**?

Rule #1: Rich not Anemic Domain Model

```
/** Abstract value object as a common base  */
public abstract class SimpleValueObject<T extends Comparable>
          implements Serializable, Comparable<SimpleValueObject<T>> {

   private T value;

   public SimpleValueObject(T aValue) { value = aValue;  }

   protected T getValue() { return value; }

   public boolean equals(Object o) {
      return ...;
   }

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

   public String toString() {
      return ...;
   }

   public int compareTo(SimpleValueObject<T> o) {
      return ...;
   }
}
```

Common Base

Rule #1: Rich not Anemic Domain Model

```
/**
  * Embeddable date of birth for (re)use in value objects
  */
@Embeddable
public class DateOfBirth {
   private Date dateOfBirth;

   ...
}

/**
  * Customer entity making use of date of birth embeddable
  */
@Entity
public class Customer {


   @Embedded
   @AttributeOverride(name=„dateOfBirth",
                      column=@Column(name="CUST_BIRTHDATE"))
   private DateOfBirth dateOfBirth;

   ...
}
```

VO, Entities & JPA

Rule #1: Rich not Anemic Domain Model

VO & JPA Queries

```
/**
  * Using JPA-QL and ValueObjects
  */
@Entity
@Table(name = "TAB_USER", ...)
@NamedQueries({
    @NamedQuery(name = "User.findByEmail",
                query = "select u from User u where u.email = :email"),
    @NamedQuery(name = "User.findByEmailString",
                query = "select u from User u " +
                        "    where u.email.value = :emailString")
})
public class User extends AbstractEntity {

  protected static final String EMAIL_COLUMN = "EMAIL";

  @AttributeOverride(name = "value",
                     column = @Column(name = EMAIL_COLUMN,
                     unique = true,
                     nullable = false))
   private Email email;

  ...

}
```

Rule #1: Rich not Anemic Domain Model

```
/** CDI based JSF Controller managing user related input  */
@Named("customerController")
@RequestScoped
public class CustomerController {

   private DateOfBirth dateOfBirth;
   private ZipCode zipCode;
    ...;

   public DateOfBirth getDateOfBirth() {
      return dateOfBirth;
   }

   public void setDateOfBirth(DateOfBirth newDateOfBirth) {
      dateOfBirth = newDateOfBirth;
   }

   public ZipCode getZipCode() {
      return zipCode;
   }

   public void setZipCode(ZipCode newZipCode) {
      zipCode = newZipCode;
   }
    ...
}
```

VO & JSF

Rule #1: Rich not Anemic Domain Model

```
<!--
   Value Object binding in JSF xhtml page via
    Unified Expression Language & JSF Faces Converter
-->

<!-- using JSF converter for Value Object of type ZipCode (read) -->
<h:outputText value="#{customerController.zipCode}" />

<!-- using JSF converter for Value Object of type ZipCode (write) -->
<h:inputText value="#{customerController.zipCode}" ... />
```

VO & JSF

```
@FacesConverter(forClass = ZipCode.class)
public class ZipCodeConverter extends
                        SimpleValueObjectConverter <ZipCode> {

}
```

Rule #1: Rich not Anemic Domain Model

Anforderungen

- sprechendes **Modell**
- sprechendes **API**

- konsistent **erzeugen**
- konsistent **erhalten**

Rule #1: Rich not Anemic Domain Model

```
// JPA entity
@ValidPaymentMethod // cross-field constraint
@Entity
public class Customer {

   @Name // common constraint for type Name
    private FirstName firstName;

   @Name // common constraint for type Name
    private LastName lastName;

   @Valid // delegate validation to Class
    private CreditCard creditCard;

   @Valid // delegate validation to Class
    private BankAccount bankAccount;

   @Address(area = AddressArea.GERMANY) // special constraint
    private Address postalAddress;
    ...
}
```

konsistent

Rule #1: Rich not Anemic Domain Model

konsistent

```
// Builder Pattern - IN ACTION (easy version)

// create AddressBuilder
AddressBuilder builder = new AddressBuilder();

// set all known values
// BTW: could be also used in JSF and other frameworks
builder.setStreet(...);
builder.setStreetNumber(...);
builder.setZipCode(...);
builder.setCity(...);
builder.setState(...);
builder.setCountry(...);

// build valid and immutable Address object
Address address = builder.build();
```

