EAPLI

Princípios de Design OO: Domínio

Paulo Gandra de Sousa
pag@isep.ipp.pt

Questões comuns

- A que classe atribuir uma dada responsabilidade?
- Como organizar as responsabilidades do sistema?
- Quem deve ter responsabilidade de coordenar a interação de caso de uso?
- Quem deve ter a responsabilidade de representar e implementar a lógica de negócio?
- Como gerir o ciclo de vida de um objeto?
  - Criar um objeto?
  - persistir objetos?
- Como proteger o código para modificação?

Topic	Principles and patterns
Which class should a responsibility be assigned to?	Information Expert Tell, don't ask Single Responsibility Principle Interface Segregation Principle Intention Revealing Interfaces
How to organize the system's responsibilities?	
How to model the domain?	
How to handle an object's lifecycle?	
How to prepare the code for modification?	

Quem deve ter a responsabilidade sobre a lógica de negócio?

Anemic Domain Model

http://martinfowler.com/bliki/AnemicDomainModel.html

Onde vou escolher as classes a criar?

Modelo de domínio

Table 1.4. Analyzing the Best Model for the Business

Which is better for the business? *Though the second and third statements are similar, how should the code be designed?*	
Possible Viewpoints	**Resulting Code**

"Who cares? Just code it up."
Um, not even close.
	`patient.setShotType(ShotTypes.TYPE_FLU);`
	`patient.setDose(dose);`
	`patient.setNurse(nurse);`

"We give flu shots to patients." Better, but misses some important concepts.	`patient.giveFluShot();`
"Nurses administer flu vaccines to patients in standard doses." This seems like what we'd like to run with at this time, at least until we learn more.	`Vaccine vaccine = vaccines.standardAdultFluDose();` `nurse.administerFluVaccine(patient, vaccine);`

One rule

Persistence Ignorance

Objects are allways in a valid state

An object cannot be constructed neither modified in a way that it does not hold its internal consistency.

Entities

DDD

- Objects in the real world which we would like to track its **identity**

- Example:
  - Person
  - Product
  - Sale

Entity: example

```
Class Product{
      public Product(String sku, Money price) {…}

      public ProductID getProductID() { … }

      public boolean equals(Object other) {
         if (other==this) return true;
         if (!(other instanceof Product)) return false;
         return (this.getProductID() ==
(Product)other.getProductID());
      }
}
```

What to use as identity?

- Domain identity, e.g.,
  - NIF
  - Order number
  - Student number

Carefull

- Entity (DDD) ≠ Entity (ER)

- Be carefull about database ids
  - Persistence ignorance
  - Surrogate keys are for referencial integrity not for (domain) identification
  - ORM tools <identity> mapping is NOT a domain identity

DDD Entity as a JPA managed class

```
@Entity
Class Product{
      // database ID
      @ID
      @GeneratedValue(strategy=GenerationType.AUTO)
      private Long id;

      // domain ID
      public ProductID getProductID() { … }
      private void setProductID(ProductID ref) { … }
      …
}
```

When to assign the identity?

- User enters the identity
  - Possibly from a list of known values from another system
- The domain layer assigns an identity
- The persistence layer assigns an identity
  - E.g., sequence number tables

Value Objects

DDD

- Problema
  - Alguns objetos interessam pelo valor dos seus atributos e não pela sua identidade, ex., Cor
  - Servem para descrever ou quantificar uma entidade
- Solução
  - Criar objetos **imutáveis** que apenas são identificados pela igualdade dos seus atributos e não necessitam de ter identidade

