

# Dive Into *Refactoring*

👉 Offline Edition



**REFACTORING**  
· GURU ·

by Alexander Shvets

# Dive Into Refactoring

⚡ Offline Edition (:programmingLangauge:)

v.version:

Purchased by Praveen Konduru  
praveen.konduru@gmail.com (#9820)

# Boring Copyright Page

Hi! My name is Alexander Shvets, I'm the author of the online course Dive Into Refactoring, which includes this book.



This book is licensed for your personal use. Please, don't share it with other people, except your family members. If you would like to share the book with your friend or colleague, please purchase a copy of the course and gift it to him or her.

If you're reading this book and did not purchase it, or it was not purchased for you, then please be a decent person and purchase your own copy of the course.

Thank you for respecting years of hard work I've put into creating the course, as well as this book!

© Alexander Shvets, Refactoring.Guru, 2017

✉ [support@refactoring.guru](mailto:support@refactoring.guru)

 Illustrations by Dmitry Zhart

# Preface

I tried to cram all of the available information throughout the full course into this book. For the most part, I succeeded. But some things, such as live examples, are just impossible to present in the form of a static electronic book. Therefore, consider this book as auxiliary material, but not a replacement for the full refactoring course.

The book is divided into two large sections: **Code Smells** and **Refactoring Techniques**. The first part describes various signs and symptoms of dirty code. The second part shows different methods to treat dirty code and make it clean.

The book can be read both from cover to cover, as well as in random order. Despite the fact that all the topics are closely intertwined, you can easily jump around the chapters, using the great number of links scattered through the text.

The code examples in this version of the book are in **:programmingLangauge:**. There are other versions which are available for download inside your account.

# Code Smells

Code smells are key signs that refactoring is necessary. In the process of refactoring, we get rid of smells, enabling further development of the application with equal or greater speed.



The lack of regular refactoring, can lead to a complete paralysis of a project over time, wasting a few years of development and requiring you to spend several more years to rewrite it from scratch.

Therefore, it is necessary to get rid of code smells while they are still small.

# Bloaters

Bloaters are code, methods and classes that have increased to such gargantuan proportions that they're hard to work with. Usually these smells don't crop up right away, rather they accumulate over time as the program evolves (and especially when nobody makes an effort to eradicate them).

## § Long Method

A method contains too many lines of code. Generally, any method longer than ten lines should make you start asking questions.

## § Large Class

A class contains many fields/methods/lines of code.

## § Primitive Obsession

- Use of primitives instead of small objects for simple tasks (such as currency, ranges, special strings for phone numbers, etc.)
- Use of constants for coding information (such as a constant `USER_ADMIN_ROLE = 1` for referring to users with administrator rights.)
- Use of string constants as field names for use in data arrays.

## § **Long Parameter List**

More than three or four parameters for a method.

## § **Data Clumps**

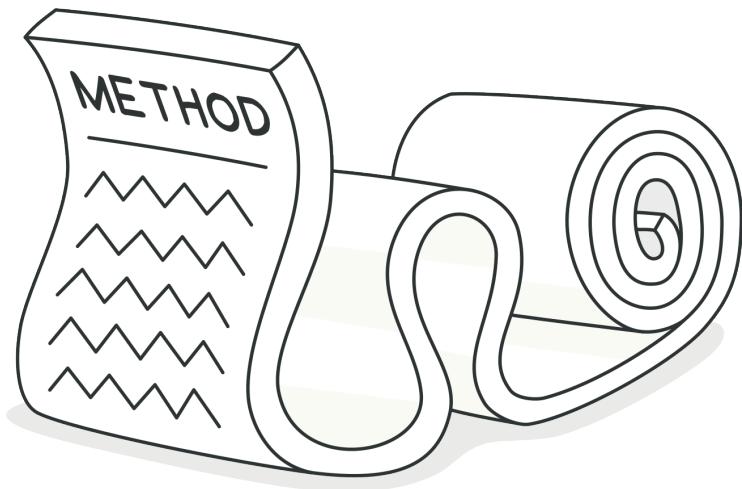
Sometimes different parts of the code contain identical groups of variables (such as parameters for connecting to a database). These clumps should be turned into their own classes.



# Long Method

## Signs and Symptoms

A method contains too many lines of code. Generally, any method longer than ten lines should make you start asking questions.



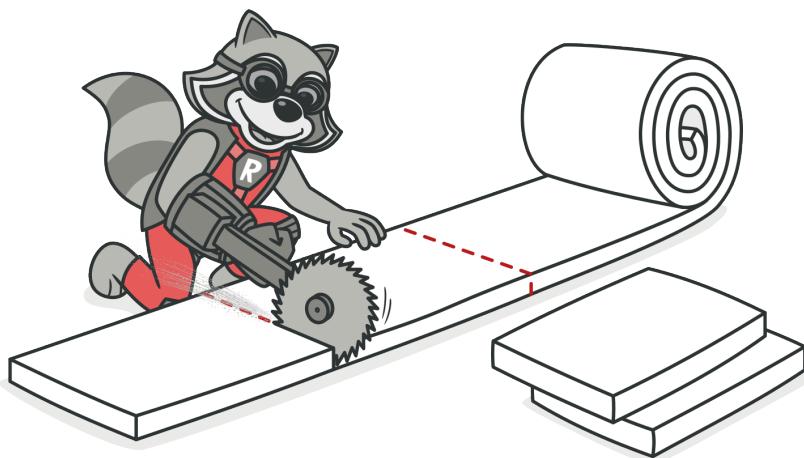
## Reasons for the Problem

Like the Hotel California, something is always being added to a method but nothing is ever taken out. Since it's easier to write code than to read it, this "smell" remains unnoticed until the method turns into an ugly, oversized beast.

Mentally, it's often harder to create a new method than to add to an existing one: "But it's just two lines, there's no use in creating a whole method just for that..." Which means that another line is added and then yet another, giving birth to a tangle of spaghetti code.

## Treatment

As a rule of thumb, if you feel the need to comment on something inside a method, you should take this code and put it in a new method. Even a single line can and should be split off into a separate method, if it requires explanations. And if the method has a descriptive name, nobody will need to look at the code to see what it does.

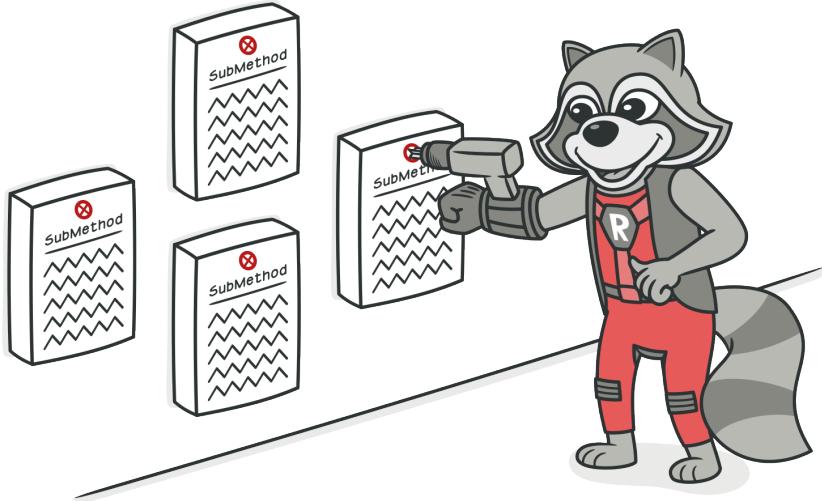


- To reduce the length of a method body, use [Extract Method](#).

- If local variables and parameters interfere with extracting a method, use **Replace Temp with Query**, **Introduce Parameter Object** or **Preserve Whole Object**.
- If none of the previous recipes help, try moving the entire method to a separate object via **Replace Method with Method Object**.
- Conditional operators and loops are a good clue that code can be moved to a separate method. For conditionals, use **Decompose Conditional**. If loops are in the way, try **Extract Method**.

## Payoff

- Among all types of object-oriented code, classes with short methods live longest. The longer a method or function is, the harder it becomes to understand and maintain it.
- In addition, long methods offer the perfect hiding place for unwanted duplicate code.



## Performance

Does an increase in the number of methods hurt performance, as many people claim? In almost all cases the impact is so negligible that it's not even worth worrying about.

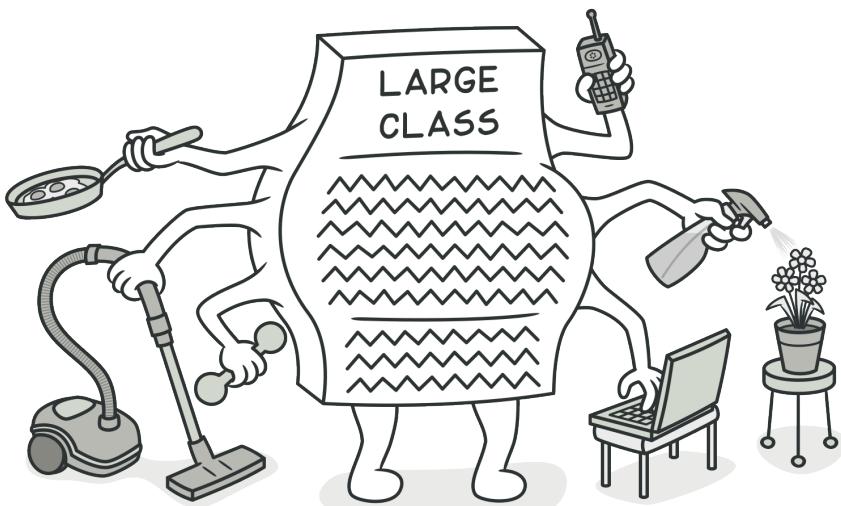
Plus, now that you have clear and understandable code, you're more likely to find truly effective methods for restructuring code and getting real performance gains if the need ever arises.



# Large Class

## Signs and Symptoms

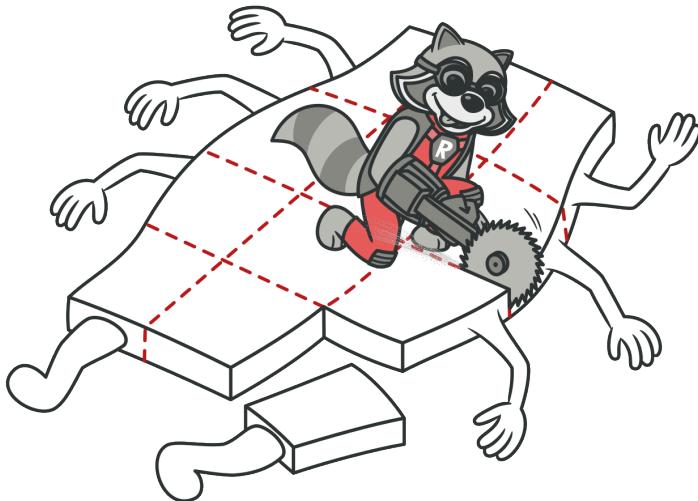
A class contains many fields/methods/lines of code.



## Reasons for the Problem

Classes usually start small. But over time, they get bloated as the program grows.

As is the case with long methods as well, programmers usually find it mentally less taxing to place a new feature in an existing class than to create a new class for the feature.

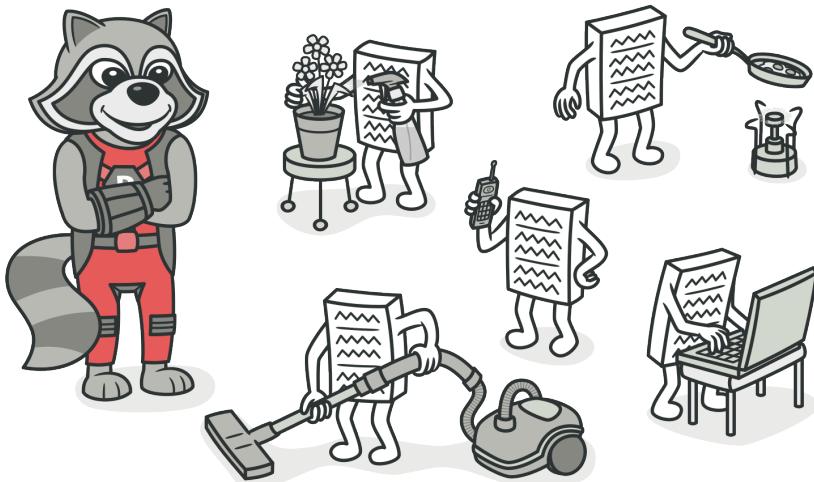


## Treatment

When a class is wearing too many (functional) hats, think about splitting it up:

- **Extract Class** helps if part of the behavior of the large class can be spun off into a separate component.
- **Extract Subclass** helps if part of the behavior of the large class can be implemented in different ways or is used in rare cases.
- **Extract Interface** helps if it's necessary to have a list of the operations and behaviors that the client can use.
- If a large class is responsible for the graphical interface, you may try to move some of its data and behavior to a separate domain object. In doing so, it may be necessary to store

copies of some data in two places and keep the data consistent. **Duplicate Observed Data** offers a way to do this.



