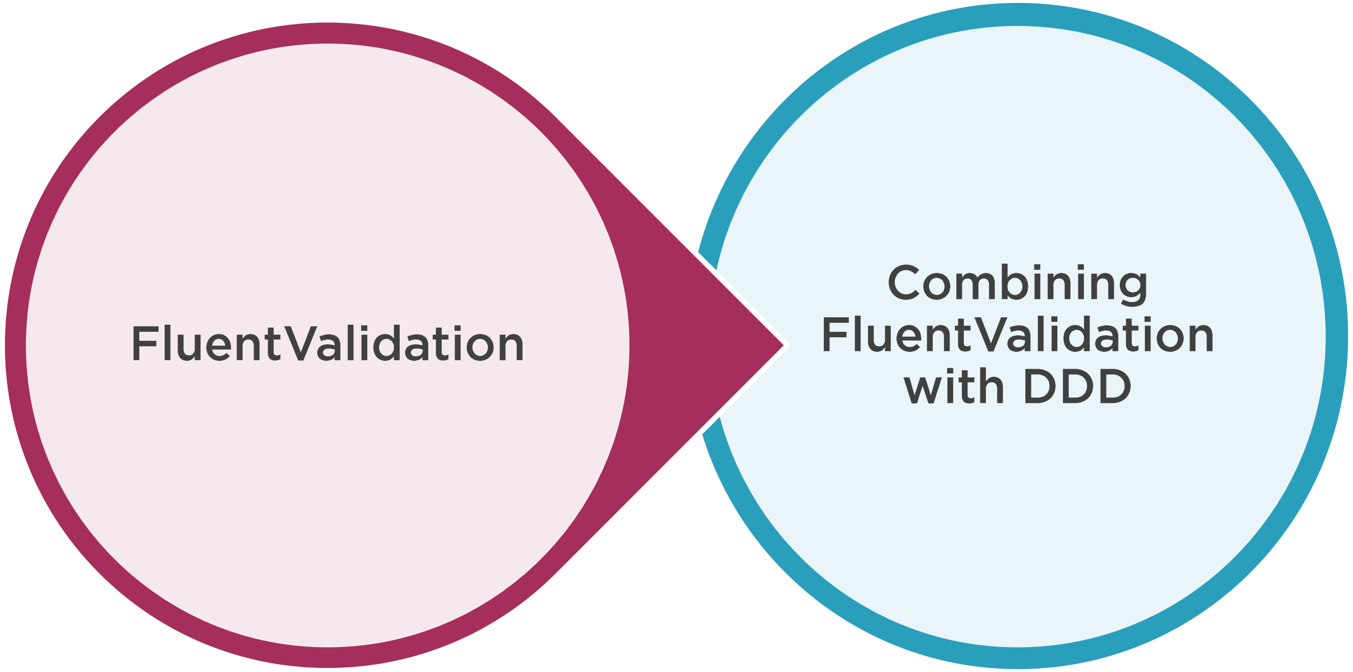
Diving Deeper into DDD and Validation

Vladimir Khorikov

@vkhorikov www.enterprisecraftsmanship.com

Validation



Introduction

- Defining explicit errors
- Standardizing the API output
- Performing complex validations
- DDD trilemma

Defining Explicit Errors

```
public static Result<State> Create(string input, string[] allStates)
{
    if (string.IsNullOrWhiteSpace(input))
        return Result.Failure<State>("Value is required");

    string name = input.Trim().ToUpper();

    if (name.Length > 2)
        return Result.Failure<State>("Value is too long");

    if (allStates.Any(x => x == name) == false)
        return Result.Failure<State>("State is invalid");

    return Result.Success(new State(name));
}
```

❌ Strings are not reliable errors

❌ Error messages should not be handled by the domain layer

Defining Explicit Errors

How to fix this?

✓ Define each error explicitly

Standardizing the API Output

```
{
    "type": "https://tools.ietf.org/html/rfc7231#section-6.5.1",
    "title": "One or more validation errors occurred.",
    "status": 400,
    "traceId": "00-853c3195f5d6bc4a8c852bfb12472dec-31e93af943966e4d-00",
    "errors": {
        "Addresses[0].State": [
            "value.is.required"
        ]
    }
}
```

```
{
    "id": 3
}
```

Successful responses

Recap: Explicit Errors and API Output

✓ **Introduced explicit errors**

- **Easier to debug**

Recap: Explicit Errors and API Output

```
public sealed class Error : ValueObject {
    public string Code { get; }
    public string Message { get; }

    internal Error(string code, string message) {
        Code = code;
        Message = message;
    }

    protected override IEnumerable<object> GetEqualityComponents() {
        yield return Code;
    }
}
```

Part of the contract with the clients

For debugging purposes only

Only Code participates in equality comparison

Recap: Explicit Errors and API Output

✓ **Introduced explicit errors**

- Easier to debug

✓ **All error codes must be unique**

- Check the uniqueness with a unit test

Recap: Explicit Errors and API Output

```
public sealed class Error : ValueObject
{
    public string Code { get; }
    public string Message { get; }
    public string HttpCode { get; } // e.g 404, 401, etc
}
```

✓ **Enables mapping to different HTTP response codes**

⚠ **Violation of domain model purity**

Recap: Explicit Errors and API Output

```
public static class Errors {
    public static class Student {
        public static Error EmailIsTaken() =>
            new Error("student.email.is.taken", "Student email is taken");
    }

    public static Error InternalServerError(string message) =>
        new Error("internal.server.error", message);
}
```

Domain error

Infrastructure error

⚠ Infrastructure errors shouldn’t reside in the domain layer

✓ Small concession

Recap: Explicit Errors and API Output

Success

✓ Result object

```
{
    "result": {
        "id": 3
    },
    "errorCode": null,
    "errorMessage": null,
    "invalidField": null,
    "timeGenerated": "2021-05-04"
}
```