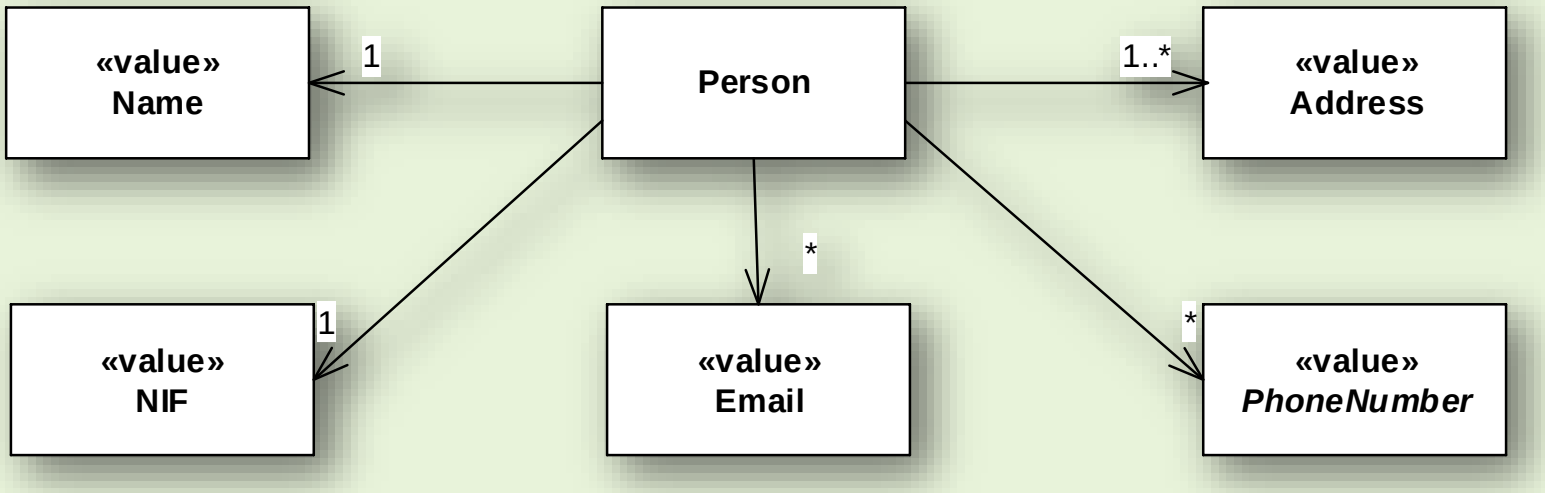
Value object: example

```
Class Color {
     public Color(int r, int g, int b) {…}
     public Color(String RAL) {…}

     public float redPercentage() {…}
     public float greenPercentage() {…}
     public float bluePercentage() {…}
     public boolean isPrimaryColor() {…}

     // imutable; creates a new object
     public Color combinedWith(Color other) {…}

     // equality by value
     public boolean equals(Object other) {…}
}
```

The domain, SRP and Value Objects

Primitive types are not the best option to represent domain concepts!

Favour imutability of your objects.



DDD Value Objects as JPA components

```
@Embeddable
class Color {
      private int red;
      private int green;
      private int blue;
      …
}


@Entity
Class Car {
      …
      private Color color;
      …
}
```

DDD Value Objects as JPA managed classes

```
@Entity
class Color {
        // database ID not to be exposed to domain
        @Id
        @GeneratedValue(strategy=GenerationType.AUTO)
        private int id;

        // domain values
        private int red;
        private int green;
        private int blue;
        …

        // avoid instantiation
        private Color() {...}

        //factory method
        static Color fromRGB(int r, int g, int b) { … }
}


@Entity
Class Car {
        …
        @OneToOne
        private Color color;
        …
}
```

Factory method hides the lookup/write to the DB if necessary.

Domain Layer API

All methods of a domain object should handle domain entities and value objects only; no primitive types.

```
void setName(String name)
```

vs.

```
void setName(Name aName)
```

Provide a convenience valueOf() method to convert from primitives read from the outside (e.g., UI, DB)

```
Class Name {
      public static Name valueOf(String name) {…}
}
```

Service

DDD

- Problem:
  - Some business operations are not naturally placed in a certain domain object
- Solution:
  - Create a service object that handles only that operation and coordinates the necessary domain objects.

Service: example

DDD

- Money transfer between two accounts:

```
Account.transferTo(  Account other,
                     Money amount
                     )
```

