Elements of a Domain Model

Steve Smith

Force Multiplier for
Dev Teams

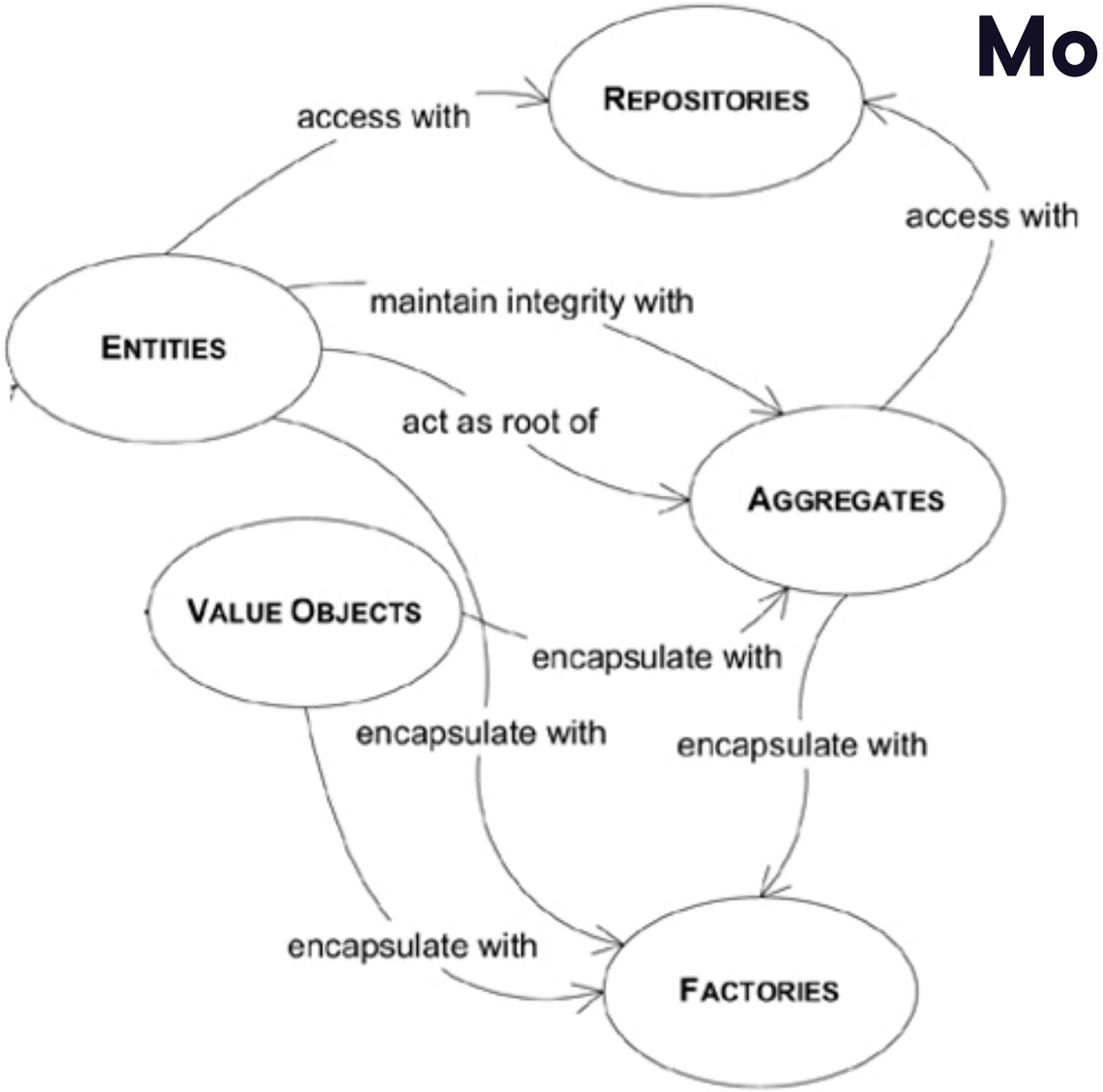
@ardalis | ardalis.com

Julie Lerman

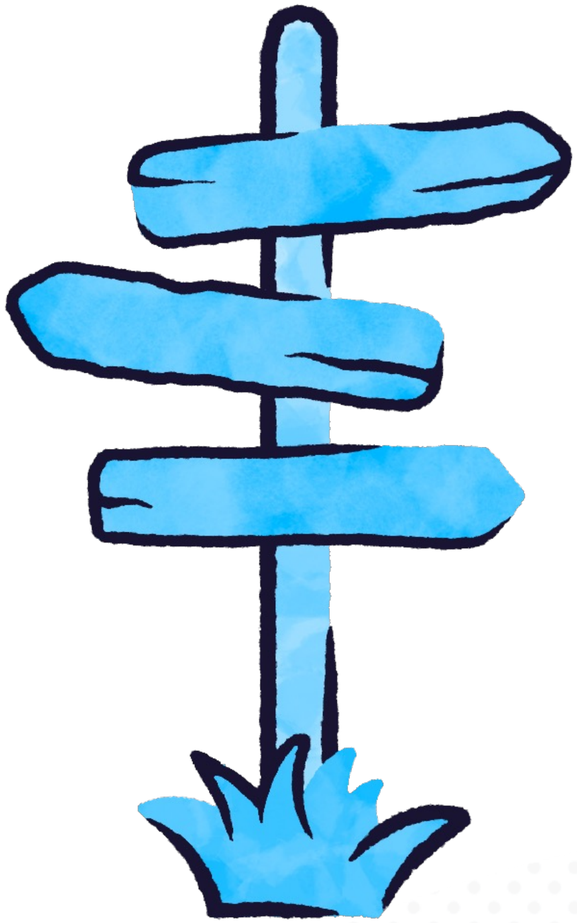
Software Coach,
DDD Champion

@julielerman | thedatafarm.com

Understanding Terms of Modeling



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

Shared

Driven

Context

Kernel

Model

Bounded

Ubiquitous

Common

Domain

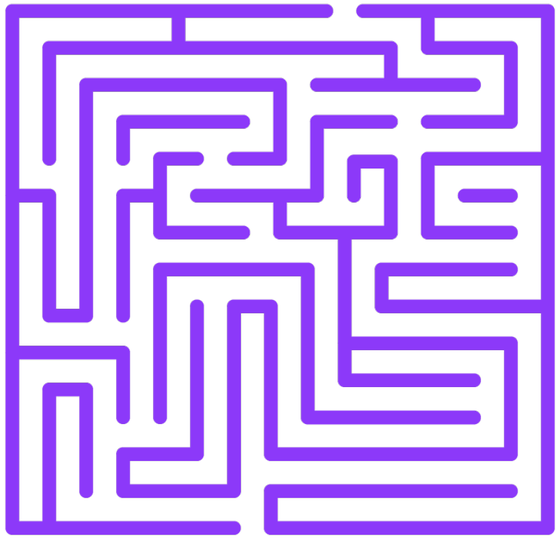
Communication

Understanding

