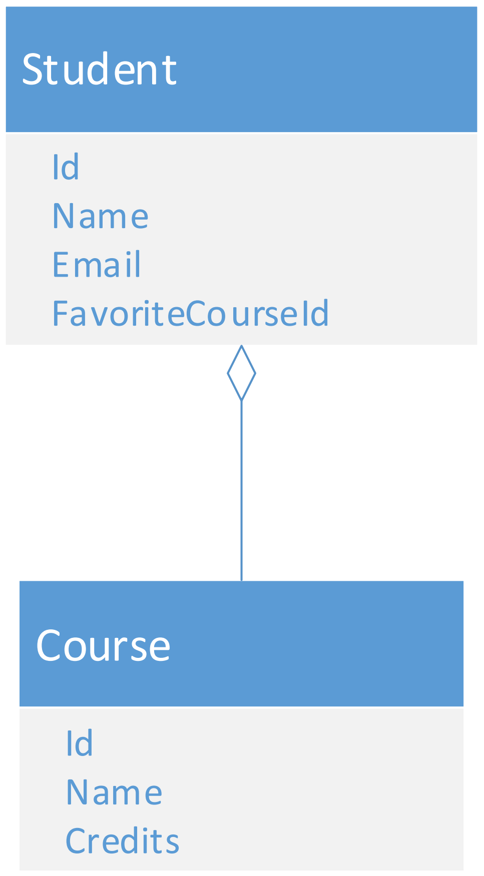
Mapping Backing Fields

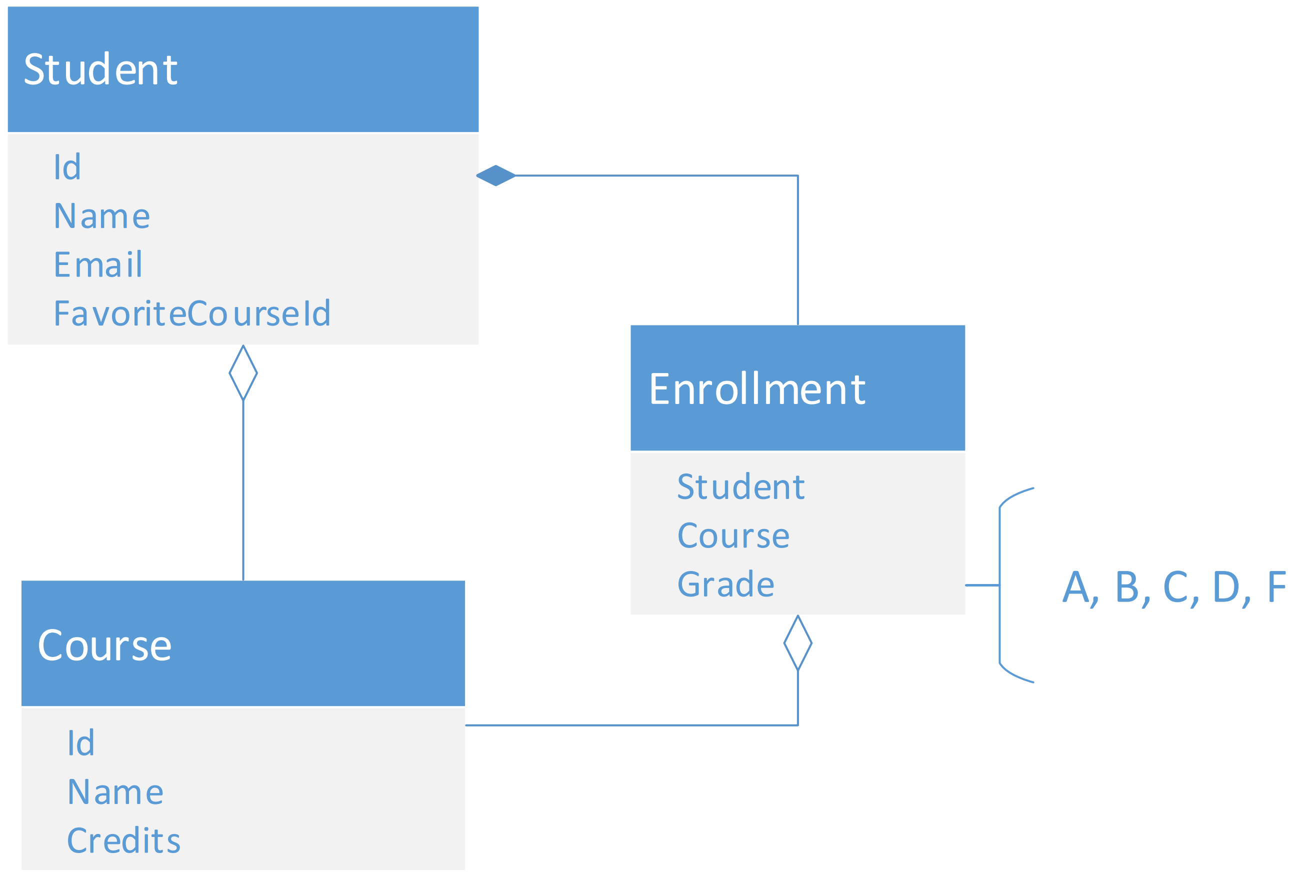
Vladimir Khorikov

@vkhorikov www.enterprisecraftsmanship.com

Introducing a One-to-many Relationship



Introducing a One-to-many Relationship



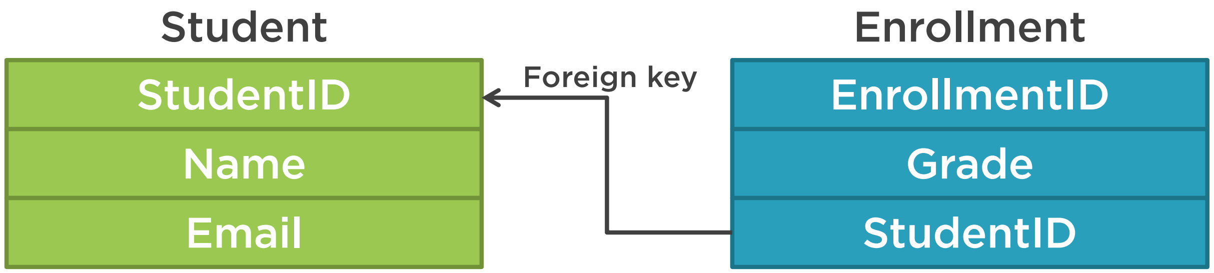
Demo

Add the one-to-many relationship the conventional way

Discuss its drawbacks

Refactor toward encapsulation

Recap: Introducing a One-to-many Relationship



```
student.Enrollments
```

```
enrollment.Student
```

Recap: Introducing a One-to-many Relationship

```