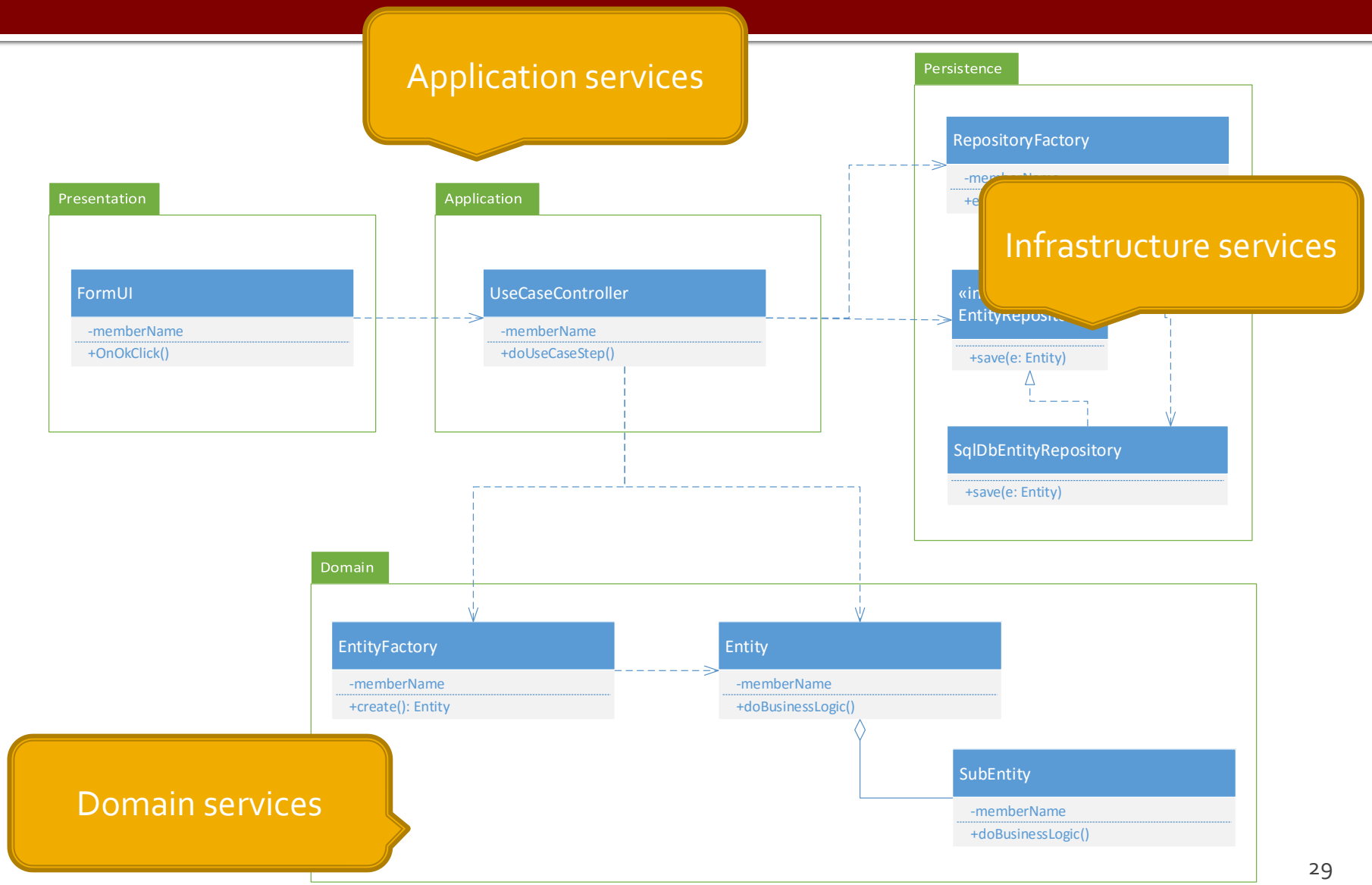
vs.

```
TransferService.transfer(  Money amount,
                           Account from,
                           Account to
                           )
```

Domain, Application and Infrastructure Services

- Domain
  - Coordinate domain activities
- Infrastructure
  - Provide infrastructure functionality hiding the details (and decoupling) from the domain
  - E.g., persistence, email
- Application
  - The interface to the domain layer
  - May have transactional control and access control

