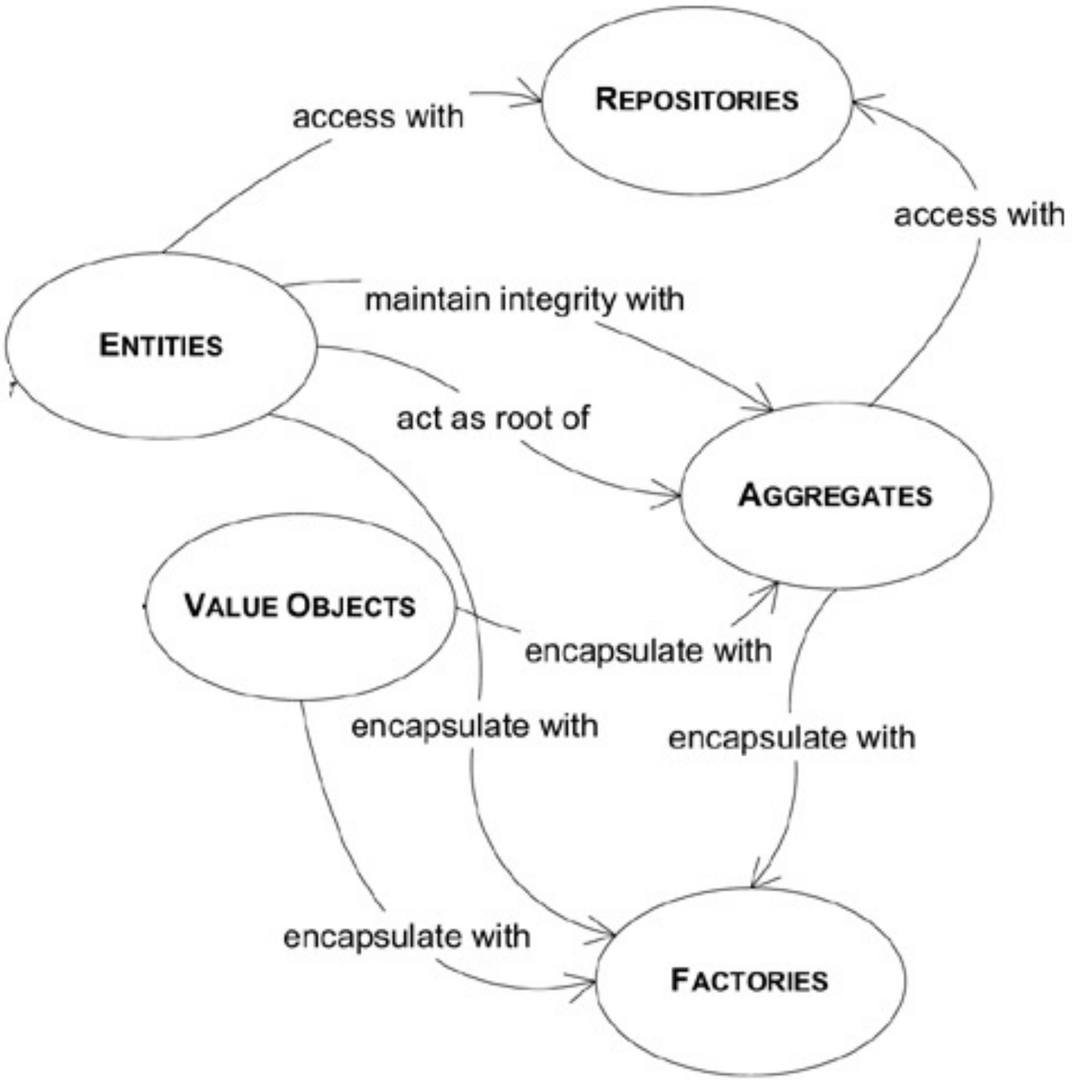
# Elements of a Domain Model: Entities

**Steve Smith**
Force Multiplier for
Dev Teams
@ardalis ardalis.com

**Julie Lerman**
Software coach,
DDD Champion
@julielerman thedatafarm.com

# Understanding terms of modeling




Source: Evans, Domain-Driven Design, p. 65

# Module Overview

- The domain layer in your software
- Focusing on behaviors in a model
- Rich domain models vs. anemic domain models
- Entities in a domain model
- Differentiate entities with complex needs from those needing only CRUD
- Implementing entities in code

