Domain Driven Design

What is DDD?

"Domain-Driven Design is an approach to software development that centers the development on programming a domain model that has a rich understanding of the processes and rules of a domain."

Martin Fowler

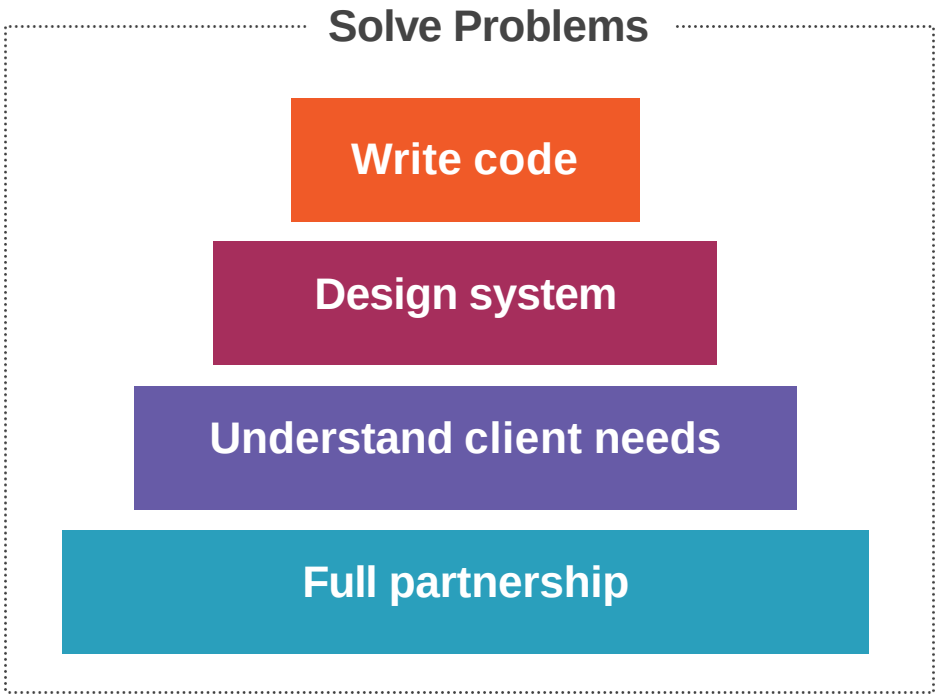
Value Proposition of DDD

Principles and patterns to solve difficult problems

History of success with complex projects

Aligns with practices from our own experience

Clear, readable, testable code that represents the domain


Solve Problems
Write code
Design system
Understand client needs
Full partnership

Separate Subdomains For Each Problem

Purchase materials	tasks	ubiquitous language	unique problems
Engineering	tasks	ubiquitous language	unique problems
Manage employees	tasks	ubiquitous language	unique problems
Marketing	tasks	ubiquitous language	unique problems
Sales	tasks	ubiquitous language	unique problems

Interaction with domain experts

Model a single subdomain at a time

Implementation of subdomains

Separation of Concerns
plays an important role in implementing subdomains

Benefits of Domain-Driven Design

- Flexible
- Customer's vision/ perspective of the problem
- Path through a very complex problem
- Well-organized and easily tested code
- Business logic lives in one place
- Many great patterns to leverage

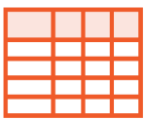
DDD aims to
tackle business complexity,
not technical complexity

“While Domain-Driven Design provides many technical benefits, such as maintainability, it should be applied only to complex domains where the model and the linguistic processes provide clear benefits in the communication of complex information, and in the formulation of a common understanding of the domain.”

Eric Evans, Domain-Driven Design

Be Thoughtful About Possible Overuse

DDD is for handling complexity in business problems



Not just CRUD or data-driven applications

Not just technical complexity without business domain complexity

Problem Domain

D is for DOMAIN

Problem Domain

The specific problem the software you're working on is trying to solve

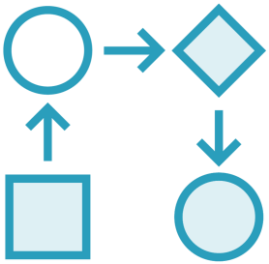
Core Domain

The key differentiator for the customer's business -- something they must do well and cannot outsource

Subdomains

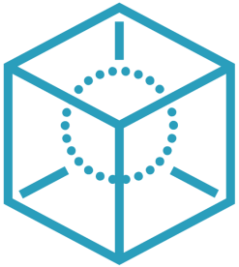
Separate applications or features your software must support or interact with

Our Goals for Learning About the Domain



Understand client's business

Identify processes beyond project scope



Look for subdomains we should include

Look for subdomains we can ignore

As software developers, we fail in two ways:
We build the thing wrong, or
We build the wrong thing.

Steve Smith

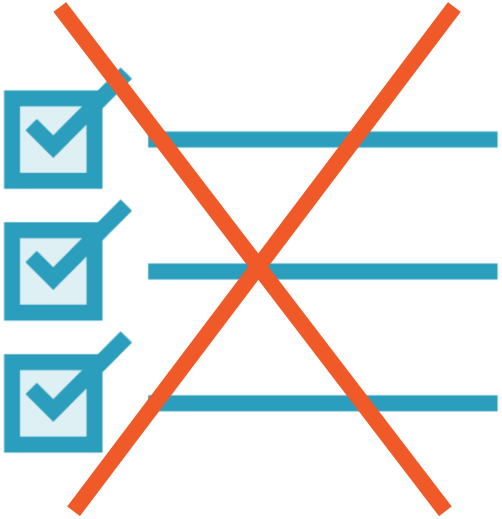
Shift Thinking from DB-Driven to Domain-Driven

Designing software based on data storage needs

Designing software based on business needs

The domain is the heart of business software.

