Domain-Driven ASP.NET Core Applications

Steve Smith

Ardalis.com

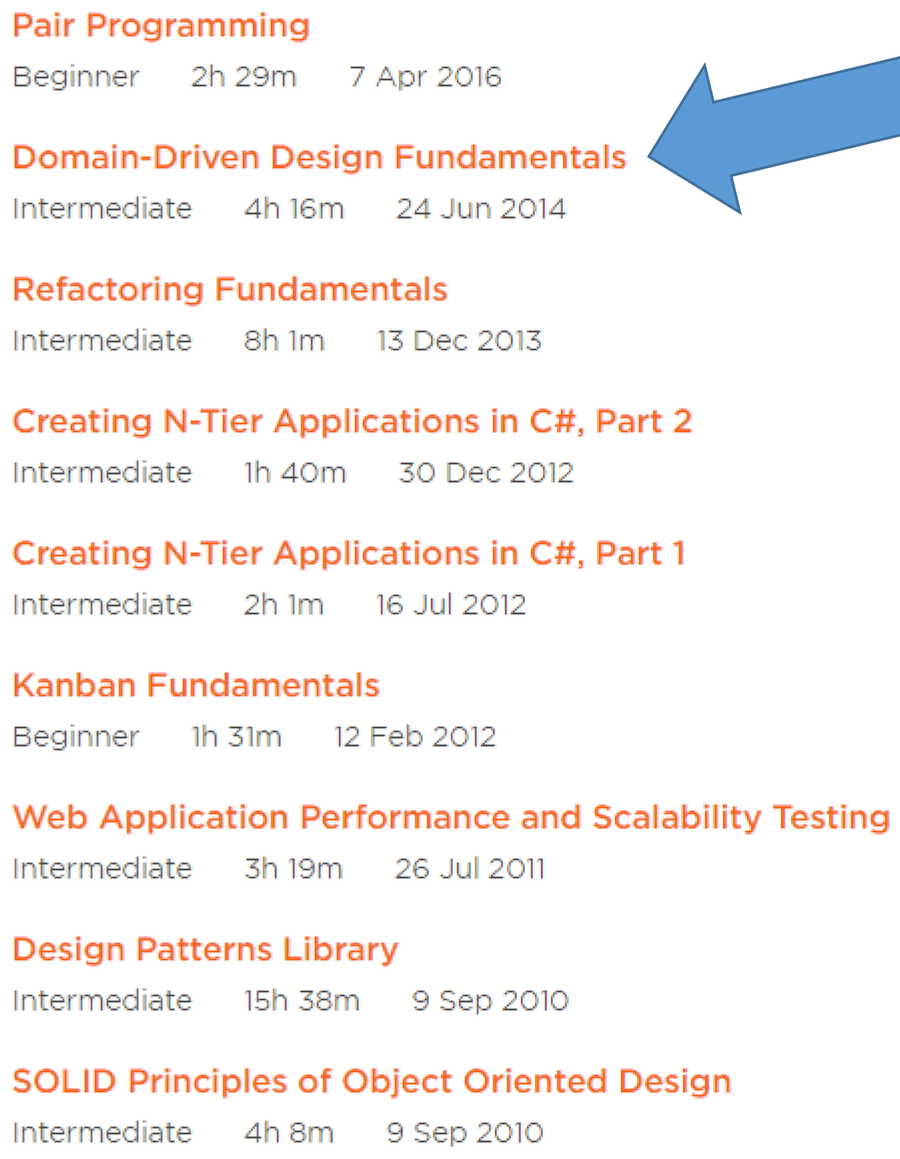
@ardalis

Tweet Away!

- Live Tweeting and Photos are encouraged
- Questions and Feedback are welcome
- Use #Codemash and/or #DDDASPNETCore

Pluralsight

I have some 1-month free passes; see me if you'd like one



Pair Programming	Beginner	2h 29m	7 Apr 2016
Domain-Driven Design Fundamentals	Intermediate	4h 16m	24 Jun 2014
Refactoring Fundamentals	Intermediate	8h 1m	13 Dec 2013
Creating N-Tier Applications in C#, Part 2	Intermediate	1h 40m	30 Dec 2012
Creating N-Tier Applications in C#, Part 1	Intermediate	2h 1m	16 Jul 2012
Kanban Fundamentals	Beginner	1h 31m	12 Feb 2012
Web Application Performance and Scalability Testing	Intermediate	3h 19m	26 Jul 2011
Design Patterns Library	Intermediate	15h 38m	9 Sep 2010
SOLID Principles of Object Oriented Design	Intermediate	4h 8m	9 Sep 2010

(Rough) Agenda - Morning

- Agenda (You Are Here)
- Logistics and Setup Questions
- Clean Architecture Overview
  - Lab 0: Make Sure Everything Works
- ASP.NET Core Overview
  - Minimal – more on this as we go and full-day workshop tomorrow
- DDD Overview: Entities, Value Objects, and Aggregates
- Lab 1 – Begin Building the Guestbook Application
- Repositories and Services
- Lab 2 – Add Persistence to Guestbook Application
  - Lunch!

(Rough) Agenda - Afternoon

- Discussion / Question and Answer / Finish Morning Labs
- Domain Events
- Lab 3 – Implementing Domain Events
- Testing and Testability
- Lab 4 – Add Unit and Integration Tests
- Specification Pattern
- Review and More Labs (Extra Credit / Homework)
  - Specification Lab
  - Caching / SignalR
  - Enforce Aggregates, Encapsulate Collections
  - Enforce Infrastructure Decoupling

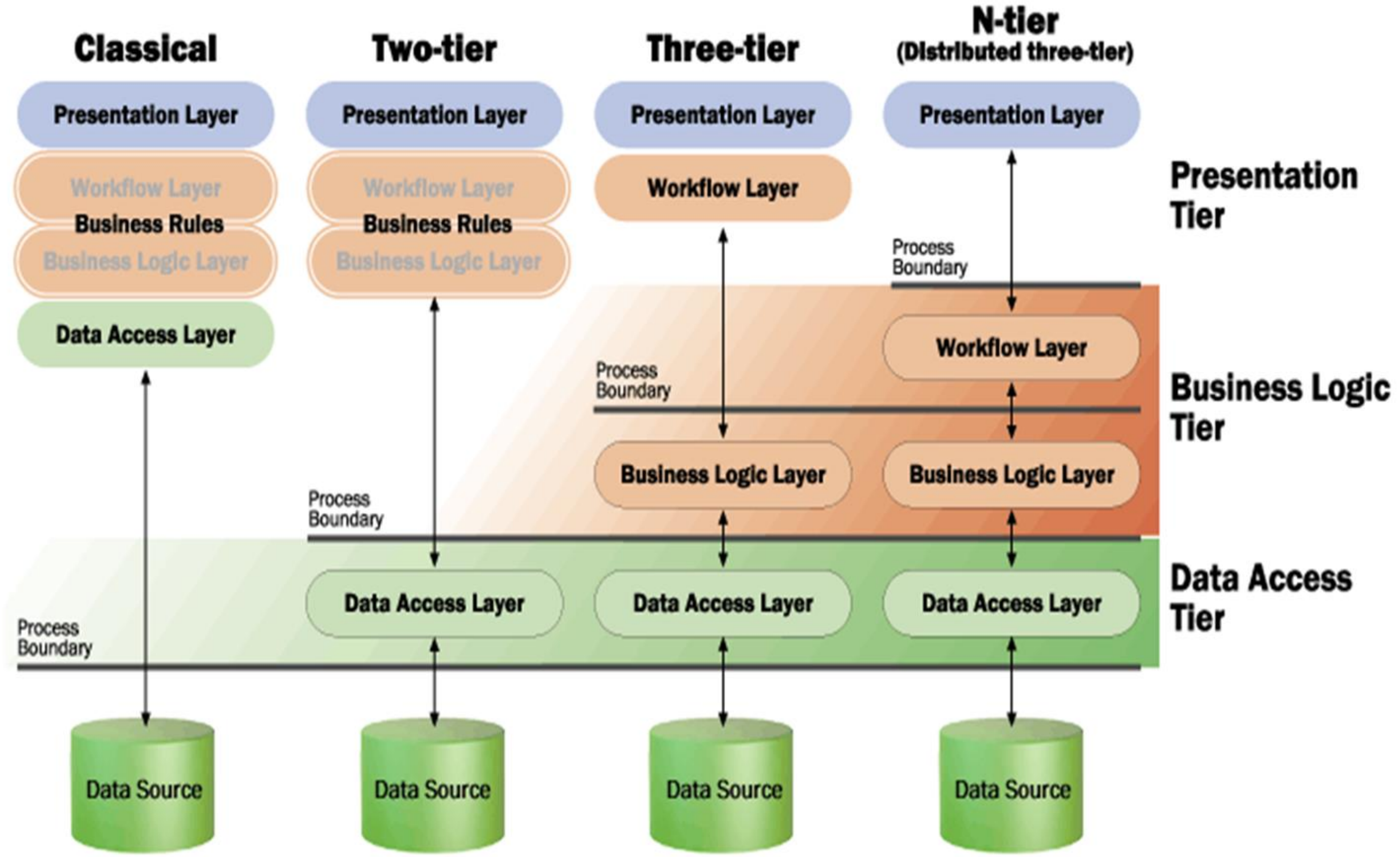
Demonstration

A Guestbook

Clean Architecture

AKA “Onion”, “Hexagonal”, “Ports and Adapters”

N Tier Diagram (circa 2001)



Benefits of N-Tier Design

- Improved Productivity via Reuse
- Improved Productivity via Team Segmentation
- Improved Maintainability
- Looser Coupling
- More Physical Deployment Options

Drawbacks and Risks of N-Tier Design

- Reduced Performance, especially when physically separated
- More Complex Design
- More Complex Deployment

Domain-Centric Design

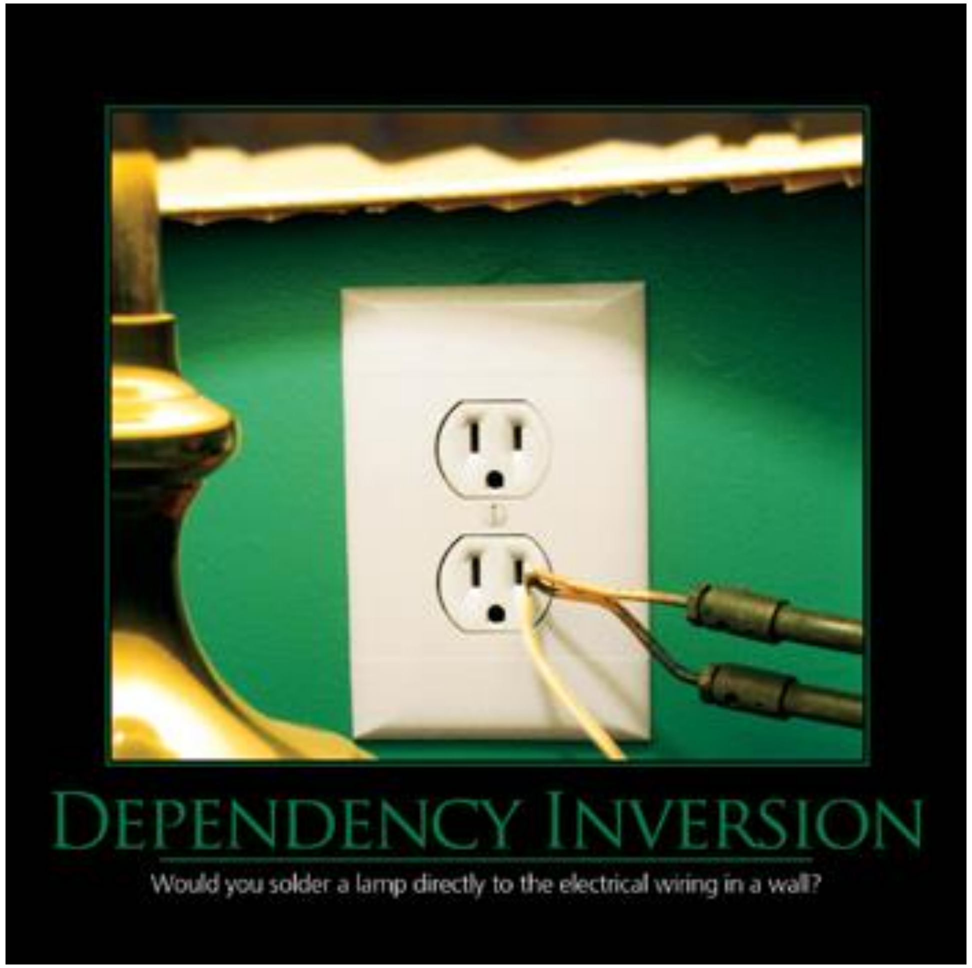
- The problem-space of your application is its *domain*
- The objects you design to *model* the domain are *domain objects*
  - Also referred to as *model objects* or *the model*
- Domain objects
  - Encapsulate application business logic and rules
  - Maintain any state required to do so
  - Do not depend on external *infrastructure* concerns
- More on Domain-Centric (or Domain *Driven*) Design later

Refactoring to Invert Dependencies

Guiding Principle: Dependency Inversion Principle (DIP)

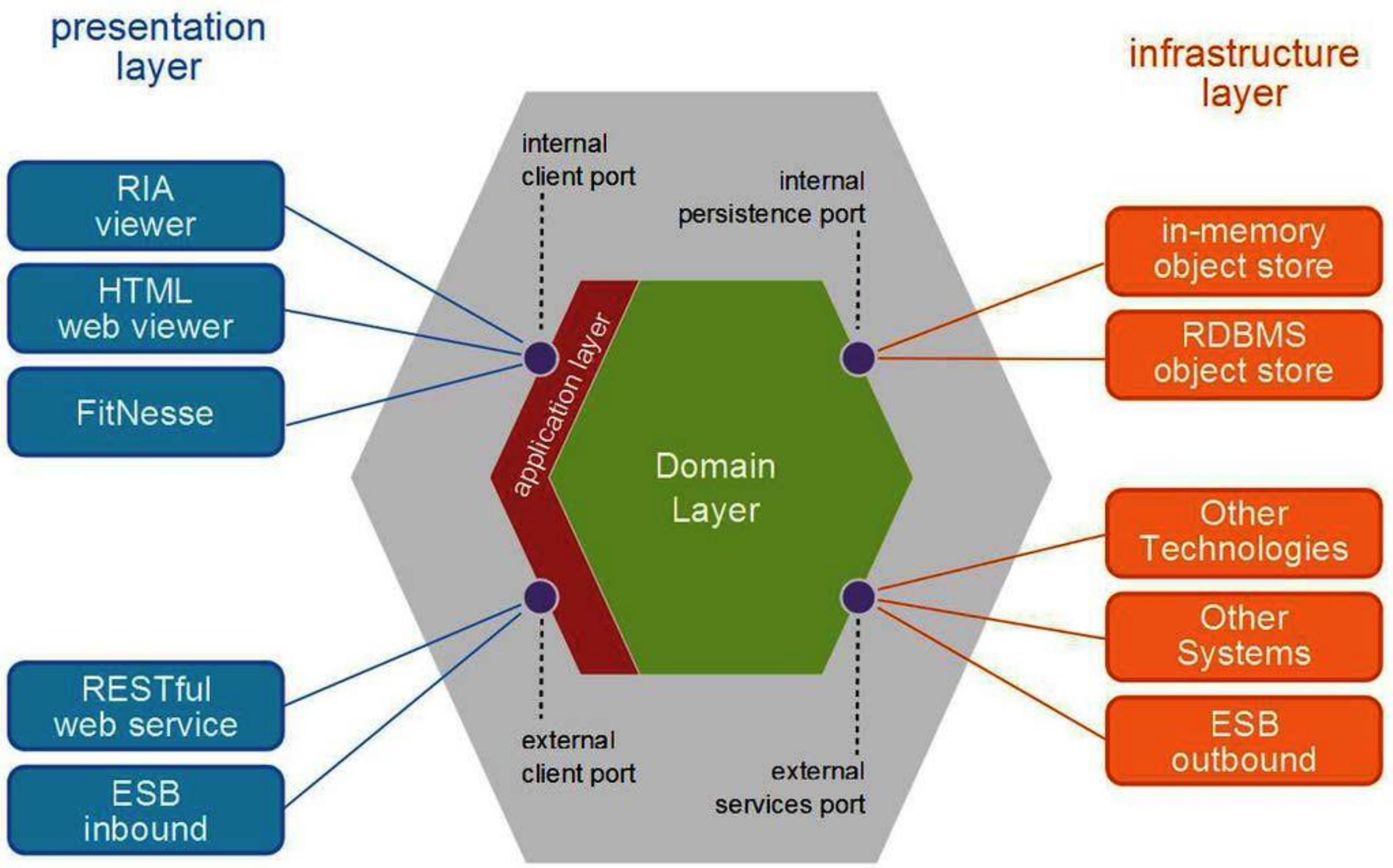
> *High-level modules should not depend on low-level modules. Both should depend on abstractions.*

> *Abstractions should not depend on details. Details should depend on abstractions.*


DEPENDENCY INVERSION
Would you solder a lamp directly to the electrical wiring in a wall?

