Working with Repositories

Steve Smith

Force Multiplier for
Dev Teams

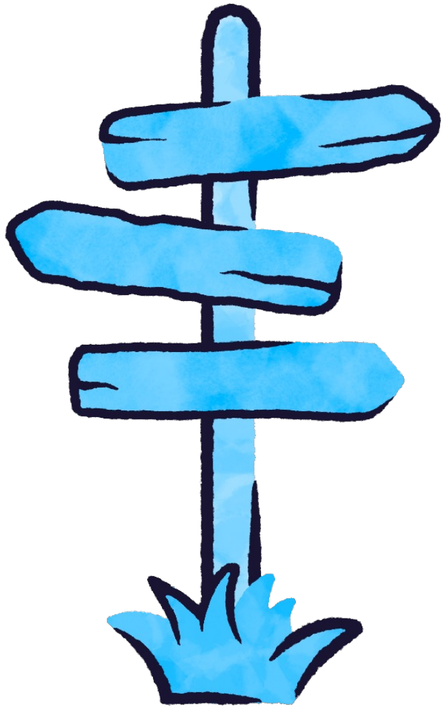
@ardalis | ardalis.com

Julie Lerman

Software Coach,
DDD Champion

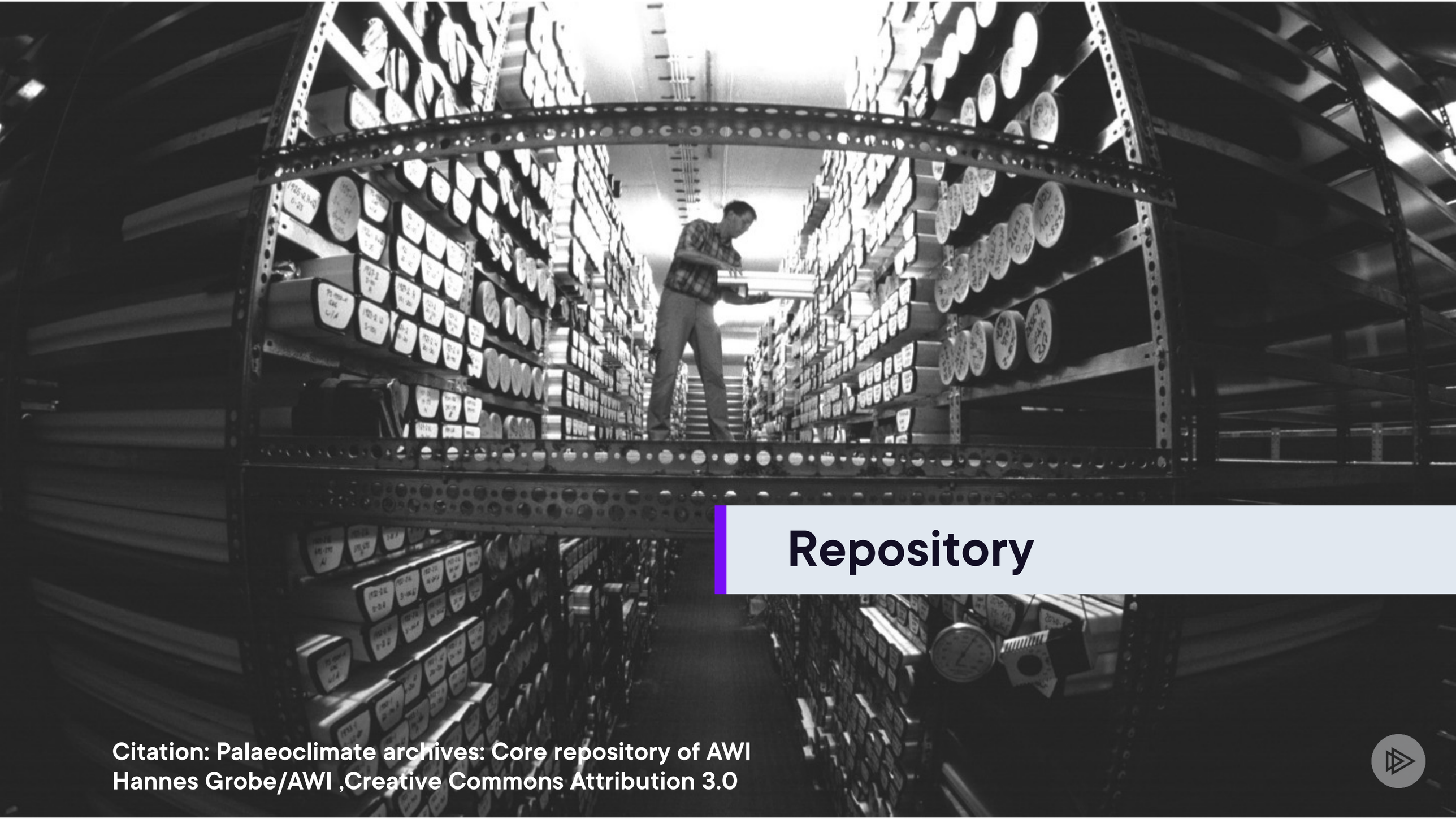
@julielerman | thedatafarm.com

Module Overview



- Define repositories
- Tips for designing repositories
- Benefits of repositories
- Pros and cons of interfaces and generic repos
- Specification pattern to aid repositories
- Repository implementations in our app

Introducing Repositories



Repository

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

Julie Lerman

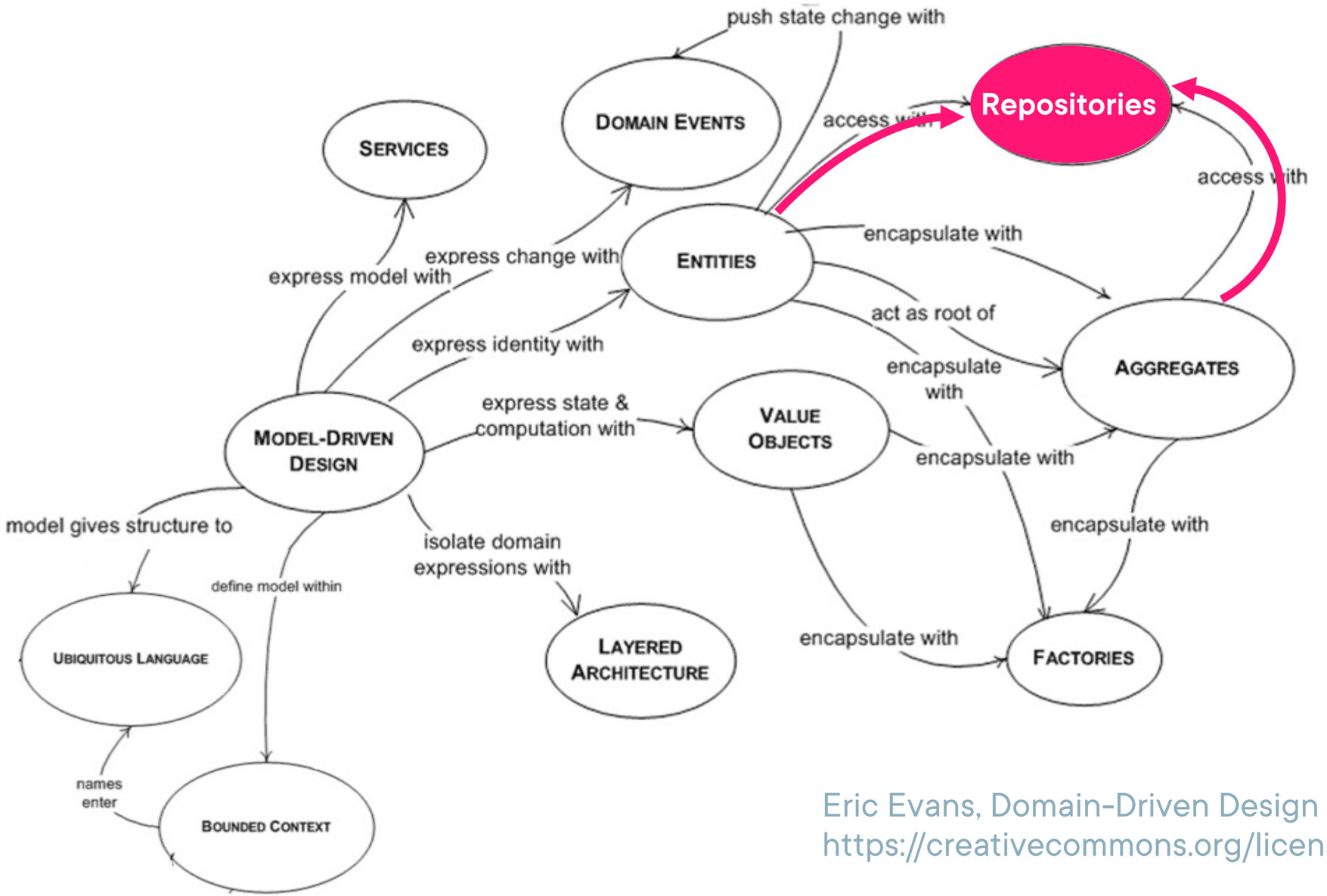
“Considering repositories had a huge impact on how I thought about software design.”

Repository Pattern

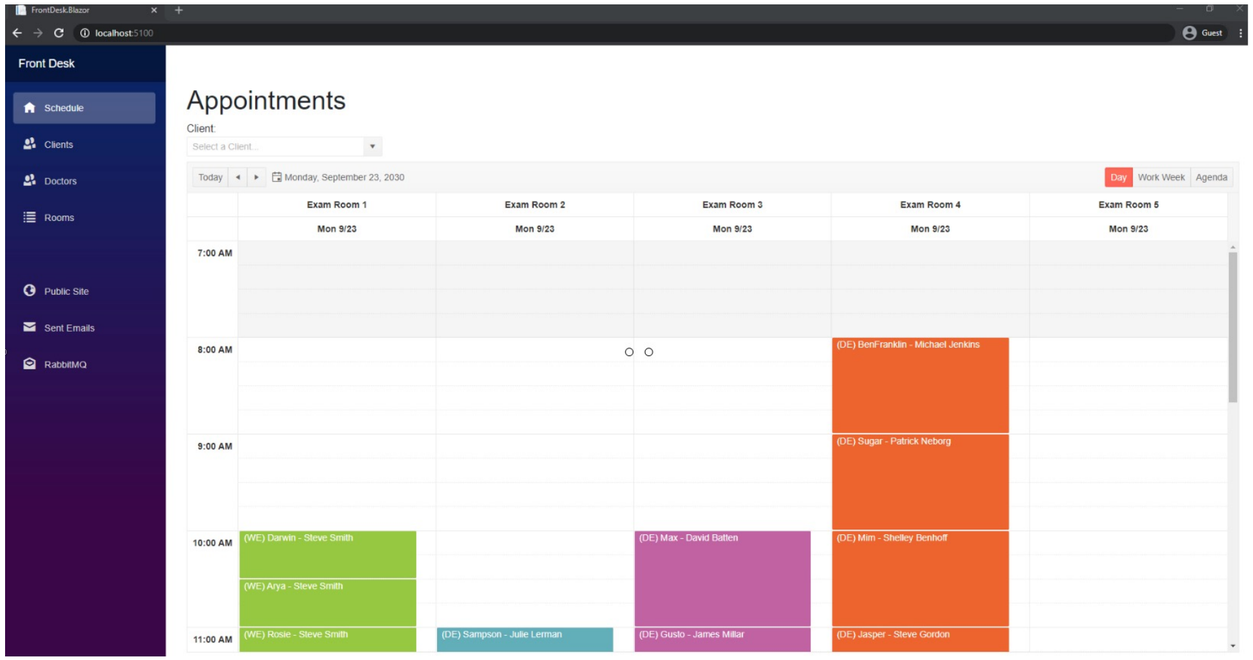
C# 8 Design Patterns: Data Access Patterns

