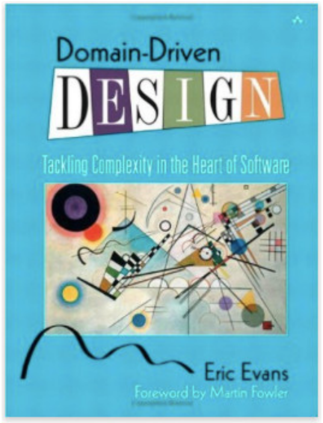
EDOM - Engenharia de Domínio
Mestrado em Engenharia Informática
Lecture 03.1
Modeling Core Concepts

Alexandre Bragança atb@isep.ipp.pt

Dep. de Engenharia Informática – ISEP

2016/2017

Acknowledgement


Domain-Driven
DESIGN
Tackling Complexity in the Heart of Software
Eric Evans
Foreword by Martin Fowler

"Domain-Driven Design: Tackling Complexity in the Heart of Software", Eric Evans, Addison Wesley, 2003

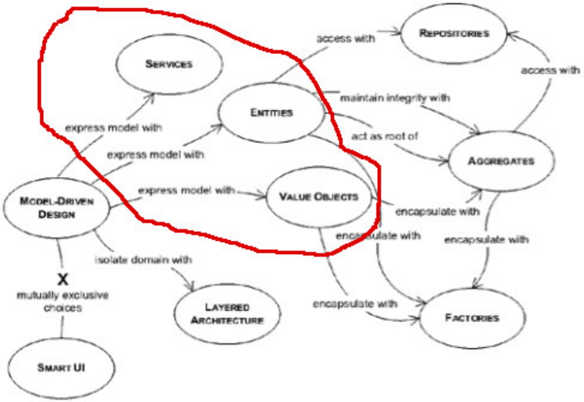
This lecture is based on the contents of this book.

There is a reference document explaining the principles of the book that you can find at: `https://domainlanguage.com/ddd/reference/PatternSummariesUnderCreativeCommons.doc`)

There is a reference document containing some updates that you can find at: `https://domainlanguage.com/ddd/reference/DDD_Reference_2015-03.pdf`)

We will be focusing on the section "Building Blocks of a Model-Driven Design".

Modeling Core Concepts



- We will discuss: **Entities**, **Value Objects**, **Services**... and **Associations**.

#1 Entities

Many objects are not fundamentally defined by their attributes, but rather by a thread of continuity and identity.

- They represent a thread of identity that runs through time and often across distinct representations.
- Sometimes such an object must be matched with another object even though attributes differ.
- An object must be distinguished from other objects even though they might have the same attributes.
- Mistaken identity can lead to data corruption.
- An object defined primarily by its identity is called an **ENTITY**.

Dois objectos que precisamos "separar" mesmo em ∞ mesmos atributos

#1 Entities: Example

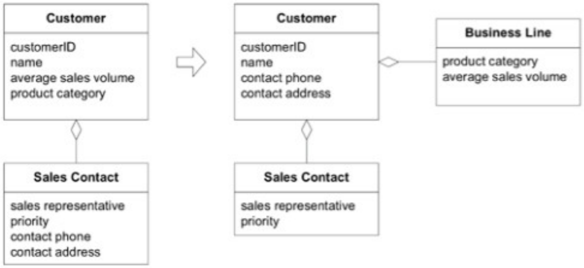
Consider transactions in a banking application

- Two deposits of the same amount to the same account on the same day are still **distinct transactions**, so they have identity and are **ENTITIES**.
- On the other hand, the amount attributes of those two transactions are probably instances of some money object. These values have no identity since there is no usefulness in distinguishing them.

Guidelines for Entities:

- When an object is distinguished by its identity keep the class definition simple and focused on **life cycle continuity and identity**.
- Define a means of distinguishing each object regardless of its form or history.
- Define an **operation that is guaranteed to produce a unique result for each object**, possibly by attaching a symbol that is guaranteed unique (Once this ID symbol is created and stored as an attribute of the Entity, it is designated as **immutable**).
- This means of identification may **come from the outside, or it may be an arbitrary identifier created by and for the system**.

#1 Entities: Modeling



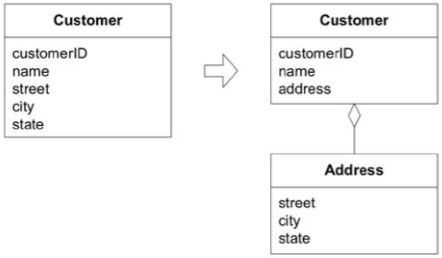
- Strip the Entity definition down to the most intrinsic characteristics, particularly those that **identify it or are commonly used to find or match it**.
- Add only the behavior that is essential to the concept and attributes that are required by that behavior.
- Beyond that, look to remove behavior and attributes into other objects associated with the core Entity.