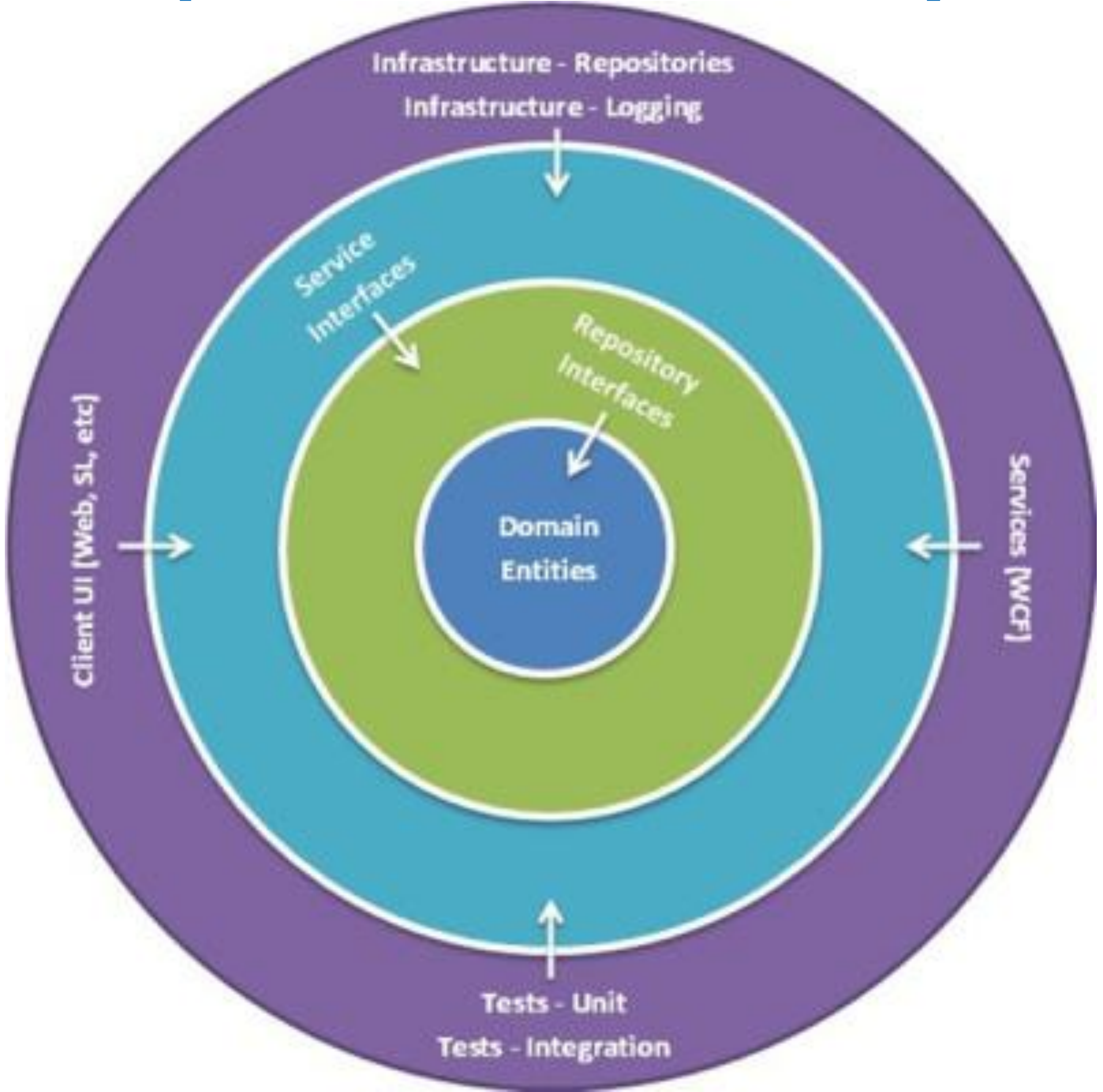
Refactoring Goal: Onion Architecture

a.k.a. Hexagonal Architecture, Ports and Adapters Architecture, Clean Architecture



http://pragprog.com/magazines/2009-12/going-naked

DIP Architecture (aka Ports and Adapters)



Clean DDD Architecture Principles

- All code depends on layers closer to center
- Domain Model at center (Core)
- Inner layers define interfaces; outer layers implement these interfaces
- Layered behavior around the Domain Model
- Infrastructure and UI concerns pushed to edge
  - Depend on Application Core, not vice versa

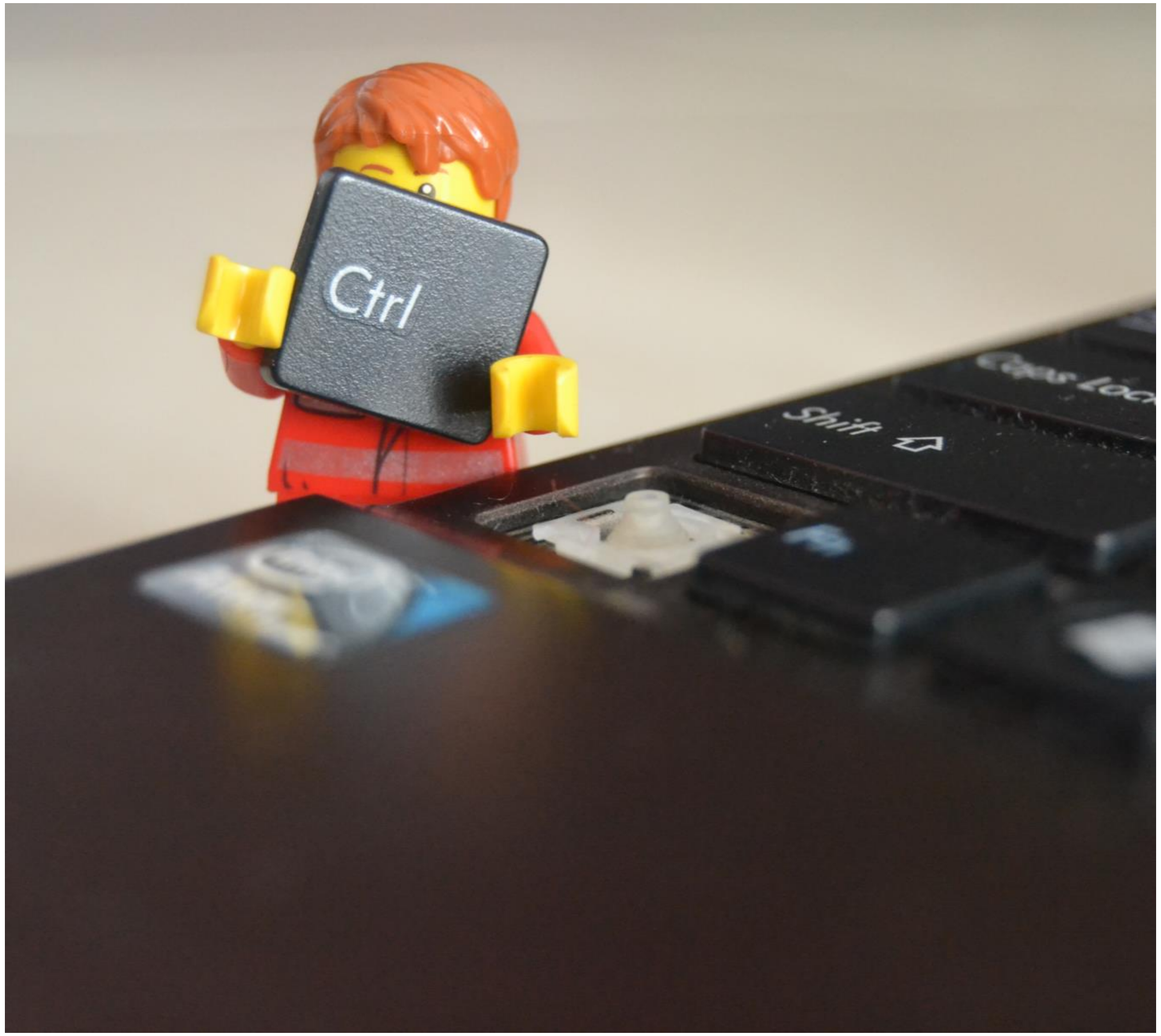
Demonstration

Lab 0

ASP.NET Core, MVC, and EF

Taking Control

With ASP.NET Core, YOU are in control



Define App Features in Startup

- Configure Application Services
  - Dependency Injection (DI) is built-in
  - Configure all App dependencies in the ConfigureServices method

- Configure Application Middleware
  - Define the application’s Request Pipeline
  - Very fine-grained
  - Large features, like MVC, encapsulated in helpers (e.g. app.UseMvc())

New Tools

- dotnet command line interface (CLI)
  - dotnet new
  - dotnet restore
  - dotnet build
  - dotnet run
  - dotnet test
- Visual Studio Code
  - Lightweight, cross-platform editor
  - Plugins for C#, HTML/CSS, Web Serving/IIS Express, and much more

Demonstration

A Simple MVC App

Domain-Driven Design

Overview

Why You Should Care about DDD

Principles & patterns to **solve difficult problems**

History of **success** with complex projects

Aligns with practices from experts' **experience**

Clear, testable code that represents the domain

Benefits of Domain Driven Design

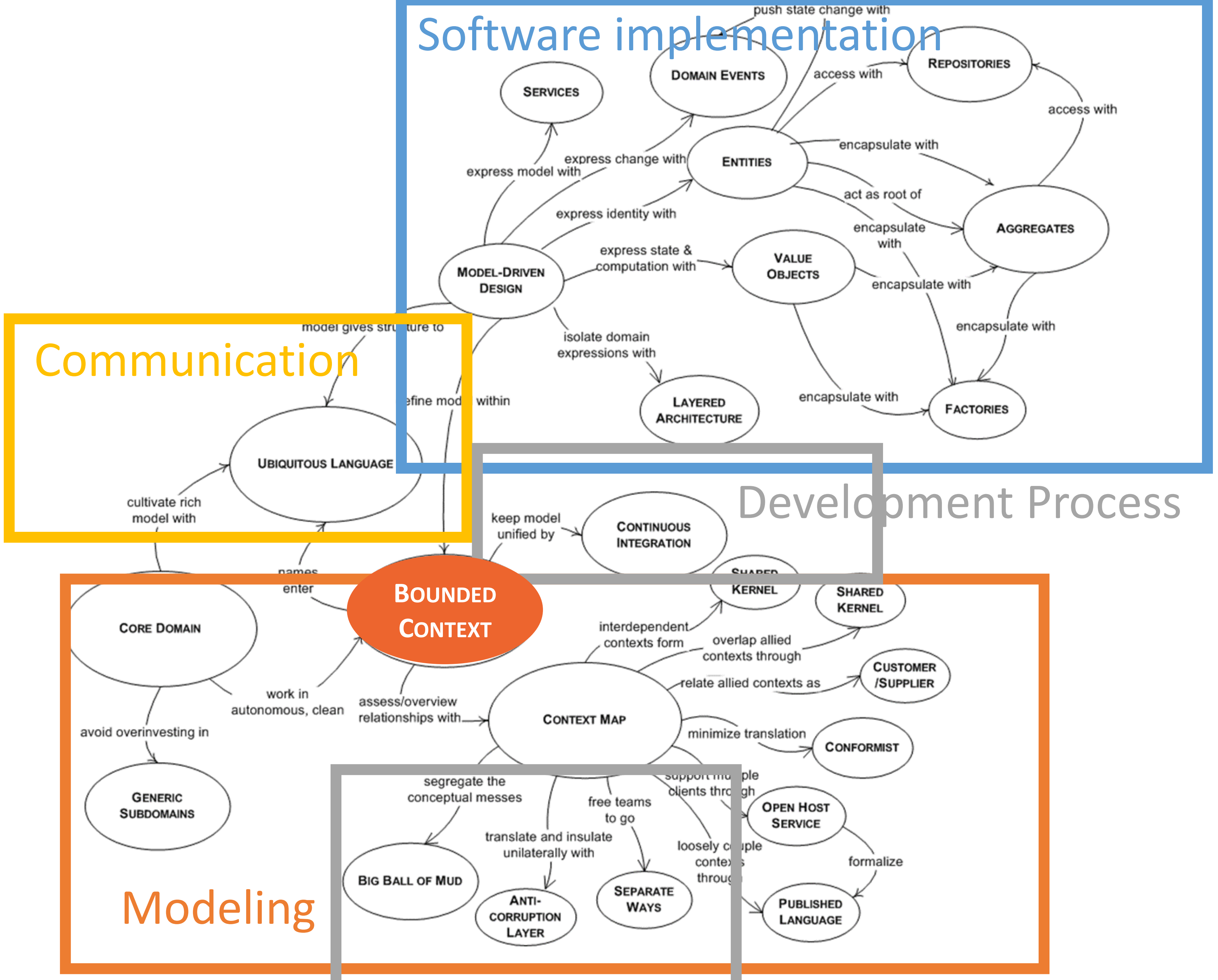
- Flexible
- Customer’s vision/perspective of the problem
- Path through a very complex problem
- **Well-organized and easily tested code**
- **Business logic lives in one place.**
- Many great patterns to leverage

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

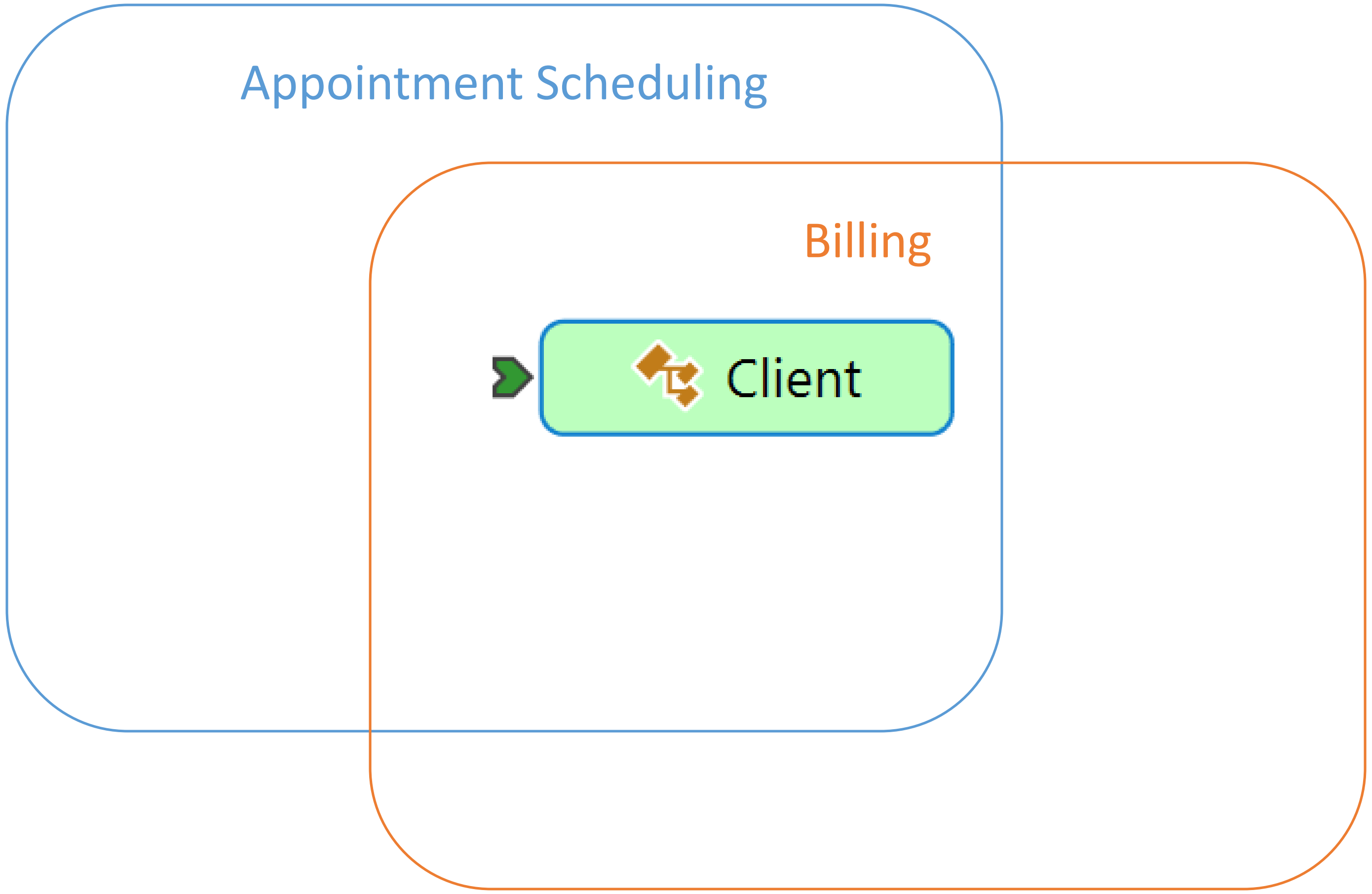
Drawbacks of DDD

- Time and Effort
  - Discuss & model the problem with domain experts
  - Isolate domain logic from other parts of application
- Learning curve (why you’re watching this course)
  - New principles
  - New patterns
  - New processes
- Only makes sense when there is complexity in the problem
  - Not just CRUD or data-driven applications
  - Not just technical complexity without business domain complexity
- Team or Company Buy-In to DDD


Software implementation
Communication
Development Process
Modeling
Services
Domain Events
Repositories
Entities
Aggregates
Value Objects
Model-Driven Design
Layered Architecture
Factories
Ubiquitous Language
Continuous Integration
Bounded Context
Shared Kernel
Shared Kernel
Core Domain
Customer /Supplier
Context Map
Conformist
Generic Subdomains
Open Host Service
Big Ball of Mud
Anti-corruption Layer
Separate Ways
Published Language
push state change with
access with
access with
express model with
express change with
encapsulate with
express identity with
act as root of
encapsulate with
express state & computation with
encapsulate with
encapsulate with
isolate domain expressions with
encapsulate with
model gives structure to
define model within
cultivate rich model with
keep model unified by
names enter
interdependent contexts form
overlap allied contexts through
relate allied contexts as
work in autonomous, clean
assess/overview relationships with
avoid overinvesting in
minimize translation
support multiple clients through
segregate the conceptual messes
free teams to go
translate and insulate unilaterally with
loosely couple contexts through
formalize

Domain-Driven Design

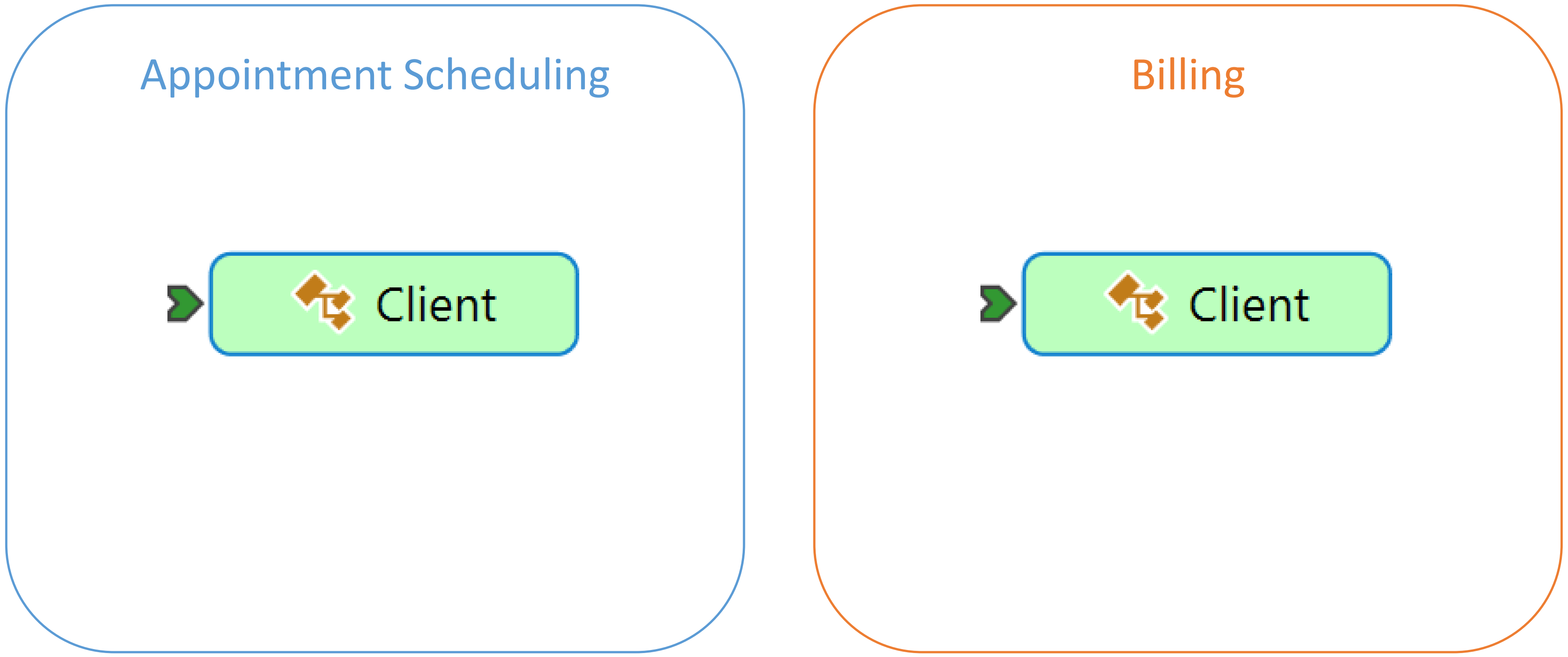
Bounded Contexts


Appointment Scheduling
Billing
Client

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



What’s important to one bounded context may not be as important in another.

