Refactoring Towards Better Encapsulation

Vladimir Khorikov
PROGRAMMER

@vkhorikov www.enterprisecraftsmanship.com

Overview

Implementing proper encapsulation

Avoiding domain knowledge duplication

Working with SQL database efficiently

Combining specifications

Creation of new objects

Strongly Typed Specifications

Strongly typed specifications

Plain C# expressions

Generic specifications

Strongly Typed Specifications

Specifications: General Guidelines

MovieForKids

MPAA rating restrictions

Release date restrictions

AvailableOnCD

- Better encapsulation
- Easier to work with

Specifications: General Guidelines

```
public interface ISpecification<T>
{
    Expression<Func<T, bool>> ToExpression();
    bool IsSatisfiedBy(T entity);
}
```

```
public abstract class Specification<T> : ISpecification<T>
{
    /* ... */
}
```

YAGNI violation

Specifications: General Guidelines

- Avoid ISpecification interface
- Make specifications as specific as possible

Specifications: General Guidelines

```
public sealed class MovieForKidsSpecification : Specification<Movie> {
    public override Expression<Func<Movie, bool>> ToExpression() {
        return movie => movie.MpaaRating <= MpaaRating.PG;
    }
}

public class MpaaRatingAtMostSpecification : Specification<Movie> {
    private readonly MpaaRating _mpaaRating;

    public MpaaRatingAtMostSpecification(MpaaRating maxRating) {
        _mpaaRating = maxRating;
    }

    public override Expression<Func<Movie, bool>> ToExpression() {
        return movie => movie.MpaaRating <= _mpaaRating;
    }
}
```

Specifications: General Guidelines

```
public class MovieWithActorSpecification : Specification<Movie>
{
    private readonly string _actorName;

    public MovieWithActorSpecification(string actorName)
    {
        _actorName = actorName;
    }

    public override Expression<Func<Movie, bool>> ToExpression()
    {
        /* ... */
    }
}
```

Specifications: General Guidelines

```
public class MovieWithMpaaRatingBetweenSpecification : Specification<Movie>
{
    private readonly MpaaRating _min;
    private readonly MpaaRating _max;

    public MovieWithMpaaRatingBetweenSpecification(MpaaRating min, MpaaRating max)
    {
        _max = max;
        _min = min;
    }

    public override Expression<Func<Movie, bool>> ToExpression()
    {
        return movie => movie.MpaaRating >= _min && movie.MpaaRating <= _max;
    }
}
```

Specifications: General Guidelines

```
public sealed class MovieForKidsSpecification : Specification<Movie>
{
    public override Expression<Func<Movie, bool>> ToExpression()
    {
        return movie => movie.MpaaRating <= MpaaRating.PG;
    }
}
```

```
public class MovieForAdultsOnlySpecification : Specification<Movie>
{
    public override Expression<Func<Movie, bool>> ToExpression()
    {
        return movie => movie.MpaaRating >= MpaaRating.PG13;
    }
}
```

"Although programmers can be generous in providing parameters to customize, eventually they can make the parameterized specification too complex to use and difficult to maintain."

Martin Fowler and Eric Evans

Specifications: General Guidelines

- Avoid ISpecification interface
- Make specifications as specific as possible
- Make specifications immutable

Combining Specifications

Encapsulating specifications

Combining them together

Combining Specifications

```
var forKids = new MovieForKidsSpecification();
var onCD = new AvailableOnCDSpecification();
Specification<Movie> specification = forKids.And(onCD);
```

```
Specification<Movie> specification = forKids.Not().And(onCD);
```

And

Or

Not

Recap: Combining Specifications

- Combining specifications together
- Three types of combinations: And, Or, and Not

