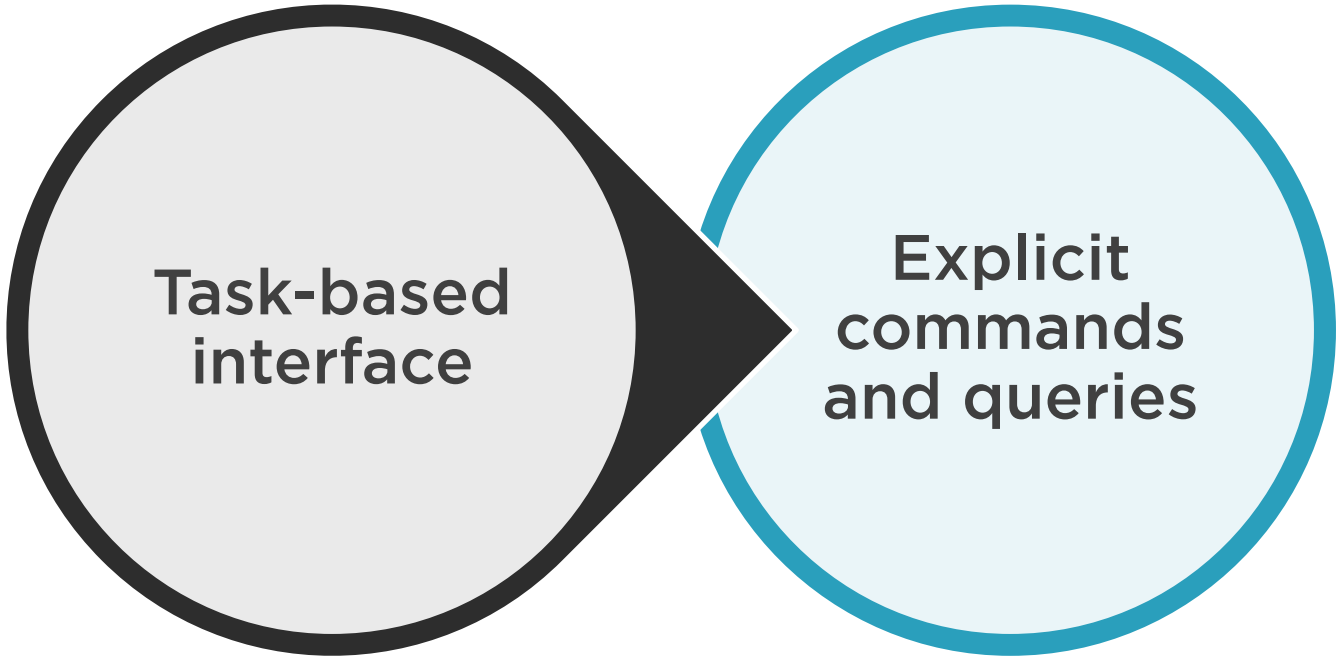
Segregating Commands and Queries

Vladimir Khorikov

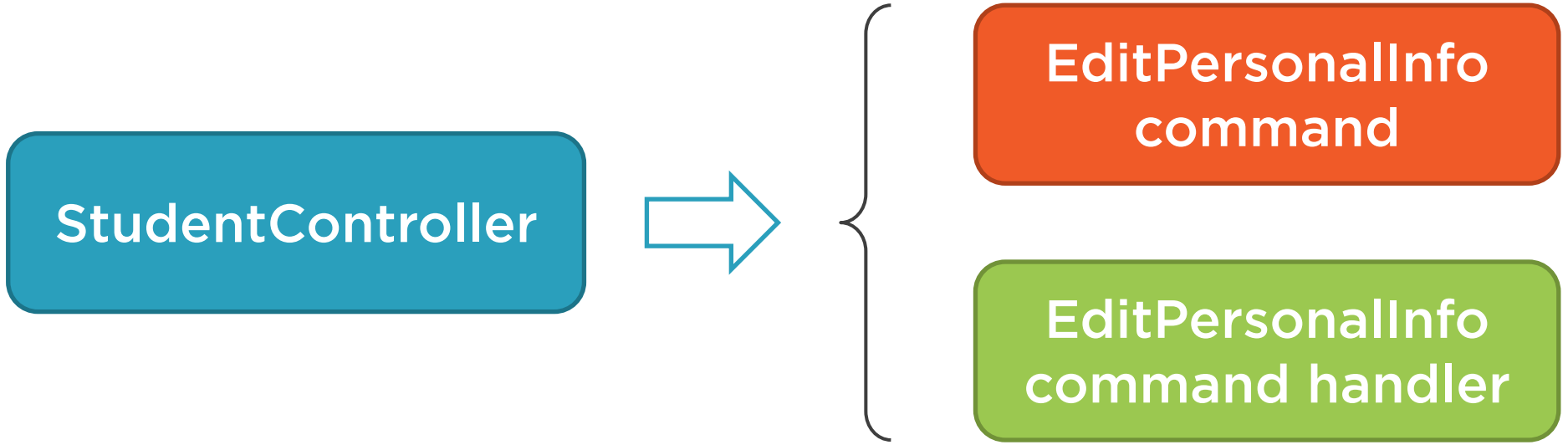
@vkhorikov www.enterprisecraftsmanship.com