Focus on Behaviors, Not Attributes



Anemic and Rich Domain Models

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.

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

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

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

Entities

Entity

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

Putting Value Objects into Context for .NET Devs

Value Types	Reference Types	DDD Entities & Value Objects
Defined with structs	**Defined with classes**	**Typically defined with classes**

“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

Value Object

An immutable class whose identity is dependent on the combination of its values.

Immutable - refers to a type whose state cannot be changed once the object has been instantiated

It may surprise you to learn that we should strive to model using Value Objects instead of Entities wherever possible. Even when a domain concept must be modeled as an Entity, the Entity's design should be biased toward serving as a value container rather than a child Entity container.

Vaughn Vernon – Implementing Domain Driven Design

When Considering Domain Objects

Our Instinct:

1. **Probably an entity**
2. **Maybe a value object**

Vaughn Vernon's guidance:

1. **Is this a value object?**
2. **Otherwise, an entity**

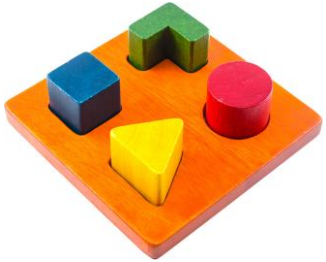
Bounded Context

Bounded Context

A specific responsibility, with explicit boundaries that separate it from other parts of the system

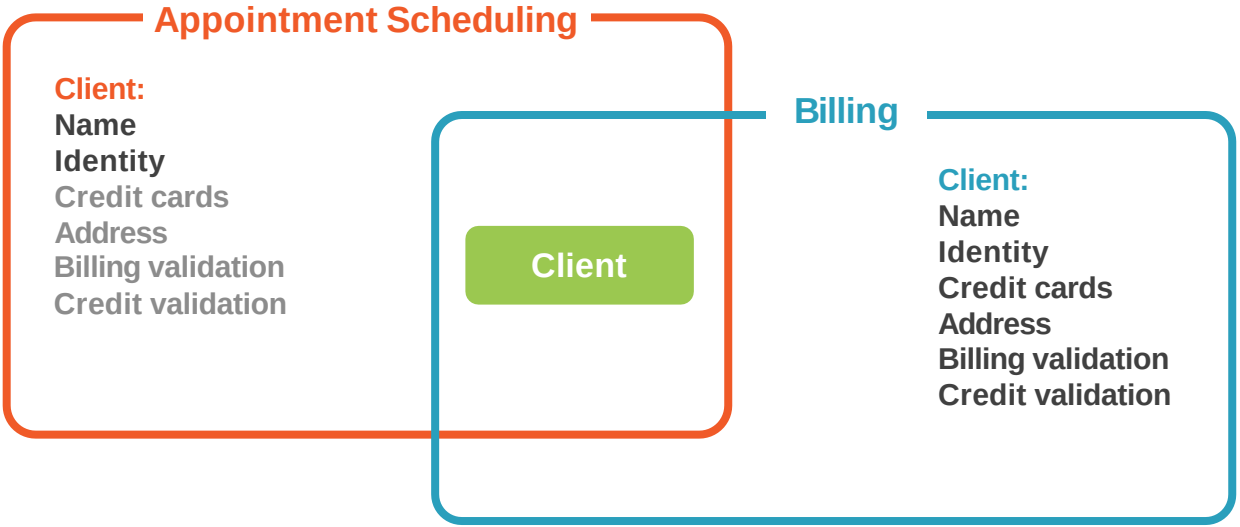
Defining Bounded Contexts

Define a strong boundary around the concepts of each model



Ensure model's concepts don't leak into other models where they don't make sense

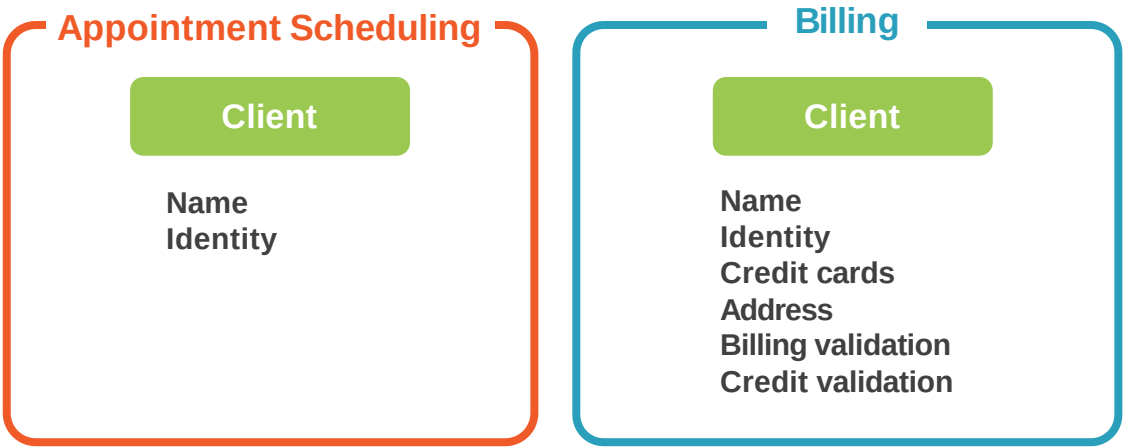
Bounded Context



Explicitly define the context within which a model applies… Keep the model strictly consistent within these bounds, but don’t be distracted or confused by issues outside.