Filip Ekberg

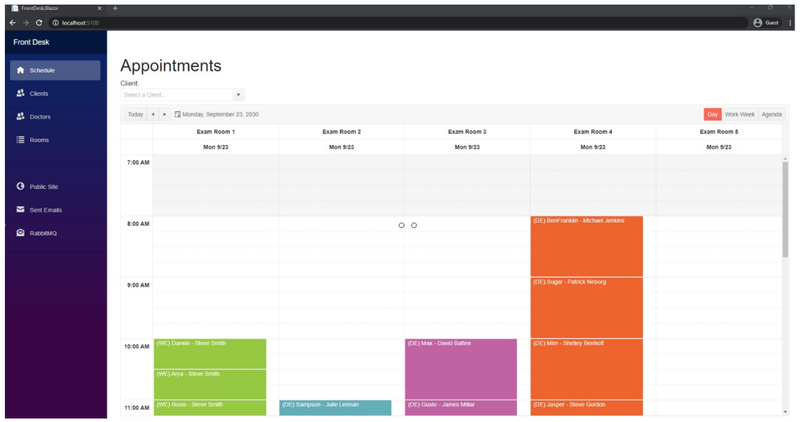
Repositories in the DDD Mind Map



Persisting Objects

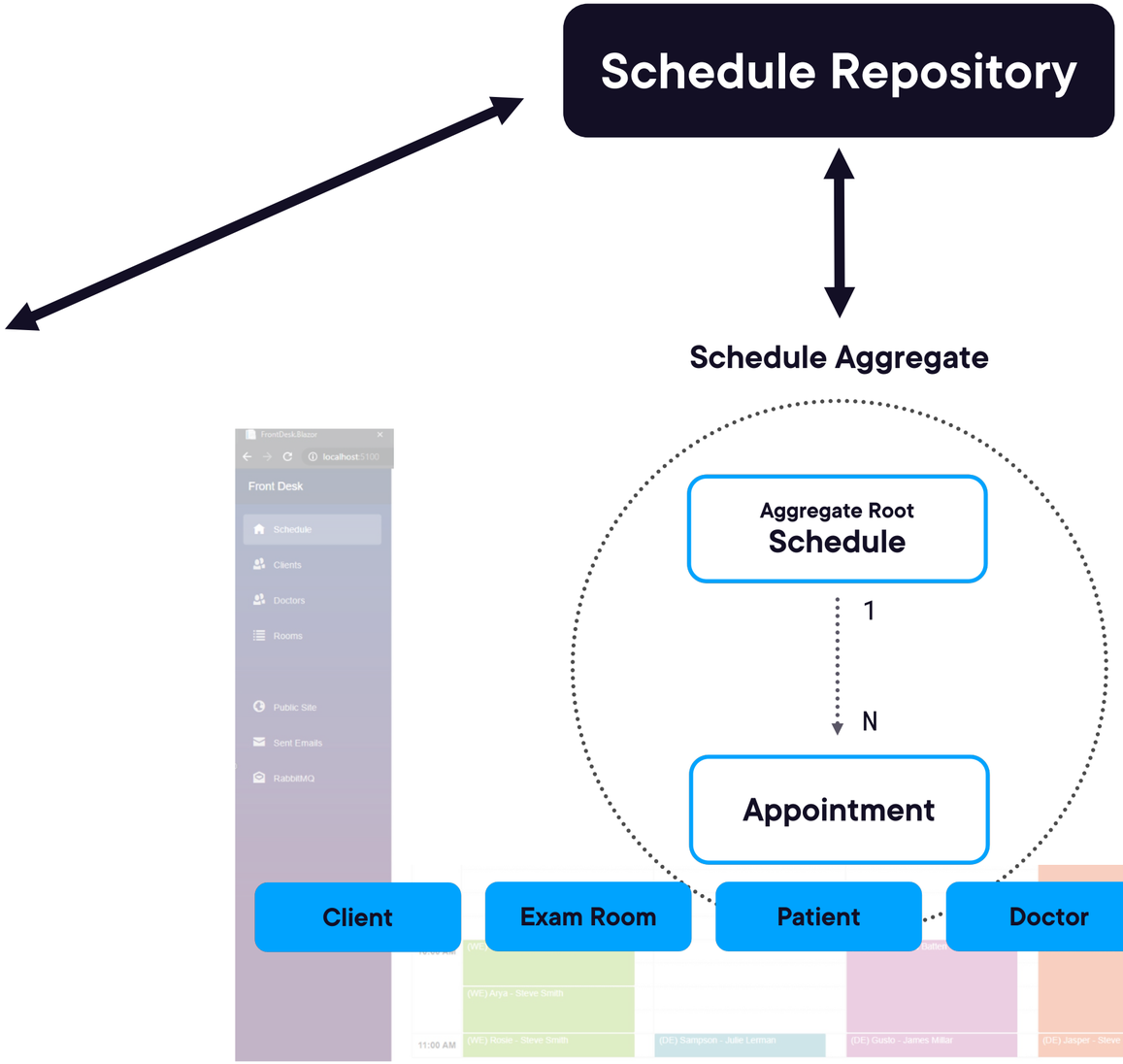


Persisting Objects



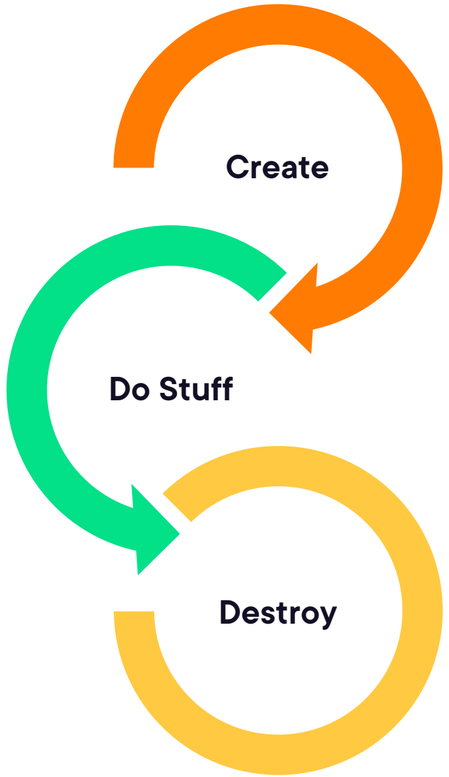
Random data access code in your system makes it difficult to maintain the integrity of your models

Persisting Objects



Object Life Cycles

No Persistence



With Persistence

Use a repository to manage the life cycle of persisted objects.

With Persistence

Persistence Ignorance

Business objects have no logic related to how data is stored and retrieved.

Eric Evans

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

Repository Benefits

- Provides common abstraction for persistence
- Promotes separation of concerns
- Communicates design decisions
- Enables testability
- Improved maintainability

Repository Tips

Think of it as an in-memory collection

Remove

Retrieve

Add

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();
}
```

Include Methods to Add and Remove

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

Custom Query Implementation Using EF Core

EfScheduleRepository.cs

```
public Schedule
GetScheduleForDateWithAppointments(int clinicId,
  DateTimeOffset date)
{
  var endDate = date.AddDays(1);

  var schedule =  _dbContext.Set<Schedule>()
    .Include(s => s.Appointments.Where( a =>
      a.TimeRange.Start > date &&
      a.TimeRange.End < endDate))

    .FirstOrDefault(schedule =>
      schedule.ClinicId == clinicId);

  return schedule;
}
```

Get a Client with Their Patients

EfClientRepository.cs

```
public Client GetClientByIdWithPatients(int clientId)
{
  var client =  _dbContext.Set<Client>()
    .Include(c => c.Patients)
    .FirstOrDefault(client => client.Id == clientId);