Design

Anemic

Behavior



Value objects can take time to grasp

You are not alone!

Entity and Context Are Common Terms in Software

Entity

In Entity Framework Core:

A data model class that with a key that is mapped to a table in a database

In DDD:

A domain class with an ID for tracking

Context

In Entity Framework Core:

A DbContext class acts as a repository for entities and defines how entities map to the database

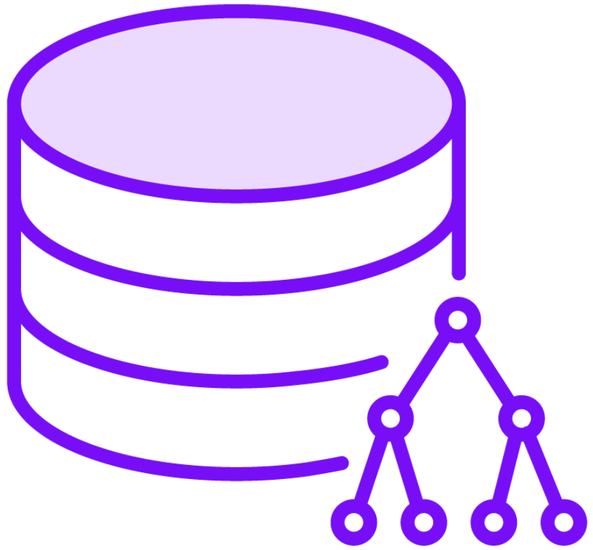
In DDD:

A Bounded Context defines the scope and boundaries of a subset of a domain

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

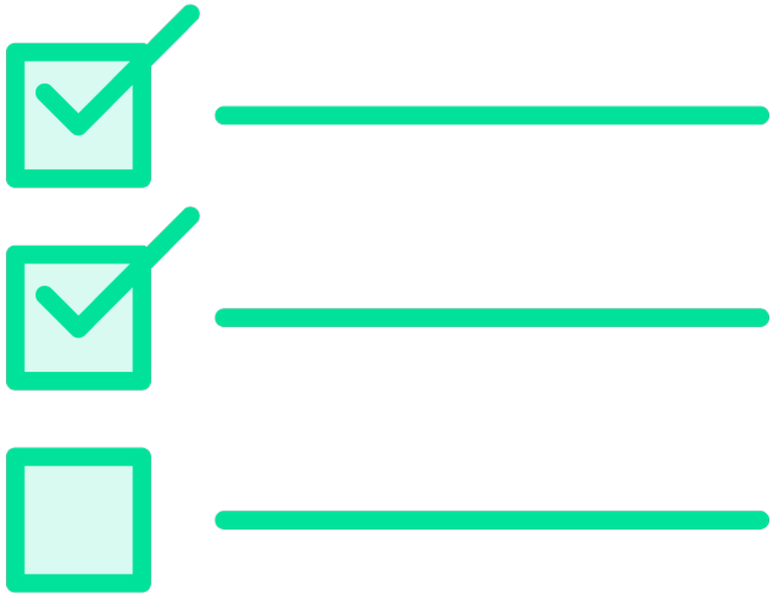
Eric Evans

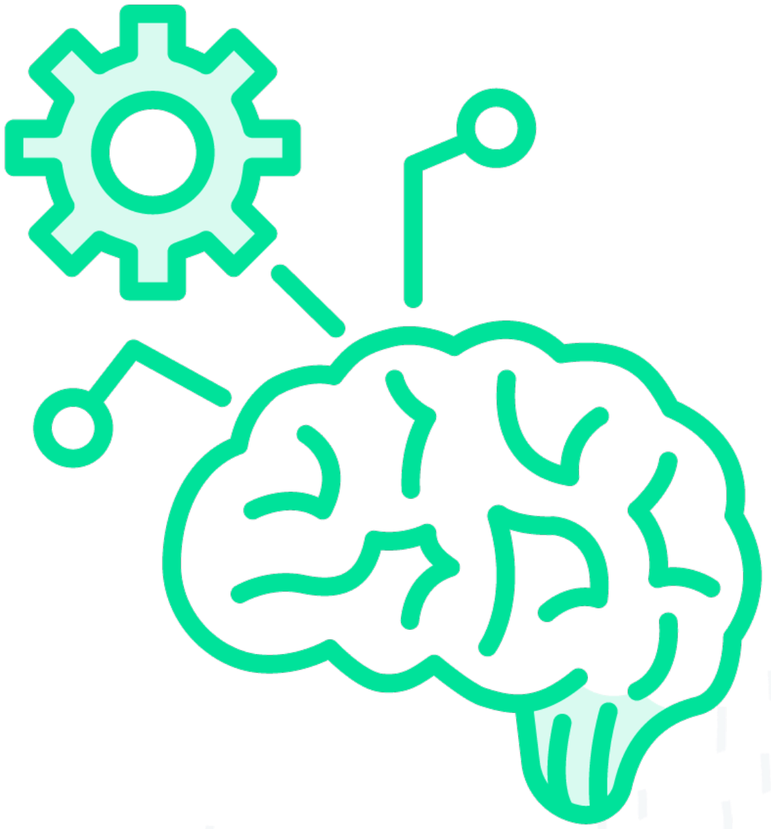
[The Domain Layer is] responsible for representing concepts of the business, information about the business situation, and business rules. State that reflects the business situation is controlled and used here, even though the technical details of storing it are delegated to the infrastructure. This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

Citation: Domain-Driven Design, Addison-Wesley 2004

The domain is the heart of business software.

