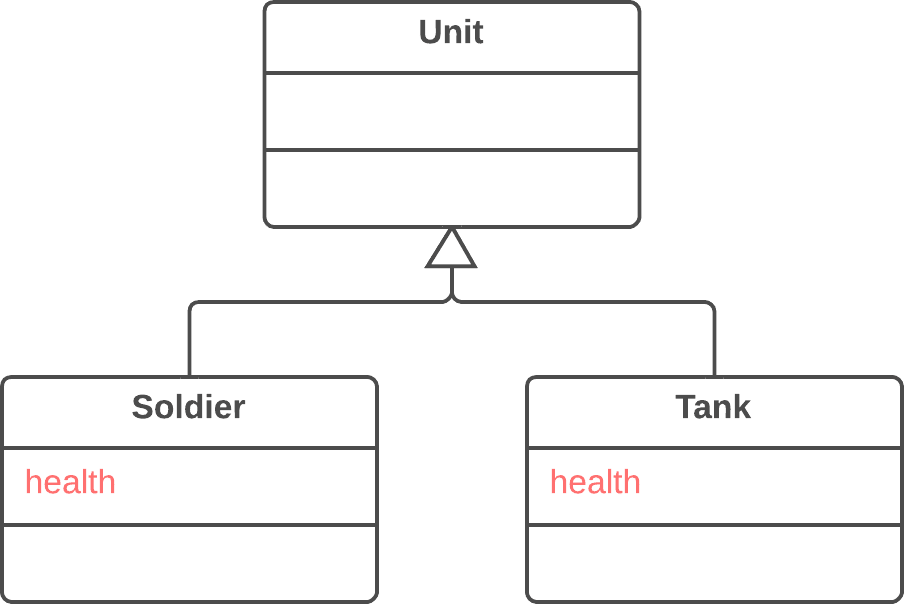
**Pull Up Field**

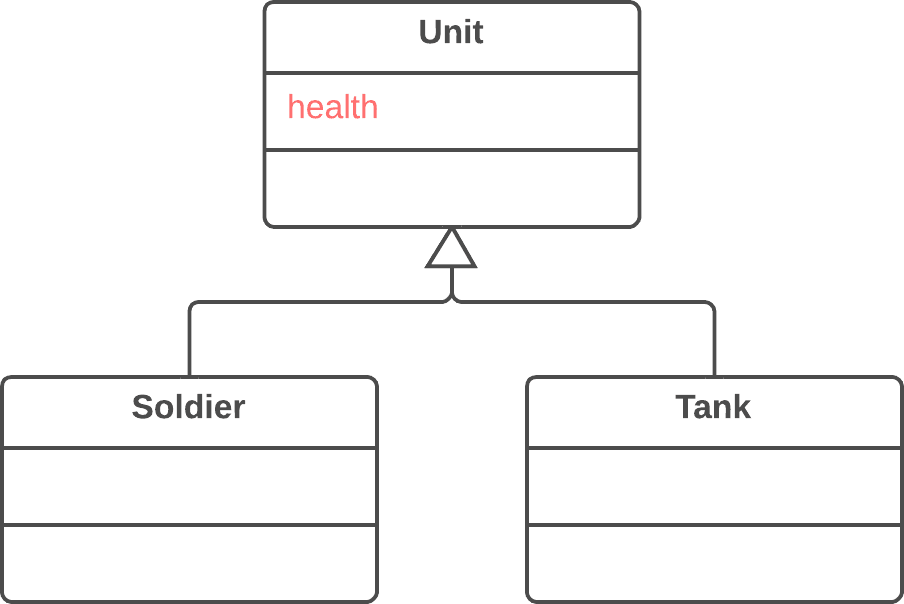
**Problem**

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.