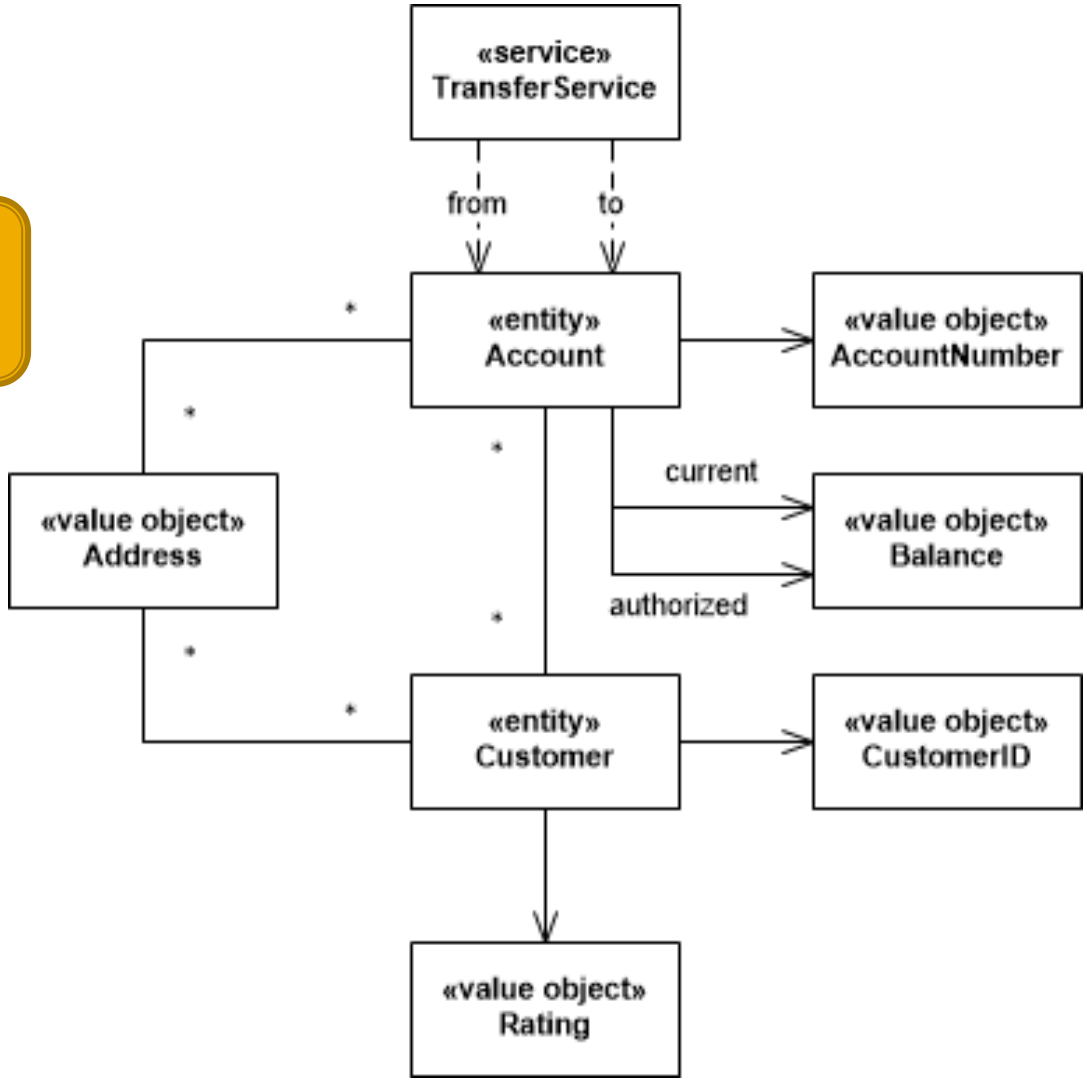
Sterotypical architecture



Entities, Value Objects and Services, is this enough?

