Class Code Smells and Refactorings

Steve Smith
FORCE MULTIPLIER FOR DEV TEAMS
@ardalis | ardalis.com | weeklydevtips.com

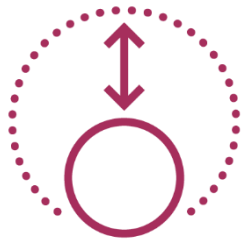
Objectives

Learn various class-related code smells

Learn refactoring techniques to address them

Smell: Large Class

A large class probably has too many responsibilities and can be split into two or more smaller, more focused and cohesive classes.



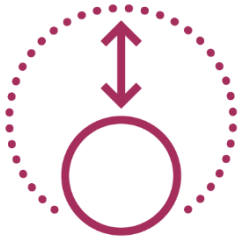
Bloater

Refactoring Large Classes

- Extract Method
- Extract Class
- Extract Subclass or Interface
- Replace Conditional Dispatcher with Command
- Replace State-Altering Conditionals with State
- Replace Implicit Language with Interpreter

Smell: Class Doesn't Do Much

Sometimes after refactoring or redesign a class is left without much to do. If whatever it does makes more sense somewhere else, pull it out and delete the class.



Bloater

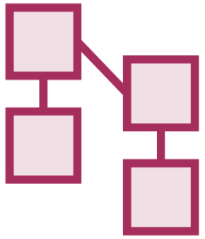
Smell: Lazy Class

A class that doesn't do enough to justify its existence.

Dispensable

Smell: Temporary Field

Class has a field that is only set in certain situations, often used to pass state between methods instead of using parameters.



Object-Orientation Abuser

Temporary Field

```
public class Employee
{
  private decimal _earningsForBonus;

// other fields and methods

  private decimal CalculateBonus()
  {
    return _earningsForBonus * BonusPercentage();
  }


  private void CalculateEarningsForBonus()
  {
    _earningsForBonus = YearToDateEarnings() + OvertimeEarnings() * 2;
  }
}
```

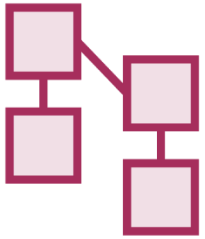
Temporary Field (Refactored)

```
public class BonusCalculator
{
    private Employee _employee;

    public BonusCalculator(Employee employee)
    {
        _employee = employee;
    }
    public decimal CalculateBonus()
    {
        return CalculateEarningsForBonus() * _employee.BonusPercentage();
    }
    private decimal CalculateEarningsForBonus()
    {
        return _employee.YearToDateEarnings() + _employee.OvertimeEarnings() * 2;
    }
}
```

Smell: Alternative Classes with Different Interfaces

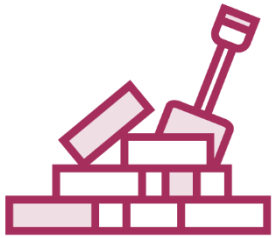
Common operations should share common semantics, such as names, parameters, and parameter orders.



