

The “Domain Model” Supporting Architecture



Dino Esposito

@despos | <http://software2cents.wordpress.com>

Implementing the Domain Layer

Classic Approach

Object-oriented Model

Domain Services

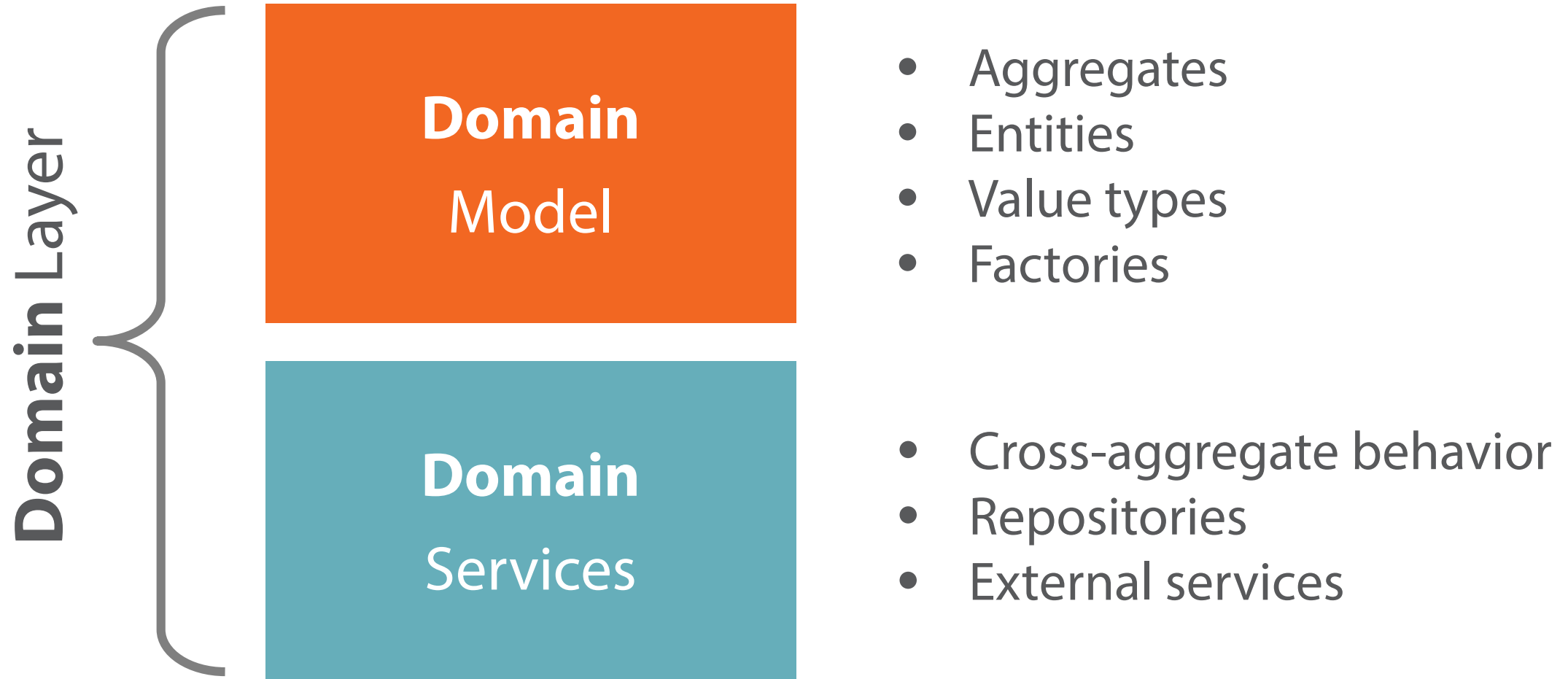
Key Points

**Object-oriented
Model for Domain**

Domain Services

**Events, Processes
and More ...**

Domain Model Supporting Architecture



Key DDD Misconceptions



Perceived simply as having an **object model** with some special characteristics

Database is merely part of the infrastructure and can be neglected

Ubiquitous Language is a guide to naming classes in the object model

Clearing Misconceptions

Just an object model

- Context mapping is paramount
- Modeling the domain through objects is just one of the possible options