Agenda



- Scalability
- Performance
- Simplicity

Commands in CQS vs. Commands in CQRS



Moved all logic from controller to handler

Commands in CQS vs. Commands in CQRS

```
[HttpPut("{id}")]
public IActionResult EditPersonalInfo(long id, [FromBody] StudentPersonalInfoDto dto)
{
    var command = new EditPersonalInfoCommand
    {
        Email = dto.Email,
        Name = dto.Name,
        Id = id
    };
    var handler = new EditPersonalInfoCommandHandler(_unitOfWork);
    Result result = handler.Handle(command);

    return result.IsSuccess ? Ok() : Error(result.Error);
}
```

Commands in CQS vs. Commands in CQRS

Command

Controller method

```
[HttpPut("{id}")]
public IActionResult EditPersonalInfo(
    long id,
    [FromBody] StudentPersonalInfoDto dto)
{
    /* ... */
}
```

CQS command

Class

```
public sealed class EditPersonalInfoCommand
    : ICommand
{
    public long Id { get; set; }
    public string Name { get; set; }
    public string Email { get; set; }
}
```

CQRS command

Commands in CQS vs. Commands in CQRS

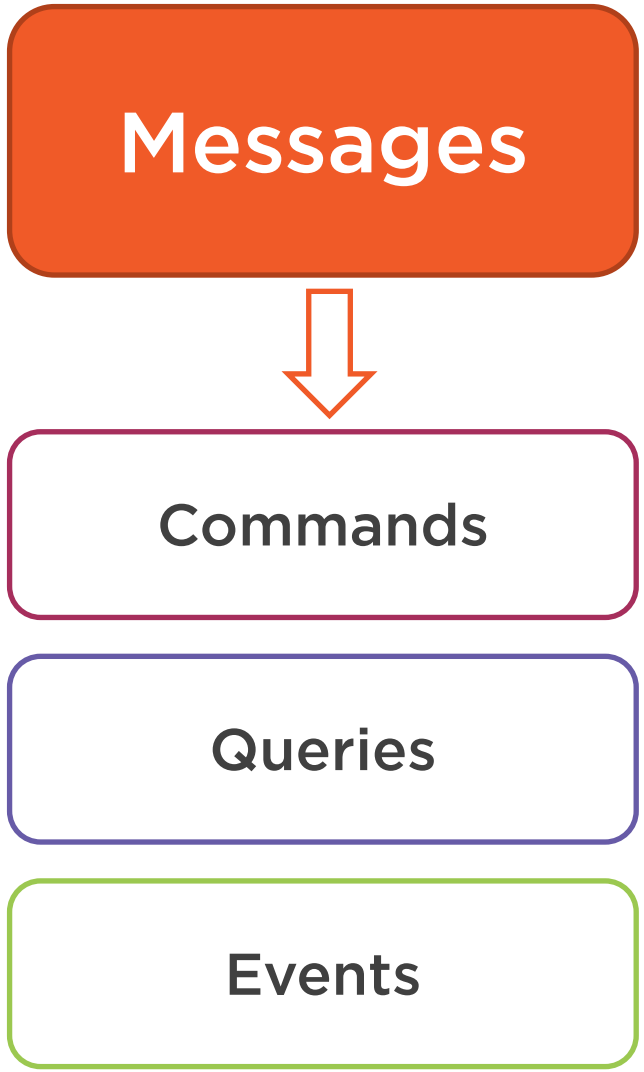
Command is a serializable method call

Is there an analogy for command handler?

