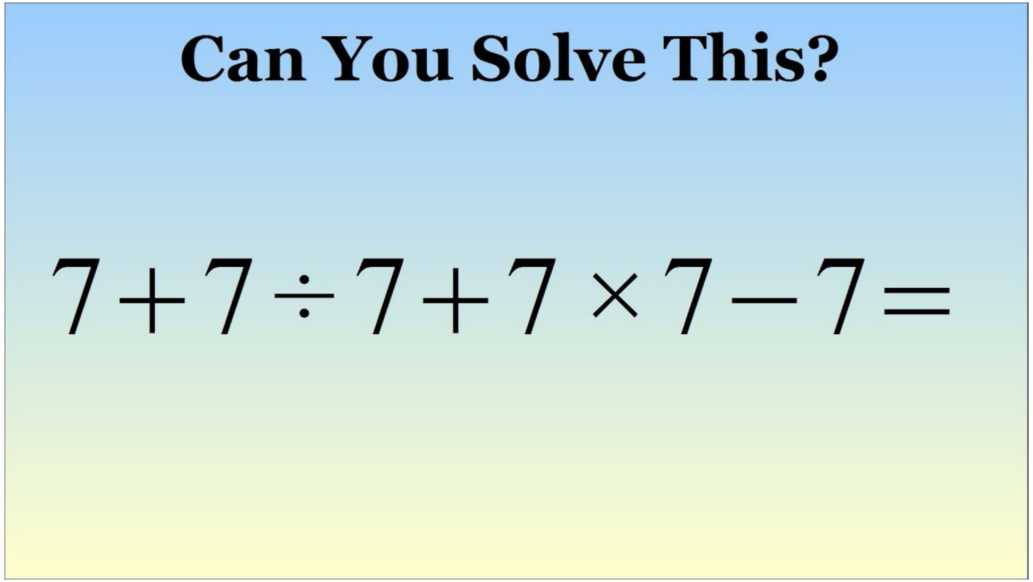
COMS W4156 Advanced Software Engineering (ASE)

November 1, 2022

Agenda

1. Questions about First Individual Assessment
2. Refactoring


Can You Solve This?

$$7 + 7 \div 7 + 7 \times 7 - 7 =$$

First Individual Assessment

due 11:59pm November 4, that's this Friday!

First Assessment Contribution to Overall Grade

First assessment ~= Midterm exam

20% of overall grade

20% matters!

Must submit by uploading to this assignment in courseworks, we will not grade assessments submitted any other way

First Assessment Timing

The assessment should have become available in courseworks at 12:01am this morning, did it?

The assessment is due by 11:59pm this Friday

Should take about four hours but you have up to four days to work on it

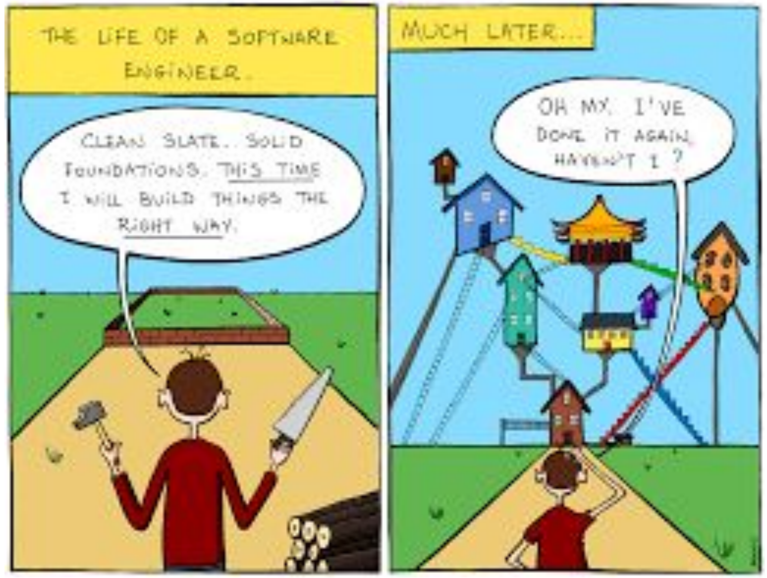
Not timed!

Questions?