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

Identifying Event Leads to Understanding Behaviors

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

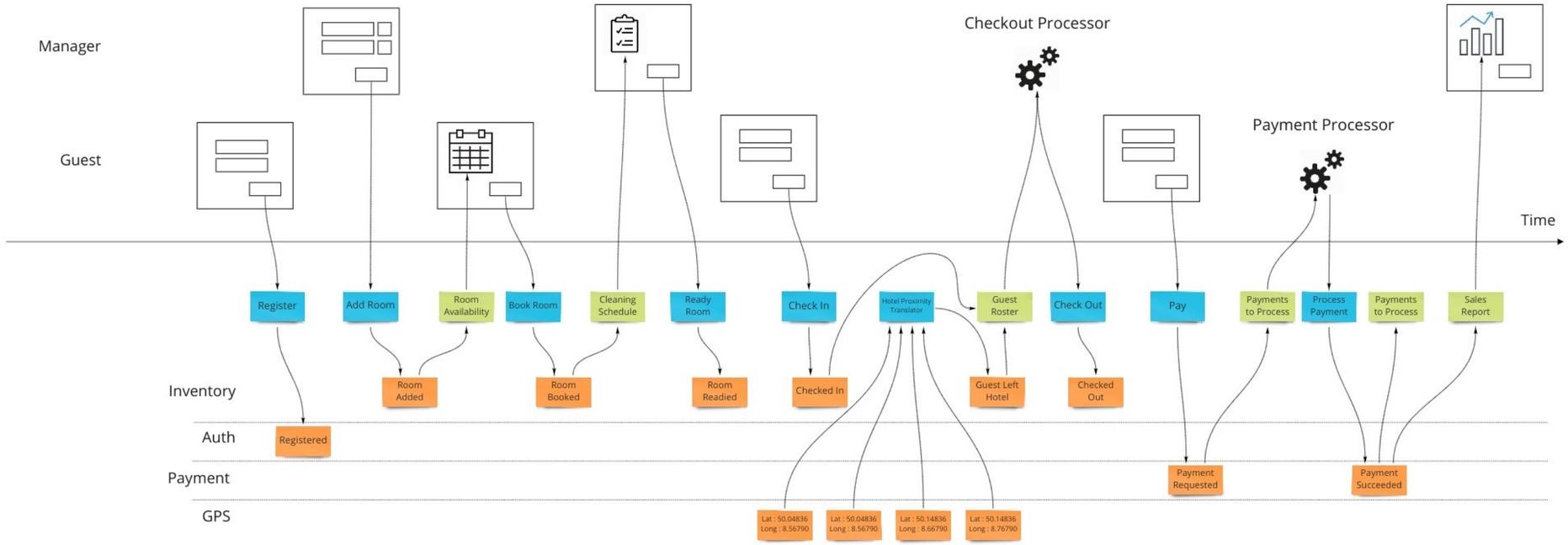
Accept a new patient

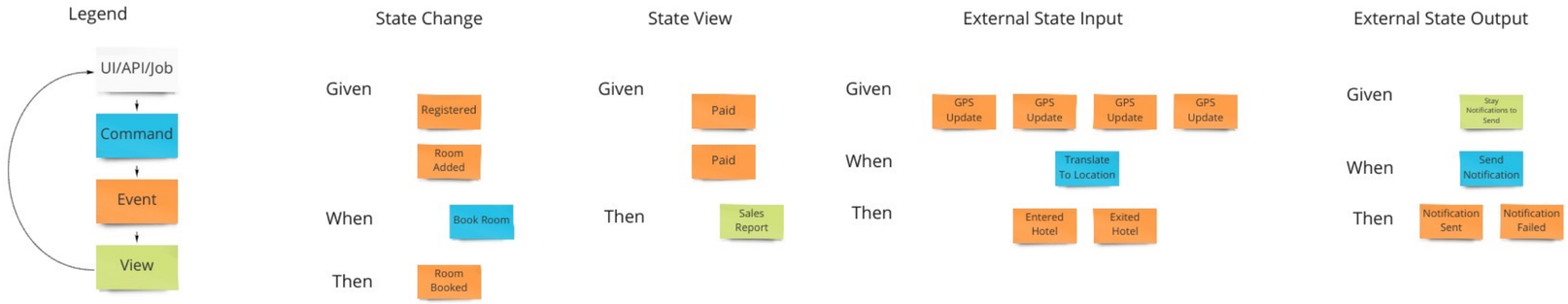
Event Storming

Alberto Brandolini eventstorming.org

Event Storming

Event Modeling





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

Citation: martinfowler.com/bliki/AnemicDomainModel.html

Rich Domain Models

Martin Fowler

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.