public virtual ICollection<Enrollment> Enrollments { get; set; }
```

~~student.Enrollments.Add(~~
~~new Enrollment(course, student, grade));~~

✗ No encapsulation

```
student.Enrollments.Clear();          student.Enrollments = null;
```

✗ Meaningless operations

✗ To wide API surface area

Recap: Hiding the Collection Behind a Backing Field

✓ **Encapsulated the enrollment collection**

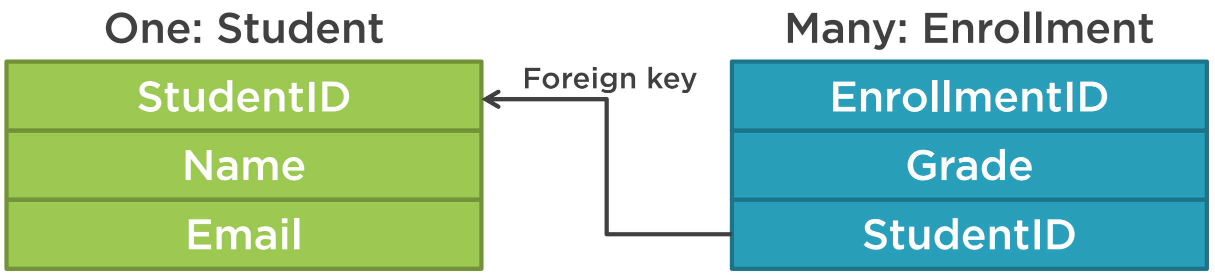
```
private readonly List<Enrollment> _enrollments = new List<Enrollment>();
public virtual IReadOnlyList<Enrollment> Enrollments => _enrollments.ToList();
```