Commands in CQS vs. Commands in CQRS

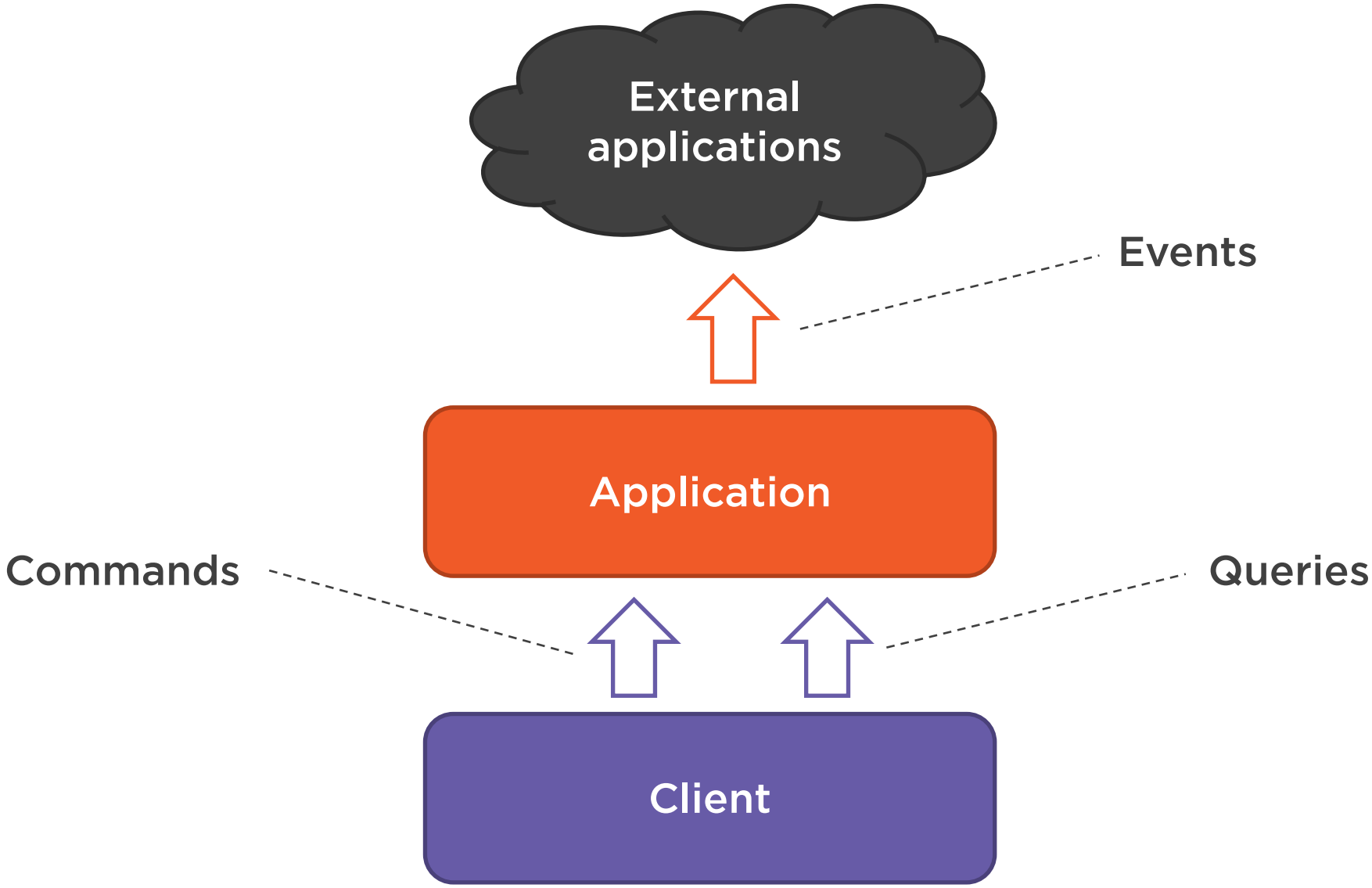
```
public sealed class EditPersonalInfoCommandHandler : ICommandHandler<EditPersonalInfoCommand>
{
    public Result Handle(EditPersonalInfoCommand command)
    {
        var repository = new StudentRepository(_unitOfWork);
        Student student = repository.GetById(command.Id);

        if (student == null)
            return Result.Fail($"No student found for Id {command.Id}");

        student.Name = command.Name;
        student.Email = command.Email;

        _unitOfWork.Commit();

        return Result.Ok();
    }
}
```

Commands and Queries in CQRS



- Tell the application to do something
- Ask the application about something
- Inform external applications

