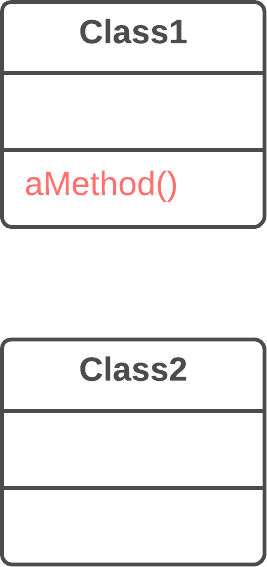
**Move Method**

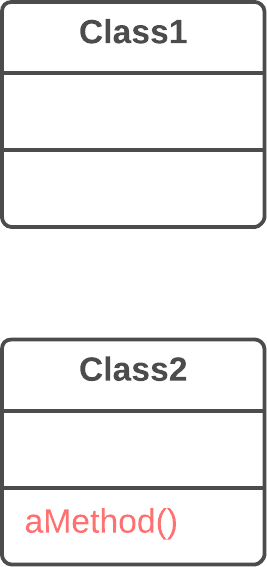
**Problem**

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.

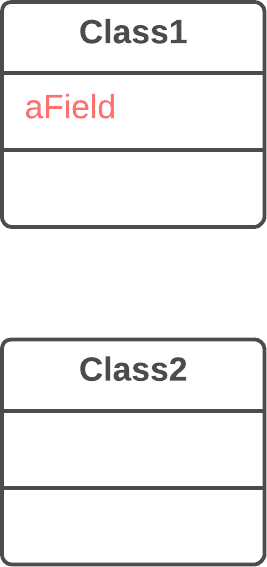
Now you have a way to refer to the recipient object and a new method in its class. With all this under your belt, you can turn the old method into a reference to the new method.

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**

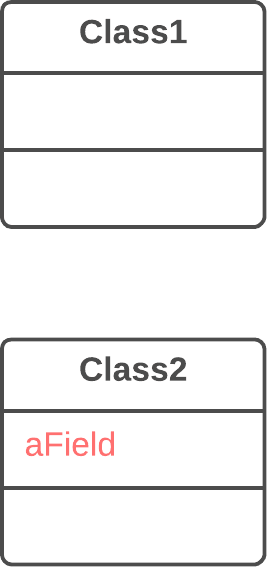
**Problem**

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.

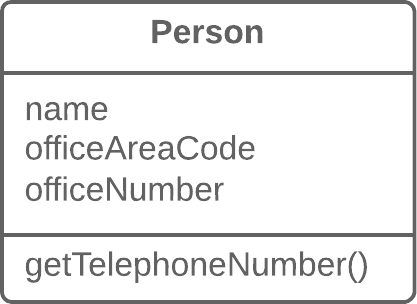
**How to Refactor**

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**

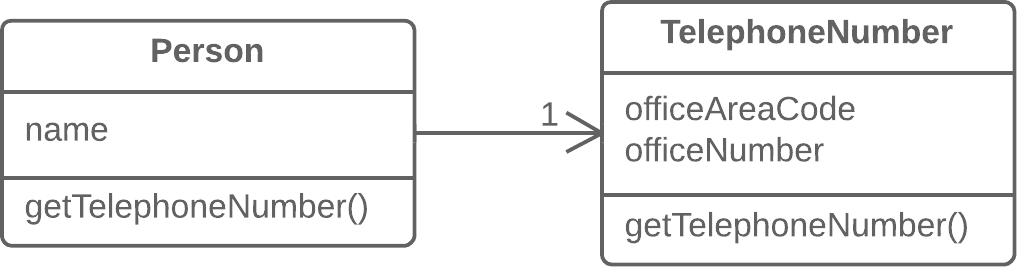
**Problem**

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.

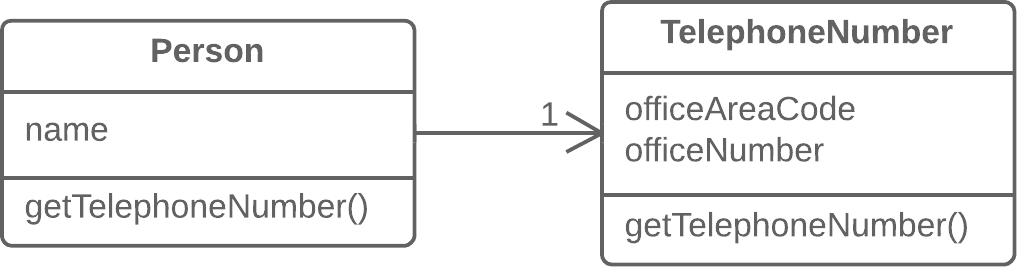
After you’re done moving, take one more look at the resulting classes. An old class with changed responsibilities may be renamed for increased clarity. Check again to see whether you can get rid of two-way class relationships, if any are present.

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.