Failure

✓ Error details

```
{
    "result": null,
    "errorCode": "student.email.is.taken",
    "errorMessage": "Student email is taken",
    "invalidField": null,
    "timeGenerated": "2021-05-04"
}
```

Recap: Explicit Errors and API Output

```
public class ModelStateValidator
{
    public static IActionResult ValidateModelState(ActionContext context)
    {
        (string fieldName, ModelStateEntry entry) = context.ModelState
            .First(x => x.Value.Errors.Count > 0);
        string errorSerialized = entry.Errors.First().ErrorMessage;

        Error error = Error.Deserialize(errorSerialized);
        Envelope envelope = Envelope.Error(error, fieldName);
        var envelopeResult = new EnvelopeResult(envelope, HttpStatusCode.BadRequest);

        return envelopeResult;
    }
}
```

Demo

How to check for email uniqueness

How to Check for Email Uniqueness

```
[HttpPost]
public IActionResult Register(RegisterRequest request)
{
    Student existingStudent = _studentRepository.GetByEmail(email);
    if (existingStudent != null)
        return Error(Errors.Student.EmailIsTaken());

    var student = new Student(email, name, addresses);
    _studentRepository.Save(student);

    return Ok();
}
```

Doesn’t belong to the domain layer

✗ **Not-always-valid domain model**

✗ **Not fully encapsulated**

How to Check for Email Uniqueness

```
// Student class
public Result<Student, Error> Create(Email email, string name,
    Address[] addresses, StudentRepository repository)
{
    Student existingStudent = repository.GetByEmail(email);
    if (existingStudent != null)
        return Errors.Student.EmailIsTaken();

    return new Student(email, name, addresses);
}
```

✓ **Always-valid domain model**

✗ **Impure domain model**

A pure domain model is a model that doesn’t reach out to out-of-process dependencies.

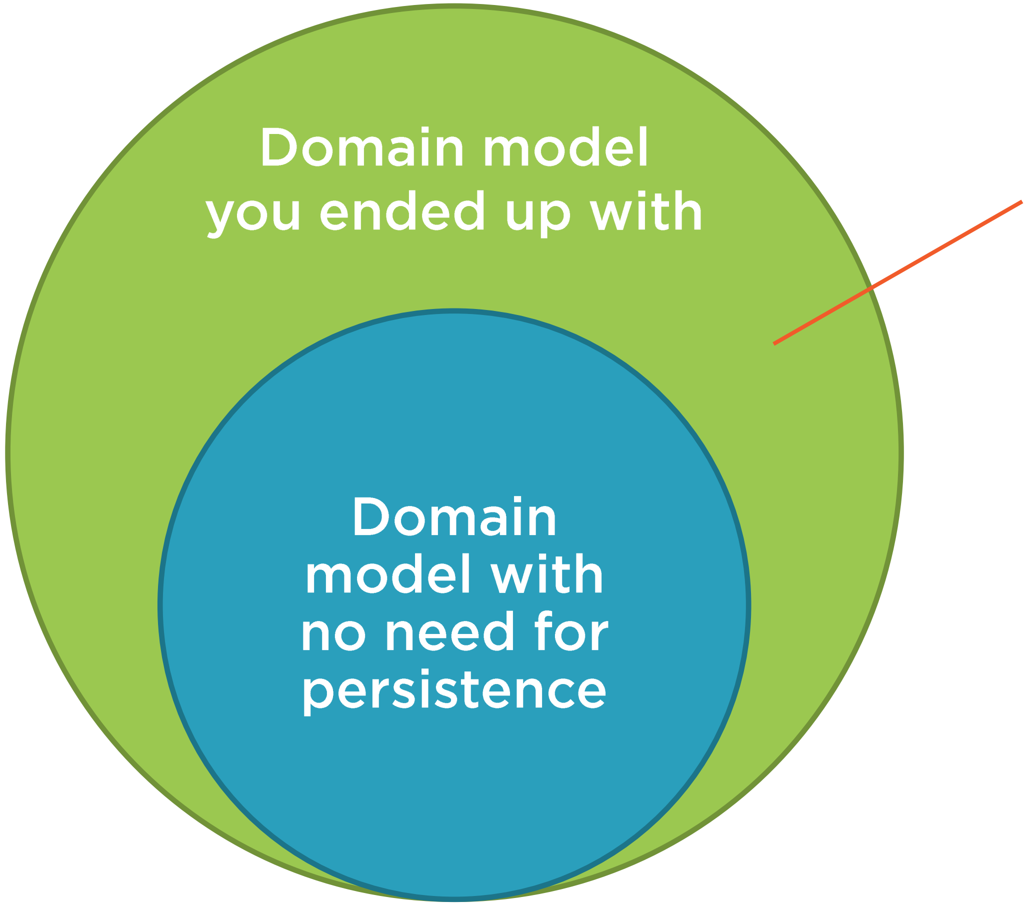
How to Check for Email Uniqueness

What if we replace the repository with an interface or a delegate?