Database agnostic

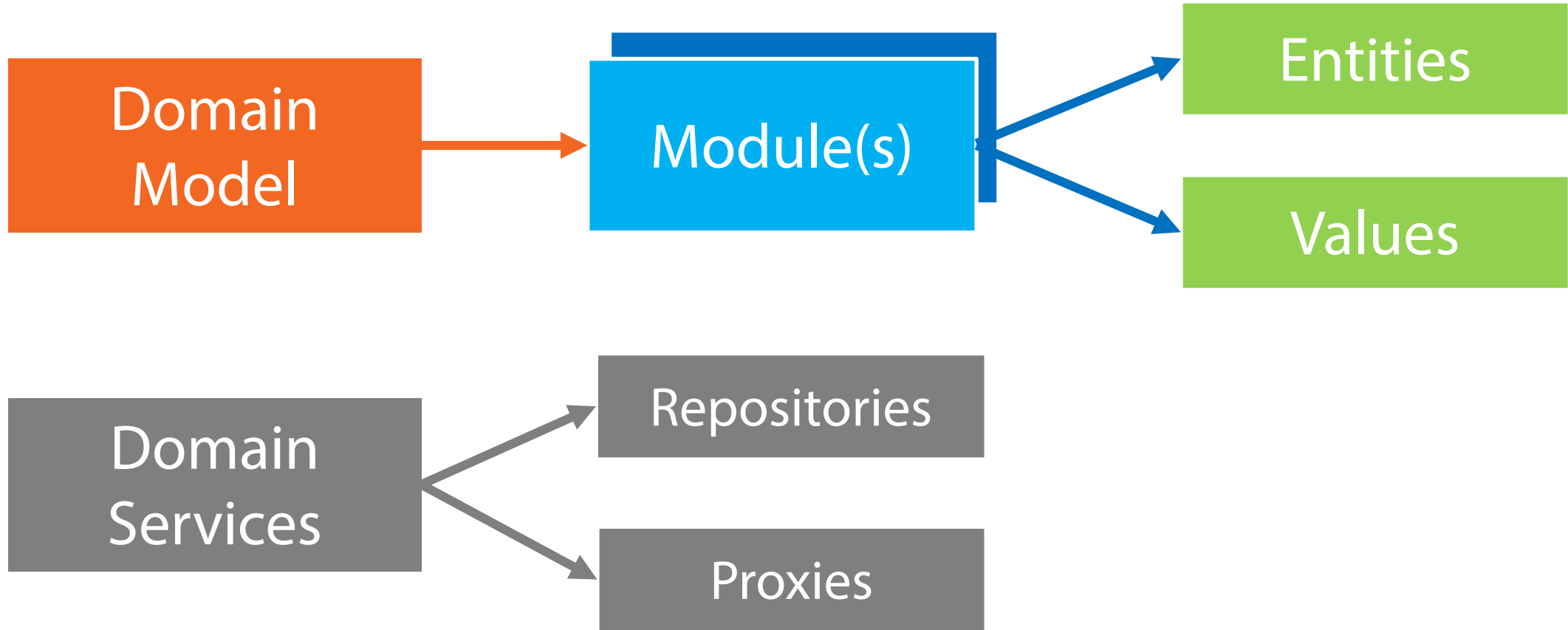
- The object model must be easy to persist
- Persistence, though, should not be the primary concern
- Primary concern is making sense of the business domain

Ubiquitous Language

- Understand the language to understand the business
- Keep language of business in sync with code

It's all about
behavior.

Domain Layer



Aspects of a Domain Model **Module**



Value Objects

Entities

Aggregates

DDD Value Types

Collection of individual values

Fully defined by its collection of attributes

Immutable

More precise and accurate than primitive types

Value Type

```
public int OutsideTemperature { get; set; }
```

```
public Temperature OutsideTemperature { get; set; }
```

- Constructor(s)
- Min and Max constants
- Ad hoc setters with built-in validation and compensation logic
- Ad hoc getters
- Ad hoc comparison
- Additional properties