Citation: 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

Eric Evans

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

entity

Entities Have Identity and Are Mutable

ID=358

Appointment

ID=172

Sampson

Jan 12 10:00 am

Check-Up

Entities Have Identity and Are Mutable

ID=358

Appointment

ID=172

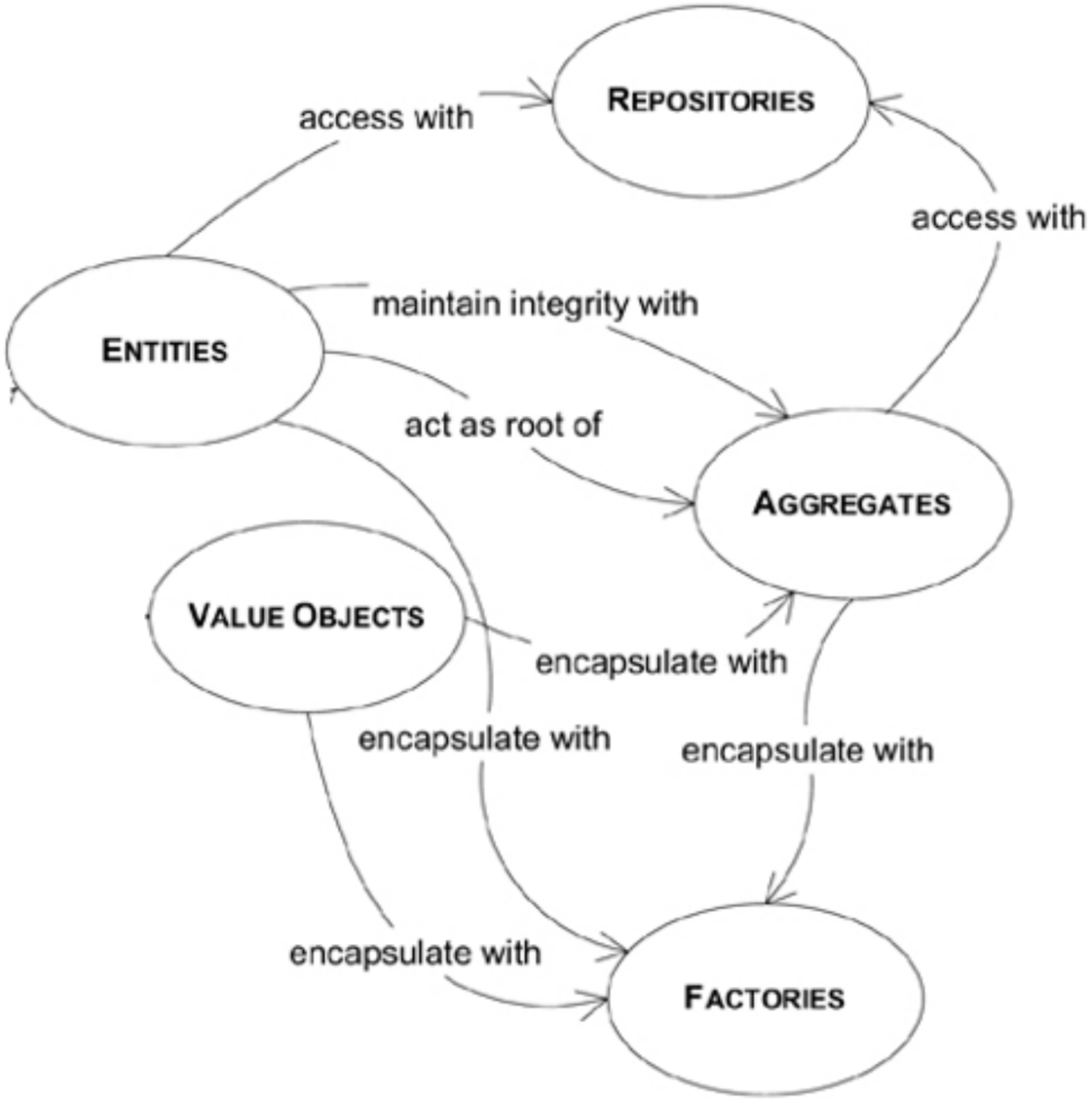
Sampson

Jan 12 9:30 am

Check-Up

& Vaccinations

Entities in the Navigation Map



Entities in the Appointment Scheduling Context

Client : BaseEntity<int>

- FullName
- Patients

Doctor : BaseEntity<int>

- Name

Appointment : BaseEntity<Guid>

- ClientID
- DoctorID
- PatientID
- RoomID
- StartEndTime

Room : BaseEntity<int>

- Name

Patient : BaseEntity<int>

- AnimalType
- ClientID
- Gender
- Name
- PreferredDoctor

Entities in the Appointment Scheduling Context

Client : BaseEntity<int>

- FullName
- Patients

Doctor : BaseEntity<int>

- Name

Appointment : BaseEntity<Guid>

- ClientID
- DoctorID
- PatientID
- RoomID
- StartEndTime

Room : BaseEntity<int>

- Name

Patient : BaseEntity<int>

- AnimalType
- ClientID
- Gender
- Name
- PreferredDoctor

Entities in the Appointment Scheduling Context

Client : BaseEntity<int>

- FullName
- Patients

Doctor : BaseEntity<int>

- Name

Appointment : BaseEntity<Guid>