Commands and Queries in CQRS



Domain-Driven Design in Practice

by Vladimir Khorikov

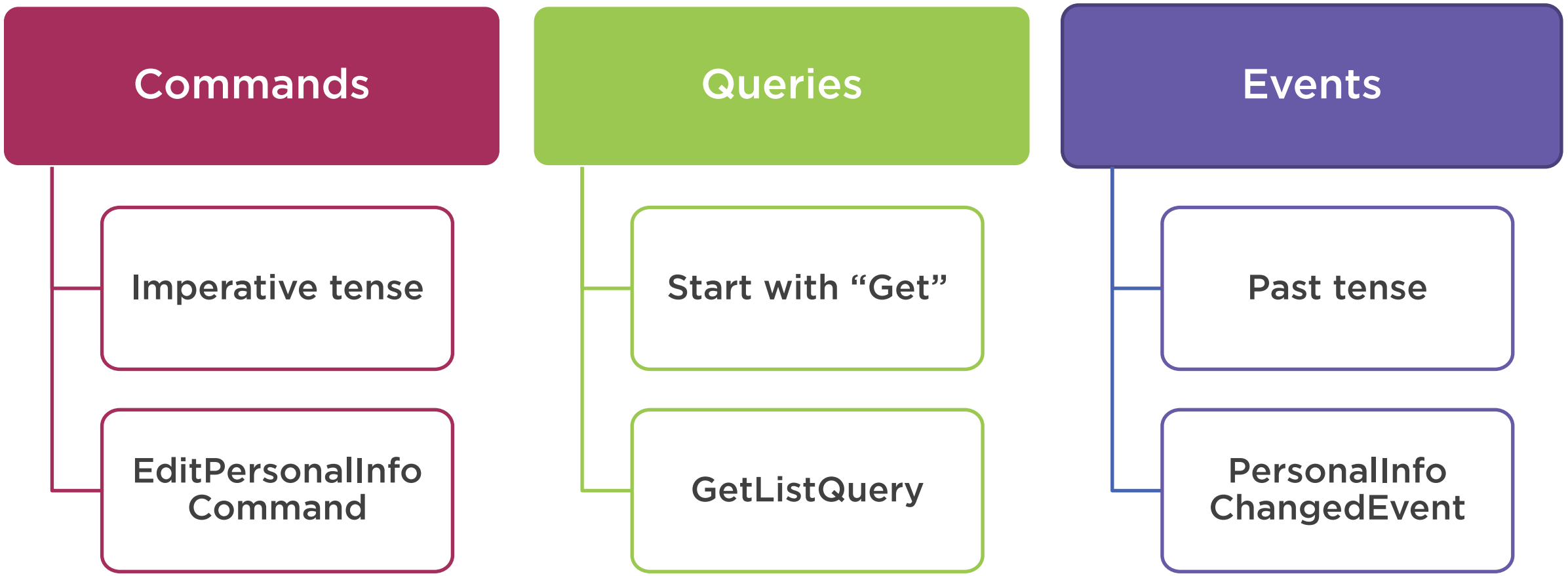
A descriptive, in-depth walk-through for applying Domain-Driven Design principles in practice.

▶ Resume Course

Table of contents | Description | Transcript | Exercise files | Discussion | Learning Check | Recommended