Object-Orientation Abuser

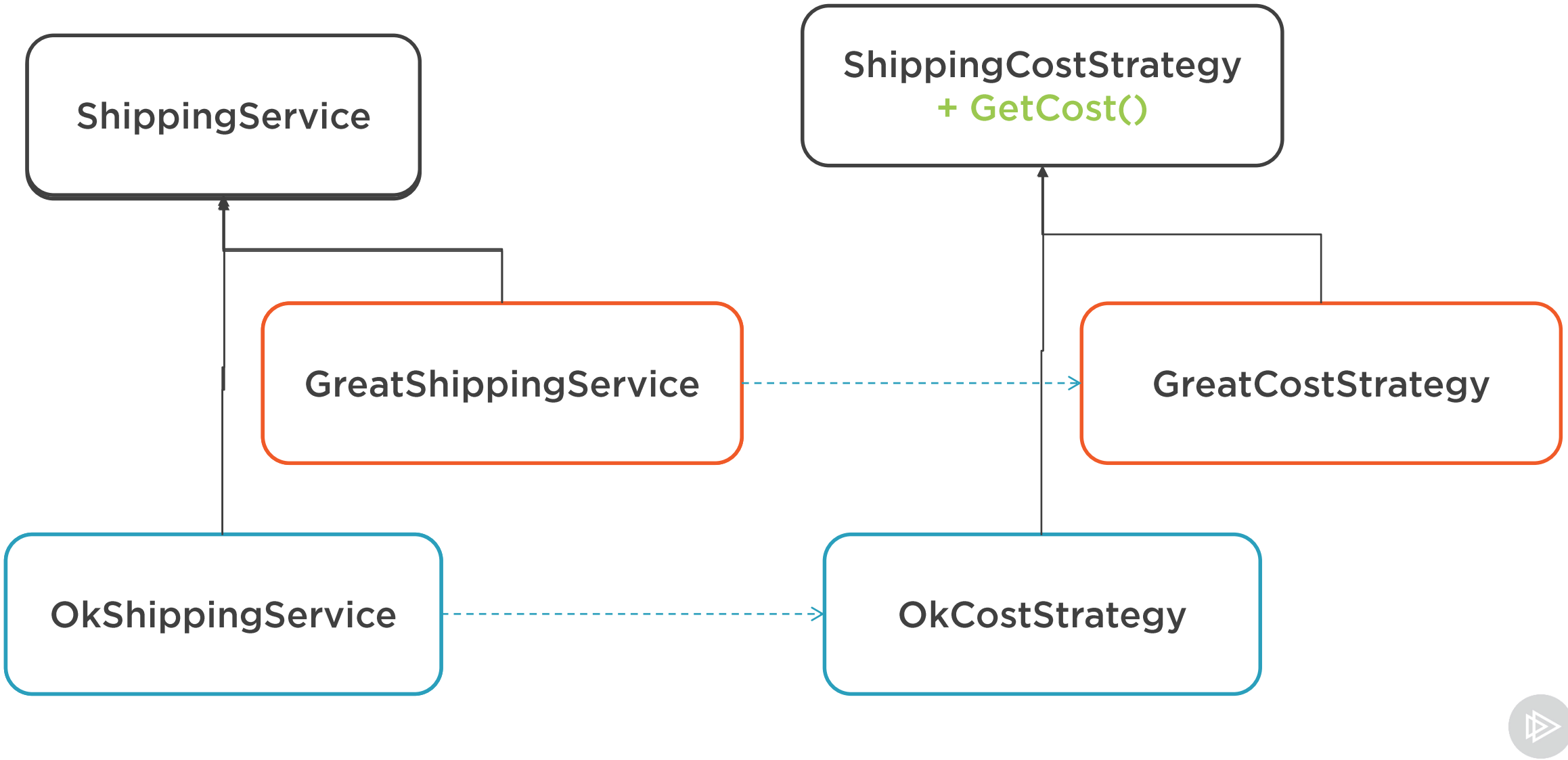
Smell: Parallel Inheritance Hierarchies

Two concepts are both modeled with inheritance hierarchies, and any change in one requires a matching change in the other.