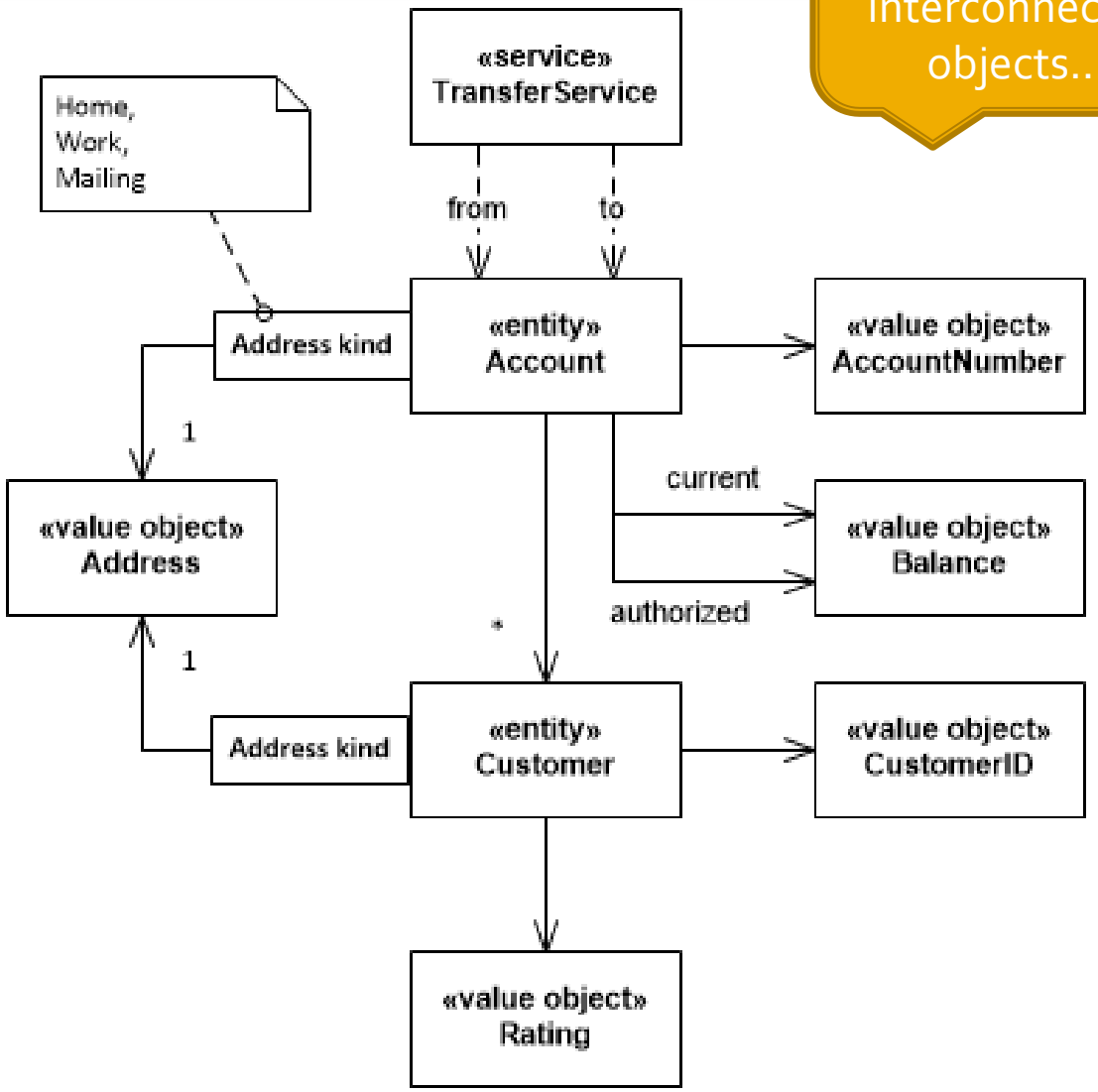
An Example

N-to-M relationships are hard



A pragmatic design

- Remove unnecessary associations
- Force traversal direction of bidirectional associations
- Reduce cardinality by qualification of the association



we are still left with a tangle of interconnected objects...