Eric Evans

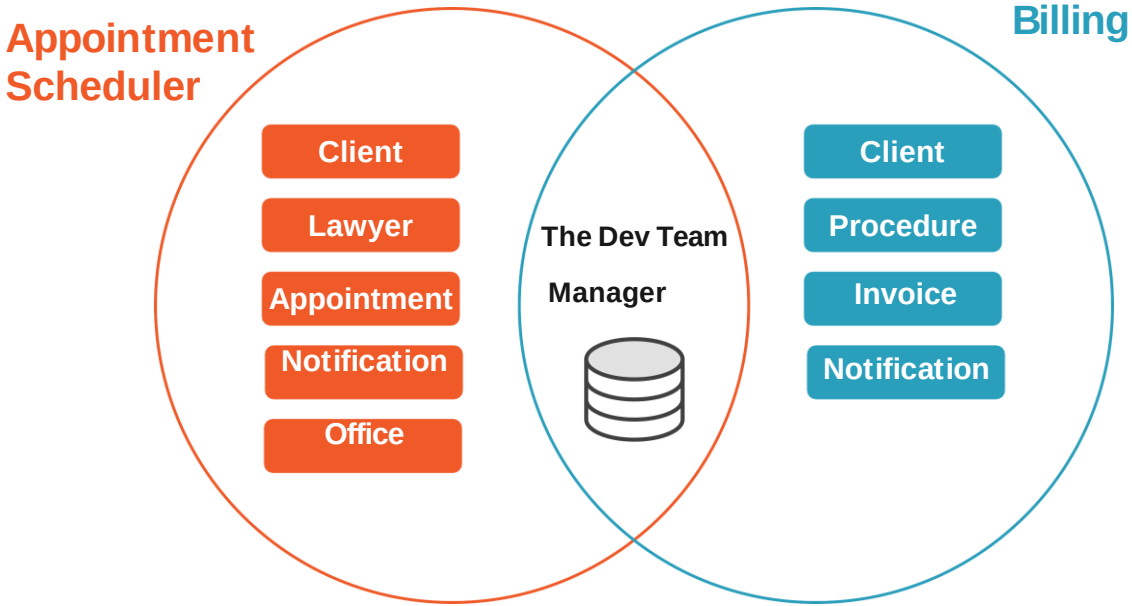
Bounded Context



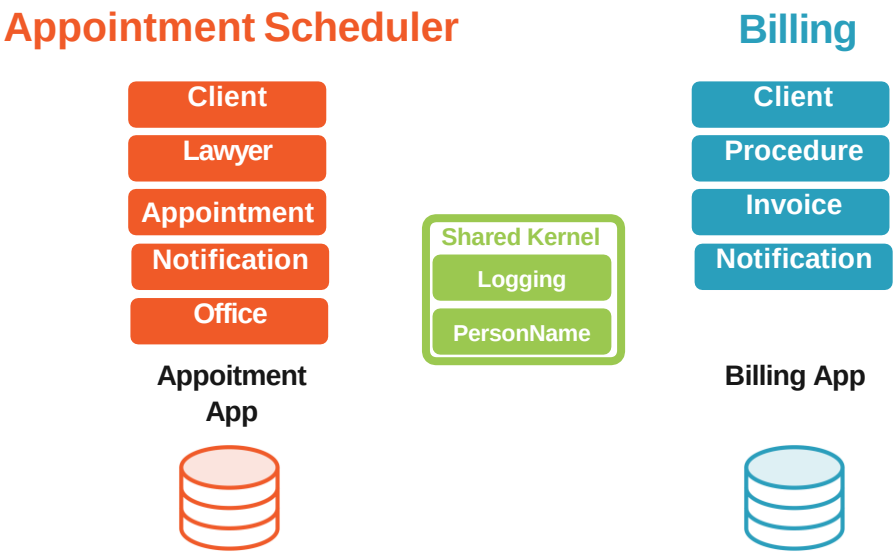
Context Map

Demonstrates how bounded contexts connect to one another while supporting communication between teams.

Context Maps



Context Maps



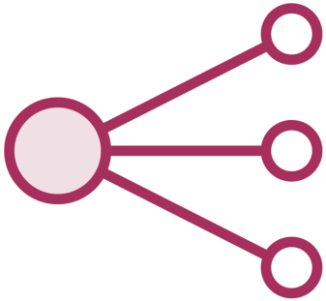
So Many Databases for Bounded Contexts

So Many Databases for Microservices

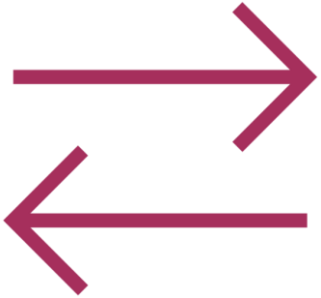
“If you’re in a company where you share your database and it gets updated by hundreds of different processes, it's very hard to create the kind of models that we're talking about and then write software that does anything interesting with those models.”