✓ **Removed the setter**

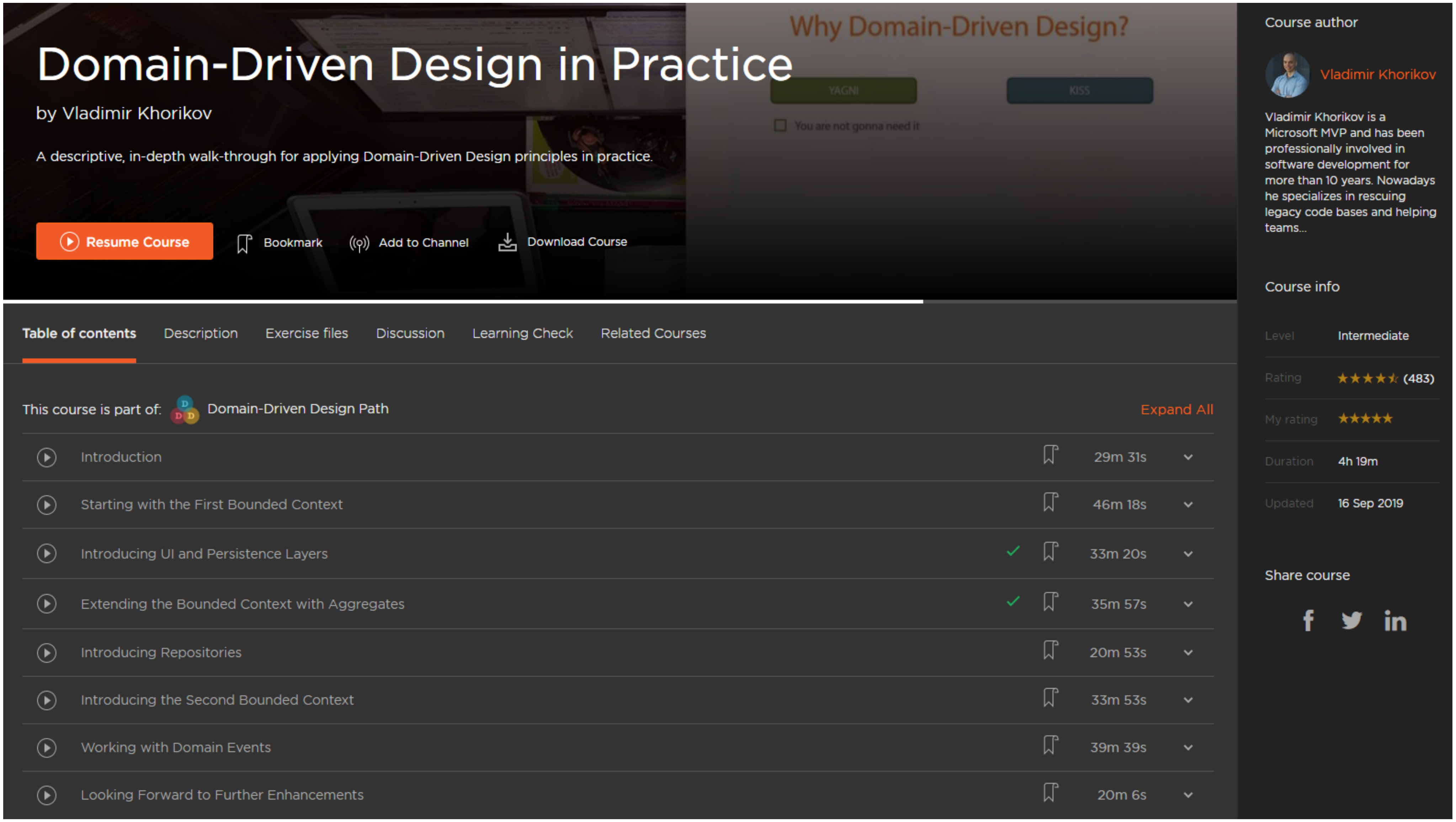
✓ **Made the collection read-only**

Recap: Hiding the Collection Behind a Backing Field

```
public void EnrollIn(Course course, Grade grade)
{
    var enrollment = new Enrollment(course, this, grade);
    _enrollments.Add(enrollment);
}
```



- ✓ Student is responsible for creation and deletion of its enrollments
- ✓ Enrollment is an internal entity



Domain-Driven Design in Practice

by Vladimir Khorikov

A descriptive, in-depth walk-through for applying Domain-Driven Design principles in practice.