Aggregate

DDD

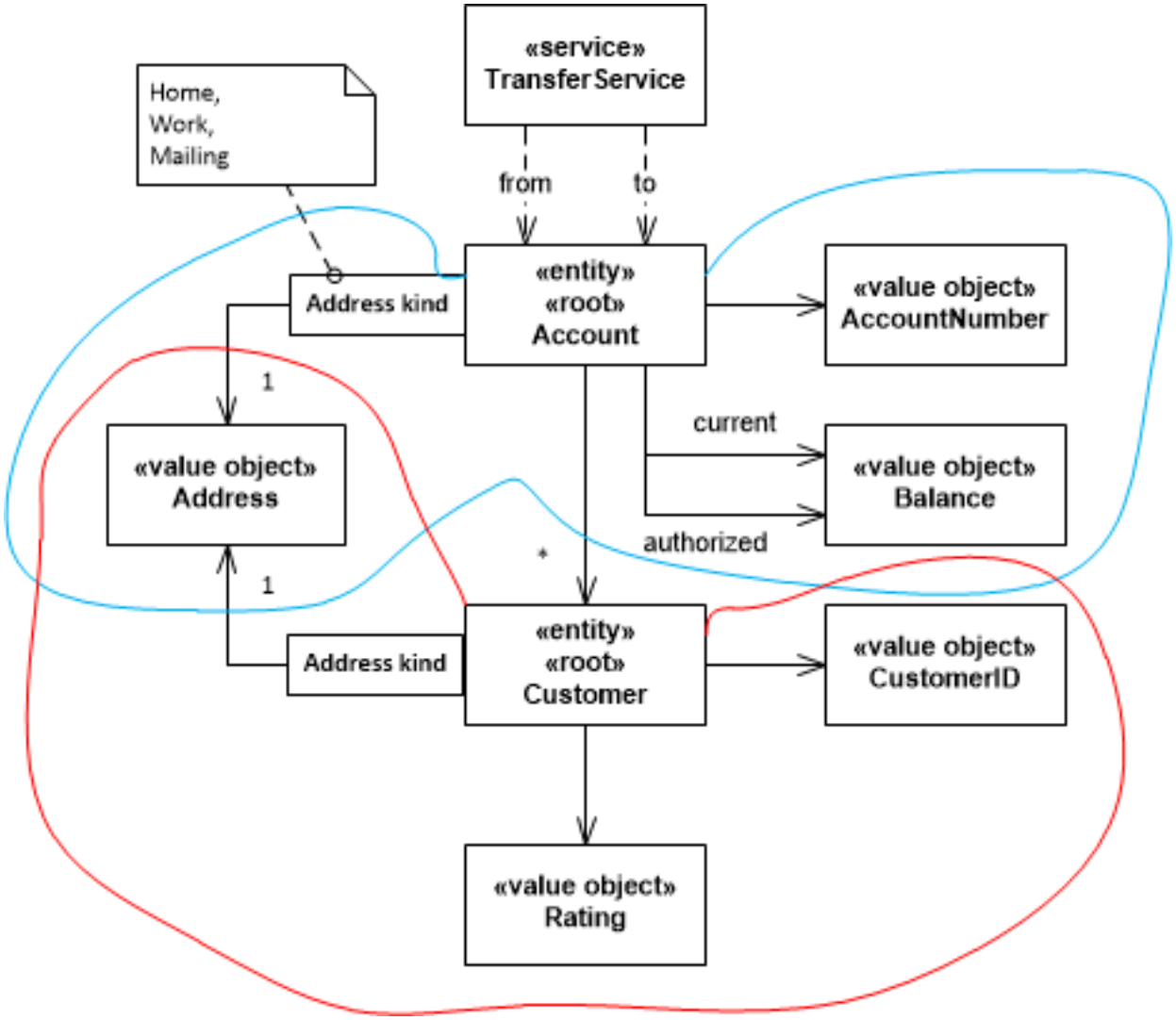
- Some objects are closely related together and we need to control the scope of data changes so that invariants are enforced

- **Therefore**
  - Keep related objects with frequent changes bundled in an aggregate
  - control access to the "inner" objects thru one single "root" object

A more pragmatic design

Legend:

- Account aggregate
- Customer aggregate



Address is a value object so it can be freely shared among several aggregates

Aggregate boundaries

- Efficient aggregate design is hard
- The model must be practical

- Are movements part of the Account aggregate?

- Entities, Value objects and Aggregates are about **things**
- Services are about **operations**
- But, **what** happens in the system is also important.

Domain Event

DDD

- Some things that happen **in the domain** are worth noting.
- **Therefore**
  - Model **activity in the domain** as a series of discrete events represented as a domain object.