Rule #1: Rich not Anemic Domain Model

konsistent

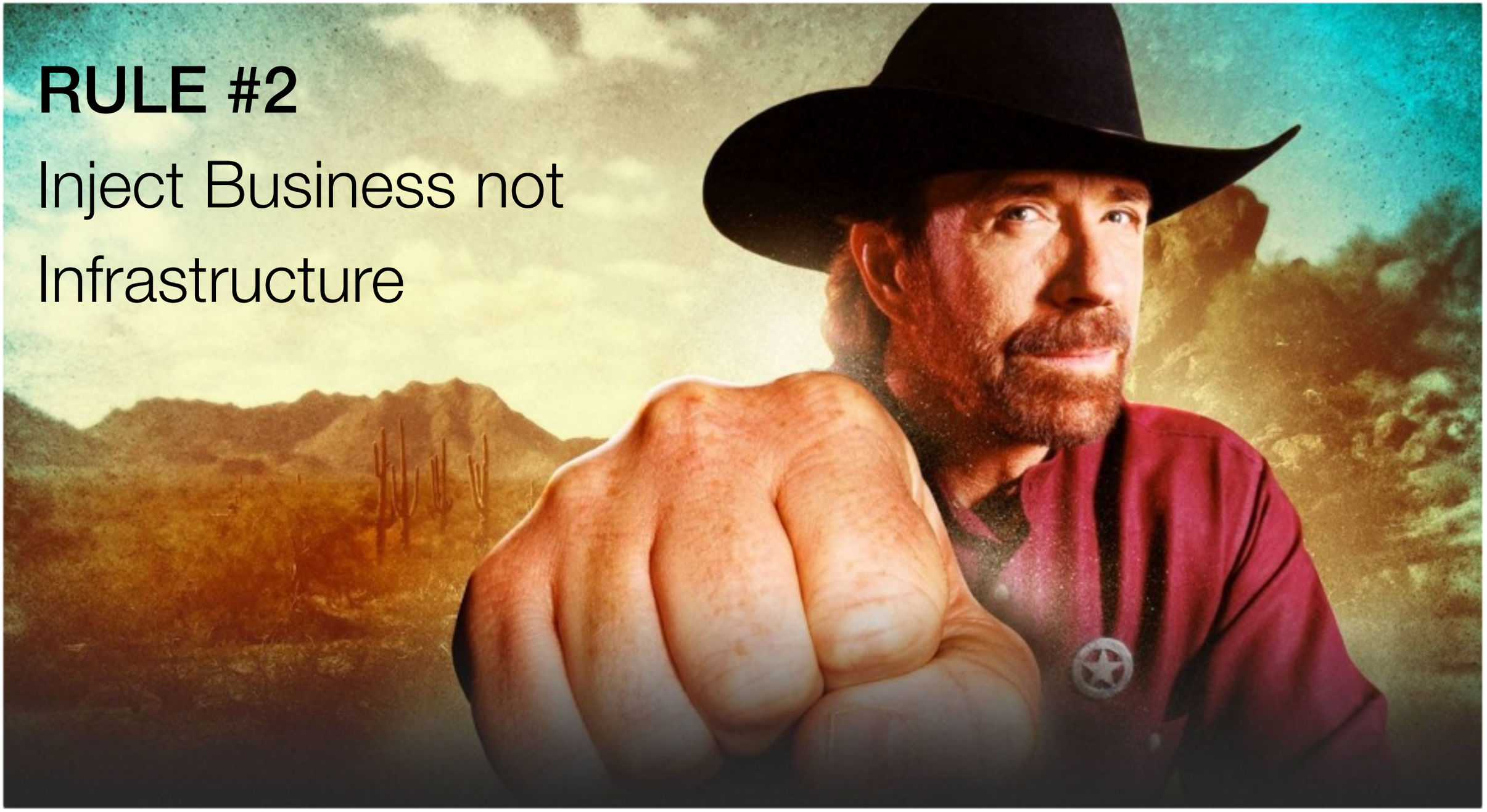
```
// Builder Pattern - IN ACTION (extended versio

// create AddressBuilder
Address address = newAddress()

   // set values
   // BTW: could be also used in JSF with own EL Resolver
   .forStreetNumber(...).ofStreet(...)
   .inCity(...).withZipCode(...)
   .lyingInState(...).ofCountry(...)

// build Address object
.build(...);
```

Rule #1: Rich not Anemic Domain Model

konsistent

```
// Class with Builder - extended version
public class Address {

   public static Builder newAddress() { return new Builder();  }
   ...

   public static class Builder {

      private Address address = new Address();

      public Builder ofStreet(String initialStreet) {
         address.street = validateStreet(initialStreet);
         return this;
      }

      public Address build() {
         validate();
         Address validAddress = address;
         address = new Address();
         return validAddress;
      }
   }
}
```

Always follow the Rules …



Rule #2: Inject Business not Infrastructure

```
// JSF UI Controller
@javax.enterprise.context.RequestScoped
@javax.inject.Named("createCustomerController")
public class CreateCustomerController {

    // inject CustomerService and MailService via @Inject

    @Inject  // CDI managed controller layer bean (SessionScoped)
    private AuthenticationController authController;

    private Customer customer;

    public String createCustomer() {
       currentCallCenterAgent = authController.getLoggedInCca();
       customer.setAuditInformation(currentCallCenterAgent);
       customerService.create(customer);
       mailService.sendWelcomeMail(customer);
       return CUSTOMER_CREATED;
    }

    // getter / setter for customer
    ...
}
```