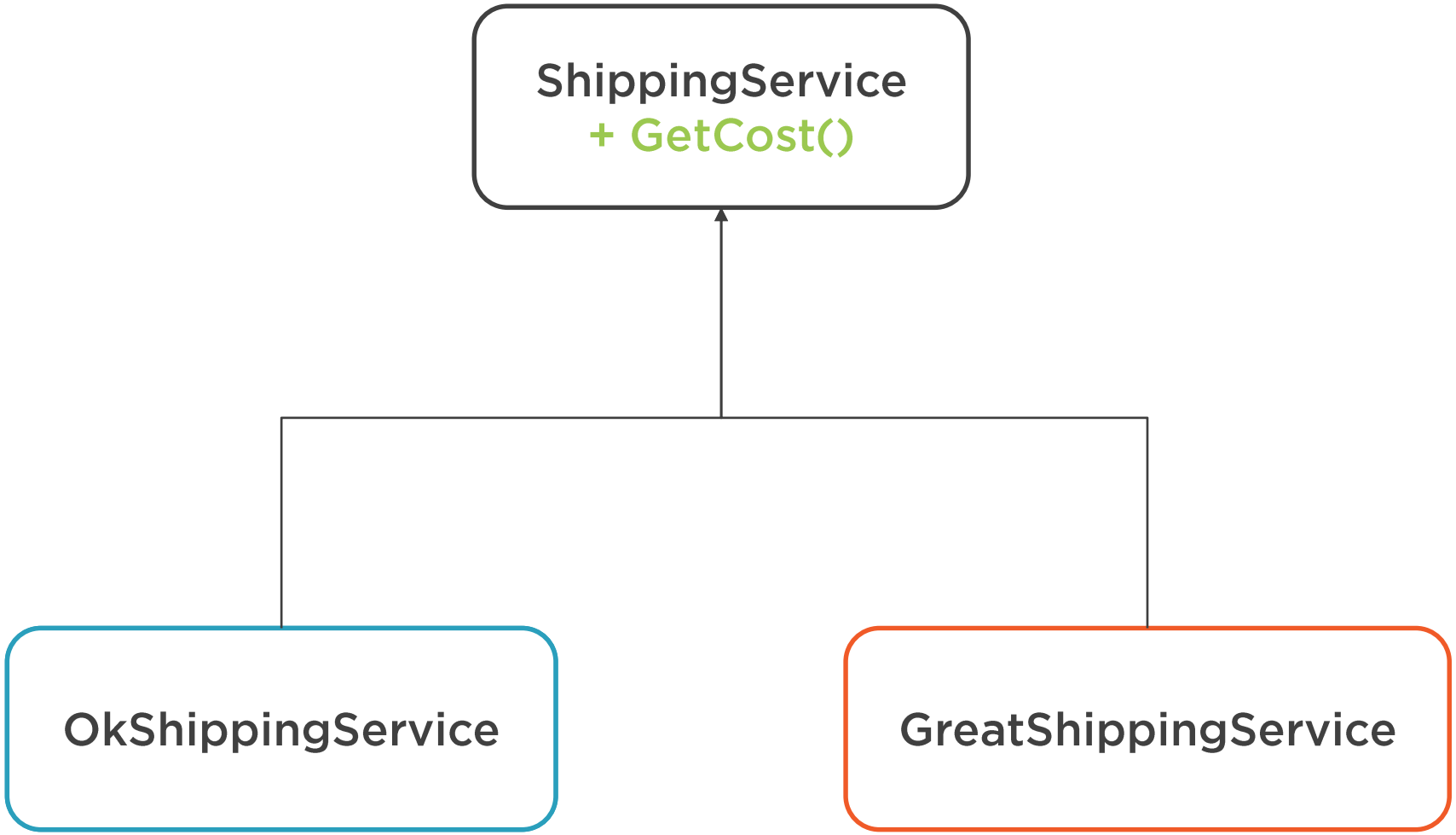


Change Preventer

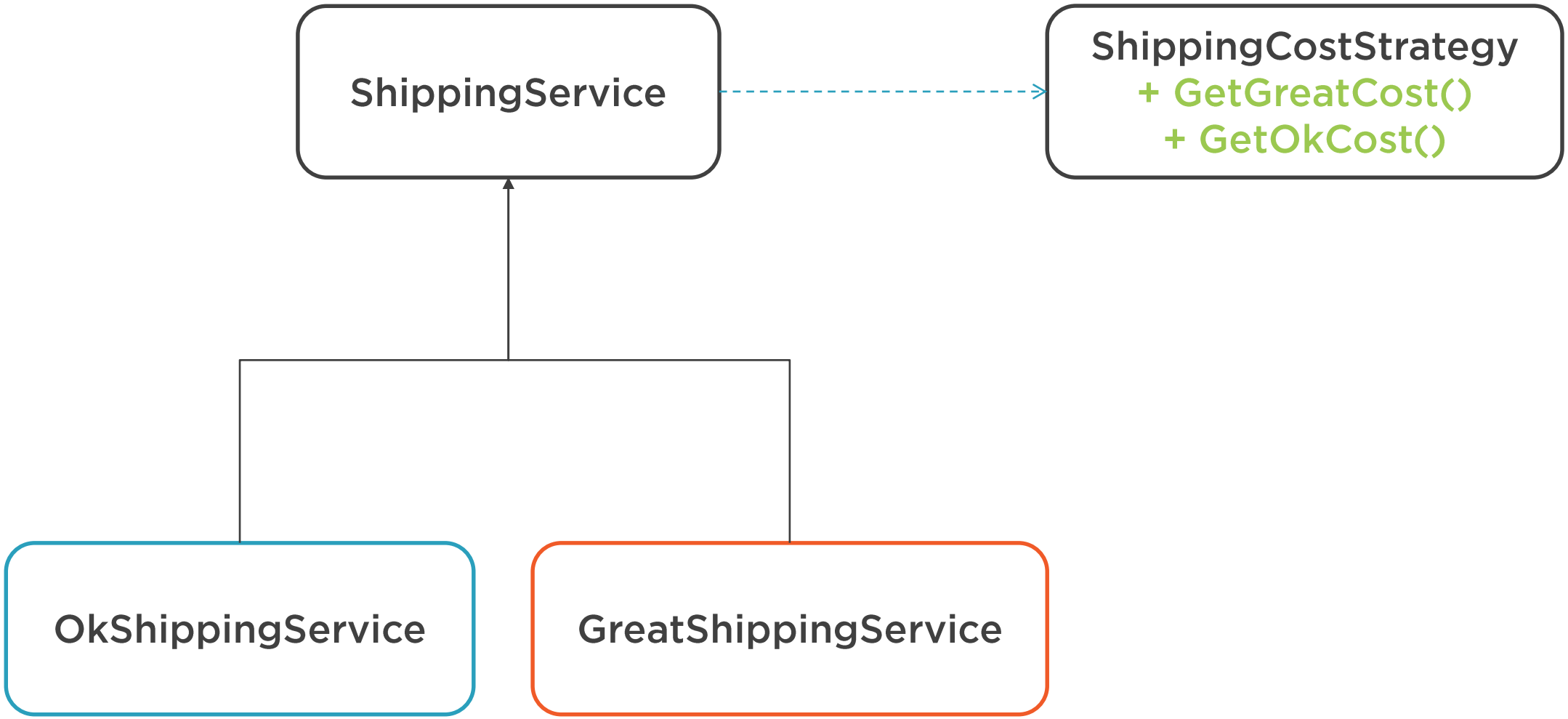
Parallel Inheritance Hierarchies



Parallel Inheritance Hierarchies



Parallel Inheritance Hierarchies



Smell: Data Class

Classes that lack behavior and have only fields and/or properties. Such classes lack encapsulation and must be manipulated by other classes, rather than bundling state and behavior together.

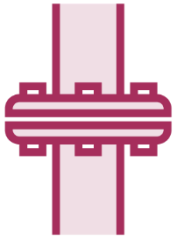
Dispensable

Demo

Data Class

Smell: Feature Envy

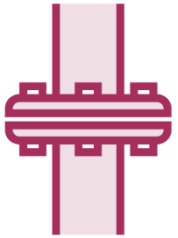
Occurs when behavior lives in one object but requires data from another. Related to Data Class, but can also occur between two classes that each have their own behavior.



Coupler

Smell: Hidden Dependencies

Classes have external dependencies they do not specify through their constructors. Calling code must inspect the class (or discover runtime errors) to identify its dependencies. Instead, follow the Explicit Dependencies Principle.

