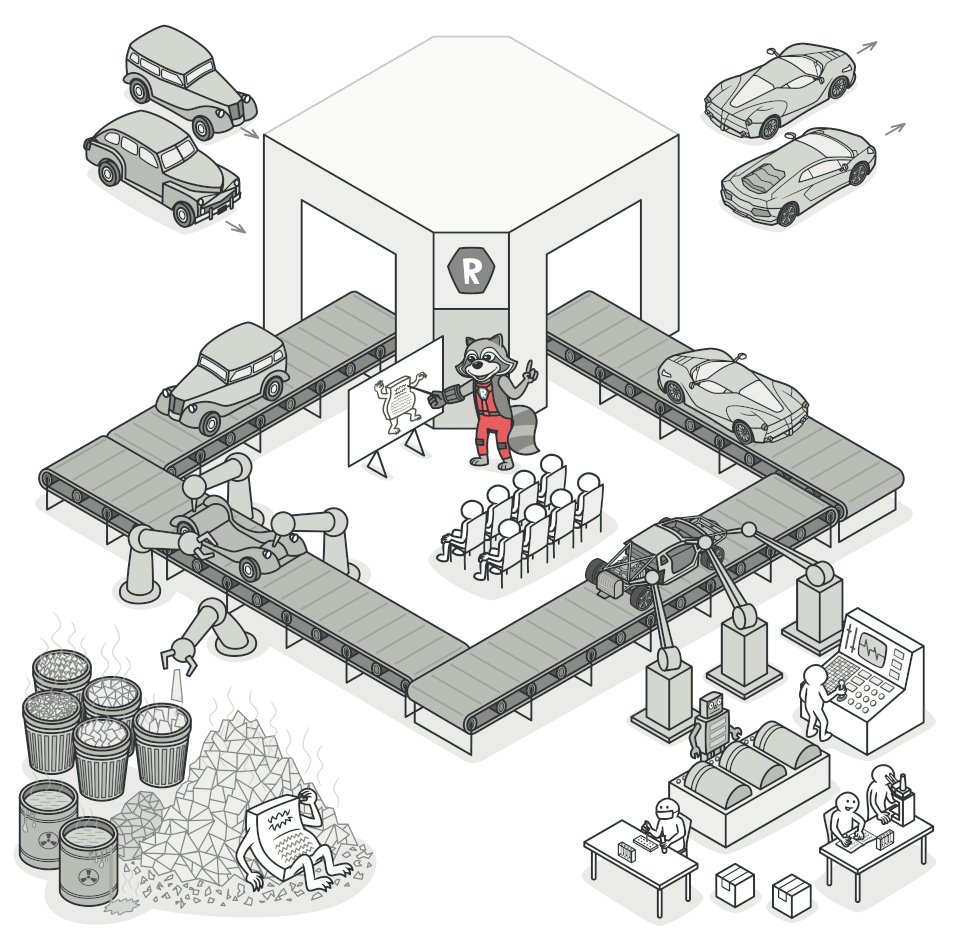
**Q 1: what is Refactoring**



# Refactoring

Refactoring is a systematic process of improving codewithout creating new functionality that can transforma mess into clean code and simple design.

### **Dirty Code**

Dirty code is result of inexperiencemultiplied by tight deadlines,mismanagement, and nastyshortcuts taken during thedevelopment process.

**Clean code**

The main purpose of refactoring is to fight technical debt. It transforms a mess into clean code and simple design.

Nice! But what’s clean code, anyway? Here are some of its features:

**Clean code is obvious for other programmers.**

And I’m not talking about super sophisticated algorithms. Poor variable naming, bloated classes and methods, magic numbers -you name it- all of that makes code sloppy and difficult to grasp.

**Clean code doesn’t contain duplication.**

Each time you have to make a change in a duplicate code, you have to remember to make the same change to every instance. This increases the cognitive load and slows down the progress.

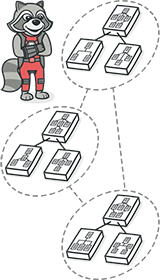
**Clean code contains a minimal number of classes and other moving parts.**

Less code is less stuff to keep in your head. Less code is less maintenance. Less code is fewer bugs. Code is liability, keep it short and simple.

**Clean code passes all tests.**

You know your code is dirty when only 95% of your tests passed. You know you’re screwed when your test coverage is 0%.

**Clean code is easier and cheaper to maintain!**



**Q 2: Moving Features between Objects**

Even if you have distributed functionality among different classes in a less-than-perfect way, there’s still hope.

These refactoring techniques show how to safely move functionality between classes, create new classes, and hide implementation details from public access.

[**Move Method**](https://refactoring.guru/move-method)

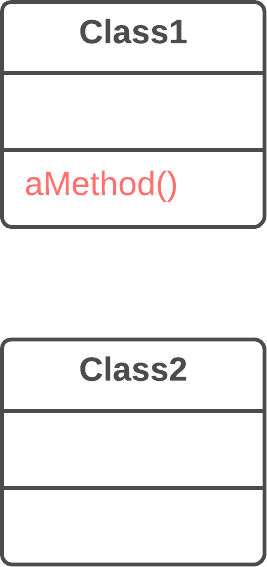
**Problem:** A method is used more in another class than in its own class.

**Solution:** Create a new method in the class that uses the method the most, then move code from the old method to there. Turn the code of the original method into a reference to the new method in the other class or else remove it entirely.

# Move Method

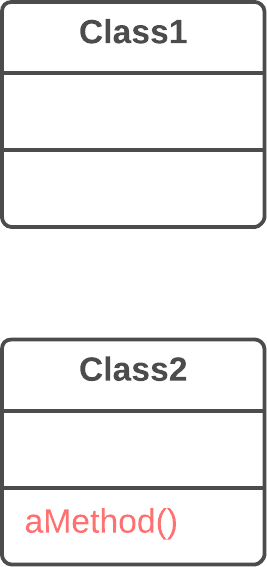
### **Problem**

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 there. Turn the code of the original method into a reference to the new method in the other class or else remove it entirely.



**Why Refactor**

1. You want to move a method to a class that contains most of the data used by the method. This makes **classes more internally coherent**.
2. You want to move a method in order to reduce or eliminate the dependency of the class calling the method on the class in which it’s located. This can be useful if the calling class is already dependent on the class to which you’re planning to move the method. This **reduces dependency between classes**.

**How to Refactor**

1. Verify all features used by the old method in its class. It may be a good idea to move them as well. As a rule, if a feature is used only by the method under consideration, you should certainly move the feature to it. If the feature is used by other methods too, you should move these methods as well. Sometimes it’s much easier to move a large number of methods than to set up relationships between them in different classes.