DDD Entities

Need an identity

Uniqueness is important to the specific object

Typically made of data and behavior

Contain domain logic, but not persistence logic

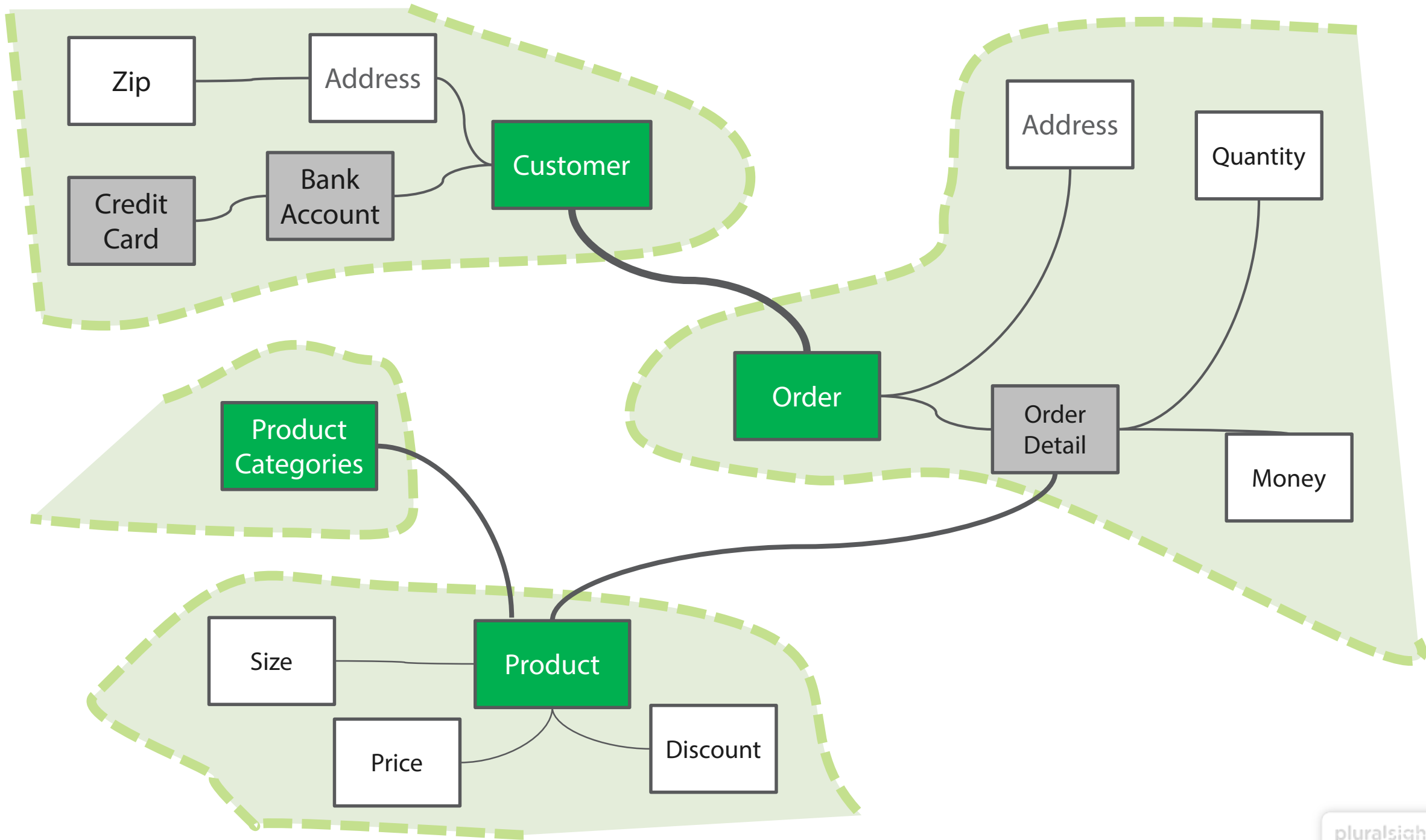
DDD Aggregates

A few individual entities constantly used and referenced together

Cluster of associated objects treated as one for data changes

Aggregate roots

Preserve transactional integrity



Persistence vs. Domain Model



Persistence Model

- Object-oriented model 1:1 with underlying relational data
- Reliable and familiar to most developers
- Doesn't include business logic (except perhaps validation)



Domain Model

- Object-oriented model for business logic
- Persistable model
- No persistence logic inside

Resistance to Change

The diagram is a comparison between two architectural approaches. On the left, under the heading 'Persistence', are two teal-colored chevron arrows pointing right. The top arrow contains the text 'Objects as plain data containers' and the bottom arrow contains 'Business logic belongs to other components'. On the right, under the heading 'Domain', are two orange-colored chevron arrows pointing left. The top arrow contains the text 'Data and behavior in the same objects' and the bottom arrow contains 'Business logic expressed as the combination of objects'. The arrows are arranged in a symmetrical fashion, facing each other.

Objects as plain
data containers

Business logic
belongs to other
components

Persistence

Data and
behavior in the
same objects

Business logic
expressed as the
combination of
objects

Domain

DEMO

What's Behavior?