—Steve Smith

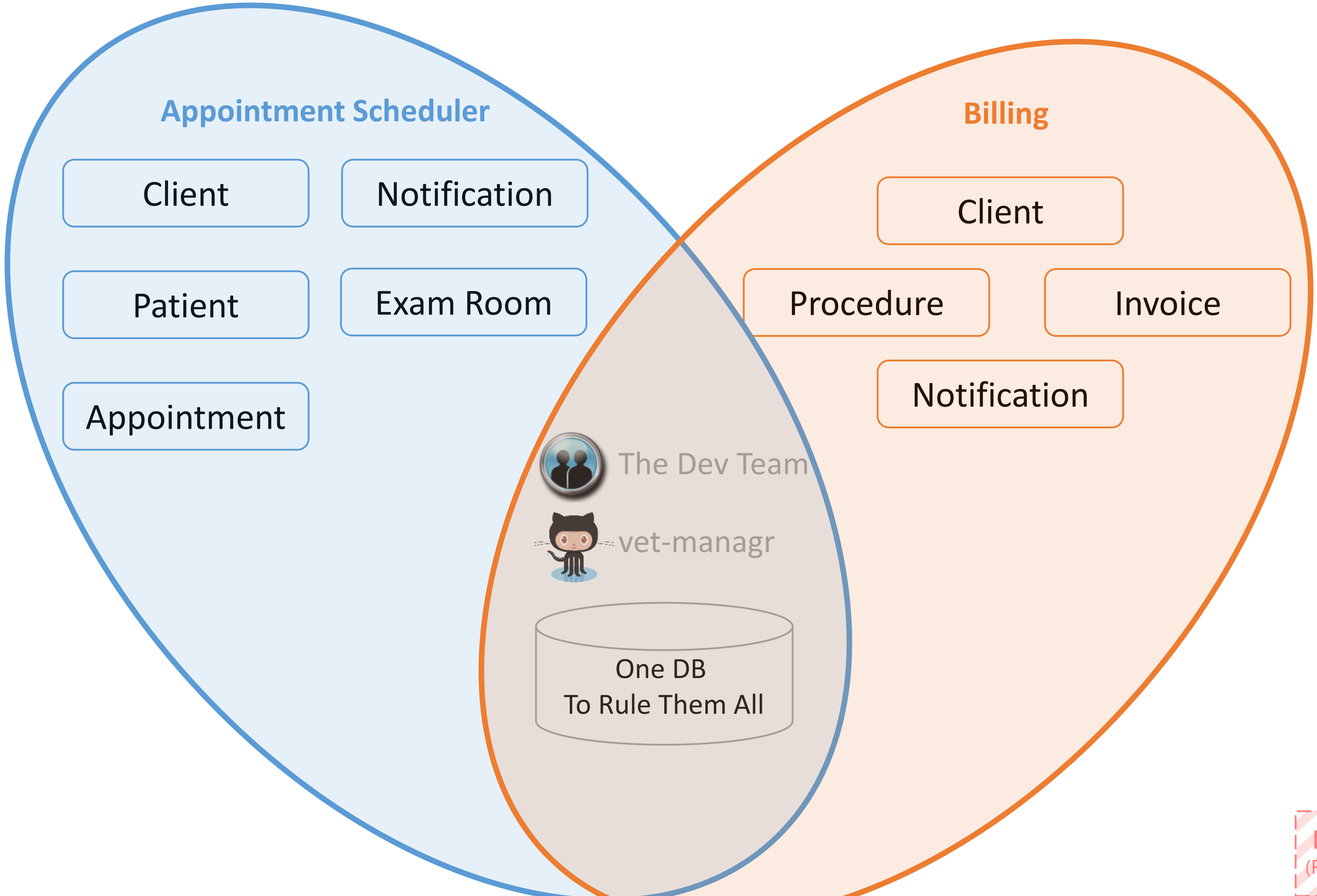
CAUTION
THIS SIGN HAS
SHARP EDGES
DO NOT TOUCH THE EDGES OF THIS SIGN
ALSO, THE BRIDGE IS OUT AHEAD

Bounded Context

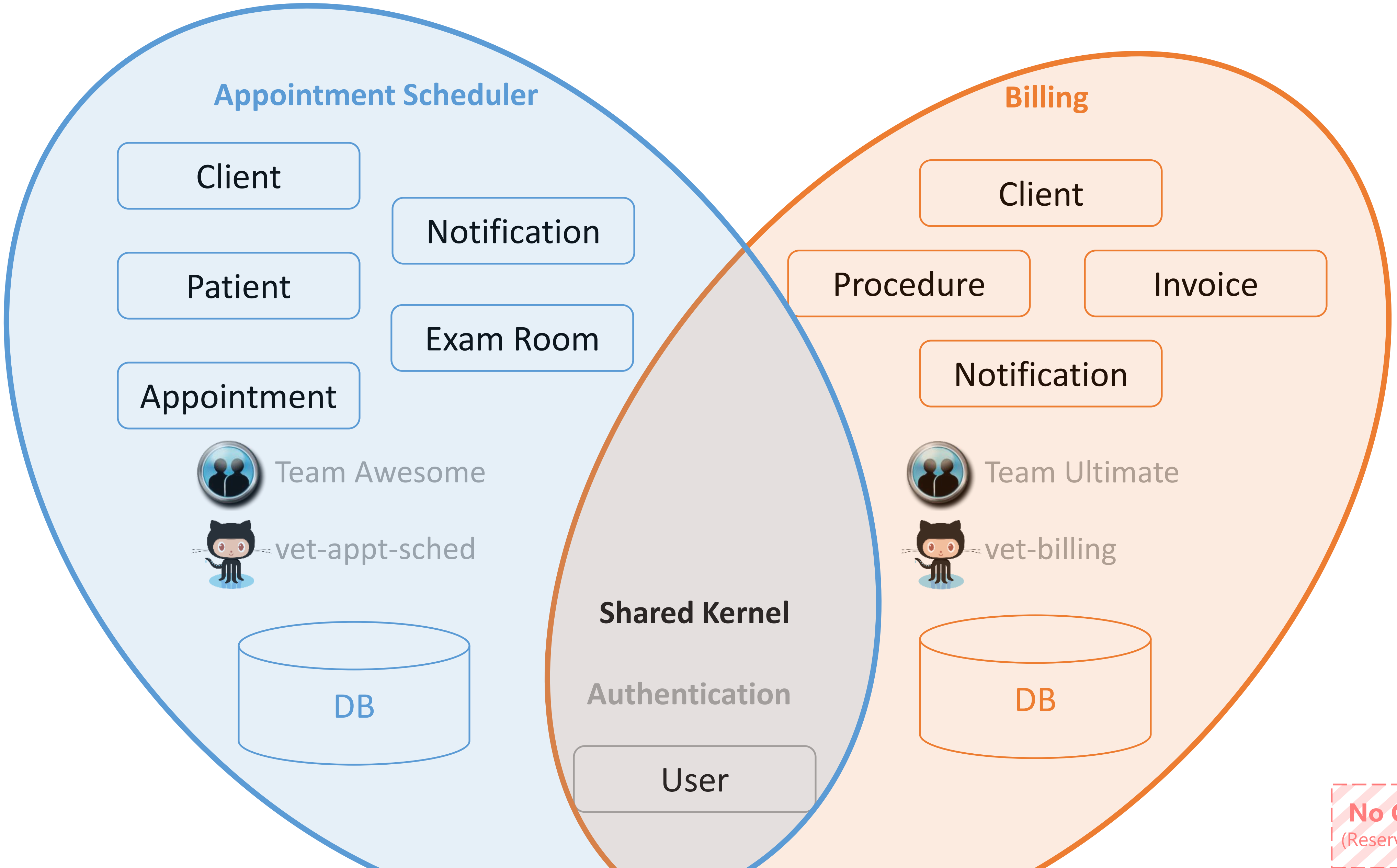
- Relates to the Solution you are building

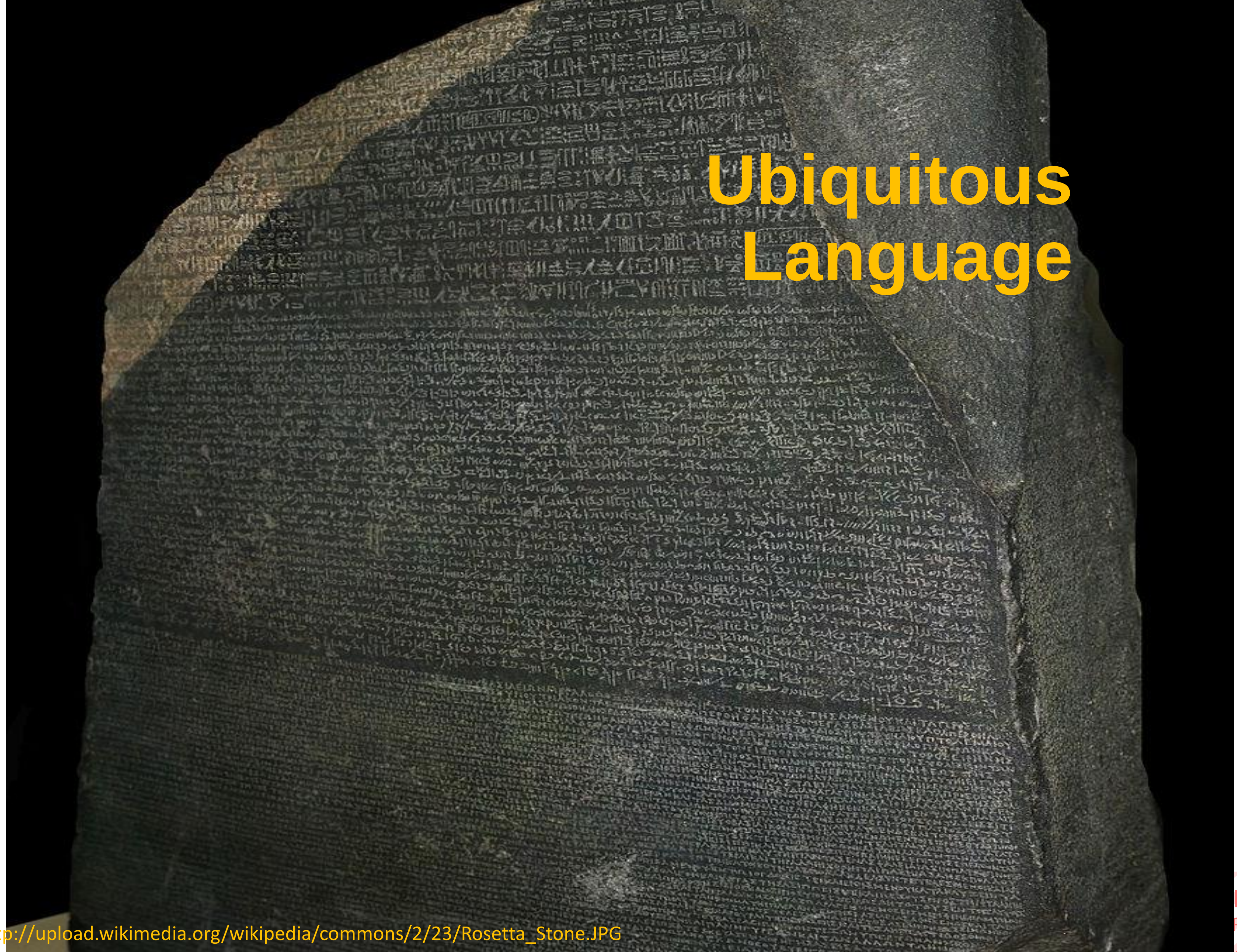
- The domain represents the problem your solution is solving
  - Core Domain – the business's primary focus/industry
  - Sub-Domain – a particular area within a larger domain. Maps to a software solution and bounded context.

Context Maps



Context Maps





Ubiquitous Language

http://upload.wikimedia.org/wikipedia/commons/2/23/Rosetta_Stone.JPG

A project faces serious problems when its language is fractured.

—Eric Evans

Ubiquitous Language

For a single Bounded Context

Used throughout that context, from conversations to code

Glossary of Terms

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.

Sub-Domains

Separate applications or features your software must support or interact with.

Bounded Context

A specific responsibility, with explicit boundaries that separate it from other parts of the system. Usually corresponds to a specific Sub-Domain.

Glossary of Terms

Context Mapping

The process of identifying bounded contexts and their relationships to one another.

Shared Kernel

Part of the model that is shared by two or more teams, who agree not to change it without collaboration

Ubiquitous Language

A language using terms from the domain model that programmers and domain experts use to discuss the system within a particular bounded context.

Domain-Driven Design

The Domain Model

Behaviors

Schedule an appointment for a checkup
Note a pet's weight
Request lab work
Notify pet owner of vaccinations due
Accept a new patient
Book a room

Not Attributes

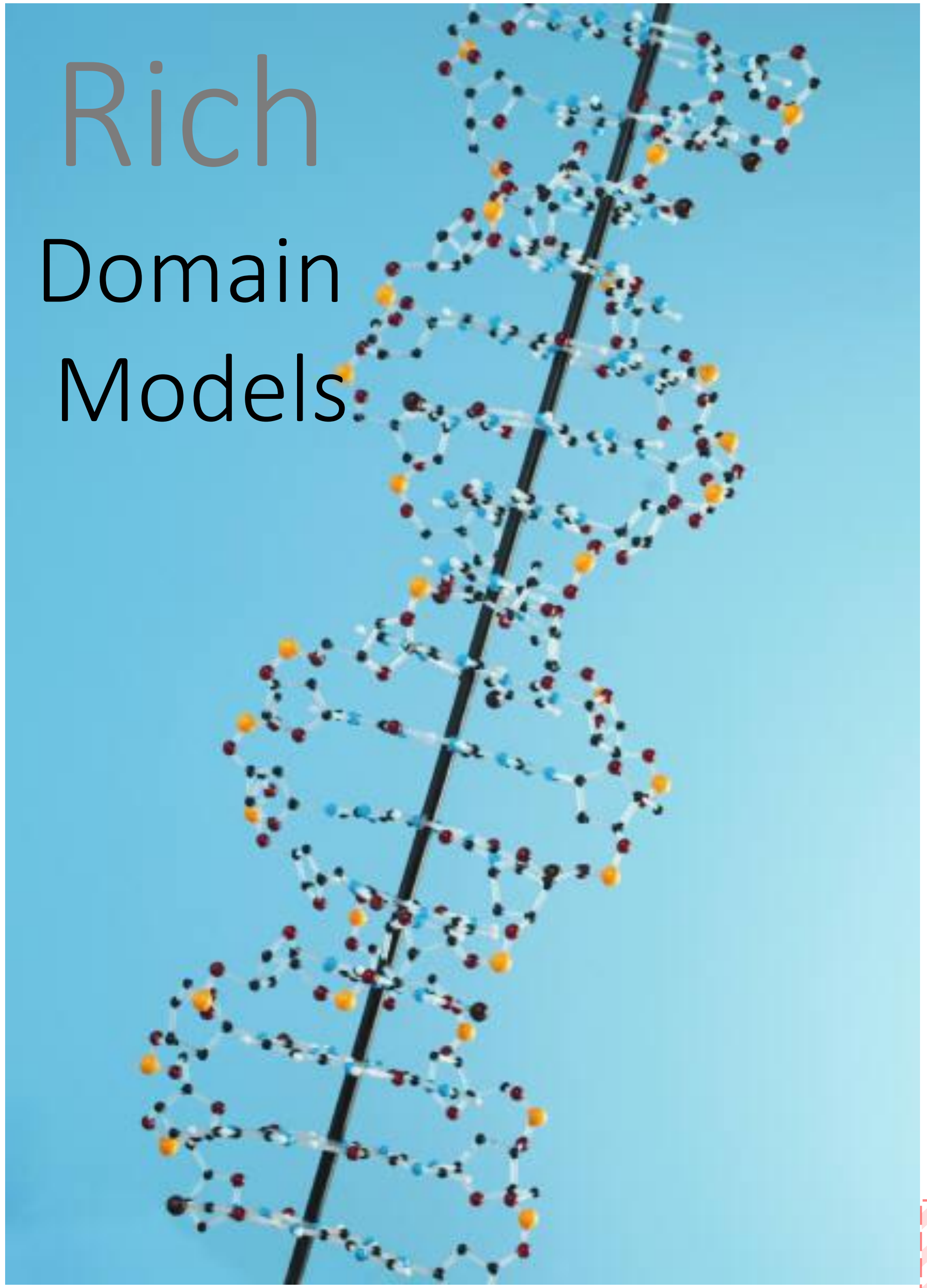
Appointment.Time Pet.Name Owner.Telephone Room.Number

Anemic

Domain Models

Rich

Domain Models

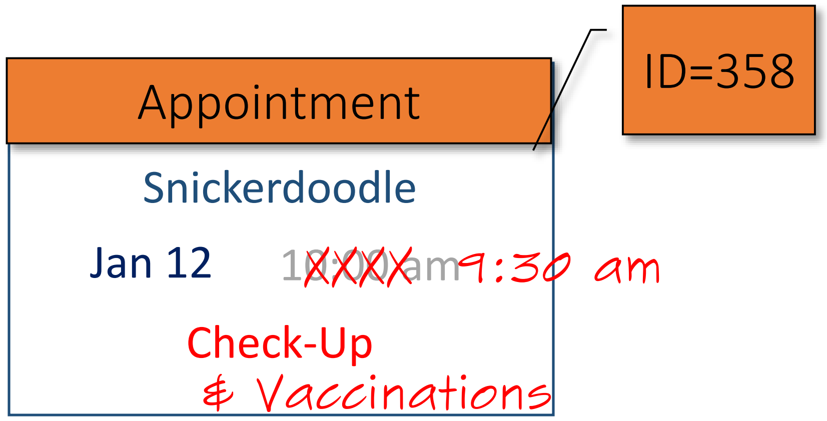


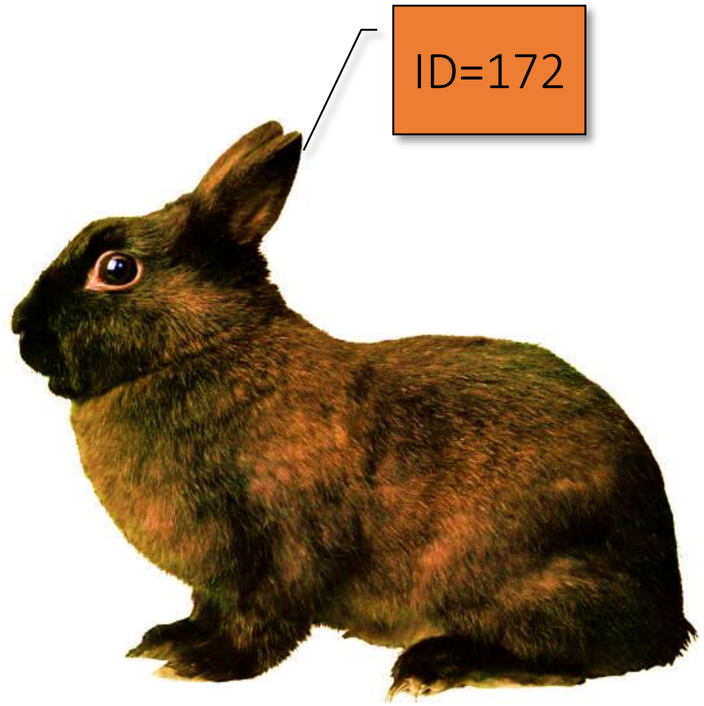
Many objects are not fundamentally defined by their attributes, but rather by a **thread of continuity** and **identity**.