Make sure that the method isn’t declared in superclasses and subclasses. If this is the case, you will either have to refrain from moving or else implement a kind of polymorphism in the recipient class in order to ensure varying functionality of a method split up among donor classes.

1. Declare the new method in the recipient class. You may want to give a new name for the method that’s more appropriate for it in the new class.
2. Decide how you will refer to the recipient class. You may already have a field or method that returns an appropriate object, but if not, you will need to write a new method or field to store the object of the recipient class.

Now you have a way to refer to the recipient object and a new method in its class. With all this under your belt, you can turn the old method into a reference to the new method.

1. Take a look: can you delete the old method entirely? If so, place a reference to the new method in all places that use the old one.

[**Move Field**](https://refactoring.guru/move-field)

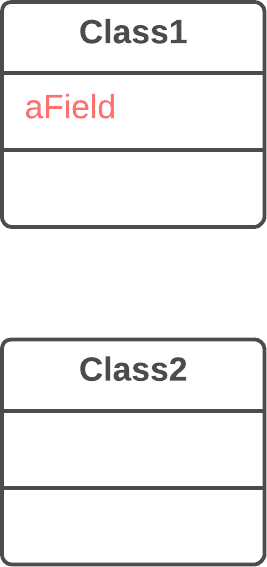
**Problem:** A field is used more in another class than in its own class.

**Solution:** Create a field in a new class and redirect all users of the old field to it.

# Move Field

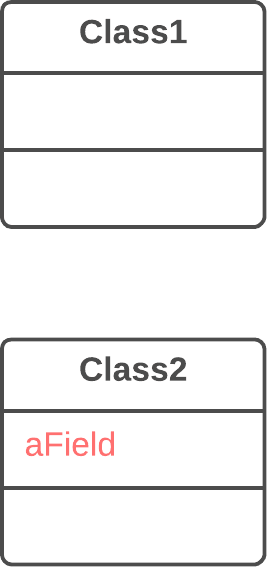
### **Problem**

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.



### **Why Refactor**

Often fields are moved as part of the [**Extract Class**](https://refactoring.guru/extract-class) technique. Deciding which class to leave the field in can be tough. Here is our rule of thumb: **put a field in the same place as the methods that use it** (or else where most of these methods are).

This rule will help in other cases when a field is simply located in the wrong place.

### **How to Refactor**

1. If the field is public, refactoring will be much easier if you make the field private and provide public access methods (for this, you can use [**Encapsulate Field**](https://refactoring.guru/encapsulate-field)).
2. Create the same field with access methods in the recipient class.
3. Decide how you will refer to the recipient class. You may already have a field or method that returns the appropriate object; if not, you will need to write a new method or field to store the object of the recipient class.
4. Replace all references to the old field with appropriate calls to methods in the recipient class. If the field isn’t private, take care of this in the superclass and subclasses.
5. Delete the field in the original class.

[**Extract Class**](https://refactoring.guru/extract-class)

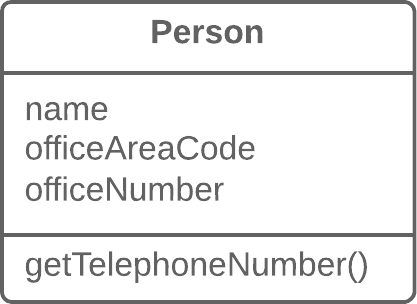
**Problem:** When one class does the work of two, awkwardness results.

**Solution:** Instead, create a new class and place the fields and methods responsible for the relevant functionality in it.

# Extract Class

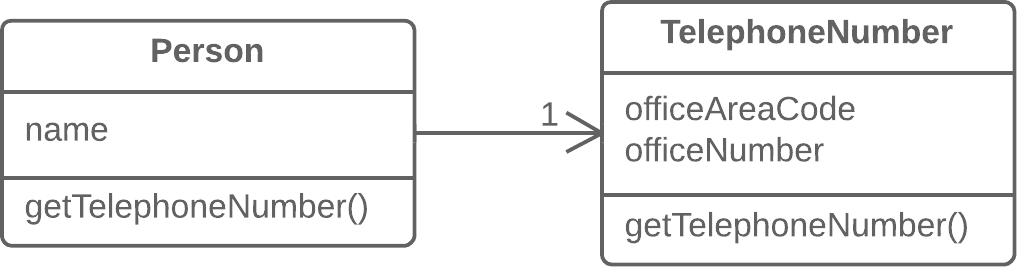
### Problem

When one class does the work of two, awkwardness results.



### Solution

Instead, create a new class and place the fields and methods responsible for the relevant functionality in it.



### **Why Refactor**

Classes always start out clear and easy to understand. They do their job and mind their own business as it were, without butting into the work of other classes. But as the program expands, a method is added and then a field... and eventually, some classes are performing more responsibilities than ever envisioned.

### Benefits

* This refactoring method will help maintain adherence to the Single Responsibility Principle. The code of your classes will be more obvious and understandable.
* Single-responsibility classes are more reliable and tolerant of changes. For example, say that you have a class responsible for ten different things. When you change this class to make it better for one thing, you risk breaking it for the nine others.

### Drawbacks

* If you “overdo it” with this refactoring technique, you will have to resort to [**Inline Class**](https://refactoring.guru/inline-class).

### How to Refactor

Before starting, decide on how exactly you want to split up the responsibilities of the class.

1. Create a new class to contain the relevant functionality.
2. Create a relationship between the old class and the new one. Optimally, this relationship is unidirectional; this allows reusing the second class without any issues. Nonetheless, if you think that a two-way relationship is necessary, this can always be set up.
3. Use [**Move Field**](https://refactoring.guru/move-field) and [**Move Method**](https://refactoring.guru/move-method) for each field and method that you have decided to move to the new class. For methods, start with private ones in order to reduce the risk of making a large number of errors. Try to relocate just a little bit at a time and test the results after each move, in order to avoid a pileup of error-fixing at the very end.

After you’re done moving, take one more look at the resulting classes. An old class with changed responsibilities may be renamed for increased clarity. Check again to see whether you can get rid of two-way class relationships, if any are present.

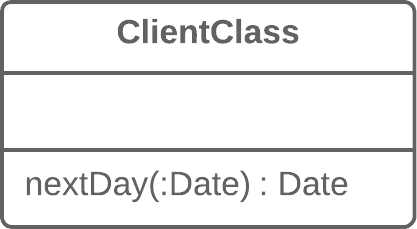
1. Also give thought to accessibility to the new class from the outside. You can hide the class from the client entirely by making it private, managing it via the fields from the old class. Alternatively, you can make it a public one by allowing the client to change values directly. Your decision here depends on how safe it’s for the behavior of the old class when unexpected direct changes are made to the values in the new class.