Rule #2: Inject Business not Infrastructure

```
// JSF UI Controller
@javax.enterprise.context.RequestScoped
@javax.inject.Named("createCustomerController")
public class CreateCustomerController {

    // inject CustomerService and MailService via @Inject

    @Inject @Current
    private CallCenterAgent currentCallCenterAgent;

    private Customer customer;

    public String createCustomer() {
        customer.setAuditInformation(currentCallCenterAgent);
        customerService.create(customer);
        mailService.sendWelcomeMail(customer);
        return CUSTOMER_CREATED;
    }

    // getter / setter for customer
    ...
}
```

Rule #2: Inject Business not Infrastructure

```
// Authentication Controller
@SessionScoped
@Named("authenticationController")
public class AuthenticationController implements Serializable {

    private CallCenterAgent loggedInCca;

    public String authenticate() {...}

    @Produces @Current @RequestScoped
    public CallCenterAgent getLoggedInCca() {
        return loggedInCca;
    }

    ...
}
```

@Inject @Current CallCenterAgent

Rule #2: Inject Business not Infrastructure

```
package de.openknowldege.qualifier

import ...

// self-made qualifier to indicate current instance of something

@Qualifier
@Target({TYPE, METHOD, PARAMETER, FIELD})
@Retention(RUNTIME)
public @interface Current{


}
```

Rule #2: Inject Business not Infrastructure

```
// JSF UI Controller
@javax.enterprise.context.RequestScoped
@javax.inject.Named("createCustomerController")
public class CreateCustomerController {

    // inject CustomerService and MailService via @Inject

    @Inject @Current
    private CallCenterAgent currentCallCenterAgent;

    private Customer customer;

    public String createCustomer() {
        customer.setAuditInformation(currentCallCenterAgent);
        customerService.create(customer);
        mailService.sendWelcomeMail(customer);
        return CUSTOMER_CREATED;
    }

    // getter / setter for customer
    ...
}
```

Rule #2: Inject Business not Infrastructure

```
// JSF UI Controller
@javax.enterprise.context.RequestScoped
@javax.inject.Named("createCustomerController")
public class CreateCustomerController {

    // inject CustomerService and MailService via @Inject

    @Inject @Current // still „bounded" to controller layer?
    private CallCenterAgent currentCallCenterAgent;

    private Customer customer;

    public String createCustomer() {
        customer.setAuditInformation(currentCallCenterAgent);
        customerService.create(customer);
        mailService.sendWelcomeMail(customer);
        return CUSTOMER_CREATED;
    }

    // getter / setter for customer
    ...
}
```

Rule #2: Inject Business not Infrastructure

```
// JSF UI Controller
@javax.enterprise.context.RequestScoped
@javax.inject.Named("createCustomerController")
public class CreateCustomerController {

    // inject CustomerService and MailService via @Inject

    @Inject @Current // still „bounded" to controller layer? NO!
    private CallCenterAgent currentCallCenterAgent;

    private Customer customer;

    public String createCustomer() {
       customer.setAuditInformation(currentCallCenterAgent);
       customerService.create(customer);
       mailService.sendWelcomeMail(customer);
       return CUSTOMER_CREATED;
    }

    // getter / setter for customer
    ...
}
```

Rule #2: Inject Business not Infrastructure

```
// Customer Service EJB
@Stateless
public class CustomerService {

    @Inject @Current
    private CallCenterAgent currentCallCenterAgent;

    @PersistenceContext
    private EntityManager em;

    // transactional by default
    public void createCustomer(Customer customer) {
       customer.setAuditInformation(currentCallCenterAgent);
       em.persist(customer);
    }

    ...
}
```

Rule #2: Inject Business not Infrastructure

```
<html ...>
  <h:body>
    <h:form>

    Vorname: <h:inputText
        value="#{createCustomerController.customer.firstname}"/>
    Name:    <h:inputText
        value="#{createCustomerController.customer.lastname}"/>


    </h:form>
  </h:body>
</html>
```

Aber was ist mit der View?

Rule #2: Inject Business not Infrastructure

```
<html ...>
  <h:body>
    <h:form>

    Vorname: <h:inputText
        value="#{customerToCreate.firstname}"/>
    Name:    <h:inputText
        value="#{customerToCreate.lastname}"/>


    </h:form>
  </h:body>
</html>
```

Fachlichkeit auch in der View?