This course is part of: Domain-Driven Design Path

Course author

Vladimir Khorikov

Vladimir Khorikov is a Microsoft MVP and has been professionally involved in software development for more than 10 years. Nowadays he specializes in rescuing legacy code bases and helping teams...

Course info

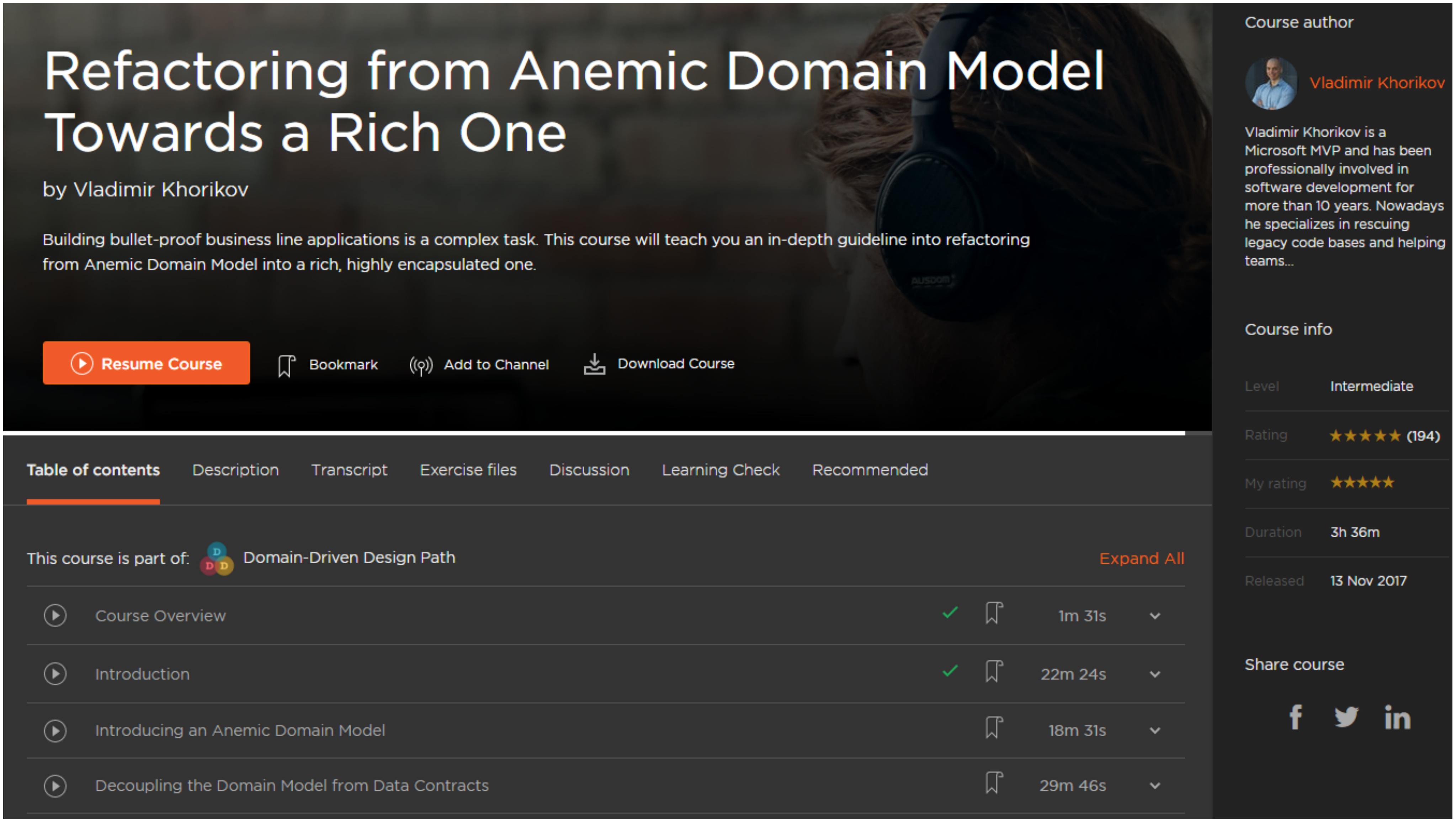
Level	Intermediate
Rating	★★★★☆ (483)
My rating	★★★★★
Duration	4h 19m
Updated	16 Sep 2019