How to Check for Email Uniqueness

`StudentRepository` **vs.** `IStudentRepository`

How to Check for Email Uniqueness



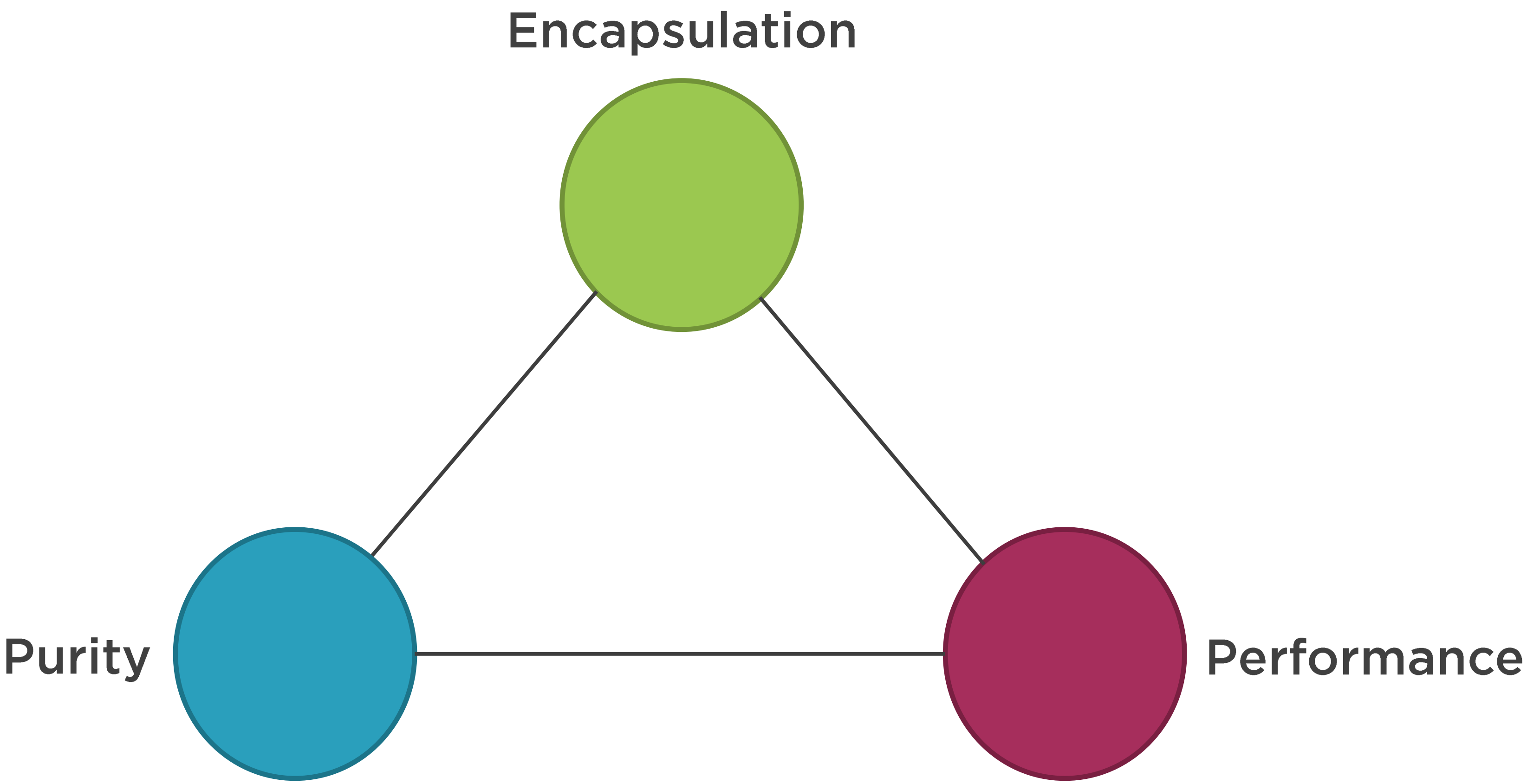
Domain model you ended up with

Domain model with no need for persistence

The difference: StudentRepository (or IStudentRepository) injected into Student

✗ No need for IStudentRepository in a persistence-ignorant domain model

DDD Trilemma



DDD Trilemma

Encapsulation : All domain logic is in the domain layer

Purity : No out-of-process dependencies

Performance : No unnecessary calls to out-of-process dependencies

DDD Trilemma

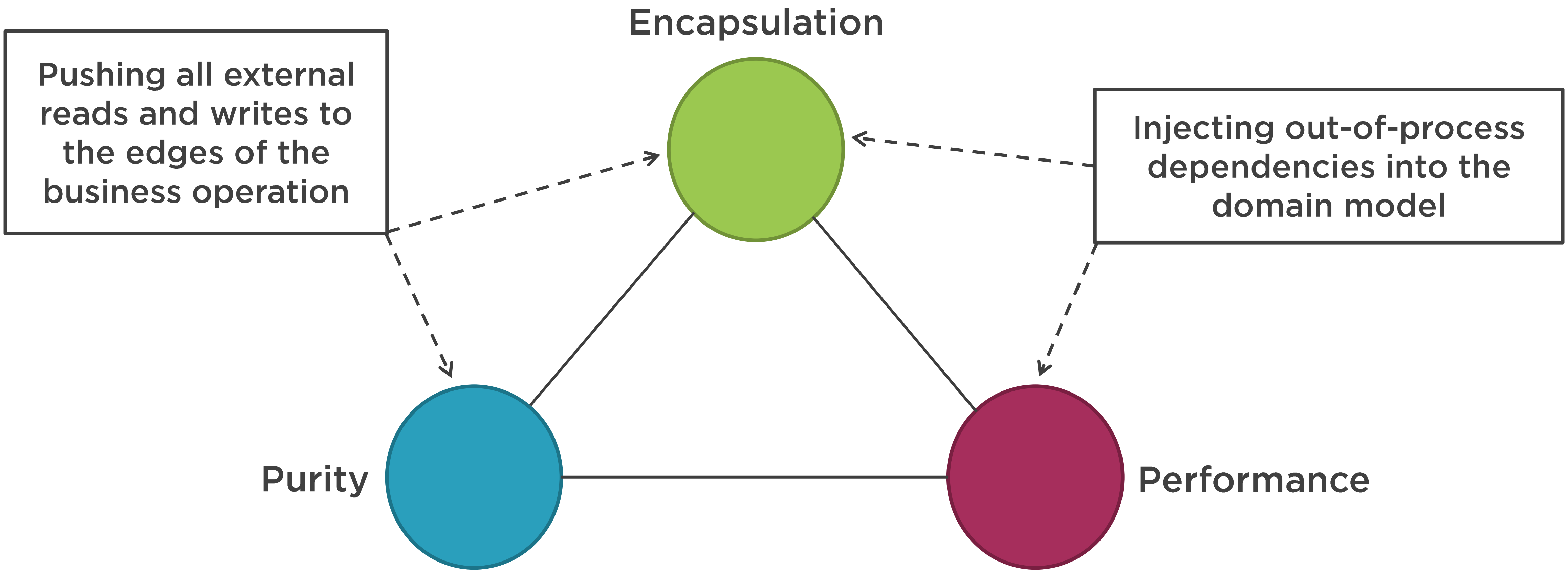
```
// Student class
public Result<Student, Error> Create(Email email, string name,
    Address[] addresses, Student[] allStudents)
{
    if (allStudents.Any(x => x.Email == email))
        return Errors.Student.EmailIsTaken();

    return new Student(email, name, addresses);
}
```