Rule #2: Inject Business not Infrastructure

```
// JSF UI Controller
@javax.enterprise.context.RequestScoped
@javax.inject.Named(“createCustomerController“)
public class CreateCustomerController {

    @Inject
    private CustomerService customerService;

    private Customer customer;

    // getter for customer
    @javax.inject.Produces
    @javax.inject.Named(“customerToCreate“)
    public Customer getCustomer {
       return customer;
    }
    ...
}
```

#{customerToCreate … }

Rule #2: Inject Business not Infrastructure

```
// EJB based Service
@Named("shoppingCartService")
@ApplicationScoped
public class ShoppingCartService {

     ...

    // call from JSF via #{shoppingCartService.add(...)}
    public void add(
        ShoppingCart shoppingCart,
        Article article) {

       ...

    }
    ...
}
```

BTW: Service Access via EL

Rule #2: Inject Business not Infrastructure

```
// JPA Repository
public class CustomerRepository {

     @Inject
     EntityManager em;

     // CDI producer method for „current“ Customer
     @Produces
     @Current
     @Named(“currentCustomer“)
     public Customer findCustomer (
        @Parameter(“customerId“) Integer customerId) {
        return em.find(Customer.class, customerId)
     }
     ...
}
```

BTW: Repository als Producer

Always follow the Rules …

Rule #3: Business drives TX not Technology

```
// Customer Service EJB
@Stateless
public class CustomerService {

    @Inject @Current
    private CallCenterAgent currentCallCenterAgent;

    @PersistenceContext
    private EntityManager em;

    // transactional by default
    public void createCustomer(Customer customer) {
       customer.setAuditInformation(currentCallCenterAgent);
       em.persist(customer);
    }

    ...
}
```

Rule #3: Business drives TX not Technology

```
// Customer Service EJB - no EJB annotation required!
@ApplicationScoped @Stateless
public class CustomerService implements Serializable {

    @Inject @Current
    private CallCenterAgent currentCallCenterAgent;

    @PersistenceContext
    private EntityManager em;

    // transactional by default
    public void createCustomer(Customer customer) {
       customer.setAuditInformation(currentCallCenterAgent);
       em.persist(customer);
    }

    ...
}
```

Rule #3: Business drives TX not Technology

```
// Customer Service
@ApplicationScoped
public class CustomerService implements Serializable {

    @Inject @Current
    private CallCenterAgent currentCallCenterAgent;

    @PersistenceContext
    private EntityManager em;

    @Transactional
    public void createCustomer(Customer customer) {
       customer.setAuditInformation(currentCallCenterAgent);
       em.persist(customer);
    }

    ...
}
```

Java EE 7, DeltaSpike or „home brewn“

Rule #3: Business drives TX not Technology

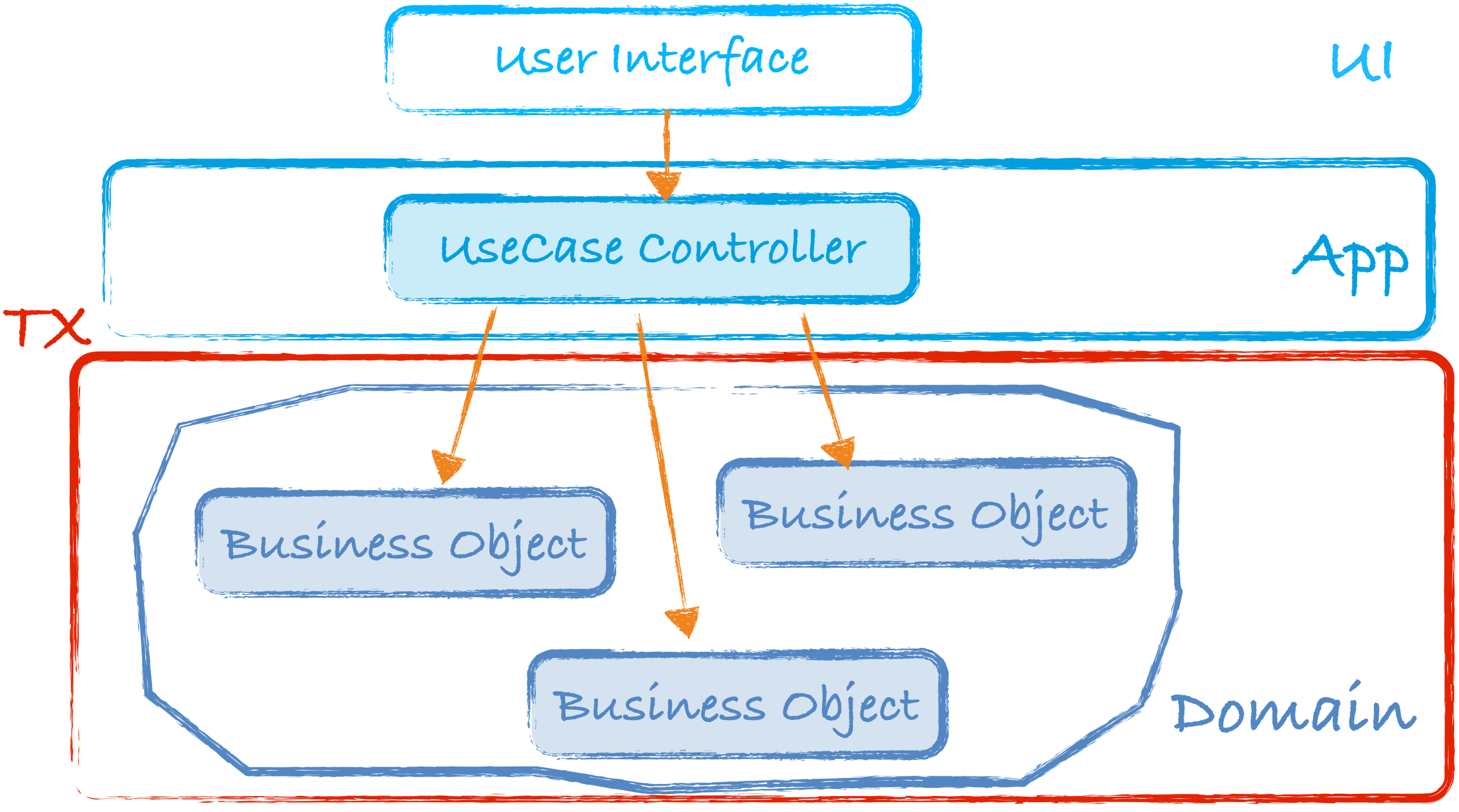
```
@Transactional
@Interceptor
public class TransactionAdvice {

  @Inject
  private UserTransaction utx;

  @AroundInvoke
  public Object applyTransaction(
            InvocationContext ic) throws Throwable {

    ...              // 1. implement utx.begin()
    ic.proceed();    // 2. call original method
    ...              // 3. implement utx.commit()

  }
```