Now is the best time to ask questions

Agenda

1. Questions about First Individual Assessment
2. Refactoring

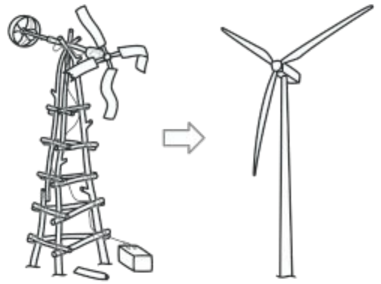

THE LIFE OF A SOFTWARE ENGINEER.
CLEAN SLATE. SOLID FOUNDATIONS. THIS TIME I WILL BUILD THINGS THE RIGHT WAY.
MUCH LATER...
OH MY. I'VE DONE IT AGAIN, HAVEN'T I ?

What is Refactoring?

In various earlier lectures, I've mentioned "refactoring" many times - not always compliant with its technical meaning

Technically, refactoring is a systematic approach to improving code quality

Refactoring operations (or recipes) should be very small sequences of edits that are *semantics-preserving:* The edits should not change the functionality of the program as a whole, so unchanged system-level functional tests should get exactly the same results before and after each refactoring operation



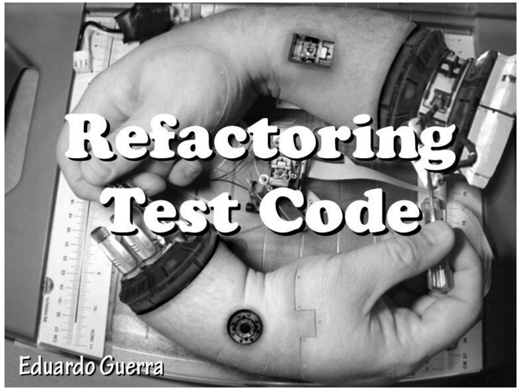
Testing During Refactoring

Ideally, all system-level tests are indeed rerun before and after each refactoring operation to verify that nothing was broken - if any error was made during the code edits, then it’s much easier to back out of one refactoring than many

But the term “refactoring” is often used to refer to any reorganization of the codebase, and the test suite needs to be refactored to match

Refactoring Test Cases

Although ideally system-level tests remain unchanged, unit and integration tests need to be refactored to match the code changes, since refactoring can add units, remove units and move functionality between units



Although test cases are code too, typically refactoring operations on test cases are adapted to their special role

Example

Let’s say we have

```
veryLongMethod (...) {
    many lines of code
}
```

and refactor to

```
veryLongMethod (...) {
    shorterMethod(...);
    if ( someCondition(...) ) {
        anotherShorterMethod(..., aMethod(...), ...);
    }
}
```

Which test cases should be changed, added or removed?

Why Refactor?