## Payoff

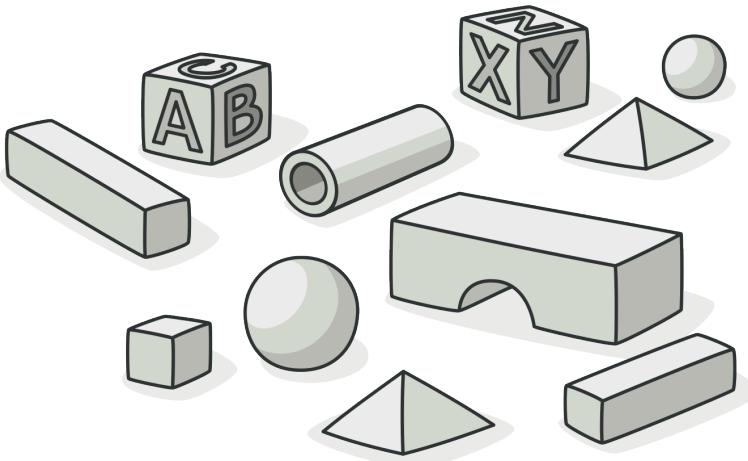
- Refactoring of these classes spares developers from needing to remember a large number of attributes for a class.
- In many cases, splitting large classes into parts avoids duplication of code and functionality.



# Primitive Obsession

## Signs and Symptoms

- Use of primitives instead of small objects for simple tasks (such as currency, ranges, special strings for phone numbers, etc.)
- Use of constants for coding information (such as a constant `USER_ADMIN_ROLE = 1` for referring to users with administrator rights.)
- Use of string constants as field names for use in data arrays.



## Reasons for the Problem

Like most other smells, primitive obsessions are born in moments of weakness. “Just a field for storing some data!” the programmer said. Creating a primitive field is so much easier than making a whole new class, right? And so it was done. Then another field was needed and added in the same way. Lo and behold, the class became huge and unwieldy.

Primitives are often used to “simulate” types. So instead of a separate data type, you have a set of numbers or strings that form the list of allowable values for some entity. Easy-to-understand names are then given to these specific numbers and strings via constants, which is why they’re spread wide and far.

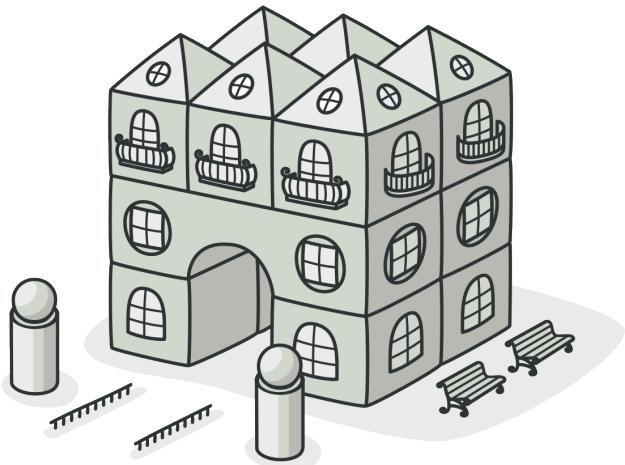
Another example of poor primitive use is field simulation. The class contains a large array of diverse data and string constants (which are specified in the class) are used as array indices for getting this data.

## Treatment

- If you have a large variety of primitive fields, it may be possible to logically group some of them into their own class. Even better, move the behavior associated with this data into the class too. For this task, try [\*\*Replace Data Value with Object\*\*](#).



- If the values of primitive fields are used in method parameters, go with [\*\*Introduce Parameter Object\*\*](#) or [\*\*Preserve Whole Object\*\*](#).
- When complicated data is coded in variables, use [\*\*Replace Type Code with Class\*\*](#), [\*\*Replace Type Code with Subclasses\*\*](#) or [\*\*Replace Type Code with State/Strategy\*\*](#).
- If there are arrays among the variables, use [\*\*Replace Array with Object\*\*](#).



## Payoff

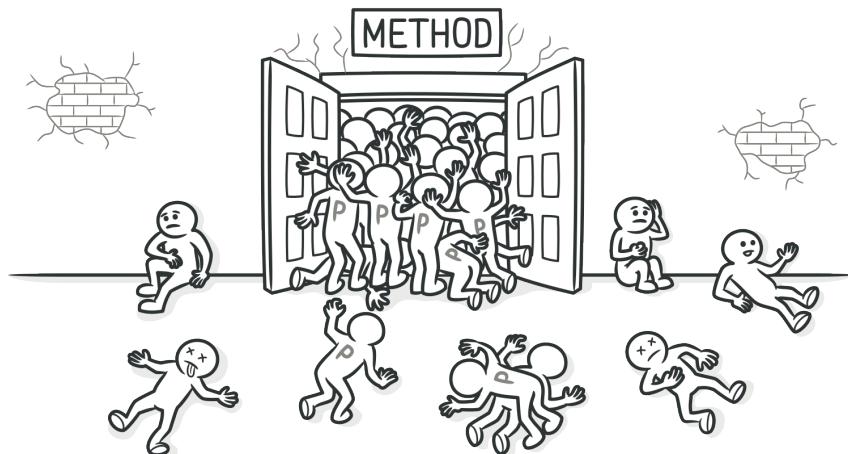
- Code becomes more flexible thanks to use of objects instead of primitives.
- Better understandability and organization of code. Operations on particular data are in the same place, instead of being scattered. No more guessing about the reason for all these strange constants and why they're in an array.
- Easier finding of duplicate code.



# Long Parameter List

## Signs and Symptoms

More than three or four parameters for a method.



## Reasons for the Problem

A long list of parameters might happen after several types of algorithms are merged in a single method. A long list may have been created to control which algorithm will be run and how.

Long parameter lists may also be the byproduct of efforts to make classes more independent of each other. For example, the code for creating specific objects needed in a method was

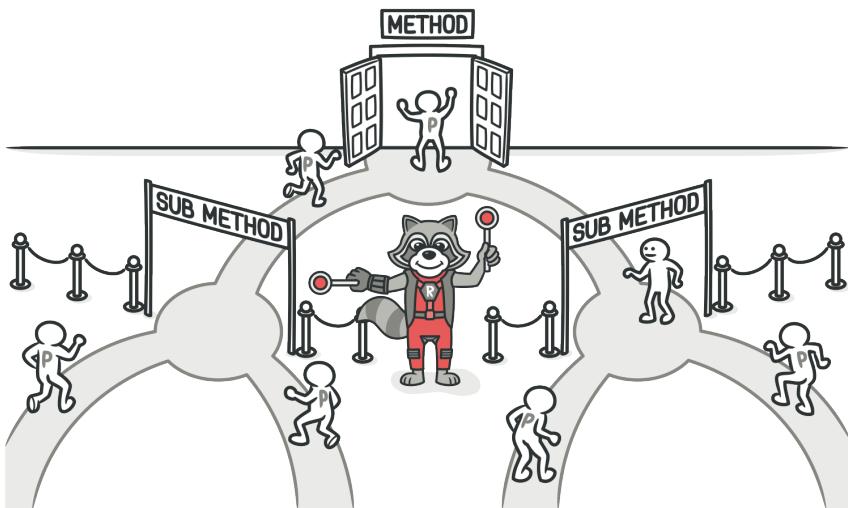
moved from the method to the code for calling the method, but the created objects are passed to the method as parameters. Thus the original class no longer knows about the relationships between objects, and dependency has decreased. But if several of these objects are created, each of them will require its own parameter, which means a longer parameter list.

It's hard to understand such lists, which become contradictory and hard to use as they grow longer. Instead of a long list of parameters, a method can use the data of its own object. If the current object doesn't contain all necessary data, another object (which will get the necessary data) can be passed as a method parameter.

## Treatment

- Check what values are passed to parameters. If some of the arguments are just results of method calls of another object, use **Replace Parameter with Method Call**. This object can be placed in the field of its own class or passed as a method parameter.
- Instead of passing a group of data received from another object as parameters, pass the object itself to the method, by using **Preserve Whole Object**.

- If there are several unrelated data elements, sometimes you can merge them into a single parameter object via Introduce Parameter Object.



## Payoff

- More readable, shorter code.
- Refactoring may reveal previously unnoticed duplicate code.

## When to Ignore

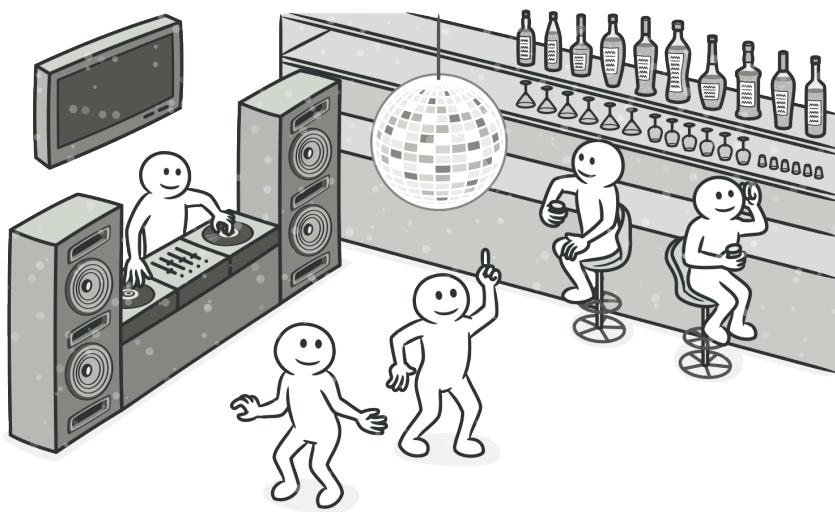
Don't get rid of parameters if doing so would cause unwanted dependency between classes.



# Data Clumps

## Signs and Symptoms

Sometimes different parts of the code contain identical groups of variables (such as parameters for connecting to a database). These clumps should be turned into their own classes.



## Reasons for the Problem

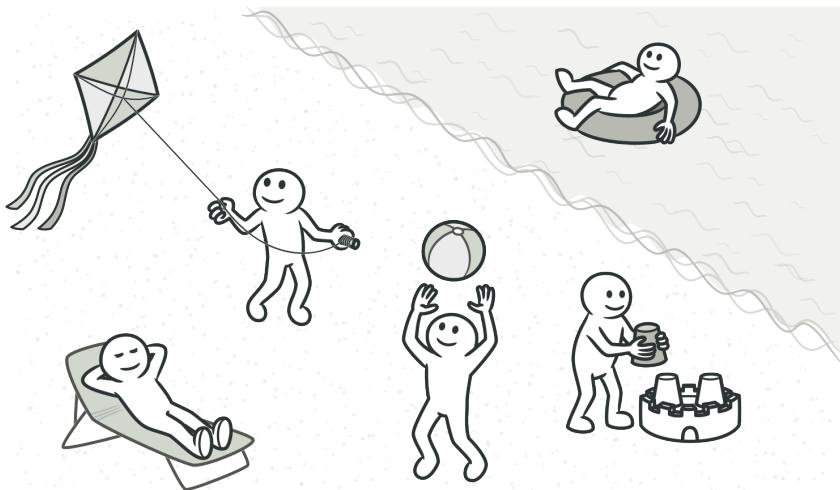
Often these data groups are due to poor program structure or "copypasta programming".

If you want to make sure whether or not some data is a data clump, just delete one of the data values and see whether the other values still make sense. If this isn't the case, this is a

good sign that this group of variables should be combined into an object.

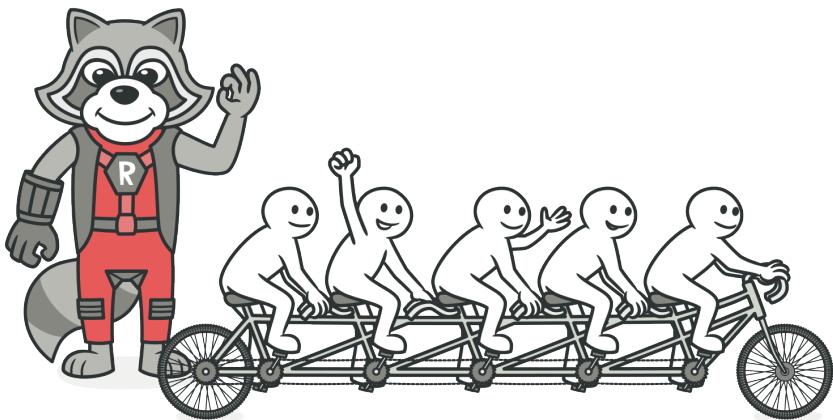
## Treatment

- If repeating data comprises the fields of a class, use **Extract Class** to move the fields to their own class.
- If the same data clumps are passed in the parameters of methods, use **Introduce Parameter Object** to set them off as a class.
- If some of the data is passed to other methods, think about passing the entire data object to the method instead of just individual fields. **Preserve Whole Object** will help with this.
- Look at the code used by these fields. It may be a good idea to move this code to a data class.