**Inline Class**

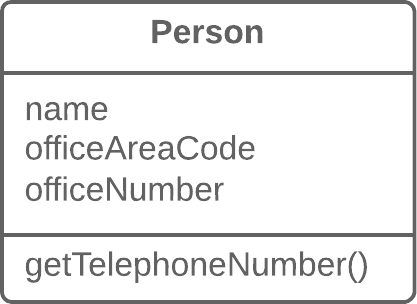
**Problem**

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.



**Why Refactor**

* Often this technique is needed after the features of one class are “transplanted” to other classes, leaving that class with little to do.

**Benefits**

* Eliminating needless classes frees up operating memory on the computer—and bandwidth in your head.

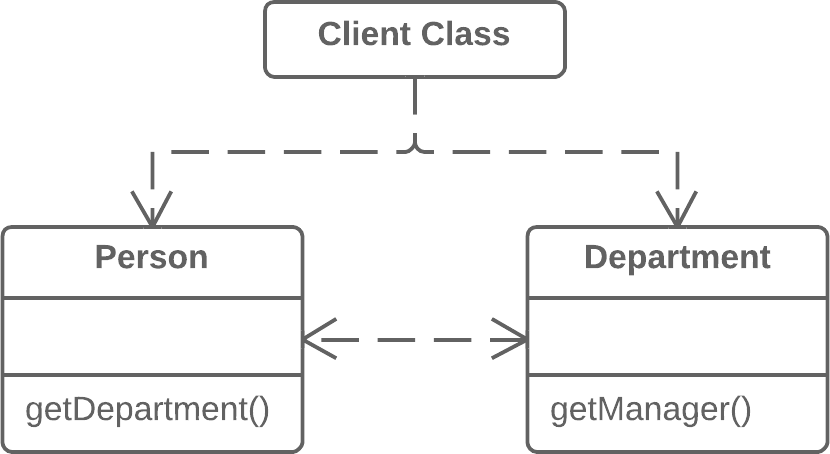
**How to Refactor**

1. In the recipient class, create the public fields and methods present in the donor class. Methods should refer to the equivalent methods of the donor class.
2. Replace all references to the donor class with references to the fields and methods of the recipient class.
3. Now test the program and make sure that no errors have been added. If tests show that everything is working A-OK, start using [**Move Method**](https://refactoring.guru/move-method) and [**Move Field**](https://refactoring.guru/move-field) to completely transplant all functionality to the recipient class from the original one. Continue doing so until the original class is completely empty.
4. Delete the original class.

**Hide Delegate**

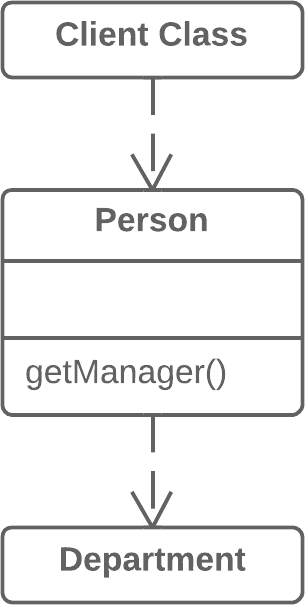
**Problem**

The client gets object B from a field or method of object А. 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.



**Why Refactor**

To start with, let’s look at terminology:

* *Server* is the object to which the client has direct access.
* *Delegate* is the end object that contains the functionality needed by the client.

A call chain appears when a client requests an object from another object, then the second object requests another one, and so on. These sequences of calls involve the client in navigation along the class structure. Any changes in these interrelationships will require changes on the client side.

**Benefits**

* Hides delegation from the client. The less that the client code needs to know about the details of relationships between objects, the easier it’s to make changes to your program.

**Drawbacks**

* If you need to create an excessive number of delegating methods, *server-class* risks becoming an unneeded go-between, leading to an excess of [**Middle Man**](https://refactoring.guru/smells/middle-man).

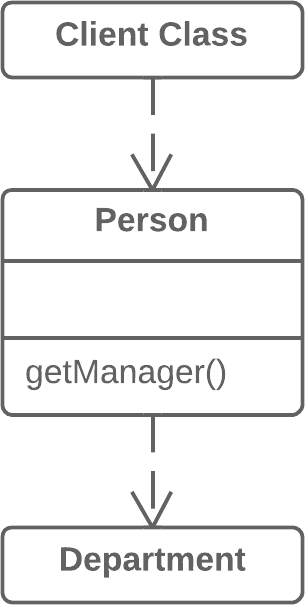
**How to Refactor**

1. For each method of the *delegate-class* called by the client, create a method in the *server-class* that delegates the call to the *delegate-class*.
2. Change the client code so that it calls the methods of the *server-class*.
3. If your changes free the client from needing the *delegate-class*, you can remove the access method to the *delegate-class* from the *server-class* (the method that was originally used to get the *delegate-class*).