Entity

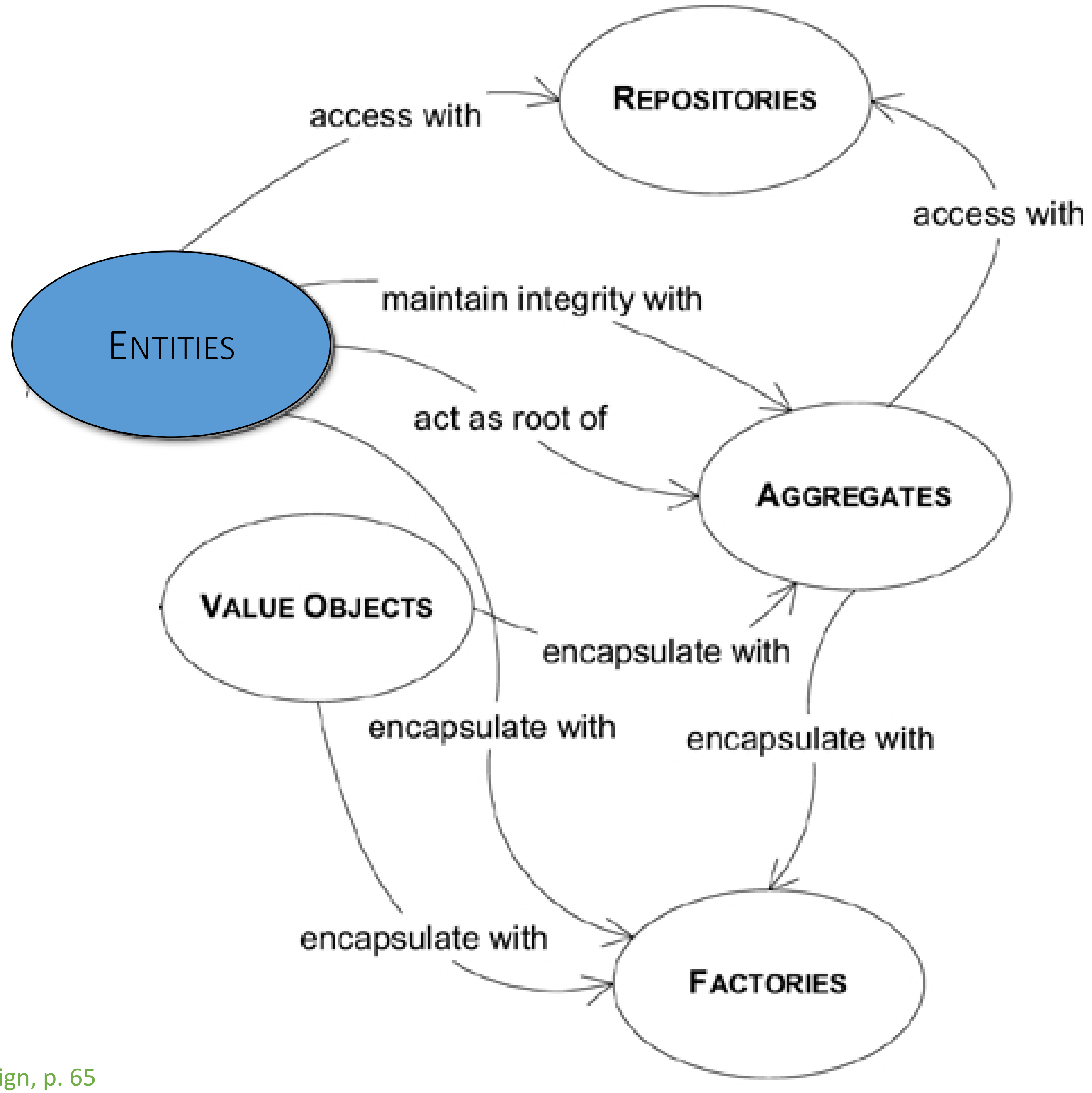
—Eric Evans
Domain-Driven Design

Entities Have Identity & Are Mutable

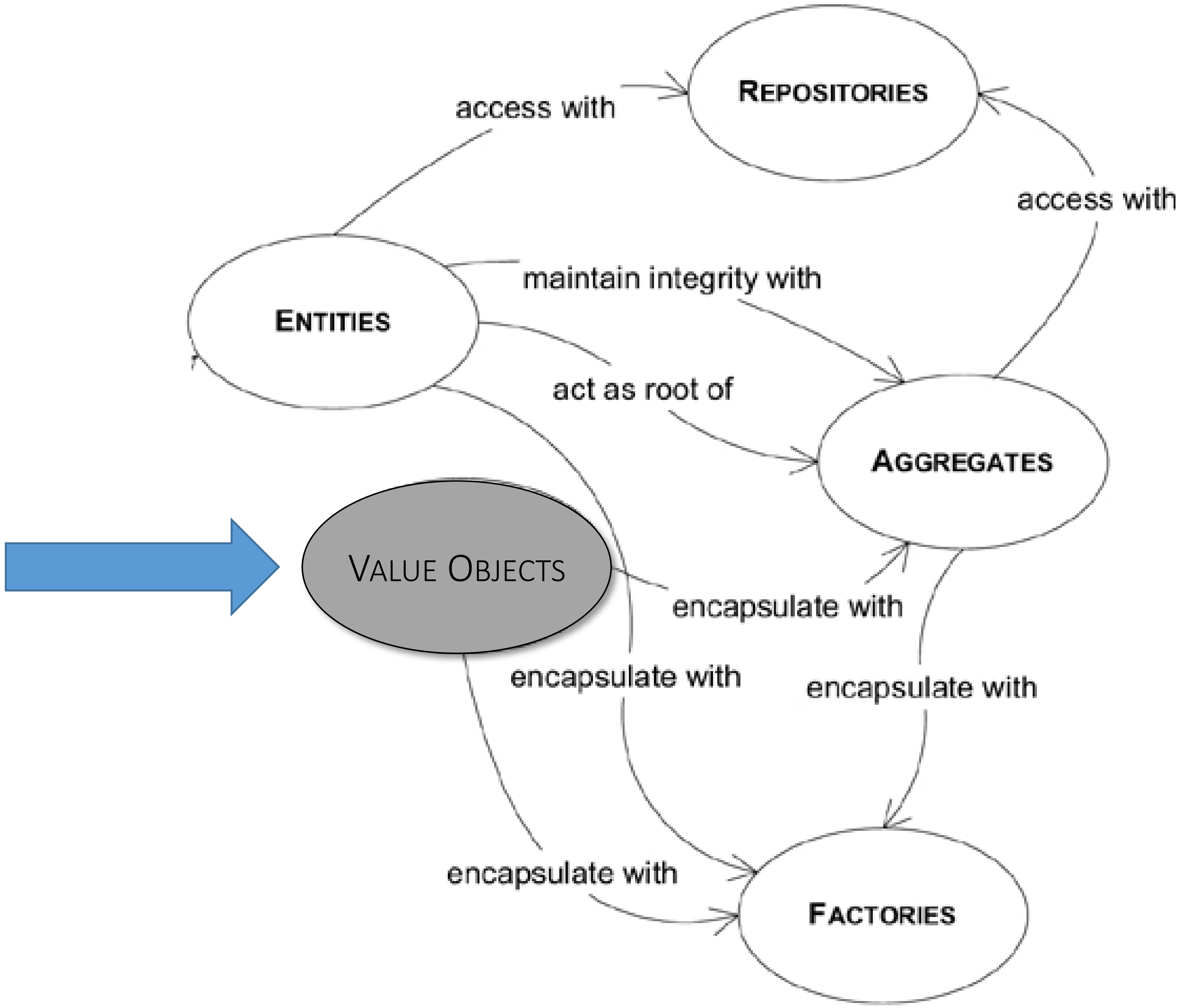




Entities in the Navigation Map



Evans,
Domain-Driven Design, p. 65


access with
REPOSITORIES
access with
maintain integrity with
ENTITIES
act as root of
AGGREGATES
VALUE OBJECTS
encapsulate with
encapsulate with
encapsulate with
encapsulate with
FACTORIES

Value Object

Measures, quantifies, or describes a thing in the domain.

Identity is based on composition of values

Immutable

Compared using all values

No side effects

Company Worth: $50,000,000

$ 50,000,000

```
Company (Entity)
    ID (guid): 9F63CE8D-9F1E-45E0-85AB-C098CC15F8E6
    Worth Unit (string): "US Dollar"
    Worth Amount (decimal): 50000000
```

```
Company (Entity)
    ID (guid): 9F63CE8D-9F1E-45E0-85AB-C098CC15F8E6
    Worth { Worth (Value Object)
                Monetary Unit (string) "U.S. Dollar"
                Amount (decimal): 50000000
```

Source:http://fit.c2.com/wiki.cgi?WholeValue (Ward Cunningham)

Patient Appointment

10:00 am Jan 4, 2014 – 11:00 am Jan 4, 2014

Staff Meeting

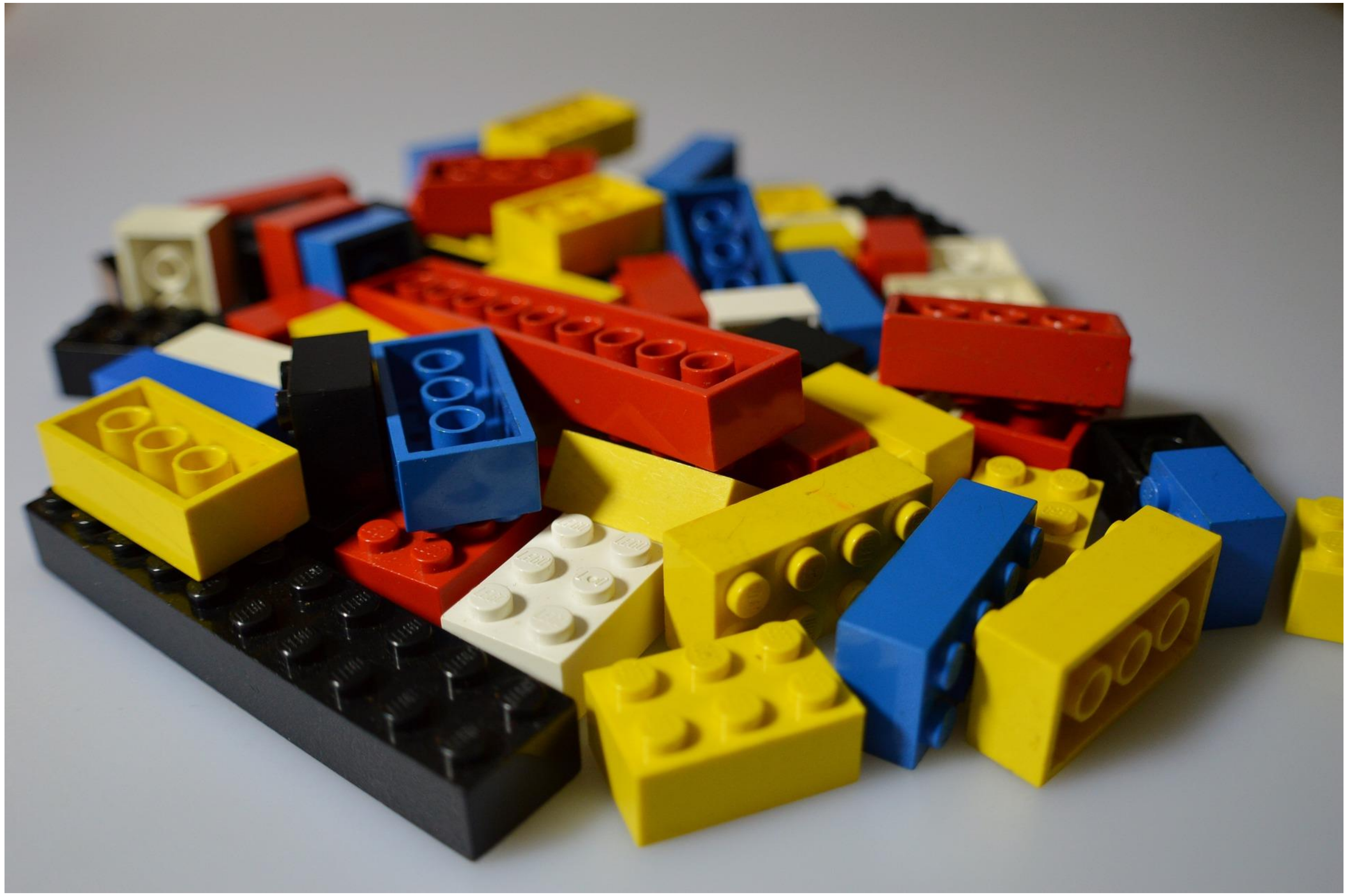
2:00 pm Feb 1, 2014 – 3:15 pm Feb 1, 2014

```
public class DateTimeRange
{
    public DateTimeRange(DateTime start, DateTime end)
   {
        Start=start;
        End=end;
    }
    public DateTime Start { get; private set; }
    public DateTime End { get; private set; }
    …
}
```

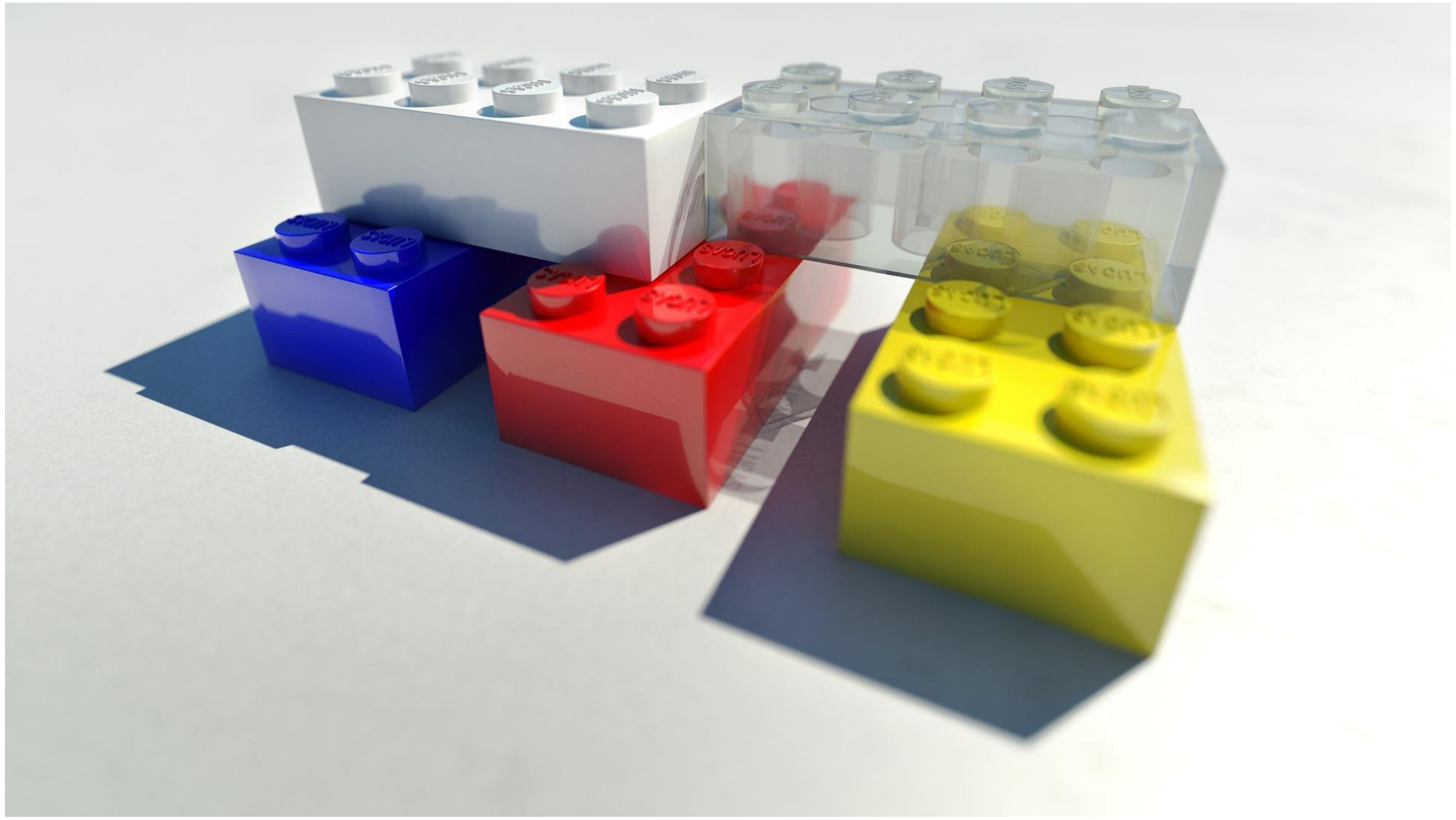
It may surprise you to learn that we should strive to model using Value Objects instead of Entities wherever possible. Even when a domain concept must be modeled as an Entity, the Entity's design should be biased toward serving as a value container rather than a child Entity container.