## Payoff

- Improves understanding and organization of code. Operations on particular data are now gathered in a single place, instead of haphazardly throughout the code.
- Reduces code size.



## When to Ignore

Passing an entire object in the parameters of a method, instead of passing just its values (primitive types), may create an undesirable dependency between the two classes.

# Object-Orientation Abusers

All these smells are incomplete or incorrect application of object-oriented programming principles.

## § Switch Statements

You have a complex `switch` operator or sequence of `if` statements.

## § Temporary Field

Temporary fields get their values (and thus are needed by objects) only under certain circumstances. Outside of these circumstances, they're empty.

## § Refused Bequest

If a subclass uses only some of the methods and properties inherited from its parents, the hierarchy is off-kilter. The unneeded methods may simply go unused or be redefined and give off exceptions.

## § Alternative Classes with Different Interfaces

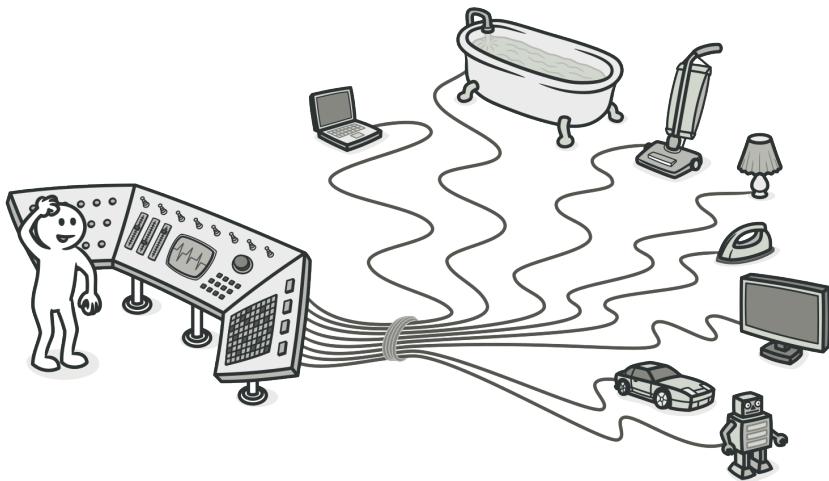
Two classes perform identical functions but have different method names.



# Switch Statements

## Signs and Symptoms

You have a complex `switch` operator or sequence of `if` statements.



## Reasons for the Problem

Relatively rare use of `switch` and `case` operators is one of the hallmarks of object-oriented code. Often code for a single `switch` can be scattered in different places in the program. When a new condition is added, you have to find all the `switch` code and modify it.

As a rule of thumb, when you see `switch` you should think of polymorphism.

## Treatment

- To isolate `switch` and put it in the right class, you may need **Extract Method** and then **Move Method**.
- If a `switch` is based on type code, such as when the program's runtime mode is switched, use **Replace Type Code with Sub-classes** or **Replace Type Code with State/Strategy**.
- After specifying the inheritance structure, use **Replace Conditional with Polymorphism**.
- If there aren't too many conditions in the operator and they all call same method with different parameters, polymorphism will be superfluous. If this case, you can break that method into multiple smaller methods with **Replace Parameter with Explicit Methods** and change the `switch` accordingly.
- If one of the conditional options is `null`, use **Introduce Null Object**.

## Payoff

Improved code organization.



## When to Ignore

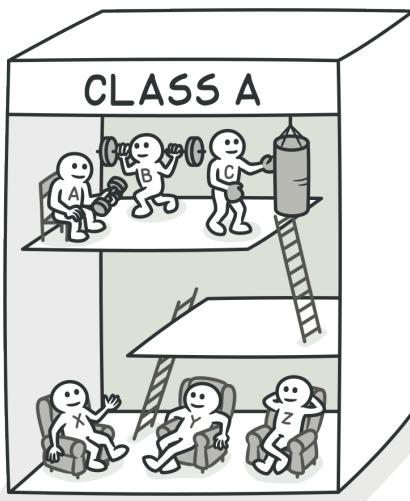
- When a `switch` operator performs simple actions, there's no reason to make code changes.
- Often `switch` operators are used by factory design patterns (**Factory Method** or **Abstract Factory**) to select a created class.



# Temporary Field

## Signs and Symptoms

Temporary fields get their values (and thus are needed by objects) only under certain circumstances. Outside of these circumstances, they're empty.



## Reasons for the Problem

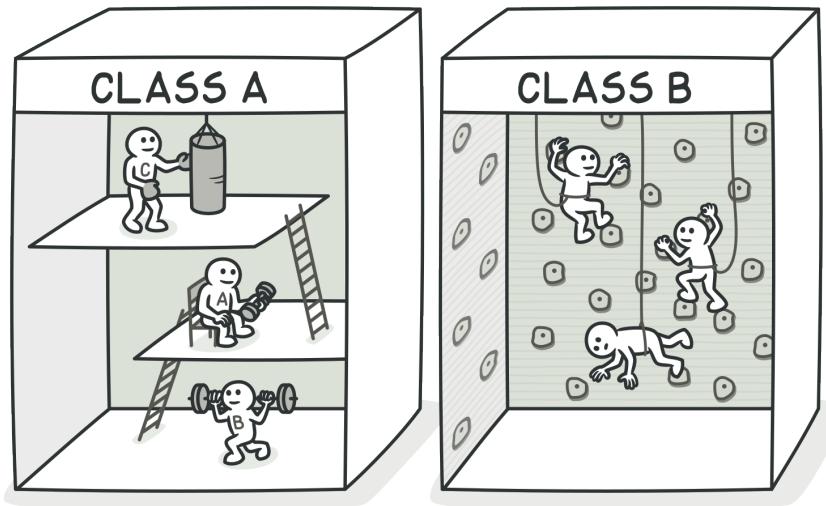
Oftentimes, temporary fields are created for use in an algorithm that requires a large amount of inputs. So instead of creating a large number of parameters in the method, the programmer decides to create fields for this data in the class. These fields are used only in the algorithm and go unused the rest of the time.

This kind of code is tough to understand. You expect to see data in object fields but for some reason they're almost always empty.



## Treatment

- Temporary fields and all code operating on them can be put in a separate class via **Extract Class**. In other words, you're creating a method object, achieving the same result as if you would perform **Replace Method with Method Object**.
- **Introduce Null Object** and integrate it in place of the conditional code which was used to check the temporary field values for existence.



## Payoff

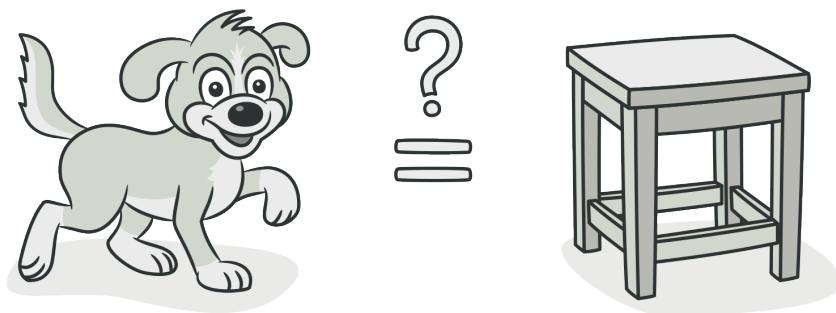
Better code clarity and organization.



# Refused Bequest

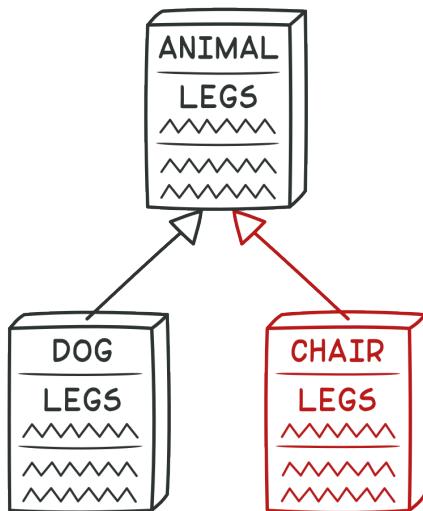
## Signs and Symptoms

If a subclass uses only some of the methods and properties inherited from its parents, the hierarchy is off-kilter. The unneeded methods may simply go unused or be redefined and give off exceptions.



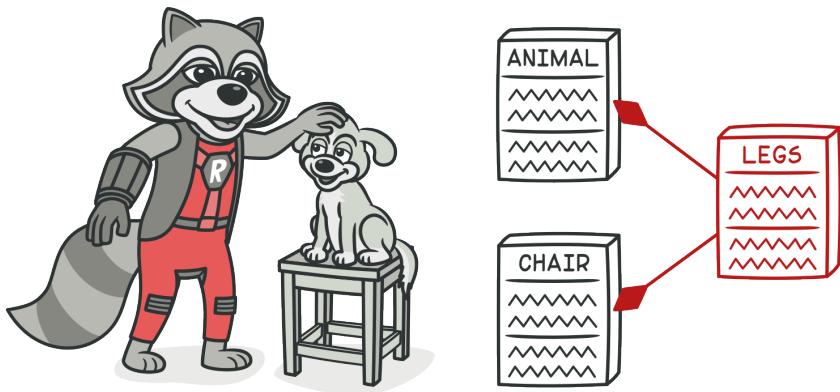
## Reasons for the Problem

Someone was motivated to create inheritance between classes only by the desire to reuse the code in a superclass. But the superclass and subclass are completely different.



## Treatment

- If inheritance makes no sense and the subclass really does have nothing in common with the superclass, eliminate inheritance in favor of **Replace Inheritance with Delegation**.
- If inheritance is appropriate, get rid of unneeded fields and methods in the subclass. Extract all fields and methods needed by the subclass from the parent class, put them in a new subclass, and set both classes to inherit from it (**Extract Super-class**).



## Payoff

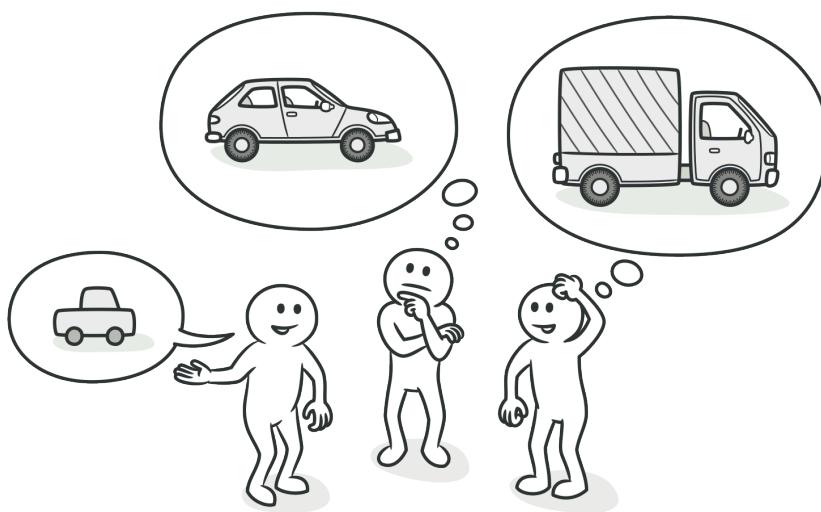
Improves code clarity and organization. You will no longer have to wonder why the `Dog` class is inherited from the `Chair` class (even though they both have 4 legs).



# Alternative Classes with Different Interfaces

## Signs and Symptoms

Two classes perform identical functions but have different method names.



## Reasons for the Problem

The programmer who created one of the classes probably didn't know that a functionally equivalent class already existed.

## Treatment

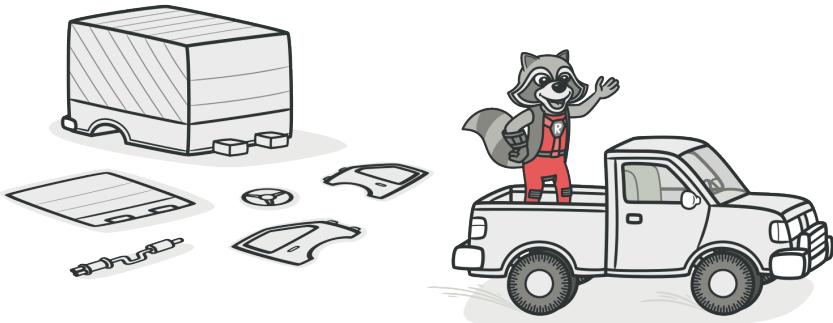
Try to put the interface of classes in terms of a common denominator:

- Rename Methods to make them identical in all alternative classes.
- Move Method, Add Parameter and Parameterize Method to make the signature and implementation of methods the same.
- If only part of the functionality of the classes is duplicated, try using Extract Superclass. In this case, the existing classes will become subclasses.
- After you have determined which treatment method to use and implemented it, you may be able to delete one of the classes.