Expand all

Naming Guidelines



Naming Guidelines

Command vs. **Event**

Edit personal info

Personal info changed

Server can reject a command

Server can't reject an event

Naming Guidelines

Commands should use the ubiquitous language

CreateStudentCommand

UpdateStudentCommand

DeleteStudentCommand

CRUD-based thinking

Naming Guidelines

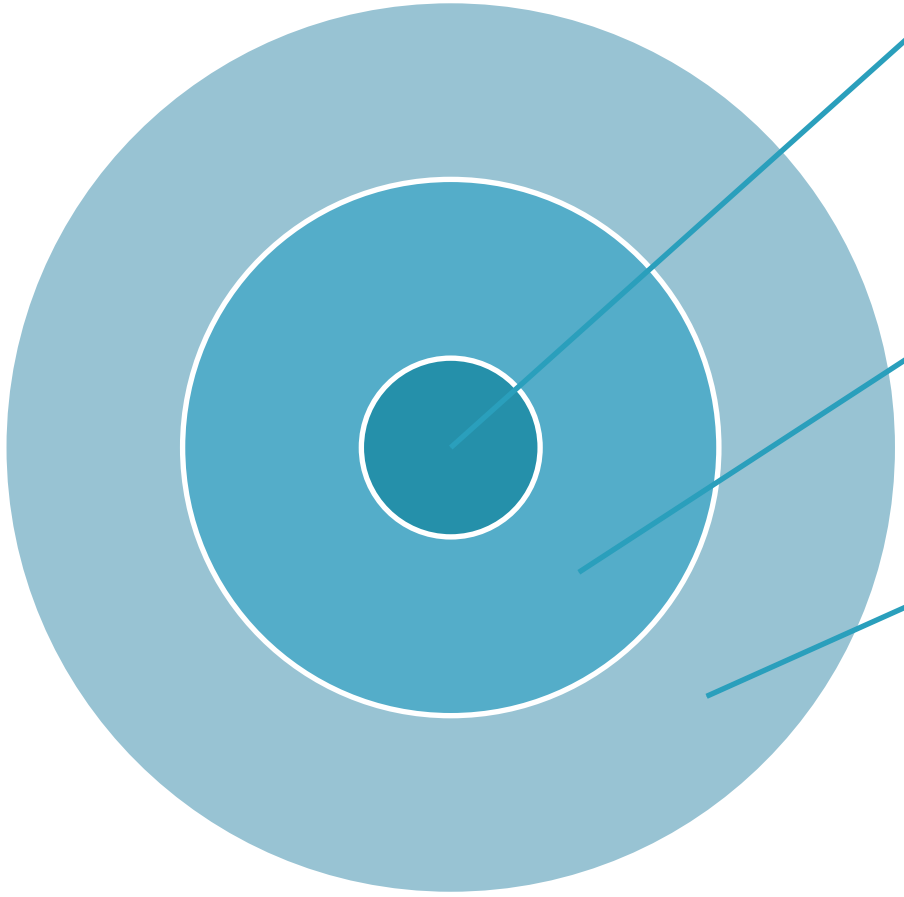
EditPersonalInfo~~Command~~

GetList~~Query~~

PersonalInfoChanged~~Event~~

Naming convention is enough
to distinguish between them

Commands and Queries in the Onion Architecture



Entities
Aggregates
Value Objects
Domain Events
Pure Domain Services

Repositories
Impure Domain Services

Application Services
UI

Core domain

Non-core domain

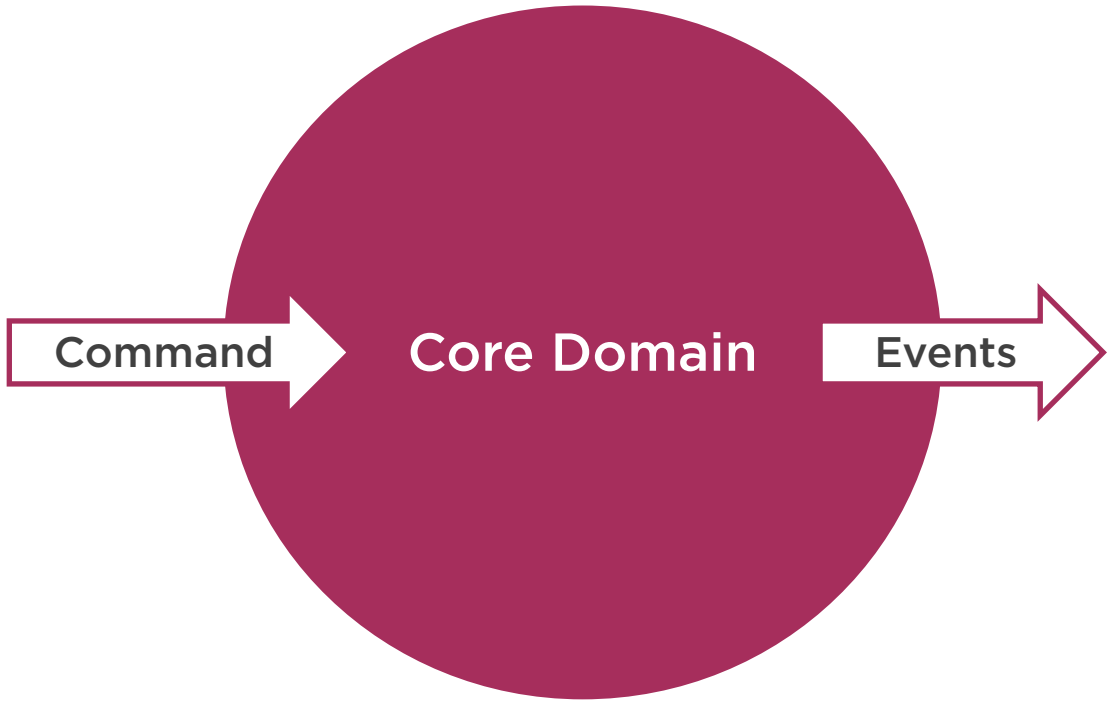
Non-domain

Commands and Queries in the Onion Architecture

All messages are part of the core domain

Command	=	An operation to do
Query	=	A question to ask
Event	=	An outcome for external apps