#2 Value Objects

Many objects have no conceptual identity. These objects describe some characteristic of a thing.

- An object that represents a descriptive aspect of the domain with no conceptual identity is called a **VALUE OBJECT**.
- VALUE OBJECTS are instantiated to represent elements of the design that we care about only for *what* they are, no *who* or *which* they are. Basic examples are numbers or names. We do not care about their identity...
- A VALUE OBJECT can be an assemblage of other objects.
- VALUE objects can even reference ENTITIES. For instance, a map service can have a value object representing a route between two towns. This value object can have references to these two towns (which are entities).

#2 Value Objects: Example



Address is a Value Object...but it could be an Entity...

- When you care only about the attributes of an element of the model, classify it as a **Value Object**.
- Make it express the meaning of the attributes it conveys and give it related functionality.
- Treat the Value Object as **immutable**.
- Do not give it any identity and avoid the design complexities necessary to maintain Entities.
- **Note**: Associations may regard both Entities and Value Objects. However, it makes no sense to have bidirectional associations between two value objects! [1]

[1] Without identity, it is meaningless to say that an object points back to the same VALUE OBJECT that points to it. The most you could say is that it points to an object that is equal to the one pointing to it, but you would have to enforce that invariant somewhere.

#3 Services

In some cases, the clearest and most pragmatic design includes operations that do not conceptually belong to any object. Rather than force the issue, we can follow the natural contours of the problem space and include ***SERVICES*** *explicitly in the model.*

- Some concepts from the domain aren't natural to model as objects.
- Forcing the required domain functionality to be the responsibility of an Entity or Value Object either distorts the definition of a model-based object or adds meaningless artificial objects.
- **A Service is an operation offered as an interface that stands alone in the model, without encapsulating state, as Entities and Value Objects do.**
- A Service tends to be named for an activity, rather than an entity - **a verb** rather than a noun.

#3 Main Characteristics of a Service

A good Service has the three characteristics:

1. The operation relates to a domain concept that is not natural part of an Entity or Value Object.
2. The interface is defined in terms of other elements of the domain model.
3. The operation is stateless[2].

Regarding implementation:

- A "doer" object may be satisfactory as an implementation of a Service's interface.
- A simple **Singleton**[3] can be written easily to provide access.

[2]Statelessness here means that any client can use any instance of a particular service without regard to the instance's individual history.

[3]Gamma et al., 1995

#3 Service Example

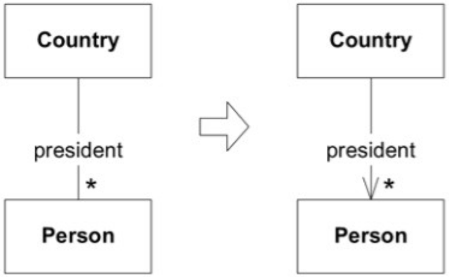
Partition Services into Layers

Application	*Funds Transfer App Service* - Digest input (such as an XML request). - Sends message to domain service for fulfillment. - Listens for confirmation. - Decides to send notification using infrastructure service.
Domain	*Funds Transfer Domain Service* - Interacts wit the necessary Account and Ledger objects, making appropriate debits and credits. - Supplies confirmation of result (transfer allowed or not, and so on).
Infrastructure	*Send Notification Service* - Sends e-mails, letters, and other communications as directed by the application.

#4 Associations: Avoid Many-to-Many Associations

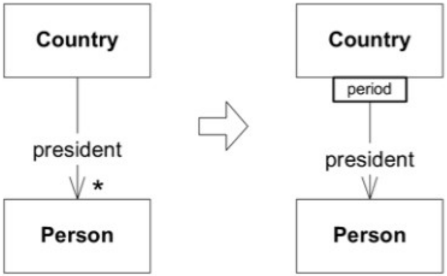
- *For every traversable association in the model, there is a mechanism in the software with the same properties.*
- For instance, a model that shows an association between a customer and a sales representative corresponds to two things:
    - On one hand, it abstracts a **relationship** developers deemed relevant between two real people,
    - On the other hand, it corresponds to an **object pointer between two Java objects**, or an **encapsulation of a database lookup**, or **some comparable implementation**.
- **Many-to-many** associations may exist but they complicate implementation. They also **communicate very little about the nature of the relationship**.
- There are ways of making the associations more tractable:
    - Imposing a traversal direction
    - Adding a qualifier, effectively reducing multiplicity
    - Eliminating nonessential associations