**Q 3: Introduce Local Extension**

**Introduce Local Extension**

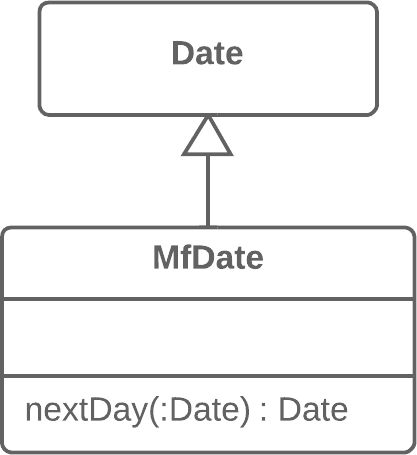
### **Problem**

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.



### **Why Refactor**

The class that you’re using doesn’t have the methods that you need. What’s worse, you can’t add these methods (because the classes are in a third-party library, for example). There are two ways out:

* Create a **subclass** from the relevant class, containing the methods and inheriting everything else from the parent class. This way is easier but is sometimes blocked by the utility class itself (due to final).
* Create a **wrapper** class that contains all the new methods and elsewhere will delegate to the related object from the utility class. This method is more work since you need not only code to maintain the relationship between the wrapper and utility object, but also a large number of simple delegating methods in order to emulate the public interface of the utility class.

### Benefits

* By moving additional methods to a separate extension class (wrapper or subclass), you avoid gumming up client classes with code that doesn’t fit. Program components are more coherent and are more reusable.

### How to Refactor

1. Create a new extension class:
   * Option A: Make it a child of the utility class.
   * Option B: If you have decided to make a wrapper, create a field in it for storing the utility class object to which delegation will be made. When using this option, you will need to also create methods that repeat the public methods of the utility class and contain simple delegation to the methods of the utility object.
2. Create a constructor that uses the parameters of the constructor of the utility class.
3. Also create an alternative “converting” constructor that takes only the object of the original class in its parameters. This will help to substitute the extension for the objects of the original class.
4. Create new extended methods in the class. Move foreign methods from other classes to this class or else delete the foreign methods if their functionality is already present in the extension.
5. Replace use of the utility class with the new extension class in places where its functionality is needed.

### **Q 4:** [**Dealing with Generalization**](https://refactoring.guru/refactoring/techniques/dealing-with-generalization)

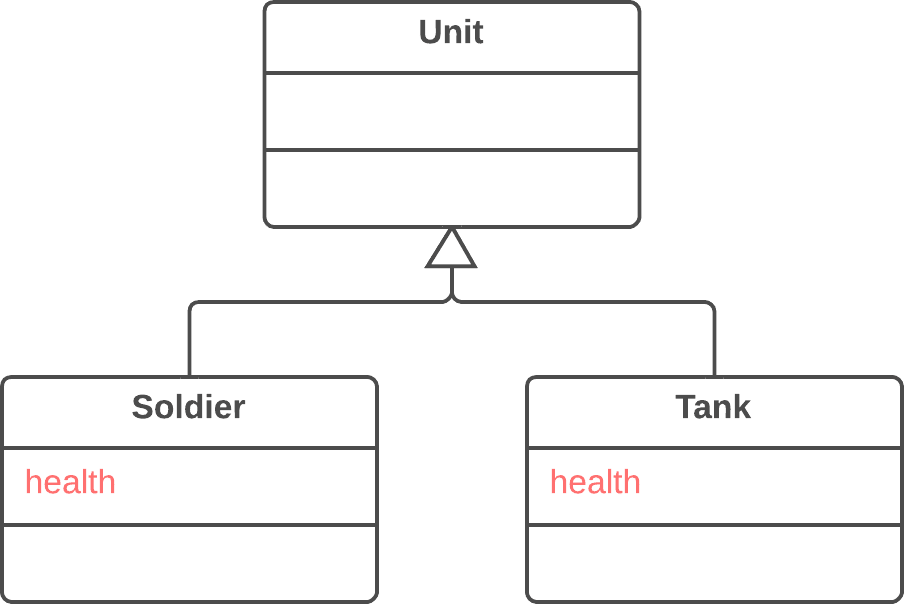
Abstraction has its own group of refactoring techniques, primarily associated with moving functionality along the class inheritance hierarchy, creating new classes and interfaces, and replacing inheritance with delegation and vice versa.

* [**Pull Up Field**](https://refactoring.guru/pull-up-field)
* [**Pull Up Method**](https://refactoring.guru/pull-up-method)
* [**Push Down Field**](https://refactoring.guru/push-down-field)
* [**Push Down Method**](https://refactoring.guru/push-down-method)
* [**Extract Subclass**](https://refactoring.guru/extract-subclass)
* [**Extract Superclass**](https://refactoring.guru/extract-superclass)

# Pull Up Field

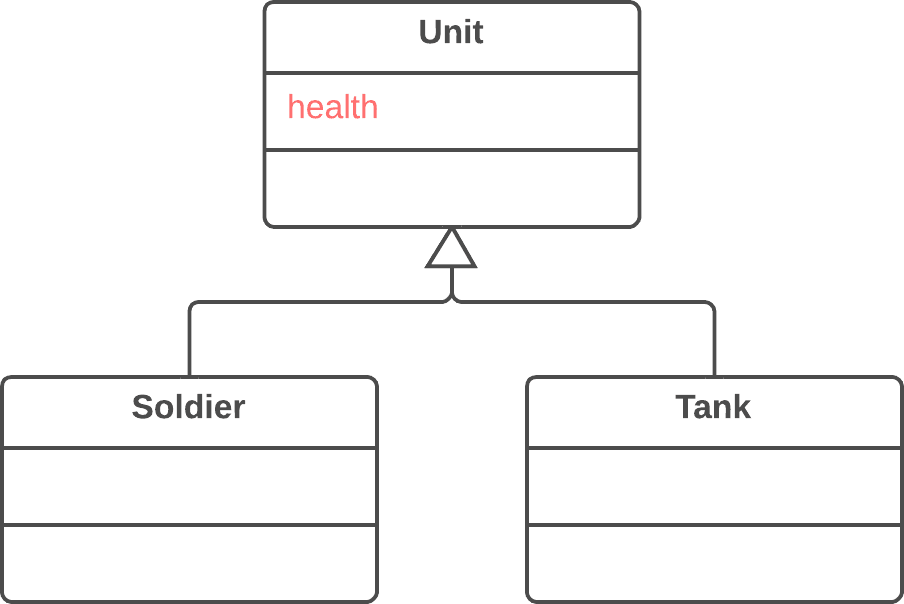
### Problem

Two classes have the same field.