## Payoff

- You get rid of unnecessary duplicated code, making the resulting code less bulky.

- Code becomes more readable and understandable (you no longer have to guess the reason for creation of a second class performing the exact same functions as the first one).



## When to Ignore

Sometimes merging classes is impossible or so difficult as to be pointless. One example is when the alternative classes are in different libraries that each have their own version of the class.

# Change Preventers

These smells mean that if you need to change something in one place in your code, you have to make many changes in other places too. Program development becomes much more complicated and expensive as a result.

## § **Divergent Change**

You find yourself having to change many unrelated methods when you make changes to a class. For example, when adding a new product type you have to change the methods for finding, displaying, and ordering products.

## § **Shotgun Surgery**

Making any modifications requires that you make many small changes to many different classes.

## § **Parallel Inheritance Hierarchies**

Whenever you create a subclass for a class, you find yourself needing to create a subclass for another class.

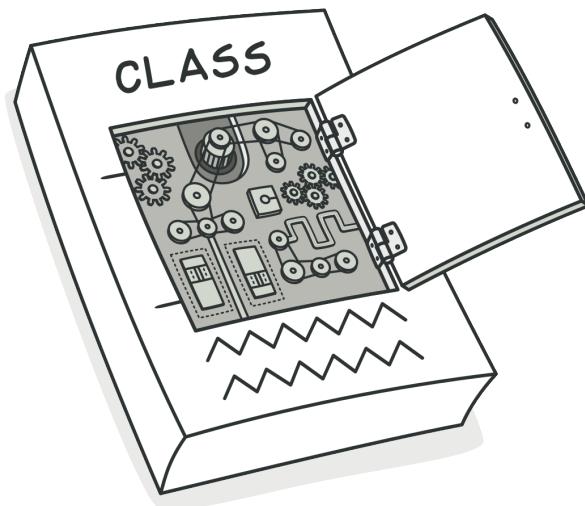


# Divergent Change

*Divergent Change* resembles **Shotgun Surgery** but is actually the opposite smell. *Divergent Change* is when many changes are made to a single class. *Shotgun Surgery* refers to when a single change is made to multiple classes simultaneously.

## Signs and Symptoms

You find yourself having to change many unrelated methods when you make changes to a class. For example, when adding a new product type you have to change the methods for finding, displaying, and ordering products.



## Reasons for the Problem

Often these divergent modifications are due to poor program structure or "copypasta programming".

## Treatment

- Split up the behavior of the class via [Extract Class](#).
- If different classes have the same behavior, you may want to combine the classes through inheritance ([Extract Superclass](#) and [Extract Subclass](#)).



## Payoff

- Improves code organization.
- Reduces code duplication.

- **Simplifies support.**

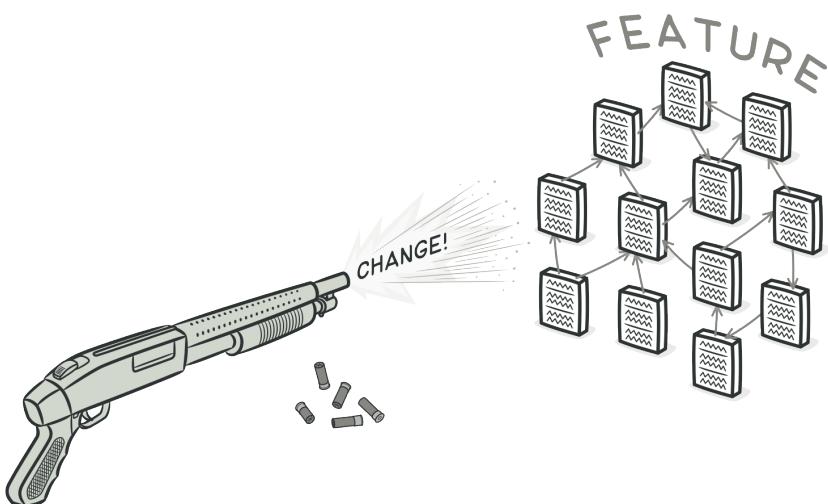


# Shotgun Surgery

*Shotgun Surgery* resembles **Divergent Change** but is actually the opposite smell. *Divergent Change* is when many changes are made to a single class. *Shotgun Surgery* refers to when a single change is made to multiple classes simultaneously.

## Signs and Symptoms

Making any modifications requires that you make many small changes to many different classes.



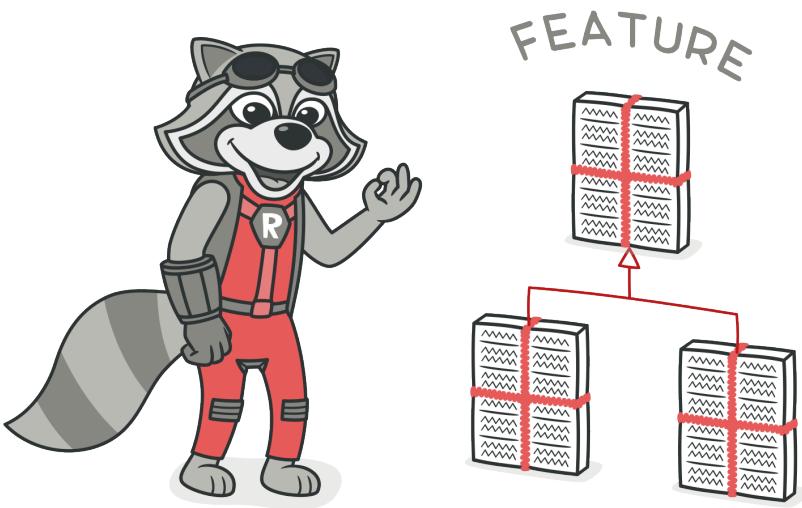
## Reasons for the Problem

A single responsibility has been split up among a large number of classes. This can happen after overzealous application of Divergent Change.



## Treatment

- Use Move Method and Move Field to move existing class behaviors into a single class. If there's no class appropriate for this, create a new one.
- If moving code to the same class leaves the original classes almost empty, try to get rid of these now-redundant classes via Inline Class.



## Payoff

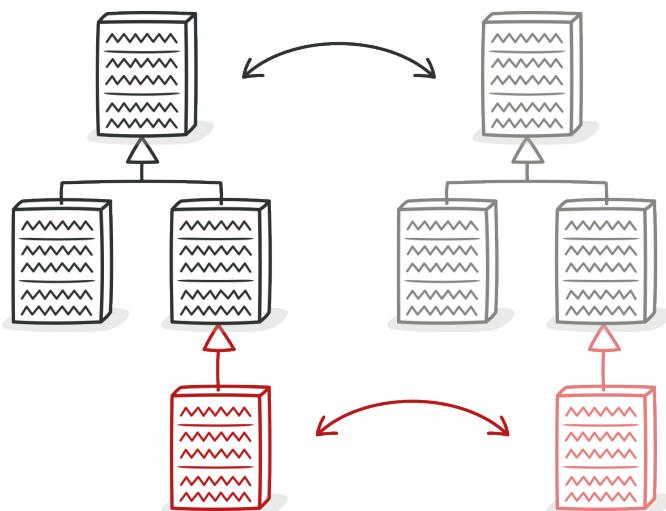
- Better organization.
- Less code duplication.
- Easier maintenance.



# Parallel Inheritance Hierarchies

## Signs and Symptoms

Whenever you create a subclass for a class, you find yourself needing to create a subclass for another class.



## Reasons for the Problem

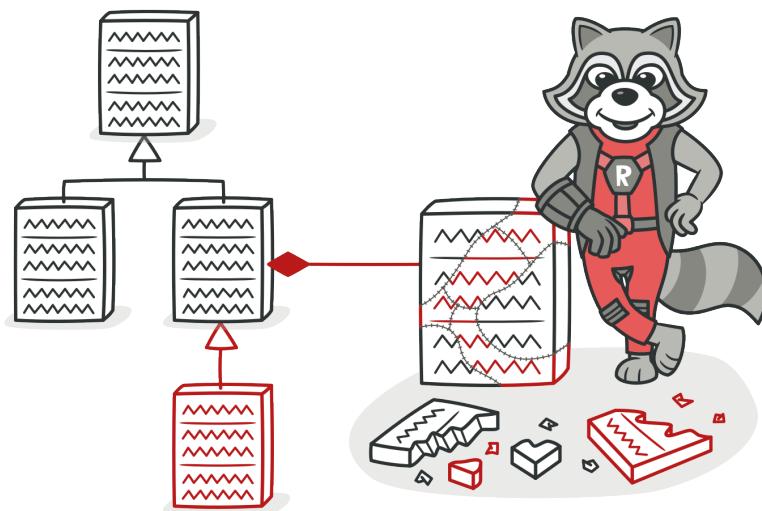
All was well as long as the hierarchy stayed small. But with new classes being added, making changes has become harder and harder.

## Treatment

You may de-duplicate parallel class hierarchies in two steps. First, make instances of one hierarchy refer to instances of another hierarchy. Then, remove the hierarchy in the referred class, by using **Move Method** and **Move Field**.

## Payoff

- Reduces code duplication.
- Can improve organization of code.



## When to Ignore

Sometimes having parallel class hierarchies is just a way to avoid even bigger mess with program architecture. If you find that your attempts to de-duplicate hierarchies produce even

uglier code, just step out, revert all of your changes and get used to that code.

# Dispensables

A dispensable is something pointless and unneeded whose absence would make the code cleaner, more efficient and easier to understand.

## § Comments

A method is filled with explanatory comments.

## § Duplicate Code

Two code fragments look almost identical.

## § Lazy Class

Understanding and maintaining classes always costs time and money. So if a class doesn't do enough to earn your attention, it should be deleted.

## § Data Class

A data class refers to a class that contains only fields and crude methods for accessing them (getters and setters). These are simply containers for data used by other classes. These classes don't contain any additional functionality and can't independently operate on the data that they own.

## § Dead Code

A variable, parameter, field, method or class is no longer used (usually because it's obsolete).

## § **Speculative Generality**

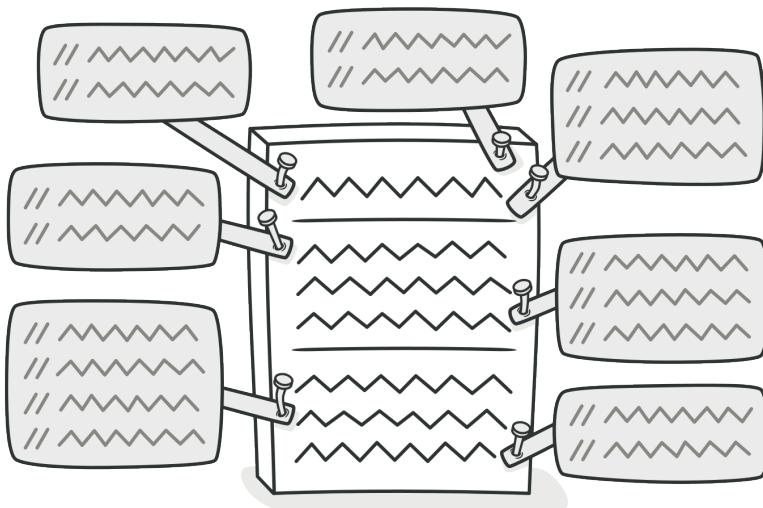
| There's an unused class, method, field or parameter.



# Comments

## Signs and Symptoms

A method is filled with explanatory comments.



## Reasons for the Problem

Comments are usually created with the best of intentions, when the author realizes that his or her code isn't intuitive or obvious. In such cases, comments are like a deodorant masking the smell of fishy code that could be improved.

The best comment is a good name for a method or class.

If you feel that a code fragment can't be understood without comments, try to change the code structure in a way that makes comments unnecessary.

## Treatment

- If a comment is intended to explain a complex expression, the expression should be split into understandable subexpressions using **Extract Variable**.
- If a comment explains a section of code, this section can be turned into a separate method via **Extract Method**. The name of the new method can be taken from the comment text itself, most likely.
- If a method has already been extracted, but comments are still necessary to explain what the method does, give the method a self-explanatory name. Use **Rename Method** for this.
- If you need to assert rules about a state that's necessary for the system to work, use **Introduce Assertion**.

## Payoff

Code becomes more intuitive and obvious.