The way in which one acts or conducts oneself, especially towards others

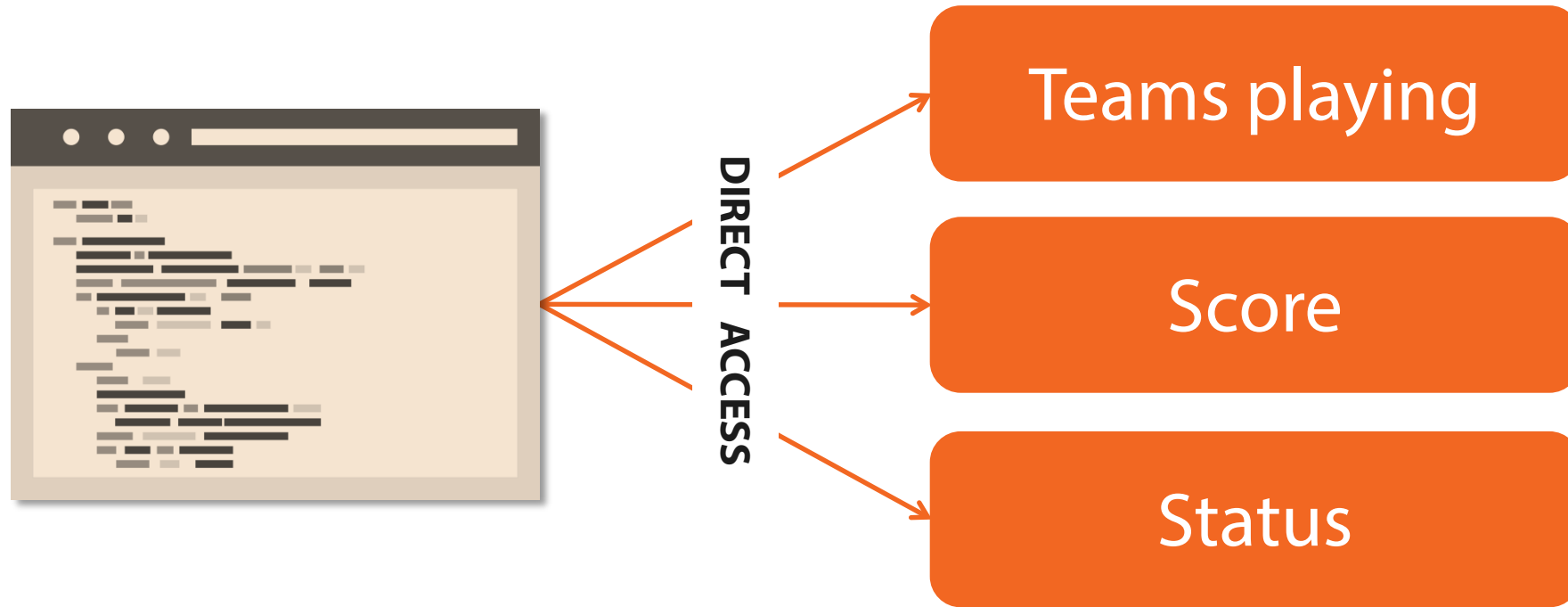


Methods that validate the state of the object

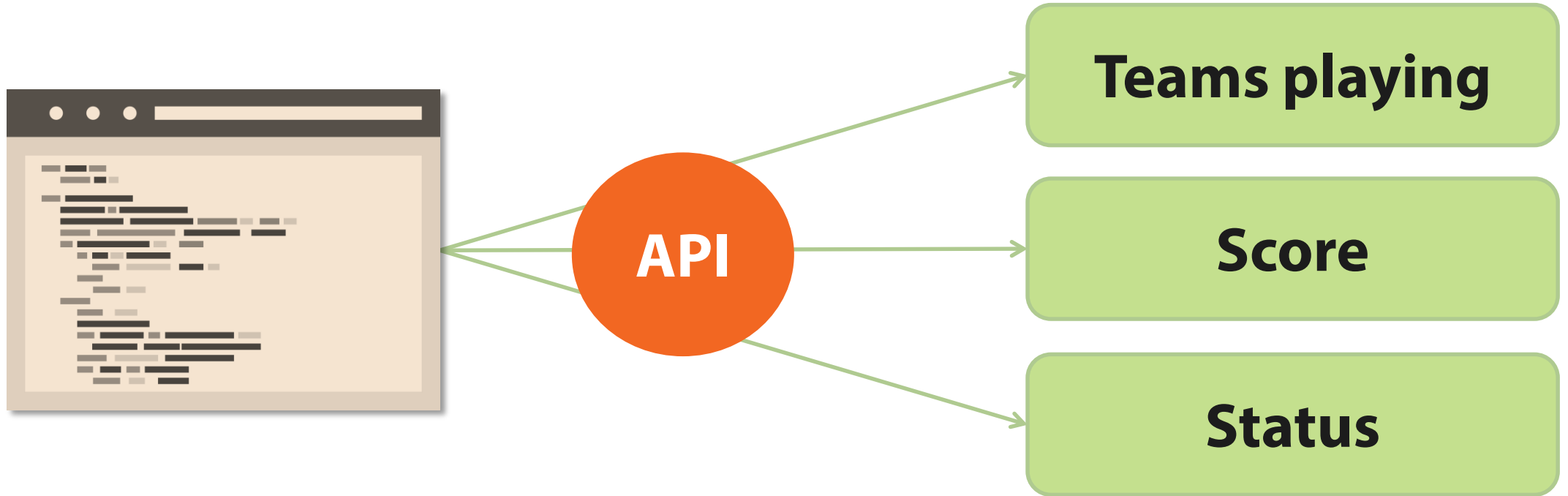
Methods that invoke business actions to perform on the object

