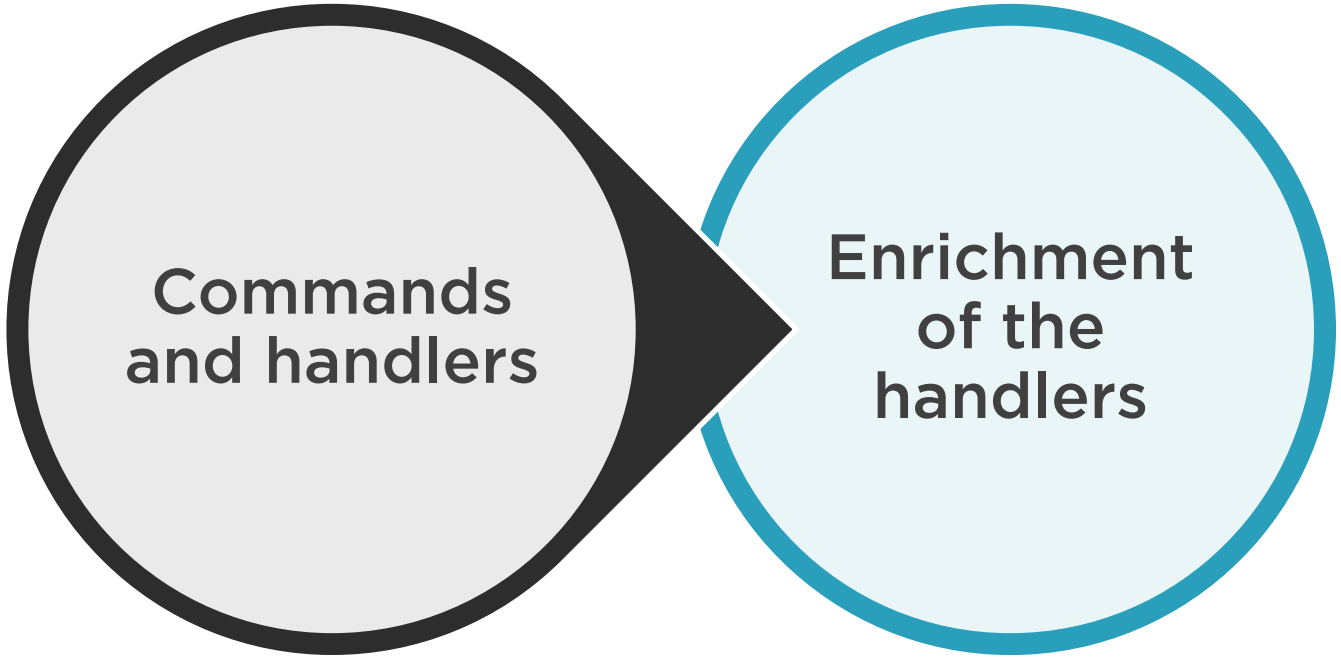
Implementing Decorators upon Command and Query Handlers

Vladimir Khorikov

@vkhorikov www.enterprisecraftsmanship.com

Agenda



New Requirement

Decorator Pattern

Implemented the first decorator

Re-runs the same command handler for 3 times

Decorator is a class or a method that modifies the behavior of an existing class or method without changing its public interface.

Decorator Pattern

```
public sealed class DatabaseRetryDecorator<TCommand> : ICommandHandler<TCommand>
    where TCommand : ICommand
{
    private readonly ICommandHandler<TCommand> _handler;

    public Result Handle(TCommand command) {
        for (int i = 0; ; i++) {
            try {
                Result result = _handler.Handle(command);
                return result;
            }
            catch (Exception ex) {
                if (i >= _config.NumberOfDatabaseRetries || !IsDatabaseException(ex))
                    throw;
            }
        }
    }
}
```

Didn’t require to modify the existing command handlers

Decorator Pattern

```
public sealed class Messages
{
    private readonly IServiceProvider _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;
    }
}
```