# The Importance of Understanding DDD Terms

Collaboration

Driven

Shared

Context

Model

Kernel

Bounded

Common

Ubiquitous

Domain

Communication

Understanding

Design

Anemic

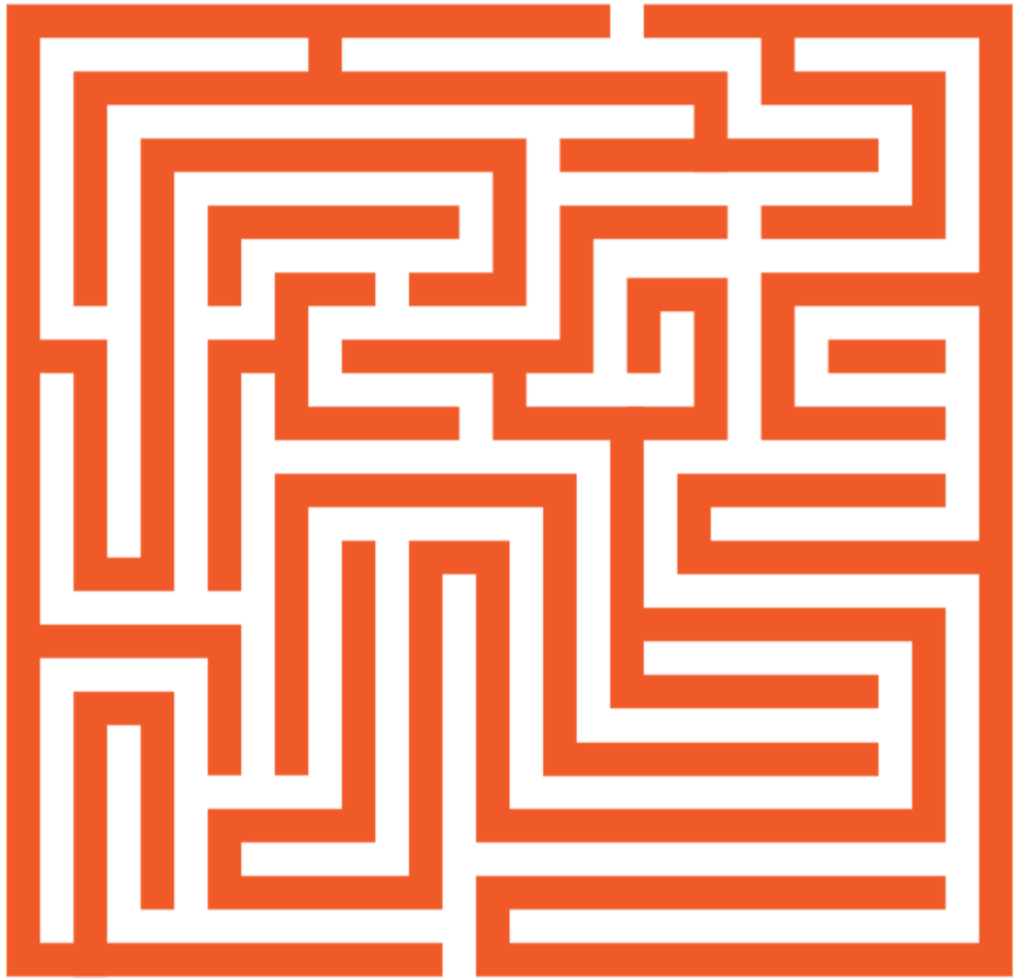
Behavior

# Entity & Context are Common Software Terms

| | Entity Framework Core | Domain-Driven Design |
| --- | --- | --- |
| Entity | **A data model class with a key that is mapped to a table in a database** | **A domain class with an identity for tracking** |
| Context | **A DbContext class provides access to entities and defines how entities map to the database** | **A Bounded Context defines the scope and boundaries of a subset of a domain** |

Value objects can take time to grasp.

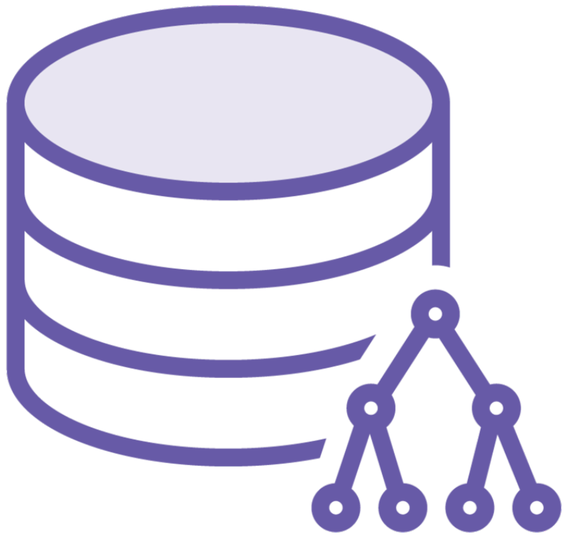
You are not alone!