Eric Evans

Some Common Patterns for Data Syncing



Publisher Subscriber

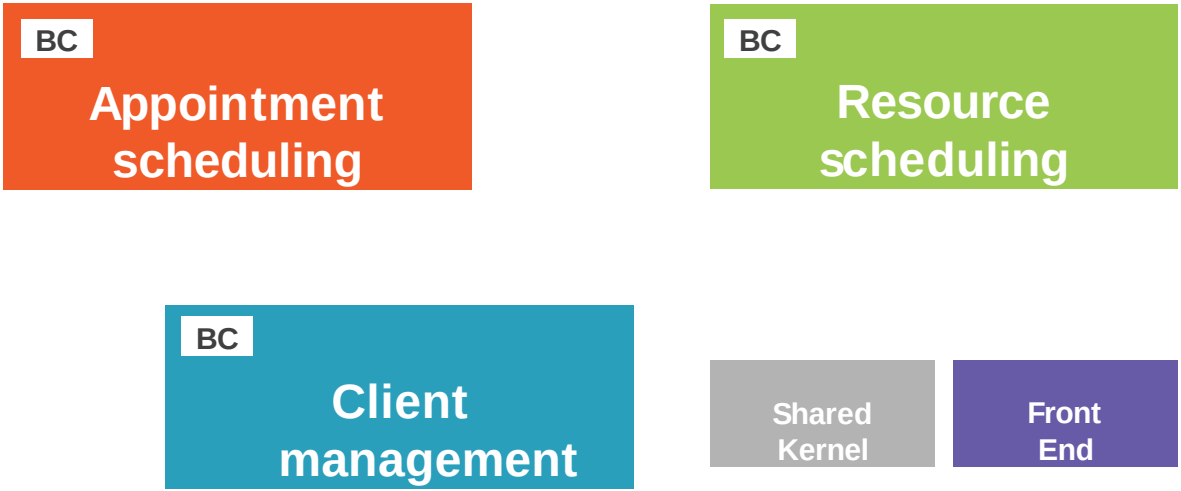


2-way Synchronization

Implementations

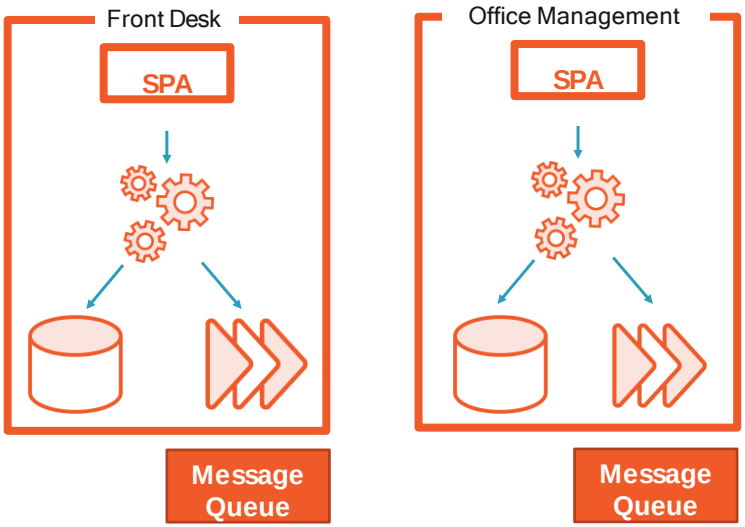
Message Queues | **Database processes** | **Batch jobs** | **Synchronization APIs** | **& more …**

Bounded Contexts in the Application



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

Synchronizing Data Across Bounded Contexts



Eventual Consistency

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

Ubiquitous Language

The language we use is key to the shared understanding we want to have with our domain experts in order to be successful.

Pro Tip!

Try to explain back to the customer what you think they explained to you

Avoid:
“What I meant was…”

A project faces serious problems when its language is fractured.

Eric Evans

A ubiquitous language applies to a single bounded context and is used throughout conversations and code for that context.

Recognize that a change in the ubiquitous language is a change in the model.

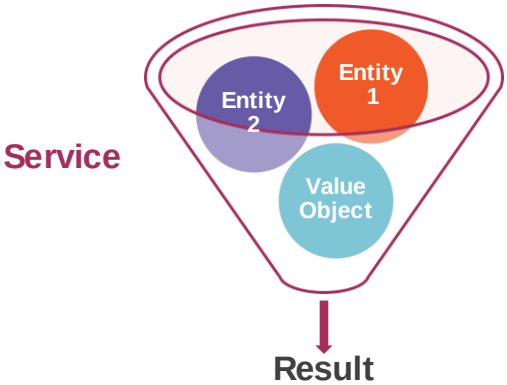
Eric Evans

Domain Service

Some operations make more sense in a ***domain service***.