### Solution

Remove the field from subclasses and move it to the superclass.



### Why Refactor

Subclasses grew and developed separately, causing identical (or nearly identical) fields and methods to appear.

### Benefits

* Eliminates duplication of fields in subclasses.
* Eases subsequent relocation of duplicate methods, if they exist, from subclasses to a superclass.

### How to Refactor

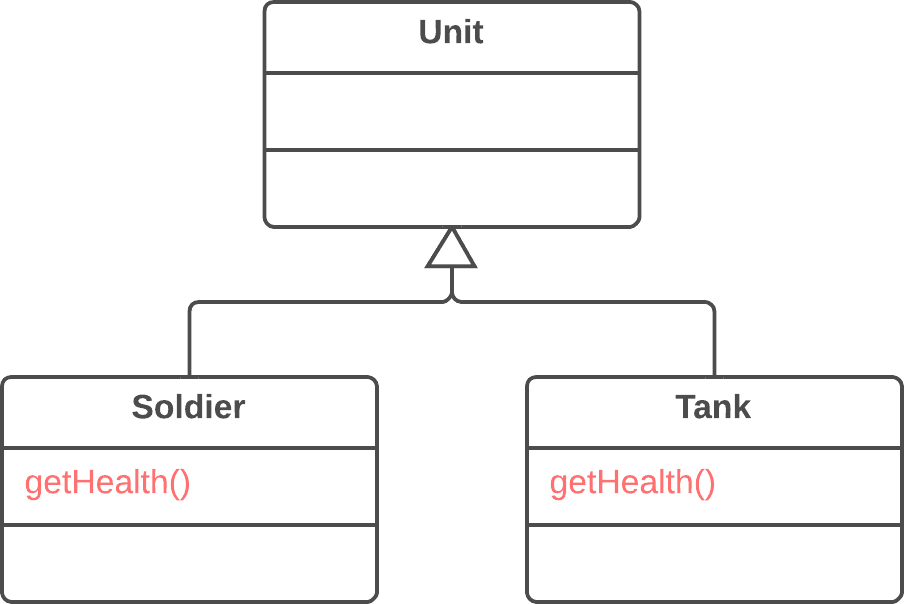
1. Make sure that the fields are used for the same needs in subclasses.
2. If the fields have different names, give them the same name and replace all references to the fields in existing code.
3. Create a field with the same name in the superclass. Note that if the fields were private, the superclass field should be protected.
4. Remove the fields from the subclasses.
5. You may want to consider using [**Self Encapsulate Field**](https://refactoring.guru/self-encapsulate-field) for the new field, in order to hide it behind access methods.

# Pull Up Method

### Problem

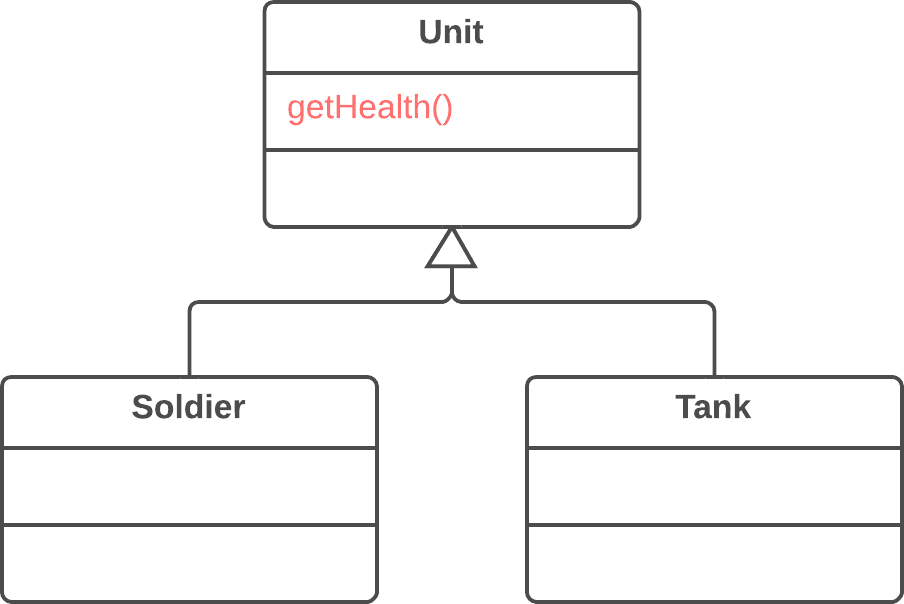
Your subclasses have methods that perform similar work.

.



### Solution

Make the methods identical and then move them to the relevant superclass



### Why Refactor

Subclasses grew and developed independently of one another, causing identical (or nearly identical) fields and methods.

### Benefits

* Gets rid of duplicate code. If you need to make changes to a method, it’s better to do so in a single place than have to search for all duplicates of the method in subclasses.
* This refactoring technique can also be used if, for some reason, a subclass redefines a superclass method but performs what’s essentially the same work.

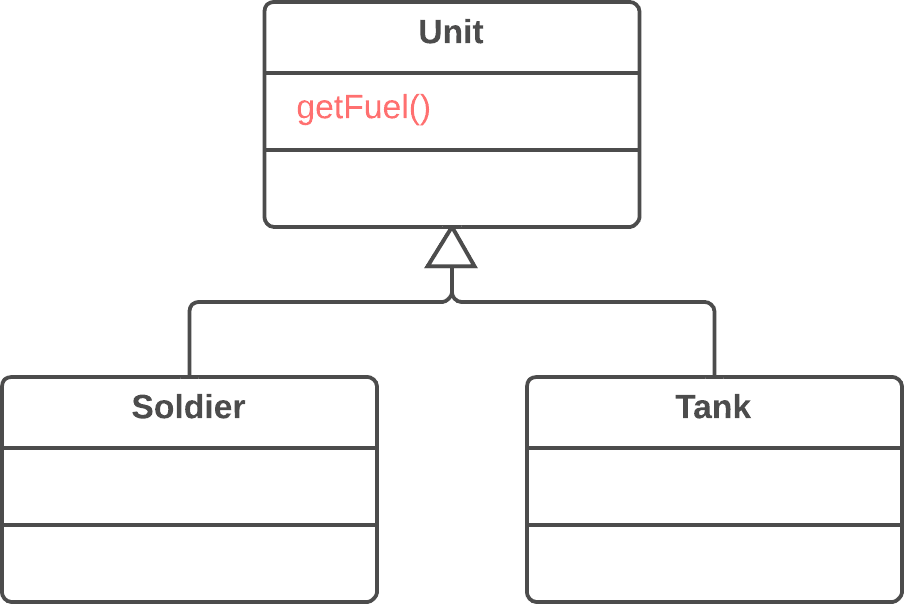
### How to Refactor

1. Investigate similar methods in superclasses. If they aren’t identical, format them to match each other.
2. If methods use a different set of parameters, put the parameters in the form that you want to see in the superclass.
3. Copy the method to the superclass. Here you may find that the method code uses fields and methods that exist only in subclasses and therefore aren’t available in the superclass. To solve this, you can:
   * For fields: use either [**Pull Up Field**](https://refactoring.guru/pull-up-field) or Self-[**Encapsulate Field**](https://refactoring.guru/encapsulate-field) to create getters and setters in subclasses; then declare these getters abstractly in the superclass.
   * For methods: use either [**Pull Up Method**](https://refactoring.guru/pull-up-method) or declare abstract methods for them in the superclass (note that your class will become abstract if it wasn’t previously).
4. Remove the methods from the subclasses.
5. Check the locations in which the method is called. In some places you may be able to replace use of a subclass with the superclass.

