Turning Optional Calls into Calls on Optional Objects

Zoran Horvat
OWNER AT CODING HELMET CONSULTANCY

@zoranh75 codinghelmet.com

Requirements

Electrical circuitry may fail

- Customer may then claim circuitry warranty

Warranty is claimed with date when malfunction was detected

- This complication is intended to render Null Object not applicable

Dealing with Null References

Null Object and Special Case

Can replace most of the occurrences of null

Optional objects

Handle cases where Null Object and Special Case are not applicable

Case #1 – Branching

```
if (Circuitry != null)
{
  Circuitry
    .MarkDefective(date);

  CircuitryWarranty = warranty;
}
```

Case #2 – Null Object

```
Circuitry = SomeObject;

Circuitry.MarkDefective(date);

CircuitryWarranty = warranty;
```

- **Can we introduce Null Object here?**
- **Suppose that we can apply it**
  `SomeObject` is Null Object
- `MarkDefective()` **would do nothing**
  This is what Null Object can do
- **But Null Object cannot stop this operation from executing**

Case #1 – Branching

```
if (Circuitry != null)
{
  CircuitryWarranty
    .Claim(
      Circuitry.DefectDetectedOn,
      action);
}
```

Case #2 – Null Object

```
Circuitry = SomeObject;

DateTime date =
  Circuitry.DefectDetectedOn;

Warranty.Claim(date, action)
```

- **Can we introduce Null Object here?**
- **Suppose that we can apply it**
  `SomeObject` is Null Object
- `DefectDetectedOn` **returns date**
  Null Object has to make it up
- **But `Claim()` can still make changes to the system**

```
if (Circuitry != null)
{
  Circuitry
    .MarkDefective(date);

  CircuitryWarranty = warranty;
}
```

- **Do we have to fall back to branching?**
- **Understanding null conditions**

  If there is an object…

  Then perform operation
- **We can call this “optional call”**
- **Optional object idea**

  Perform operation unconditionally

  Optional object does nothing if it is empty

Null Object vs. Null Reference

Null Object implementation

```
Circuitry = nullObj;

Warranty.Claim(
  Circuitry.DefectDetectedOn,
  onValidClaim);
```

`DefectDetectedOn` **is always called**

`Claim()` **is always invoked**

We want `Circuitry` **to always be non-null**

Null reference implementation

```
if (Circuitry != null)
{
  Warranty.Claim(
    Circuitry.DefectDetectedOn,
    onValidClaim);
}
```

`DefectDetectedOn` **not called if null**

`Claim()` **not invoked if null**

We want `Claim()` **to be invoked only if circuitry exists**

Designing the Optional Object

-null object which indicates
in it or there is nothing in it?

a collection.
ntain more than one element.

Optional Objects: The Working Principle

Implemented as the collection

Contain one object, or

Contain no objects

Turn calls optional

Call if there is an object inside

No call if there is no object inside

Collect the benefits

No more null tests

No more null returns

Note on Functional Programming Langauges

against a pattern

against a pattern

r to Pluralsight courses on F#

Demo

New `Option` class has been implemented

Entire implementation will not be shown here

One more important aspect will be discussed

- Steps to write code which produces proper fluent interface

```
name

   .When(s => s.Length > 3)


   .Do(s =>
       Console.WriteLine($"{s} long"))

   .WhenSome()
   .Do(s =>
       Console.WriteLine($"{s} short"))

   .WhenNone()
   .Do(() =>
       Console.WriteLine("missing"))

   .Execute();
```

- **What will appear after dot?**
- **Only** `When…()` **methods should be available**
- **Return from** `When()` **exposes** `Do()` **or** `MapTo()`
- **Return from** `Do()` **rules out** `MapTo()` **on all subsequent objects**
  Only `When…()` and `Execute()` are available after the first `Do()` call
- **Final** `Execute()` **following the last** `Do()` **terminates the chain**

```csharp
string label =
  name
    .When(s => s.Length > 3)
    .MapTo(s =>
           s.Substring(0, 3) + ".")

    .WhenSome()
    .MapTo(s => s)

    .WhenNone()
    .MapTo(() => "<empty>")

    .Map();
```

- **Similar sequence here**
- **Final call is `Map()` and it returns a string in this example**
- **This magic is supported by a system of types**
- **Each call returns new object which carries knowledge accumulated in all previous calls**
- **The first `MapTo()` call rules out `Do()` calls further on**
- **Only compatible `MapTo()` method calls can follow**
  Those `MapTo()` which map to the same resulting type
- **Keep C# code strongly typed**
  Compiler then helps find errors

Symptoms of Combinatorial Explosion

doubles" again

grows by an order of magnitude

Consequences of Combinatorial Explosion

Each class alone gets complicated

Lots of duplicated code that cannot be removed

Duplication appears in classes without the common base type

Orthogonal Concepts Lead to Explosion

Consequence:
2^4=16 classes to implement them!

```csharp
class Option<T> : IEnumerable<T>
{
  private IEnumerable<T> Content { get; }

  private Option(IEnumerable<T> content)
  {
    this.Content = content;
  }

  public static Option<T> Some(T value) =>
    new Option<T>(new[] {value});

  public static Option<T> None() =>
    new Option<T>(new T[0]);

  public IEnumerator<T> GetEnumerator() =>
    this.Content.GetEnumerator();

  IEnumerator IEnumerable.GetEnumerator() =>
    this.GetEnumerator();

}
```

- Each of my projects has `Option` implementation similar to this
- Version implementing `IEnumerable<T>` is easy to use
- It lacks pattern matching
- Single-line methods don’t require pattern matching

Object-oriented Design Method

Start from the calling side

Let the caller define interface it needs

Let interfaces and classes grow

Type safety saves the caller from mistakes

Design custom types to navigate the caller

Objects should only expose methods that are safe to call

Invalid calls must be impossible

Current object must not expose inaccessible methods

Summary

There are cases where an object is just missing

- That renders Null Object and Special Case patterns non-applicable
- That doesn’t mean we should use null

Alternative to null are Options

- Option either contains an object, or contains nothing
- Turns code readable again
- No chance of `NullReferenceException` improves stability

Next module –
Avoiding `switch` *instruction*