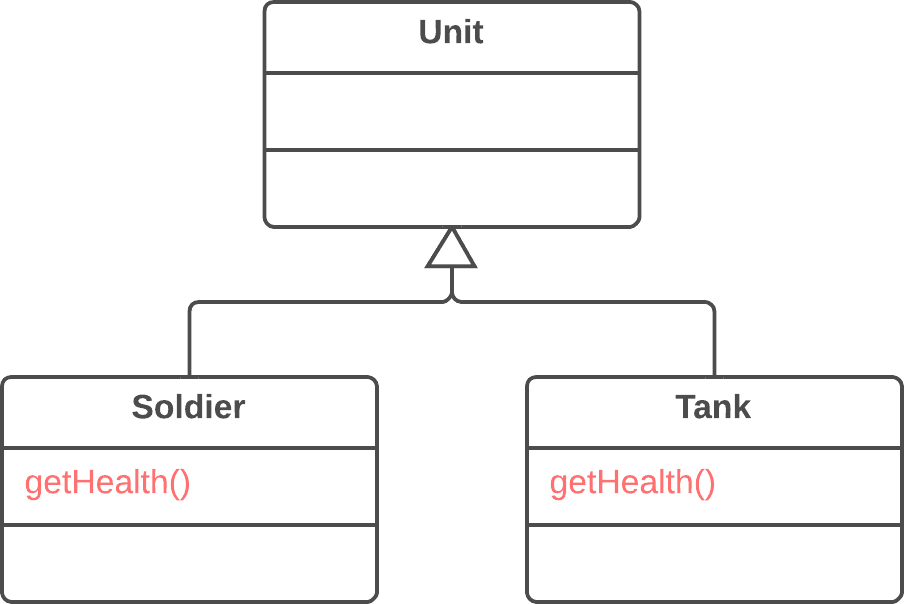
**How to Refactor**

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**

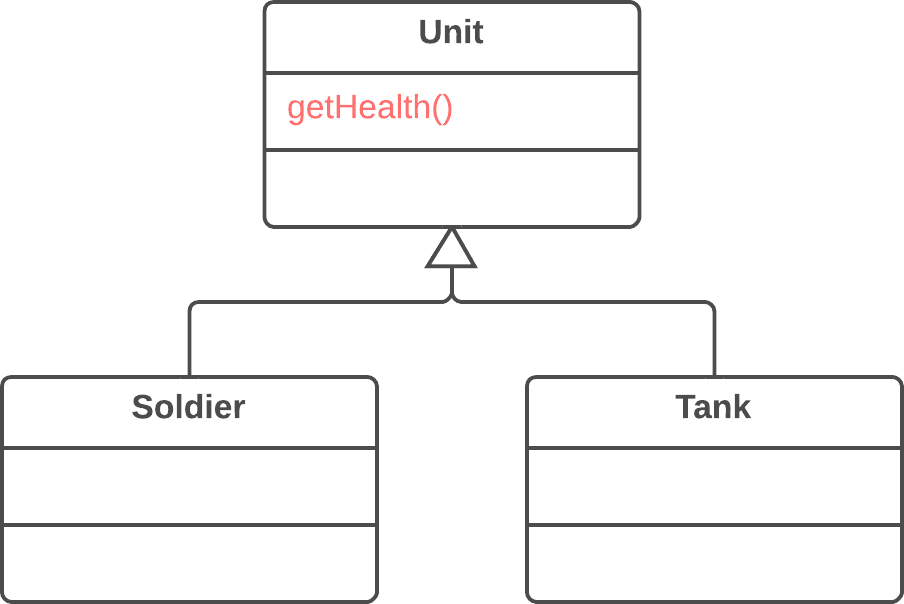
**Problem**

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.

**Pull Up Constructor Body**

**Problem**

Your subclasses have constructors with code that’s mostly identical.

**class** **Manager** **extends** Employee {

**public** Manager(**String** name, **String** id, **int** grade) {

**this**.name = name;

**this**.id = id;

**this**.grade = grade;

}

// ...

}

**Solution**

Create a superclass constructor and move the code that’s the same in the subclasses to it. Call the superclass constructor in the subclass constructors.

**class** **Manager** **extends** Employee {

**public** Manager(**String** name, **String** id, **int** grade) {

**super**(name, id);

**this**.grade = grade;

}

// ...

}

**Why Refactor**

How is this refactoring technique different from [**Pull Up Method**](https://refactoring.guru/pull-up-method)?

1. In Java, subclasses can’t inherit a constructor, so you can’t simply apply [**Pull Up Method**](https://refactoring.guru/pull-up-method) to the subclass constructor and delete it after removing all the constructor code to the superclass. In addition to creating a constructor in the superclass it’s necessary to have constructors in the subclasses with simple delegation to the superclass constructor.
2. In C++ and Java (if you didn’t explicitly call the superclass constructor) the superclass constructor is automatically called prior to the subclass constructor, which makes it necessary to move the common code only from the beginning of the subclass constructors (since you won’t be able to call the superclass constructor from an arbitrary place in a subclass constructor).
3. In most programming languages, a subclass constructor can have its own list of parameters different from the parameters of the superclass. Therefore you should create a superclass constructor only with the parameters that it truly needs.

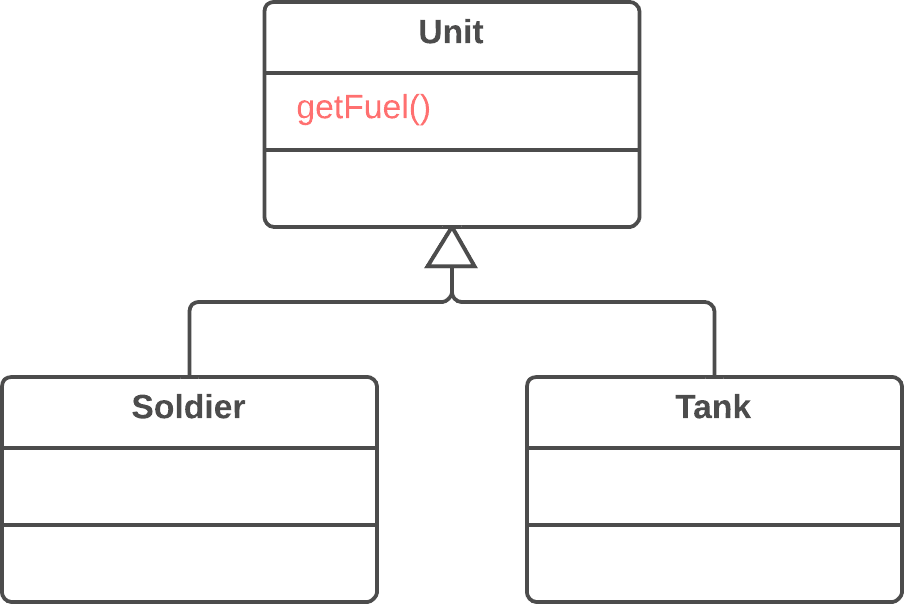
**How to Refactor**

1. Create a constructor in a superclass.
2. Extract the common code from the beginning of the constructor of each subclass to the superclass constructor. Before doing so, try to move as much common code as possible to the beginning of the constructor.
3. Place the call for the superclass constructor in the first line in the subclass constructors.