  return client;
}
```

General Repository Tips

Use repositories for aggregate roots only

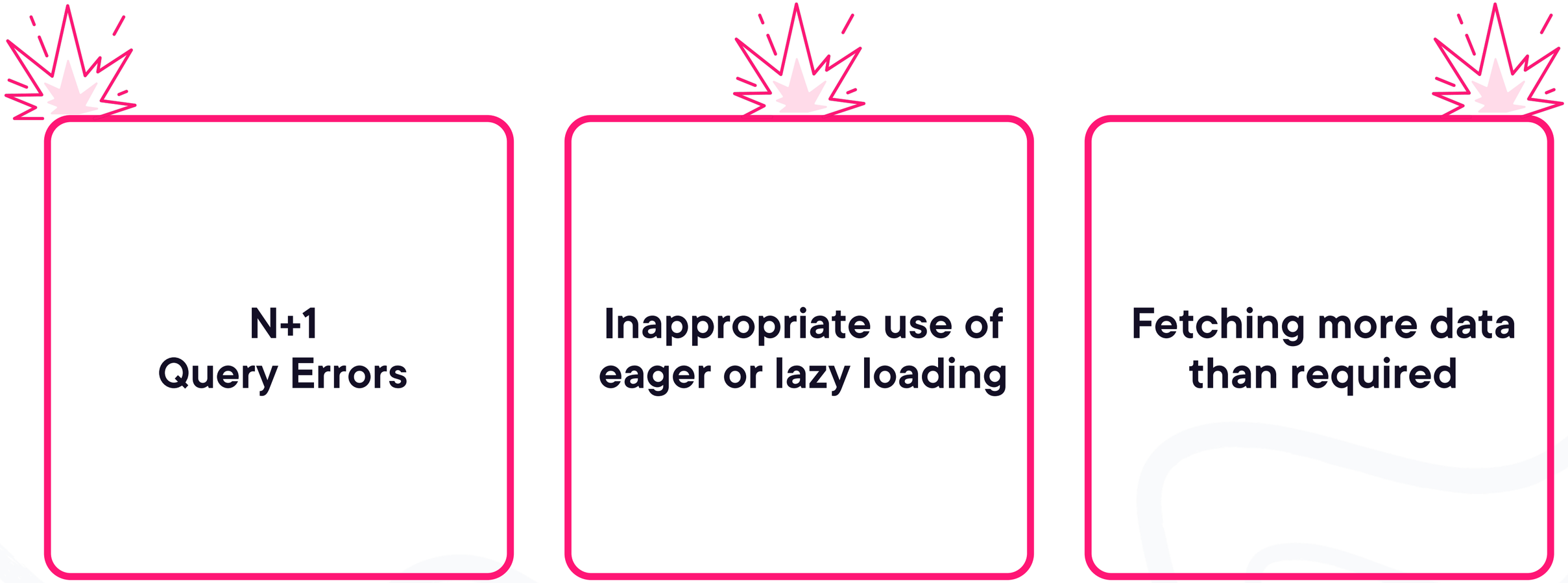
Client focuses on model, repository on persistence

Avoiding Repository Blunders

Client code can be ignorant of repository implementation

...but developers cannot

Problems Caused by Repository Logic



N+1 Query Errors

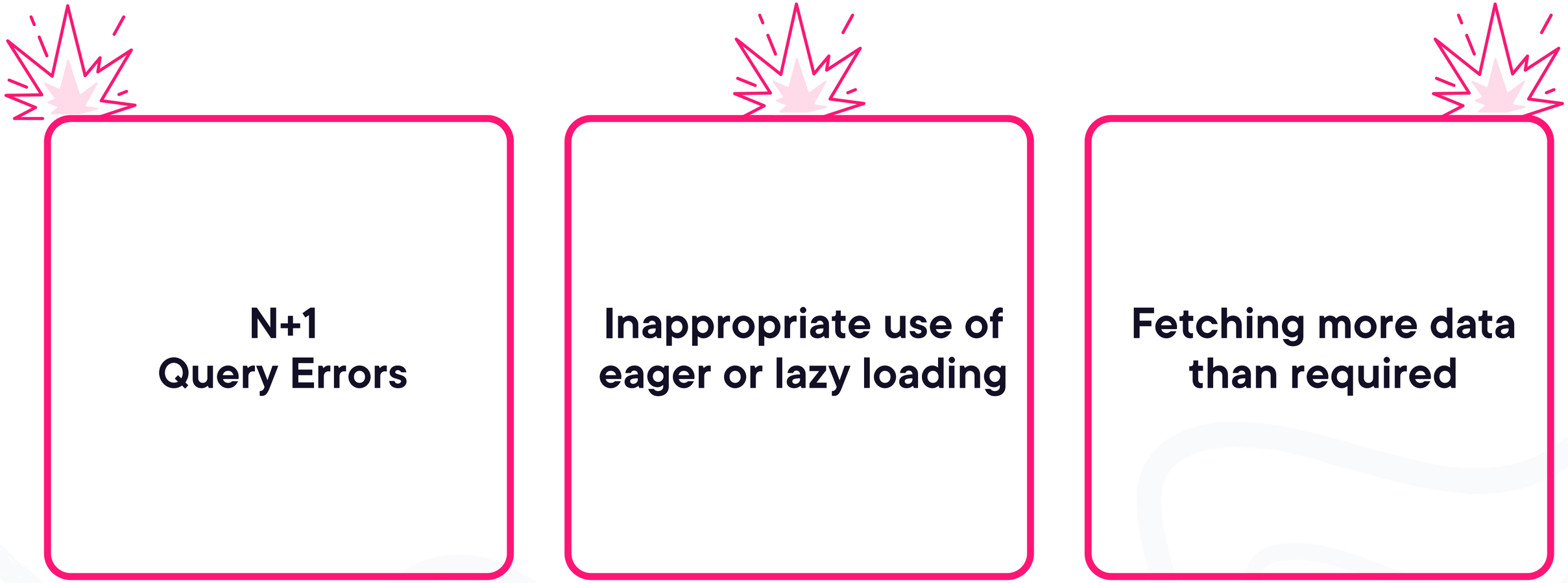
```
var clients=_context.Clients.ToList();

foreach (var client in clients)
{

   _context.Patients.Where(p=>p.ClientId==client.Id)
     .ToList();
}
```

```
select Clients.* from Clients
select Patients.* from Patients where ClientId=1
select Patients.* from Patients where ClientId=2
select Patients.* from Patients where ClientId=3
select Patients.* from Patients where ClientId=4
select Patients.* from Patients where ClientId=5
select Patients.* from Patients where ClientId=6
select Patients.* from Patients where ClientId=7
select Patients.* from Patients where ClientId=8
select Patients.* from Patients where ClientId=9
select Patients.* from Patients where ClientId=10
```

Problems Caused by Repository Logic



Database Profiling Can Surface Many Problems

Database IDE profilers

Code-based profiling or logging

3rd Party Profilers

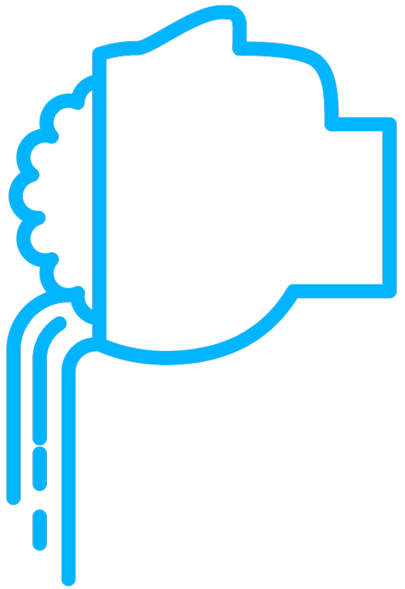
Addressing the Debates About Using Repositories

A twist on a Bjarne Stroustrup quote about langiuages:

There are two kinds of design patterns: the ones people complain about and the ones nobody uses.

NEVER
use a repository with EF Core!

ALWAYS
use a repository with EF Core!



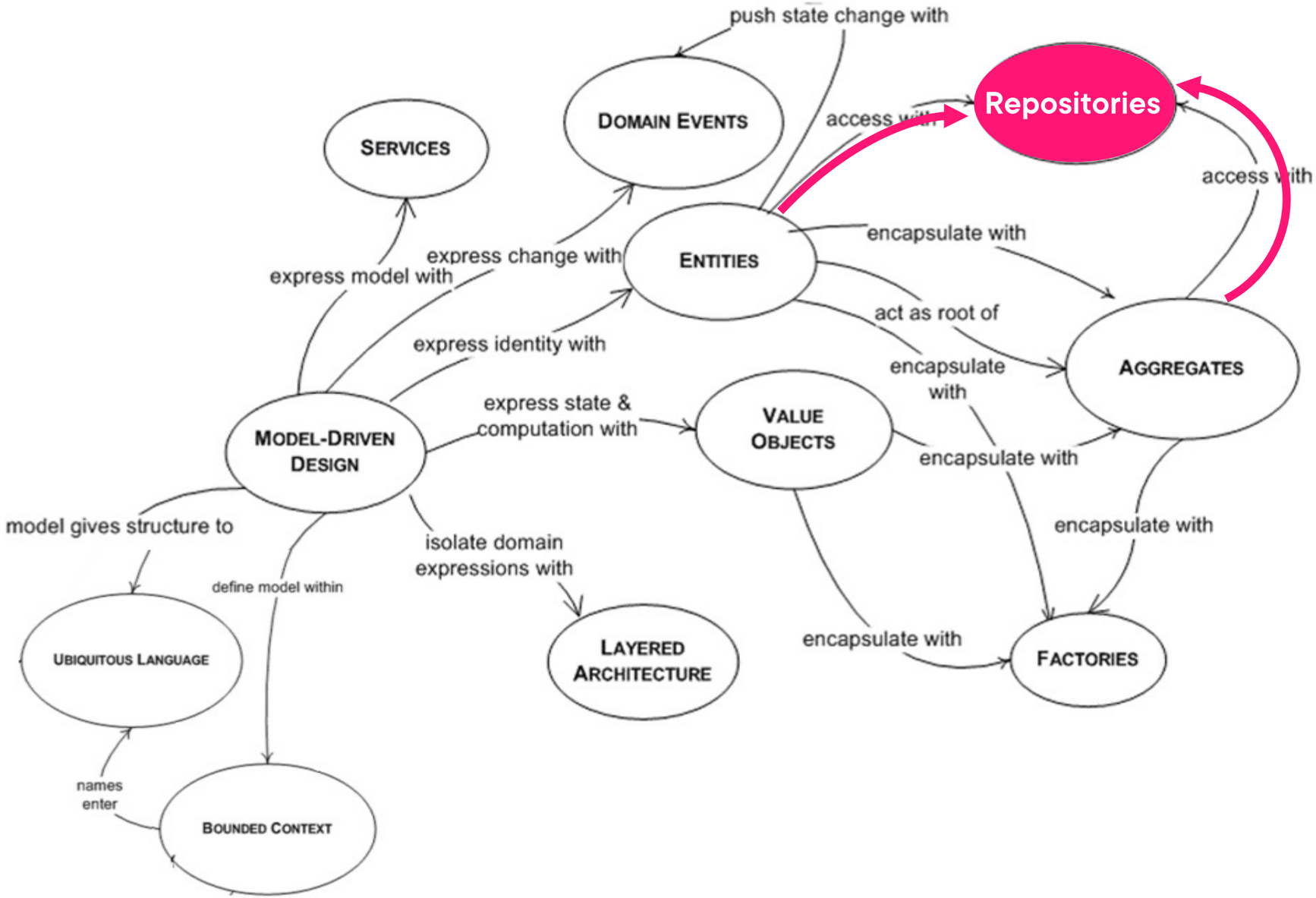
Sharing our knowledge...

...so you can make educated decisions

Repository

An abstraction your domain model uses to define what persistence needs it has

Repositories in the DDD Mind Map



A domain model should be persistence ignorant as well as ignorant of implementation details.

SOLID Principles

S Single Responsibility

O Open/Closed

L Liskov Substitution

I Interface Segregation

D Dependency Inversion

Citation: Source: SOLID Principles for C# Developers (Pluralsight course), Steve Smith

SOLID and DDD

D

Dependency Inversion

We can define an abstraction in the domain model

Implement that abstraction in another project that depends on the domain model

SOLID and DDD

I

Interface Segregation

Clients should not be forced to depend on methods they don’t use.

Prefer small, cohesive interfaces to large, “fat” ones.

Façade Pattern

Using a class to contain a complicated class or API and only expose the methods needed by your program

C# Design Patterns: Façade, David Starr, Pluralsight course

Abstracting persistence in our domain model

A persistence abstraction (a.k.a. a Repository)

Abstraction defines “what” is needed

Implementations define “how” it’s done

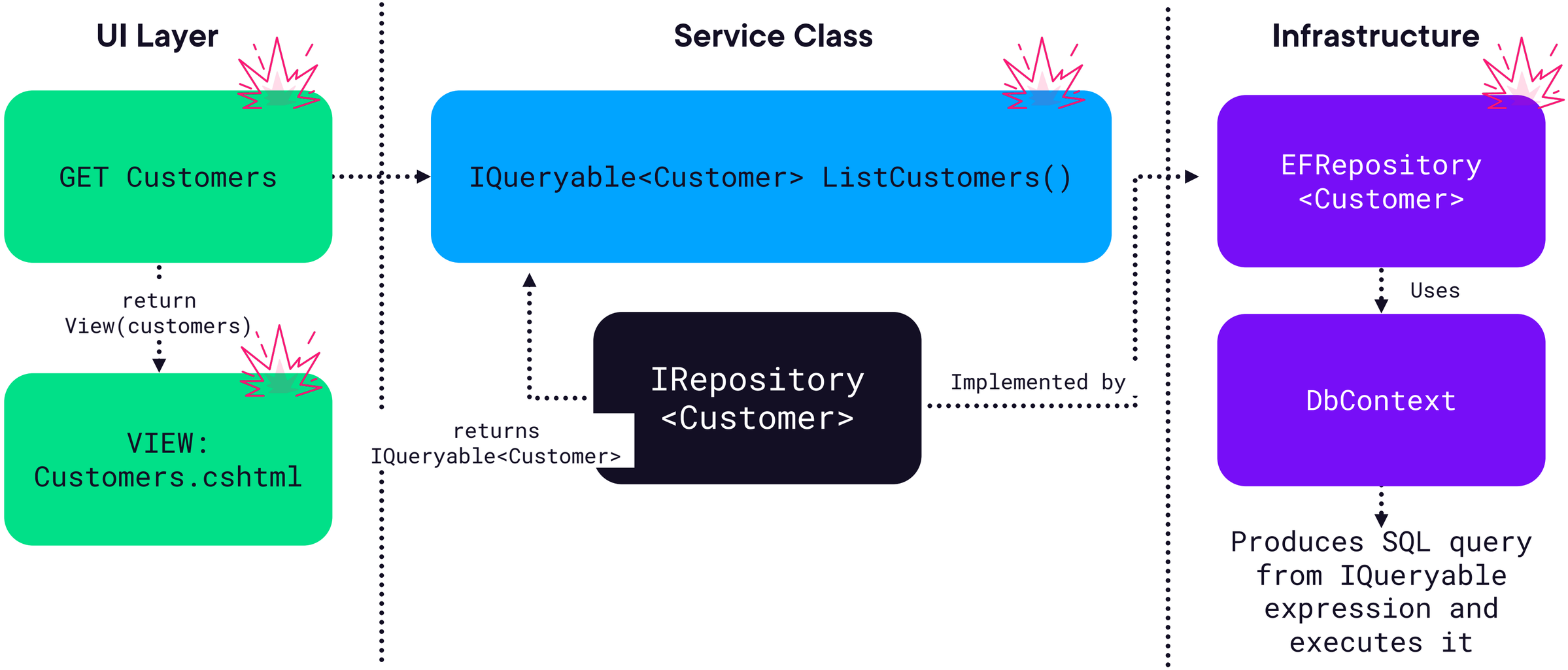
EF Core is easily used by implementation classes

DDD prevents coupling domain problems with persistence problems.

Returning IQueryables: Pros and Cons

Should repositories return IQueryable?

Where is Query Logic Defined? Using IQueryable Expression



Returning IQueryable from Repository

The Good

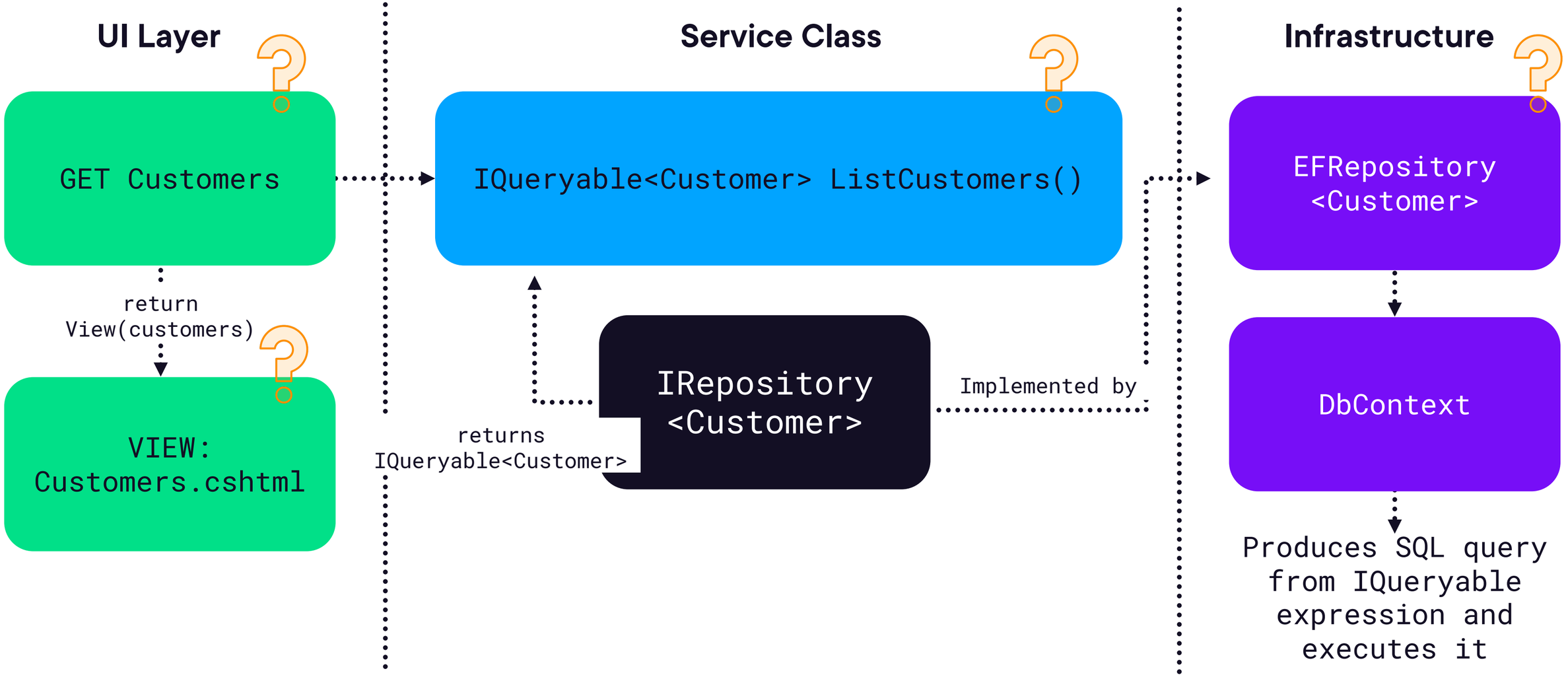
- Flexibility
- Can build query from multiple locations
- Minimal Repository code required
- Restrict data returned to just what is needed
- Reuse small set of Repository methods

vs.

The Bad and The Ugly

- Query logic spread out everywhere
- Violating Single Responsibility Principle
- Violating Separation of Concerns
- Confusion about when the query actually executes
- Code compiles, but blows up when executed

Where Is the Query Executed? Using IQueryable Expression



Returning IQueryable from Repository

The Good

- Flexibility
- Can build query from multiple locations
- Minimal Repository code required
- Restrict data returned to just what is needed
- Reuse small set of Repository methods

vs.

The Bad and The Ugly

- Query logic spread out everywhere
- Violating Single Responsibility Principle
- Violating Separation of Concerns
- Confusion about when the query actually executes
- Code compiles, but blows up when executed
- No encapsulation

Accept Arbitrary Predicates

(instead of returning IQueryable from Repository List methods)

ICustomerRepository.cs