✓ **Pure domain model**

✓ **Encapsulated domain model**

✗ **Poor performance**

DDD Trilemma

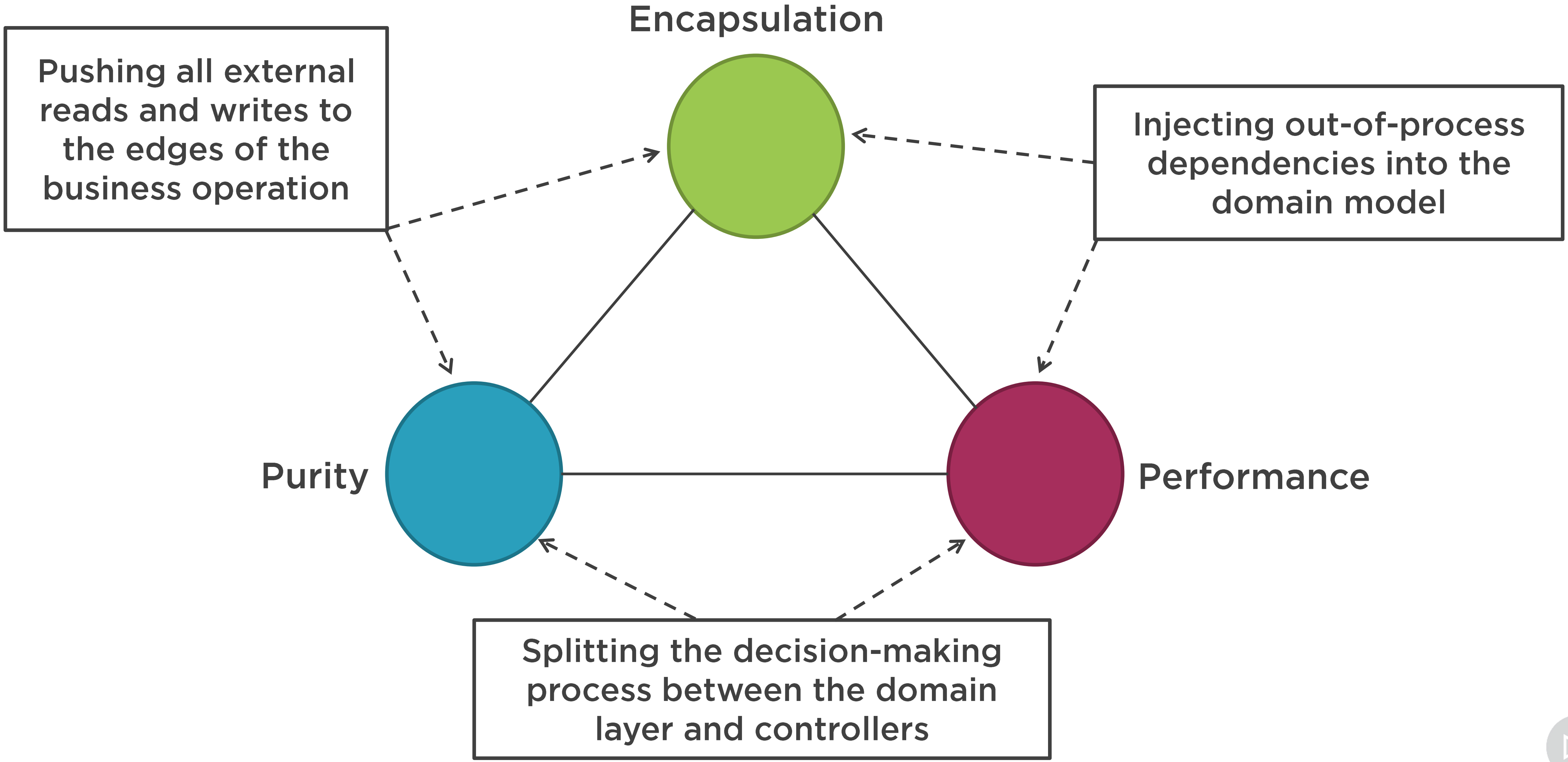


Encapsulation
Pushing all external reads and writes to the edges of the business operation
Injecting out-of-process dependencies into the domain model
Purity
Performance

DDD Trilemma

```
// Student class
public Result<Student, Error> Create(Email email, string name,
    Address[] addresses, StudentRepository repository)
{
    Student existingStudent = repository.GetByEmail(email);
    if (existingStudent != null)
        return Errors.Student.EmailIsTaken();

    return new Student(email, name, addresses);
}
```