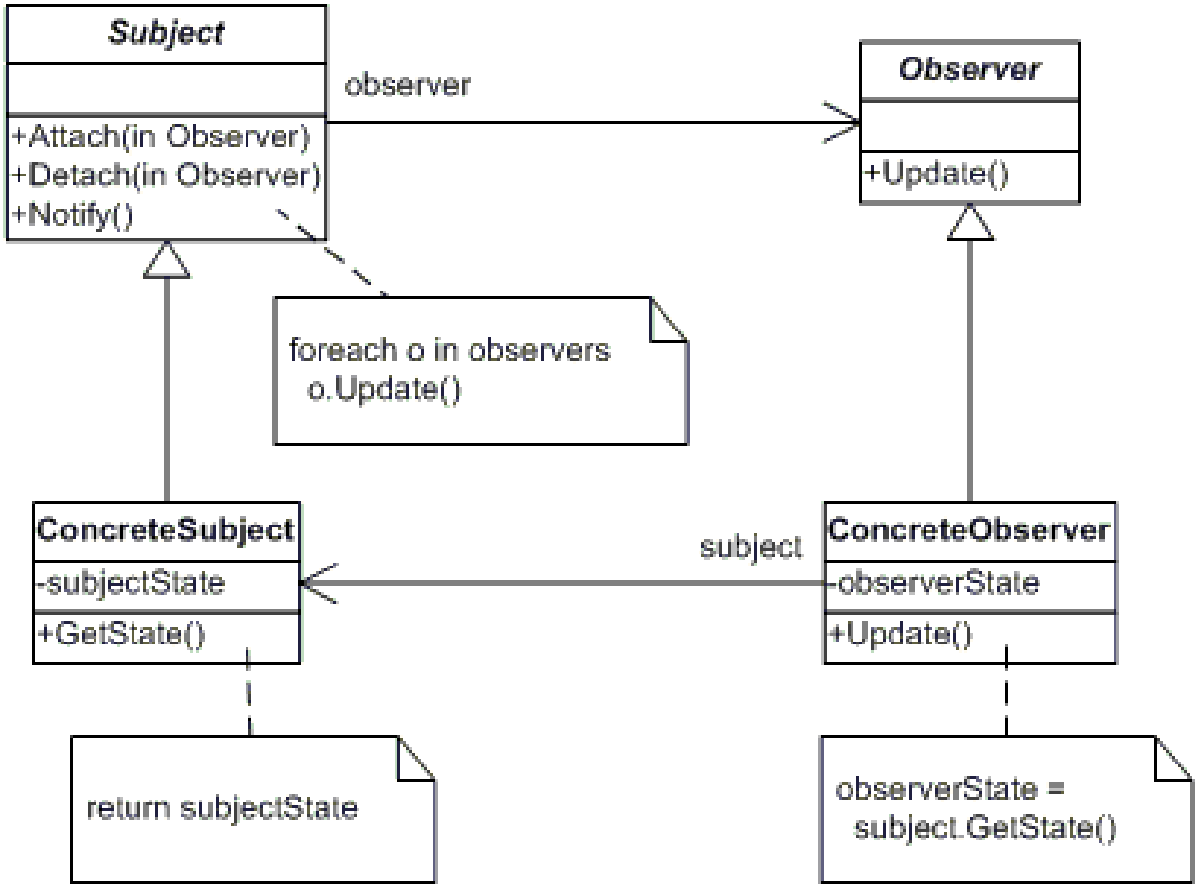
Observer

GoF

- A domain event might be of interest to some object.
- **Therefore**
  - Make the interested object an observer of the event's issuing party.

Observer

GoF



fonte: Design Patterns: Elements of Reusable Object-Oriented Software

Suporte na plataforma Java

- Classe **`Observable`**
  - As classes que "publicam" devem ser derivadas desta classe que implementa o comportamento standard para adicionar vistas e notificar todos os "subscritores"
  - **`void addObserver(Observer o)`**
  - **`protected void setChanged()`**
  - **`void notifyObservers(Object arg)`**
- Interface **`Observer`**
  - As classes "subscritor" devem implementar esta interface contendo um método que será invocado quando o "publicador" for actualizado
  - **`void update(Observable o, Object arg)`**