Methods that express **business processes** involving the object

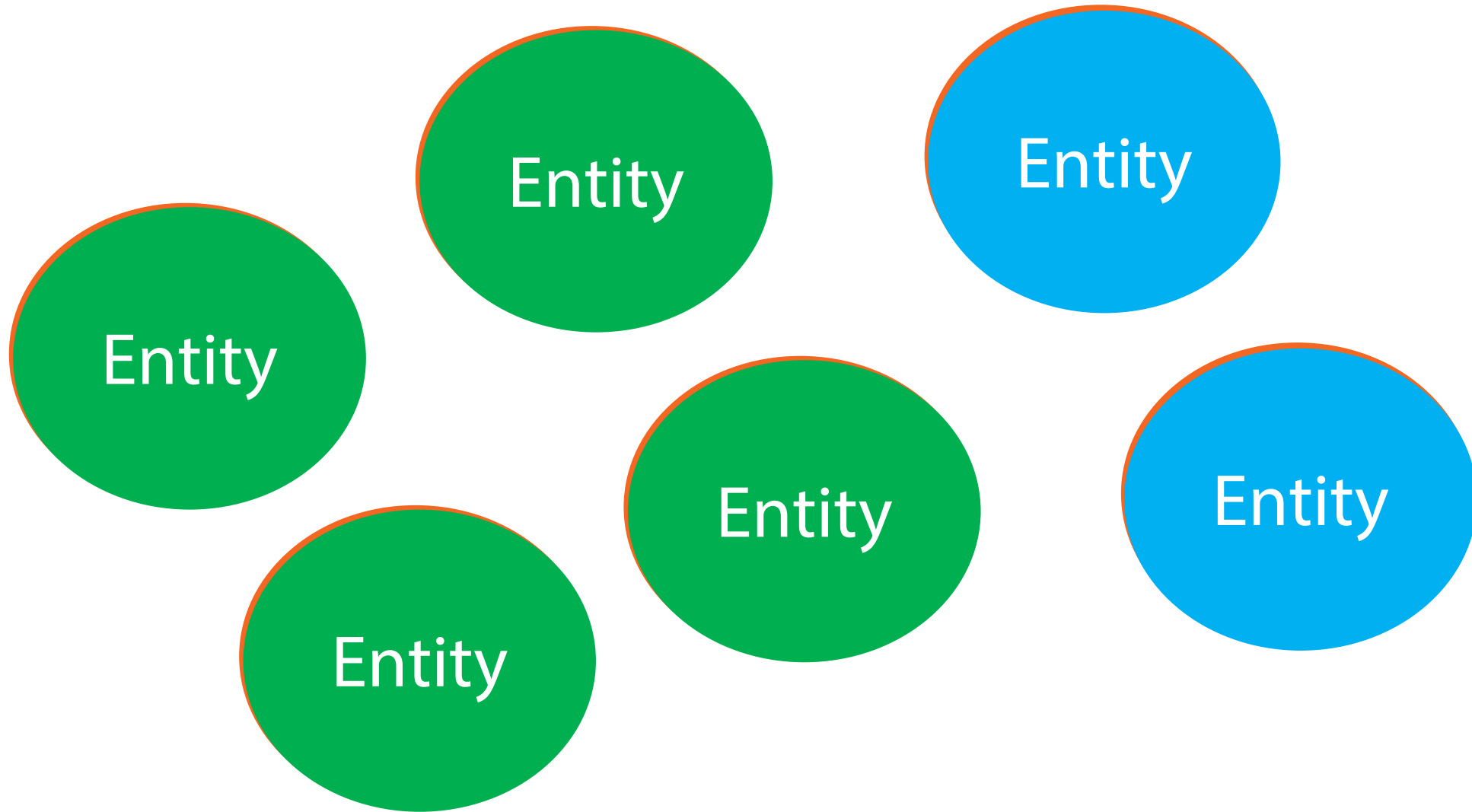
Sport Game

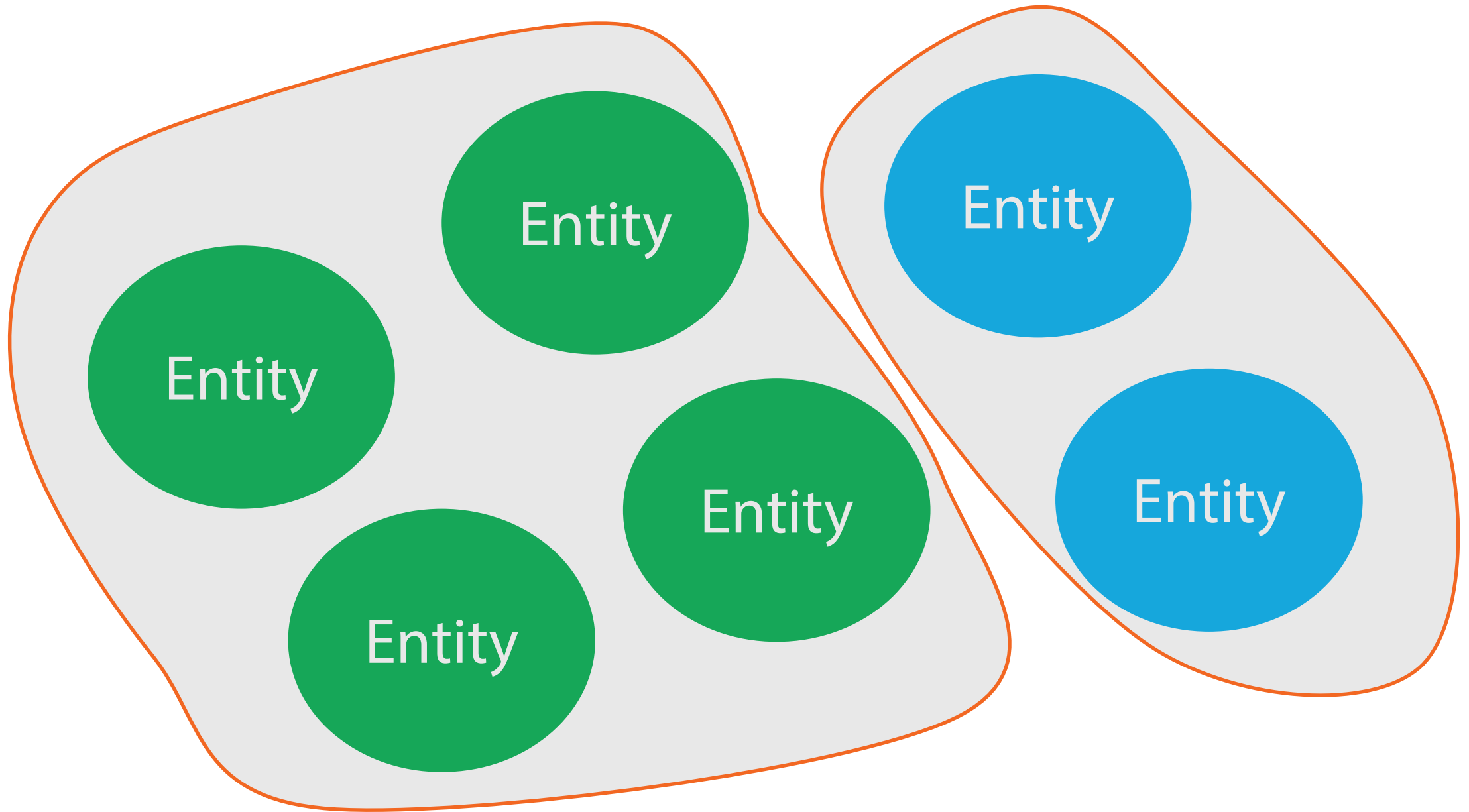


Sport Game



DEMO

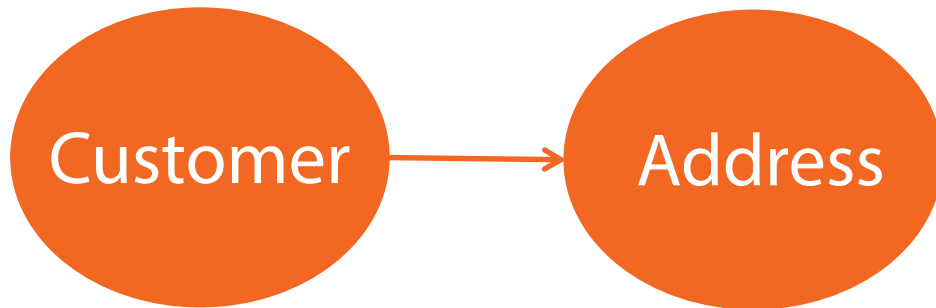




Facts of an Aggregate

Work with fewer objects and coarse grained and with fewer relationships

- Protect as much as possible the graph of entities from outsider access
- Ensure the state of child entities is always consistent
- Actual boundaries of aggregates are determined by business rules



Distinct aggregates



Aggregate

Common Responsibilities Associated with an Aggregate Root

- Ensure encapsulated objects are always in a consistent state
- Take care of persistence for all encapsulated objects
- Cascade updates and deletions through the encapsulated objects