DDD Trilemma



Pushing all external reads and writes to the edges of the business operation

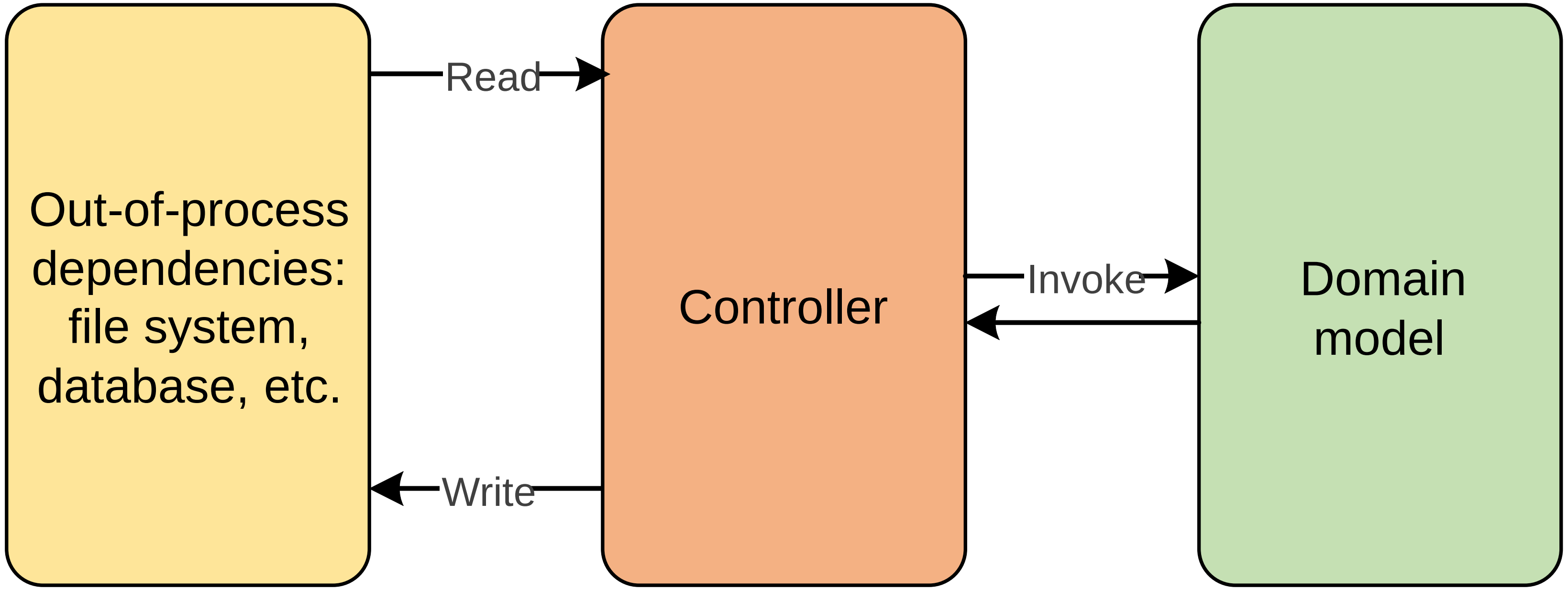
Injecting out-of-process dependencies into the domain model