—Vaughn Vernon
Implementing Domain Driven Design

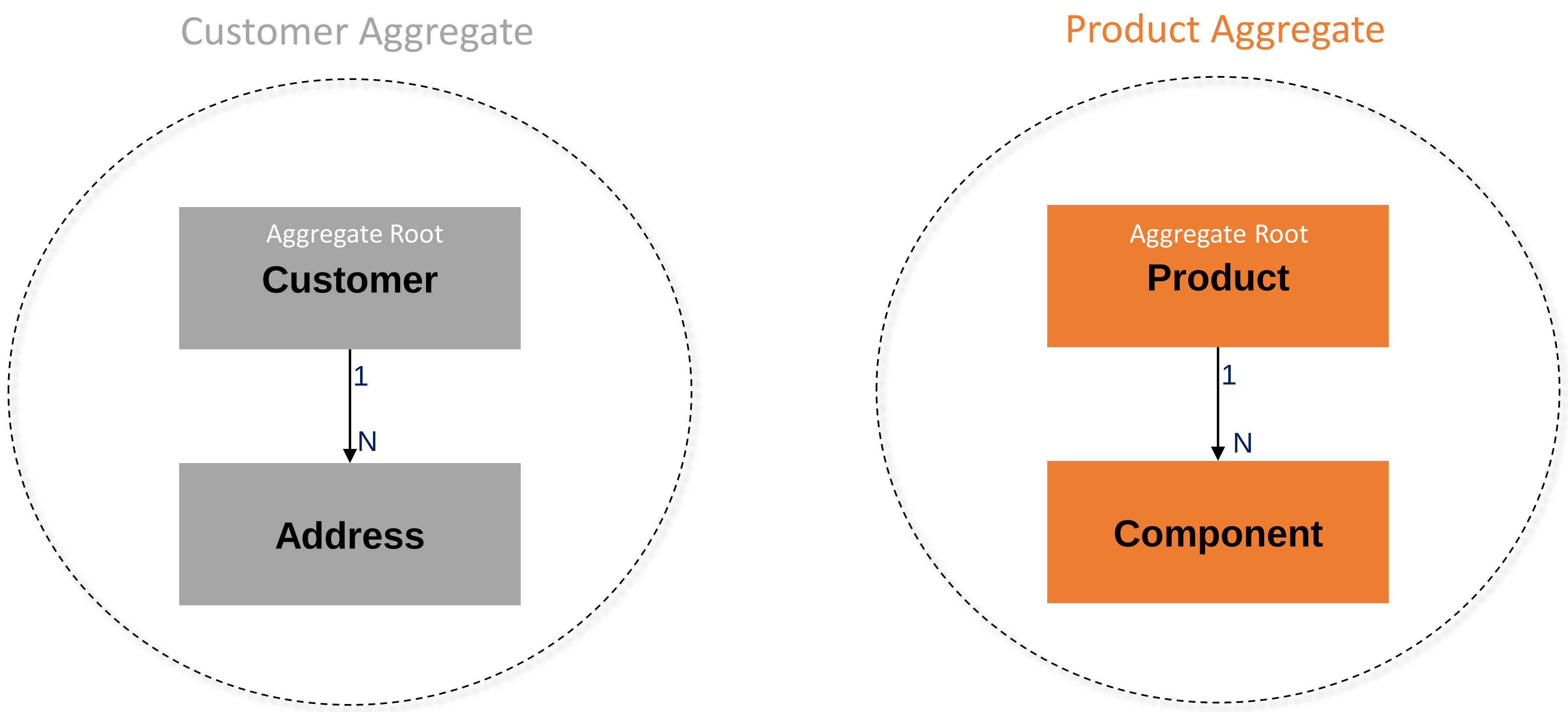
Aggregates



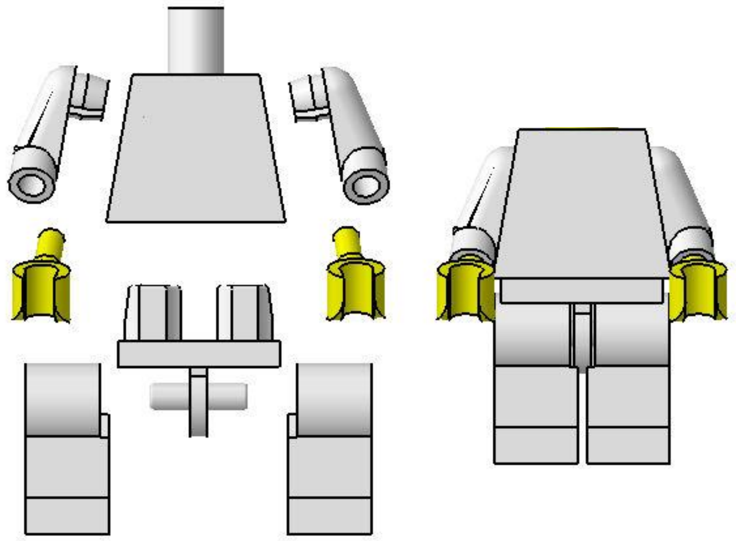
Aggregates

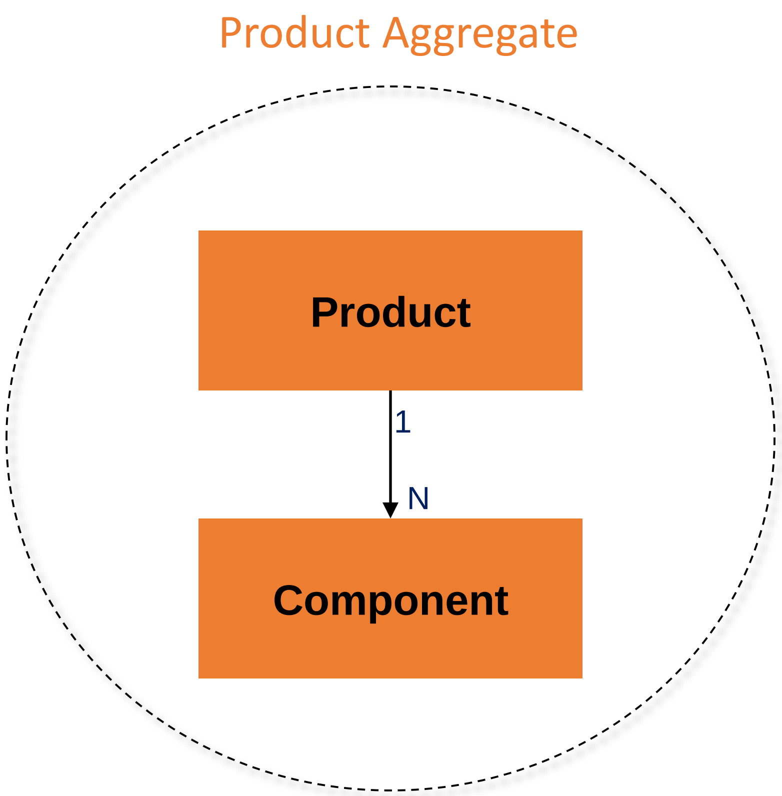


Aggregates


Customer Aggregate
Aggregate Root
Customer
1
N
Address
Product Aggregate
Aggregate Root
Product
1
N
Component

Aggregates

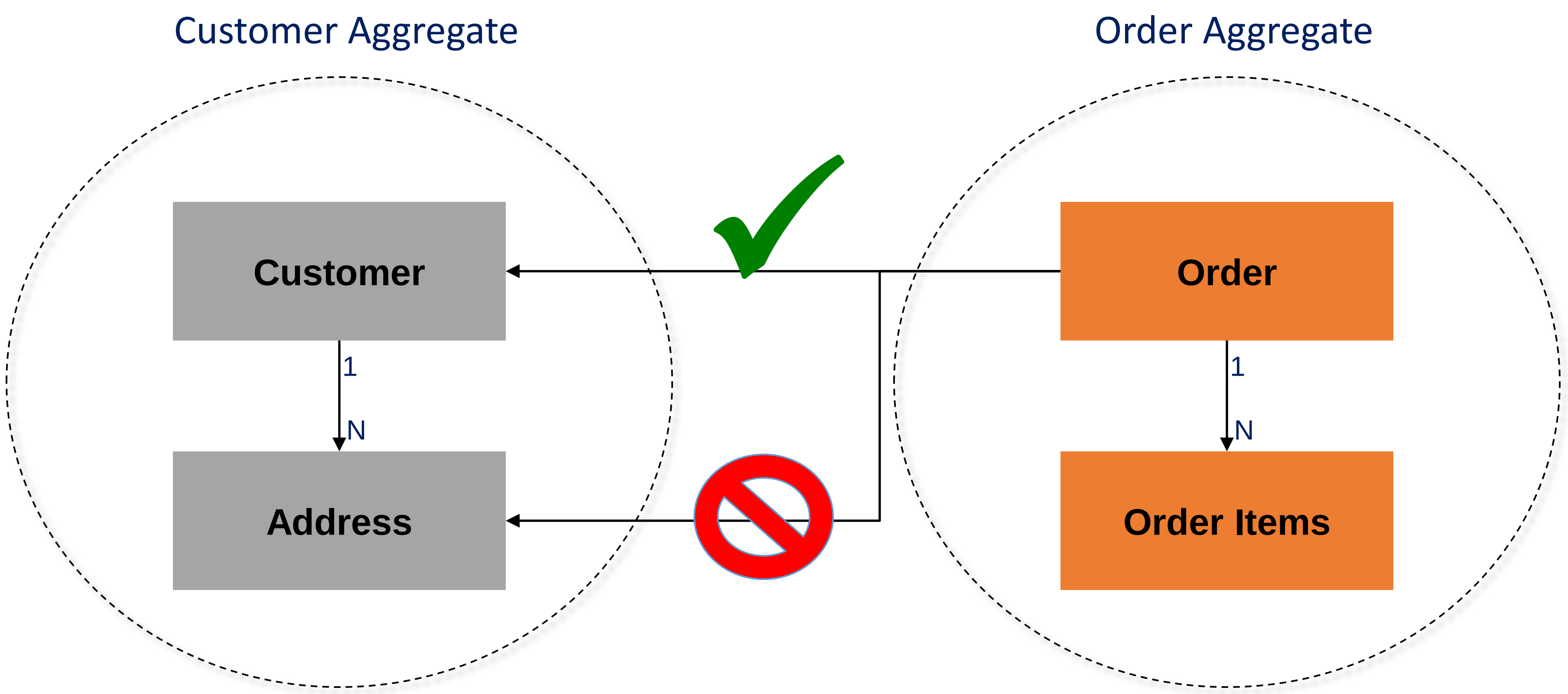




An aggregate is a cluster of associated objects that we treat as a unit for the purpose of data changes.

—Eric Evans
Domain-Driven Design

Relationships Between Aggregates



Examples of Aggregate Invariants

Total items on purchase order **do not exceed limit**	Two appointments **do not overlap** one another	End date **follows** Begin date

Aggregate Tips

Aggregates are **not always** the answer!

Aggregates can **connect only by the root**

Don’t overlook using **FKs for non-root entities**

- **Too many FKs** to non-root entities may suggest a problem

“**Aggregates of one**” are acceptable

“Rule of **Cascading Deletes**”

Glossary of Terms from this Module

Aggregate

A transactional graph of objects

Aggregate Root

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

Invariant

A condition that should always be true for the system to be in a consistent state

Persistence Ignorant Classes

Classes that have no knowledge about how they are persisted

Demonstration

Lab 1

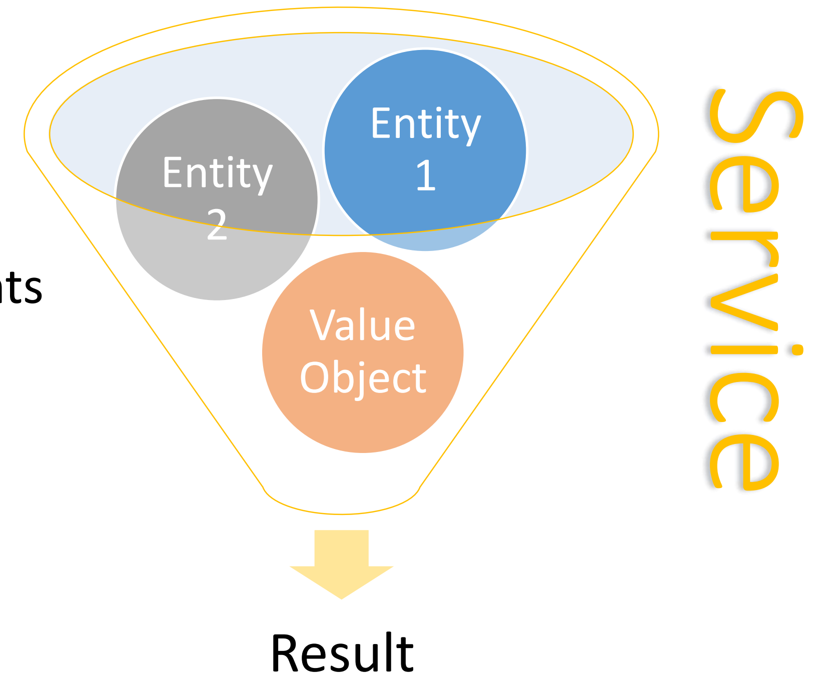
Domain Services

Important operations that don't belong to a particular Entity or Value Object

Good Domain Services:

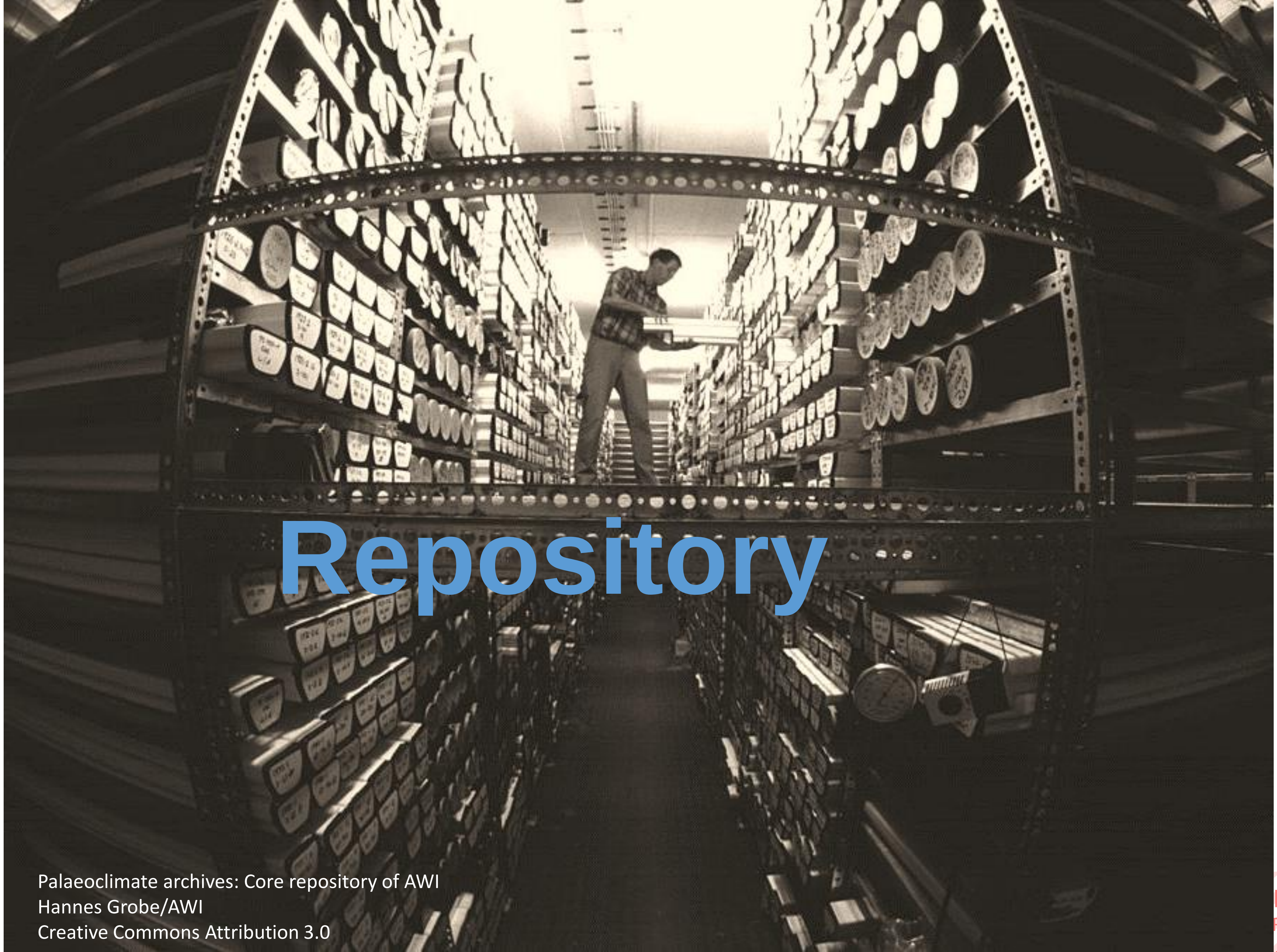
- Not a natural part of an Entity or Value Object
- Have an interface defined in terms of other domain model elements
- Are stateless (but may have side effects)

Live in the Core of the application



Examples of Services in Different Layers

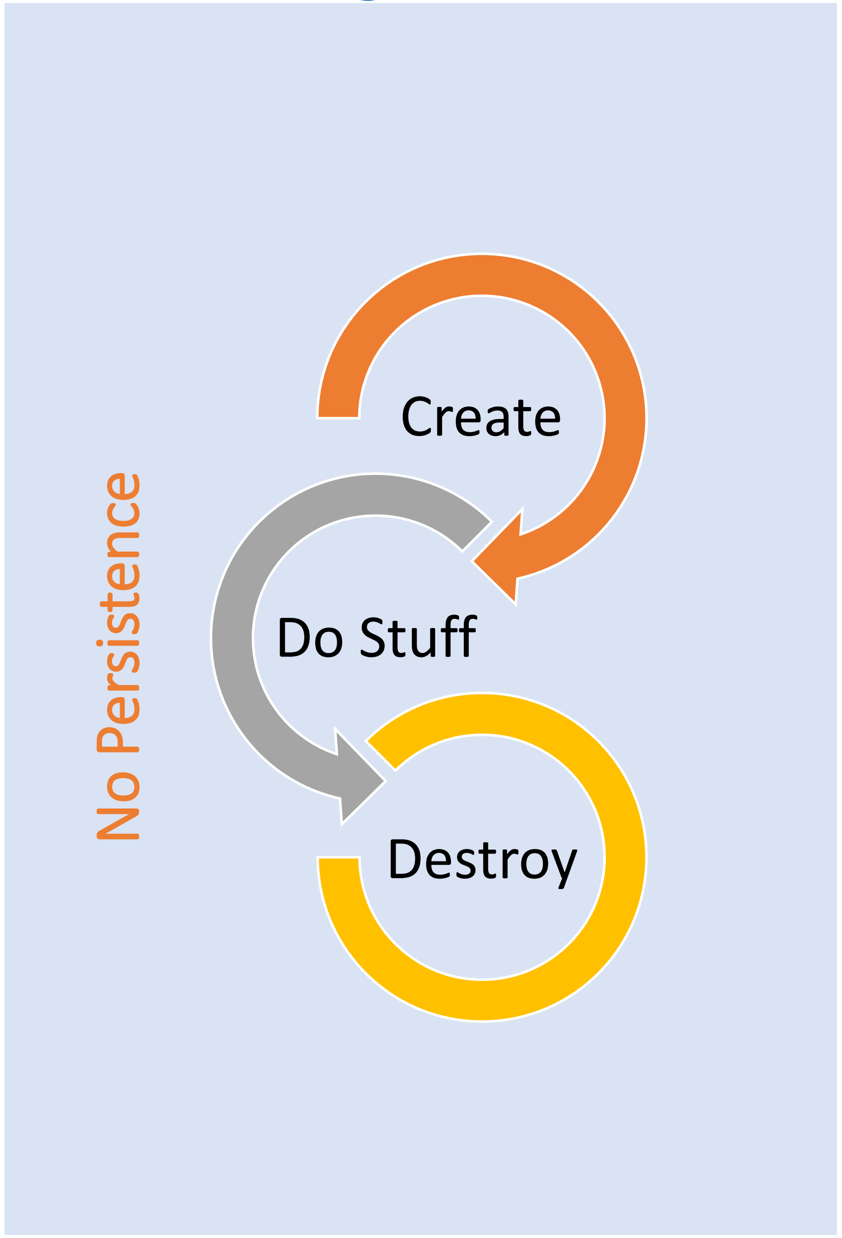
UI Layer & Application Layer	Domain (“Application Core”)	Infrastructure
Message Sending	Transfer Between Accounts	Send Email
Message Processing	Process Order	Log to a File
XML Parsing		
UI Services		