```
public interface ICustomerRepository
{
  IEnumerable<Customer> List(Expression<Func<Customer,bool>> predicate);
}
```

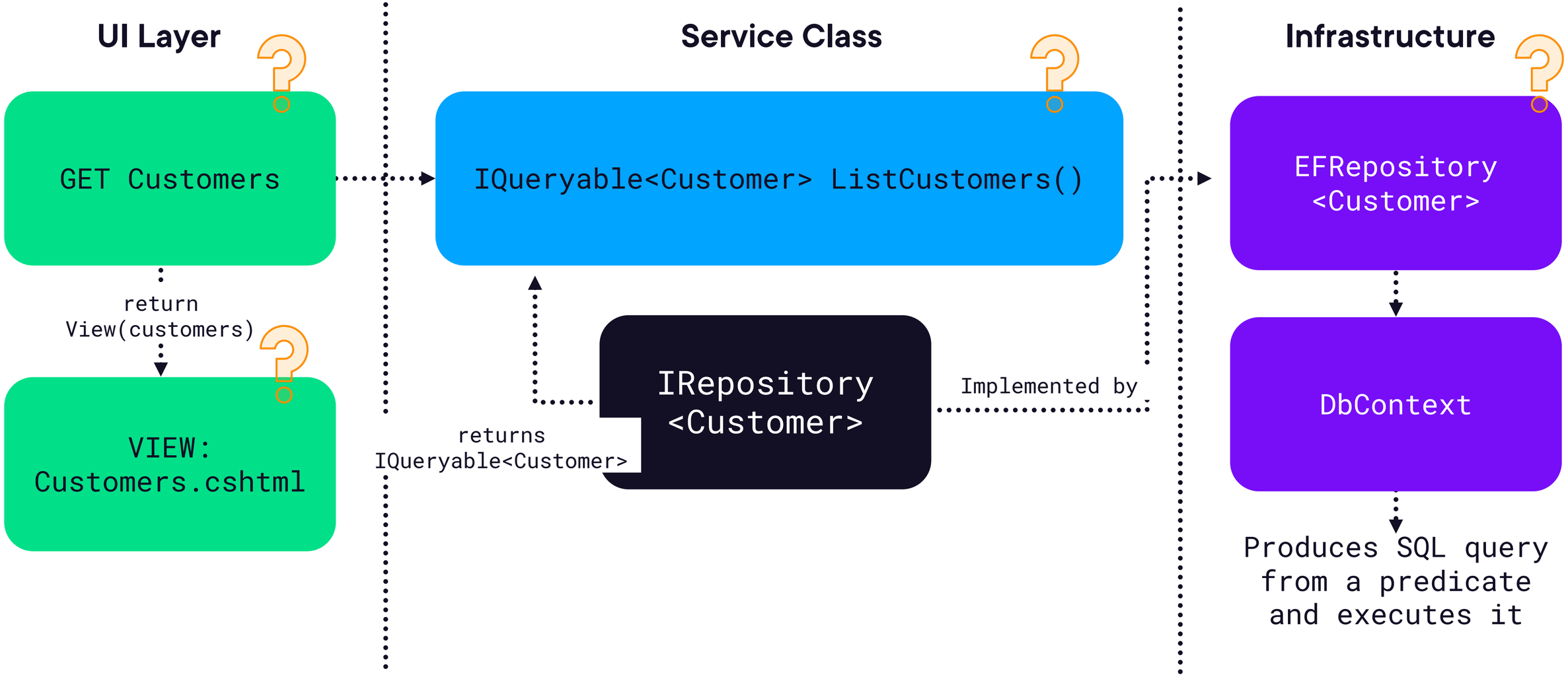
```
public IEnumerable<Customer> List(Expression<Func<Customer,bool>> predicate)
{
  return _db.Customers.Where(predicate);
}
```

Predicate

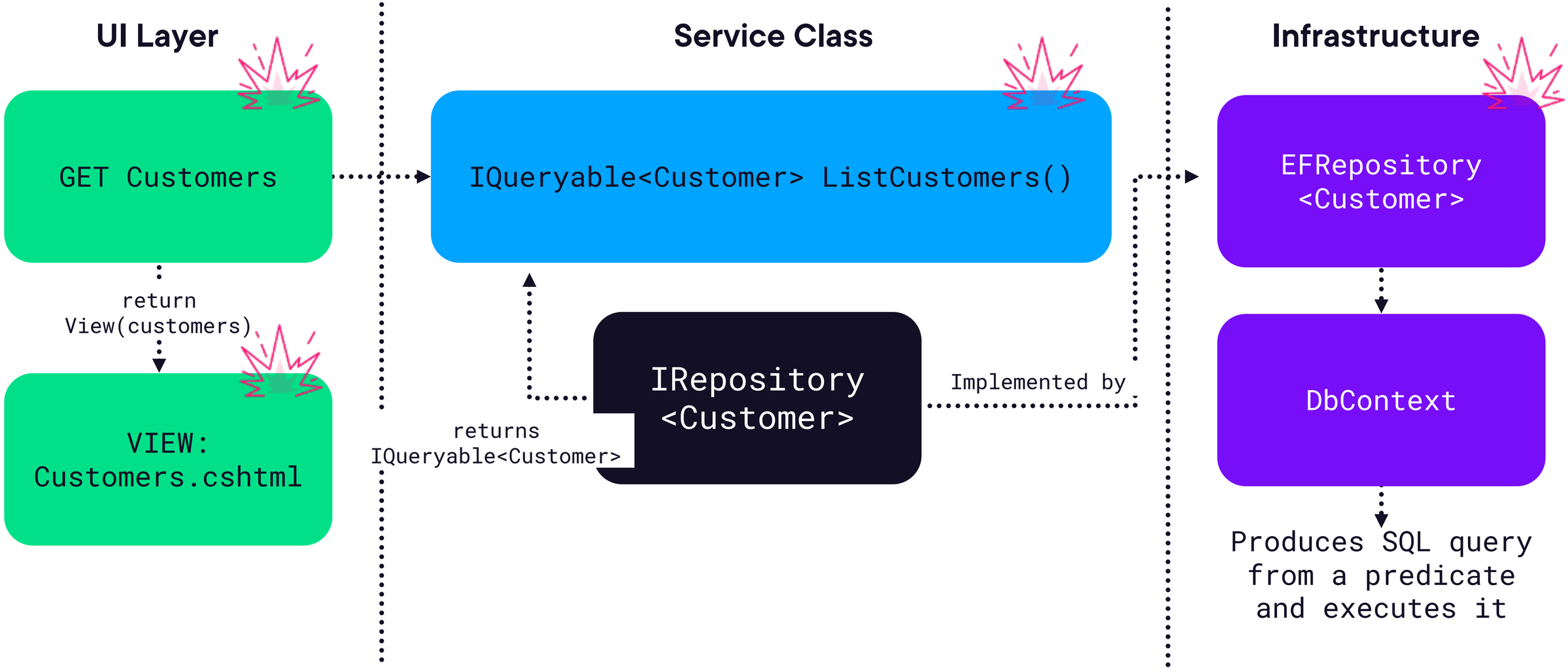
Expression used in the search condition of a query's where clause

C# Design Patterns: Façade, David Starr, Pluralsight course

When Is the Query Executed? Using Predicate



Where Is the Query Defined? Using Predicate



Passing Predicates to the Repository

The Good

Flexibility

~~Can build query from multiple locations~~

Minimal Repository code required

Restrict data returned to just what is needed

Reuse small set of Repository methods

vs.

The Bad and The Ugly

Query logic spread out everywhere

Violating Single Responsibility Principle

Violating Separation of Concerns

~~Confusion about when the query actually executes~~

Code compiles, but blows up when executed

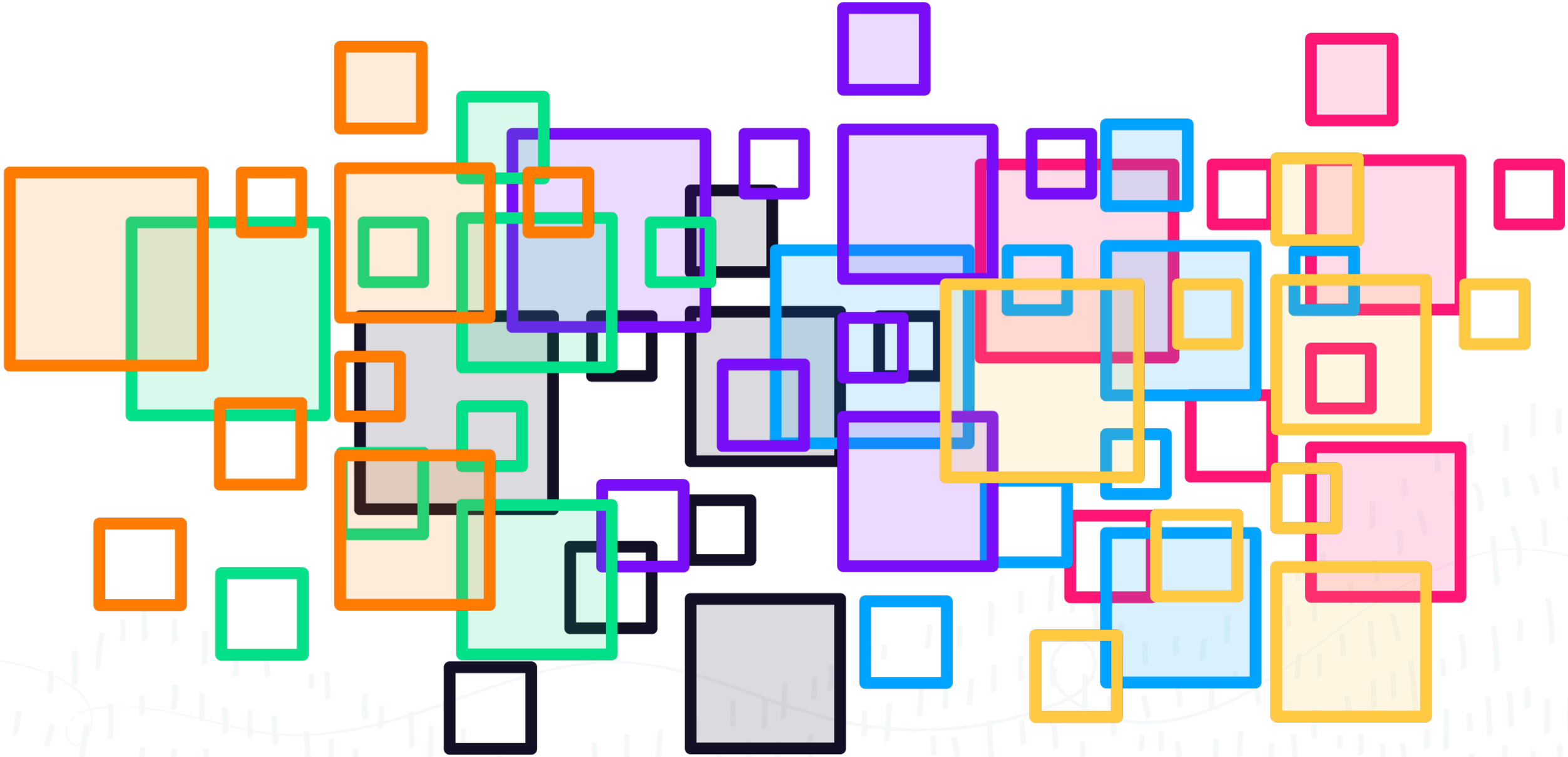
No encapsulation

One Common Solution: Custom Query Methods!