Splitting the decision-making process between the domain layer and controllers

DDD Trilemma

✓ **Give up performance in favor of encapsulation and purity when possible**

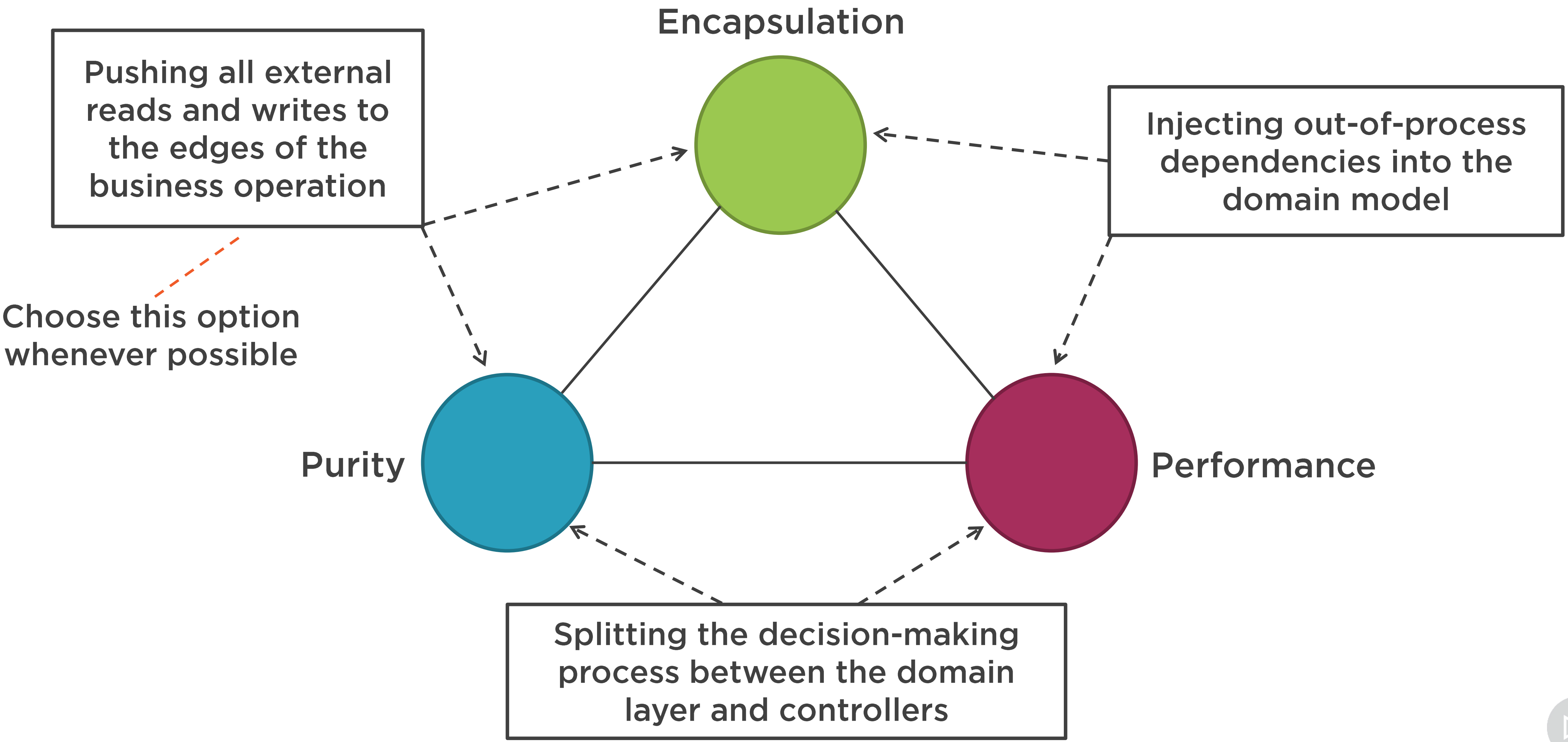
DDD Trilemma



❌ Too many data to query

❌ Need to query additional data in the middle of a business operation

DDD Trilemma


Encapsulation
Purity
Performance
Pushing all external reads and writes to the edges of the business operation
Choose this option whenever possible
Injecting out-of-process dependencies into the domain model
Splitting the decision-making process between the domain layer and controllers