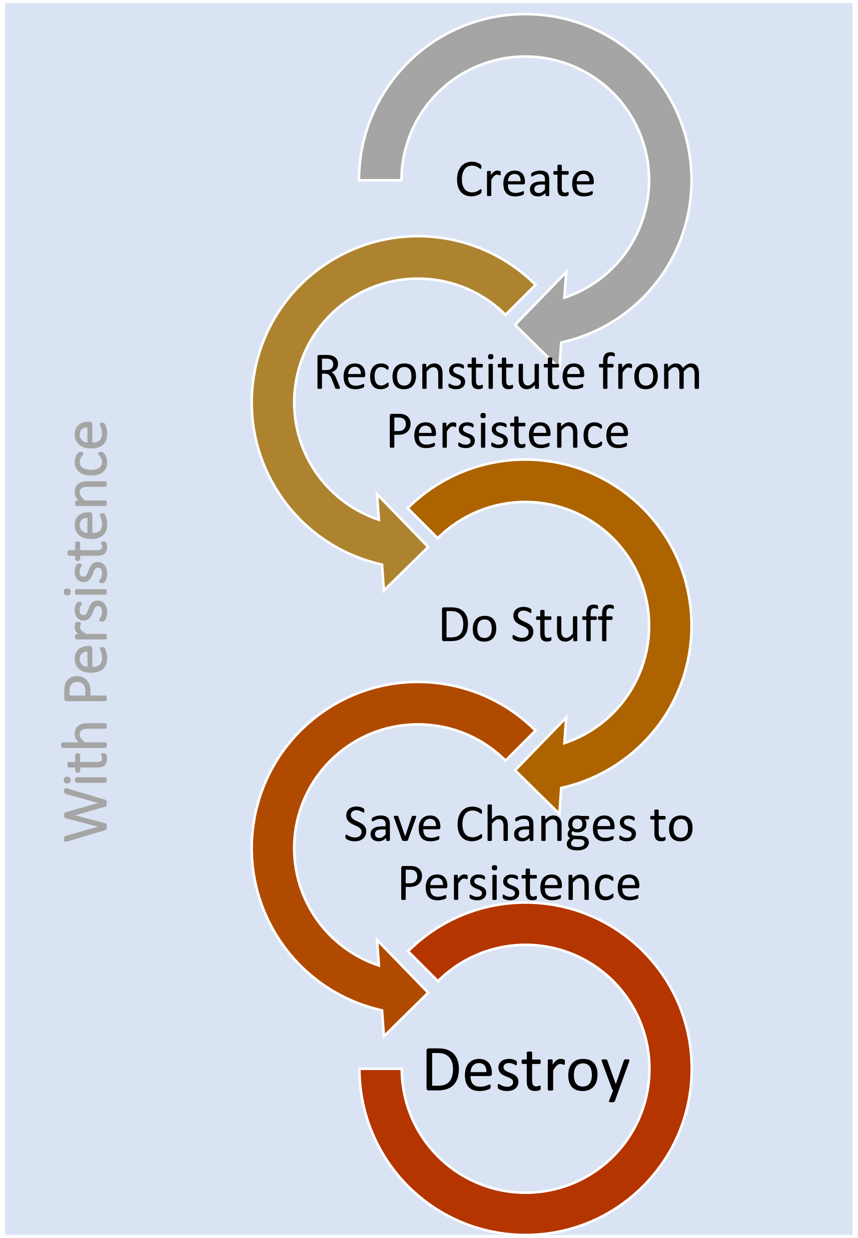


Repository

Palaeoclimate archives: Core repository of AWI
Hannes Grobe/AWI
Creative Commons Attribution 3.0

Object Life Cycles





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

Repository Tips

```
public Schedule GetScheduledAppointmentsForDate(int clinicId, DateTime date) {
  var scheduleGraph = QueryScheduleForThisOffice(clinicId)
                      .Select(s => new
                      {
                          Schedule = s,
                          Appointments = s.Appointments
                              .Where(a =>
                              DbFunctions.DiffDays(date, a.TimeRange.Start) == 0
                              )
                      })
                      .SingleOrDefault();
  var schedule = scheduleGraph.Schedule;
  schedule.DateRange = new DateTimeRange(date, date.AddDays(1));
  return schedule;
}
```

object selection

on persistence

Repository Benefits

Provides common abstraction for persistence

Promotes Separation of Concerns

Communicates Design Decisions

Enables Testability

Improved Maintainability

Client code can be ignorant of repository implementation

…but developers cannot

Common Repository Blunders		
N+1 Query Errors	Inappropriate use of eager or lazy loading	Fetching more data than required

Repositories Factories

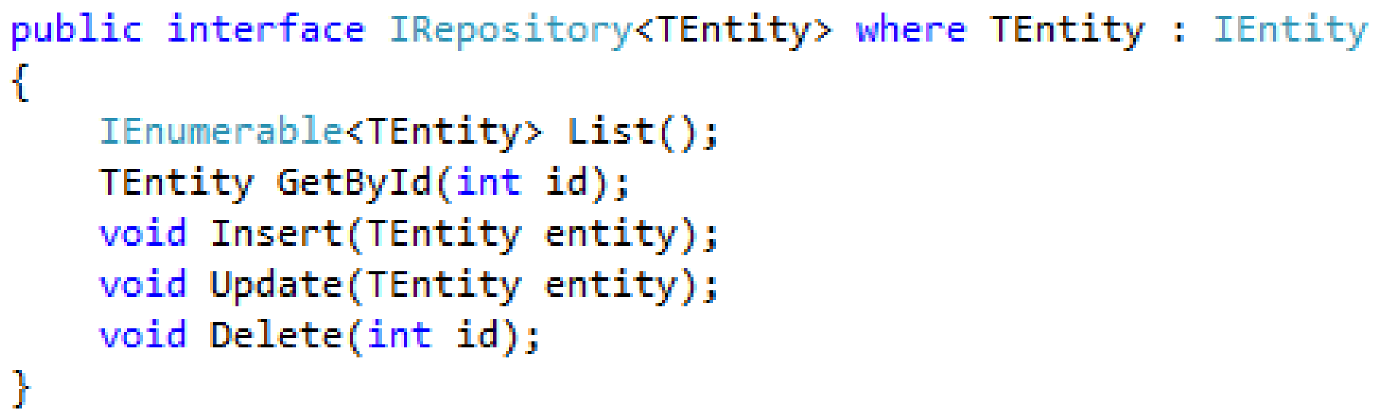
objects

- Factories create new objects
- Repositories find and update existing objects
- A Repository can use a Factory to create its objects

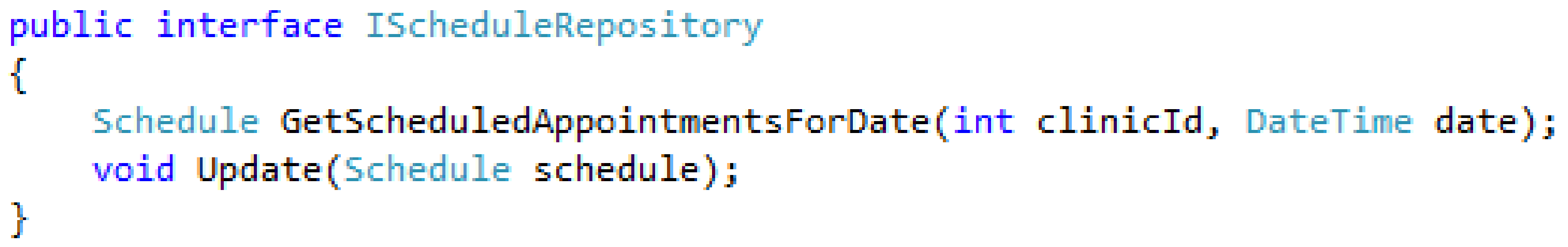
persistence

- Factories : no, no, no
- Repositories: yes, yes, yes

To IRepository<T> or Not To IRepository<T> ?

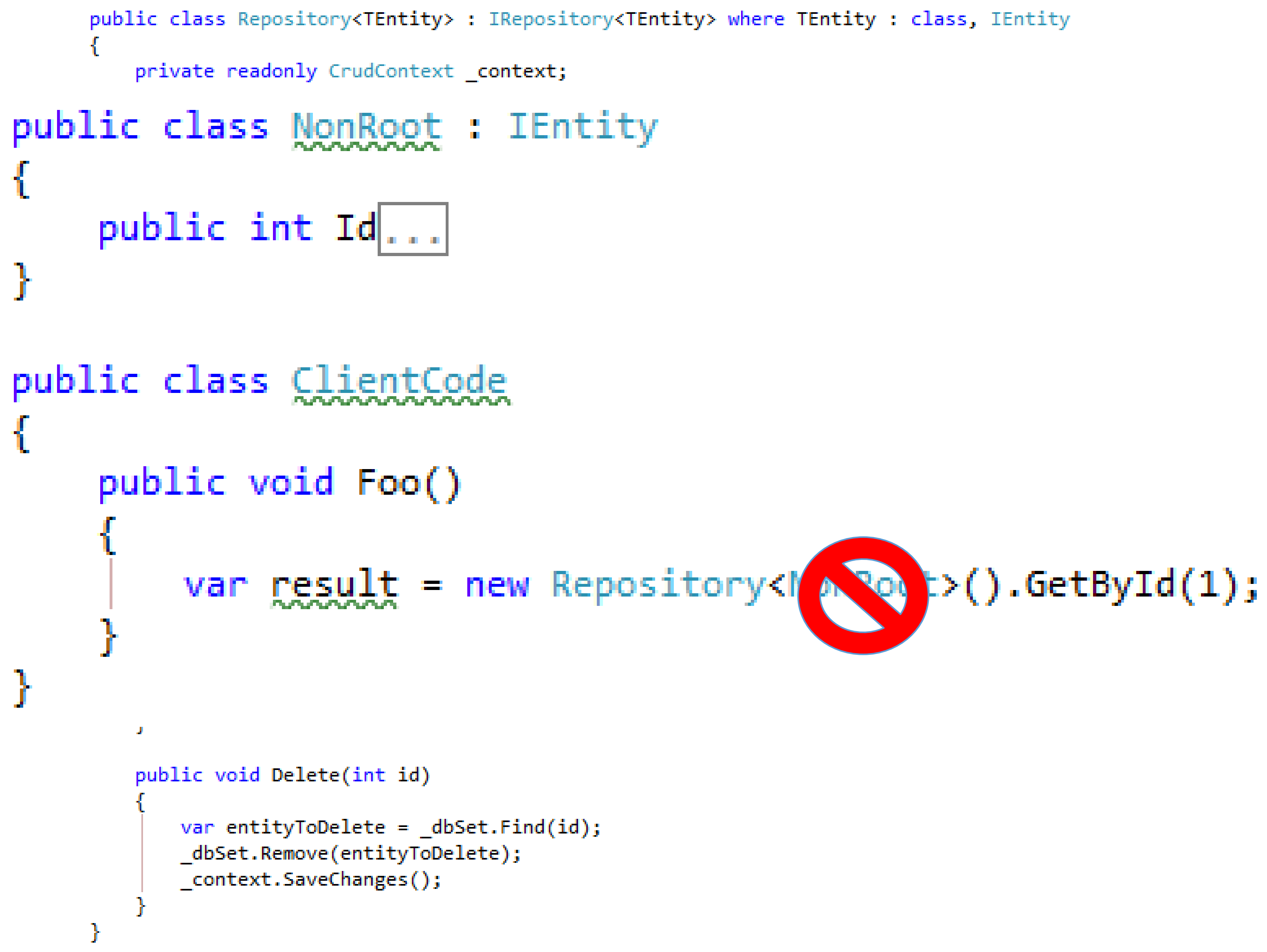


```
public interface IRepository<TEntity> where TEntity : IEntity
{
    IEnumerable<TEntity> List();
    TEntity GetById(int id);
    void Insert(TEntity entity);
    void Update(TEntity entity);
    void Delete(int id);
}
```



```
public interface IScheduleRepository
{
    Schedule GetScheduledAppointmentsForDate(int clinicId, DateTime date);
    void Update(Schedule schedule);
}
```

Generic Repositories in DDD



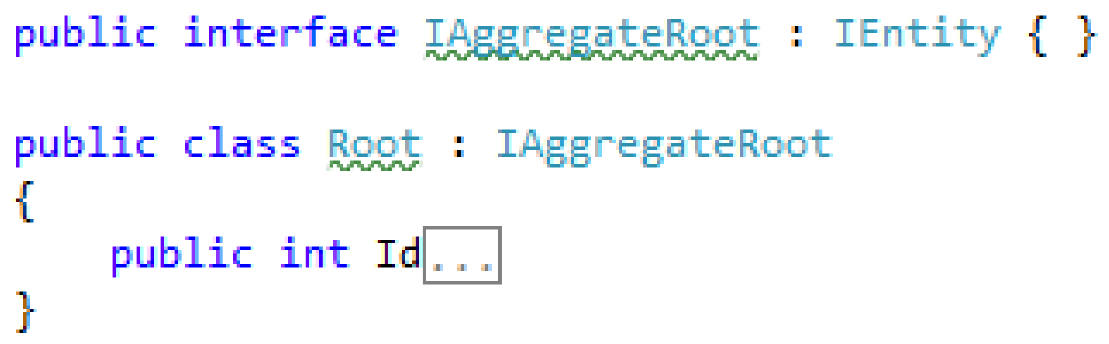
```
public class Repository<TEntity> : IRepository<TEntity> where TEntity : class, IEntity
{
    private readonly CrudContext _context;
```

```
public class NonRoot : IEntity
{
    public int Id...
}

public class ClientCode
{
    public void Foo()
    {
        var result = new Repository<NonRoot>().GetById(1);
    }
}
```

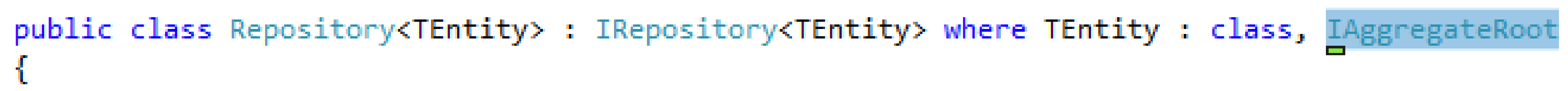
```
    public void Delete(int id)
    {
        var entityToDelete = _dbSet.Find(id);
        _dbSet.Remove(entityToDelete);
        _context.SaveChanges();
    }
}
```

Generic Repositories in DDD

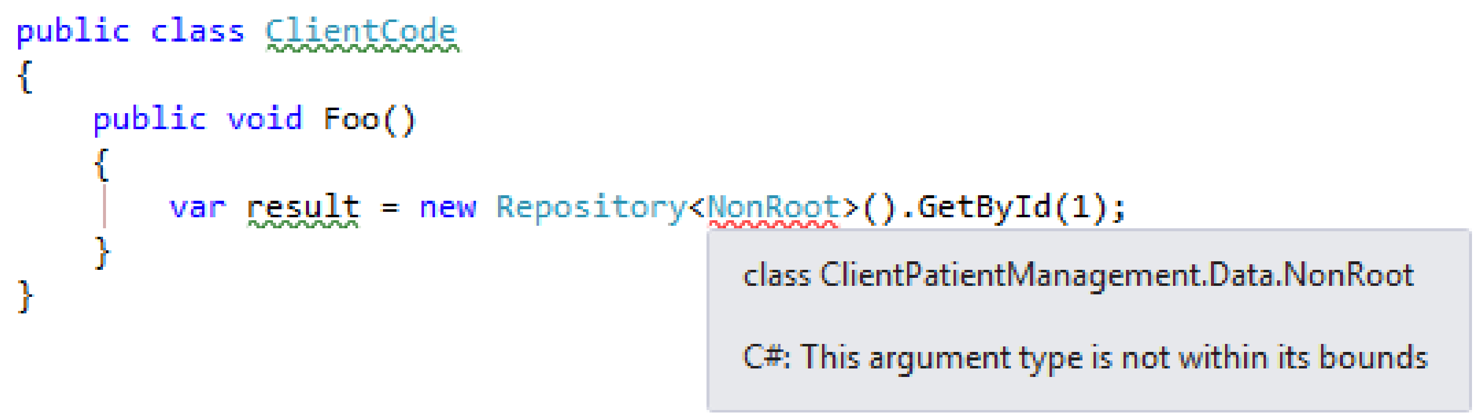


```
public interface IAggregateRoot : IEntity { }

public class Root : IAggregateRoot
{
    public int Id...
}
```



```
public class Repository<TEntity> : IRepository<TEntity> where TEntity : class, IAggregateRoot
{
```



```
public class ClientCode
{
    public void Foo()
    {
        var result = new Repository<NonRoot>().GetById(1);
    }
}
```

class ClientPatientManagement.Data.NonRoot

C#: This argument type is not within its bounds

Glossary of Terms from this Module

Repository

A class that encapsulates the data persistence for an aggregate root

ACID

Atomic, Consistent, Isolated, and Durable

Glossary of 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 it’s property values) used for tracking and persistence.