DatabaseRetry Decorator

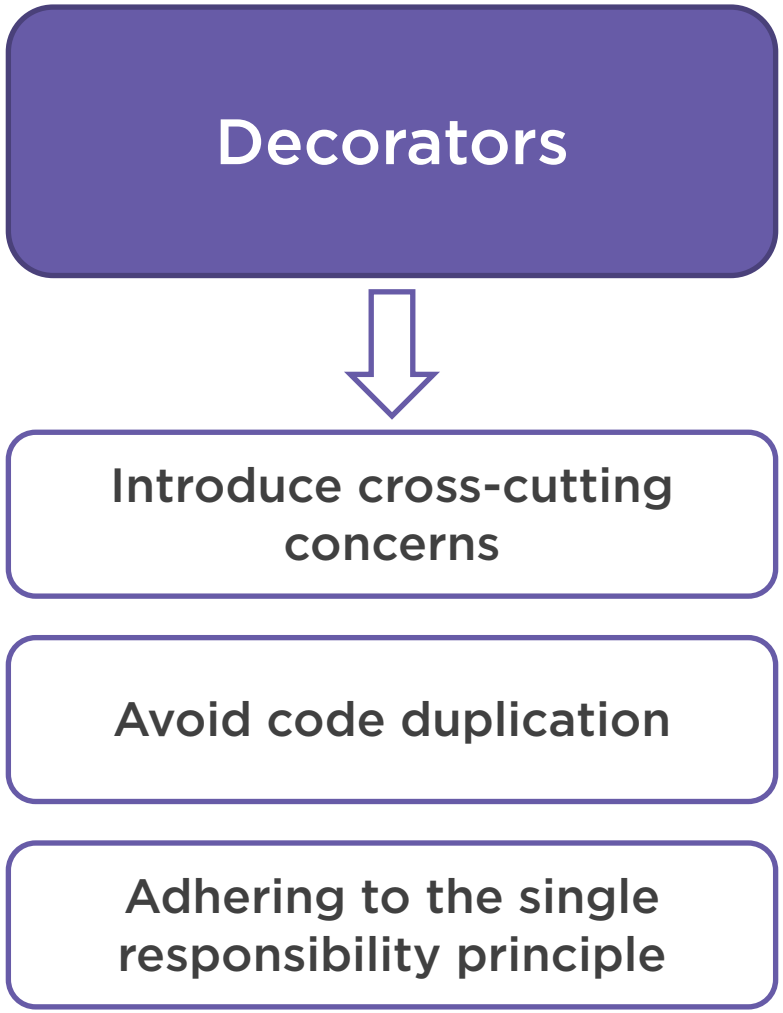
~~EditPersonalInfo CommandHandler~~

Decorator Pattern

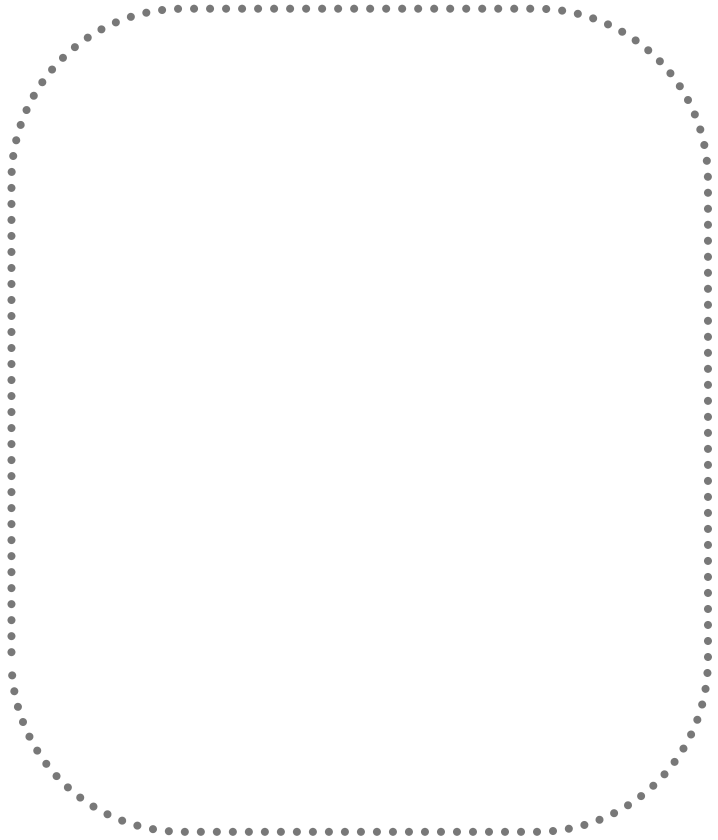
Startup.cs

```
services.AddTransient<ICommandHandler<EditPersonalInfoCommand>>(provider =>
    new DatabaseRetryDecorator<EditPersonalInfoCommand>(
        new EditPersonalInfoCommandHandler(provider.GetService<SessionFactory>()),
        provider.GetService<Config>()));
```