# Focusing on the Domain

# D is for DOMAIN

# Shift Thinking from DB-Driven to Domain-Driven



**Designing software based on data storage needs**

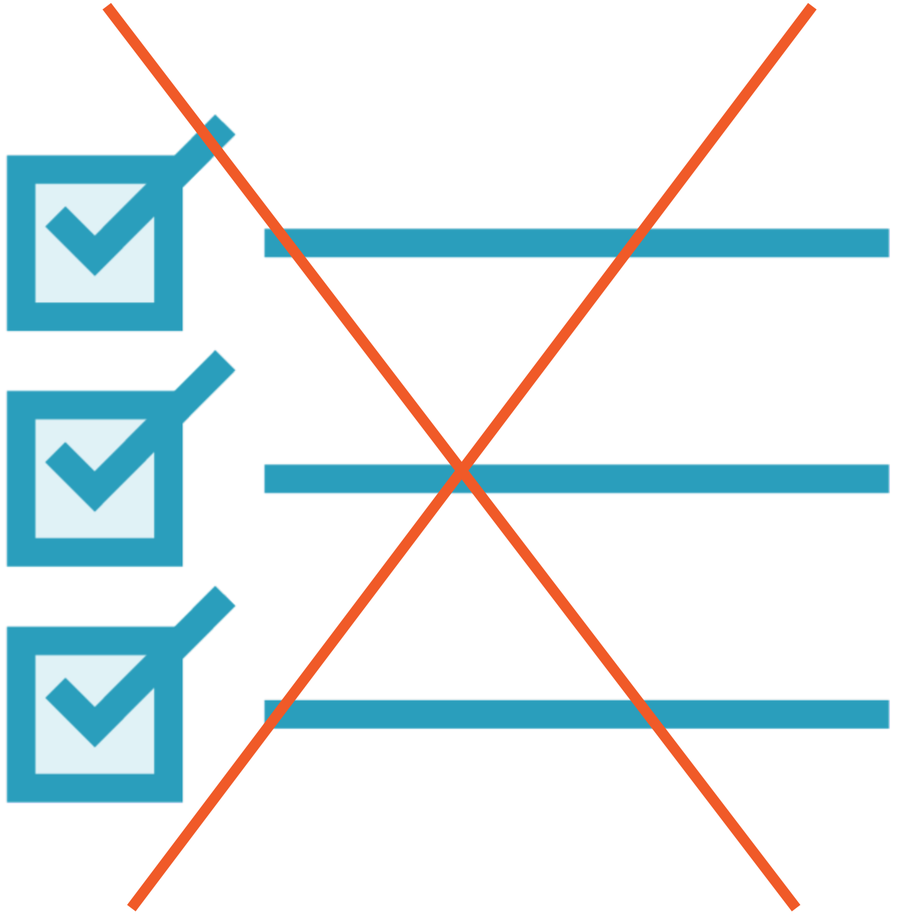
**Designing software based on business needs**

[The Domain Layer is] responsible for representing concepts of the business, information about the business situation, and business rules. State that reflects the business situation is controlled and used here, even though the technical details of storing it are delegated to the infrastructure. *This layer is the heart of business software.*

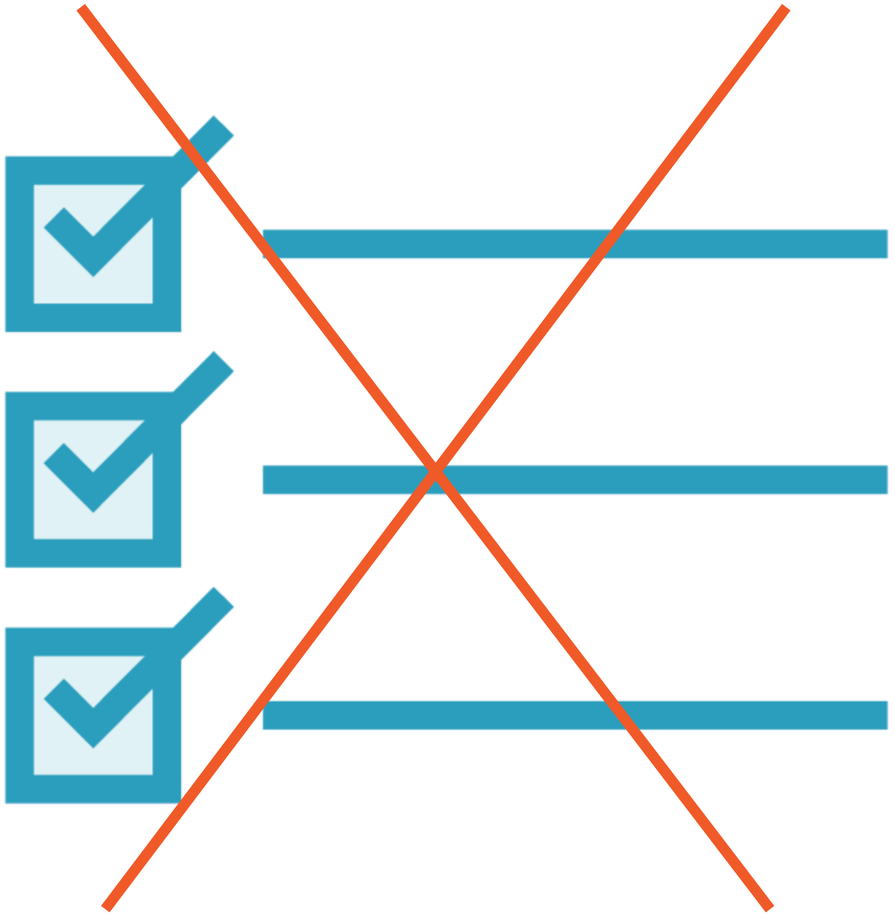
**Eric Evans**

The domain is the heart of business software.