## When to Ignore

Comments can sometimes be useful:

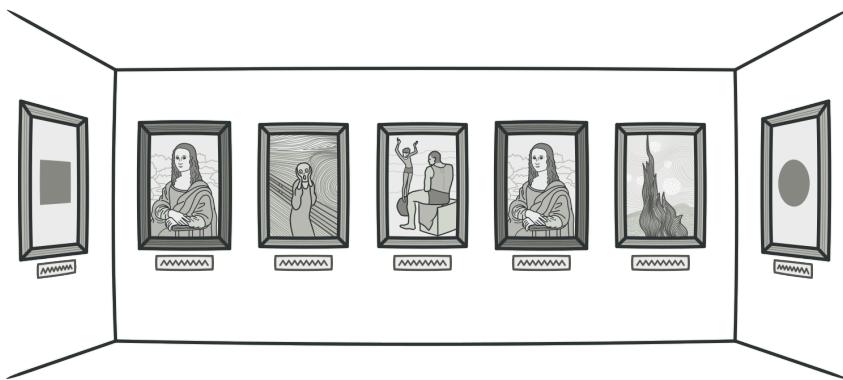
- When explaining **why** something is being implemented in a particular way.
- When explaining complex algorithms (when all other methods for simplifying the algorithm have been tried and come up short).



# Duplicate Code

## Signs and Symptoms

Two code fragments look almost identical.



## Reasons for the Problem

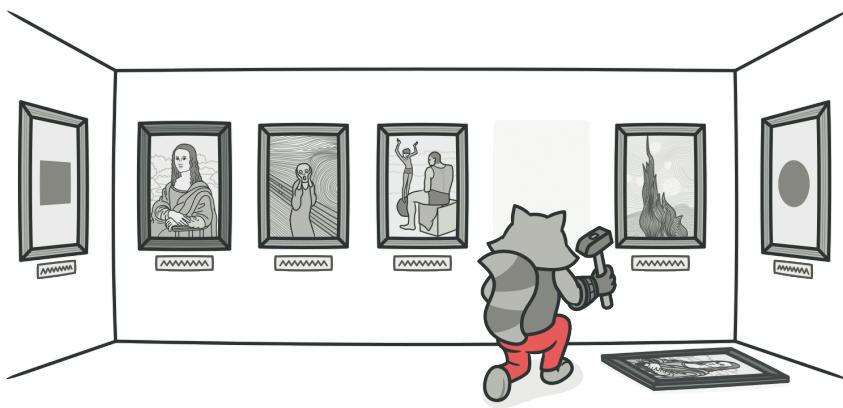
Duplication usually occurs when multiple programmers are working on different parts of the same program at the same time. Since they're working on different tasks, they may be unaware their colleague has already written similar code that could be repurposed for their own needs.

There's also more subtle duplication, when specific parts of code look different but actually perform the same job. This kind of duplication can be hard to find and fix.

Sometimes duplication is purposeful. When rushing to meet deadlines and the existing code is “almost right” for the job, novice programmers may not be able to resist the temptation of copying and pasting the relevant code. And in some cases, the programmer is simply too lazy to de-clutter.

## Treatment

- If the same code is found in two or more methods in the same class: use **Extract Method** and place calls for the new method in both places.



- If the same code is found in two subclasses of the same level:

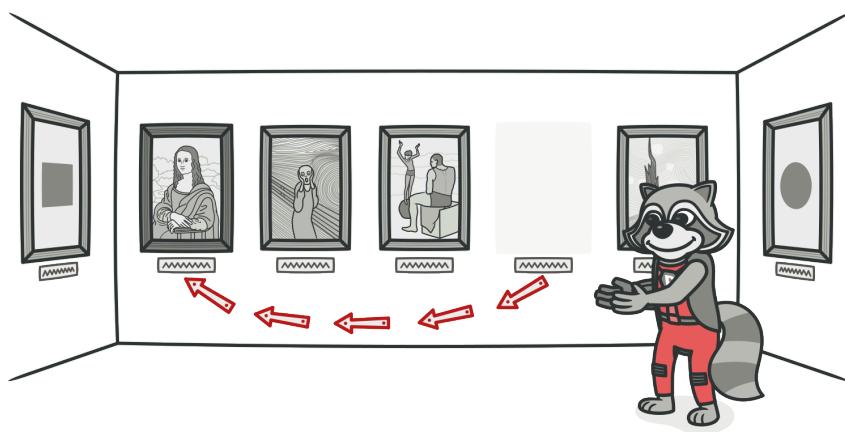
- Use **Extract Method** for both classes, followed by **Pull Up Field** for the fields used in the method that you're pulling up.
  - If the duplicate code is inside a constructor, use **Pull Up Constructor Body**.
  - If the duplicate code is similar but not completely identical, use **Form Template Method**.
  - If two methods do the same thing but use different algorithms, select the best algorithm and apply **Substitute Algorithm**.
- If duplicate code is found in two different classes:
    - If the classes aren't part of a hierarchy, use **Extract Super-class** in order to create a single superclass for these classes that maintains all the previous functionality.
    - If it's difficult or impossible to create a superclass, use **Extract Class** in one class and use the new component in the other.
  - If a large number of conditional expressions are present and perform the same code (differing only in their conditions), merge these operators into a single condition using **Consolidate Conditional Expression** and use **Extract Method** to place

the condition in a separate method with an easy-to-understand name.

- If the same code is performed in all branches of a conditional expression: place the identical code outside of the condition tree by using **Consolidate Duplicate Conditional Fragments**.

## Payoff

- Merging duplicate code simplifies the structure of your code and makes it shorter.
- Simplification + shortness = code that's easier to simplify and cheaper to support.



## When to Ignore

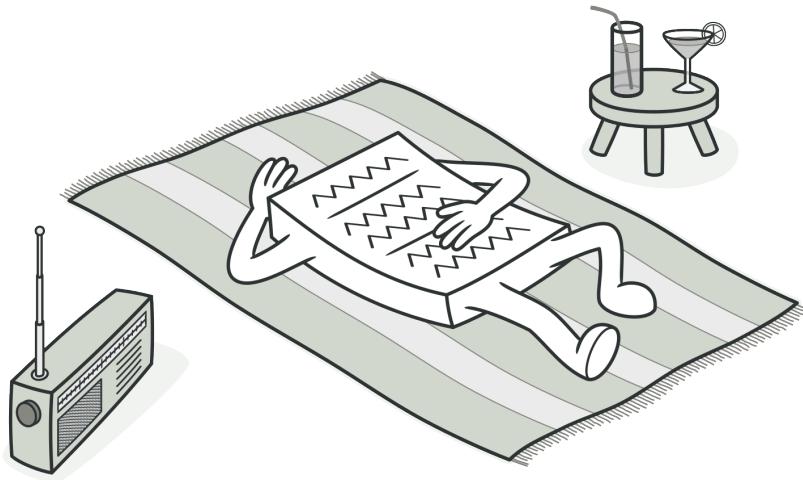
In very rare cases, merging two identical fragments of code can make the code less intuitive and obvious.



# Lazy Class

## Signs and Symptoms

Understanding and maintaining classes always costs time and money. So if a class doesn't do enough to earn your attention, it should be deleted.



## Reasons for the Problem

Perhaps a class was designed to be fully functional but after some of the refactoring it has become ridiculously small.

Or perhaps it was designed to support future development work that never got done.

## Treatment

- Components that are near-useless should be given the Inline Class treatment.
- For subclasses with few functions, try Collapse Hierarchy.



## Payoff

- Reduced code size.
- Easier maintenance.

## When to Ignore

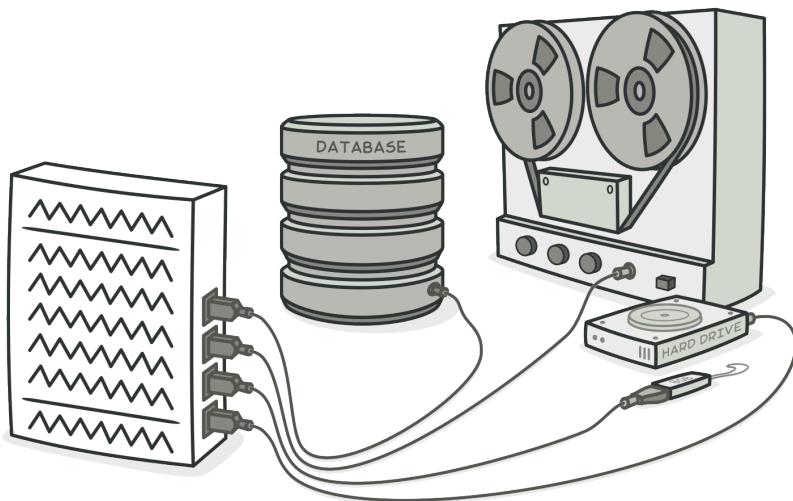
Sometimes a *Lazy Class* is created in order to delineate intentions for future development. In this case, try to maintain a balance between clarity and simplicity in your code.



# Data Class

## Signs and Symptoms

A data class refers to a class that contains only fields and crude methods for accessing them (getters and setters). These are simply containers for data used by other classes. These classes don't contain any additional functionality and can't independently operate on the data that they own.



## Reasons for the Problem

It's a normal thing when a newly created class contains only a few public fields (and maybe even a handful of getters/setters). But the true power of objects is that they can contain behavior types or operations on their data.

## Treatment

- If a class contains public fields, use **Encapsulate Field** to hide them from direct access and require that access be performed via getters and setters only.
- Use **Encapsulate Collection** for data stored in collections (such as arrays).
- Review the client code that uses the class. In it, you may find functionality that would be better located in the data class itself. If this is the case, use **Move Method** and **Extract Method** to migrate this functionality to the data class.
- After the class has been filled with well thought-out methods, you may want to get rid of old methods for data access that give overly broad access to the class data. For this, **Remove Setting Method** and **Hide Method** may be helpful.



## Payoff

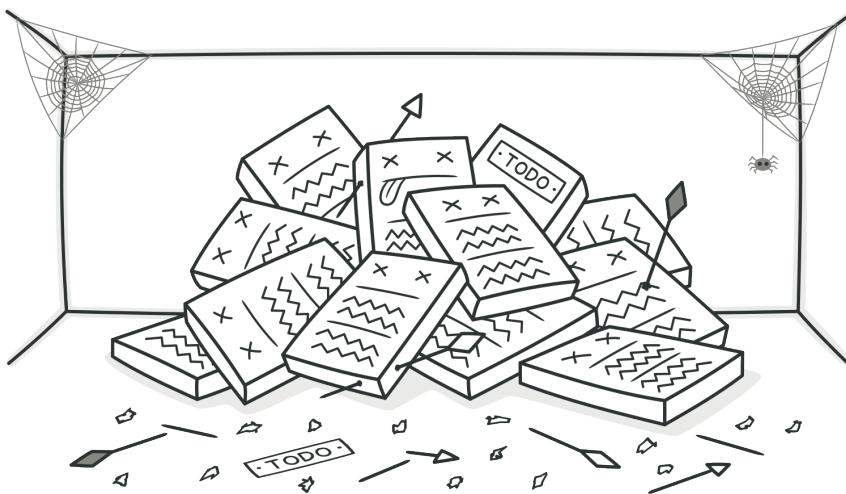
- Improves understanding and organization of code. Operations on particular data are now gathered in a single place, instead of haphazardly throughout the code.
- Helps you to spot duplication of client code.



# Dead Code

## Signs and Symptoms

A variable, parameter, field, method or class is no longer used (usually because it's obsolete).



## Reasons for the Problem

When requirements for the software have changed or corrections have been made, nobody had time to clean up the old code.

Such code could also be found in complex conditionals, when one of the branches becomes unreachable (due to error or other circumstances).

## Treatment

The quickest way to find dead code is to use a good [IDE](#).

- Delete unused code and unneeded files.
- In the case of an unnecessary class, [Inline Class](#) or [Collapse Hierarchy](#) can be applied if a subclass or superclass is used.
- To remove unneeded parameters, use [Remove Parameter](#).



## Payoff

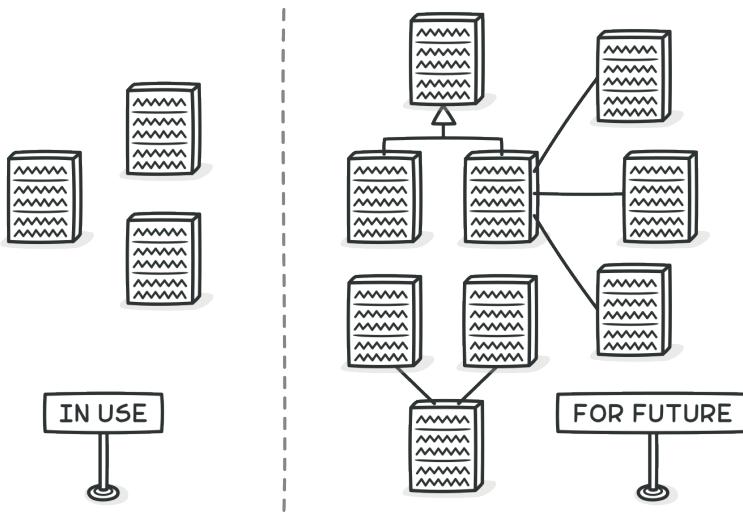
- Reduced code size.
- Simpler support.