*XML registration omitted

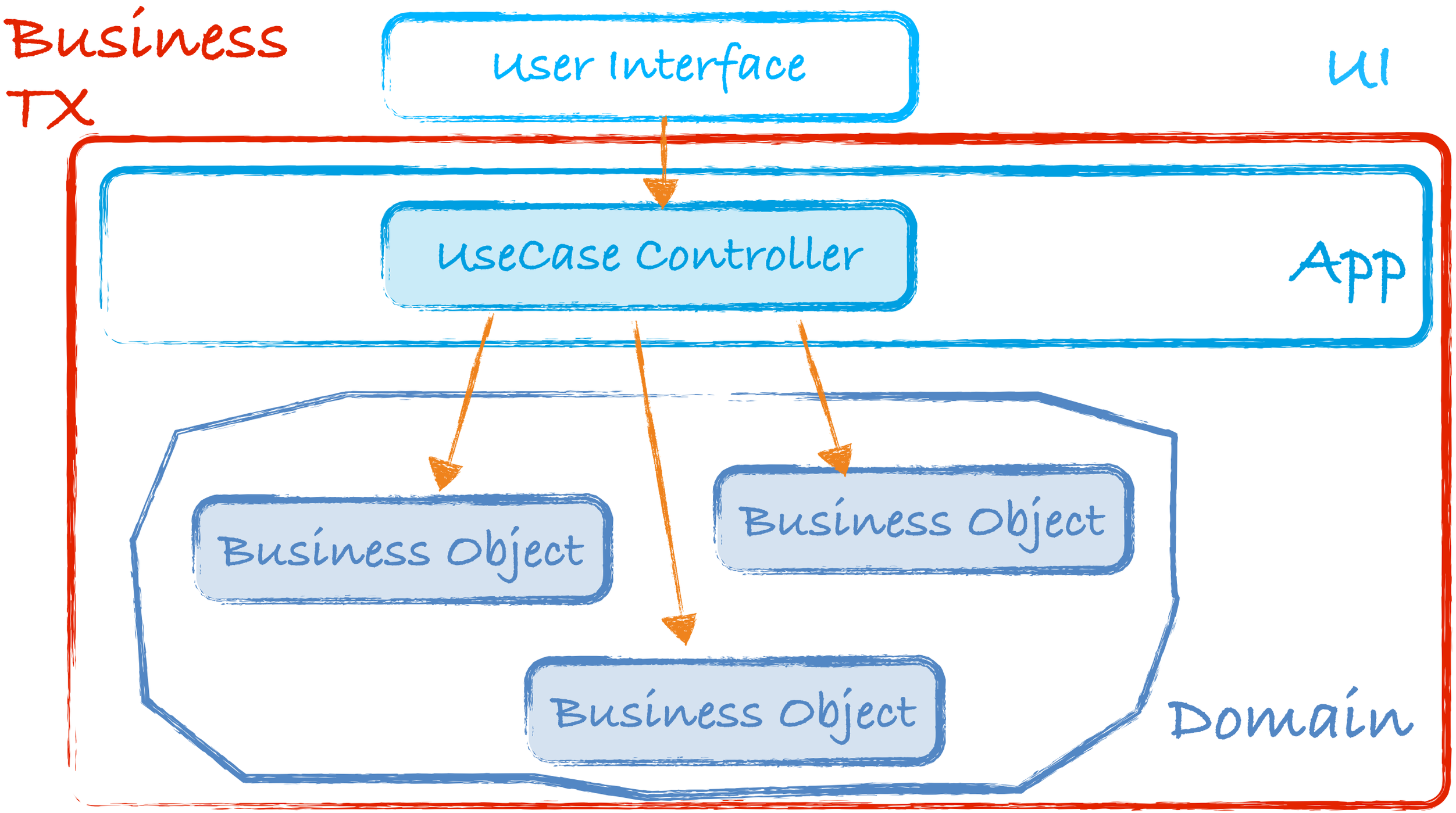
Rule #3: Business drives TX not Technology

```
// Customer Service - no annotation required!
@ApplicationScoped
public class CustomerService implements Serializable {

    @Inject @Current
    private CallCenterAgent currentCallCenterAgent;

    @PersistenceContext
    private EntityManager em;

    @Transactional
    public void createCustomer(Customer customer) {
        customer.setAuditInformation(currentCallCenterAgent);
        em.persist(customer);
    }

    ...
}
```