# Focus on Behaviors, Not Attributes



# Focus on Behaviors, Not Attributes



# Behavior Examples

**Schedule** an appointment for a checkup

**Book** a room

**Add** item to vet's calendar

**Note** a pet's weight

**Request** lab work

**Notify** pet owner of vaccinations due

**Accept** a new patient

# Behavior Examples as Events

**Scheduled** an appointment for a checkup

**Booked** a room

**Added** item to vet's calendar

**Noted** a pet's weight

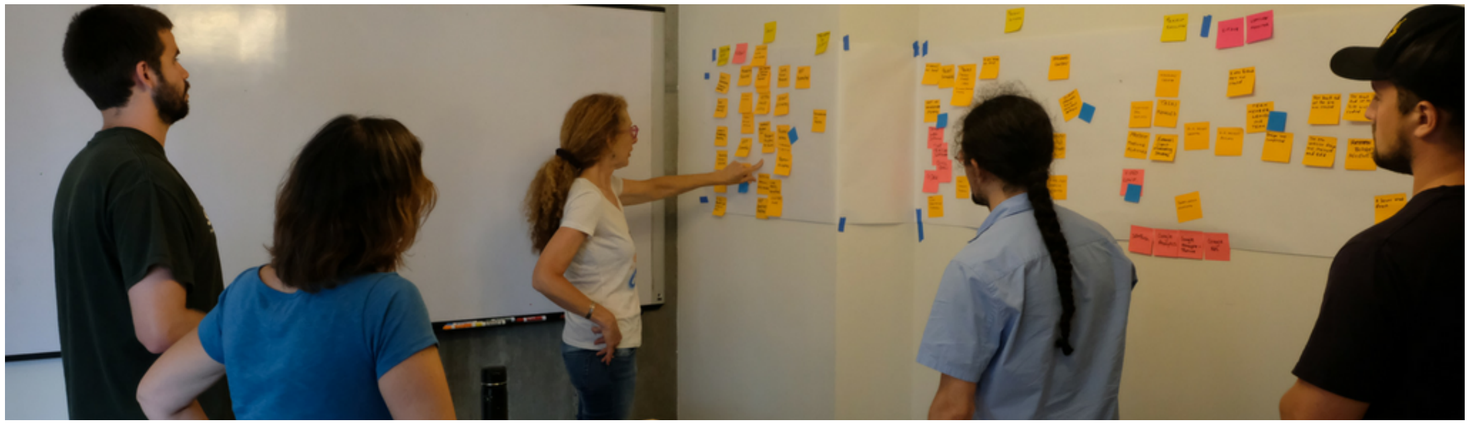
**Requested** lab work

**Notified** pet owner of vaccinations due

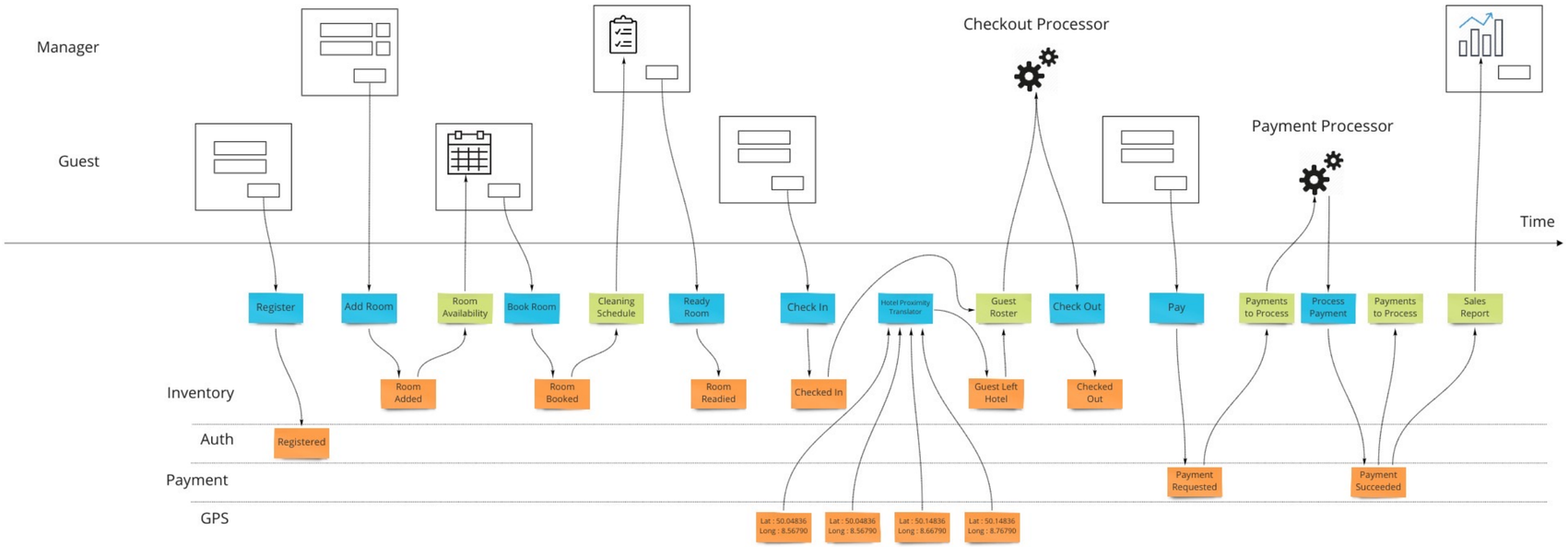
**Accepted** a new patient

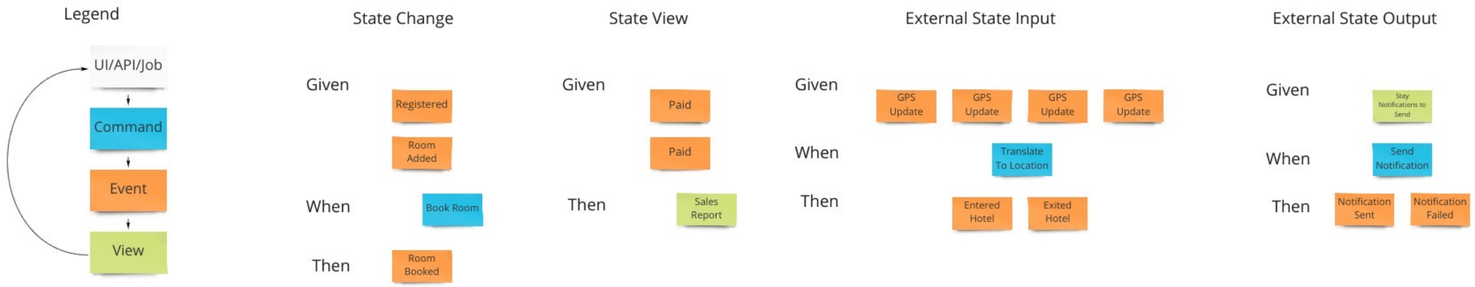
# Identifying Events Leads to Understanding Behaviors

# Event Storming



# Event Modeling




Source: EventModeling.org

# Comparing Anemic and Rich Domain Models

# Domain Model Types

**Anemic**

**Rich**

# Martin Fowler on Recognizing Anemic Domains

**Looks like the real thing with objects named for nouns in the domain**

**Little or no behavior**

**Equate to property bags with getters and setters**

**All business logic has been relegated to service objects**

martinfowler.com/bliki/AnemicDomainModel.html

# Rich Domain Models

The fundamental horror of this anti-pattern is that it's so contrary to the basic idea of object-oriented design; which is to combine data and process together.

**Martin Fowler**

martinfowler.com/bliki/AnemicDomainModel.html

Strive for *rich* domain models

Have an awareness of the strengths and weaknesses of those that are not so rich.