DDD Trilemma

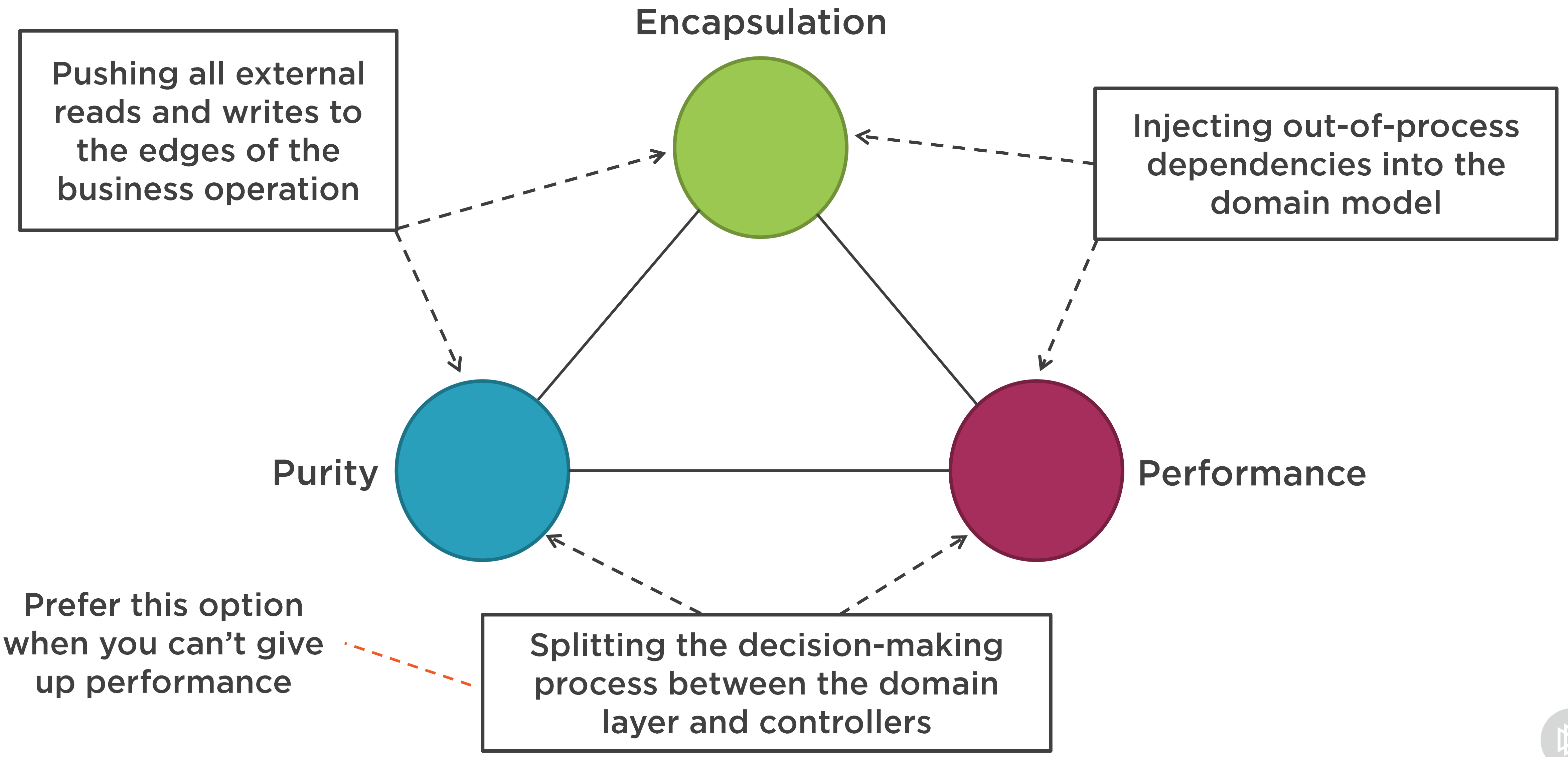
Purity

vs

Encapsulation

✓ Choose purity over encapsulation

DDD Trilemma



Pushing all external reads and writes to the edges of the business operation

Injecting out-of-process dependencies into the domain model

Splitting the decision-making process between the domain layer and controllers

Prefer this option when you can't give up performance

DDD Trilemma

⚠ **Business logic is the most important part of the application**

✗ Don't mix it with other responsibilities

✓ Domain layer should only be responsible for the domain logic

DDD Trilemma

Always-valid domain model

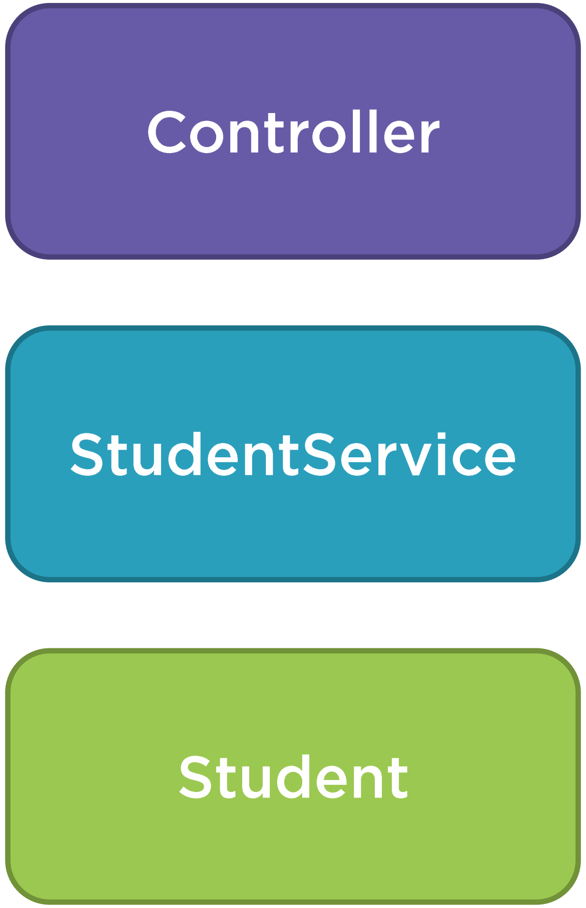
Validation is domain logic

Validation is parsing

⚠ Concession 1: Putting simple validations outside the domain layer

⚠ Concession 2: Preferring domain model purity over encapsulation

DDD Trilemma



Controller

StudentService

Student

: Communicating with the database

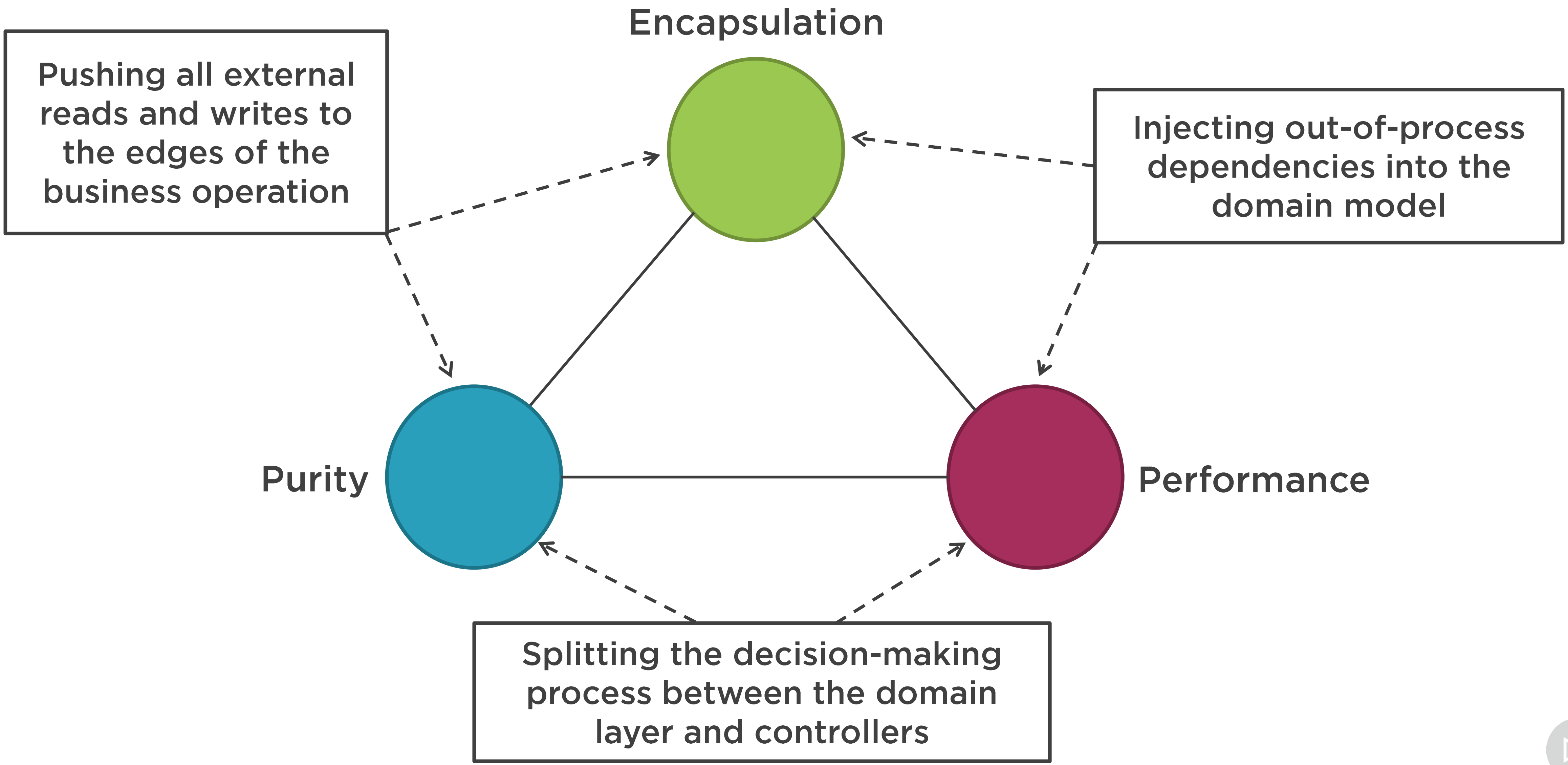
✗ Doesn’t change the trade-off

Recap: Finishing up the Rest of Validations

```
[HttpPost("{id}/enrollments")]
public IActionResult Enroll(long id, EnrollRequest request)
{
    for (int i = 0; i < request.Enrollments.Length; i++)
    {
        CourseEnrollmentDto dto = request.Enrollments[i];

        Grade grade = Grade.Create(dto.Grade).Value;

        string courseName = (dto.Course ?? "").Trim();
        Course course = _courseRepository.GetByName(courseName);
        if (course == null)
            return Error(Errors.General.ValueIsInvalid(),
                $"{nameof(request.Enrollments)}[{i}].{nameof(dto.Course)}");

        Result<object, Error> result = student.Enroll(course, grade);
        if (result.IsFailure)
            return Error(result.Error);
    }

    return Ok();
}
```

Which option we choose with this approach?