**Push Down Method**

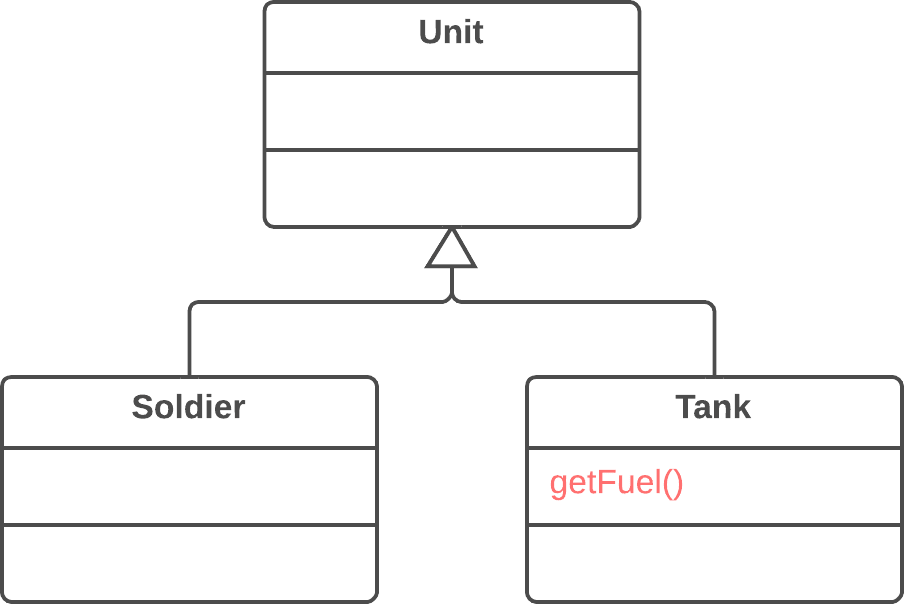
### Problem

Is behavior implemented in a superclass used by only one (or a few) subclasses?



### Solution

Move this behavior to the subclasses.



### Why Refactor

At first a certain method was meant to be universal for all classes but in reality is used in only one subclass. This situation can occur when planned features fail to materialize.

Such situations can also occur after partial extraction (or removal) of functionality from a class hierarchy, leaving a method that’s used in only one subclass.

If you see that a method is needed by more than one subclass, but not all of them, it may be useful to create an intermediate subclass and move the method to it. This allows avoiding the code duplication that would result from pushing a method down to all subclasses.

### Benefits

* Improves class coherence. A method is located where you expect to see it.

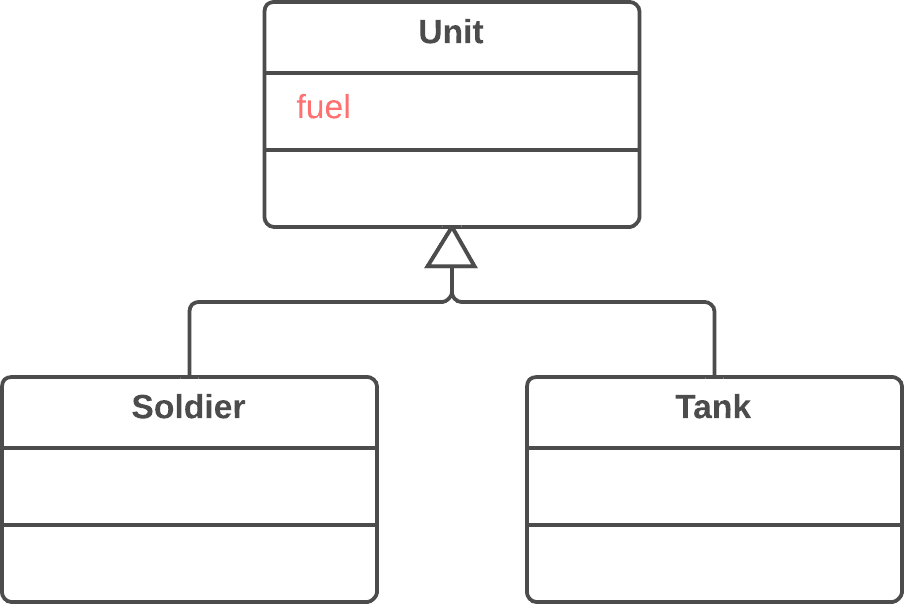
### How to Refactor

1. Declare the method in a subclass and copy its code from the superclass.
2. Remove the method from the superclass.
3. Find all places where the method is used and verify that it’s called from the necessary subclass.

# Push Down Field

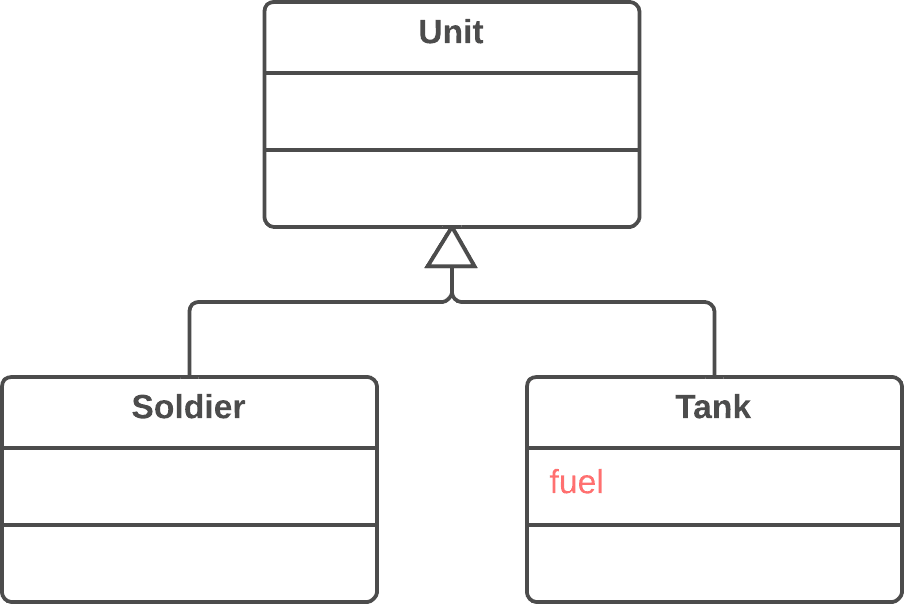
### Problem

Is a field used only in a few subclasses?