- Access to encapsulated objects must always happen by navigation

One repository per aggregate

In terms of code, what does it **mean** to be an aggregate root?

DEMO

Domain Services



Implement the domain logic that doesn't belong to a particular aggregate and most likely span over multiple entities.

Domain Services



Coordinate the activity of aggregates and repositories with the purpose of implementing a business action.

Domain Services



May consume services from the infrastructure, such as when sending an email or a text message is necessary.

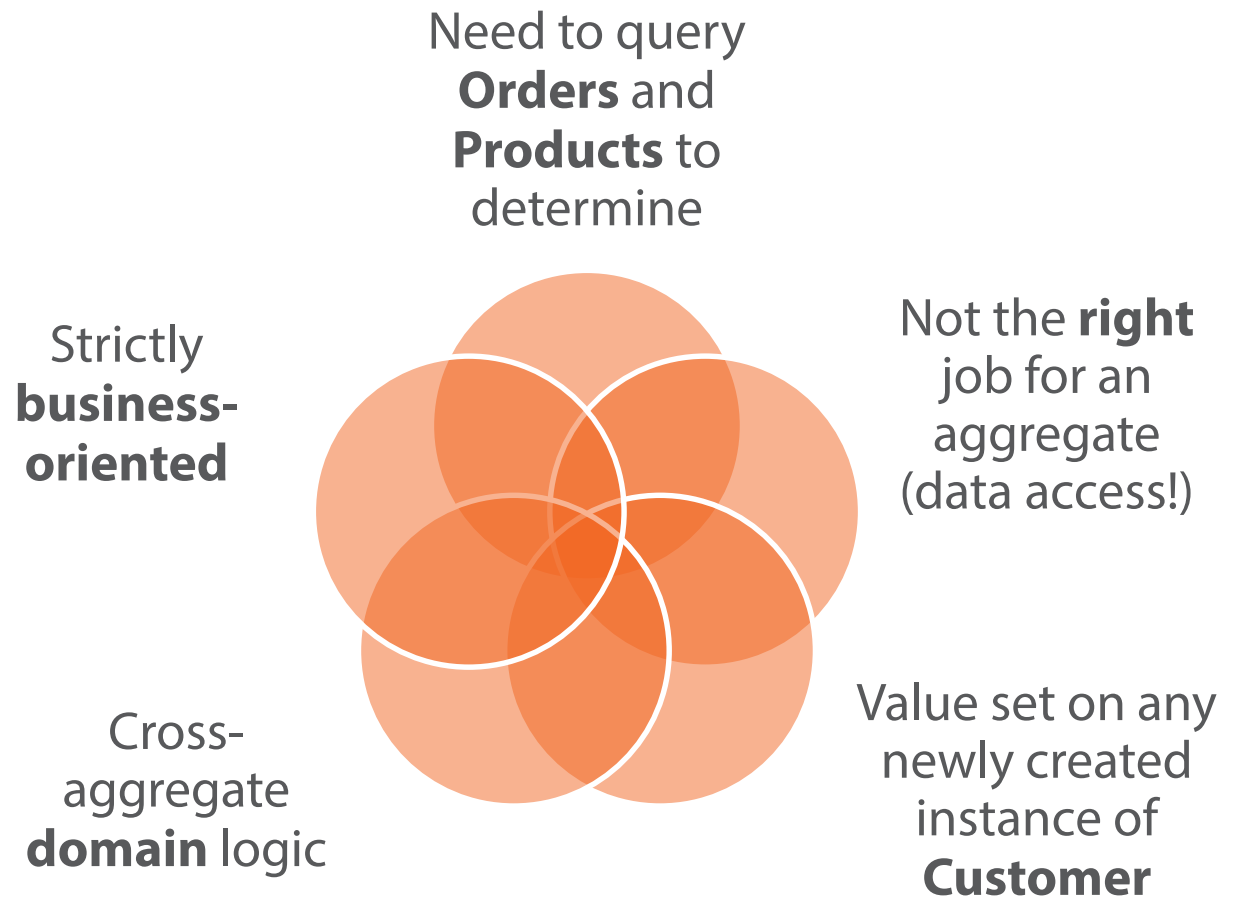
Actions in domain services come from **requirements** and are approved by **domain experts**.

Names used in domain services are strictly part of the ubiquitous language.