Commands and Queries in the Onion Architecture



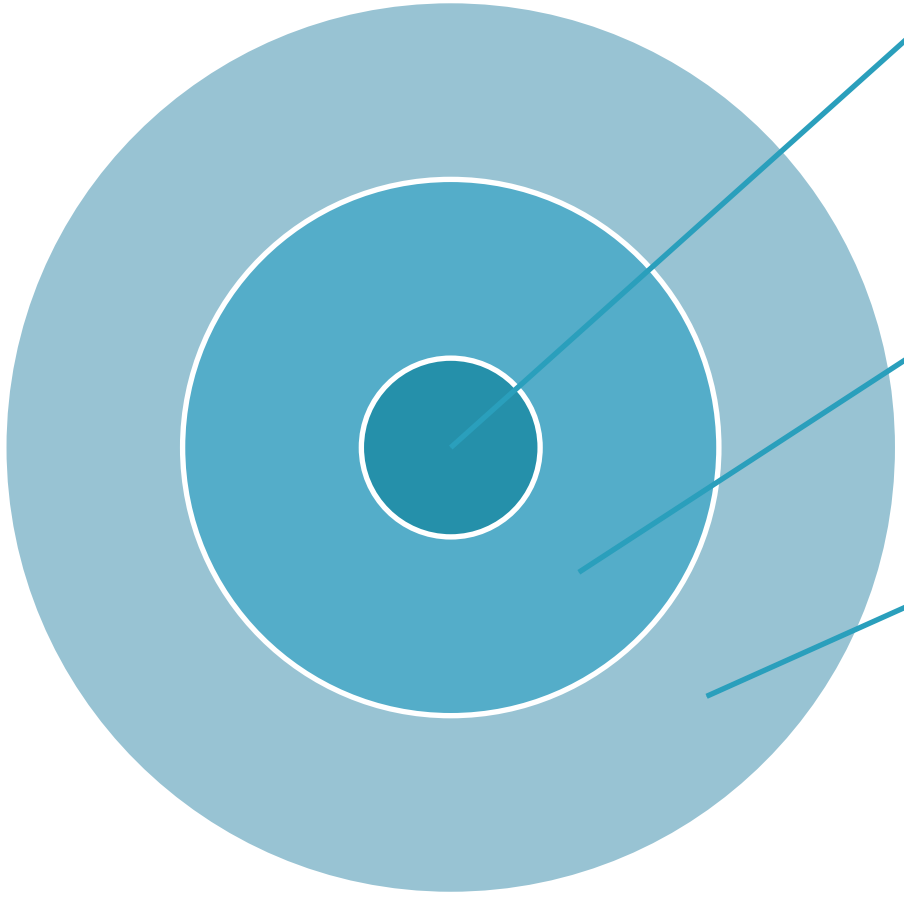
Commands

- Trigger reaction
- Push model

Events

- Result of the reaction
- Pull model

Commands and Queries in the Onion Architecture



Entities
Aggregates
Value Objects
Domain Events
Pure Domain Services

Repositories
Impure Domain Services

Application Services
UI

Should be isolated from the outside world

Command handlers go here

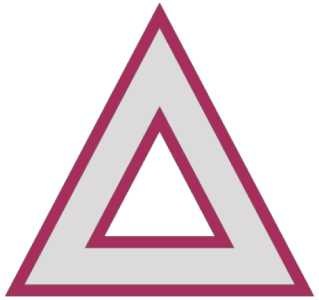
Commands vs. DTOs

```
[HttpPut("{id}")]
public IActionResult EditPersonalInfo(long id, [FromBody] StudentPersonalInfoDto dto) {
    var command = new EditPersonalInfoCommand {
        Email = dto.Email, Name = dto.Name, Id = id
    };
    var handler = new EditPersonalInfoCommandHandler(_unitOfWork);
    Result result = handler.Handle(command);

    return result.IsSuccess ? Ok() : Error(result.Error);
}
```

~~[HttpPut("{id}")]~~
~~public IActionResult EditPersonalInfo([FromBody] EditPersonalInfoCommand command) {~~
~~   var handler = new EditPersonalInfoCommandHandler(_unitOfWork);~~
~~   Result result = handler.Handle(command);~~