Domain services Provide a place in the model to hold behavior that doesn't belong elsewhere in the domain

Domain Service Orchestrates Processes Across Objects



Features of a Domain Service

- Not a natural part of an entity or value object
- Has an interface defined in terms of other domain model elements
- Stateless, but may have side effects
- Lives in the core of the application

Examples of Services in Different Layers

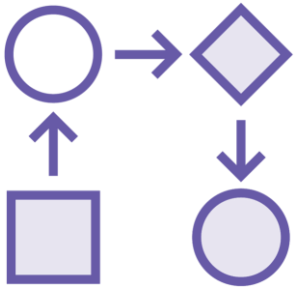
UI and App

Message Sending

Message Processing

XML Parsing

UI Services



Domain

Orchestrating workflow

Transfer Between Accounts

Process Order

Infrastructure

Send Email

Log to a File

Aggregates and Associations

Aggregate

A transactional graph of objects

Aggregate Root

The entry point of an aggregate which ensures the integrity of the entire graph

Associations

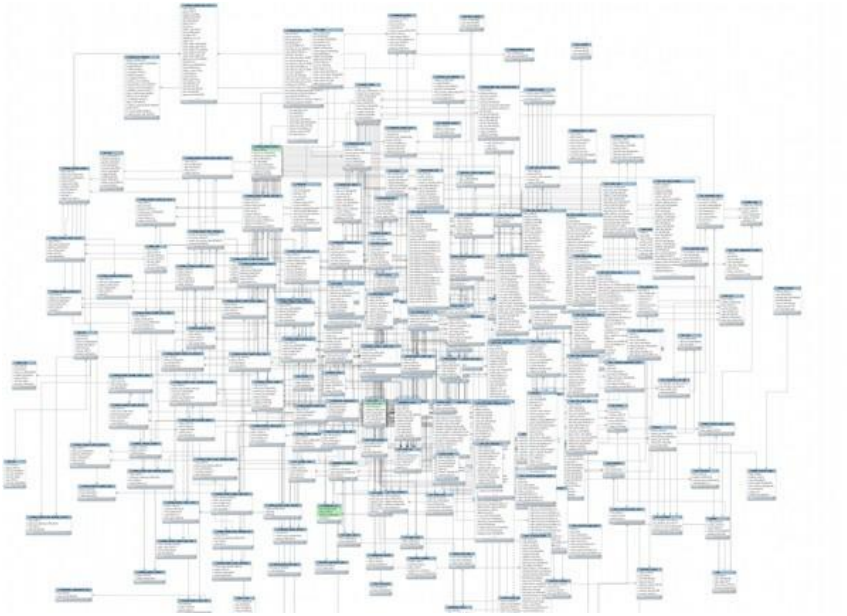
The modeled relationship between entities

Navigation Properties

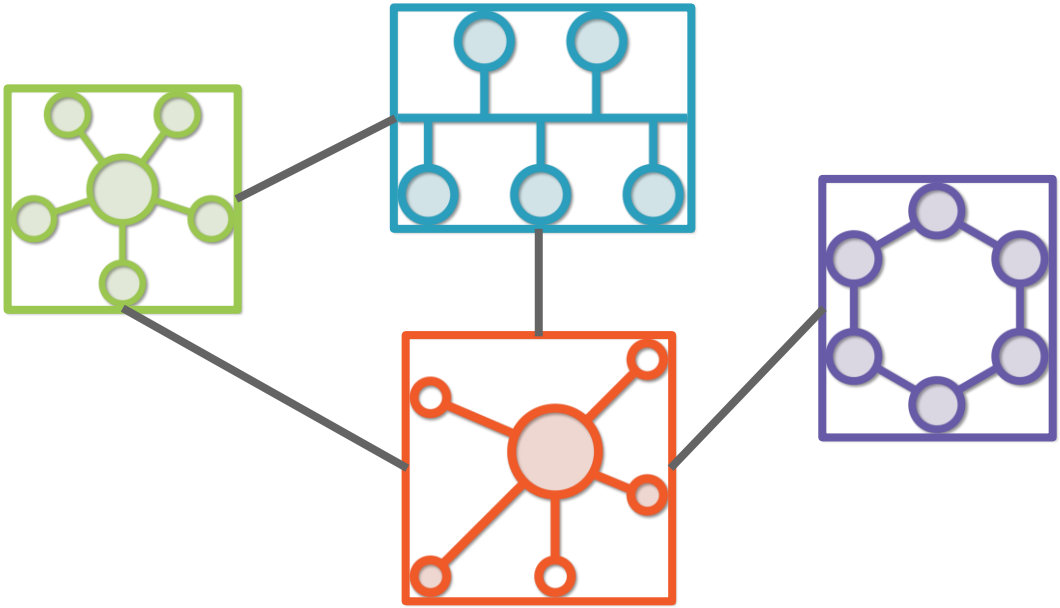
An ORM term to describe properties that reference related objects