**Remove Middle Man**

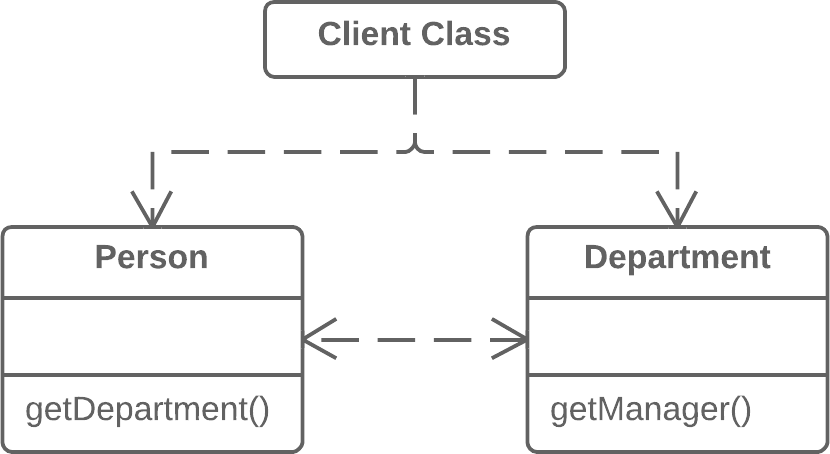
**Problem**

A class has too many methods that simply delegate to other objects.



**Solution**

Delete these methods and force the client to call the end methods directly.



**Why Refactor**

To describe this technique, we’ll use the terms from [**Hide Delegate**](https://refactoring.guru/hide-delegate), which are:

* *Server* is the object to which the client has direct access.
* *Delegate* is the end object that contains the functionality needed by the client.

There are two types of problems:

1. The *server-class* doesn’t do anything itself and simply creates needless complexity. In this case, give thought to whether this class is needed at all.
2. Every time a new feature is added to the *delegate*, you need to create a delegating method for it in the *server-class*. If a lot of changes are made, this will be rather tiresome.

**How to Refactor**

1. Create a getter for accessing the *delegate-class* object from the *server-class* object.
2. Replace calls to delegating methods in the *server-class* with direct calls for methods in the *delegate-class*.

**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.

**class** **Report** {

// ...

**void** sendReport() {

Date nextDay = **new** Date(previousEnd.getYear(),

previousEnd.getMonth(), previousEnd.getDate() + 1);

// ...

}

}

**Solution**

Add the method to a client class and pass an object of the utility class to it as an argument.

**class** **Report** {

// ...

**void** sendReport() {

Date newStart = nextDay(previousEnd);

// ...

}

**private** **static** Date nextDay(Date arg) {

**return** **new** Date(arg.getYear(), arg.getMonth(), arg.getDate() + 1);

}

}

**Why Refactor**

You have code that uses the data and methods of a certain class. You realize that the code will look and work much better inside a new method in the class. But you can’t add the method to the class because, for example, the class is located in a third-party library.

This refactoring has a big payoff when the code that you want to move to the method is repeated several times in different places in your program.

Since you’re passing an object of the utility class to the parameters of the new method, you have access to all of its fields. Inside the method, you can do practically everything that you want, as if the method were part of the utility class.

**Benefits**

* Removes code duplication. If your code is repeated in several places, you can replace these code fragments with a method call. This is better than duplication even considering that the foreign method is located in a suboptimal place.

**Drawbacks**

* The reasons for having the method of a utility class in a client class won’t always be clear to the person maintaining the code after you. If the method can be used in other classes, you could benefit by creating a wrapper for the utility class and placing the method there. This is also beneficial when there are several such utility methods. [**Introduce Local Extension**](https://refactoring.guru/introduce-local-extension) can help with this.

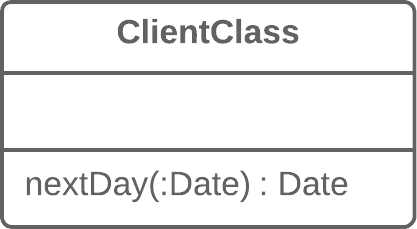
**How to Refactor**

1. Create a new method in the client class.
2. In this method, create a parameter to which the object of the utility class will be passed. If this object can be obtained from the client class, you don’t have to create such a parameter.
3. Extract the relevant code fragments to this method and replace them with method calls.
4. Be sure to leave the *Foreign method* tag in the comments for the method along with the advice to place this method in a utility class if such becomes possible later. This will make it easier to understand why this method is located in this particular class for those who’ll be maintaining the software in the future.

**Introduce Local Extension**

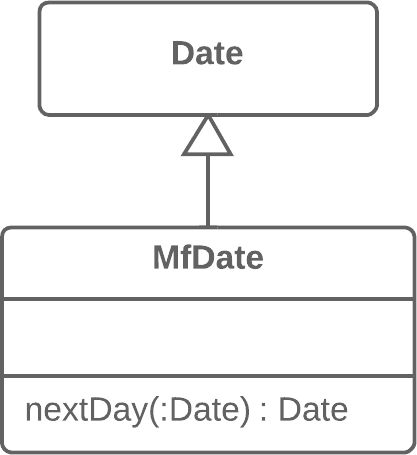
**Problem**

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.