Decorator Pattern

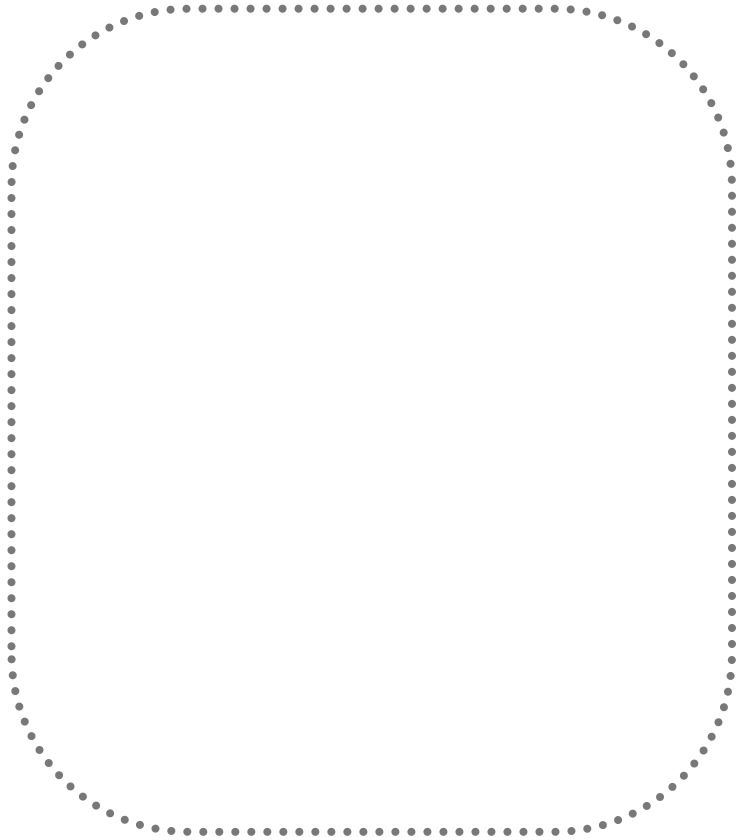


Decorator Pattern



Technical issues

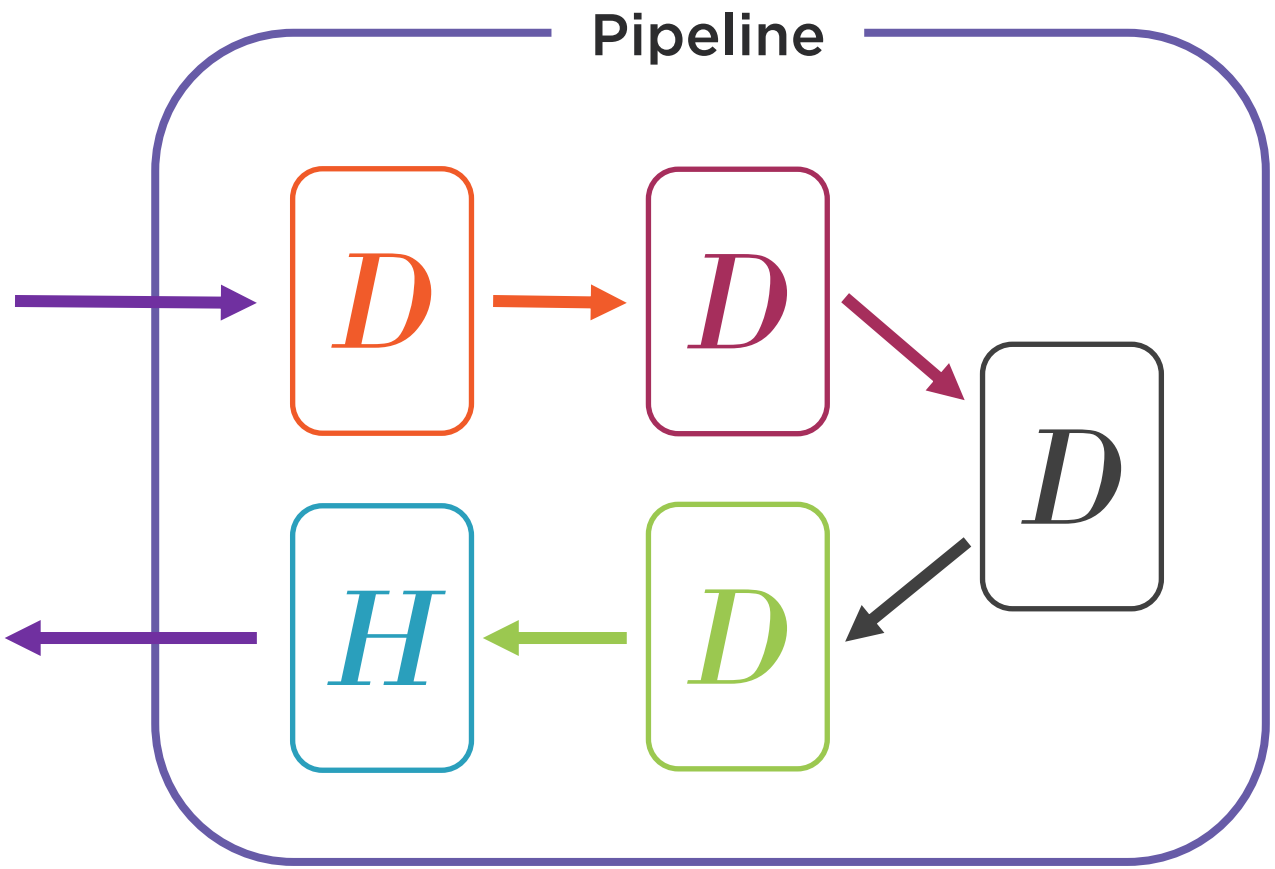
Business use cases



Decorators

Handlers

Decorator Pattern



- Rich functionality
- Simple components

Recap: Streamlining the Decorator Configuration

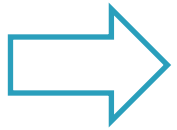
```
[DatabaseRetry]
[AuditLog]
public sealed class EditPersonalInfoCommandHandler :
    ICommandHandler<EditPersonalInfoCommand>
{
}
```

Simple and declarative

Cohesive

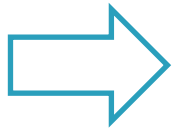
Recap: Streamlining the Decorator Configuration

```
[DatabaseRetry]
[AuditLog]
public sealed class EditPersonalInfoCommandHandler :
    ICommandHandler<EditPersonalInfoCommand>
{
}
```



Several log records

```
[AuditLog]
[DatabaseRetry]
public sealed class EditPersonalInfoCommandHandler :
    ICommandHandler<EditPersonalInfoCommand>
{
}
```



One log records

Use a DI library like Simple Injector

Decorators vs. ASP.NET Middleware

```
public sealed class ExceptionHandler {
    private readonly RequestDelegate _next;

    public ExceptionHandler(RequestDelegate next) {
        _next = next;
    }

    public async Task Invoke(HttpContext context) {
        try {
            await _next(context);
        }
        catch (Exception ex) {
            await HandleExceptionAsync(context, ex);
        }
    }
}
```

Decorator pattern