Large systems often lead to complex data models

A Very Complex Model



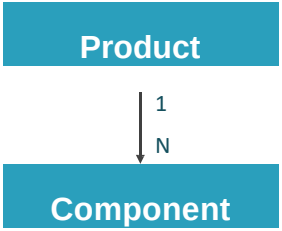
Smaller Models Can Still Interconnect



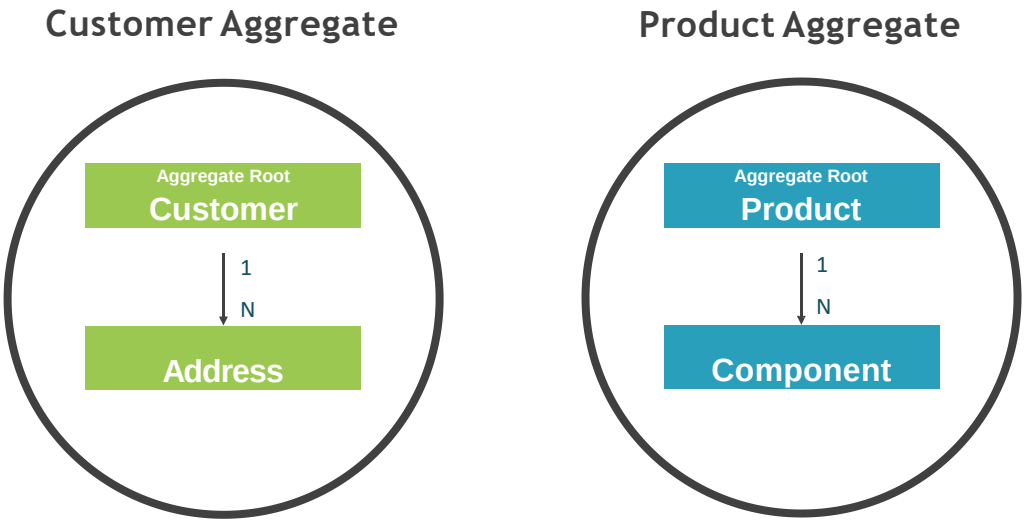
Aggregates

Customer Aggregate

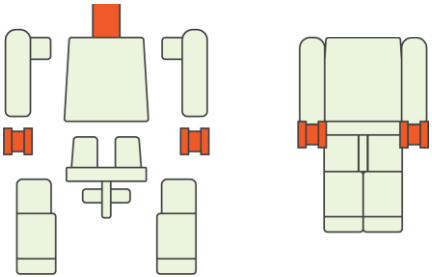
Product Aggregate

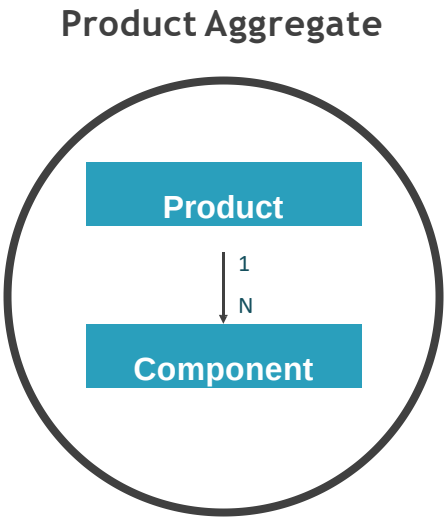


Aggregates in Our Model

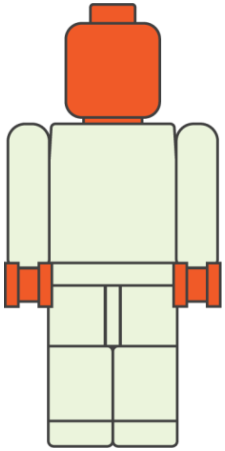


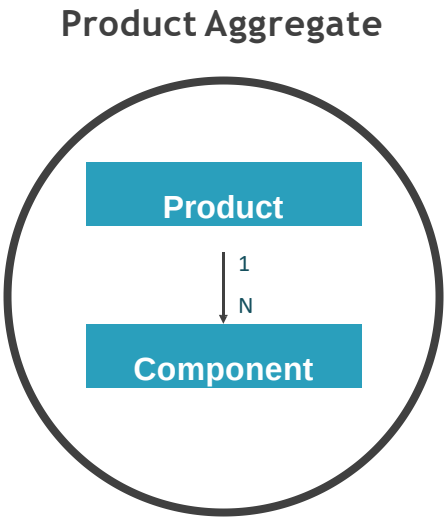
Aggregate Should Enforce Data Consistency