# Speculative Generality

## Signs and Symptoms

There's an unused class, method, field or parameter.



## Reasons for the Problem

Sometimes code is created “just in case” to support anticipated future features that never get implemented. As a result, code becomes hard to understand and support.

## Treatment

- For removing unused abstract classes, try [Collapse Hierarchy](#).
- Unnecessary delegation of functionality to another class can be eliminated via [Inline Class](#).
- Unused methods? Use [Inline Method](#) to get rid of them.
- Methods with unused parameters should be given a look with the help of [Remove Parameter](#).
- Unused fields can be simply deleted.



## Payoff

- Slimmer code.

- Easier support.

## When to Ignore

- If you're working on a framework, it's eminently reasonable to create functionality not used in the framework itself, as long as the functionality is needed by the framework's users.
- Before deleting elements, make sure that they aren't used in unit tests. This happens if tests need a way to get certain internal information from a class or perform special testing-related actions.

# Couplers

All the smells in this group contribute to excessive coupling between classes or show what happens if coupling is replaced by excessive delegation.

## § Feature Envy

A method accesses the data of another object more than its own data.

## § Inappropriate Intimacy

One class uses the internal fields and methods of another class.

## § Message Chains

In code you see a series of calls resembling

```
$a->b()->c()->d()
```

## § Middle Man

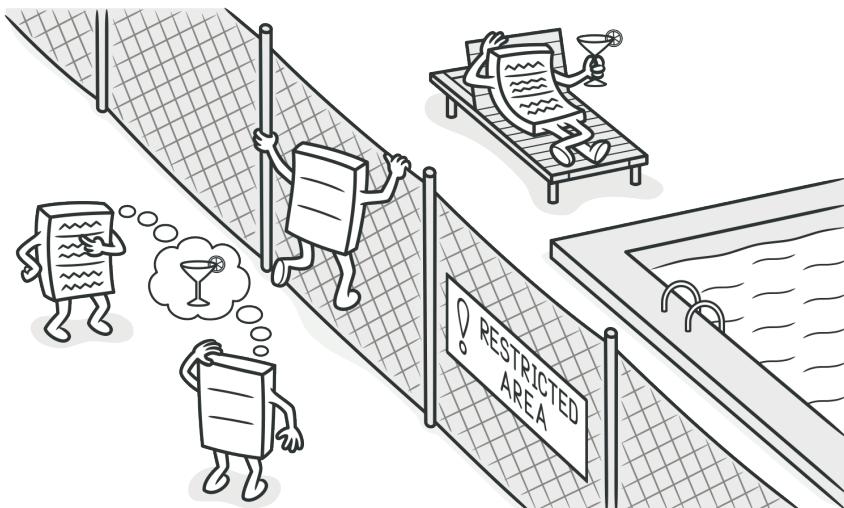
If a class performs only one action, delegating work to another class, why does it exist at all?



# Feature Envy

## Signs and Symptoms

A method accesses the data of another object more than its own data.



## Reasons for the Problem

This smell may occur after fields are moved to a data class. If this is the case, you may want to move the operations on data to this class as well.

## Treatment

As a basic rule, if things change at the same time, you should keep them in the same place. Usually data and functions that use this data are changed together (although exceptions are possible).

- If a method clearly should be moved to another place, use **Move Method**.
- If only part of a method accesses the data of another object, use **Extract Method** to move the part in question.
- If a method uses functions from several other classes, first determine which class contains most of the data used. Then place the method in this class along with the other data. Alternatively, use **Extract Method** to split the method into several parts that can be placed in different places in different classes.



## Payoff

- Less code duplication (if the data handling code is put in a central place).
- Better code organization (methods for handling data are next to the actual data).



## When to Ignore

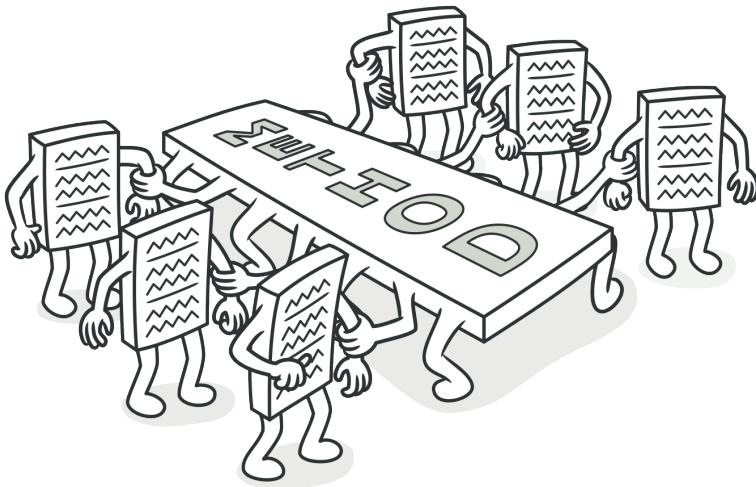
Sometimes behavior is purposefully kept separate from the class that holds the data. The usual advantage of this is the ability to dynamically change the behavior (see Strategy, Visitor and other patterns).



# Inappropriate Intimacy

## Signs and Symptoms

One class uses the internal fields and methods of another class.



## Reasons for the Problem

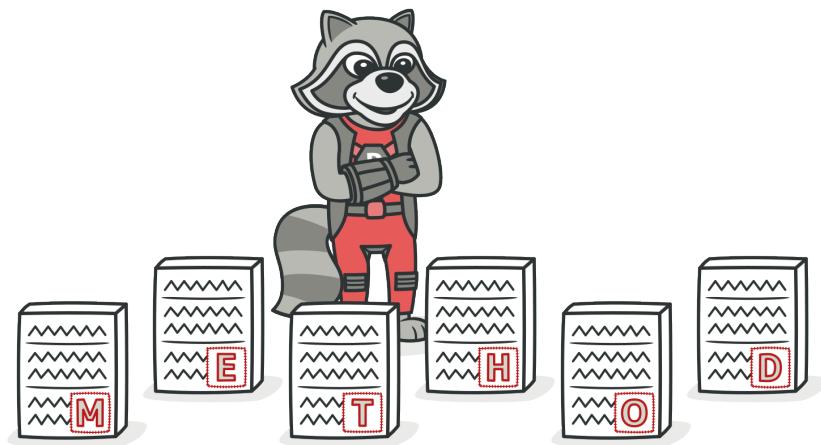
Keep a close eye on classes that spend too much time together. Good classes should know as little about each other as possible. Such classes are easier to maintain and reuse.

## Treatment

- The simplest solution is to use **Move Method** and **Move Field** to move parts of one class to the class in which those parts are used. But this works only if the first class truly doesn't need these parts.



- Another solution is to use **Extract Class** and **Hide Delegate** on the class to make the code relations “official”.
- If the classes are mutually interdependent, you should use **Change Bidirectional Association to Unidirectional**.
- If this “intimacy” is between a subclass and the superclass, consider **Replace Delegation with Inheritance**.



## Payoff

- Improves code organization.
- Simplifies support and code reuse.

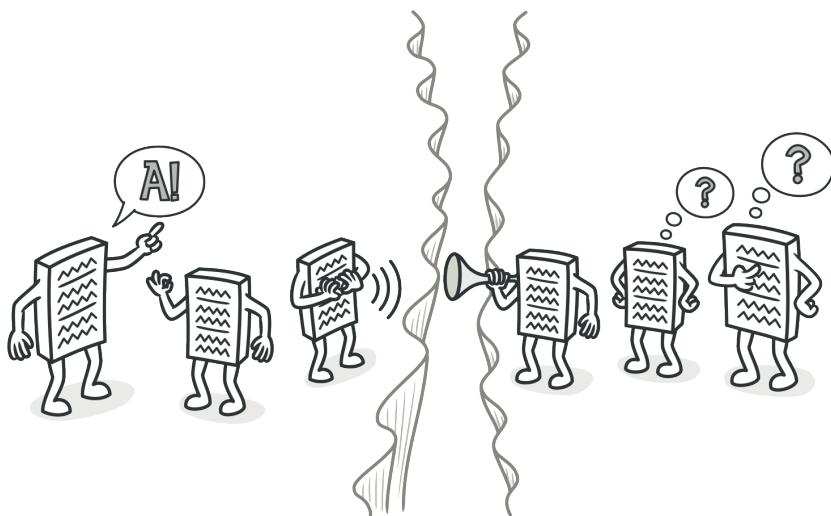


# Message Chains

## Signs and Symptoms

In code you see a series of calls resembling

```
$a->b()->c()->d()
```



## Reasons for the Problem

A message chain occurs when a client requests another object, that object requests yet another one, and so on. These chains mean that the client is dependent on navigation along the class structure. Any changes in these relationships require modifying the client.

## Treatment

- To delete a message chain, use **Hide Delegate**.
- Sometimes it's better to think of why the end object is being used. Perhaps it would make sense to use **Extract Method** for this functionality and move it to the beginning of the chain, by using **Move Method**.



## Payoff

- Reduces dependencies between classes of a chain.
- Reduces the amount of bloated code.



## When to Ignore

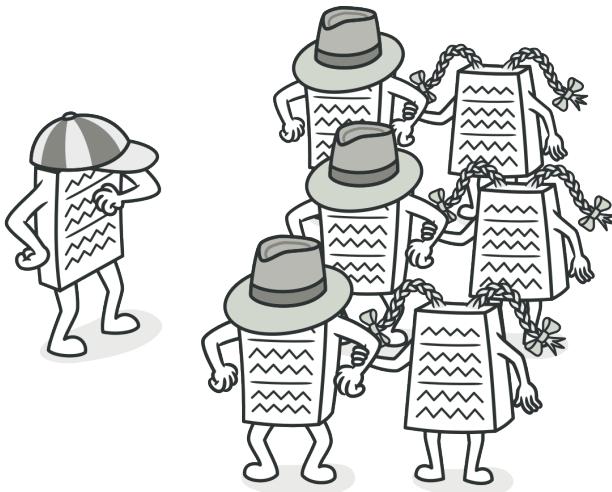
Overly aggressive delegate hiding can cause code in which it's hard to see where the functionality is actually occurring. Which is another way of saying, avoid the Middle Man smell as well.



# Middle Man

## Signs and Symptoms

If a class performs only one action, delegating work to another class, why does it exist at all?



## Reasons for the Problem

This smell can be the result of overzealous elimination of **Message Chains**.

In other cases, it can be the result of the useful work of a class being gradually moved to other classes. The class remains as an empty shell that doesn't do anything other than delegate.

## Treatment

If most of a method's classes delegate to another class, Remove Middle Man is in order.

## Payoff

Less bulky code.



## When to Ignore

Don't delete middle man that have been created for a reason:

- A middle man may have been added to avoid interclass dependencies.
- Some design patterns create middle man on purpose (such as Proxy or Decorator).

# Other Smells

Below are the smells which don't fall into any broad category.

## § Incomplete Library Class

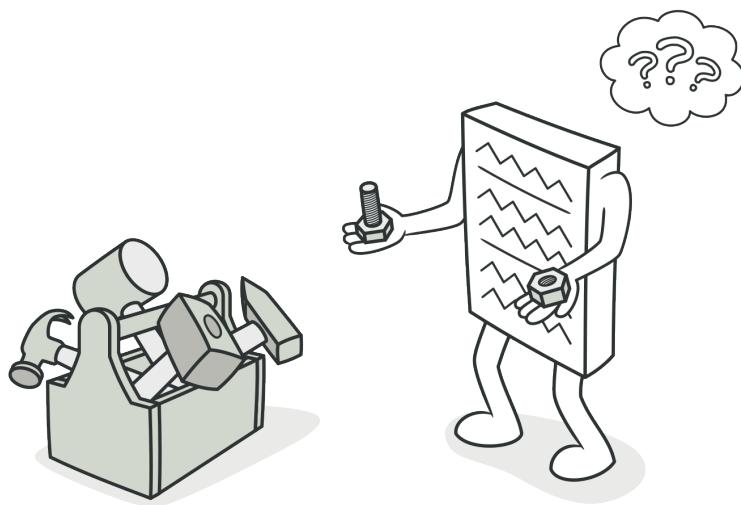
Sooner or later, **libraries** stop meeting user needs. The only solution to the problem – changing the library – is often impossible since the library is read-only.



# Incomplete Library Class

## Signs and Symptoms

Sooner or later, **libraries** stop meeting user needs. The only solution to the problem – changing the library – is often impossible since the library is read-only.