To introduce a design pattern

To remove code smells

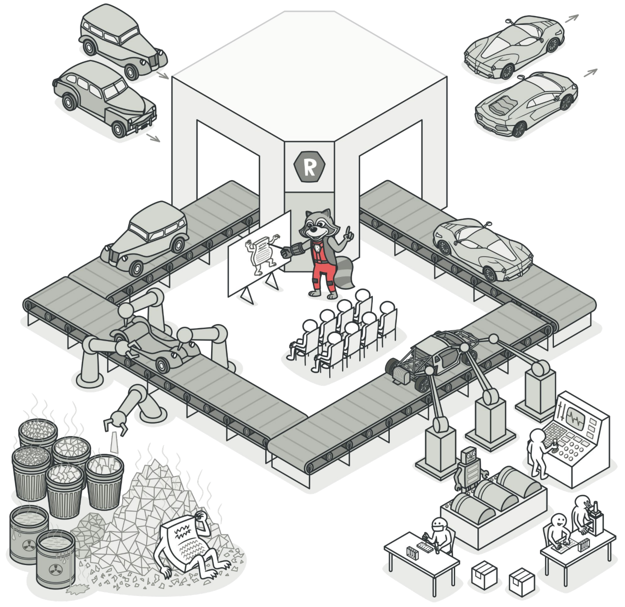
To be compliant with SOLID principles

To make it easier to fix a bug

To make it easier to add a feature

To make the code easier to read

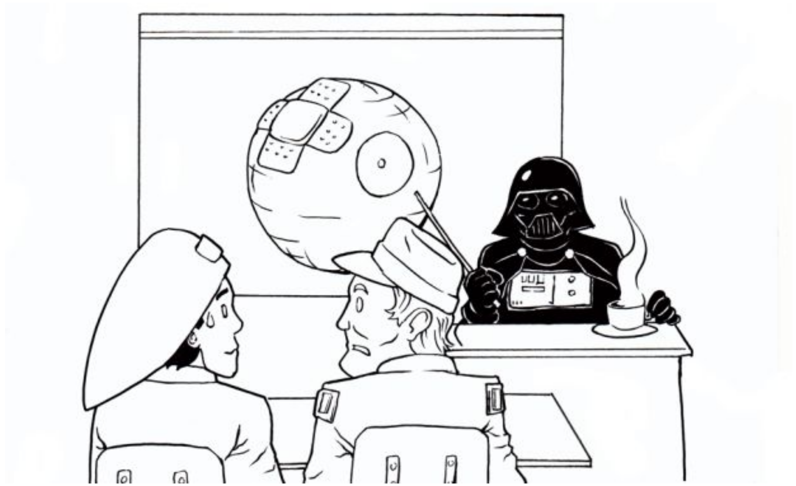
...



Refactoring vs. Other Code Changes

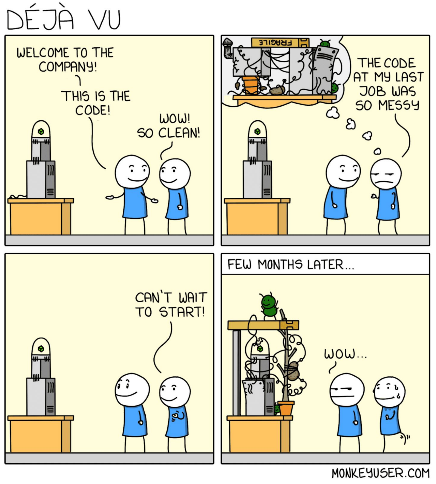
Refactoring should not be included in the same commit as bug-fixes. Why not?

Refactoring should be done in a separate commit before a bug-fix. Why?

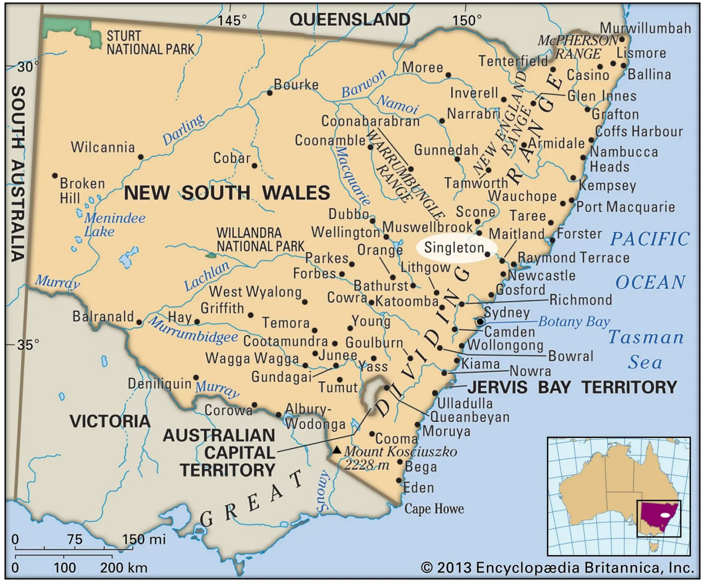


Refactoring should not be included in the same commit a new features. Why not?

Refactoring should be done in a separate commit before a new feature. Why?



Consider Introducing the Singleton Design Pattern



Singleton is a ‘creational’ design pattern that lets you ensure that a class has only one (or at most one) instance, while providing a global access point to this instance

Typically controls access to a shared resource such as database, cache, thread pool, logger, device. Why would we want no more than one?