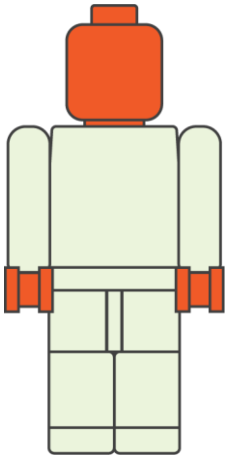


Aggregate Should Ensure a Product is Valid



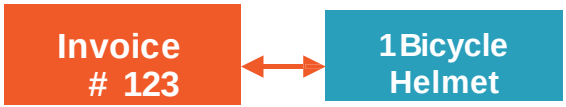


Deleting the Root Should Delete the Entire Aggregate

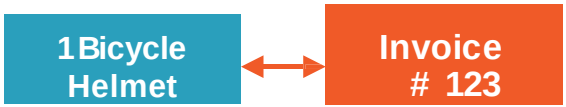


- ✓ 2 arms
- ✓ 2 legs
- ✓ 1 body
- ✓ 2 hands
- ✓ 1 pelvis
- ✓ 1 head

Bi-Directional Relationships



Bi-Directional Relationships

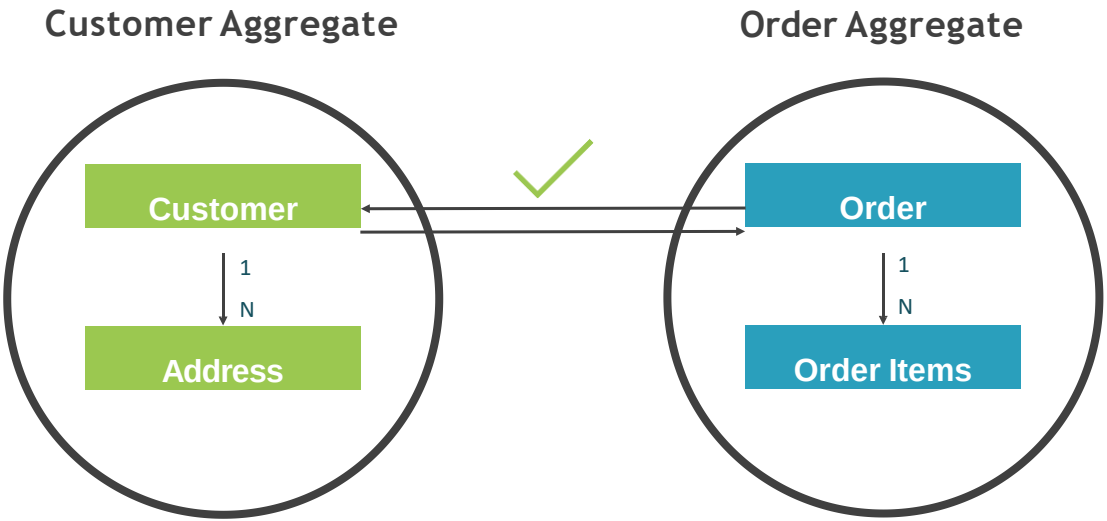


A bidirectional association means that both objects can be understood only together. When application requirements do not call for traversal in both directions, adding a traversal direction reduces interdependence and simplifies the design.

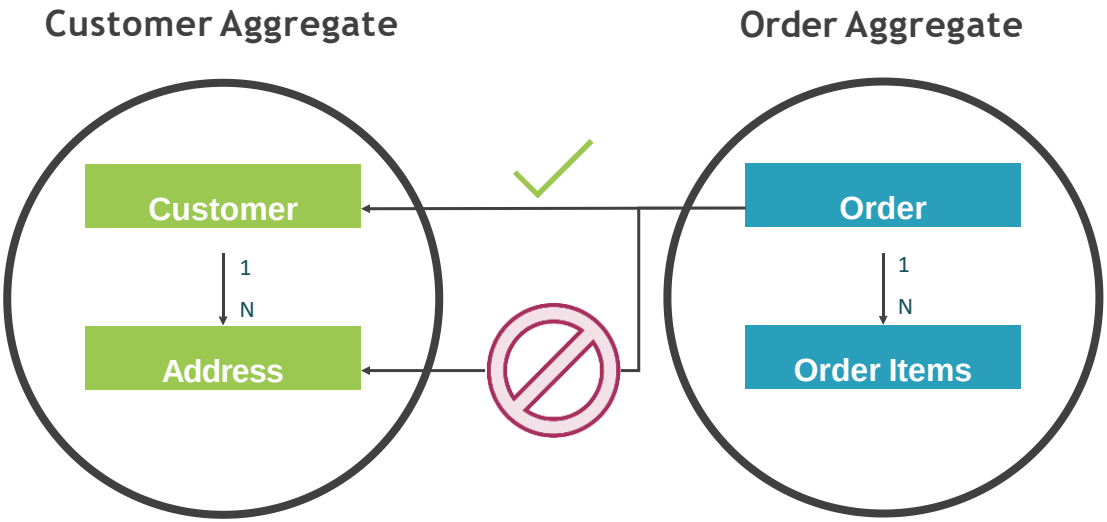
Eric Evans

External objects should interact with only the aggregate root.

Enforcing Boundaries Between Aggregates

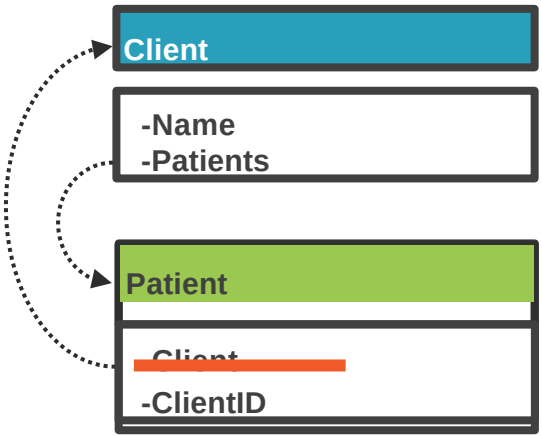


Enforcing Boundaries Between Aggregates



Object relationships are not the same as relationships between persisted data

Reference ID Value Can Save ORM Confusion



Repositories

Repository

A class that encapsulates the data persistence for an aggregate root

“A repository represents all objects of a certain type as a conceptual set… like a collection with more elaborate querying capability.”