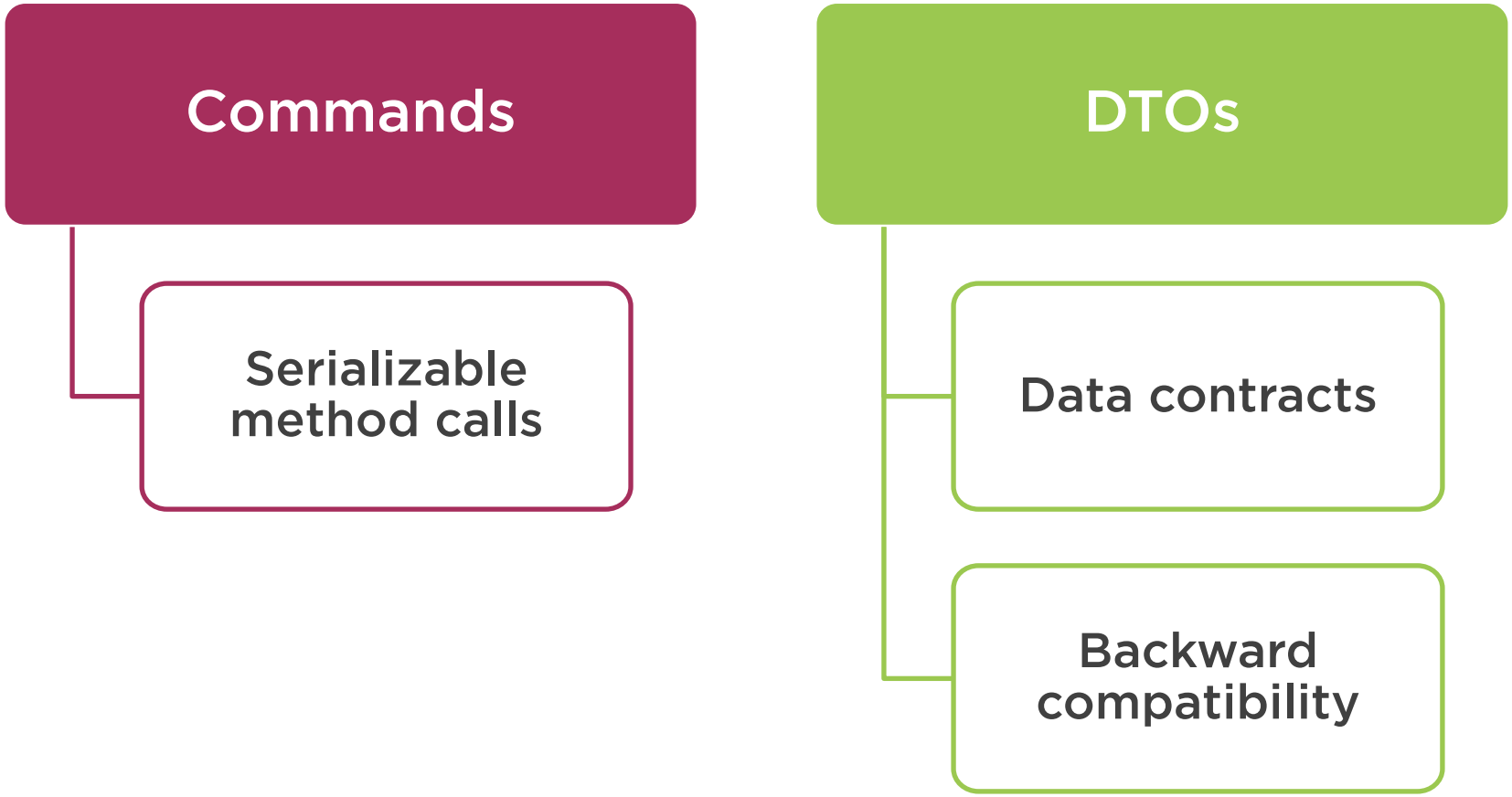
~~   return result.IsSuccess ? Ok() : Error(result.Error);~~
~~}~~

Commands vs. DTOs

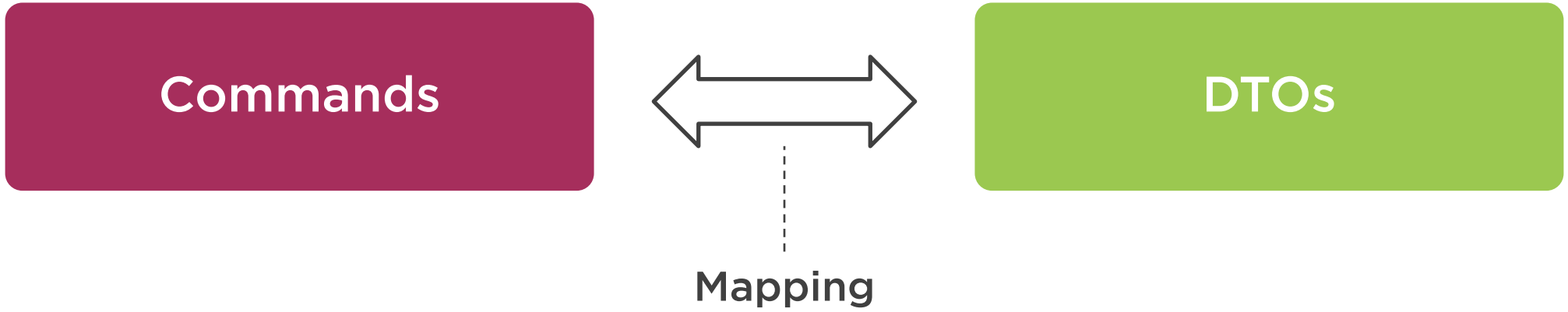


Commands and DTOs tackle different problems

Commands vs. DTOs



Commands vs. DTOs



- Backward compatible
- Easy to refactor

Commands vs. DTOs

The use of commands as DTOs = **The use of entities as DTOs**

Hinder refactoring

Commands vs. DTOs

```
public sealed class EditPersonalInfoCommand : ICommand
{
    public long Id { get; }
    public string Name { get; }
    public string Email { get; }

    public EditPersonalInfoCommand(long id, string name, string email)
    {
        Id = id;
        Name = name;
        Email = email;
    }
}
```

FirstName

LastName

Can’t modify the command

Commands vs. DTOs

DTOs = Backward compatibility

Commands = Actions upon the application

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.