# Understanding Entities

# Two Types of Objects in DDD

**Defined by an identity**

**Defined by its values**

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

**Eric Evans**

# Entity

# Entities Have Identity And Are Mutable

ID=358

Appointment

Sampson

Jan 12 9:30 am

Check-Up

& Vaccinations

ID=172

# Entities Have Identity And Are Mutable

ID=358

Appointment

Sampson

Jan 12 9:30 am

Check-Up

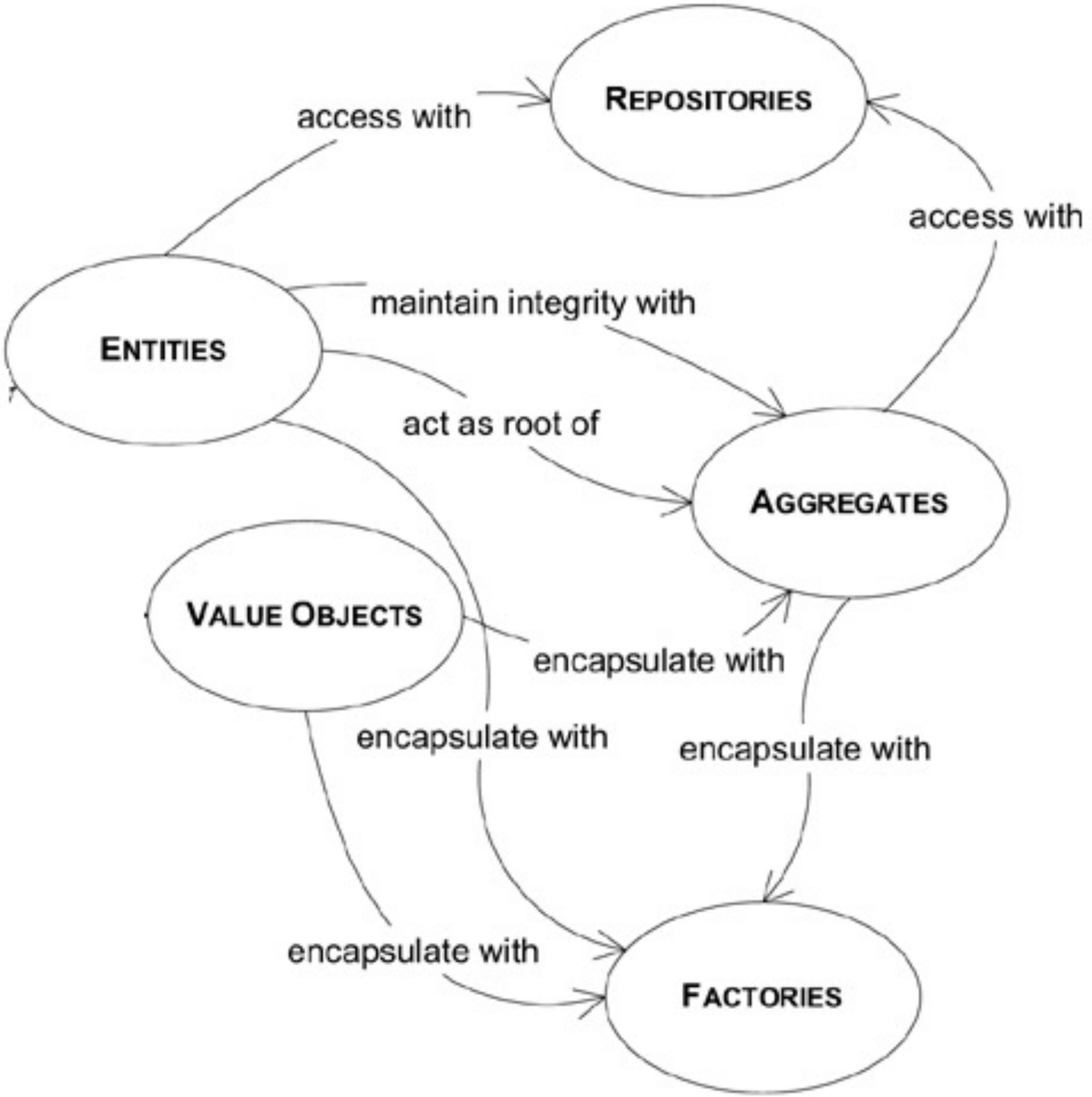
& Vaccinations

*Change appt time*
*Change appt tasks*

ID=172

*Change dog color*

# Entities in the Navigation Map




Source: Evans, Domain-Driven Design, p. 65

# Entities in the Appointment Scheduling Context

**Client : BaseEntity<int>**

- FullName
- Patients

**Patient : BaseEntity<int>**

- AnimalType
- ClientId
- Gender
- Name
- PreferredDoctor

**Appointment : BaseEntity<Guid>**

- ClientId
- DoctorId
- PatientId
- RoomId
- StartEndTime

**Doctor : BaseEntity<int>**

- Name

**Room : BaseEntity<int>**

- Name

# Entities in the Appointment Scheduling Context

**Client : BaseEntity <int>**

- FullName
- Patients

**Patient : BaseEntity <int>**

- AnimalType
- ClientId
- Gender
- Name
- PreferredDoctor

**Appointment : BaseEntity <Guid>**

- ClientId
- DoctorId
- PatientId
- RoomId
- StartEndTime

**Doctor : BaseEntity <int>**

- Name

**Room : BaseEntity <int>**

- Name

# Entities in the Appointment Scheduling Context

**Client : BaseEntity <int>**

- FullName
- Patients

**Patient : BaseEntity <int>**

- AnimalType
- ClientId
- Gender
- Name
- PreferredDoctor

**Appointment : BaseEntity <Guid>**

- ClientId
- DoctorId
- PatientId
- RoomId
- StartEndTime

**Doctor : BaseEntity <int>**

- Name

**Room : BaseEntity <int>**

- Name

# Differentiating CRUD from Complex Problems that Benefit from DDD

# Entities in the Appointment Scheduling Context

**Client : BaseEntity <int>**

- FullName
- Patients

**Patient : BaseEntity <int>**

- AnimalType
- ClientId
- Gender
- Name
- PreferredDoctor

**Appointment : BaseEntity <Guid>**

- ClientId
- DoctorId
- PatientId
- RoomId
- StartEndTime

**Doctor : BaseEntity <int>**

- Name

**Room : BaseEntity <int>**

- Name

# Managing Supporting Data

**Client : BaseEntity <int>**

- FullName
- Patients
- More details...

**Doctor : BaseEntity <int>**

- Name
- More details ...

**Patient : BaseEntity <int>**

- AnimalType
- ClientId
- Gender
- Name