Eric Evans
Domain-Driven Design

Provides common abstraction for persistence

Promotes separation of concerns

Communicates design decisions

Enables testability

Improved maintainability

Think of it as an in-memory collection

Implement a known, common access Interface

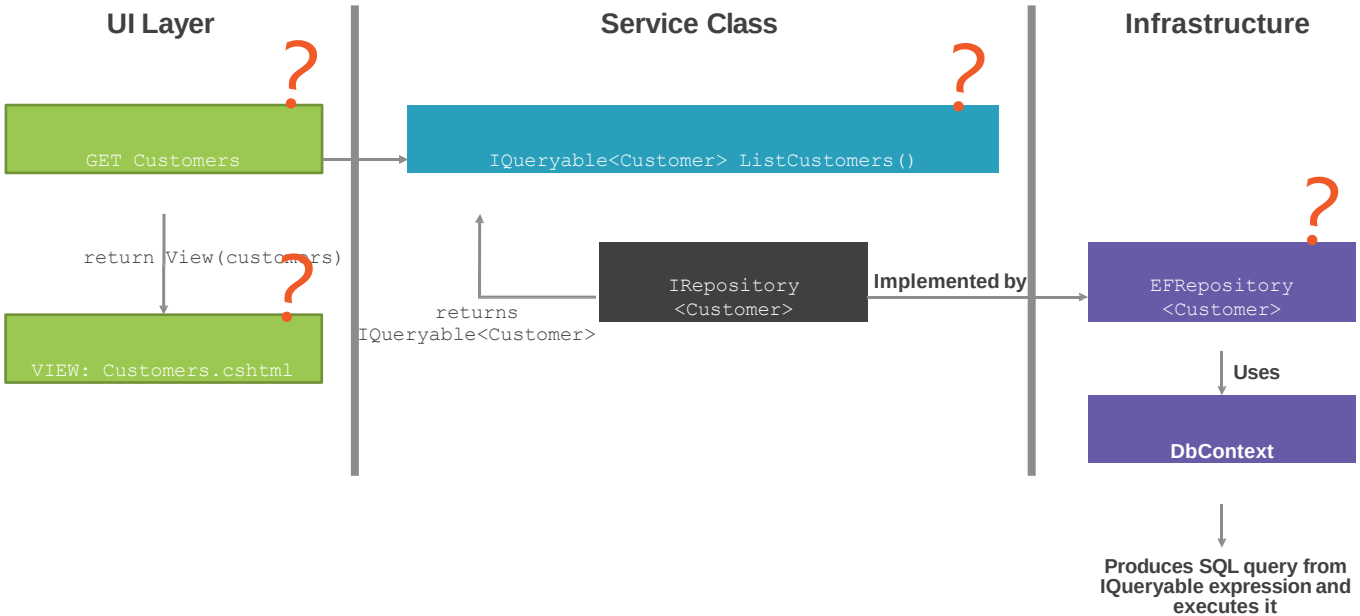
```
public interface IRepository<T>
{
  T GetById(int id);
  void Add(T entity);
  void Remove(T entity);

  void Update(T entity);
  IEnumerable<T> List();
}
```

General Repository Tips

Use repositories for aggregate roots only

Client focuses on model, repository on persistence

When is the Query Executed?



Domain Events

Domain Event

A class that captures the occurrence of an event in a domain object

Hollywood Principle

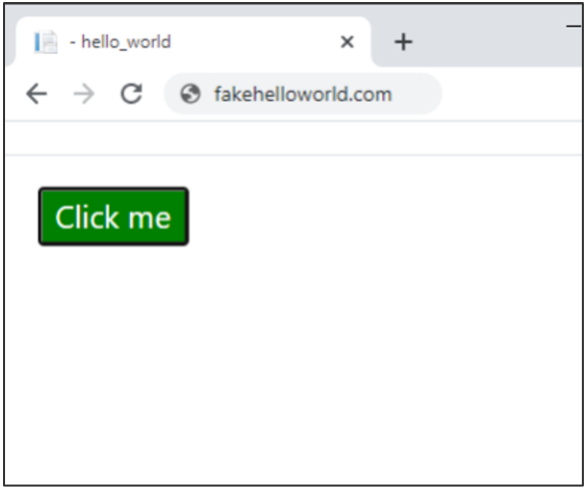
“Don’t call us, we’ll call you”

“Use a domain event to capture an occurrence of something that happened in the domain.”

Vaughn Vernon
Implementing Domain-Driven Design

```
<button id="b"
  style="background-color: blue;
         color:white"
  onclick="changeColor()"> Click me
</button>


<script>
  function changeColor() {
    document
     .getElementById("b");
     .style.backgroundColor = "green";
    }
</script>
```

Familiar Events



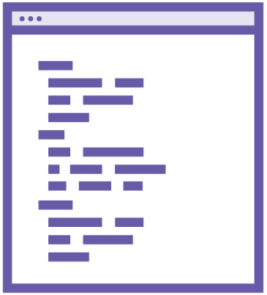
User Interface Events vs. Domain Events

User Interface Events

onclick

onkeypress

onsubmit



Domain Events

AppointmentScheduled

AppointmentConfirmed

ClientRegistered

Domain Events Pointers

When this happens, then something else should happen.
“If that happens…”, “Notify the user when…” , “Inform the user if…”

Domain events represent the past

Typically, they are immutable

Name the event using the bounded context’s ubiquitous language

Use the command name causing the event to be fired

Domain Event Examples

User Authenticated

Appointment Confirmed

Payment Received

Literature

Eric Evans:

Domain-Driven Design

Tackling Complexity in the Heart of Software

Thank You!