### Solution

Move the field to these subclasses.



### Why Refactor

Although it was planned to use a field universally for all classes, in reality the field is used only in some subclasses. This situation can occur when planned features fail to pan out, for example.

This can also occur due to extraction (or removal) of part of the functionality of class hierarchies.

### Benefits

* Improves internal class coherency. A field is located where it’s actually used.
* When moving to several subclasses simultaneously, you can develop the fields independently of each other. This does create code duplication, yes, so push down fields only when you really do intend to use the fields in different ways.

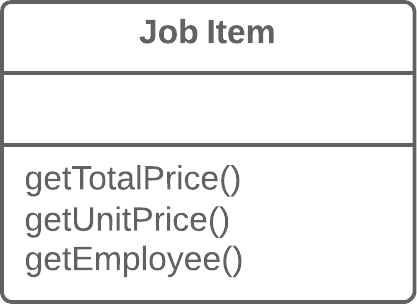
### How to Refactor

1. Declare a field in all the necessary subclasses.
2. Remove the field from the superclass.

# Extract Subclass

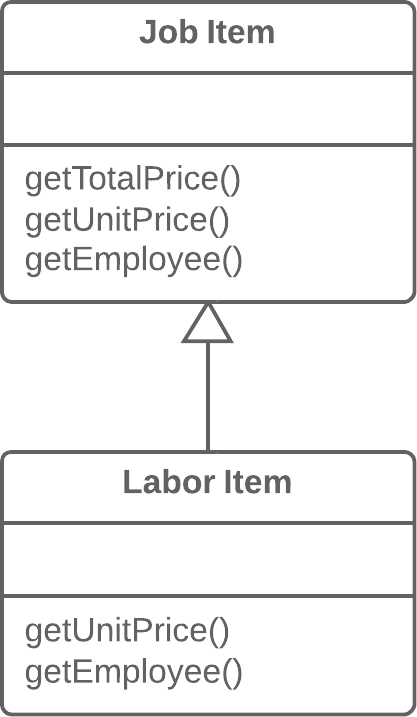
### Problem

A class has features that are used only in certain cases.



### Solution

Create a subclass and use it in these cases.



### Why Refactor

Your main class has methods and fields for implementing a certain rare use case for the class. While the case is rare, the class is responsible for it and it would be wrong to move all the associated fields and methods to an entirely separate class. But they could be moved to a subclass, which is just what we’ll do with the help of this refactoring technique.

### Benefits

* Creates a subclass quickly and easily.
* You can create several separate subclasses if your main class is currently implementing more than one such special case.

### Drawbacks

* Despite its seeming simplicity, Inheritance can lead to a dead end if you have to separate several different class hierarchies. If, for example, you had the class Dogs with different behavior depending on the size and fur of dogs, you could tease out two hierarchies:
  + by size: Large, Medium and Small
  + by fur: Smooth and Shaggy

And everything would seem well, except that problems will crop up as soon as you need to create a dog that’s both Large and Smooth, since you can create an object from one class only. That said, you can avoid this problem by using Compose instead of Inherit (see the [**Strategy**](https://refactoring.guru/design-patterns/strategy) pattern). In other words, the Dog class will have two component fields, size and fur. You will plug in component objects from the necessary classes into these fields. So you can create a Dog that has LargeSize and ShaggyFur.

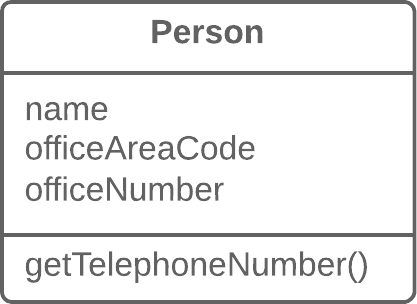
### How to Refactor

1. Create a new subclass from the class of interest.
2. If you need additional data to create objects from a subclass, create a constructor and add the necessary parameters to it. Don’t forget to call the constructor’s parent implementation.
3. Find all calls to the constructor of the parent class. When the functionality of a subclass is necessary, replace the parent constructor with the subclass constructor.
4. Move the necessary methods and fields from the parent class to the subclass. Do this via [**Push Down Method**](https://refactoring.guru/push-down-method) and [**Push Down Field**](https://refactoring.guru/push-down-field). It’s simpler to start by moving the methods first. This way, the fields remain accessible throughout the whole process: from the parent class prior to the move, and from the subclass itself after the move is complete.
5. After the subclass is ready, find all the old fields that controlled the choice of functionality. Delete these fields by using polymorphism to replace all the operators in which the fields had been used. A simple example: in the Car class, you had the field isElectricCar and, depending on it, in the refuel() method the car is either fueled up with gas or charged with electricity. Post-refactoring, the isElectricCar field is removed and the Car and ElectricCar classes will have their own implementations of the refuel() method.

# Extract Class

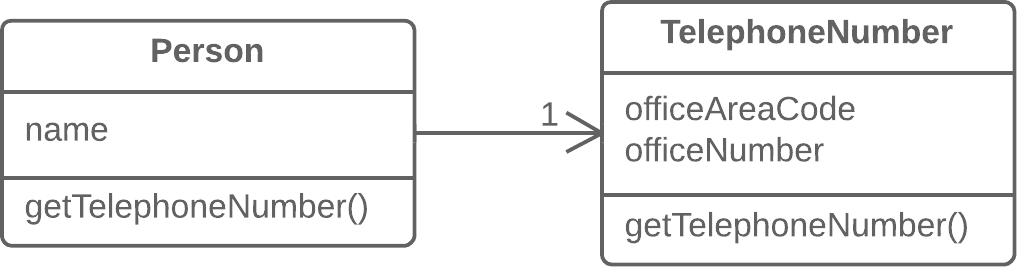
### Problem

When one class does the work of two, awkwardness results.



### Solution

Instead, create a new class and place the fields and methods responsible for the relevant functionality in it.



### Why Refactor

Classes always start out clear and easy to understand. They do their job and mind their own business as it were, without butting into the work of other classes. But as the program expands, a method is added and then a field... and eventually, some classes are performing more responsibilities than ever envisioned.