Other kinds of design patterns (covered later in course) are ‘behavioral’ and ‘structural’ as well as additional ‘creational’ patterns

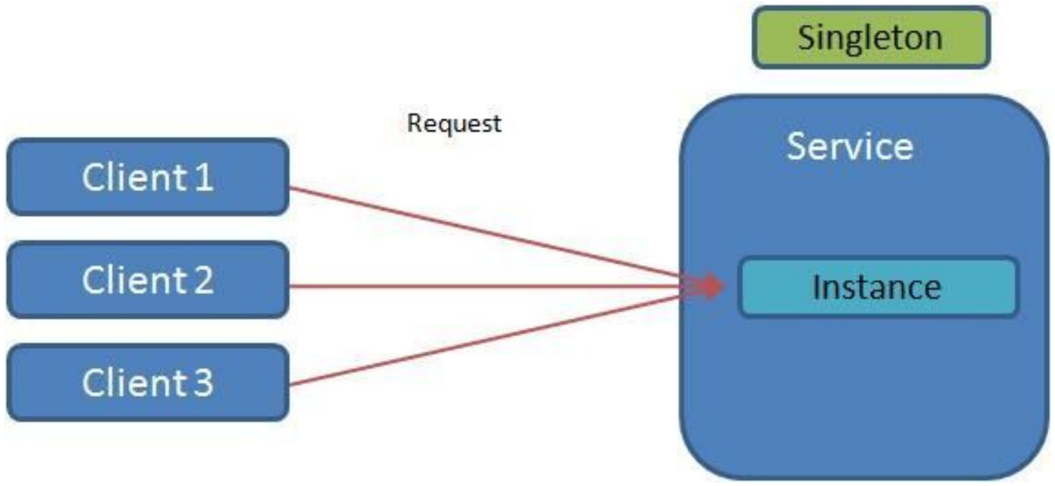
When to Use Singleton

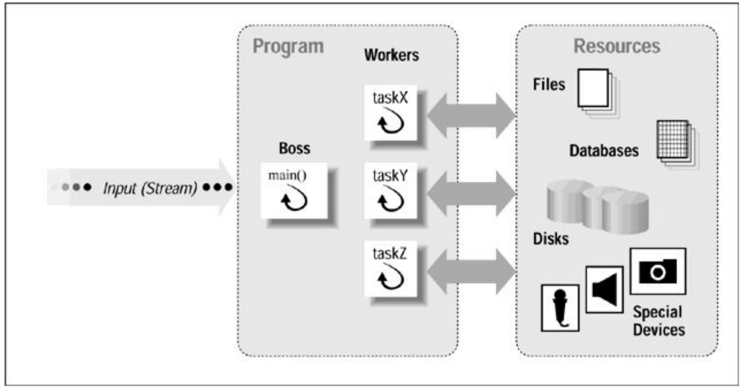
Use when a class in your program should have just a single instance available to all clients; for example, a single database object shared by different parts of the program

The Singleton pattern disables all other means of creating objects of a given class except for the special creation method, which either creates a new object or returns an existing one if it has already been created

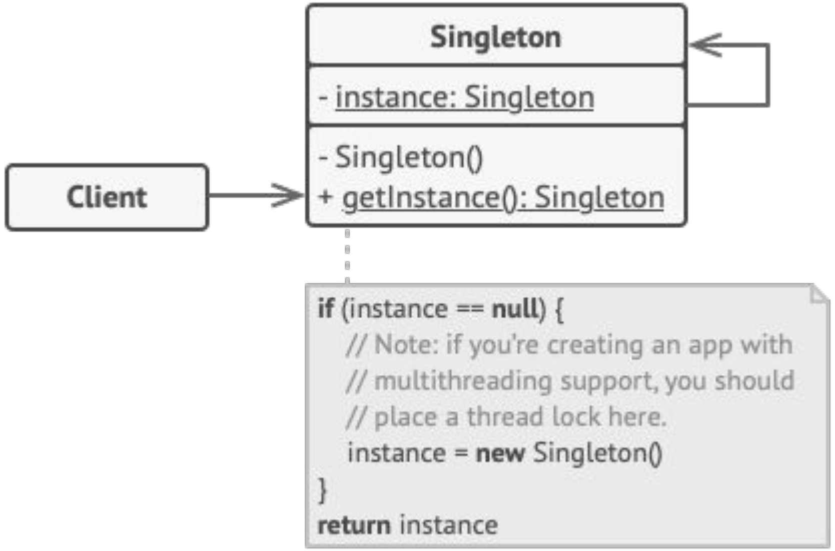
Use the Singleton pattern when you need stricter control over global variables - Singleton guarantees that there’s just one (or at most one) instance and nothing, except for the Singleton class itself, can replace the cached instance

Could be generalized to situations where you want exactly N (or up to N), where N != 1

