Turning Algorithms into Strategy Objects

Zoran Horvat

OWNER AT CODING HELMET CONSULTANCY

@zoranh75 codinghelmet.com

Dealing with Varying Algorithms

Some algorithms show tendency to vary in time

How can we vary a piece of analytical code?

By turning it into an object!

Objects can be varied by replacement

The Goal of This Module

Turn an algorithm into a replaceable component

Then replace the algorithm at run time when needed

```
class CompositePainter
{
  .ctor(IEnumerable<IPainter>,
        Func<double, IEnumerable<IPainter>, IPainter>);
}

CompositePainter obj =
  new CompositePainter(sequence, lambda);
```

- This class doesn't require a derived class to be operational

- Compose an object by passing an external lambda expression

```
class CombiningPainter : CompositePainter
{
  .ctor(IEnumerable<IPainter> painters)
    : base(painters, this.MyFunction);

  IPainter MyFunction(double, IEnumerable<IPainter>>);
}

CompositePainter obj =
  new CombiningPainter(sequence);
```

- It is still possible to derive a class
  Subclass can pass own member as constructor parameter

- New object instantiated with one constructor parameter less

- Derived class has reduced number of base constructor parameters

Composition vs. Inheritance

Composition is more flexible

Compose a new object
to replace the existing one

Inheritance reduces requirements

Consumer can instantiate
derived class without knowing
all the details

**Use both,
but favor composition more**

**Add inheritance later
to improve consuming
code readability**

```
private IPainter Combine(double sqMeters,
  IEnumerable<ProportionalPainter> painters)
{

  TimeSpan time =
      TimeSpan.FromHours(
        1 /
        painters
          .Where(painter => painter.IsAvailable)
          .Select(painter =>
            1 /
            painter
              .EstimateTimeToPaint(sqMeters)
              .TotalHours)
          .Sum());

  double cost =
    painters
      .Where(painter => painter.IsAvailable)
      .Select(painter =>
              painter.EstimateCompensation(sqMeters) /
              painter.EstimateTimeToPaint(sqMeters)
                .TotalHours * time.TotalHours)
            .Sum();

  return new ProportionalPainter()
  {
    TimePerSqMeter =
      TimeSpan.FromHours(time.TotalHours / sqMeters),
    DollarsPerHour = cost / time.TotalHours
  };
}
```

- **This method only works correctly on sequence of `ProportionalPainter` objects**
- **We want to make it more universal**
- **This algorithm has some major qualities**

  Works in time proportional to length of the sequence

  Non-linear painters might require a converging algorithm
- **Figure which parts of this algorithm are universal**

  E.g. if we only knew time and money estimates...

Creating Proper Template Methods

Not every algorithm can become general

Or we might have to modify it before making it general

Evolving the Design

Always start with something that works

Refactor one step at the time

Move in direction of better design

But it must still work correctly

Traits of design improvements

Feature consumer is typically simplified

Extensive use of `Func`, `IEnumerable`, `Tuple`...

Dealing with Future Requirements

Don't change existing classes when a requirement is changed or added

Add one more class to the design instead

Replace objects at run time to select one implementation or the other

Why Insist on This Principle?

Change in code may cause regression

Prior state in which some feature was not operational

Change in code causes bugs in features that used to work fine

Try not to change code

Add new classes instead

Summary

The problem of generalizing an algorithm

- Make it applicable to different kinds of input objects

How to reuse an algorithm?

- Solution #1: copy & paste
- Solution #2: do something better

Summary

General way to generalize algorithms

- Restructure the algorithm
- Make universal parts stand apart from specific, varying parts
- Wrap varying parts into a collaborating class
- Replace strategy object at run time

Next module –

Using Immutable Objects