### Benefits

* This refactoring method will help maintain adherence to the Single Responsibility Principle. The code of your classes will be more obvious and understandable.
* Single-responsibility classes are more reliable and tolerant of changes. For example, say that you have a class responsible for ten different things. When you change this class to make it better for one thing, you risk breaking it for the nine others.

### Drawbacks

* If you “overdo it” with this refactoring technique, you will have to resort to [**Inline Class**](https://refactoring.guru/inline-class).

### How to Refactor

Before starting, decide on how exactly you want to split up the responsibilities of the class.

1. Create a new class to contain the relevant functionality.
2. Create a relationship between the old class and the new one. Optimally, this relationship is unidirectional; this allows reusing the second class without any issues. Nonetheless, if you think that a two-way relationship is necessary, this can always be set up.
3. Use [**Move Field**](https://refactoring.guru/move-field) and [**Move Method**](https://refactoring.guru/move-method) for each field and method that you have decided to move to the new class. For methods, start with private ones in order to reduce the risk of making a large number of errors. Try to relocate just a little bit at a time and test the results after each move, in order to avoid a pileup of error-fixing at the very end.

After you’re done moving, take one more look at the resulting classes. An old class with changed responsibilities may be renamed for increased clarity. Check again to see whether you can get rid of two-way class relationships, if any are present.

1. Also give thought to accessibility to the new class from the outside. You can hide the class from the client entirely by making it private, managing it via the fields from the old class. Alternatively, you can make it a public one by allowing the client to change values directly. Your decision here depends on how safe it’s for the behavior of the old class when unexpected direct changes are made to the values in the new class.

**Q 5:** **Composing Methods**

Much of refactoring is devoted to correctly composing methods. In most cases, excessively long methods are the root of all evil. The vagaries of code inside these methods conceal the execution logic and make the method extremely hard to understand—and even harder to change.

The refactoring techniques in this group streamline methods, remove code duplication, and pave the way for future improvements.

[**Extract Method**](https://refactoring.guru/extract-method)

**Problem:** You have a code fragment that can be grouped together.

**Solution:** Move this code to a separate new method (or function) and replace the old code with a call to the method.

[**Inline Method**](https://refactoring.guru/inline-method)

**Problem:** When a method body is more obvious than the method itself, use this technique.

**Solution:** Replace calls to the method with the method’s content and delete the method itself.

[**Extract Variable**](https://refactoring.guru/extract-variable)

**Problem:** You have an expression that’s hard to understand.

**Solution:** Place the result of the expression or its parts in separate variables that are self-explanatory.

[**Inline Temp**](https://refactoring.guru/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**](https://refactoring.guru/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**](https://refactoring.guru/split-temporary-variable)

**Problem:** You have a local variable that’s used to store various intermediate values inside a method (except for cycle variables).

**Solution:** Use different variables for different values. Each variable should be responsible for only one particular thing.

[**Remove Assignments to Parameters**](https://refactoring.guru/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.

[**Replace Method with Method Object**](https://refactoring.guru/replace-method-with-method-object)

**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.

[**Substitute Algorithm**](https://refactoring.guru/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.

# Q 6: Replace Method with Method Object

### Problem

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

**class** **Order** {

// ...

**public** **double** price() {

**double** primaryBasePrice;

**double** secondaryBasePrice;

**double** tertiaryBasePrice;

// Perform long computation.

}

}

**class** **Order** {

// ...

**public** **double** price() {

**return** **new** PriceCalculator(**this**).compute();

}

}

### 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** **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.

}

}

### Why Refactor

A method is too long and you can’t separate it due to tangled masses of local variables that are hard to isolate from each other.

The first step is to isolate the entire method into a separate class and turn its local variables into fields of the class.

Firstly, this allows isolating the problem at the class level. Secondly, it paves the way for splitting a large and unwieldy method into smaller ones that wouldn’t fit with the purpose of the original class anyway.

### Benefits

* Isolating a long method in its own class allows stopping a method from ballooning in size. This also allows splitting it into submethods within the class, without polluting the original class with utility methods.

### Drawbacks

* Another class is added, increasing the overall complexity of the program.

### How to Refactor

1. Create a new class. Name it based on the purpose of the method that you’re refactoring.
2. In the new class, create a private field for storing a reference to an instance of the class in which the method was previously located. It could be used to get some required data from the original class if needed.
3. Create a separate private field for each local variable of the method.
4. Create a constructor that accepts as parameters the values of all local variables of the method and also initializes the corresponding private fields.
5. Declare the main method and copy the code of the original method to it, replacing the local variables with private fields.
6. Replace the body of the original method in the original class by creating a method object and calling its main method

**Q 7:Simplifying Method Calls**

These techniques make method calls simpler and easier to understand. This, in turn, simplifies the interfaces for interaction between classes.

[**Rename Method**](https://refactoring.guru/rename-method)

**Problem:** The name of a method doesn’t explain what the method does.

**Solution:** Rename the method.

[**Add Parameter**](https://refactoring.guru/add-parameter)

**Problem:** A method doesn’t have enough data to perform certain actions.

**Solution:** Create a new parameter to pass the necessary data.

[**Remove Parameter**](https://refactoring.guru/remove-parameter)

**Problem:** A parameter isn’t used in the body of a method.

**Solution:** Remove the unused parameter.

[**Separate Query from Modifier**](https://refactoring.guru/separate-query-from-modifier)

**Problem:** Do you have a method that returns a value but also changes something inside an object?

**Solution:** Split the method into two separate methods. As you would expect, one of them should return the value and the other one modifies the object.

[**Parameterize Method**](https://refactoring.guru/parameterize-method)

**Problem:** Multiple methods perform similar actions that are different only in their internal values, numbers or operations.

**Solution:** Combine these methods by using a parameter that will pass the necessary special value.

[**Replace Parameter with Explicit Methods**](https://refactoring.guru/replace-parameter-with-explicit-methods)

**Problem:** A method is split into parts, each of which is run depending on the value of a parameter.

**Solution:** Extract the individual parts of the method into their own methods and call them instead of the original method.

[**Preserve Whole Object**](https://refactoring.guru/preserve-whole-object)

**Problem:** You get several values from an object and then pass them as parameters to a method.

**Solution:** Instead, try passing the whole object.

[**Replace Parameter with Method Call**](https://refactoring.guru/replace-parameter-with-method-call)

**Problem:** Calling a query method and passing its results as the parameters of another method, while that method could call the query directly.