- PreferredDoctor
- More details ...

**Room : BaseEntity <int>**

- Name
- More details ...

# CRUD Classes for Client & Patient Managmenet

```
public class Client : BaseEntity<int>, IAggregateRoot
{
  public string FullName { get; set; }
  public string PreferredName { get; set; }
  public string Salutation { get; set; }
  public string EmailAddress { get; set; }
  public int PreferredDoctorId { get; set; }
  public IList<Patient> Patients { get; private set; }
   = new List<Patient>();

  public Client(string fullName,
    string preferredName,
    string salutation,
    int preferredDoctorId,
    string emailAddress)
  {
    FullName = fullName;
    PreferredName = preferredName;
    Salutation = salutation;
    PreferredDoctorId = preferredDoctorId;
    EmailAddress = emailAddress;
  }

  public override string ToString()
  {
    return FullName.ToString();
  }
}
```

```
public class Patient : BaseEntity<int>
{
  public int ClientId { get; private set; }
  public string Name { get; set; }
  public string Sex { get; set; }
  public AnimalType AnimalType { get; set; }
  public int? PreferredDoctorId { get; set; }

  public Patient(int clientId,
    string name, string sex,
    int? preferredDoctorId = null)
  {
    ClientId = clientId;
    Name = name;
    Sex = sex;
    PreferredDoctorId = preferredDoctorId;
  }

  public override string ToString()
  {
    return Name;
  }
}
```

# Entities in the Appointment Scheduling Context

**Client : BaseEntity <int>**

- FullName
- Patients

**Patient : BaseEntity <int>**

- AnimalType
- ClientId
- Gender
- Name
- PreferredDoctor

**Appointment : BaseEntity <Guid>**

- ClientId
- DoctorId
- PatientId
- RoomId
- StartEndTime

**Doctor : BaseEntity <int>**

- Name

**Room : BaseEntity <int>**

- Name

# Complex Entities vs. Read-Only Entities in Scheduling Context

- ✓ Retrieve
- ✓ Requires Unique ID

**Client :BaseEntity<int>**

- FullName
- Patients

**Patient : BaseEntity<int>**

- AnimalType
- ClientId
- Gender
- Name
- PreferredDoctor

**Appointment : BaseEntity<Guid>**

- ClientId
- DoctorId
- PatientId
- RoomId
- StartEndTime

- ✓ Retrieve
- ✓ Track
- ✓ Edit
- ✓ Requires Unique ID

**Doctor : BaseEntity<int>**

- Name

**Room : BaseEntity<int>**

- Name

# Switching Between Contexts in a UI

User interface should be designed to hide the existence of bounded contexts from end users


