# Untangling Structure from Operations on Business Data


**Zoran Horvat**

OWNER AT CODING HELMET CONSULTANCY

@zoranh75 codinghelmet.com

# Two Faces of Abstract Types in Programming

**Corresponds to a physical phenomenon**

`IPainter` class represents a *person* who paints the wall

**Simplifies another phenomenon (real or not)**

`IPainter` can also be a *group of people* working together

# Composite Pattern

**Compose a number of objects into a new object exposing the same public interface.**

**Composite object looks like a single object**

It can receive method calls just like a single object does

# Naming Concrete Classes

**Give concrete classes meaningful names**

lementing an interface

`Painter` **implementing** `IPainter`

`ProportionalPainter` **implements** `IPainter` **and gives it a meaning**

# Recognizing Non-Object-Oriented Code

**Client is doing everything by itself**

It should call other objects to do the work instead

The client is complicated

**Added requirement asks for added code in the client**

It should ask for adding a new class instead

Existing classes should remain the same

# Name Classes the Same as You Talk

If talking about painters, avoid presenting `IEnumerable<Painter>`

Avoid artificial mapping between spoken language and code

We speak of painters, and see the type named `Painters`

# Keep Abstractions Abstract

`Proportional-Painter` class didn't tell out that an object contains other objects

`Painters` class communicates that it contains other objects

Communicate this information only if it helps the client

# Should We Expose a Collection?

**Are we only interested in having the wall painted?**

It's not important if there are multiple painters working

**Are we interested to organize workers?**

It is important to know how many of them there are

```
painters
    .GetAvailable()
    .GetCheapestOne(sqMeters);
```

```
painters
    .FindCheapest(sqMeters);
```

```
painters
    // No call to GetAvailable!
    .GetCheapestOne(sqMeters);
```

- **Consumer is calling two primitives to complete the operation**
  - Isolate available painters
  - Then pick the cheapest one
- **Should we add this combination to the `Painters` class?**
- **No, feature provider should only expose primitives.**
- **What if the consumer chooses not to filter unavailable painters out? (E.g. the consumer is investigating the market.)**
- **For that reason, do not combine primitives at the providing end.**

# Delegate vs. Abstract Method

## Func/Action Delegate

Delegate can be supplied from the outside

We can extend the implementation without adding new classes

## Abstract Method in the Base Class

Abstract method requires a derived class

We have to add new derived classes in order to extend the implementation

# Summary

**The problem of repeatedly iterating through a sequence**

- Wrap the sequence in a class
- Do not expose general-purpose collection methods
- Expose meaningful primitive operations instead

**Client code becomes simpler**

- Client combines primitive methods and builds larger features

# Summary

## Composite pattern

- Container exposes the same interface as its contained elements
- Construct a special mapping function
- Maps many objects into a single object of the same kind

## Collection vs. Composite idea

- Both are valid and useful

***Next module –***
*Turning Algorithms into Strategy Objects*