Share course

Recap: Hiding the Collection Behind a Backing Field

```
public sealed class SchoolContext : DbContext
{
    public DbSet<Student> Students { get; set; }
    public DbSet<Course> Courses { get; set; }
}
```

⚠ **Don’t expose internal entities as DbSets**



Refactoring from Anemic Domain Model Towards a Rich One

by Vladimir Khorikov

Building bullet-proof business line applications is a complex task. This course will teach you an in-depth guideline into refactoring from Anemic Domain Model into a rich, highly encapsulated one.

This course is part of: Domain-Driven Design Path

Course Overview	✓	1m 31s
Introduction	✓	22m 24s
Introducing an Anemic Domain Model		18m 31s
Decoupling the Domain Model from Data Contracts		29m 46s

Vladimir Khorikov

Vladimir Khorikov is a Microsoft MVP and has been professionally involved in software development for more than 10 years. Nowadays he specializes in rescuing legacy code bases and helping teams...

Level	Intermediate
Rating	★★★★★ (194)
My rating	★★★★★
Duration	3h 36m
Released	13 Nov 2017

Introducing a Collection Invariant

New invariant

Can’t enroll a student into the same course twice

Recap: Introducing a Collection Invariant

New invariant

Can’t enroll a student into the same course twice

```
public string EnrollIn(Course course, Grade grade)
{
    if (_enrollments.Any(x => x.Course == course))
        return $"Already enrolled in course '{course.Name}'";

    var enrollment = new Enrollment(course, this, grade);
    _enrollments.Add(enrollment);

    return "OK";
}
```

EF Core only intersects calls to navigation properties

Recap: Introducing a Collection Invariant

```
public class EFStudentProxy : Student {
    public override IReadOnlyList<Enrollment> Enrollments {
        get {
            InitializeBackingField(_enrollments);
            return base.Enrollments;
        }
    }
}
```

Loads data

Passes control to your code

```
public class NHStudentProxy : Student {
    public override IReadOnlyList<Enrollment> Enrollments {
        get {
            InitializeBackingField(_enrollments);
            return base.Enrollments;
        }
    }

    public override string EnrollIn(Course course, Grade grade) {
        InitializeBackingField(_enrollments);
        base.EnrollIn(course, grade)
    }
}
```

Loads data

Passes control to your code

Recap: Introducing a Collection Invariant

Have to forgo lazy loading in some scenarios

- Include().SingleOrDefault()
- Find() with explicit loading of relationships

✓ Use Find() by default

Recap: Introducing a Collection Invariant

```
public string EnrollIn(Course course, Grade grade)
{
    if (Enrollments.Any(x => x.Course == course))
        return $"Already enrolled in course '{course.Name}'";

    var enrollment = new Enrollment(course, this, grade);
    _enrollments.Add(enrollment);

    return "OK";
}
```