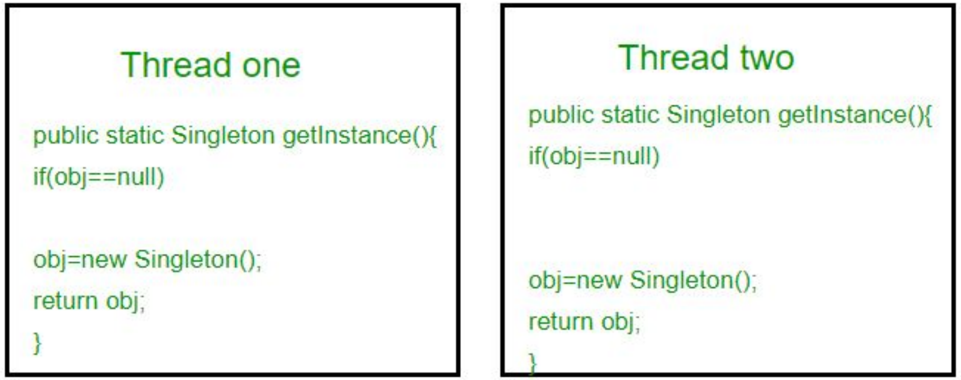

Singleton
Request
Service
Client 1
Client 2
Client 3
Instance


Program
Workers
Resources
Files
taskX
Boss
main()
taskY
Databases
Input (Stream)
Disks
taskZ
Special Devices

Singleton and Thread Safety



```
if (instance == null) {
   // Note: if you're creating an app with
   // multithreading support, you should
   // place a thread lock here.
   instance = new Singleton()
}
return instance
```


Thread one

```
public static Singleton getInstance(){
if(obj==null)

obj=new Singleton();
return obj;
}
```

Thread two

```
public static Singleton getInstance(){
if(obj==null)


obj=new Singleton();
return obj;
}
```

How to Refactor for Singletons

1. Add a private static field to the class for storing the singleton instance
2. Declare a public static creation method for getting the singleton instance
3. Implement "lazy initialization" inside the static method: it should create a new object on its first call and put it into the static field, and always return that instance on all subsequent calls
4. Make the constructor private so the static method will still be able to call the constructor, but other code cannot
5. Go through client code and replace all direct calls to the singleton's constructor with calls to its static creation method

Tools for Refactoring

Common refactoring operations are automated by some IDEs (e.g., IntelliJ IDEA, VS Code) and IDE plugins (e.g., see Eclipse Marketplace)

There are many refactoring catalogs and cookbooks

super easy
cookbook for beginners
5-Ingredient Recipes and Essential Techniques to Get You Started in the Kitchen
lisa grant

Simple Refactoring Example: Renaming

Why rename an identifier? The class/method/variable name is not descriptive enough, something new has been introduced requiring existing code to use a more specific name, ...

Conventional find-and-replace editing operations are text-based and could unintentionally change unrelated items with same name (or name subtoken)

```
Server = new Server(path, port, endpoint);

Server.init();

Server.run();
```

```
Server = new WebSocketServer(path, port,
broadcastingServerEndpoint);

Server.validate();

Server.run();
```

Renaming Refactoring

When renaming a field, want to rename all uses and its getter/setter methods

When renaming a method, want to rename all calls and all overridden/implemented methods in subclasses throughout the codebase

When renaming a class, want to rename code that uses the class throughout the codebase -- and possibly variables, inheritors and other parts of the code to align with new name

Automated support within an IDE is particularly valuable, since it already tracks definitions and uses of identifiers

Possibly can also help rename non-code uses in comments and configuration files, but recall the pitfalls of find-and-replace…

Extract Method Refactoring

Problem - You have a code fragment that appears multiple times (and is longer than some minimum threshold. e.g., 3 lines, 5 lines, 10 lines…)

Solution - Move this code to a separate new method and replace the old code with a call to this method

```
void printOwing() {
  printBanner();

  // Print details.

  System.out.println("name:  " + name);
  System.out.println("amount:  " +
getOutstanding());
}
```

```
void printOwing() {

  printBanner();

  printDetails(getOutstanding());

}


void printDetails(double outstanding) {

  System.out.println("name:  " + name);

  System.out.println("amount:  " +
outstanding);

  }
```