**Push Down Method**

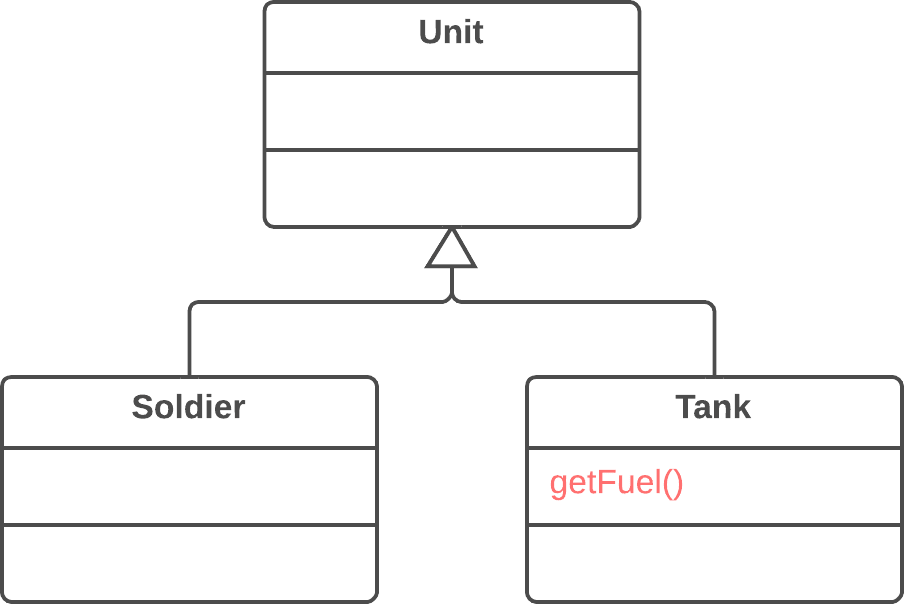
**Problem**

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.