- ClientID
- DoctorID
- PatientID
- RoomID
- StartEndTime

Room : BaseEntity<int>

- Name

Patient : BaseEntity<int>

- AnimalType
- ClientID
- Gender
- Name
- PreferredDoctor

Managing Supporting Data

Client : BaseEntity<int>

- FullName
- Patients
- More details...

Doctor : BaseEntity<int>

- Name
- More details...

Room : BaseEntity<int>

- Name
- More details...

Patient : BaseEntity<int>

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

Differentiating CRUD from Complex Problems That Benefit from DDD

Entities in the Appointment Scheduling Context

Client : BaseEntity<int>

- FullName
- Patients

Doctor : BaseEntity<int>

- Name

Appointment : BaseEntity<Guid>

- ClientID
- DoctorID
- PatientID
- RoomID
- StartEndTime

Room : BaseEntity<int>

- Name

Patient : BaseEntity<int>

- AnimalType
- ClientID
- Gender
- Name
- PreferredDoctor

Managing Supporting Data

Client : BaseEntity<int>

- FullName
- Patients
- More details...

Doctor : BaseEntity<int>

- Name
- More details...

Room : BaseEntity<int>

- Name
- More details...

Patient : BaseEntity<int>

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

CRUD Classes for Client and Patient Management

```
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;
  }
}
```