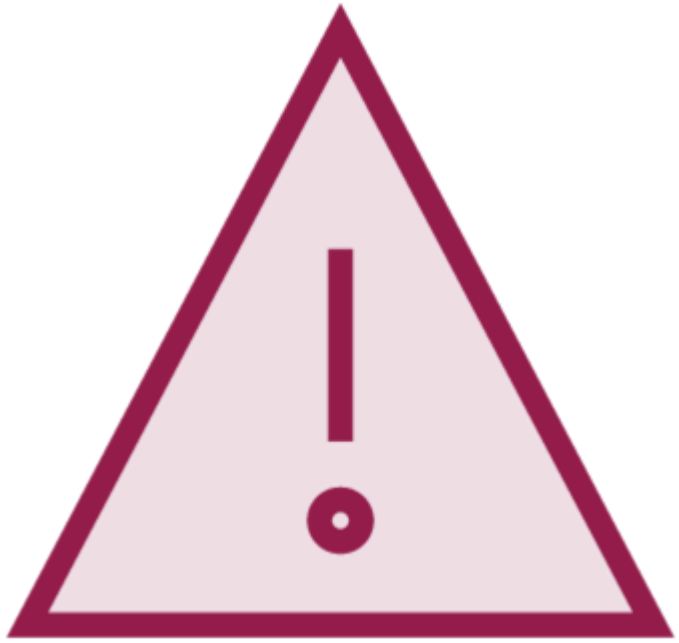


Coupler

Specific Class Refactorings

Encapsulate Field

1. Create get/set property accessors for the field
   - Optionally, use an auto-property
2. Find all references to the field
3. Replace each reference to use the property
4. Compile after each change
5. Change the field to be private
6. Compile and Run Tests



Be careful removing setters and setter access from entities and DTOs

EF requires setters to populate entities

- Private setters will still work

DTOs used for serialization or model binding must have public setters

Encapsulate Collection

1. Add explicit Add/Remove item methods
2. Initialize the field to an empty collection
3. Compile
4. Find references that set the collection
   - Modify to use new Add/Remove methods
5. Compile and Run Tests
6. Find references that get the collection, then modify its contents
   - Modify to use new Add/Remove methods
7. Compile and Run Tests
8. Modify the property get to return a read-only collection
9. Compile and Run Tests

Demo

Encapsulated Collections

“Moving methods is the bread and butter of refactoring.”

Martin Fowler

Move Method

1. **Examine all fields/methods used by the method and defined in its current class**
   - Consider whether you can move them all
   - Be sure to check sub- and super-classes

   Declare the method on the target class
   - Copy the code from the source method and adjust to make it work
2. **Compile**
3. **Reference the target class's method from the original class**
   1. Delegate from the old method to the new one
4. **Compile and Run Tests**

Extract Class

1. Determine how to split the class's responsibilities
2. Create a new class to hold the split-off responsibility
3. Add a field to the old class with the type of the new one
4. Use Move Method (and Move Field if necessary) to move class members
5. Compile and Run Tests
6. Decide whether/how to expose the new class through the original class

Replace Inheritance with Delegation

1. Create and initialize a field in the subclass with the type of the superclass
2. Change each method defined in the subclass to use this new field
3. Compile and Run Tests
4. Remove the subclass declaration
5. For each superclass method a client uses, add a delegating method
6. Compile and Run Tests

Demo

Replace Inheritance with Delegation

Replace Conditional with Polymorphism

1. If necessary, Extract Method to get the conditional in its own method
2. If necessary, Move Method to pull the conditional to the top of the inheritance hierarchy
3. Choose one subclass, override the conditional method
    1. Copy the appropriate body of the conditional into the subclass's method
4. Compile and Run Tests
5. Remove that leg of the conditional
6. Compile and Run Tests
7. Repeat until all parts of the conditional have their own methods
8. Make the superclass method abstract

Demo

Replace Conditional with Polymorphism

Key Takeaways

- Large Class
- Class Doesn’t Do Much / Lazy Class
- Temporary Field
- Alternative Classes with Different Interfaces
- Parallel Inheritance Hierarchies
- Data Class
- Feature Envy
- Hidden Dependencies
- Encapsulate Field / Collection
- Move Method
- Extract Class
- Replace Inheritance with Delegation
- Replace Conditional with Polymorphism