Rule #3: Business drives TX not Technology



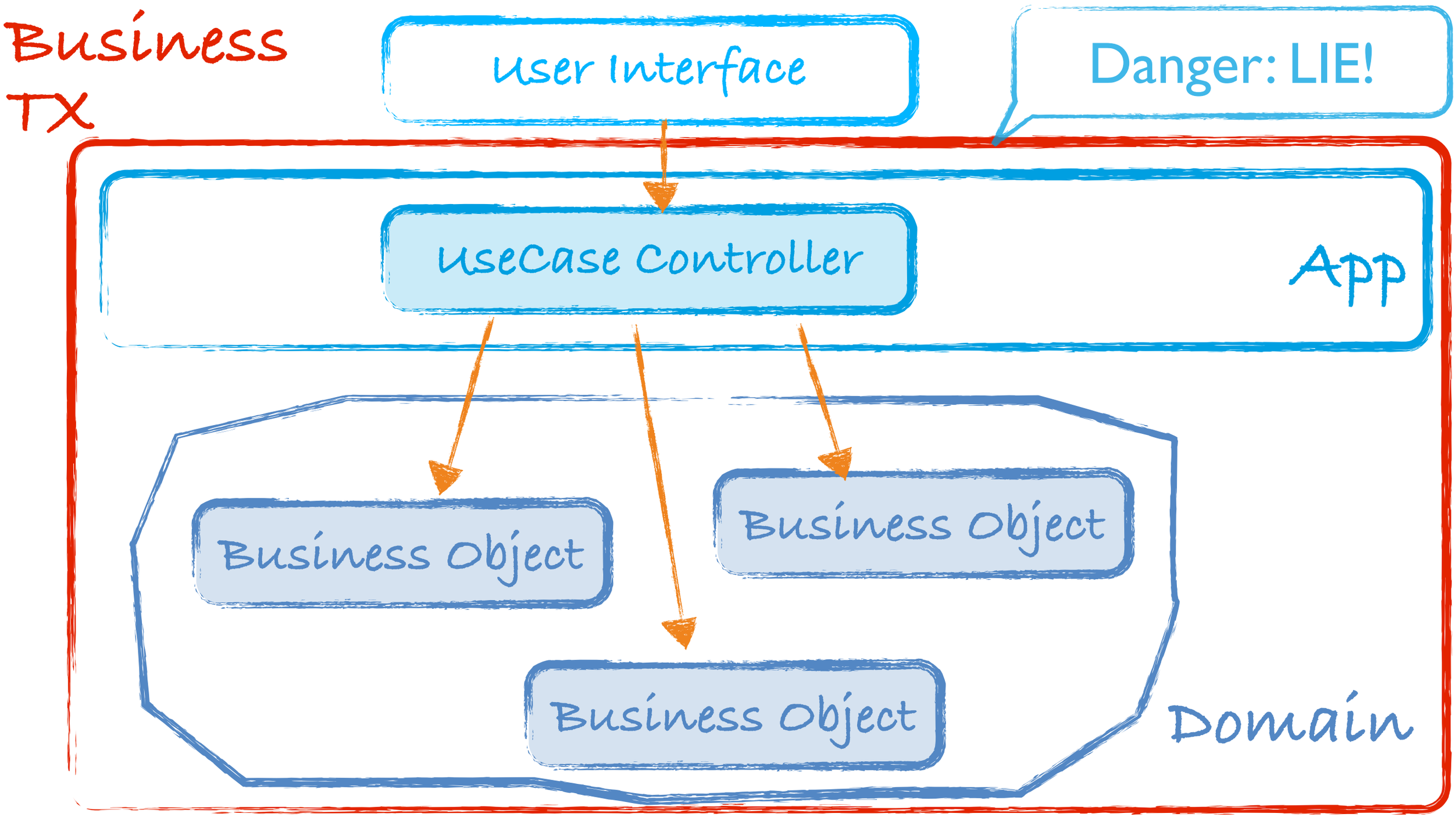
Rule #3: Business drives TX not Technology