Recap: Finishing up the Rest of Validations



Pushing all external reads and writes to the edges of the business operation

Injecting out-of-process dependencies into the domain model

Splitting the decision-making process between the domain layer and controllers

Recap: Finishing up the Rest of Validations

```
[HttpPost("{id}/enrollments")]
public IActionResult Enroll(long id, EnrollRequest request)
{
    for (int i = 0; i < request.Enrollments.Length; i++)
    {
        CourseEnrollmentDto dto = request.Enrollments[i];

        Grade grade = Grade.Create(dto.Grade).Value;

        string courseName = (dto.Course ?? "").Trim();
        Course course = _courseRepository.GetByName(courseName);
        if (course == null)
            return Error(Errors.General.ValueIsInvalid(),
                $"{nameof(request.Enrollments)}[{i}].{nameof(dto.Course)}");

        Result<object, Error> result = student.Enroll(course, grade);
        if (result.IsFailure)
            return Error(result.Error);
    }

    return Ok();
}
```

Which option we choose with this approach?

How to choose encapsulation over purity?

How to choose encapsulation and purity over performance?

Recap: Finishing up the Rest of Validations

```csharp
[HttpPost("{id}/enrollments")]
public IActionResult Enroll(long id, EnrollRequest request) {
    (string Course, string Grade)[] input = request.Enrollments
        .Select(x => (x.Course, x.Grade))
        .ToArray();
    Course[] allCourses = _courseRepository.GetAll();

    Result<Enrollment[], Error> enrollmentsOrError = Enrollment.Create(input, allCourses);
    if (enrollmentsOrError.IsFailure)
        return Error(enrollmentsOrError.Error);

    Result<object, Error> result = student.Enroll(enrollmentsOrError.Value);
    if (result.IsFailure)
        return Error(result.Error);

    return Ok();
}
```

✓ **The logic becomes unit-testable**

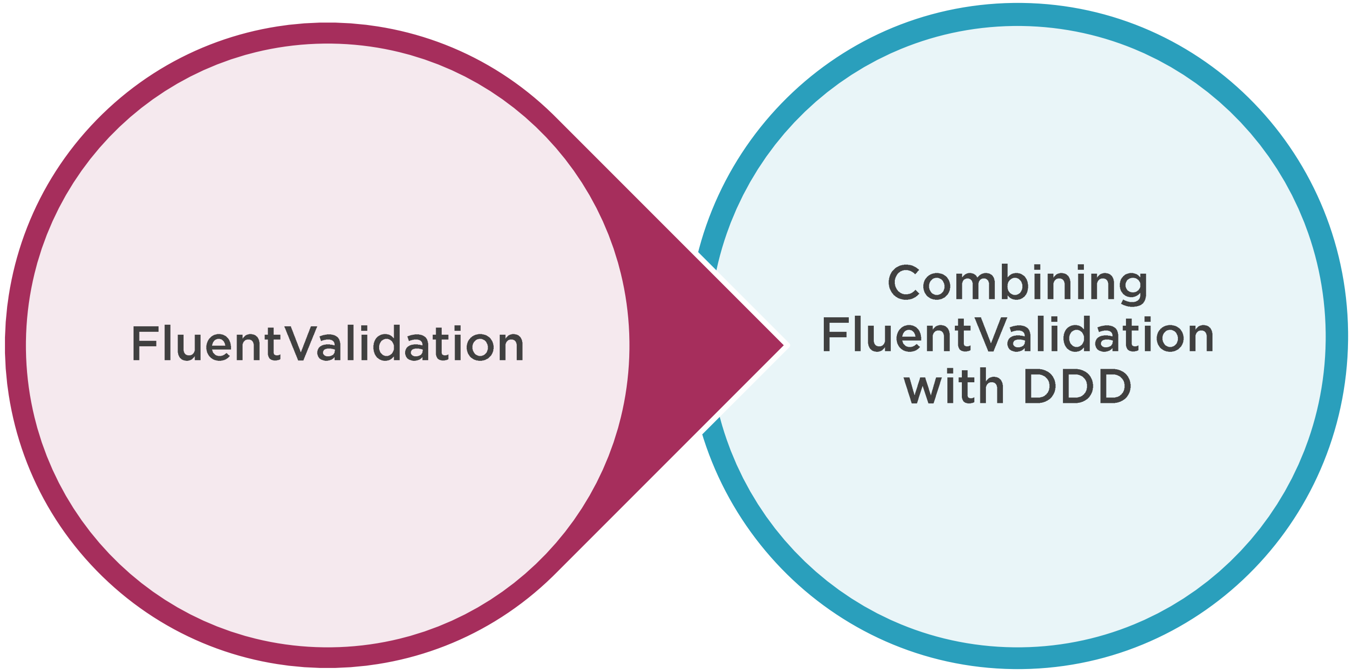
http://bit.ly/code-validation

Course Summary

https://enterprisecraftsmanship.com

http://bit.ly/vlad-updates

Course Summary



FluentValidation

Combining FluentValidation with DDD

Course Summary

Three building blocks of the good validation technique

- Always-valid domain model
- All validation rules are part of the domain layer
- Validation is parsing

Don’t put the simplest validation rules to the domain layer

DDD trilemma

Contacts

http://bit.ly/vlad-updates

@vkhorikov

https://enterprisecraftsmanship.com