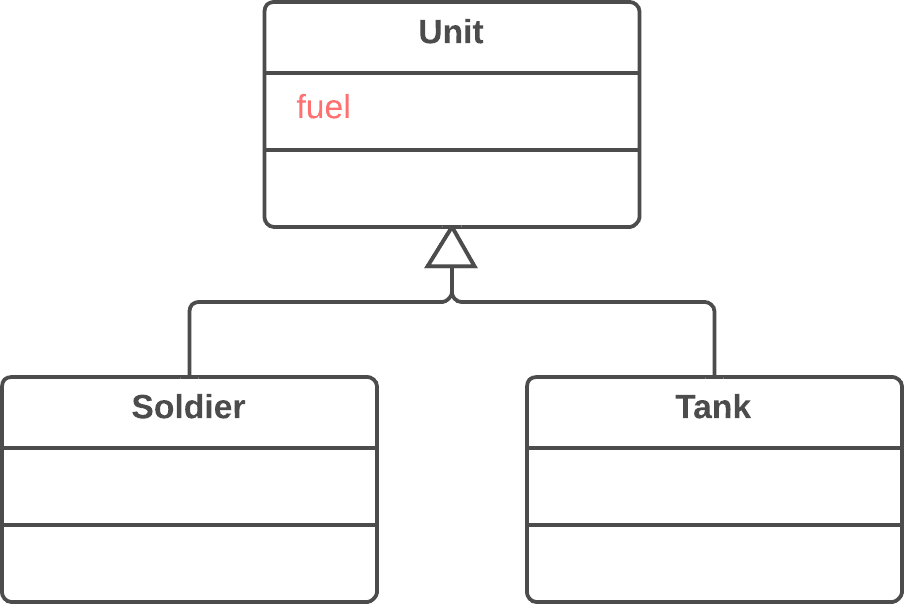
**How to Refactor**

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**

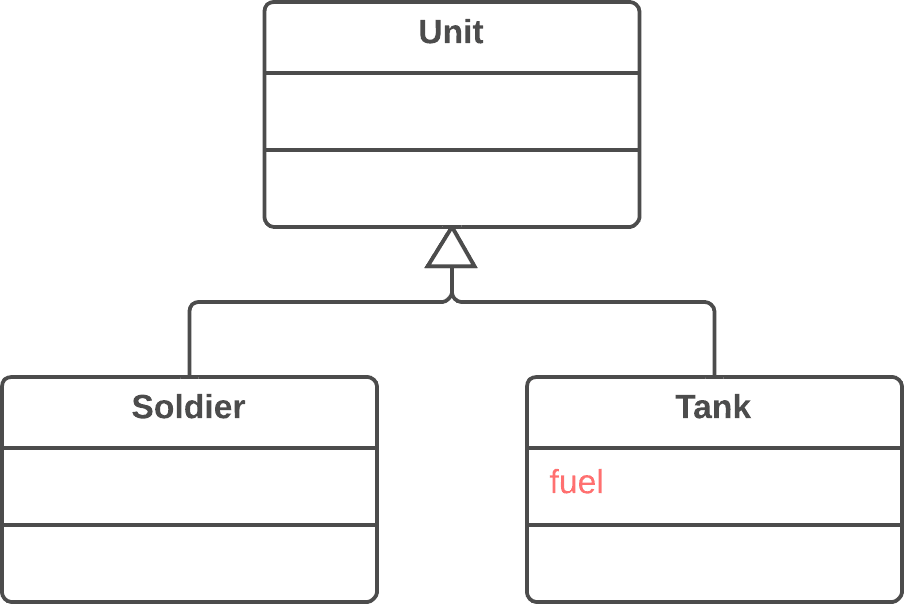
**Problem**

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.

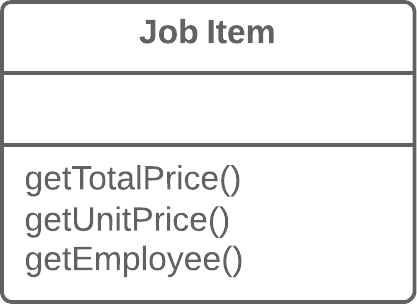
**How to Refactor**

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

**Extract Subclass**

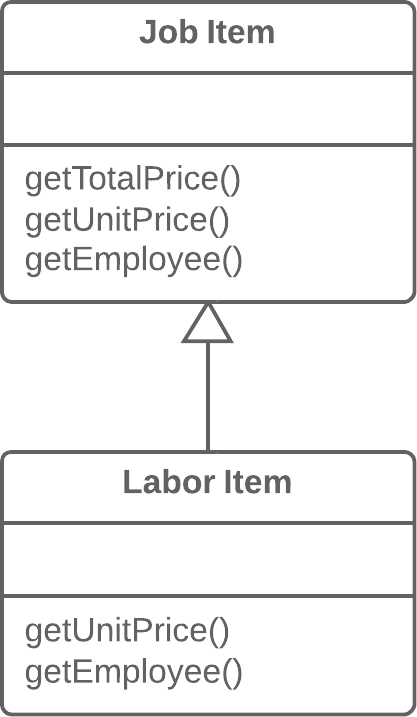
**Problem**

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.

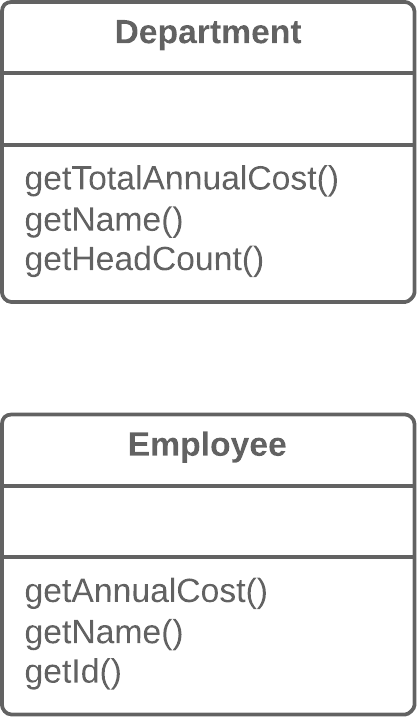
**How to Refactor**

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 Superclass**

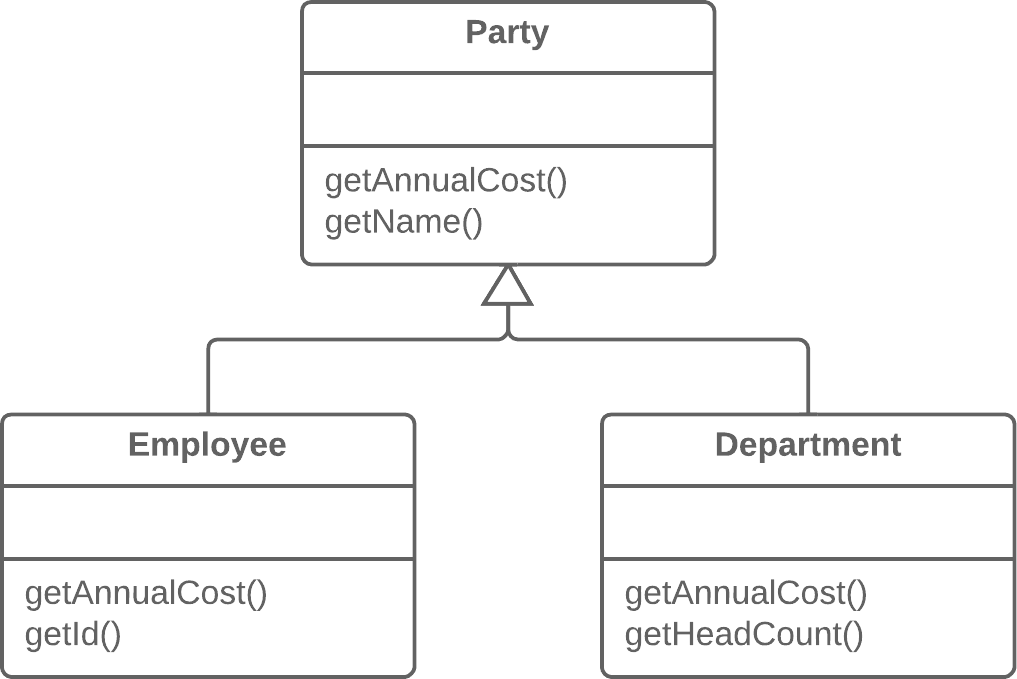
**Problem**

You have two classes with common fields and methods.