Property, not backing field

✗ **Error-prone**

✗ **Hard to understand**

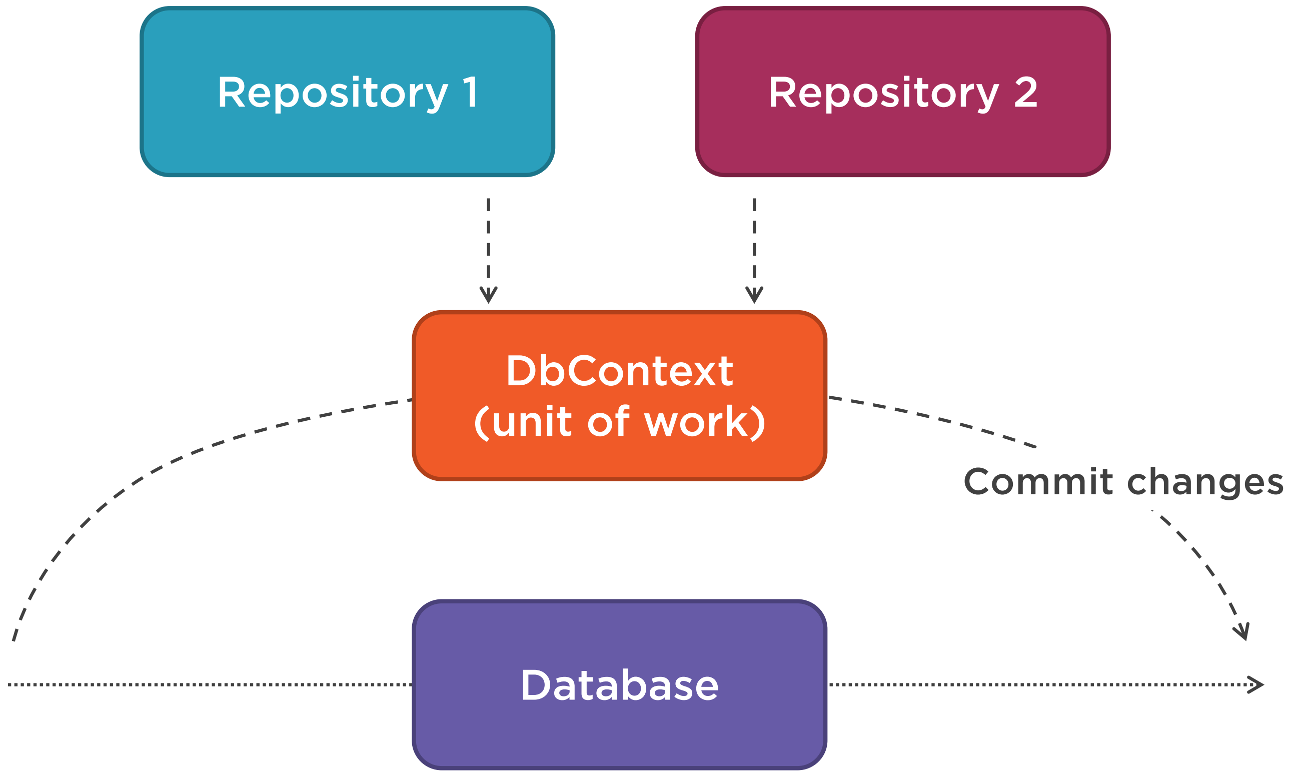
✗ **Violation of the SoC principle**

Recap: Introducing a Collection Invariant

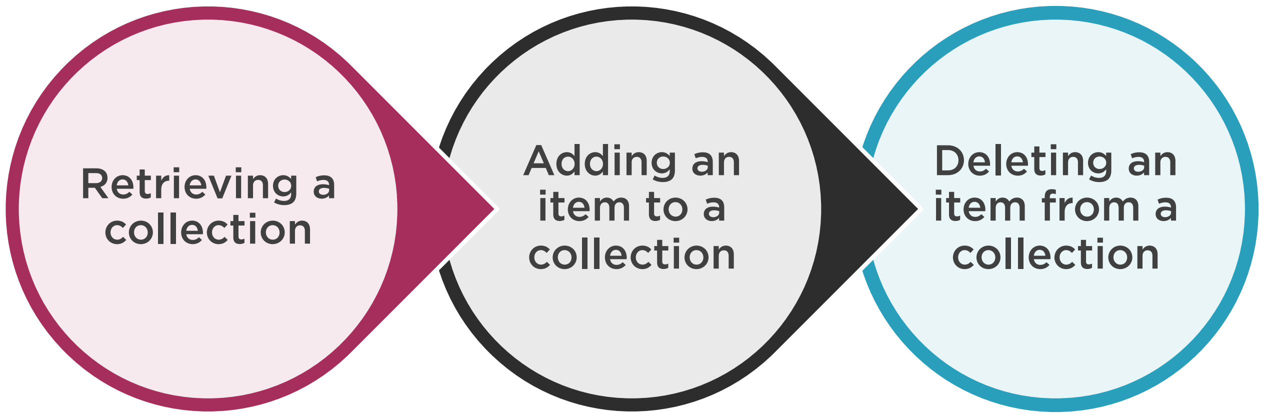
✓ **Introduced a repository**

⚠ **Still need the DbContext**

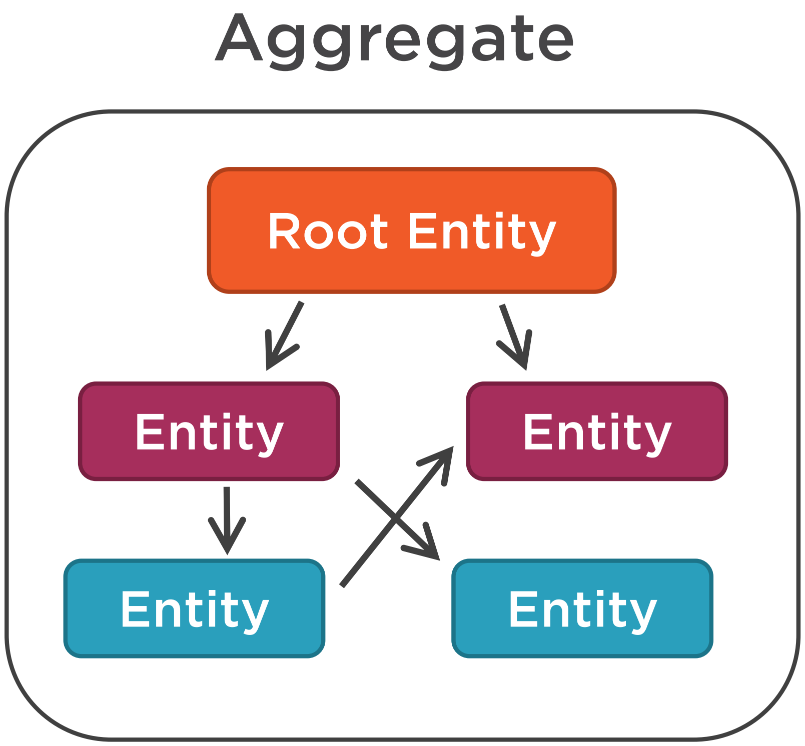
Recap: Introducing a Collection Invariant



Deleting an Item from the Collection


Retrieving a collection
Adding an item to a collection
Deleting an item from a collection

Recap: Deleting an Item from the Collection



✓ Student creates enrollments

✓ Student removes enrollments

```
x.HasMany(p => p.Enrollments).WithOne(p => p.Student)
    .OnDelete(DeleteBehavior.Cascade)
```

Shortcomings of Mapping to Backing Fields

Restrictions on mapping to backing fields

`typeof(_enrollments)` $\supseteq$ `typeof(Enrollments)`

(backing field) (property)

Shortcomings of Mapping to Backing Fields

```
public class Student : Entity
{
    private readonly ICollection<Enrollment> _enrollments;
    public virtual IReadOnlyList<Enrollment> Enrollments;
}
```

✗ **ICollection doesn’t inherit from IReadOnlyList**

```
public class Student : Entity
{
    private readonly ICollection<Enrollment> _enrollments;
    public virtual IEnumerable<Enrollment> Enrollments;
}
```

✓ **ICollection inherits from IEnumerable**

```
public class Student : Entity