#4 Associations: Imposing a Traversal Direction



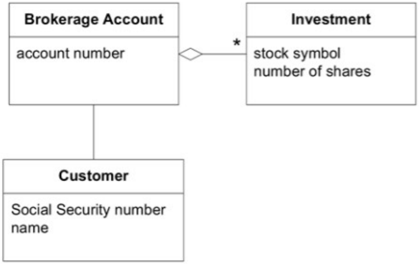
- **It is important to constrain relationships as much as possible**.
- A bidirectional association means that both objects can be understood only together.
- When application requirements do no call for traversal in both directions, adding a traversal direction reduces interdependence and simplifies design.
- In this example, the `Person` class becomes **independent** of the far less fundamental concept of `President`.

#4 Associations: Adding a Qualifier



- **More deep knowledge of the model can lead to qualified relationships**.
- In this example, a country has only one president at a time.
- This qualifier reduces the multiplicity to **one-to-one**, and explicitly embeds an **important rule in to the model**.
- If, by principle, we do this, then we also add value to the relationships that are really bidirectional.
- **When possible, eliminate all associations that are no required for the "solution".**

#4 Associations: Example: Associations in a Brokerage Account



```
public class BrokerageAccount {
  String accountNumber;
  Customer customer;
  Set investments;

  // Constructors, etc. omitted

  public Customer getCustomer() {
    return customer;
  }

  public Set getInvestments() {
    return investments;
  }
}
```

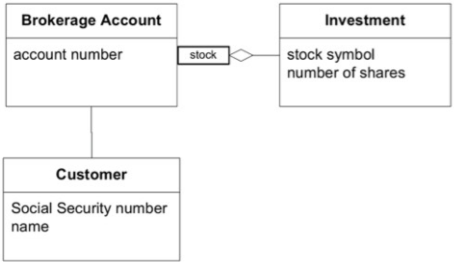
#4 Associations: Example: Associations in a Brokerage Account

- A possible implementation if the data is in a relational database...

```
public class BrokerageAccount {
  String accountNumber;
  String customerSocialSecurityNumber;

  // Omit constructors, etc.

  public Customer getCustomer() {
    String sqlQuery = "SELECT * FROM CUSTOMER WHERE"+
      "SS_NUMBER='"+customerSocialSecurityNumber+"'";
    return QueryService.findSingleCustomerFor(sqlQuery);
  }

  public Set getInvestments() {
    String sqlQuery = "SELECT * FROM INVESTMENT WHERE"+
      "BROKERAGE_ACCOUNT='"+accountNumber+"'";
    return QueryService.findInvestmentsFor(sqlQuery);
  }
}
```

#4 Associations: Example: Associations in a Brokerage Account

- If only one investment by stock...



```
public class BrokerageAccount {
  String accountNumber;
  Customer customer;
  Map investments;

  // Omitting constructors, etc.

  public Customer getCustomer() {
    return customer;
  }

  public Investment
        getInvestment(String
        stockSymbol) {
    return (Investment)investments.
      get(stockSymbol);
  }
}
```

#4 Associations: Example: Associations in a Brokerage Account

- A possible implementation if the data is in a relational database...

```java
public class BrokerageAccount {
  String accountNumber;
  String customerSocialSecurityNumber;

  //Omitting constructors, etc.

  public Customer getCustomer() {
    String sqlQuery = "SELECT * FROM CUSTOMER WHERE SS_NUMBER='"
      + customerSocialSecurityNumber + "'";
    return QueryService.findSingleCustomerFor(sqlQuery);
  }

  public Investment getInvestment(String stockSymbol) {
    String sqlQuery = "SELECT * FROM INVESTMENT "
      + "WHERE BROKERAGE_ACCOUNT='" + accountNumber +
      "'" + "AND STOCK_SYMBOL='" + stockSymbol +"'";
    return QueryService.findInvestmentFor(sqlQuery);
  }
}
```

Further readings

- http://blog.sapiensworks.com/topics/#domain-driven-design
- http://wiki.c2.com/?ValueObject
- https://dzone.com/refcardz/getting-started-domain-driven
- https://en.wikipedia.org/wiki/Domain-driven_design
- https://www.infoq.com/minibooks/domain-driven-design-quickly
- https://docs.microsoft.com/en-us/dotnet/standard/microservices-architecture/microservice-ddd-cqrs-patterns/domain-events-design-implementation