Inline Method Refactoring

Problem - When a method body is needed but the method is not, “why do we have this method?” (Only if not redefined in subclasses, then need to keep it)

Solution - Replace calls to the method with the method’s content and delete the method itself

```
class PizzaDelivery {
  // …
  int getRating() {
    return moreThanFiveLateDeliveries() ?
2 : 1;
  }
  boolean moreThanFiveLateDeliveries() {
    return numberOfLateDeliveries > 5;
  }
}
```

```
class PizzaDelivery {
  // …
  int getRating() {
    return numberOfLateDeliveries >  5 ? 2 :
1;
  }
}
```

Extract Variable Refactoring

Problem - You have an expression that's hard to understand, particularly if multi-line

Solution - Place the result of the expression or its parts in separate variables that are self-explanatory (may be step on way to extract method)

```
void renderBanner() {
  if ((platform.toUpperCase().indexOf( "MAC")
> -1) &&
       (browser.toUpperCase().indexOf( "IE")
> -1) &&
        wasInitialized() && resize >  0 )
  {
    // do something
  }
}
```

```
void renderBanner() {

  final boolean isMacOs =
platform.toUpperCase().indexOf( "MAC") > -1;

  final boolean isIE =
browser.toUpperCase().indexOf( "IE") > -1;

  final boolean wasResized = resize >  0;

  if (isMacOs && isIE && wasInitialized() &&
wasResized) {
    // do something
  }
}
```

Replace Method with Method Object Refactoring

Problem - You have a long method in which the local variables are so intertwined that you can't apply *Extract Method*

Solution - Transform the method into a separate class so that the local variables become fields of the class. Then you can split the method into several methods within the same class

```
class Order {
  // …
  public double price() {
    double primaryBasePrice;
    double secondaryBasePrice;
    double tertiaryBasePrice;
    // Perform long computation.
  }
}
```

```
class Order {
  // …
  public double price() {
    return new
PriceCalculator(this).compute();
  }
}

class PriceCalculator {
  private double primaryBasePrice;
  private double secondaryBasePrice;
  private double tertiaryBasePrice;

  public PriceCalculator(Order order) {
    // Copy relevant information from the
    // order object.
  }

  public double compute() {
    // Perform long computation.
  }
}
```

Composing Methods Refactorings

Inline Temp
Problem: You have a temporary variable that's assigned the result of a simple expression and nothing more.
Solution: Replace the references to the variable with the expression itself.

Replace Temp with Query
Problem: You place the result of an expression in a local variable for later use in your code.
Solution: Move the entire expression to a separate method and return the result from it. Query the method instead of using a variable. Incorporate the new method in other methods, if necessary.

Split Temporary Variable
Problem: You have a local variable that's used to store various intermediate values inside a method.
Solution: Use different variables for different values. Each variable should be responsible for only one specific thing.

Remove Assignments to Parameters
Problem: Some value is assigned to a parameter inside method's body.
Solution: Use a local variable instead of a parameter.

Substitute Algorithm
Problem: So you want to replace an existing algorithm with a new one?
Solution: Replace the body of the method that implements the algorithm with a new algorithm.

Categories of Refactoring Operations

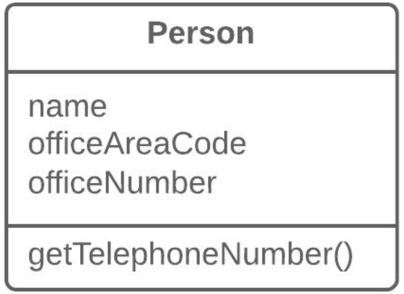
Composing Methods - Excessively long methods are the root of all evil. Methods that are hard to understand are hard to change. Streamline methods, remove code duplication, and pave the way for future improvements