VaughnVernon.com

# Ubiquitous Language to the Rescue

**Appointment Scheduling Bounded Context**

**Client**

**Identity**

***Read-only*** **name of pet owner**

**Client/Patient Management Bounded Context**

**Client**

**Identity**

***Mutable*** **name of pet owner**

**Contact info**

**Billing info**

# Using GUIDs or Ints for Identity Values

# GUIDs as Unique Identifiers with No Database Dependency

```
cb8804a1-2773-4538-8cf7-53a0463ab911
```

# GUIDs or Ints for Identifiers?

**GUID doesn't depend on database**

**Database performance favors int for keys**

**You can use both!**

# Talking with Eric Evans About the Responsibility of Entities

# Learn more about the Single Responsibility Principle

## SOLID Principles for C# Developers

Steve Smith

What is the single responsibility of an entity?

Entities can get loaded down with logic

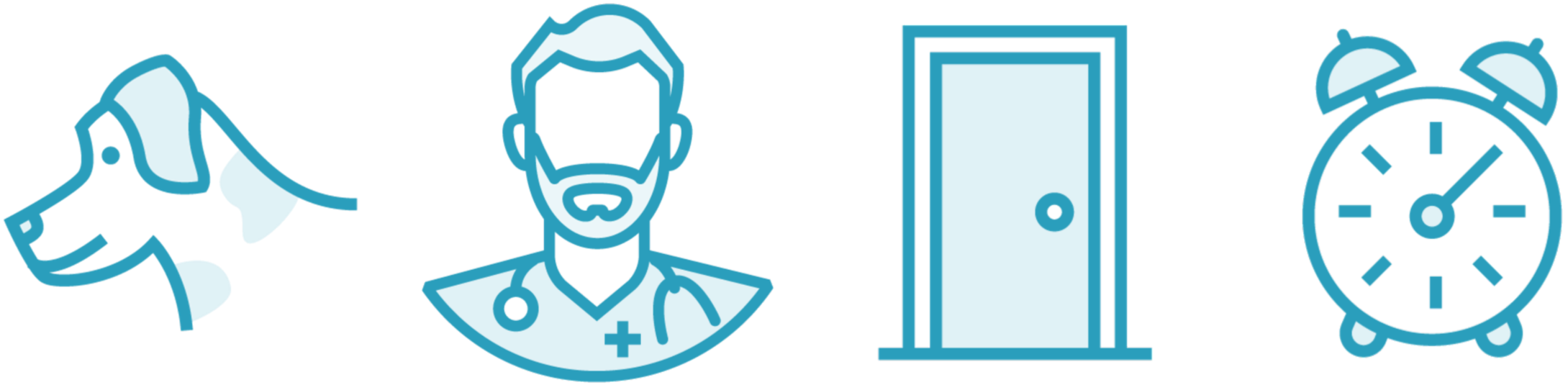
Their core responsibility is identity and life-cycle

“Single Responsibility is a good principle to apply to entities. It points you toward the sort of responsibility that an entity should retain. Anything that doesn't fall in that category we ought to put somewhere else.”