**Solution**

Create a shared superclass for them and move all the identical fields and methods to it.



**Why Refactor**

One type of code duplication occurs when two classes perform similar tasks in the same way, or perform similar tasks in different ways. Objects offer a built-in mechanism for simplifying such situations via inheritance. But oftentimes this similarity remains unnoticed until classes are created, necessitating that an inheritance structure be created later.

**Benefits**

* Code deduplication. Common fields and methods now “live” in one place only.

**When Not to Use**

* You can not apply this technique to classes that already have a superclass.

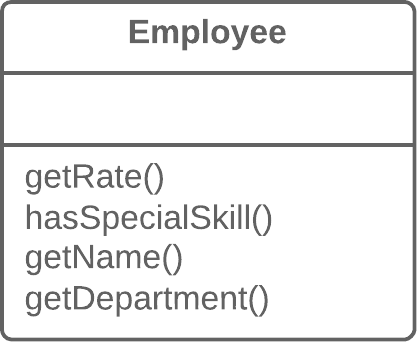
**How to Refactor**

1. Create an abstract superclass.
2. Use [**Pull Up Field**](https://refactoring.guru/pull-up-field), [**Pull Up Method**](https://refactoring.guru/pull-up-method), and [**Pull Up Constructor Body**](https://refactoring.guru/pull-up-constructor-body) to move the common functionality to a superclass. Start with the fields, since in addition to the common fields you will need to move the fields that are used in the common methods.
3. Look for places in the client code where use of subclasses can be replaced with your new class (such as in type declarations).

**Extract Interface**

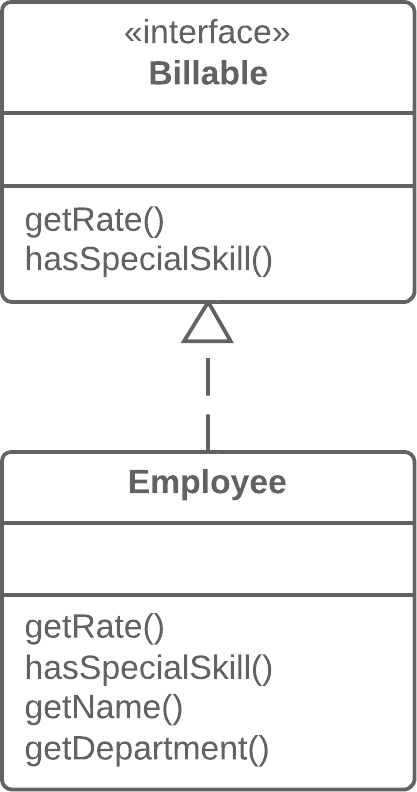
**Problem**

Multiple clients are using the same part of a class interface. Another case: part of the interface in two classes is the same.



**Solution**

Move this identical portion to its own interface.



**Why Refactor**

1. Interfaces are very apropos when classes play special roles in different situations. Use [**Extract Interface**](https://refactoring.guru/extract-interface) to explicitly indicate which role.
2. Another convenient case arises when you need to describe the operations that a class performs on its server. If it’s planned to eventually allow use of servers of multiple types, all servers must implement the interface.

**Good to Know**

There’s a certain resemblance between [**Extract Superclass**](https://refactoring.guru/extract-superclass) and [**Extract Interface**](https://refactoring.guru/extract-interface).

Extracting an interface allows isolating only common interfaces, not common code. In other words, if classes contain [**Duplicate Code**](https://refactoring.guru/smells/duplicate-code), extracting the interface won’t help you to deduplicate.

All the same, this problem can be mitigated by applying [**Extract Class**](https://refactoring.guru/extract-class) to move the behavior that contains the duplication to a separate component and delegating all the work to it. If the common behavior is large in size, you can always use [**Extract Superclass**](https://refactoring.guru/extract-superclass). This is even easier, of course, but remember that if you take this path you will get only one parent class.

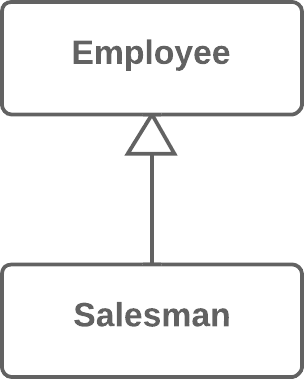
**How to Refactor**

1. Create an empty interface.
2. Declare common operations in the interface.
3. Declare the necessary classes as implementing the interface.
4. Change type declarations in the client code to use the new interface.

**Collapse Hierarchy**

**Problem**

You have a class hierarchy in which a subclass is practically the same as its superclass.



**Solution**

Merge the subclass and superclass.