Immutable

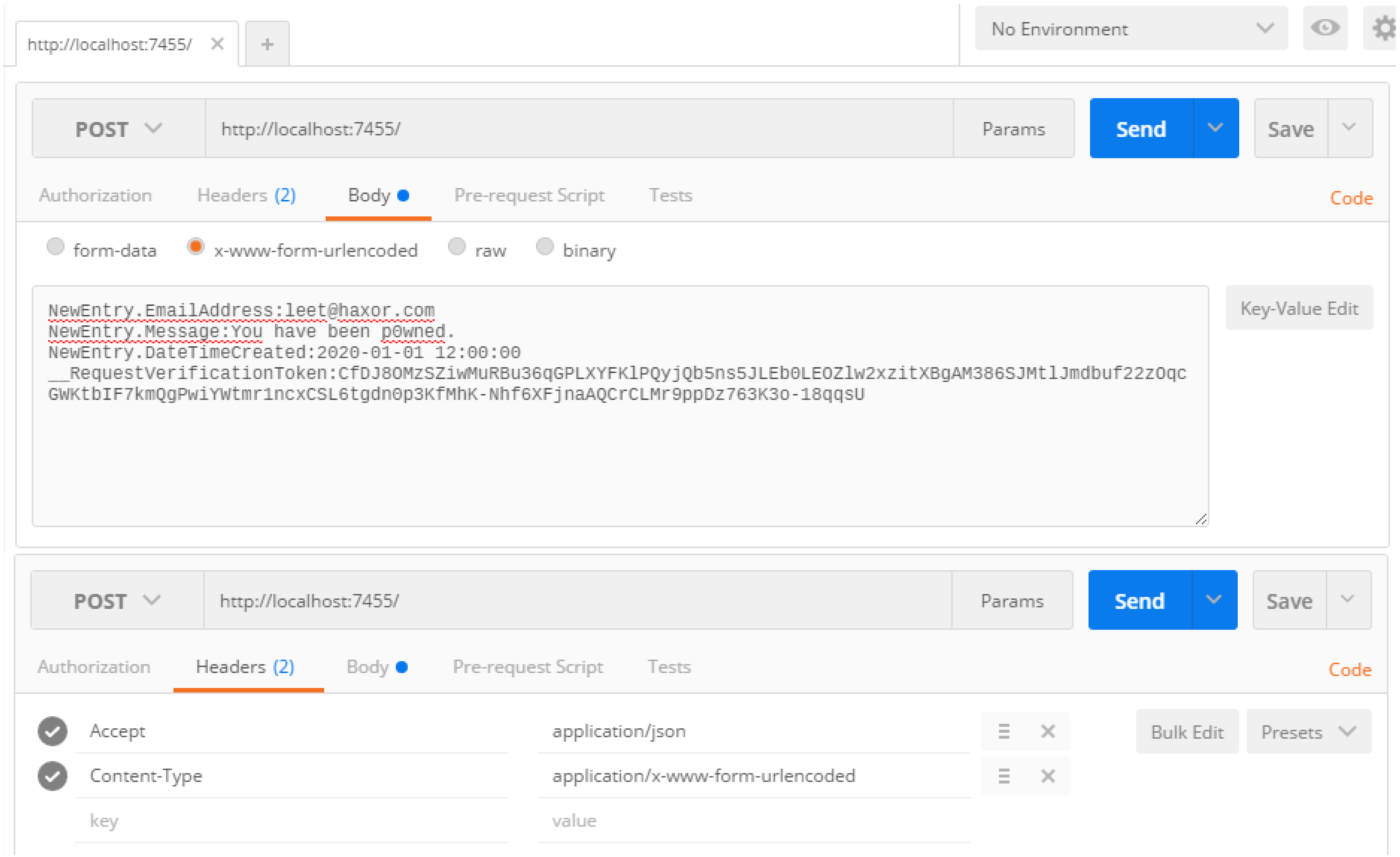
Refers to a type whose state cannot be changed once the object has been instantiated.

Glossary of Terms from this Module

Value Object	An immutable class whose identity is dependent on the combination of its values
Services	Provide a place in the model to hold behavior that doesn't belong elsewhere in the domain
Side Effects	Changes in the state of the application or interaction with the outside world (e.g. infrastructure)

Demonstration

Lab 2



POST http://localhost:7455/

Body: ( ) form-data [✓ x-www-form-urlencoded] ( ) raw ( ) binary

```
NewEntry.EmailAddress:leet@haxor.com
NewEntry.Message:You have been p0wned.
NewEntry.DateTimeCreated:2020-01-01 12:00:00
__RequestVerificationToken:CfDJ8OMzSZiwMuRBu36qGPLXYFKlPQyjQb5ns5JLEb0LEOZlw2xzitXBgAM386SJMtlJmdbuf22zOqc
GWKtbIF7kmQgPwiYWtmr1ncxCSL6tgdn0p3KfMhK-Nhf6XFjnaAQCrCLMr9ppDz763K3o-18qqsU
```

POST http://localhost:7455/

Headers (2)

✓	Accept	application/json
✓	Content-Type	application/x-www-form-urlencoded
	key	value

Lunch!

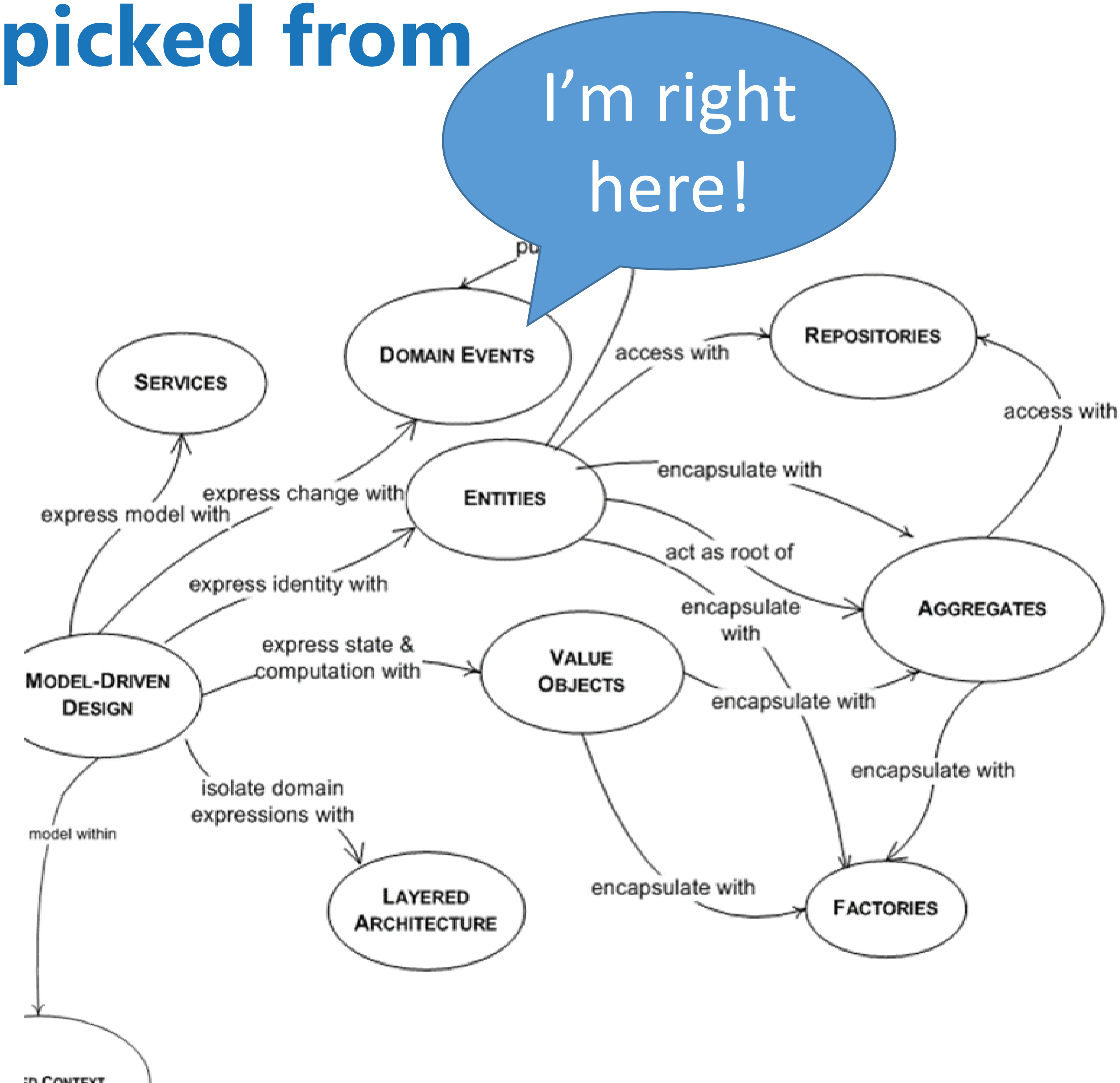
Domain-Driven Design

Decoupling with Domain Events

Usually the last kid picked from the DDD patterns

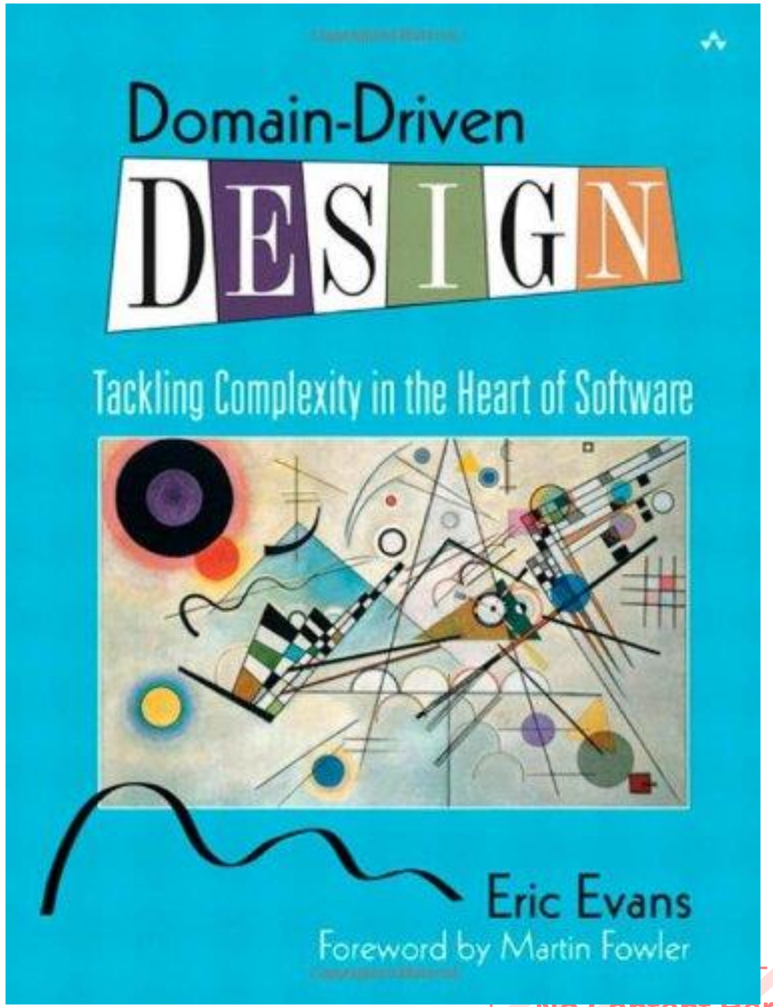
- Repositories ✓
- Factories ✓
- Services ✓
- Entities ✓
- Value Objects ✓
- Aggregates ✓

- Oh yeah, *Domain Events*



Domain-Driven Design

- Domain Events not covered in original DDD book (2004)
- Covered by Martin Fowler in 2005
- Evans published article on them in 2010


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

Kinds of Events

- Application Events
  - Page Load, Button Click, Window Scroll
- System Events
  - Restart, Backup Completed
- Domain Events
  - Appointment Confirmed, Checkout Completed, Analysis Finished

An example scenario

Given a customer has created an order

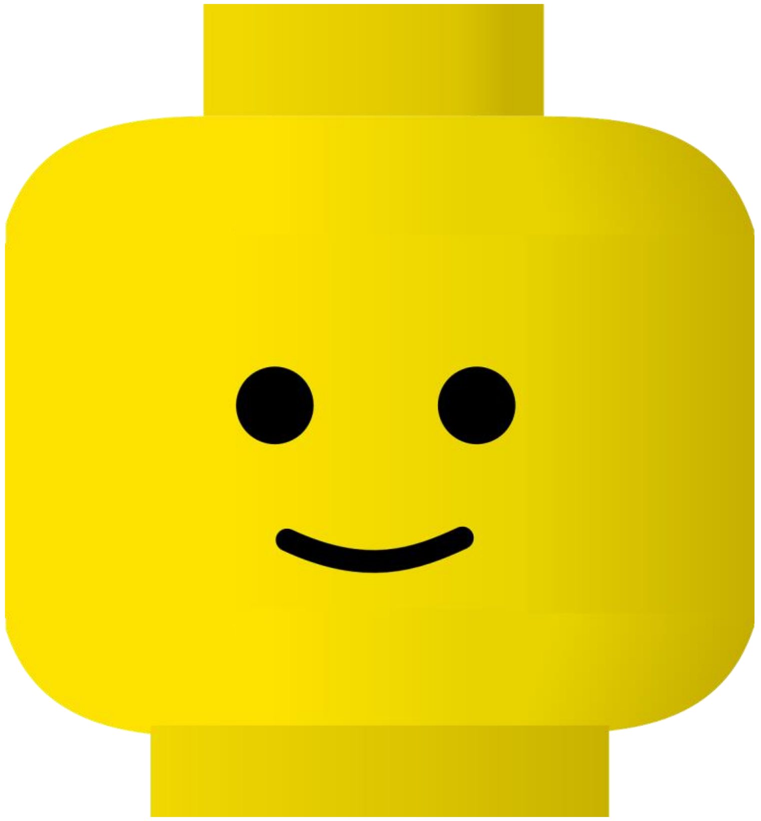
When the customer completes their purchase

Then the customer receives an email confirmation

An example solution

```
OrderService
  Checkout(Order order)
    - Save pending order in database
    - Send confirmation email
```

So far, so good



More requirements

Given a customer has created an order

When the customer completes their purchase

Then the customer’s card is charged

And if it fails, send a different message to the customer

Then inventory is checked to ensure the order can be fulfilled

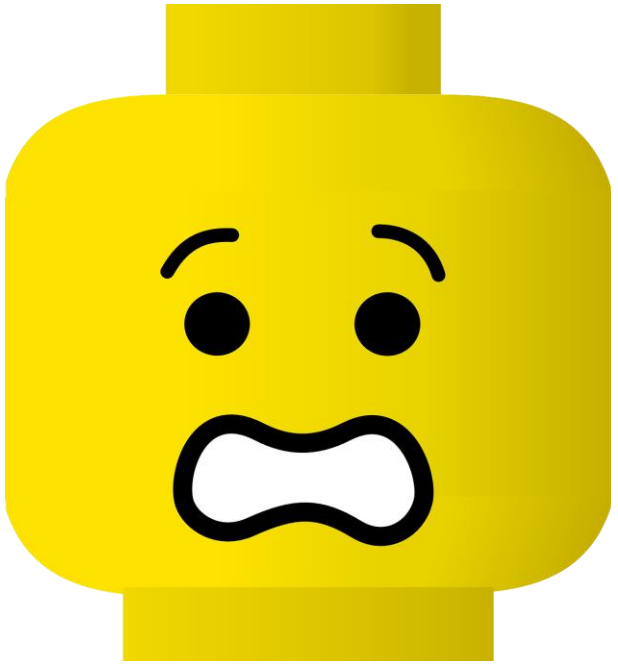
And if not a different message is sent to the customer

An example solution

OrderService

Checkout(Order order)

- Attempt to process payment
- If payment fails, send notification email; **exit**
- Save pending order in database
- Confirm inventory is available
- If insufficient email, send notification email; **exit**
- Send successful order confirmation email



Analysis

- Complexity is growing
- Must change with each new requirement
  - Open/Closed Principle
- Checkout’s responsibilities are growing
  - Single Responsibility Principle
- Potentially different abstraction levels (emails, order processing rules)

An event-driven solution

```
OrderService
  Checkout(Order order)
    - Dispatch new OrderCompletedEvent(order)
```

Event Handlers:

- OrderPaymentHandler
- OrderInventoryHandler
- OrderNotificationHandler

Event-Driven Programming

An Event

- Something that happened
  - …that other parts of the application may need to know about
- It’s a message
  - …which should be immutable, since it represents the past
- Usually asynchronous
  - …especially across process boundaries

Domain Events

Model something that happens which is of interest to a domain expert

May lead to (or result from) a state change in a domain object

Are part of the **domain model**

Are usually handled **synchronously** within the application (but may themselves raise events outside of the application)

Event Processing

- An event is raised
- Handlers **within the current process** handle the event
- If external applications need to be notified, specific handlers can adapt and send events to these applications as well
- If unknown or future external applications will need to respond to events, a *service bus* can be implemented

Common Scenario

- How do you add logic to entities that affects multiple entities?

Example:

When a customer’s total amount purchased exceeds $1000, notify a salesperson to contact them.

Avoid: Injecting Dependencies into Entities

- How do you add logic to entities that affects multiple entities?

Don’t solve this by using Dependency Injection on your entities. You should be able to easily instantiate entities without dependencies.

Avoid: Shifting Entity logic to Services

- How do you add logic to entities that affects multiple entities?

Don’t move logic that belongs within an entity into a service just because other entities are interested in what’s happening. This leads to the **anemic domain model** antipattern.

Only do this if the behavior spans multiple aggregates, in which case, the logic doesn’t belong to a particular entity or aggregate.

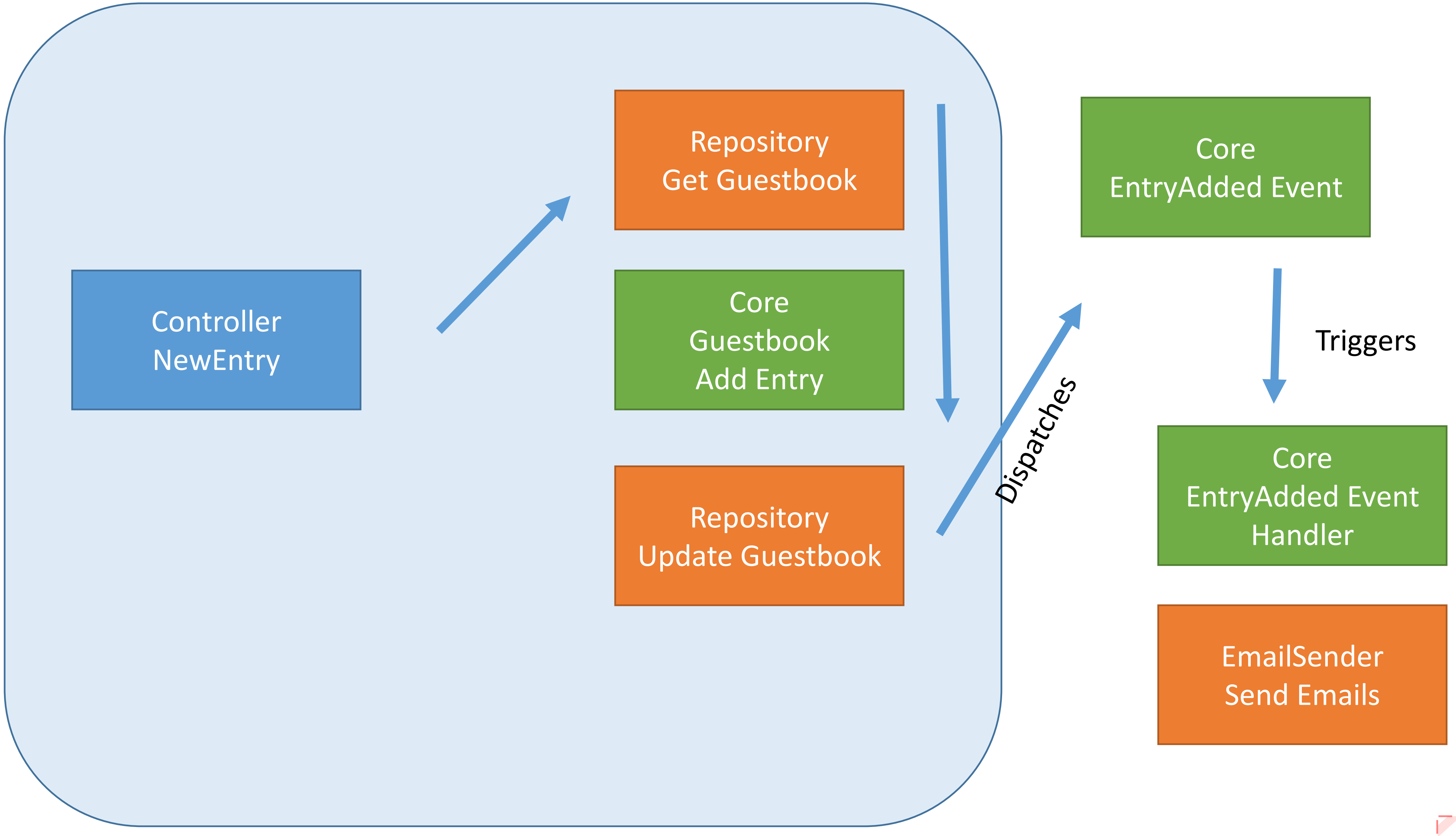
Use Domain Events (and perhaps Aggregates)

- How do you add logic to entities that affects multiple entities?