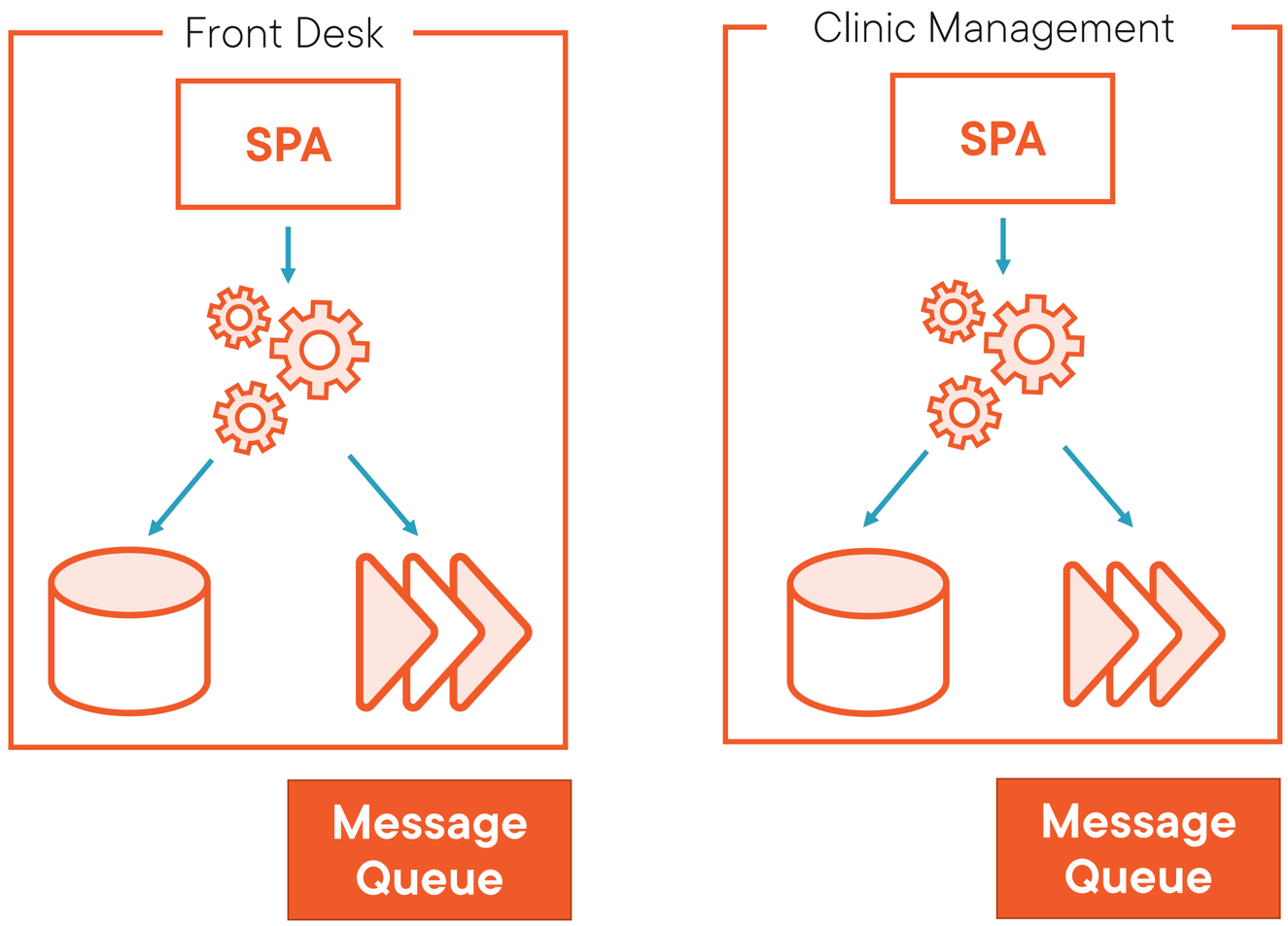
**Eric Evans**

# Implementing Entities in Code

# Appointment Class Involves



# Synchronizing Data Across Bounded Contexts


Front Desk
SPA
Message Queue
Clinic Management
SPA
Message Queue

# Eventual Consistency

**Systems do not need to be strictly synchronized, but the changes will eventually get to their destination.**

# Module Review and Resources

# Key Terms from this Module

## Anemic Domain Model

Model with classes focused on state management. Good for CRUD.

## Rich Domain Model

Model with logic focused on behavior, not just state. Preferred for DDD.

## Entity

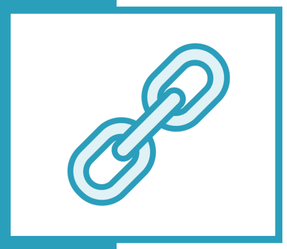
A mutable class with an identity (not tied to its property values) used for tracking and persistence.

# Key Takeaways

- Implementing the bounded context
- Anemic models are better suited for CRUD.
- Rich domain models help us solve complex problems.
- Entities are driven by their identity value.
- Difference between rich entities and reference entities
- Message queues are one way to share data across bounded contexts.

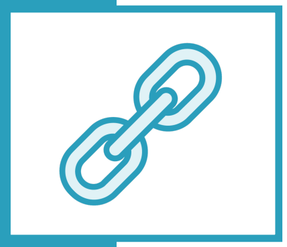
# Up Next: Value Objects & Services

# Resources Referenced in This Module

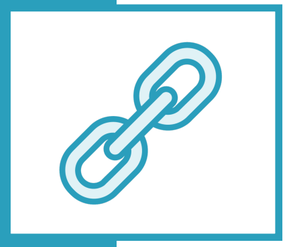


**Jimmy Bogard, Services in DDD bit.ly/1ifravE**

**Pluralsight course: SOLID Principles of OO Design bit.ly/solid-smith**

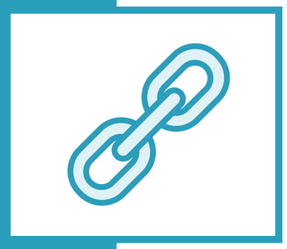


**Martin Fowler on Anemic Domain Models: martinfowler.com/bliki/AnemicDomainModel.html**

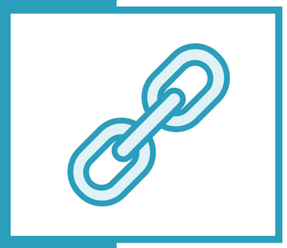


**Vaughn Vernon vaughnvernon.com**

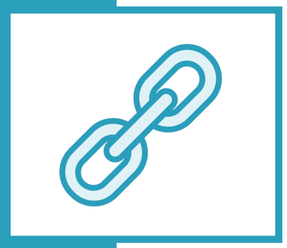
# Resources Referenced in This Module



**Eric Evans domainlanguage.com**



**Event storming via Alberto Brandolini eventstorming.com**



**Event Modeling via Adam Dymitruk eventmodeling.org**

# Elements of a Domain Model

**Steve Smith**
Force Multiplier for
Dev Teams
@ardalis ardalis.com

**Julie Lerman**
Software coach,
DDD Champion
@julielerman thedatafarm.com