**Why Refactor**

Your program has grown over time and a subclass and superclass have become practically the same. A feature was removed from a subclass, a method was moved to the superclass... and now you have two look-alike classes.

**Benefits**

* Program complexity is reduced. Fewer classes mean fewer things to keep straight in your head and fewer breakable moving parts to worry about during future code changes.
* Navigating through your code is easier when methods are defined in one class early. You don’t need to comb through the entire hierarchy to find a particular method.

**When Not to Use**

* Does the class hierarchy that you’re refactoring have more than one subclass? If so, after refactoring is complete, the remaining subclasses should become the inheritors of the class in which the hierarchy was collapsed.
* But keep in mind that this can lead to violations of the *Liskov substitution principle*. For example, if your program emulates city transport networks and you accidentally collapse the Transport superclass into the Car subclass, then the Plane class may become the inheritor of Car. Oops!

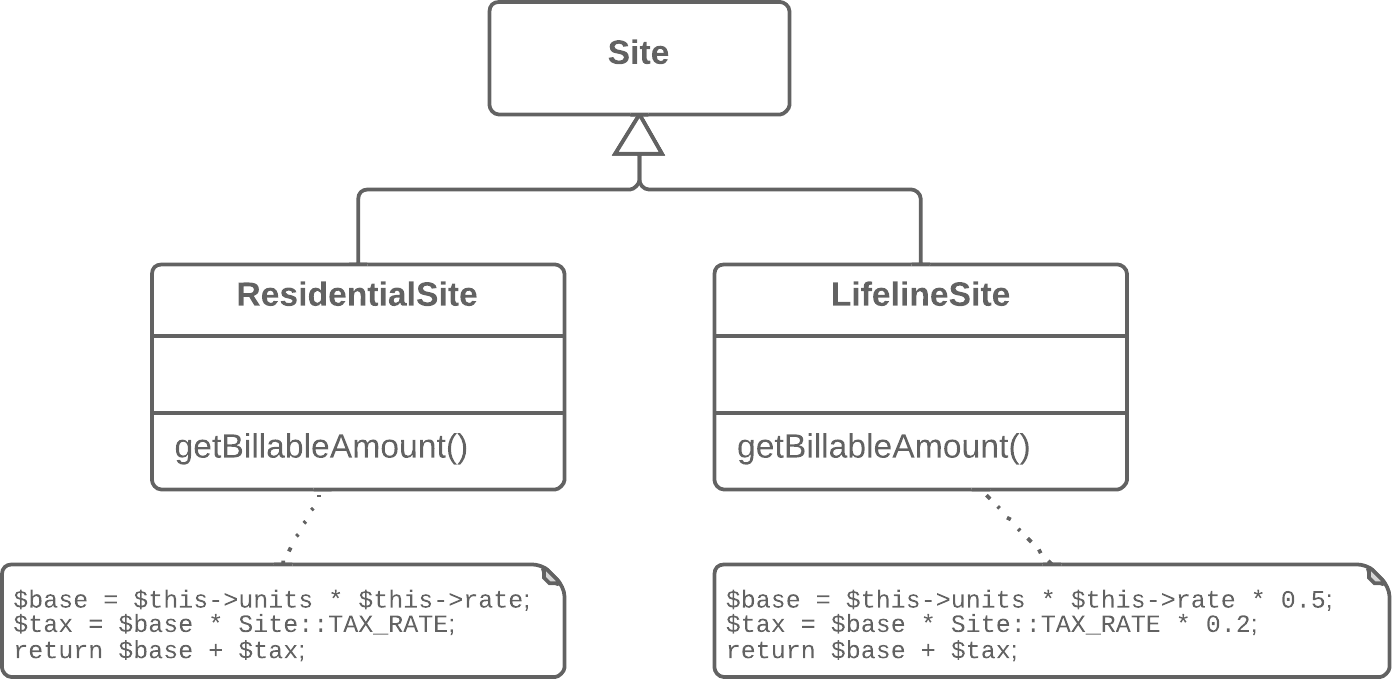
**How to Refactor**

1. Select which class is easier to remove: the superclass or its subclass.
2. Use [**Pull Up Field**](https://refactoring.guru/pull-up-field) and [**Pull Up Method**](https://refactoring.guru/pull-up-method) if you decide to get rid of the subclass. If you choose to eliminate the superclass, go for [**Push Down Field**](https://refactoring.guru/push-down-field) and [**Push Down Method**](https://refactoring.guru/push-down-method).
3. Replace all uses of the class that you’re deleting with the class to which the fields and methods are to be migrated. Often this will be code for creating classes, variable and parameter typing, and documentation in code comments.
4. Delete the empty class.