{
    private readonly List<Enrollment> _enrollments;
    public virtual IReadOnlyList<Enrollment> Enrollments;
}
```

✓ **List inherits from IReadOnlyList**

Shortcomings of Mapping to Backing Fields

Can’t use a custom type for navigation properties

Shortcomings of Mapping to Backing Fields

New requirement

Student's favorite course is optional

```
public class Student : Entity
{
    public Course FavoriteCourse {}
}
```

→

```
public class Student : Entity
{
    public Maybe<Course> FavoriteCourse {}
}
```

```
public class Student : Entity
{
    private readonly Course _favoriteCourse;
    public Maybe<Course> FavoriteCourse => _favoriteCourse;
}
```

✗ **Doesn't work**

Shortcomings of Mapping to Backing Fields

Maybe<>

```
public class Student : Entity
{
    public Maybe<Course> FavoriteCourse {}
}
```

C# 8

```
public class Student : Entity
{
    public Course? FavoriteCourse {}
}
```

❌ **Doesn’t provide as strong guarantees**

Summary

One-to-many relationships and mapping to backing fields

Encapsulated the work with the collection

Introduced a new validation for adding items to the collection

EF Core only intercepts calls to navigation properties

Used a repository to encapsulate data retrieval logic

Deleted an item from the collection

Shortcomings of mapping to backing fields in EF Core

In the Next Module

Working with Disconnected Graphs of Objects