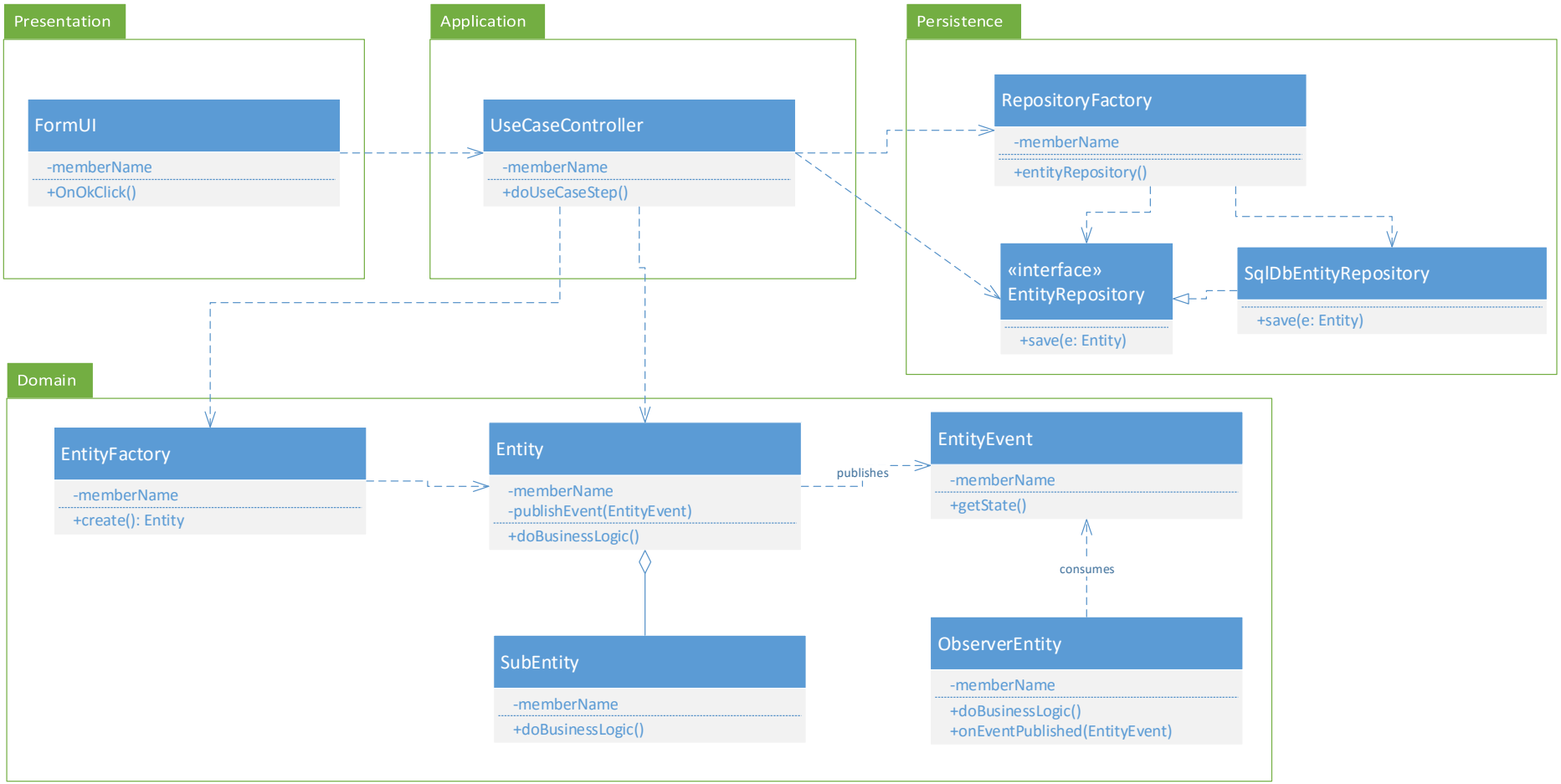
Exemplo (Java)

```
public class Subject extends Observable
{
    public void doSomeAction()
    {
            ...
            DomainEvent e = new SomethingHappened(123);
            setChanged();
            notifyObservers(e);
            ...
    }
}
```

Exemplo (Java)

```
public class MyObserver implements Observer
{
    public void update(Observable sender, Object param)
    {
            ...
    }

    public void doWhatever(Subject x)
    {
            ...
            // registar interesse nas notificações
            x.addObserver(this);
            ...
    }
}
```

Sterotypical architecture



Topic	Principles and patterns
Which class should a responsibility be assigned to?	Information Expert Tell, don't ask Single Responsibility Principle Interface Segregation Principle Intention Revealing Interfaces
How to organize the system's responsibilities?	
How to model the domain?	**Persistence Ignorance** **Entity** **Value Object** **Domain Service** **Aggregate** **Domain Event** **Observer**
How to handle an object's lifecycle?	
How to prepare the code for modification?	

Bibliografia

- Domain Driven Design (2004). Eric Evans
- Domain Driven Design reference (2011). Eric Evans. http://domainlanguage.com/ddd/patterns/DDD_Reference_2011-01-31.pdf
- Implementing Domain Driven Design (2013). Vernon Vaughn.
- Services in Domain Driven Design (2008) Jimmy Bogard. http://lostechies.com/jimmybogard/2008/08/21/services-in-domain-driven-design/
- Effective aggregate Design (2011) Vernon Vaughn. http://dddcommunity.org/library/vernon_2011/
- Design Patterns-Elements of Reusable Object-oriented Software, Gamma et al. (Gang of Four)