Complex Entities vs. Read-Only Entities in Scheduling Context

✓ Retrieve
✓ Requires Unique ID

Client :Entity<int>

- FullName
- Patients

Doctor :Entity<int>

- Name

Appointment :Entity<Guid>

- ClientID
- DoctorID
- PatientID
- RoomID
- StartEndTime

✓ Retrieve
✓ Track
✓ Edit
✓ Requires Unique ID

Room :Entity<int>

- Name

Patient :Entity<int>

- AnimalType
- ClientID
- Gender
- Name
- PreferredDoctor

Switching between Contexts in a UI

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

Complex Entities vs. Read-Only Entities in Scheduling Context

✓ Retrieve
✓ Requires Unique ID

Client :Entity<int>

- FullName
- Patients

Doctor :Entity<int>

- Name

Appointment :Entity<Guid>

- ClientID
- DoctorID
- PatientID
- RoomID
- StartEndTime

✓ Retrieve
✓ Track
✓ Edit
✓ Requires Unique ID

Room :Entity<int>

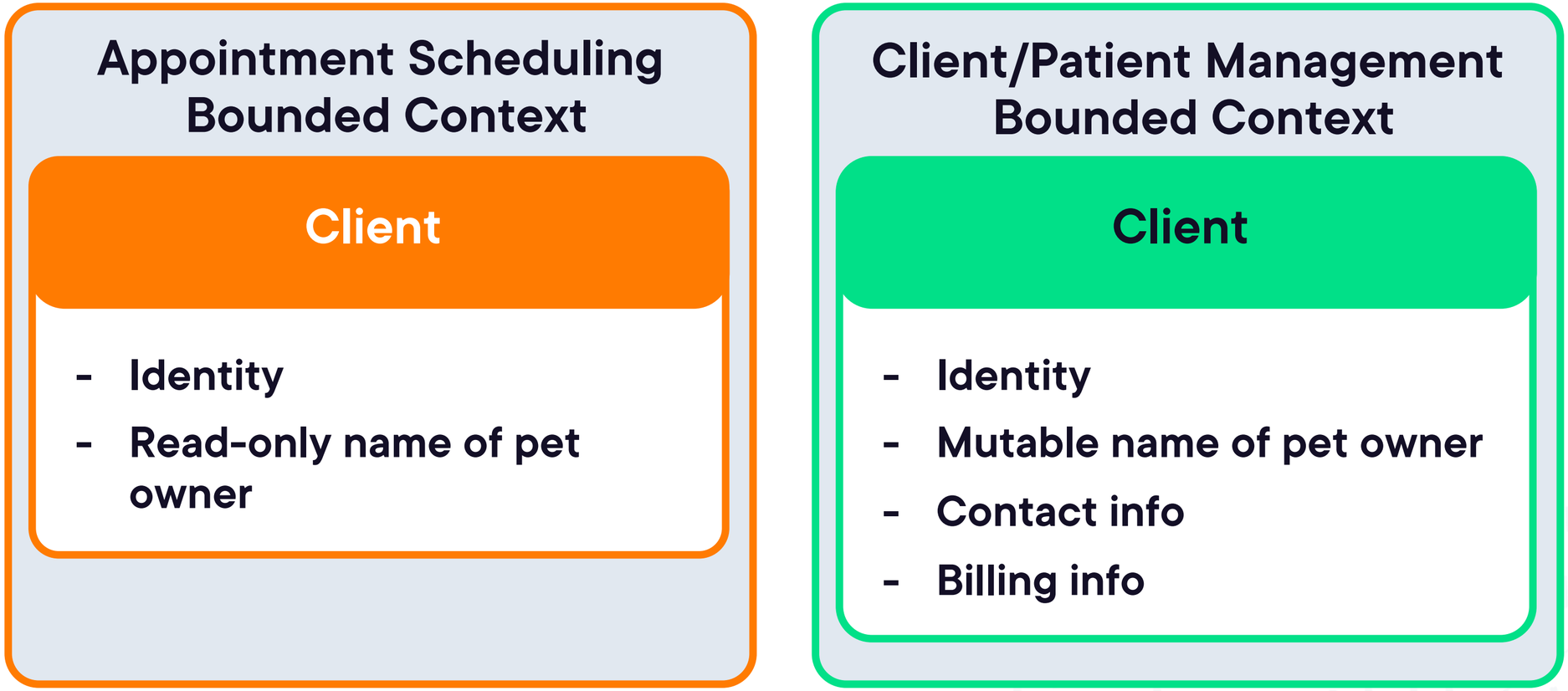
- Name

Patient :Entity<int>

- AnimalType
- ClientID
- Gender
- Name
- PreferredDoctor

VaughnVernon.com

Ubiquitous Language to the Rescue



Using GUIDs or Ints for Identity Values

Complex Entities vs. Read-Only Entities in Scheduling Context

✓ Retrieve
✓ Requires Unique ID

Client :Entity<int>

- FullName
- Patients

Doctor :Entity<int>

- Name

Appointment :Entity<Guid>

- ClientID
- DoctorID
- PatientID
- RoomID
- StartEndTime

✓ Retrieve
✓ Track
✓ Edit
✓ Requires Unique ID

Room :Entity<int>

- Name

Patient :Entity<int>

- AnimalType
- ClientID
- Gender
- Name
- PreferredDoctor

GUIDs as Unique Identifiers with No Database Dependency

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

Easier when implementing DDD, but not required

GUIDs or Ints for Identifiers?

GUID doesn't depend on database

Database performance favors int for keys

You can use both!

Conversation with Eric Evans: Entities

Learning More About the Single Responsibility Principle

SOLID Principles for C# Developers

Steve Smith

What is the single responsibility of an entity?