## Reasons for the Problem

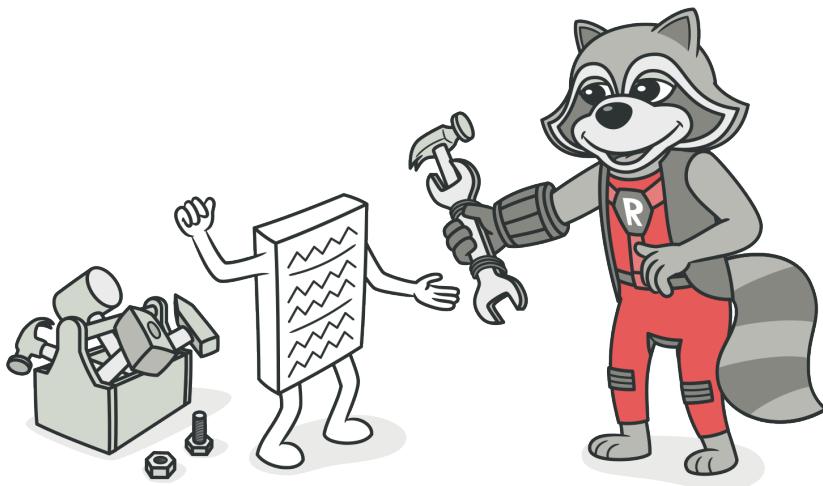
The author of the library hasn't provided the features you need or has refused to implement them.

## Treatment

- To introduce a few methods to a library class, use [Introduce Foreign Method](#).
- For big changes in a class library, use [Introduce Local Extension](#).

## Payoff

Reduces code duplication (instead of creating your own library from scratch, you can still piggy-back off an existing one).



## When to Ignore

Extending a library can generate additional work if the changes to the library involve changes in code.

# Refactoring Techniques

Refactoring is a controllable process of improving code without creating new functionality. It transforms a mess into clean code and simple design.



Clean code is code that is easy to read, write and maintain. Clean code makes software development predictable and increases the quality of a resulting product.

Refactoring techniques describe actual refactoring steps. Most refactoring techniques have their pros and cons. Therefore, each refactoring should be properly motivated and applied with caution.

In the previous chapters, you have seen how particular refactorings can help fixing problems with code. Now it's the time to look over the refactoring techniques in more detail!

# 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

**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

**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

**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

**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 (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

**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

**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

**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.



# Extract Method

## Problem

You have a code fragment that can be grouped together.

```
1 void printOwing() {  
2     printBanner();  
3  
4     //print details  
5     System.out.println("name: " + name);  
6     System.out.println("amount: " + getOutstanding());  
7 }
```

## Solution

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

```
1 void printOwing() {  
2     printBanner();  
3     printDetails(getOutstanding());  
4 }  
5  
6 void printDetails(double outstanding) {  
7     System.out.println("name: " + name);  
8     System.out.println("amount: " + outstanding);
```

```
9 }
```

## Why Refactor

The more lines found in a method, the harder it's to figure out what the method does. This is the main reason for this refactoring.

Besides eliminating rough edges in your code, extracting methods is also a step in many other refactoring approaches.

## Benefits

- More readable code! Be sure to give the new method a name that describes the method's purpose: `createOrder()`, `renderCustomerInfo()`, etc.
- Less code duplication. Often the code that's found in a method can be reused in other places in your program. So you can replace duplicates with calls to your new method.
- Isolates independent parts of code, meaning that errors are less likely (such as if the wrong variable is modified).

## How to Refactor

1. Create a new method and name it in a way that makes its purpose self-evident.

2. Copy the relevant code fragment to your new method. Delete the fragment from its old location and put a call for the new method there instead.

Find all variables used in this code fragment. If they're declared inside the fragment and not used outside of it, simply leave them unchanged – they'll become local variables for the new method.

3. If the variables are declared prior to the code that you're extracting, you will need to pass these variables to the parameters of your new method in order to use the values previously contained in them. Sometimes it's easier to get rid of these variables by resorting to **Replace Temp with Query**.
4. If you see that a local variable changes in your extracted code in some way, this may mean that this changed value will be needed later in your main method. Double-check! And if this is indeed the case, return the value of this variable to the main method to keep everything functioning.

## Anti-refactoring

- § Inline Method

## Similar refactorings

- § Move Method

## Helps other refactorings

- § Introduce Parameter Object
- § Form Template Method
- § Introduce Parameter Object
- § Parameterize Method

## Eliminates smell

- § Duplicate Code
- § Long Method
- § Feature Envy
- § Switch Statements
- § Message Chains
- § Comments
- § Data Class



# Inline Method

## Problem

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

```
1 class PizzaDelivery {  
2     //...  
3     int getRating() {  
4         return moreThanFiveLateDeliveries() ? 2 : 1;  
5     }  
6     boolean moreThanFiveLateDeliveries() {  
7         return numberOfLateDeliveries > 5;  
8     }  
9 }
```

## Solution

Replace calls to the method with the method's content and delete the method itself.

```
1 class PizzaDelivery {  
2     //...  
3     int getRating() {  
4         return numberOfLateDeliveries > 5 ? 2 : 1;
```

```
5      }
6  }
```

## Why Refactor

A method simply delegates to another method. In itself, this delegation is no problem. But when there are many such methods, they become a confusing tangle that's hard to sort through.

Often methods aren't too short *originally*, but become that way as changes are made to the program. So don't be shy about getting rid of methods that have outlived their use.

## Benefits

By minimizing the number of unneeded methods, you make the code more straightforward.

## How to Refactor

1. Make sure that the method isn't redefined in subclasses. If the method is redefined, refrain from this technique.
2. Find all calls to the method. Replace these calls with the content of the method.
3. Delete the method.

## Anti-refactoring

### § Extract Method

**Eliminates smell**

### § Speculative Generality



# Extract Variable

## Problem

You have an expression that's hard to understand.

```
1 void renderBanner() {  
2     if ((platform.toUpperCase().indexOf("MAC") > -1) &&  
3         (browser.toUpperCase().indexOf("IE") > -1) &&  
4         wasInitialized() && resize > 0 )  
5     {  
6         // do something  
7     }  
8 }
```

## Solution

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

```
1 void renderBanner() {  
2     final boolean isMacOs = platform.toUpperCase().indexOf("MAC") > -1;  
3     final boolean isIE = browser.toUpperCase().indexOf("IE") > -1;  
4     final boolean wasResized = resize > 0;  
5  
6     if (isMacOs && isIE && wasInitialized() && wasResized) {  
7         // do something  
8 }
```

```
8     }
9 }
```

## Why Refactor

The main reason for extracting variables is to make a complex expression more understandable, by dividing it into its intermediate parts. These could be:

- Condition of the `if()` operator or a part of the `?:` operator in C-based languages
- A long arithmetic expression without intermediate results
- Long multipart lines

Extracting a variable may be the first step towards performing **Extract Method** if you see that the extracted expression is used in other places in your code.

## Benefits

More readable code! Try to give the extracted variables good names that announce the variable's purpose loud and clear. More readability, fewer long-winded comments. Go for names like `customerTaxValue` , `cityUnemploymentRate` , `clientSalutationString` , etc.

## Drawbacks

More variables are present in your code. But this is counterbalanced by the ease of reading your code.

## How to Refactor

1. Insert a new line before the relevant expression and declare a new variable there. Assign part of the complex expression to this variable.
2. Replace that part of the expression with the new variable.
3. Repeat the process for all complex parts of the expression.

## Anti-refactoring

### § Inline Temp

## Similar refactorings

### § Extract Method

## Eliminates smell

### § Comments



# Inline Temp

## Problem

You have a temporary variable that's assigned the result of a simple expression and nothing more.

```
1 boolean hasDiscount(Order order) {  
2     double basePrice = order.basePrice();  
3     return basePrice > 1000;  
4 }
```

## Solution

Replace the references to the variable with the expression itself.

```
1 boolean hasDiscount(Order order) {  
2     return order.basePrice() > 1000;  
3 }
```

## Why Refactor

Inline local variables are almost always used as part of Replace Temp with Query or to pave the way for Extract Method.

## Benefits

This refactoring technique offers almost no benefit in and of itself. However, if the variable is assigned the result of a method, you can marginally improve the readability of the program by getting rid of the unnecessary variable.

## Drawbacks

Sometimes seemingly useless temps are used to cache the result of an expensive operation that's reused several times. So before using this refactoring technique, make sure that simplicity won't come at the cost of performance.

## How to Refactor

1. Find all places that use the variable. Instead of the variable, use the expression that had been assigned to it.
2. Delete the declaration of the variable and its assignment line.

## Helps other refactorings

§ [Replace Temp with Query](#)

§ [Extract Method](#)



# Replace Temp with Query

## Problem

You place the result of an expression in a local variable for later use in your code.

```
1 double calculateTotal() {  
2     double basePrice = quantity * itemPrice;  
3     if (basePrice > 1000) {  
4         return basePrice * 0.95;  
5     }  
6     else {  
7         return basePrice * 0.98;  
8     }  
9 }
```

## 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.

```
1 double calculateTotal() {  
2     if (basePrice() > 1000) {  
3         return basePrice() * 0.95;  
4     }  
5     else {  
6         return basePrice() * 0.98;  
7     }  
8 }  
9 double basePrice() {  
10    return quantity * itemPrice;  
11 }
```

## Why Refactor

This refactoring can lay the groundwork for applying [Extract Method](#) for a portion of a very long method.

The same expression may sometimes be found in other methods as well, which is one reason to consider creating a common method.

## Benefits

- Code readability. It's much easier to understand the purpose of the method `getTax()` than the line `orderPrice() * 0.2`.
- Slimmer code via deduplication, if the line being replaced is used in multiple methods.

## Good to Know

### Performance

This refactoring may prompt the question of whether this approach is liable to cause a performance hit. The honest answer is: yes, it's, since the resulting code may be burdened by querying a new method. But with today's fast CPUs and excellent compilers, the burden will almost always be minimal. By contrast, readable code and the ability to reuse this method in other places in program code – thanks to this refactoring approach – are very noticeable benefits.

Nonetheless, if your temp variable is used to cache the result of a truly time-consuming expression, you may want to stop this refactoring after extracting the expression to a new method.

## How to Refactor

1. Make sure that a value is assigned to the variable once and only once within the method. If not, use **Split Temporary Variable** to ensure that the variable will be used only to store the result of your expression.
2. Use **Extract Method** to place the expression of interest in a new method. Make sure that this method only returns a value and doesn't change the state of the object. If the method

affects the visible state of the object, use **Separate Query from Modifier.**

3. Replace the variable with a query to your new method.

## Similar refactorings

§ **Extract Method**

### Eliminates smell

§ **Long Method**

§ **Duplicate Code**



# Split Temporary Variable

## Problem

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

```
1 double temp = 2 * (height + width);
2 System.out.println(temp);
3 temp = height * width;
4 System.out.println(temp);
```

## Solution

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

```
1 final double perimeter = 2 * (height + width);
2 System.out.println(perimeter);
3 final double area = height * width;
4 System.out.println(area);
```

## Why Refactor

If you're skimping on the number of variables inside a function and reusing them for various unrelated purposes, you're sure to encounter problems as soon as you need to make changes to the code containing the variables. You will have to recheck each case of variable use to make sure that the correct values are used.

## Benefits

- Each component of the program code should be responsible for one and one thing only. This makes it much easier to maintain the code, since you can easily replace any particular thing without fear of unintended effects.
- Code becomes more readable. If a variable was created long ago in a rush, it probably has a name that doesn't explain anything: `k` , `a2` , `value` , etc. But you can fix this situation by naming the new variables in an understandable, self-explanatory way. Such names might resemble `customerTaxValue` , `cityUnemploymentRate` , `clientSalutationString` and the like.
- This refactoring technique is useful if you anticipate using **Extract Method** later.

## How to Refactor

1. Find the first place in the code where the variable is given a value. Here you should rename the variable with a name that corresponds to the value being assigned.
2. Use the new name instead of the old one in places where this value of the variable is used.
3. Repeat as needed for places where the variable is assigned a different value.

## Anti-refactoring

§ Inline Temp

## Similar refactorings

§ Extract Variable

§ Remove Assignments to Parameters

## Helps other refactorings

§ Extract Method



# Remove Assignments to Parameters

## Problem

Some value is assigned to a parameter inside method's body.

```
1 int discount(int inputVal, int quantity) {  
2     if (inputVal > 50) {  
3         inputVal -= 2;  
4     }  
5     //...  
6 }
```

## Solution

Use a local variable instead of a parameter.

```
1 int discount(int inputVal, int quantity) {  
2     int result = inputVal;  
3     if (inputVal > 50) {  
4         result -= 2;
```

```
5      }
6  //...
7 }
```

## Why Refactor

The reasons for this refactoring are the same as for [Split Temporary Variable](#), but in this case we're dealing with a parameter, not a local variable.

First, if a parameter is passed via reference, then after the parameter value is changed inside the method, this value is passed to the argument that requested calling this method. Very often, this occurs accidentally and leads to unfortunate effects. Even if parameters are usually passed by value (and not by reference) in your programming language, this coding quirk may alienate those who are unaccustomed to it.

Second, multiple assignments of different values to a single parameter make it difficult for you to know what data should be contained in the parameter at any particular point in time. The problem worsens if your parameter and its contents are documented but the actual value is capable of differing from what's expected inside the method.