Accept “next”

```
public sealed class DatabaseRetryDecorator<T>
    : ICommandHandler<T> where T : ICommand
{
    private readonly ICommandHandler<T> _handler;

    public DatabaseRetryDecorator(ICommandHandler<T> handler) {
        _handler = handler;
    }

    public Result Handle(T command) {
        for (int i = 0; ; i++) {
            try {
                return _handler.Handle(command); }
            catch (Exception ex) {
                if (i >= _config.NumberOfDatabaseRetries
                    || !IsDatabaseException(ex))
                    throw;
            }
} } }
```

Decorator pattern

Accept “handler”

Decorators vs. ASP.NET Middleware

Decorators = ASP.NET Middleware

Controllers = Handlers

Decorators vs. ASP.NET Middleware

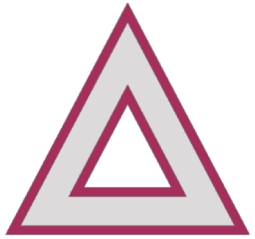
Why not use ASP.NET middleware instead of decorators?

Decorators vs. ASP.NET Middleware

Decorators vs. **Middleware**

- Additional control
- Separation of concerns
- Easy to apply selectively

Decorators vs. ASP.NET Middleware



Hard to tell ASP.NET to which API endpoints to apply the middleware

```
if (context.Request.Path.Value.StartsWith("/api/students/"))
```

```
[AuditLog]
[DatabaseRetry]
internal sealed class EditPersonalInfoCommandHandler
```

ASP.NET action filters

Decorators vs. ASP.NET Middleware

Decorators

- Additional control
- Separation of concerns
- Easy to apply selectively

vs.