**Form Template Method**

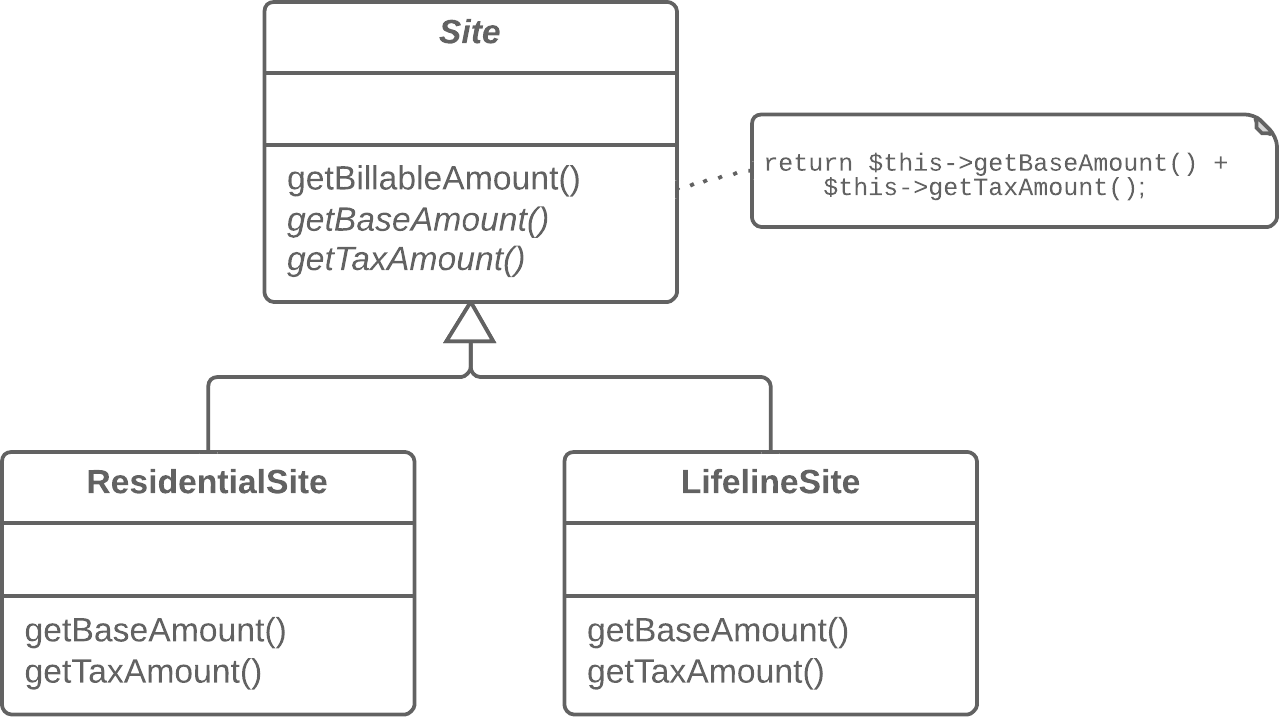
**Problem**

Your subclasses implement algorithms that contain similar steps in the same order.



**Solution**

Move the algorithm structure and identical steps to a superclass, and leave implementation of the different steps in the subclasses.



**Why Refactor**

Subclasses are developed in parallel, sometimes by different people, which leads to code duplication, errors, and difficulties in code maintenance, since each change must be made in all subclasses.

**Benefits**

* Code duplication doesn’t always refer to cases of simple copy/paste. Often duplication occurs at a higher level, such as when you have a method for sorting numbers and a method for sorting object collections that are differentiated only by the comparison of elements. Creating a template method eliminates this duplication by merging the shared algorithm steps in a superclass and leaving just the differences in the subclasses.
* Forming a template method is an example of the *Open/Closed Principle* in action. When a new algorithm version appears, you need only to create a new subclass; no changes to existing code are required.

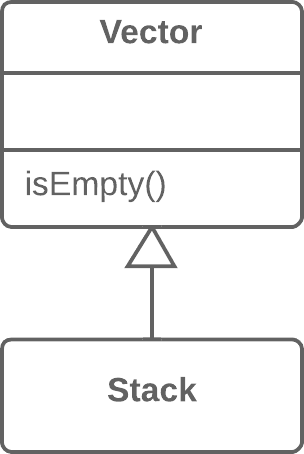
**How to Refactor**

1. Split algorithms in the subclasses into their constituent parts described in separate methods. [**Extract Method**](https://refactoring.guru/extract-method) can help with this.
2. The resulting methods that are identical for all subclasses can be moved to a superclass via [**Pull Up Method**](https://refactoring.guru/pull-up-method).
3. The non-similar methods can be given consistent names via [**Rename Method**](https://refactoring.guru/rename-method).
4. Move the signatures of non-similar methods to a superclass as abstract ones by using [**Pull Up Method**](https://refactoring.guru/pull-up-method). Leave their implementations in the subclasses.
5. And finally, pull up the main method of the algorithm to the superclass. Now it should work with the method steps described in the superclass, both real and abstract.

**Replace Inheritance with Delegation**

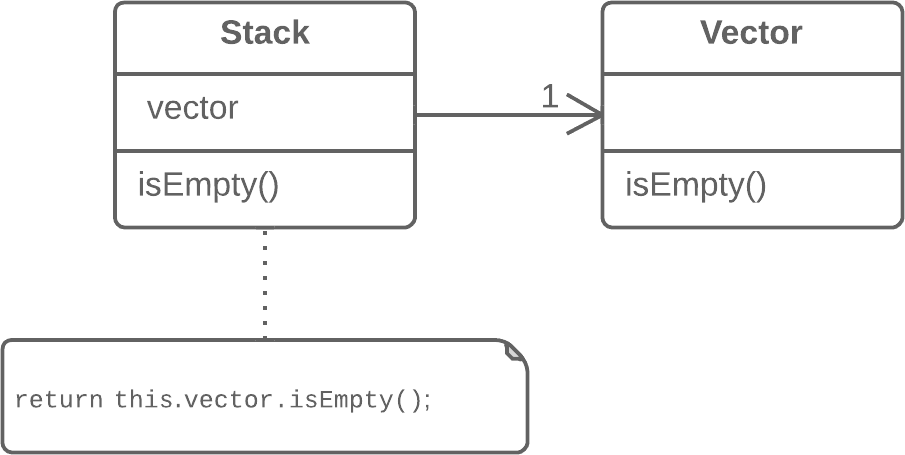
**Problem**

You have a subclass that uses only a portion of the methods of its superclass (or it’s not possible to inherit superclass data).