```
public interface ICustomerReadRepository
{
  Customer GetById(int id);
  List<Customer> List();

  // custom queries
  List<Customer> ListCustomersByState(string state);
  List<Customer> ListCustomersBySales(decimal minSales);
  List<Customer> ListCustomersWithOrders();
  List<Customer> ListCustomersWithAddresses();
  List<Customer> ListCustomersWithOrdersAndAddresses();
  List<Customer> ListCustomersByStateWithOrders(string state);
  List<Customer> ListCustomersByLastName(string lastName);
  List<Customer> ListCustomersByGeo(int latitude, int longitude, int radiusMiles);
  List<Customer> ListCustomersByShoeSize(string size);
  List<Customer> ListCustomersByFavoriteNetflixShow(string title);
  // and more get added all the time
}
```

Help! My Queries Are Getting Out of Control!



Considering Generic Repositories and Interfaces

Generic Repository Benefits

Promote code reuse

Generic constraint can protect aggregates

Generic Repository Trade-offs

Consistent persistence implementation, but possible unused methods

Individually crafted classes with a variety of bespoke methods

Trust your judgement and choose what makes sense for your application.

IRepository May Lead to Unused Methods

Interface for Any Repository

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

Implementing IRepository

```
class ScheduleRepo:IRepository<Schedule>
{
  public Schedule GetById(int id)
  { ...some logic... }

  public void Add(Schedule entity)
  {...some logic... }

  public void Remove(Schedule entity)
  { ... Do nothing! ... }

  public void Update(Schedule entity)
  { ...some logic... }

  public void IEnumerable<Schedule> List
  {}

}
```

A Targeted IScheduleRepository with Relevant Methods

IScheduleRepository.cs

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

Generic Repositories for Aggregate Roots

```
public class Root: IEntity
{
   public int Id ...

}
```

```
public class RootRepository : IRepository<Root>
{

  public IEnumerable<Root> List()...

  public Root GetById(int id)...

  public void Insert (Root entity) ...

  public void Update (Root entity) ...

  public void Delete (Root etity) ...

}
```

Generic Repositories for CRUD Work

```
public class Repository<TEntity>
  : IRepository<TEntity>
{

  private readonly CrudContext _context;
  private readonly DbSet<TEntity> _dbSet;


  public Repository(CrudContext context)...

  public IEnumerable< TEntity > List()...

  public Root GetById(int id)...

  public void Insert (TEntity entity) ...

  public void Update (TEntity entity) ...

  public void Delete (TEntity etity) ...

}
```

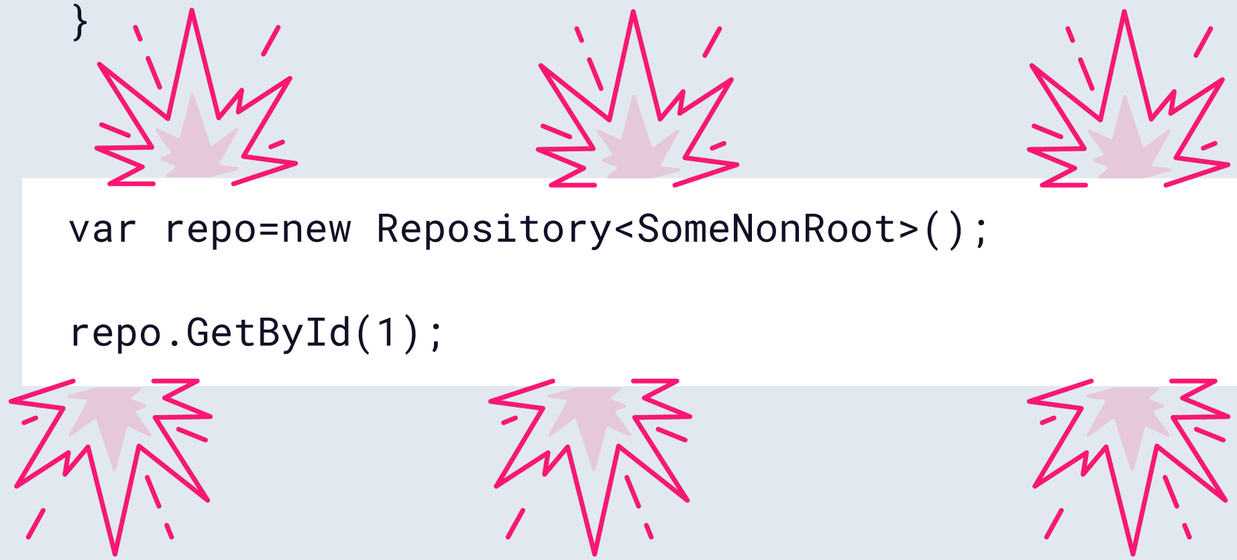
```
var repo=new Repository<Patient>();

repo.Insert(new Patient());
```

Constraining repositories to root with markers, prevents direct access to non-root entities

```
public class SomeNonRoot: IEntity
{
    public int Id...
    ...