Resume Course | Bookmark | Add to Channel | Live mentoring

Table of contents | Description | Transcript | Exercise files | Discussion | Learning Check | Recommended

Expand all

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

Course author

Vladimir Khorikov

Vladimir Khorikov is a Microsoft MVP and has been professionally involved in software development for more than 10 years.

Course info

Level	Intermediate
Rating	★★★★★ (73)
My rating	★★★★★
Duration	3h 36m
Released	13 Nov 2017

Share course

Commands vs. DTOs

It's fine not to have DTOs if you don't need backward compatibility

- A single client which you develop yourself
- Can deploy both the API and the client simultaneously

Recap: Introducing Commands and Queries

Refactored the student controller

- Uses explicit command and query objects
- In theory, you could even remove the controller

Recap: Introducing Commands and Queries

```
[HttpPut("{id}")]
public IActionResult EditPersonalInfo(
    long id, [FromBody] StudentPersonalInfoDto dto)
{
    var command = new EditPersonalInfoCommand(id, dto.Email, dto.Name);
    Result result = _messages.Dispatch(command);

    return FromResult(result);
}
```

Use the controller for ASP.NET wiring only

Recap: Introducing Commands and Queries

```
public sealed class Messages
{
    private readonly IServiceProvider _provider;

    public Messages(IServiceProvider provider)
    {
        _provider = provider;
    }

    public Result Dispatch(ICommand command)
    {
        Type type = typeof(ICommandHandler<>);
        Type[] typeArgs = { command.GetType() };
        Type handlerType = type.MakeGenericType(typeArgs);

        dynamic handler = _provider.GetService(handlerType);
        Result result = handler.Handle((dynamic)command);

        return result;
    }
}
```

Recap: Introducing Commands and Queries

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();

    services.AddSingleton(new SessionFactory(Configuration["ConnectionString"]));
    services.AddTransient<UnitOfWork>();
    services.AddTransient<ICommandHandler<EditPersonalInfoCommand>, EditPersonalInfoCommandHandler>();
    services.AddTransient<ICommandHandler<RegisterCommand>, RegisterCommandHandler>();
    services.AddTransient<ICommandHandler<UnregisterCommand>, UnregisterCommandHandler>();
    services.AddTransient<ICommandHandler<EnrollCommand>, EnrollCommandHandler>();
    services.AddTransient<ICommandHandler<TransferCommand>, TransferCommandHandler>();
    services.AddTransient<ICommandHandler<DisenrollCommand>, DisenrollCommandHandler>();
    services.AddTransient<IQueryHandler<GetListQuery, List<StudentDto>>, GetListQueryHandler>();
    services.AddSingleton<Messages>();
}
```

Recap: Introducing Commands and Queries

```
public interface ICommand
{
}

public interface ICommandHandler<TCommand>
    where TCommand : ICommand
{
    Result Handle(TCommand command);
}
```

Success of failure

```
public interface IQuery<TResult>
{
}

public interface IQueryHandler<TQuery, TResult>
    where TQuery : IQuery<TResult>
{
    TResult Handle(TQuery query);
}
```

```
public sealed class GetListQuery :
    IQuery<List<StudentDto>>
{
    /* ... */
}
```

Recap: Introducing Commands and Queries

```
public interface ICommandHandler<TCommand>
    where TCommand : ICommand
{
    Result Handle(TCommand command);
}
```

```
public interface IQueryHandler<TQuery, TResult>
    where TQuery : IQuery<TResult>
{
    Result<TResult> Handle(TQuery query);
}
```

Summary

Refactored towards explicit commands and queries

Introduced unified interface for command and query handlers

Leveraged ASP.NET dependency injection mechanism for resolving the handlers

Difference between commands and queries in CQS and CQRS taxonomies

- CQS: command is a method that mutates state
- CQRS: command represents what you can do with the application

Command: serializable method call

Command handler: an ASP.NET controller with a single method that is a CQS command

Summary

Messages: commands, queries, and events

- Command tells the application to do something
- Query asks the application about something
- Event is an informational message

Name all 3 types of messages properly:

- Commands should be in the imperative tense
- Events should be in the past tense
- Queries: same as commands, start with "Get“

Summary

Commands and queries in the onion architecture

- Commands, queries, and events should reside in the core domain layer
- Commands: push model
- Events: pull model

Commands vs DTOs

- DTOs help achieve backward compatibility
- Commands explicitly state what the application can do

In the Next Module

Implementing decorators upon command and query handlers