Rule #3: Business drives TX not Technology

```
// JSF UI Controller
@javax.enterprise.context.RequestScoped
@javax.inject.Named("createCustomerController")
public class CreateCustomerController {

    @Inject
    private CustomerService customerService;

    private Customer customer;

    @Transactional
    public String createCustomer() {
       customerService.create(customer);
       ... // some additional use case "tx" related work
       return CUSTOMER_CREATED;
    }

    // getter / setter for customer
    ...
}
```

Rule #3: Business drives TX not Technology



Rule #3: Business drives TX not Technology

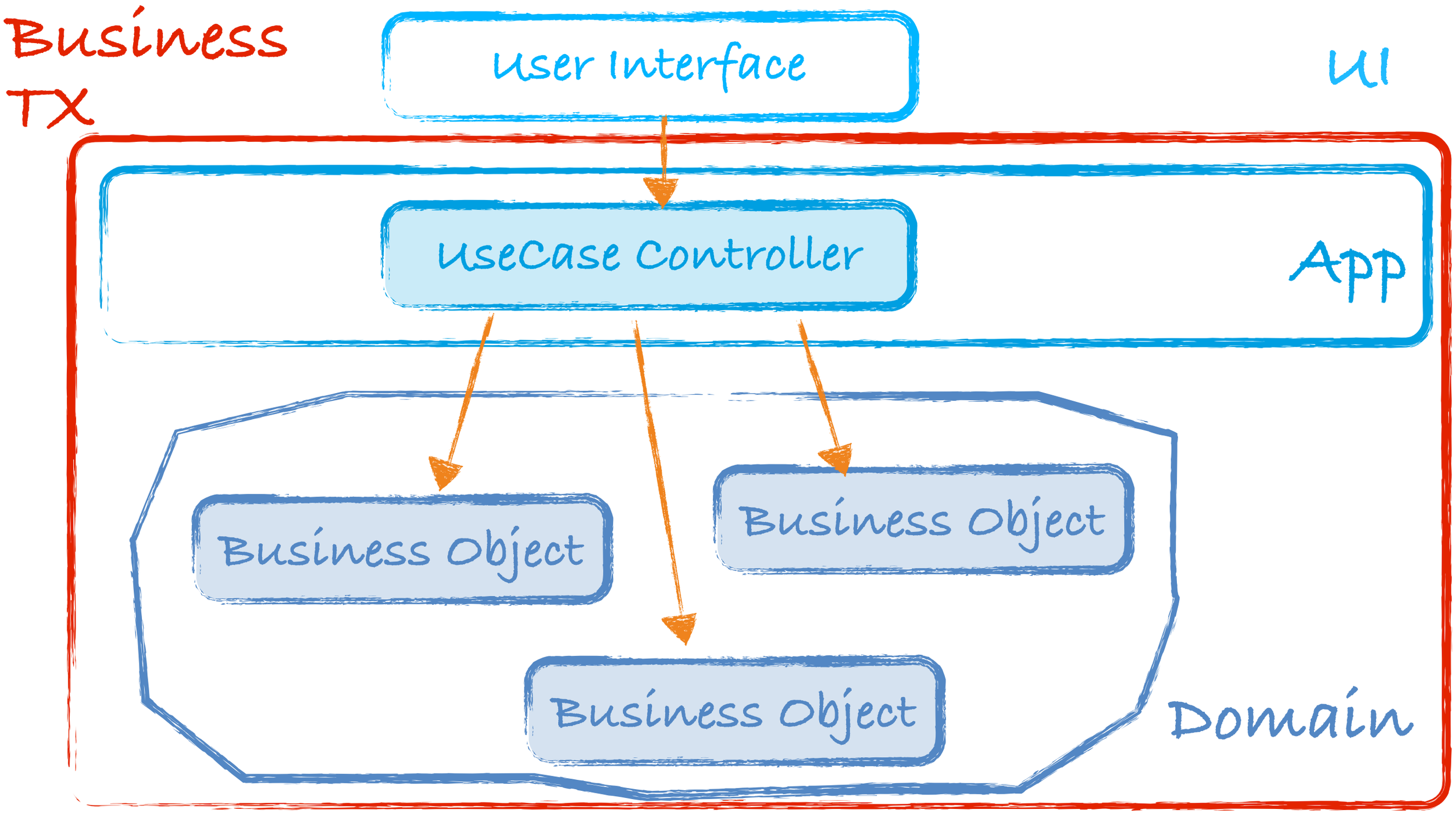
Lazy Initialization Problem besteht nach wie vor!