- Identity Specification
- Useful for dynamic search queries

When Not to Use Specifications

Only 1 out of 3 use cases

Search queries

~~In-memory validation~~

~~Search queries~~

In-memory validation

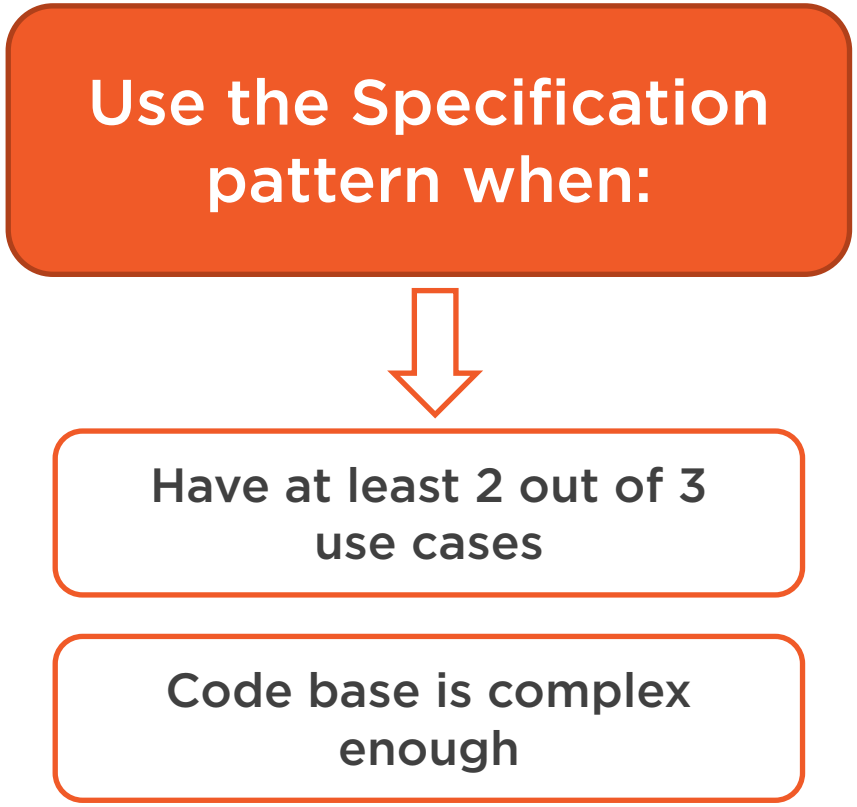
When Not to Use Specifications

Application is simple enough

Already low maintenance cost

Benefits might not justify the investment

When Not to Use Specifications



You don't have to represent all your search or validation capabilities as specifications.

Working with Multiple Classes

Recap: Working with Multiple Classes

- You can use specifications for related classes as well

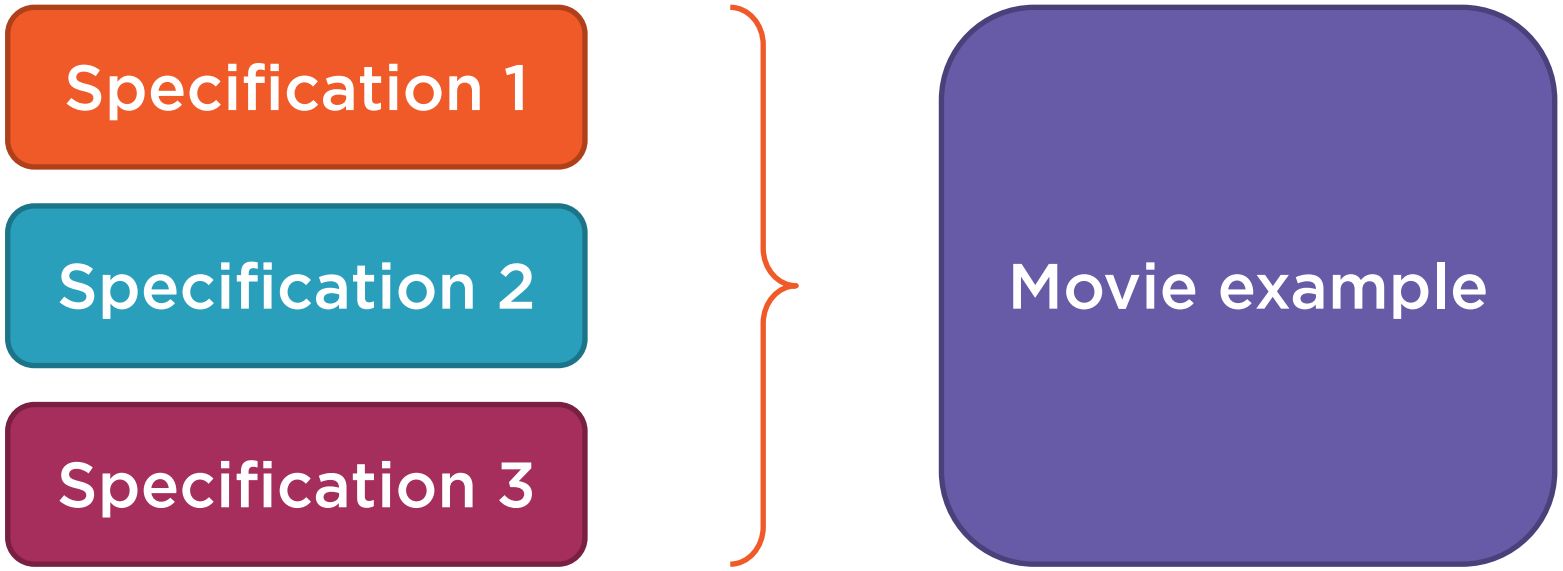
- Eagerly load all related objects when you fetch data from the database

"A way to describe what an object might do, without explaining the details of how the object does it, but in such a way that a candidate might be built to fulfill the requirement."