Domain Service Example

Determine whether a given customer reached the status of "gold" customer

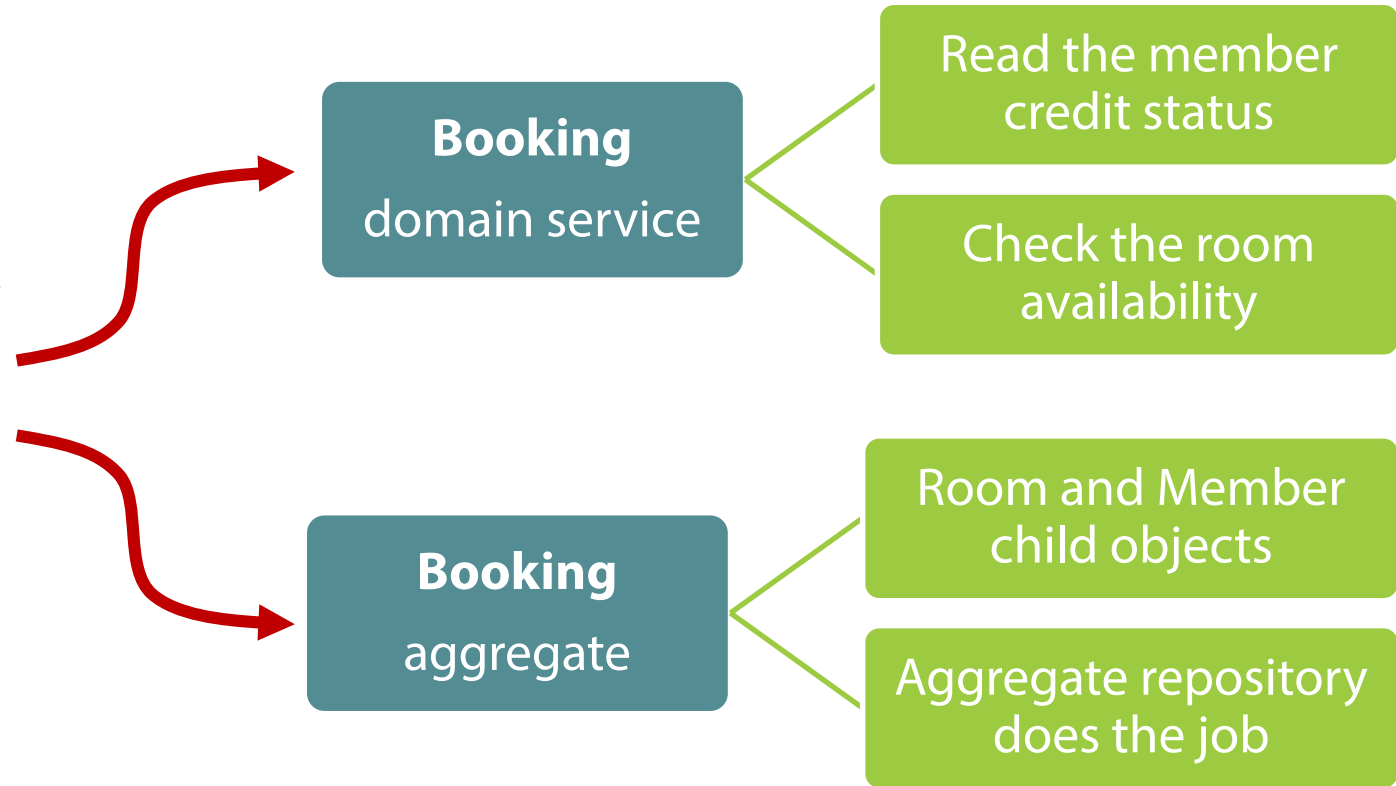
A customer earns the status of "**gold**" after she exceeds a given threshold of orders on a selected range of products.



Domain Service Example

Booking a meeting room

Booking requires **verifying** availability of the room and **processing** payment.



Repository

In DDD, a repository is just the class that handles **persistence** on behalf of entities and ideally aggregate roots.

Repositories

Most popular
type of domain
service

Take care of
persisting
aggregates

One repository
per aggregate
root

Assembly with repositories
has a direct dependency on
data stores.

A repository is where you
deal with connection strings
and use SQL commands.

There's Nearly No Wrong Way to Write a Repository Class

```
public interface IRepository<T> where T:IAggregateRoot
{
    // You can keep the interface a plain marker or
    // you can have a few common methods here.

    T Find (object id);
    bool Save (T item);
    bool Delete (T item);
}
```