- EntityManager ist an Transaktion gebunden
- Probleme bei lesenden Zugriffen ohne TX

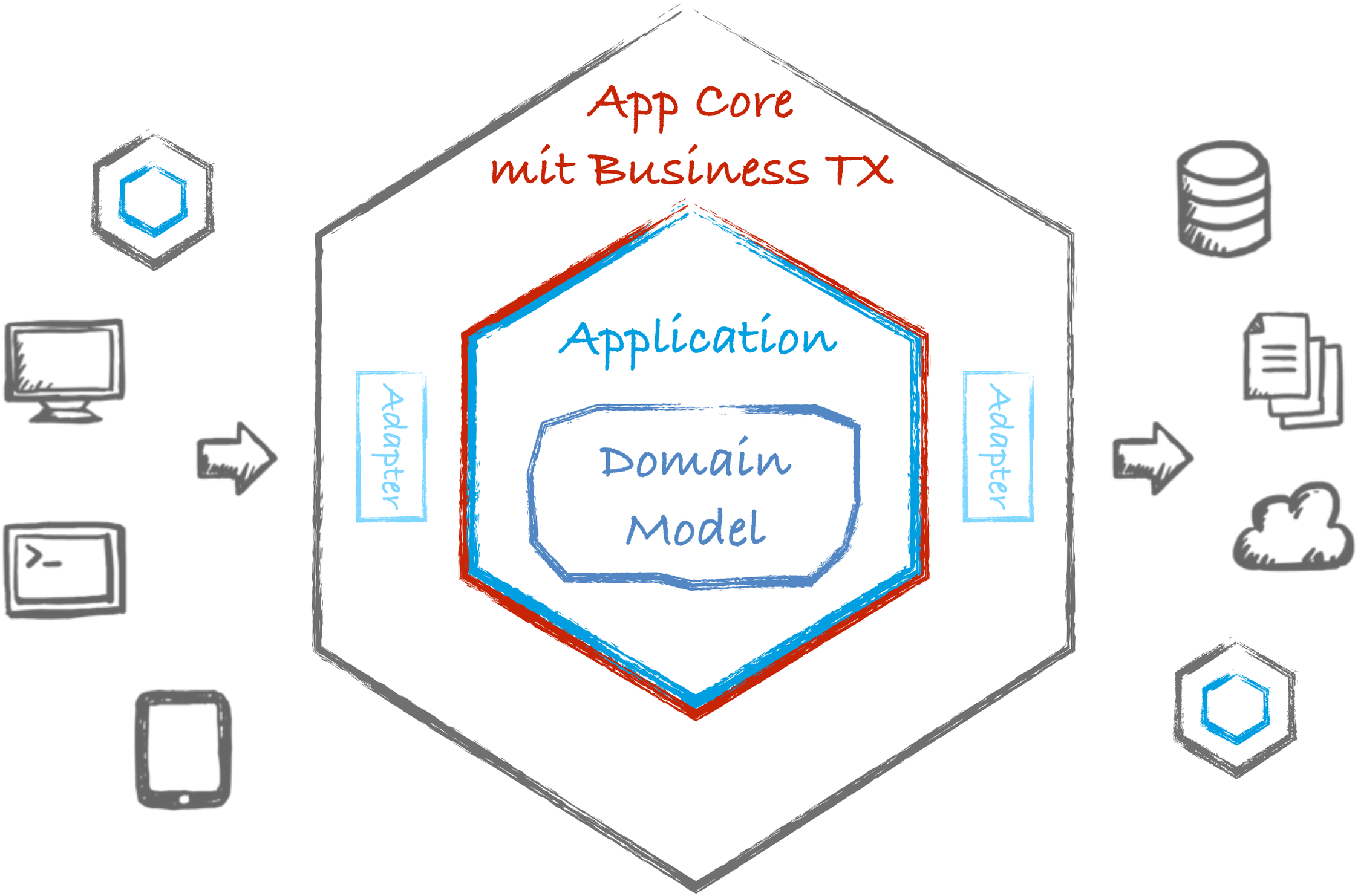
- Varianten
  - EXTENDED EM (@Stateful CM)
  - EXTENDED EM (@RequestScoped AM)
  - Fachliche Queries via EntityGraph (seit JPA 2.1)

Always follow the Rules …

Always follow the Rules …



Es wäre doch viel schöner, wenn ...



Erkenntnis des Tages - bitte mitnehmen!

„The Design is the Code, the Code is the Design.“

Erkenntnis des Tages - bitte mitnehmen!

„Muss ich denn jedes Mal den Code ändern, wenn sich die Fachlichkeit ändert?“

Erkenntnis des Tages - bitte mitnehmen!

„Zur Hölle, ja!

Wann denn

bitte sonst?“

Not your Father's

Java EE

@**mobile**Larson
@_**open**Knowledge

Lars **Röwekamp** | CIO New Technologies