}
```



```
var repo=new Repository<SomeNonRoot>();

repo.GetById(1);
```

Marker interfaces can provide protection to your aggregates

```
public interface IAggregateRoot : IEntity {}


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



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

Repository abstractions can get large...
sometimes too large.

Learn More About SOLID

Solid Principles for C# Developers

Steve Smith

bit.ly/solid_smith_csharp

Command Query Responsibility Segregation (CQRS)

Query repositories
focus on reading data

Command repositories
focus on writing data

Some CQRS Benefits with Minimal Effort

Query-focused repositories can benefit from caching

Command-focused repositories can benefit from queues

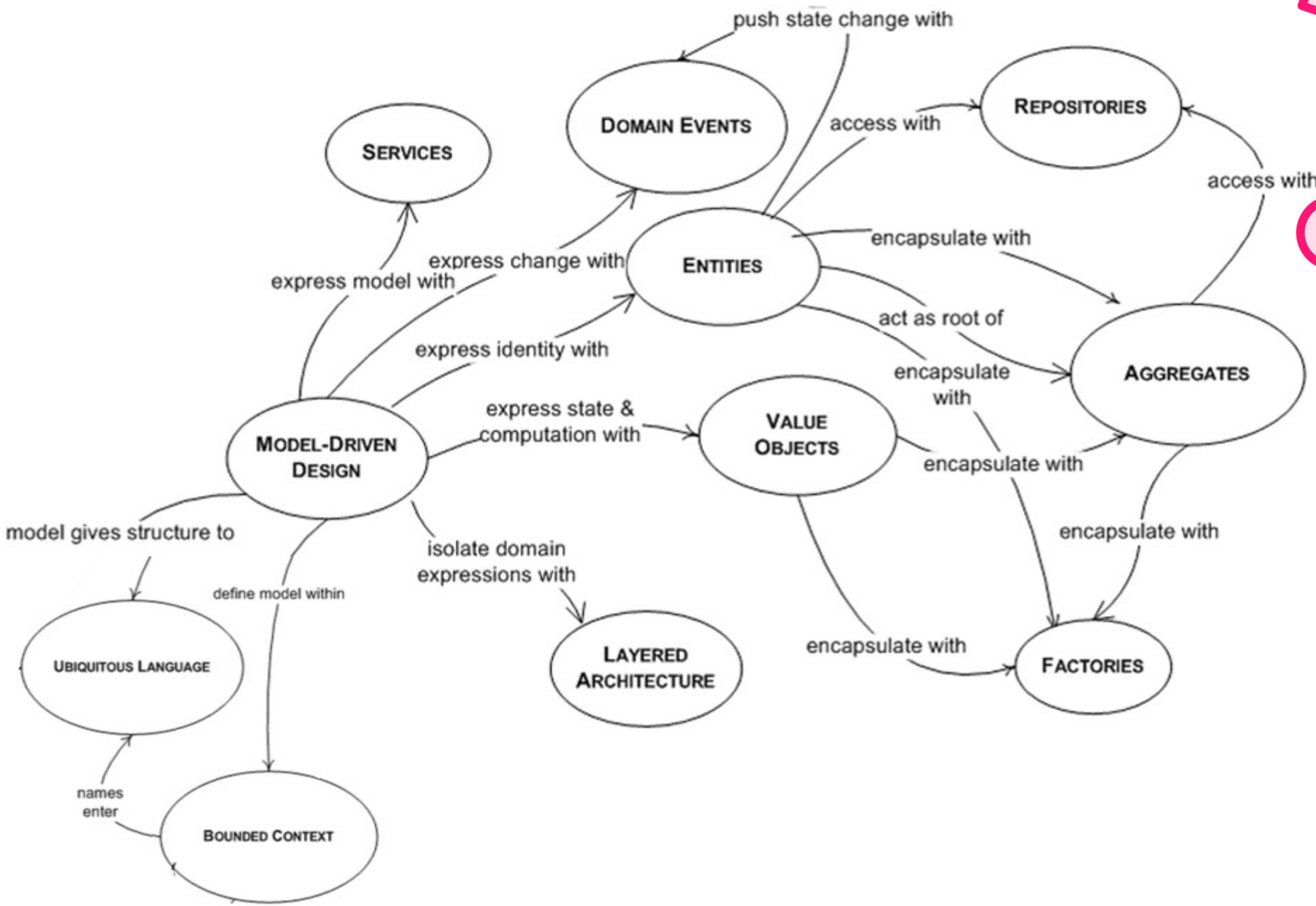
Too many read methods can interfere with caching logic.

Specification pattern can help!

Exploring Repositories in Our Application

Introducing the Specification Pattern

Specifications in the DDD Mind Map

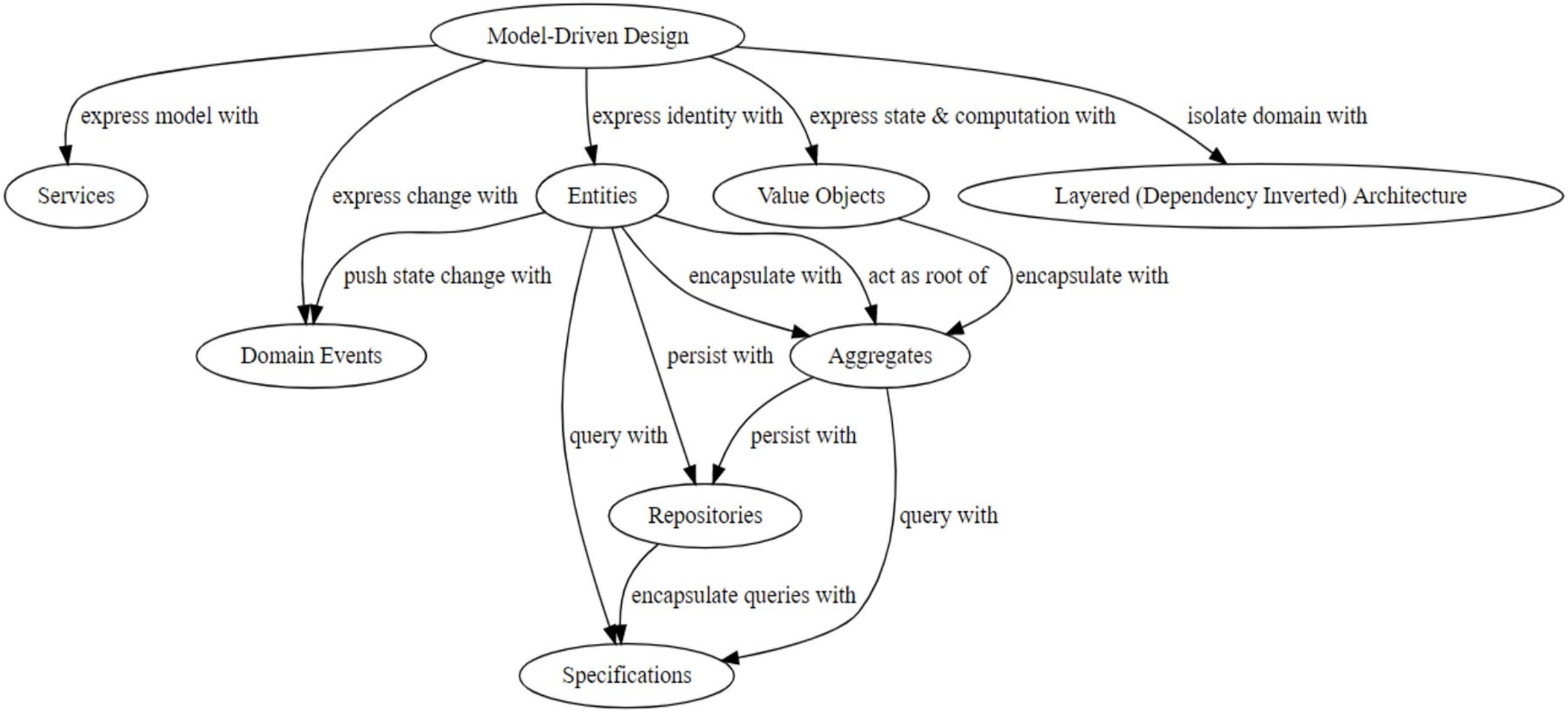


Eric Evans

“Specifications mesh smoothly with Repositories, which are the building-block mechanisms for providing query access to domain objects and encapsulating the interface to the database.”

Citation: Domain-Driven Design

Specifications in the DDD Mind Map



Specifying the State of an Object

Validation

Selection
& Querying

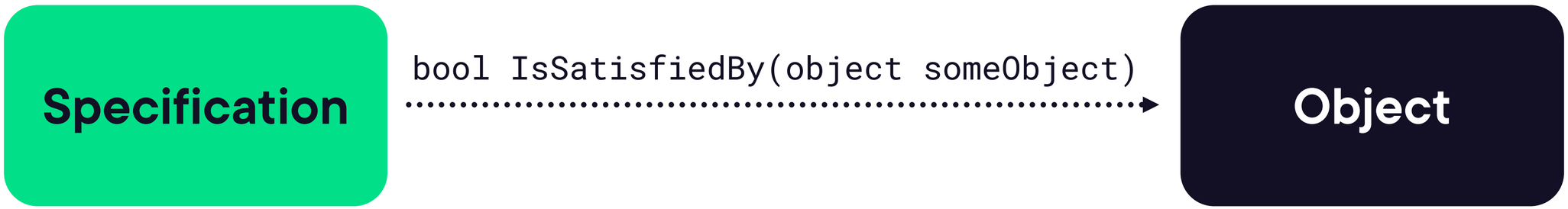
Creation for a
specific purpose

Eric Evans

“Create explicit predicate-like Value Objects for specialized purposes. A Specification is a predicate that determines if an object satisfies some criteria.”