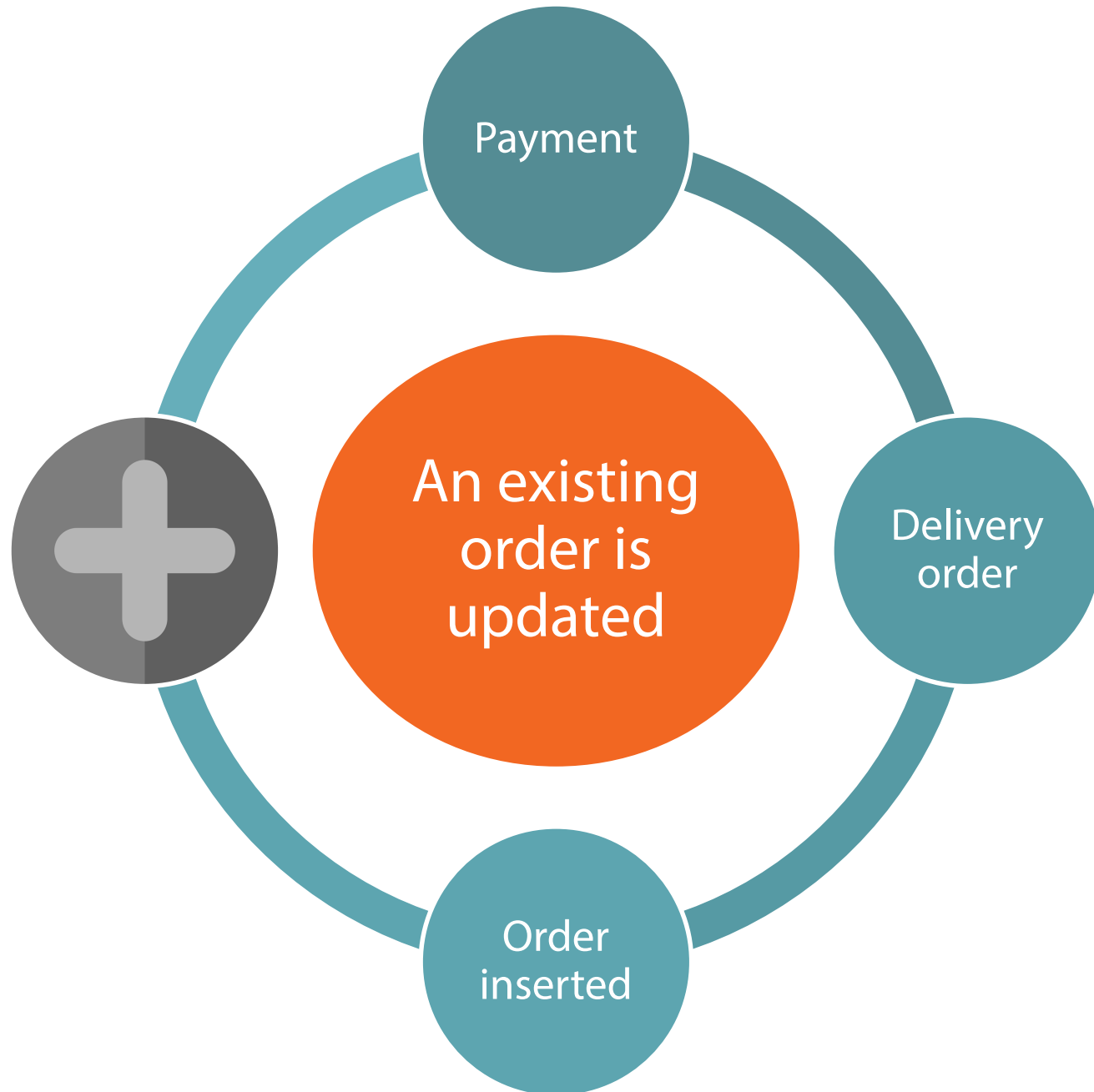
Why Should You Consider Events in a Domain Layer?

Not strictly necessary.

Just a more effective and resilient way to express the intricacy of some real-world business domains.

**Some task(s) to
perform upon
creation of the order**

Where would you
implement
such task(s)?



Option #1

```
/* Add any tasks to the method that processed the order */  
void Checkout(ShoppingCart cart)  
{  
    // Proceed through the necessary steps  
    ...  
    if (success)  
    {  
        // Execute task(s)  
        ...  
    }  
}
```

Option #1 – Facts

Not very expressive

- Monolithic code
- Future changes require to touch the service
- Risk of turning into rather convoluted code

Violations of ubiquitous language?

- The adverb “when” in requirements indicates what to do when a given “event” is observed

Option #2

Raise an event for domain-relevant facts



No need to have all handling code in one place



Raise of the event distinct from handling the event



Can handle the same event in multiple places

Option #2 – event

```
public class GoldMemberStatusReached : IDomainEvent
{
    public GoldMemberStatusReached(Customer customer)
    {
        Customer = customer;
    }

    public Customer Customer { get; set; }
}
```

Option #2 – raise

```
public void Checkout(ShoppingCart cart)
{
    // Proceed through the necessary steps
    ...

    if (success)
    {
        // Execute task(s)
        Bus.RaiseEvent(new GoldMemberStatusReached(customer));
    }
}
```

Option #2 – Facts

Bus as an "external" element

- Part of the infrastructure
- Notify listeners

More flexibility

- Can record events
- Can have any number of handlers

Events help significantly to coordinate actions within a workflow and to make use-case workflows a lot more resilient to changes.

Anemic Models

Anti-pattern because **"it takes behavior away from domain objects"**

Entities only made of properties

Required logic placed in service components

Services orchestrate the application logic

Is this **really** an anti-pattern
today?

Entity Framework and Code First

- **Define a model of classes. Is it representing the domain model?**
 - No, it's mostly representing the persistence model.
- **Add behavior to a persistence model**
 - May or may not keep the resulting model fluent enough to express the ubiquitous language and be friendly enough to be persisted via EF
- **Database is infrastructure, but also a constraint**
 - Domain model
 - Persistence model
 - Adapters

The model you end up with
when using **Code First** and
Entity Framework is a lot more
anemic than you may think.

If a developer **can** use an API the wrong way, **he will**.



**DOMAIN
model**

If a developer **can** use an API the wrong way, **he will**.