Middleware

- Good for ubiquitous and ASP.NET-related cross-cutting concerns

Decorators vs. ASP.NET Middleware

```
public sealed class ExceptionHandler {
    private readonly RequestDelegate _next;

    public ExceptionHandler(RequestDelegate next) {
        _next = next;
    }

    public async Task Invoke(HttpContext context) {
        try {
            await _next(context);
        }
        catch (Exception ex) {
            await HandleExceptionAsync(context, ex);
        }
    }
}
```

Ubiquitous

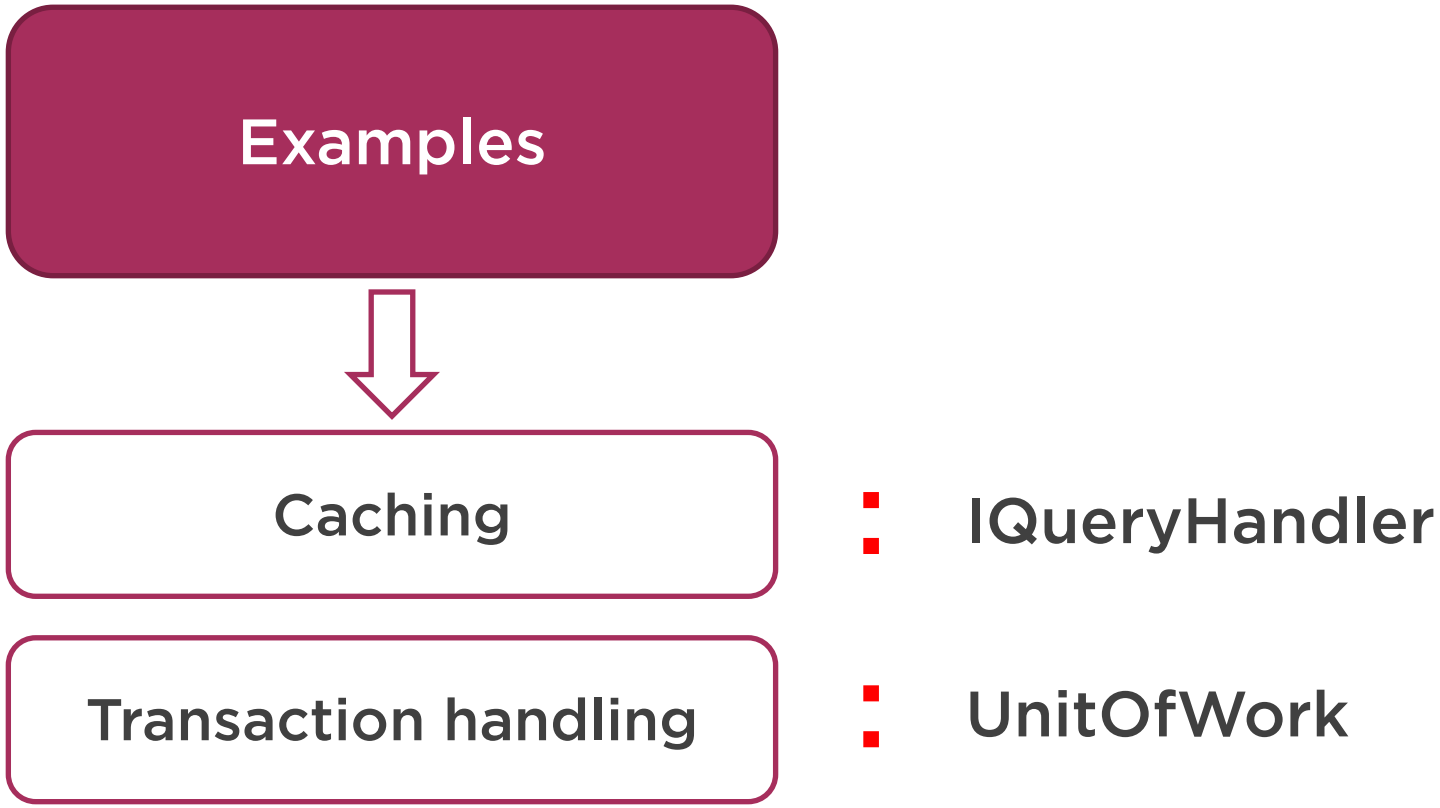
ASP.NET-related

Decorators vs. ASP.NET Middleware

Middleware = Ubiquitous and ASP.NET-related concerns

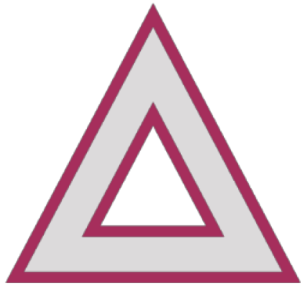
Decorators = Everything else

Decorators


Examples
Caching
IQueryHandler
Transaction handling
UnitOfWork

Command and Query Handlers Best Practices

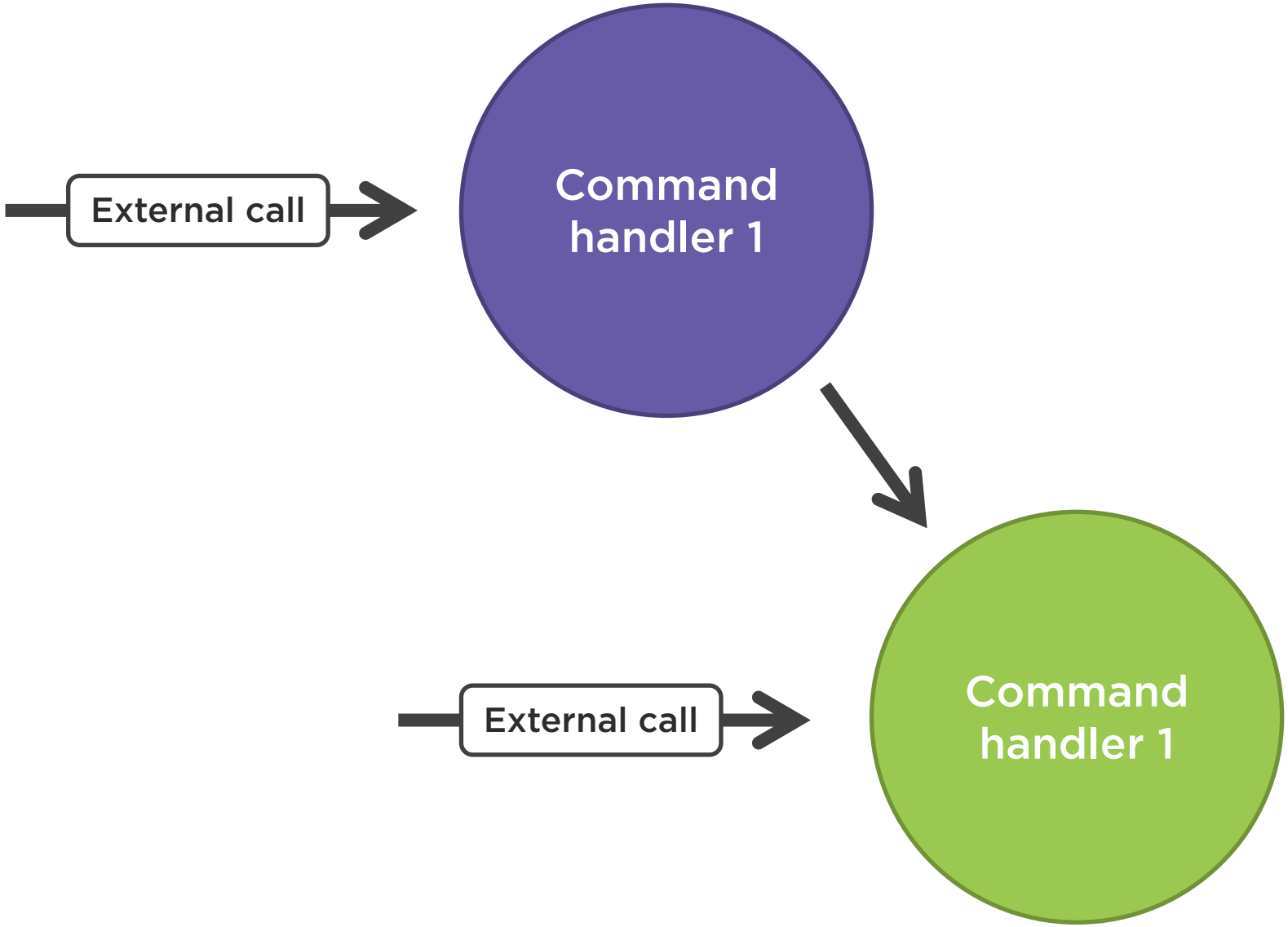
How should you organize commands, queries, and handlers?



Orphaned commands and queries

Don't reuse command handlers.