Citation: Domain-Driven Design

A Basic Specification

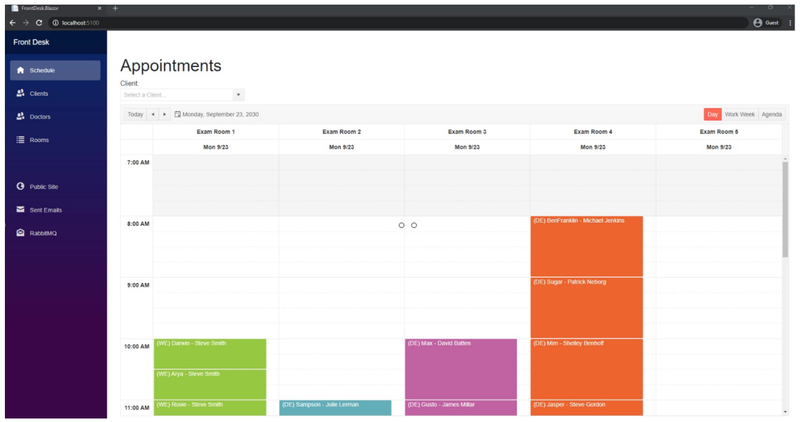


Note:

Criteria evaluated in memory

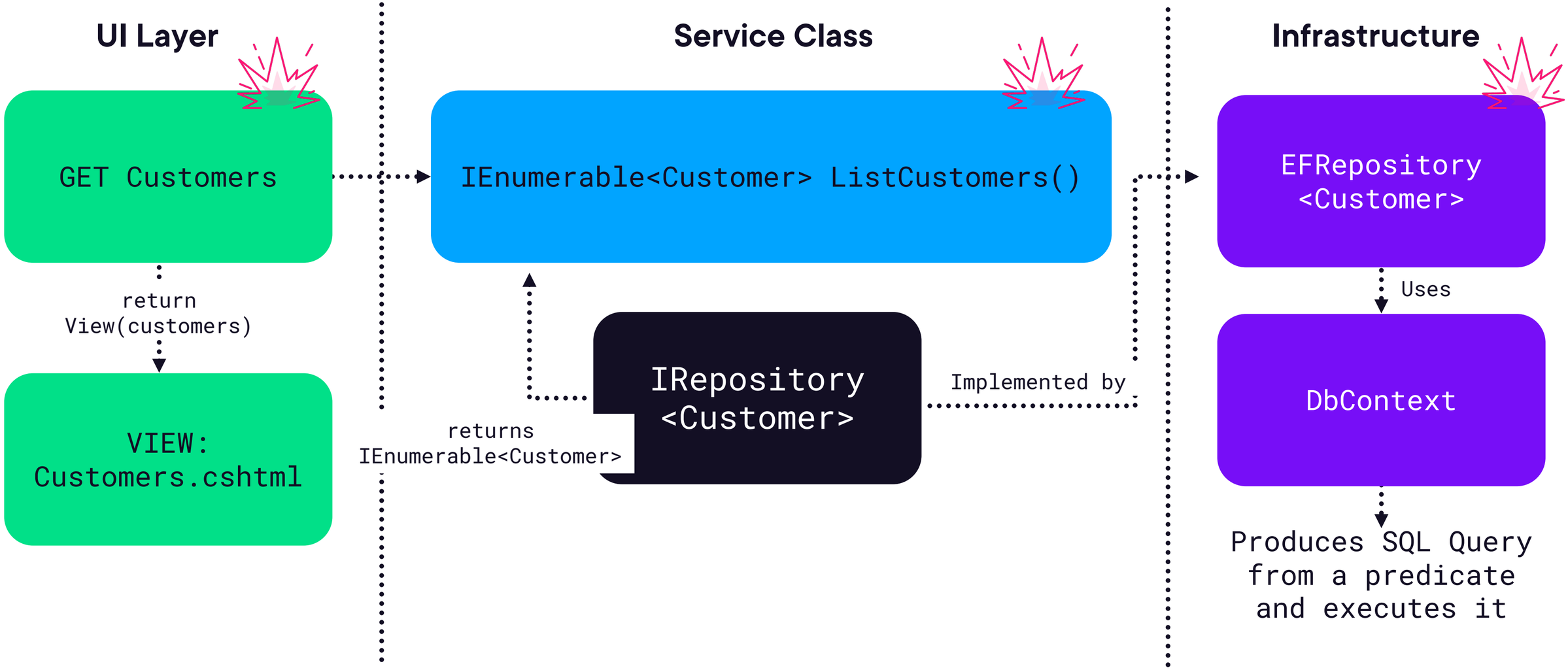
Combining Specifications with ORMs

Persisting Objects

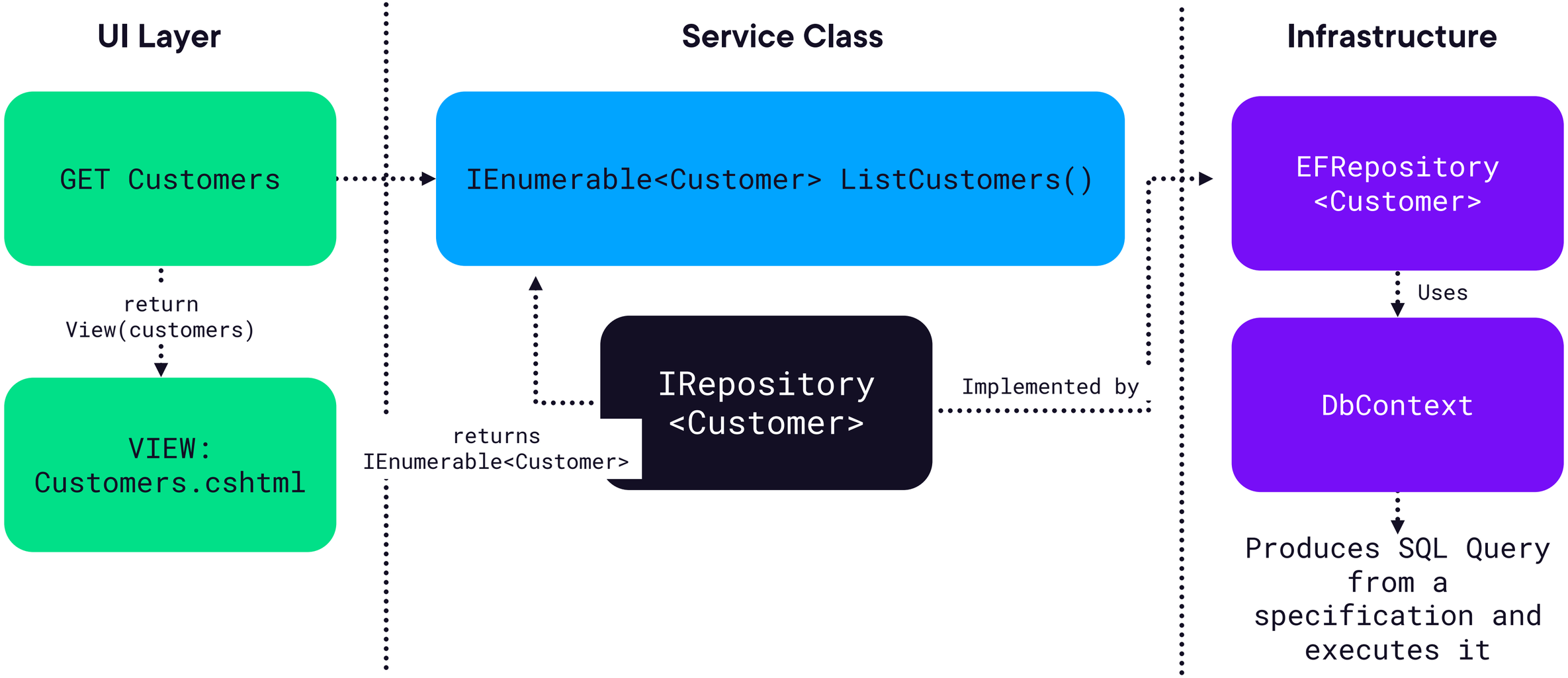


Random data access code in your system makes it difficult to maintain the integrity of your models

Where Is Query Logic Defined? Using Custom Predicate



Where Is Query Logic Defined? Using Specification



Typed Repository Interfaces Provide Needed Query Methods

```
public interface ICustomerReadRepository
{
  Customer GetById(int id);
  List<Customer> List();

  // custom queries
  List<Customer> ListCustomersByState(string state);
  List<Customer> ListCustomersBySales(decimal minSales);
  List<Customer> ListCustomersWithOrders();
  List<Customer> ListCustomersWithAddresses();
  List<Customer> ListCustomersWithOrdersAndAddresses();
  List<Customer> ListCustomersByStateWithOrders(string state);
  List<Customer> ListCustomersByLastName(string lastName);
  List<Customer> ListCustomersByGeo(int latitude, int longitude, int radiusMiles);
  List<Customer> ListCustomersByShoeSize(string size);
  List<Customer> ListCustomersByFavoriteNetflixShow(string title);
  // and more get added all the time
}
```

Typed Repository Interfaces Provide Needed Query Methods

```
public interface ICustomerReadRepository
{
  Customer GetById(int id);
  List<Customer> List();


  List<Customer> ListCustomersBySpecification(specification);
}
```

Some More Specification Benefits

Named classes via ubiquitous language

Reusable

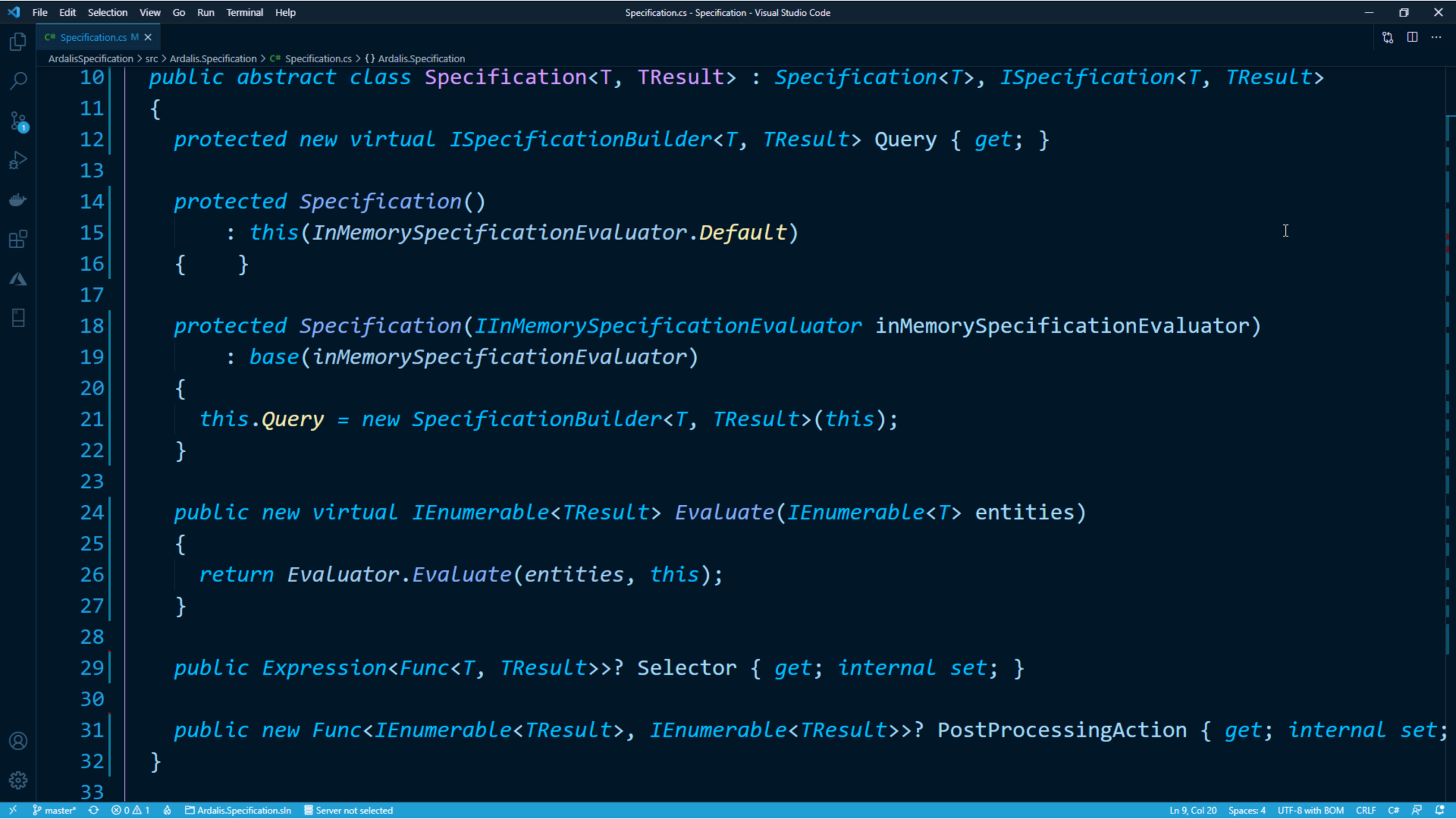
Separate persistence from domain model and UI