Eric Evans and Entities

Entities can get loaded down with logic

Their core responsibility is identity and lifecycle

Eric Evans

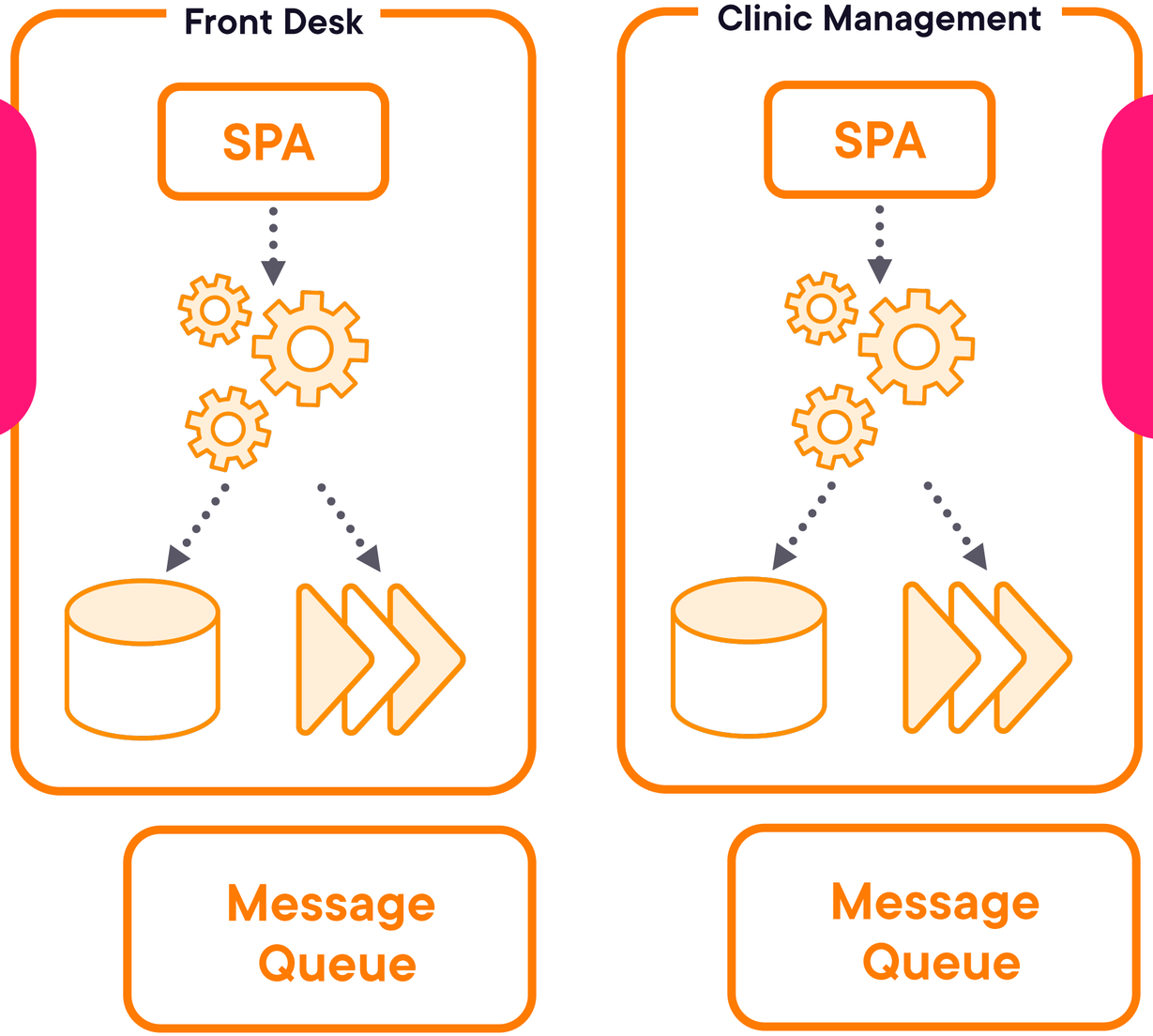
Single Responsibility is a good principle to apply to entities. It points you toward the sort of responsibility that an entity should retain.

Anything that doesn’t fall in that category we ought to put somewhere else.

Implementing Entities in Code

Synchronizing Data across Bounded Contexts

Subscribe to updates



Performs and publishes updates

Eventual Consistency

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

Key Terms and Resources

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

Value objects and domain services are coming up next