Command and Query Handlers Best Practices

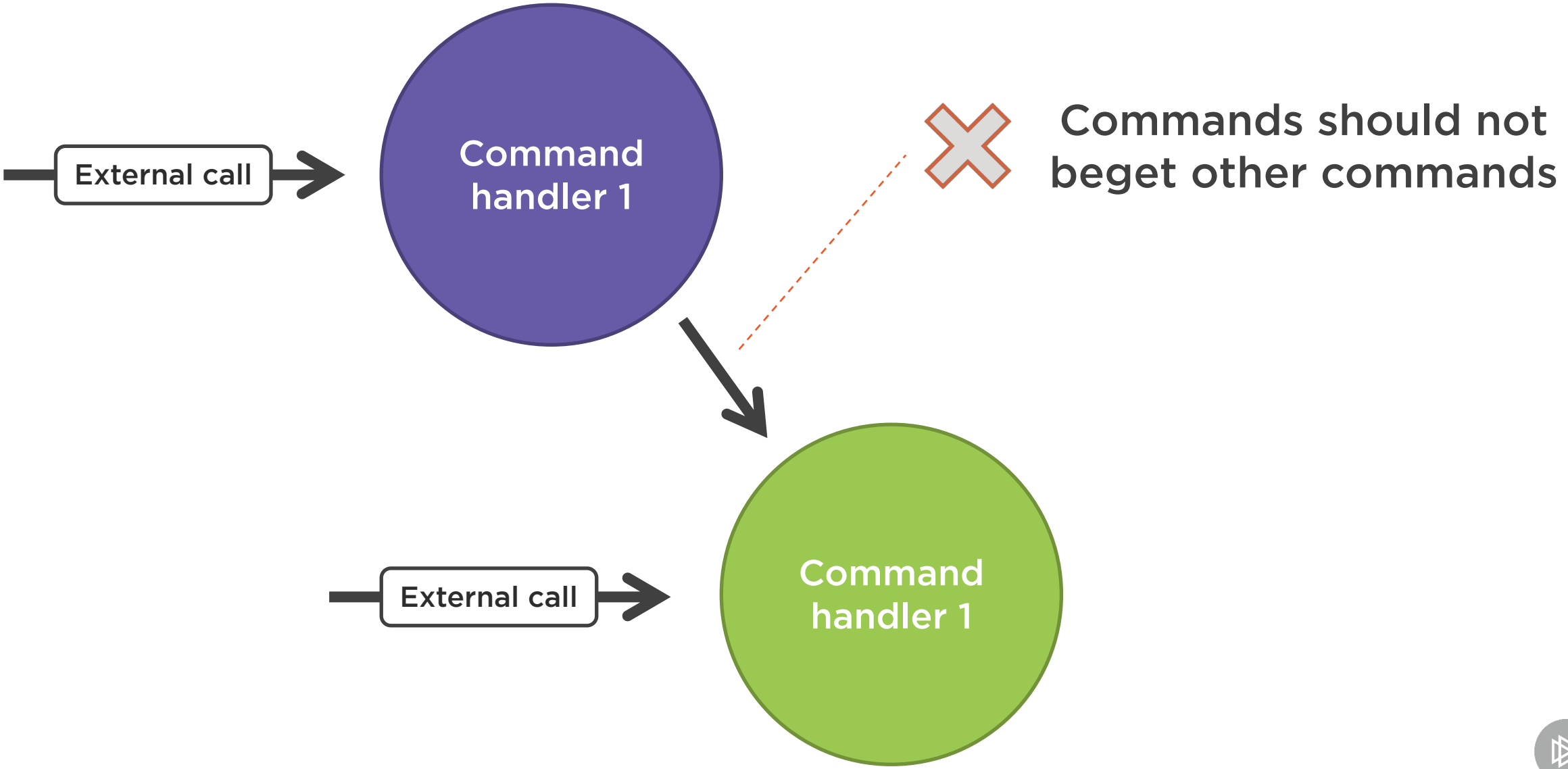


Command and Query Handlers Best Practices

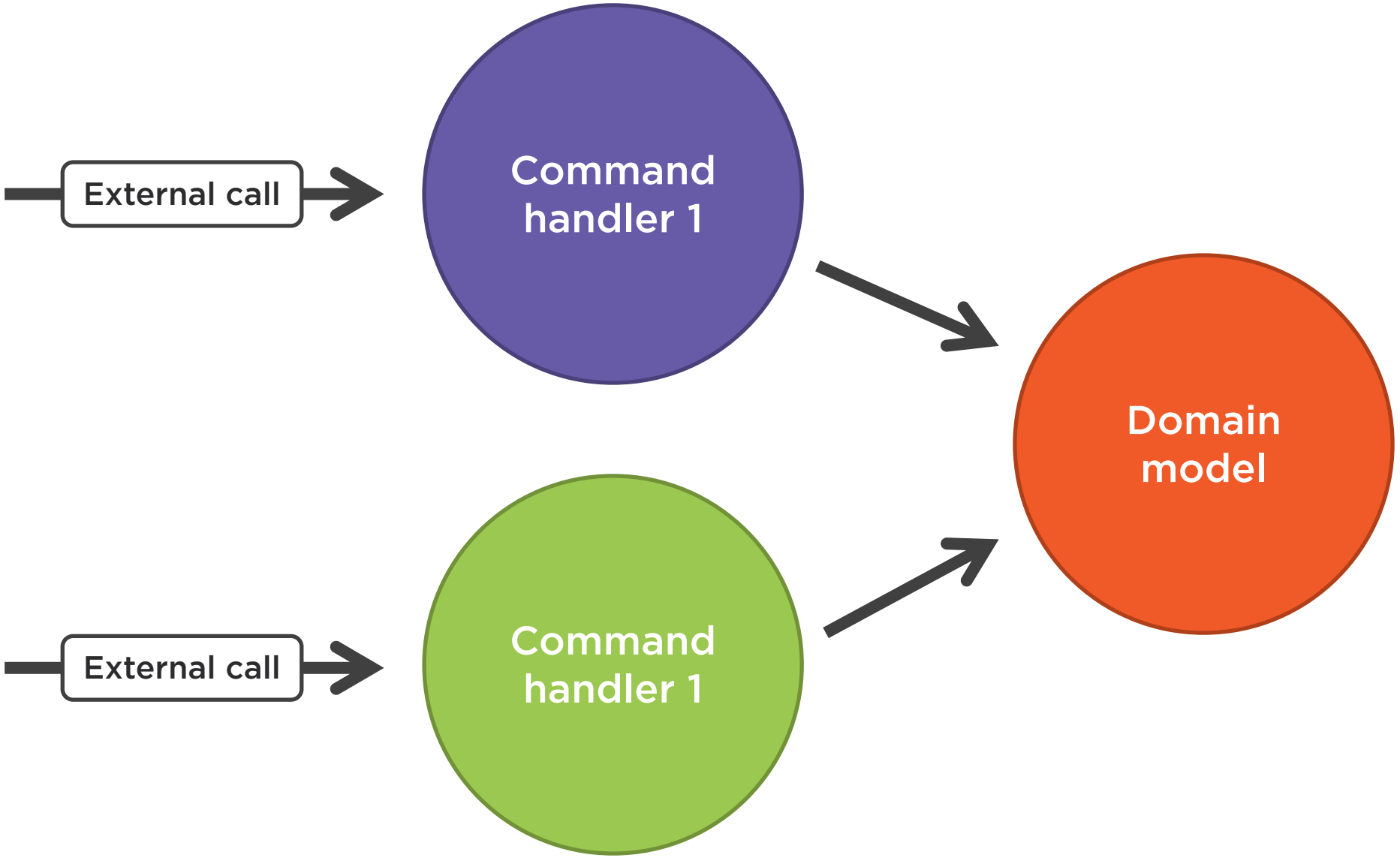
```
public sealed class UnregisterCommandHandler : ICommandHandler<UnregisterCommand>
{
    public Result Handle(UnregisterCommand command)
    {
        var unitOfWork = new UnitOfWork(_sessionFactory);
        var repository = new StudentRepository(unitOfWork);
        Student student = repository.GetById(command.Id);
        if (student == null)
            return Result.Fail($"No student found for Id {command.Id}");

        _messages.Dispatch(new DisenrollCommand(student.Id, 0, "Unregistering"));
        _messages.Dispatch(new DisenrollCommand(student.Id, 1, "Unregistering"));

        repository.Delete(student);
        unitOfWork.Commit();

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

Misuse of commands

Command and Query Handlers Best Practices



Command and Query Handlers Best Practices



Summary

Used decorators to extend command and query handlers

Decorator is a class that modifies the behavior of an another class without changing its public interface

- Allows for introducing cross-cutting concerns without code duplication
- Adherence to the single responsibility principle
- Chaining multiple decorators together allows for introduction of complex functionality

Streamlined the configuration of the decorators using attributes

- Great flexibility

Summary

Commonalities between the decorators and the ASP.NET middleware

- Both implement the decorator pattern
- Use middleware for ASP.NET-related functionality
- Use the decorators for everything else

Benefits in using hand-written decorators over ASP.NET middleware:

- Additional control over your code
- Separation of the application and ASP.NET concerns
- Better flexibility

Best practices around working with command and query handlers

- Put command and query handlers inside their respective commands and queries
- Don't reuse command handlers

In the Next Module

Simplifying the read model