Keep business logic out of persistence layer and database

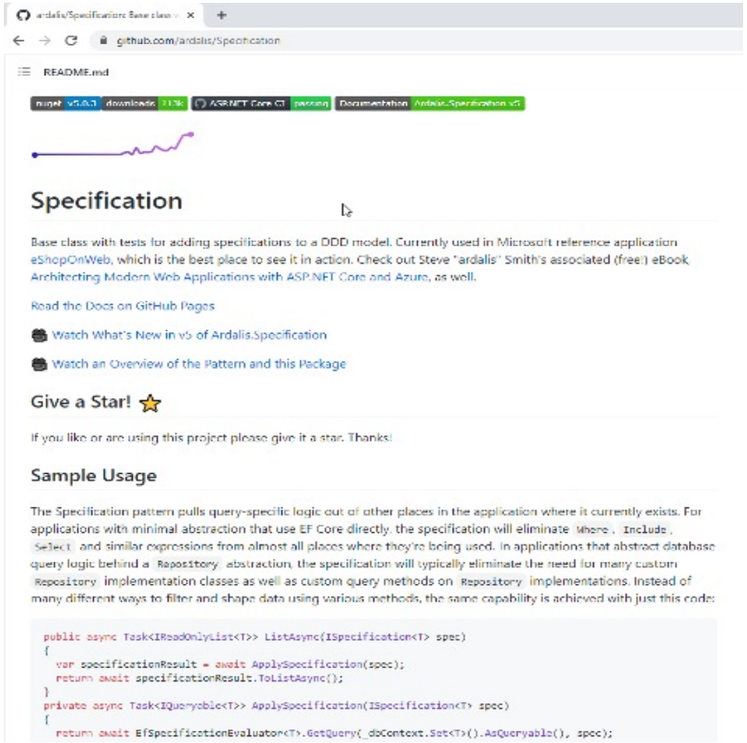
Help entities and aggregates follow Single Responsibility Principle (SRP)

Implementing Specifications

```
public abstract class BaseSpecification
{
  // TODO: Implement it yourself
}
```


File Edit Selection View Go Run Terminal Help
Specification.cs - Specification - Visual Studio Code
Specification.cs M
ArdalisSpecification > src > Ardalis.Specification > Specification.cs > {} Ardalis.Specification
public abstract class Specification<T, TResult> : Specification<T>, ISpecification<T, TResult>
{
protected new virtual ISpecificationBuilder<T, TResult> Query { get; }
protected Specification()
: this(InMemorySpecificationEvaluator.Default)
{ }
protected Specification(IInMemorySpecificationEvaluator inMemorySpecificationEvaluator)
: base(inMemorySpecificationEvaluator)
{
this.Query = new SpecificationBuilder<T, TResult>(this);
}
public new virtual IEnumerable<TResult> Evaluate(IEnumerable<T> entities)
{
return Evaluator.Evaluate(entities, this);
}
public Expression<Func<T, TResult>>? Selector { get; internal set; }
public new Func<IEnumerable<TResult>, IEnumerable<TResult>>? PostProcessingAction { get; internal set;
}
master* 0 1 Ardalis.Specification.sln Server not selected
Ln 9, Col 20 Spaces: 4 UTF-8 with BOM CRLF C#

Steve's Specification Pattern Base Class

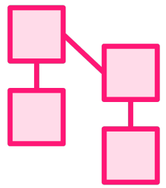


GitHub Project
github.com/ardalis/Specification

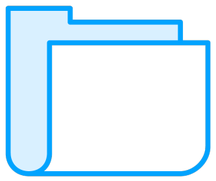
NuGet package
nuget.org/packages/Ardalis.Specification/

Implementing Specification Classes

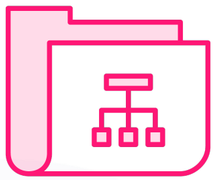
You will need to write the rules of your specifications



The classes belong in your domain model



If only a few, organize in root Specifications folder



Or, along side your aggregates in their folders

Custom Specification Inheriting from Base

```
public class ScheduleIdWithAppointmentsSpec : Specification<Schedule>
{
  public ScheduleByIdWithAppointmentsSpec(Guid scheduleId)
  {
    Query
      .Where(schedule => schedule.Id == scheduleId)
      .Include(schedule => schedule.Appointments);
  }
}
```

Examples of Applying Specifications in EF Core

```
dbContext.Customers.WithSpecification(specification).ToListAsync();
```

```
dbContext.Customers.WithSpecification(specification).FirstOrDefaultAsync();
```

```
dbContext.Customers.WithSpecification(specification)
                   .Select("whatever your expression is").ToListAsync();
```

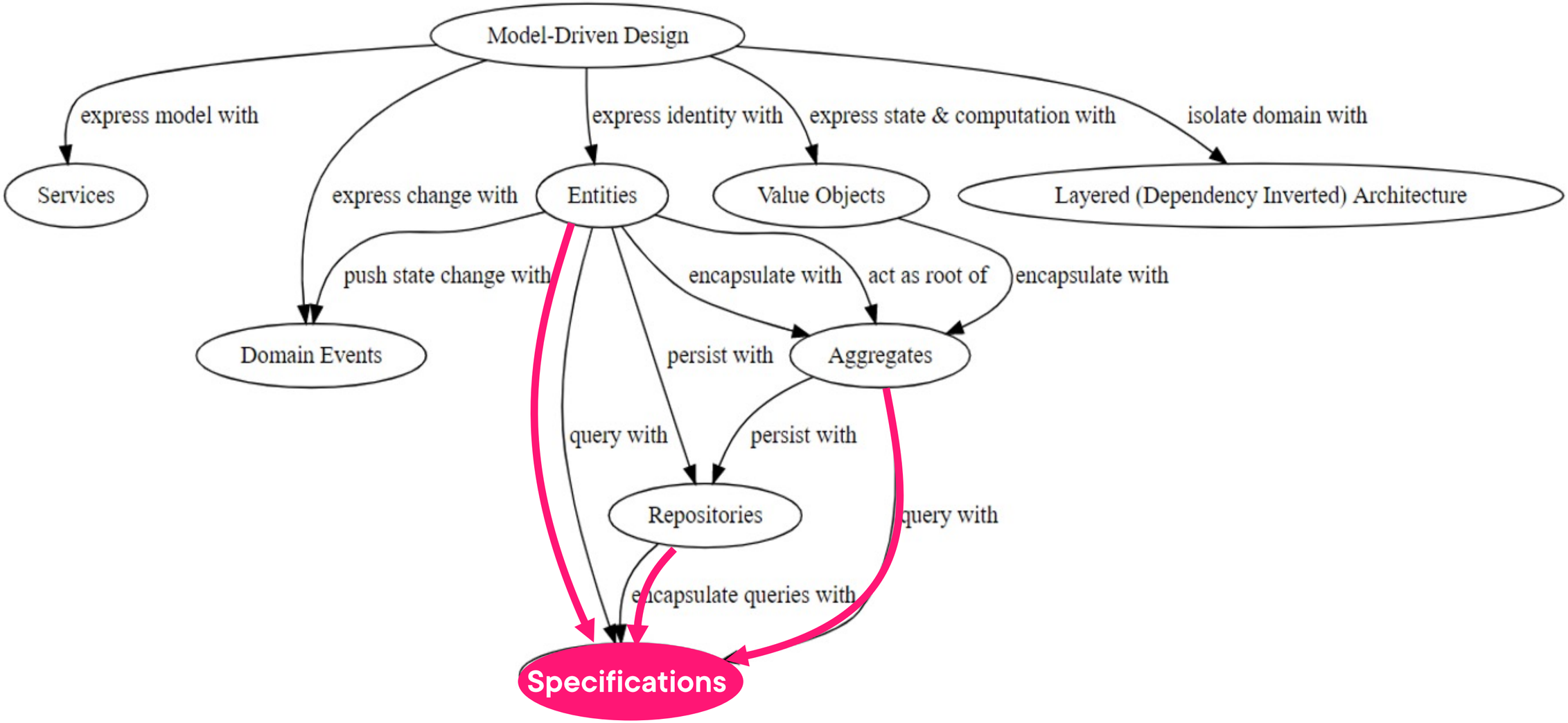
```
dbContext.Customers.WithSpecification(specification)
                   .UseWhateverExtensionsAvailableForIQueryable;
```

Using Specifications in Your Code

```
var clientSpec = new
ClientByIdIncludePatientsSpecification(appointment.ClientId);

var client = await _clientRepository.GetBySpecAsync(clientSpec);
```

Specifications in the DDD Mind Map



Using Specifications with Repositories in Our App

Module Review and Resources

Repository

A class that encapsulates the data persistence for an aggregate root

Specification Pattern

A method of encapsulating a business rule so that it can be passed to other methods which are responsible for applying it

Persistence Ignorance

Objects are unaware of where their data comes from or goes to

ACID

Atomic, Consistent, Isolated, and Durable

SOLID

A set of five software design patterns

Key Takeaways

Repository pattern and the DDD mind map

Benefits of and tips for building repositories

Repository debates:
Use them? Return IQueryables?

Specification pattern with repositories

Sample code is filled with great examples!

Resources Referenced in This Module

On Pluralsight: SOLID Principles for C# Developers
bit.ly/solid_smith_csharp

On Pluralsight: Entity Framework in the Enterprise
bit.ly/PS-EFEnterprise (See “The Great Repository Debate” module)

Specification Pattern Base Class
github.com/ardalis/Specification

Resources Referenced in This Module

Avoid In-Memory Databases for Tests
jimmybogard.com/avoid-in-memory-databases-for-tests/

On Pluralsight: C# Design Patterns: Façade by David Starr
app.pluralsight.com/library/courses/csharp-design-patterns-facade

We'll learn two more critical pieces of the DDD mind map.