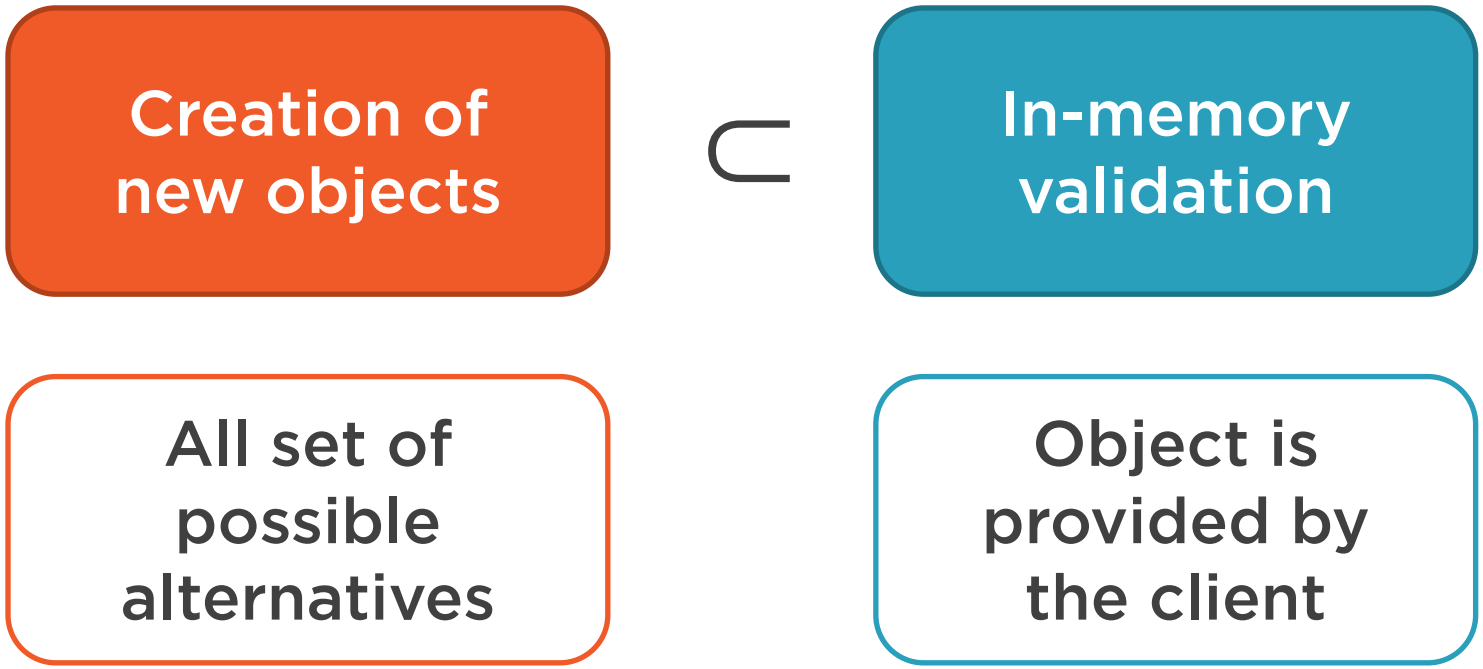
Martin Fowler and Eric Evans

Creation of New Objects

Demand for kids movies



Creation of New Objects



Brut force

Module Summary

Proper encapsulation and efficient database queries

- Specifications should contain the domain knowledge

General guidelines

- Don’t use the ISpecification interface
- Make specifications as specific as possible
- Make specifications immutable

Combining specifications together

- And, Or, and Not
- Identity specification

When not to use the Specification pattern

Combining specifications with regular filtration

Creation of new objects

Resource List

Source code	https://github.com/vkhorikov/SpecPattern
	http://bit.ly/spec-pat
Specification pattern	https://martinfowler.com/apsupp/spec.pdf
	http://bit.ly/spec-pattern
Dry principle	http://enterprisecraftsmanship.com/2015/09/11/dry-revisited/
	http://bit.ly/dry-prin

Course Summary

Use cases for the specification pattern

- In-memory validation
- Querying the database
- Creation of new objects

Plain C# expressions and generic specifications are not enough

Use strongly typed specifications

Contacts

vladimir.khorikov@gmail.com

@vkhorikov

http://enterprisecraftsmanship.com/