Raise a **domain event** to represent the action that took place.

Handle the domain event in a handler that collaborates with other entities.

Alternately, have the **Aggregate Root** register for events raised by members of its **Aggregate**.


Controller
NewEntry
Repository
Get Guestbook
Core
Guestbook
Add Entry
Repository
Update Guestbook
Dispatches
Core
EntryAdded Event
Triggers
Core
EntryAdded Event
Handler
EmailSender
Send Emails

Glossary of Terms from this Module

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”

Inversion of Control (IOC)

A pattern for loosely coupling a dependent object with an object it will need at runtime

Demonstration

Lab 3

Domain-Driven Design

Integration and Unit Testing

Unit Tests Prevent Small Thermal Exhaust Ports in Your Code

Unit Test Characteristics

- Test a single unit of code
  - A method, or at most, a class
- Do not test Infrastructure concerns
  - Database, filesystem, etc.
- Do not test “through” the UI
  - Just code testing code; no screen readers, etc.

Unit Tests are (should be) FAST

- No dependencies means 1000s of tests per second
- Run them *constantly*

Unit Tests are SMALL

- Testing one thing should be simple
  - If not, can it be made simpler?
- Should be quick to write

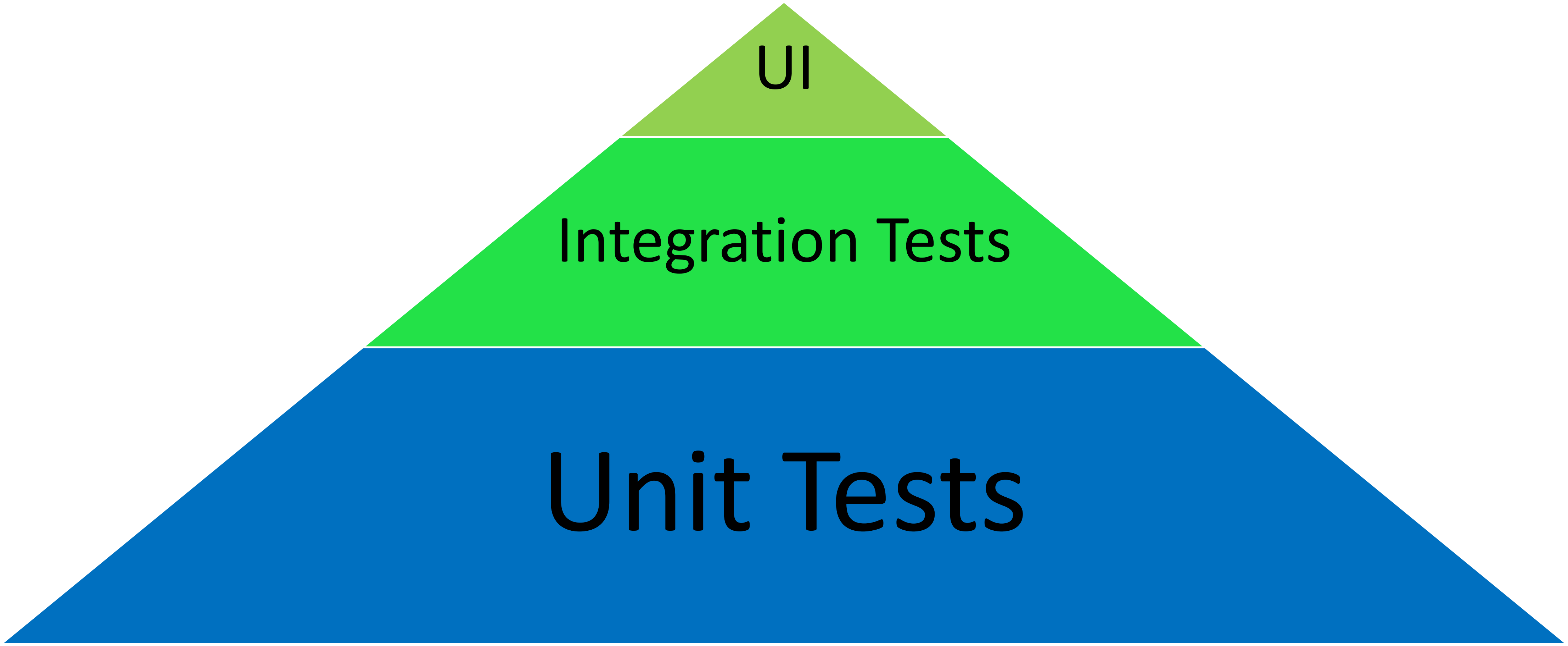
Unit Test Naming

- Descriptive And Meaningful Phrases (DAMP)
- Name Test Class: ClassNameMethodName
- Name Test Method: DoesSomethingGivenSomething
- http://ardalis.com/unit-test-naming-convention

Seams

- Represent areas of code where pieces can be decoupled
- Testable code has many seams; legacy code has few, if any

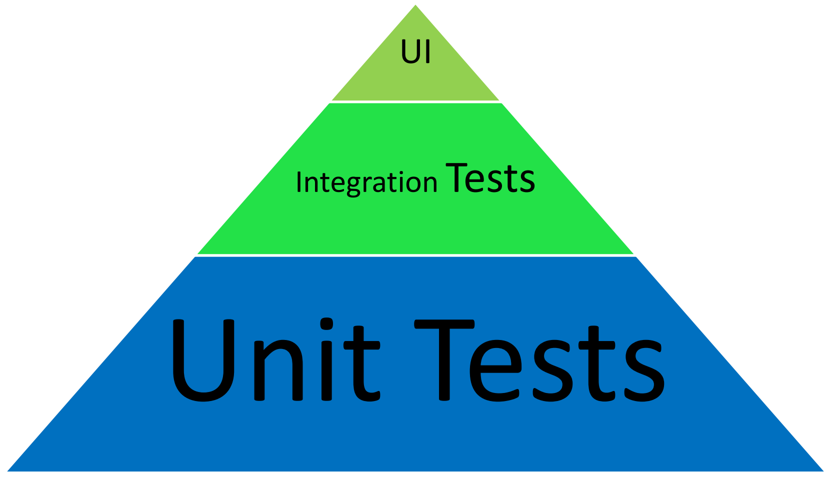
Kinds of Tests



http://martinfowler.com/bliki/TestPyramid.html

Ask yourself:

- Can I test this scenario with a Unit Test?
  - Can I test it with an Integration Test?
    - Can I test it with an automated UI Test?

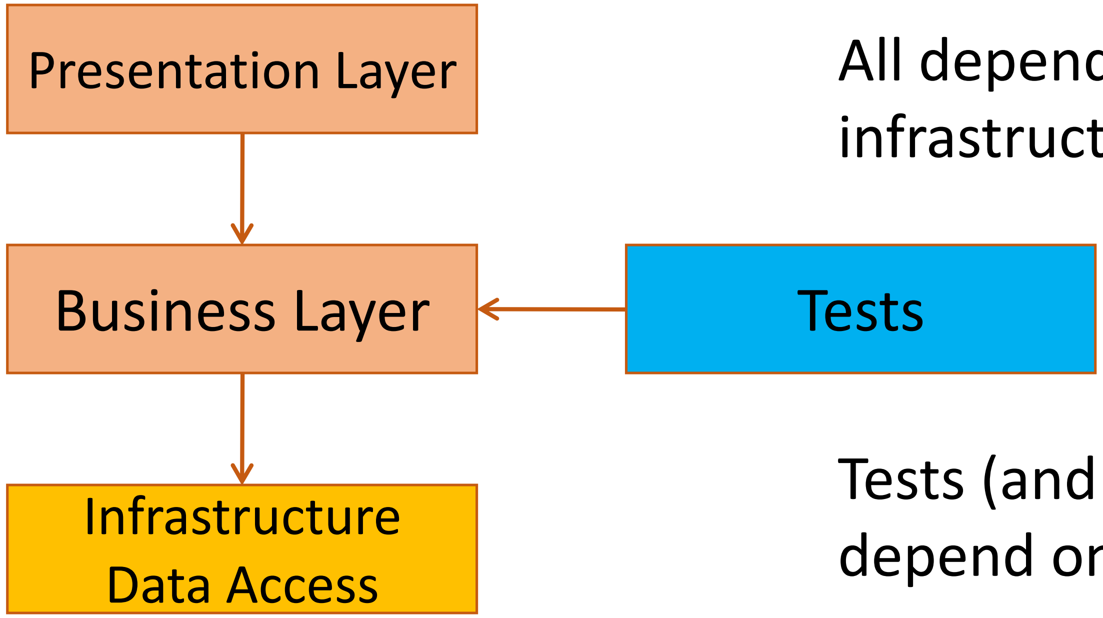


Unit Test?

- Requires a database or file?
- Sends emails?
- Must be executed through the UI?

Not a unit test

Dependencies and Coupling



All dependencies point toward infrastructure

Tests (and everything else) now depend on Infrastructure

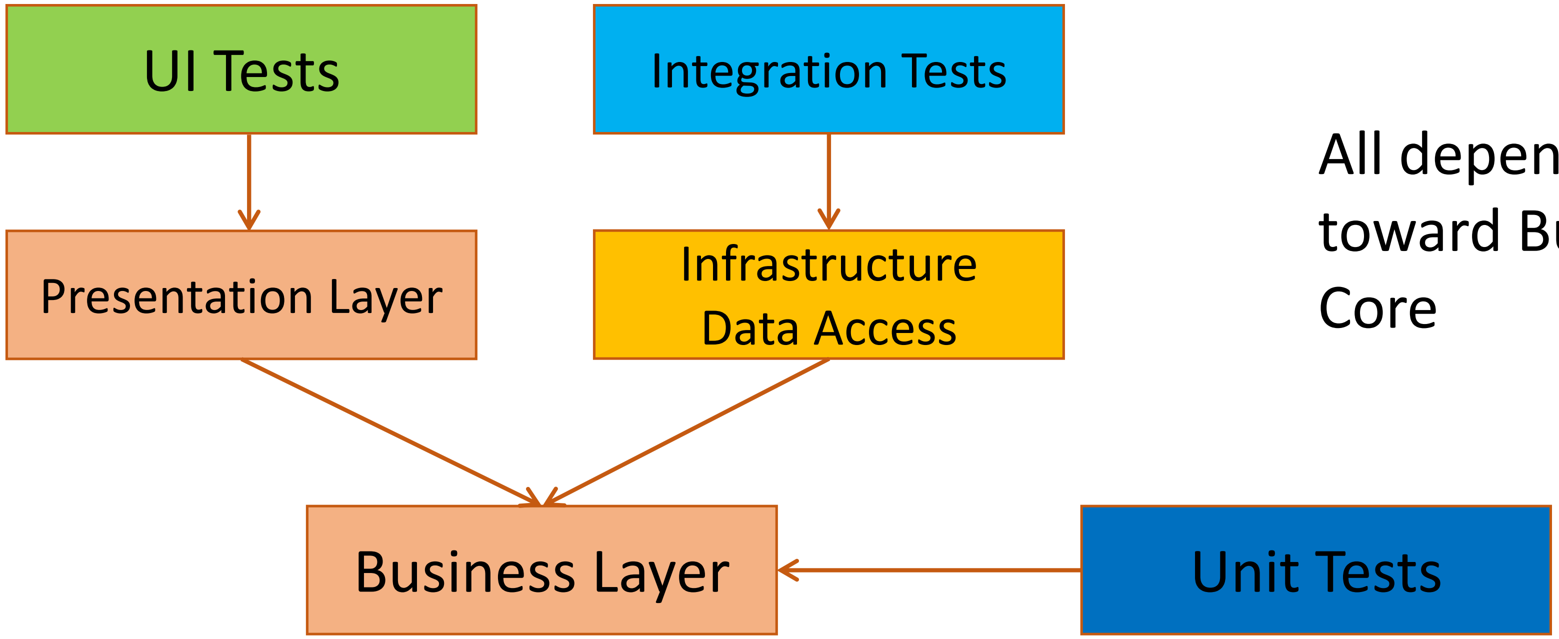
Dependency Inversion Principle

High-level modules should not depend on low-level modules. Both should depend on abstractions.

Abstractions should not depend on details. Details should depend on abstractions.

Agile Principles, Patterns, and Practices in C#

Depend on *Abstractions*



All dependencies point toward Business Logic / Core

Inject Dependencies

- Classes should follow Explicit Dependencies Principle
  - http://deviq.com/explicit-dependencies-principle

- Prefer Constructor Injection
  - Classes cannot be created in an invalid state

https://flic.kr/p/5QsGnB

Common Dependencies to Decouple

- Database
- File System
- Email
- Web APIs

- System Clock
- Configuration
- Thread.Sleep
- Random

Tight Couplers: Statics and new

- Avoid static cling
  - Calling static methods with side effects
- Remember: new is glue
  - Avoid gluing your code to a specific implementation
  - Simple types and value objects usually OK

http://ardalis.com/new-is-glue

Demonstration

Lab 4

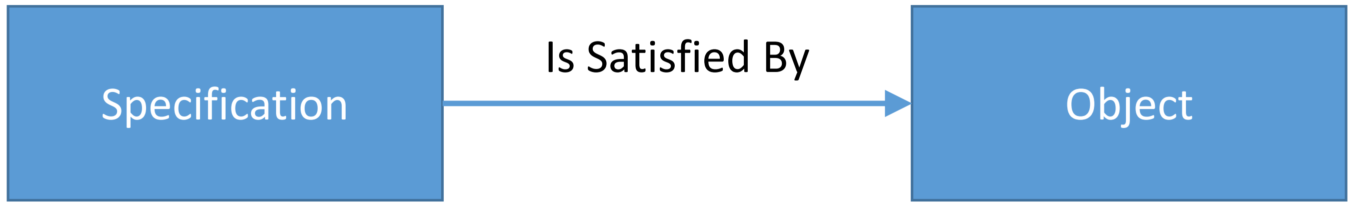
Domain-Driven Design

Specification

Specification

- Define a query as an object
- Can be tested independent of the object itself or the data store

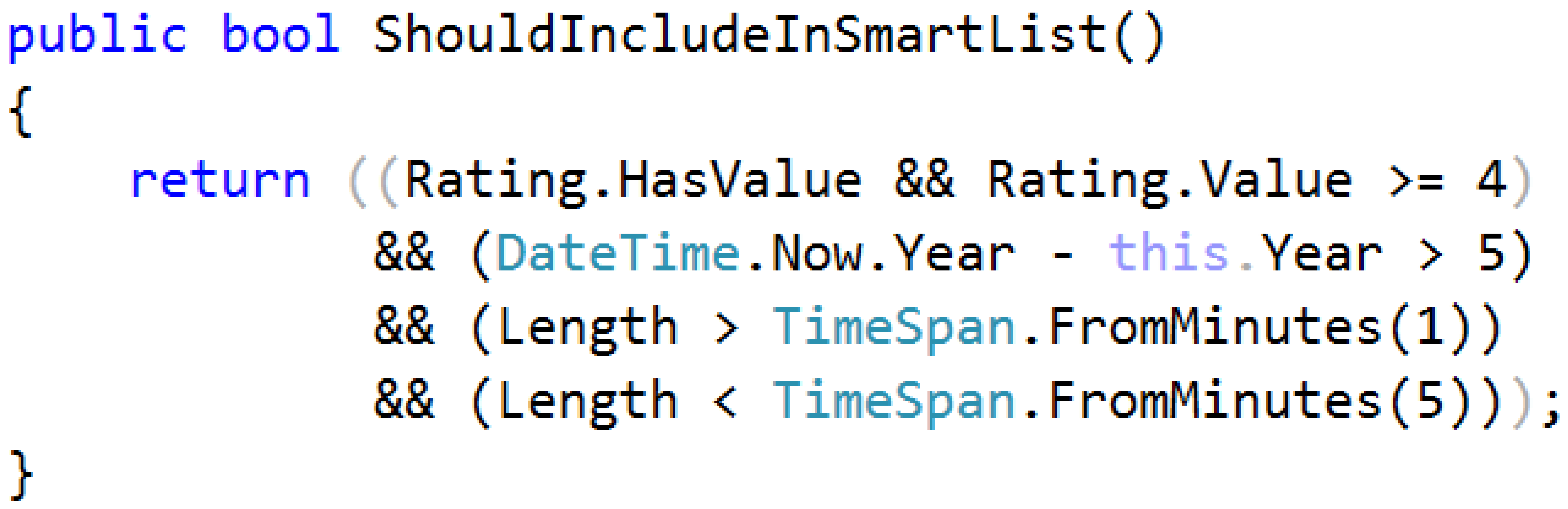
Implementation (basic)



The Specification object should be implemented as a Value Object

(Value Objects are discussed in *Domain-Driven Design Fundamentals)*

Code Smell: Complex Flags on Entities



```
public bool ShouldIncludeInSmartList()
{
    return ((Rating.HasValue && Rating.Value >= 4)
           && (DateTime.Now.Year - this.Year > 5)
           && (Length > TimeSpan.FromMinutes(1))
           && (Length < TimeSpan.FromMinutes(5)));
}
```

Sample Implementation

```
using System;
using System.Linq.Expressions;

namespace Core.Interfaces
{
    public interface ISpecification<T>
    {
        Expression<Func<T, bool>> Criteria { get; }
    }
}
```

Specification Benefits

- Gives a name to each query
- Separation of Concerns
- Produces a “library” of specification-based queries
  - Promotes reuse; reduces reinvention of wheel
- Can limit exposed surface area of data layer
  - Restrict filtering so only specifications may be used for queries

Sample Repository Update

```
// replace
public interface IFooRepository
{
    IQueryable<Foo> List();
}

// with
public interface IFooRepository
{
    IEnumerable<Foo> List(ISpec<Foo> spec);
}
```

Include

- By default, EF does not hydrate related objects or collections
  - Use code like this to do so:

```
.Include(“Orders”)
```

- Specification can include an expression for relationships to include:

```
Expression<Func<T,object>> Include { get; }
```

Usage:

```
return _dbContext.Songs
        .Include(x => specification.Include)
        .Where(specification.Criteria)
        .AsEnumerable();
```

Demonstration

Lab 5

EF Core

Collection Encapsulation

Questions?

Or tweet me @ardalis and I'll answer later.

DevIQ
DEVELOP INTELLIGENCE

Thanks!

No Content Here
(Reserved for Course Previews)

Steve Smith
Ardalis.com
@ardalis

Follow us:

@deviq
/DevIQPage
DevIQ.com/updates

Enroll in DevIQ for access to:

- full-length courses
- samples
- member-only events

DevIQ.com/enroll