**Solution:** Instead of passing the value through a parameter, try placing a query call inside the method body.

[**Introduce Parameter Object**](https://refactoring.guru/introduce-parameter-object)

**Problem:** Your methods contain a repeating group of parameters.

**Solution:** Replace these parameters with an object.

[**Remove Setting Method**](https://refactoring.guru/remove-setting-method)

**Problem:** The value of a field should be set only when it’s created, and not change at any time after that.

**Solution:** So remove methods that set the field’s value.

[**Hide Method**](https://refactoring.guru/hide-method)

**Problem:** A method isn’t used by other classes or is used only inside its own class hierarchy.

**Solution:** Make the method private or protected.

[**Replace Constructor with Factory Method**](https://refactoring.guru/replace-constructor-with-factory-method)

**Problem:** You have a complex constructor that does something more than just setting parameter values in object fields.

**Solution:** Create a factory method and use it to replace constructor calls.

[**Replace Error Code with Exception**](https://refactoring.guru/replace-error-code-with-exception)

**Problem:** A method returns a special value that indicates an error?

**Solution:** Throw an exception instead.

[**Replace Exception with Test**](https://refactoring.guru/replace-exception-with-test)

**Problem:** You throw an exception in a place where a simple test would do the job?

**Solution:** Replace the exception with a condition test.

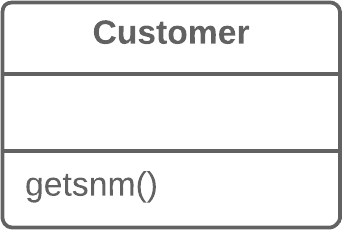
# Q 8: Rename Method

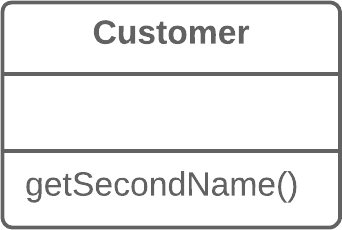
### Problem

The name of a method doesn’t explain what the method does.

### Solution

Rename the method.





### Why Refactor

Perhaps a method was poorly named from the very beginning—for example, someone created the method in a rush and didn’t give proper care to naming it well.

Or perhaps the method was well named at first but as its functionality grew, the method name stopped being a good descriptor.

### Benefits

* Code readability. Try to give the new method a name that reflects what it does. Something like createOrder(), renderCustomerInfo(), etc.

### How to Refactor

1. See whether the method is defined in a superclass or subclass. If so, you must repeat all steps in these classes too.
2. The next method is important for maintaining the functionality of the program during the refactoring process. Create a new method with a new name. Copy the code of the old method to it. Delete all the code in the old method and, instead of it, insert a call for the new method.
3. Find all references to the old method and replace them with references to the new one.
4. Delete the old method. If the old method is part of a public interface, don’t perform this step. Instead, mark the old method as deprecated.

**Q 9: Replace Parameter with Explicit Methods**

### Problem

A method is split into parts, each of which is run depending on the value of a parameter.

**void** **setValue**(**String** name, **int** value) {

**if** (name.equals("height")) {

height = value;

**return**;

}

**if** (name.equals("width")) {

width = value;

**return**;

}

Assert.shouldNeverReachHere();

}

### Solution

Extract the individual parts of the method into their own methods and call them instead of the original method.

**void** **setHeight**(**int** arg) {

height = arg;

}

**void** **setWidth**(**int** arg) {

width = arg;

}

### Why Refactor

A method containing parameter-dependent variants has grown massive. Non-trivial code is run in each branch and new variants are added very rarely.

### Benefits

* Improves code readability. It’s much easier to understand the purpose of startEngine() than setValue("engineEnabled", true).

### When Not to Use

* Don’t replace a parameter with explicit methods if a method is rarely changed and new variants aren’t added inside it.

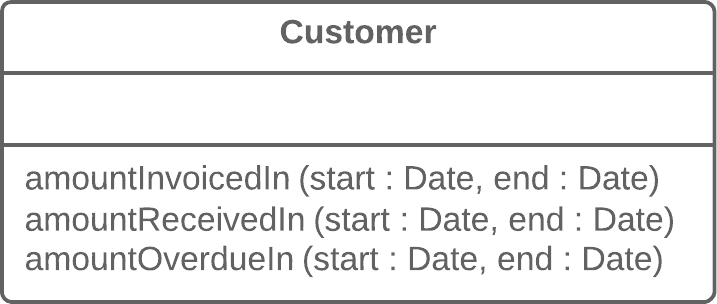
### How to Refactor

1. For each variant of the method, create a separate method. Run these methods based on the value of a parameter in the main method.
2. Find all places where the original method is called. In these places, place a call for one of the new parameter-dependent variants.
3. When no calls to the original method remain, delete it.

# Q 10: Introduce Parameter Object

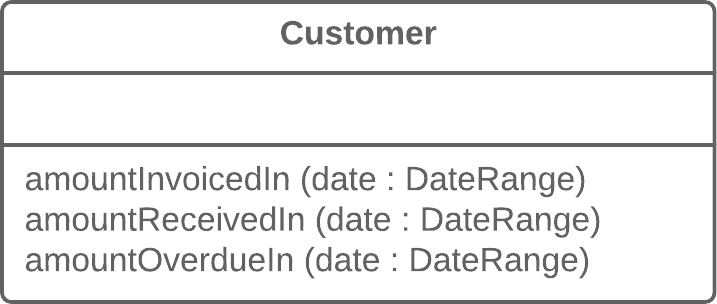
### Problem

Your methods contain a repeating group of parameters.



### Solution

Replace these parameters with an object.



### Why Refactor

Identical groups of parameters are often encountered in multiple methods. This causes code duplication of both the parameters themselves and of related operations. By consolidating parameters in a single class, you can also move the methods for handling this data there as well, freeing the other methods from this code.

### Benefits

* More readable code. Instead of a hodgepodge of parameters, you see a single object with a comprehensible name.
* Identical groups of parameters scattered here and there create their own kind of code duplication: while identical code isn’t being called, identical groups of parameters and arguments are constantly encountered.

### Drawbacks

* If you move only data to a new class and don’t plan to move any behaviors or related operations there, this begins to smell of a [**Data Class**](https://refactoring.guru/smells/data-class).

### How to Refactor

1. Create a new class that will represent your group of parameters. Make the class immutable.
2. In the method that you want to refactor, use [**Add Parameter**](https://refactoring.guru/add-parameter), which is where your parameter object will be passed. In all method calls, pass the object created from old method parameters to this parameter.
3. Now start deleting old parameters from the method one by one, replacing them in the code with fields of the parameter object. Test the program after each parameter replacement.
4. When done, see whether there’s any point in moving a part of the method (or sometimes even the whole method) to a parameter object class. If so, use [**Move Method**](https://refactoring.guru/move-method) or [**Extract Method**](https://refactoring.guru/extract-method).