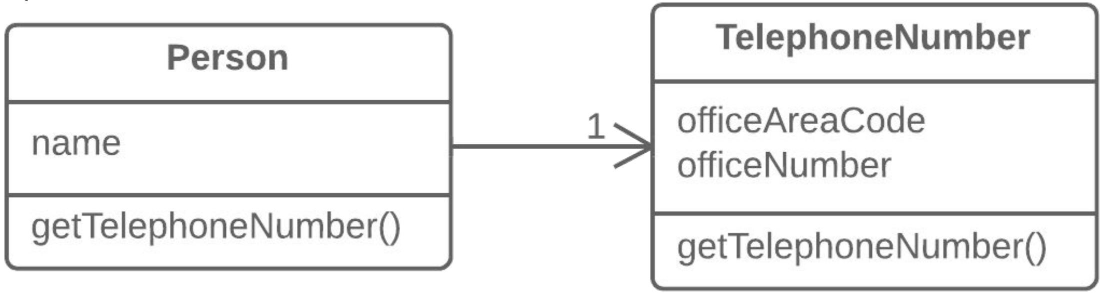
Moving Features between Objects - Even if you have distributed functionality among different classes in a less-than-perfect way, there's still hope! Safely move functionality between classes, create new classes, and hide implementation details from public access

More Refactorings: Extract Class

Problem - When one class does the work of two or more, changing one part may break other parts. Often the class started out fine (obeying SRP), but then grew

Solution - Create a new class and place the fields and methods responsible for the relevant functionality there (don't overdo it, the Inline Class refactoring operation does the opposite)

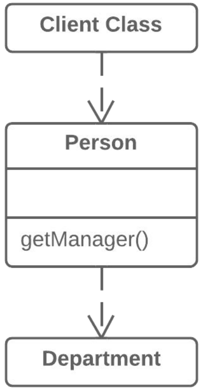


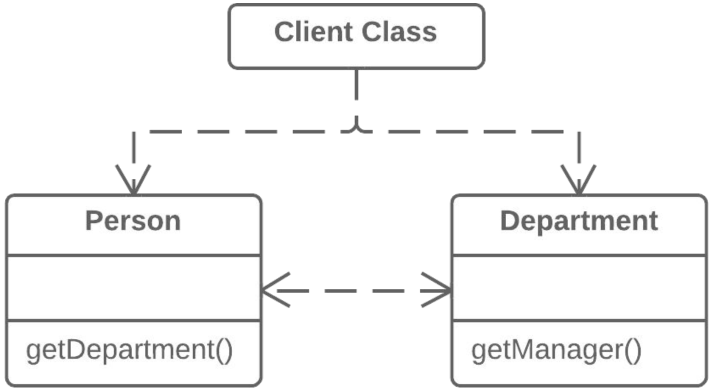


Remove Middle Man

Problem - A class has too many (any?) methods that simply delegate to other objects, every time a new method is added to the delegatee a corresponding new method needs to be added to the delegator

Solution - Delete these methods and force the client to call the end methods directly





Moving Features between Objects

Hide Delegate
Problem: The client gets object B from a field or method of object A. Then the client calls a method of object B.
Solution: Create a new method in class A that delegates the call to object B. Now the client doesn't know about, or depend on, class B.

Move Field
Problem: A field is used more in another class than in its own class.
Solution: Create a field in a new class and redirect all users of the old field to it.

Inline Class
Problem: A class does almost nothing and isn't responsible for anything, and no additional responsibilities are planned for it.
Solution: Move all features from the class to another one.

Move Method
Problem: A method is used more in another class than in its own class.
Solution: Create a new method in the class that uses the method the most, then move code from the old method to that class. Turn the code of the original method into a reference to the new method in the other class or remove it entirely.

Moving Features between Objects

Introduce Foreign Method
Problem: A utility class doesn’t contain the method that you need and you can’t add the method to the class.
Solution: Add the method to a client class and pass an object of the utility class to it as an argument.

Introduce Local Extension
Problem: A utility class doesn’t contain some methods that you need. But you can’t add these methods to the class.
Solution: Create a new class containing the methods and make it either the child or wrapper of the utility class.

Even More Categories of Refactoring Operations

Organizing Data - Reorganize data handling. Untangle class associations, making classes more portable and reusable

Simplifying Conditional Expressions - Logic in conditionals tend to get more and more complicated over time, techniques for fighting back

Simplifying Method Calls - Make method calls simpler and easier to understand, simplifying the interfaces for interaction between classes

Dealing with Generalization - Moving functionality along the class inheritance hierarchy, creating new classes and interfaces, replacing inheritance with delegation and vice versa

Each of these categories include several refactoring operations, e.g., Organizing Data has 15 !

Upcoming Assignments

First Individual Assessment
due 11:59pm November 4, that's this Friday!

Second Iteration due November 28

Second Iteration demo due December 5

Next Class

Continuous Integration

github actions workflow demo

Ask Me Anything