**Solution**

Create a field and put a superclass object in it, delegate methods to the superclass object, and get rid of inheritance.



**Why Refactor**

Replacing inheritance with composition can substantially improve class design if:

* Your subclass violates the *Liskov substitution principle*, i.e., if inheritance was implemented only to combine common code but not because the subclass is an extension of the superclass.
* The subclass uses only a portion of the methods of the superclass. In this case, it’s only a matter of time before someone calls a superclass method that he or she wasn’t supposed to call.

In essence, this refactoring technique splits both classes and makes the superclass the helper of the subclass, not its parent. Instead of inheriting all superclass methods, the subclass will have only the necessary methods for delegating to the methods of the superclass object.

**Benefits**

* A class doesn’t contain any unneeded methods inherited from the superclass.
* Various objects with various implementations can be put in the delegate field. In effect you get the [**Strategy**](https://refactoring.guru/design-patterns/strategy) design pattern.

**Drawbacks**

* You have to write many simple delegating methods.

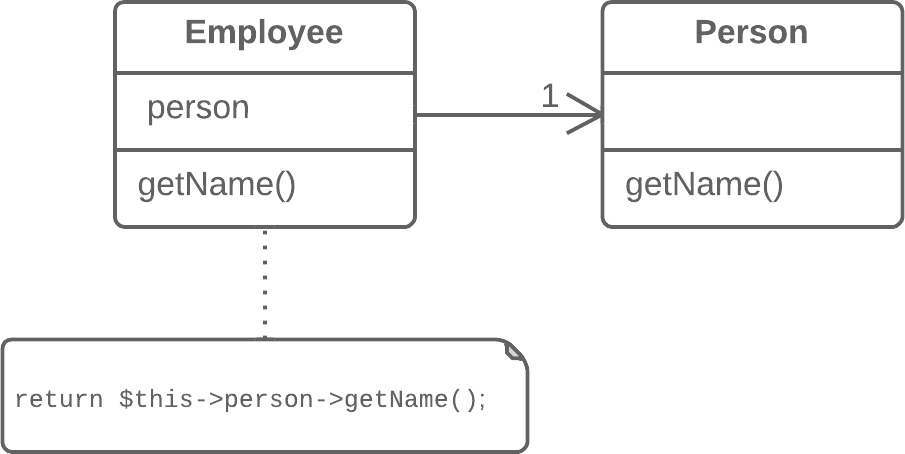
**How to Refactor**

1. Create a field in the subclass for holding the superclass. During the initial stage, place the current object in it.
2. Change the subclass methods so that they use the superclass object instead of this.
3. For methods inherited from the superclass that are called in the client code, create simple delegating methods in the subclass.
4. Remove the inheritance declaration from the subclass.
5. Change the initialization code of the field in which the former superclass is stored by creating a new object.

**Replace Delegation with Inheritance**

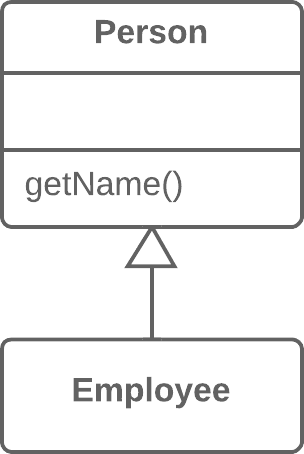
**Problem**

A class contains many simple methods that delegate to all methods of another class.



**Solution**

Make the class a delegate inheritor, which makes the delegating methods unnecessary.



**Why Refactor**

Delegation is a more flexible approach than inheritance, since it allows changing how delegation is implemented and placing other classes there as well. Nonetheless, delegation stops being beneficial if you delegate actions to only one class and all of its public methods.

In such a case, if you replace delegation with inheritance, you cleanse the class of a large number of delegating methods and spare yourself from needing to create them for each new delegate class method.

**Benefits**

* Reduces code length. All these delegating methods are no longer necessary.

**When Not to Use**

* Don’t use this technique if the class contains delegation to only a portion of the public methods of the delegate class. By doing so, you would violate the *Liskov substitution principle*.
* This technique can be used only if the class still doesn’t have parents.

**How to Refactor**

1. Make the class a subclass of the delegate class.
2. Place the current object in a field containing a reference to the delegate object.
3. Delete the methods with simple delegation one by one. If their names were different, use [**Rename Method**](https://refactoring.guru/rename-method) to give all the methods a single name.
4. Replace all references to the delegate field with references to the current object.
5. Remove the delegate field.