## Benefits

- Each element of the program should be responsible for only one thing. This makes code maintenance much easier going forward, since you can safely replace code without any side effects.
- This refactoring helps to extract **repetitive code to separate methods.**

## How to Refactor

1. Create a local variable and assign the initial value of your parameter.
2. In all method code that follows this line, replace the parameter with your new local variable.

## Similar refactorings

### § Split Temporary Variable

## Helps other refactorings

### § Extract Method



# 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.

```
1 class Order {  
2     //...  
3     public double price() {  
4         double primaryBasePrice;  
5         double secondaryBasePrice;  
6         double tertiaryBasePrice;  
7         // long computation.  
8         //...  
9     }  
10 }
```

## 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.

```
1  class Order {  
2      //...  
3      public double price() {  
4          return new PriceCalculator(this).compute();  
5      }  
6  }  
7  
8  class PriceCalculator {  
9      private double primaryBasePrice;  
10     private double secondaryBasePrice;  
11     private double tertiaryBasePrice;  
12  
13     public PriceCalculator(Order order) {  
14         // copy relevant information from order object.  
15         //...  
16     }  
17  
18     public double compute() {  
19         // long computation.  
20         //...  
21     }  
22 }
```

## 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.

## Similar refactorings

### § Replace Data Value with Object

Does the same with fields.

## Eliminates smell

### § Long Method



# Substitute Algorithm

## Problem

So you want to replace an existing algorithm with a new one?

```
1  String foundPerson(String[] people){  
2      for (int i = 0; i < people.length; i++) {  
3          if (people[i].equals("Don")){  
4              return "Don";  
5          }  
6          if (people[i].equals("John")){  
7              return "John";  
8          }  
9          if (people[i].equals("Kent")){  
10             return "Kent";  
11         }  
12     }  
13     return "";  
14 }
```

## Solution

Replace the body of the method that implements the algorithm with a new algorithm.

```
1 String foundPerson(String[] people){  
2     List candidates =  
3         Arrays.asList(new String[] {"Don", "John", "Kent"});  
4     for (int i=0; i < people.length; i++) {  
5         if (candidates.contains(people[i])) {  
6             return people[i];  
7         }  
8     }  
9     return "";  
10 }
```

## Why Refactor

1. Gradual refactoring isn't the only method for improving a program. Sometimes a method is so cluttered with issues that it's easier to tear down the method and start fresh. And perhaps you have found an algorithm that's much simpler and more efficient. If this is the case, you should simply replace the old algorithm with the new one.
2. As time goes on, your algorithm may be incorporated into a well-known library or framework and you want to get rid of your independent implementation, in order to simplify maintenance.
3. The requirements for your program may change so heavily that your existing algorithm can't be salvaged for the task.

## How to Refactor

1. Make sure that you have simplified the existing algorithm as much as possible. Move unimportant code to other methods using **Extract Method**. The fewer moving parts in your algorithm, the easier it's to replace.
2. Create your new algorithm in a new method. Replace the old algorithm with the new one and start testing the program.
3. If the results don't match, return to the old implementation and compare the results. Identify the causes of the discrepancy. While the cause is often an error in the old algorithm, it's more likely due to something not working in the new one.
4. When all tests are successfully completed, delete the old algorithm for good!

## Eliminates smell

§ Duplicate Code

§ Long Method

# 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

**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 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.

## § 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.

## § 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.

## § 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.

## § 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.

## § 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.

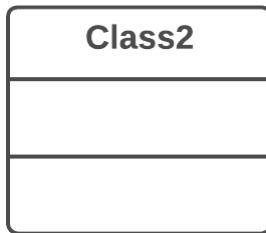
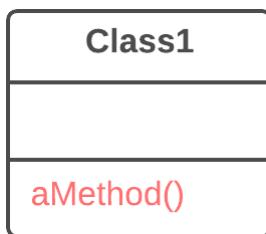
**Solution:** Create a new class containing the methods and make it either the child or wrapper of the utility class.



# Move Method

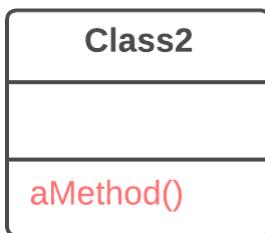
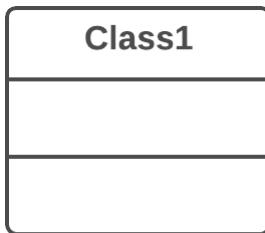
## Problem

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.

2. 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.
3. 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.

4. 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.

## Similar refactorings

§ Extract Method

§ Move Field

## Helps other refactorings

§ Extract Class

§ Inline Class

§ Introduce Parameter Object

## Eliminates smell

§ Shotgun Surgery

§ Feature Envy

§ Switch Statements

§ Parallel Inheritance Hierarchies

§ Message Chains

§ Inappropriate Intimacy

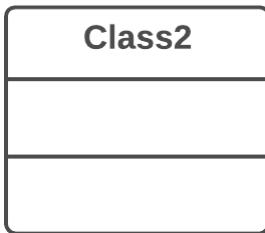
§ Data Class



# Move Field

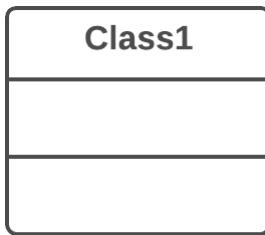
## Problem

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 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).

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.

## Similar refactorings

§ Move Field

## Helps other refactorings

§ Extract Class

§ Inline Class

## Eliminates smell

§ Shotgun Surgery

§ Parallel Inheritance Hierarchies

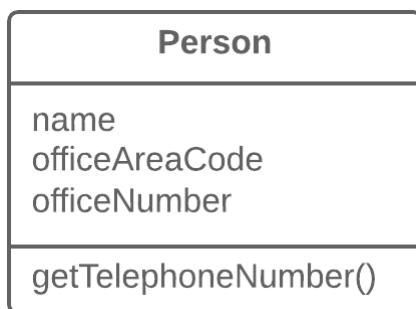
§ Inappropriate Intimacy



# Extract Class

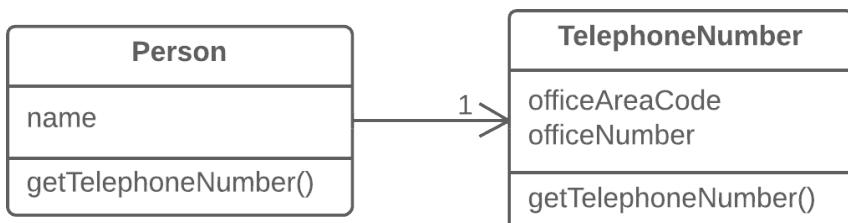
## Problem

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](#).

## 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** and **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.

4. 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.

## Anti-refactoring

### § Inline Class

## Similar refactorings

### § Extract Subclass

### § Replace Data Value with Object

## Eliminates smell

### § Duplicate Code

### § Large Class

### § Divergent Change

### § Data Clumps

### § Primitive Obsession

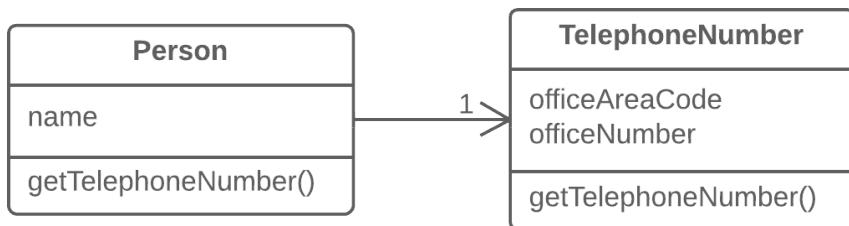
### § Temporary Field

### § Inappropriate Intimacy

# Inline Class

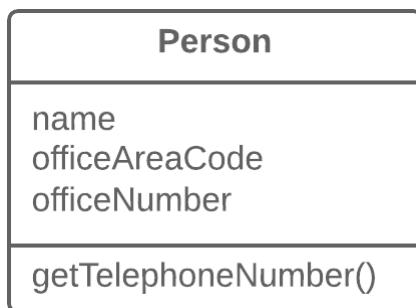
## Problem

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** and **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.

## Anti-refactoring

- § Extract Class

### Eliminates smell

- § Shotgun Surgery

- § Lazy Class

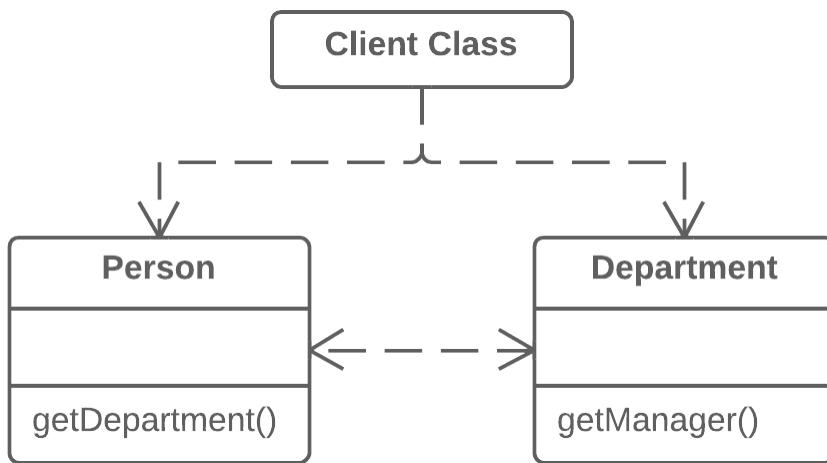
- § Speculative Generality



# Hide Delegate

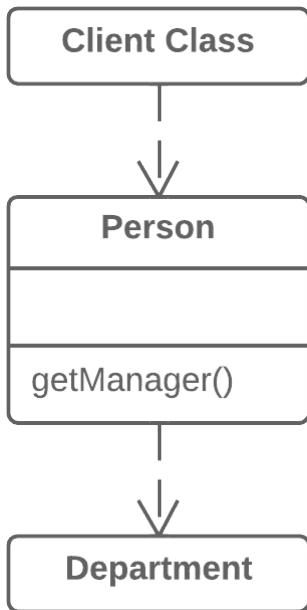
## Problem

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.



## 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.

## 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*).

## Anti-refactoring

### § Remove Middle Man

**Eliminates smell**

### § Message Chains

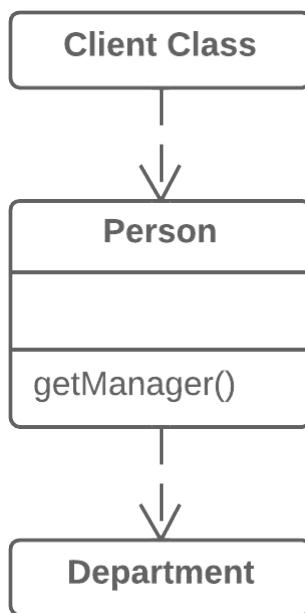
### § Inappropriate Intimacy



# Remove Middle Man

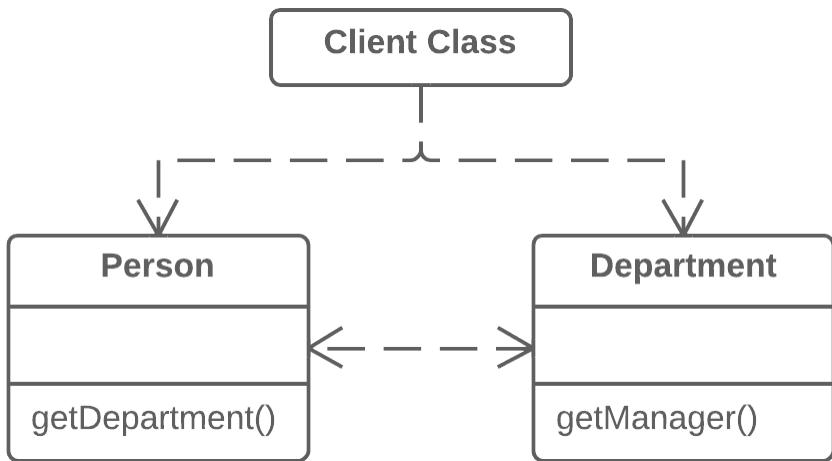
## Problem

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](#), 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*.

## Anti-refactoring

### § Hide Delegate

## Eliminates smell

### § Middle Man



# 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.

```
1 class Report {  
2     //...  
3     void sendReport() {  
4         Date nextDay = new Date(previousEnd.getYear(),  
5             previousEnd.getMonth(), previousEnd.getDate() + 1);  
6         //...  
7     }  
8 }
```

## Solution

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

```
1 class Report {  
2     //...
```

```
3 void sendReport() {  
4     Date newStart = nextDay(previousEnd);  
5     //...  
6 }  
7 private static Date nextDay(Date arg) {  
8     return new Date(arg.getYear(), arg.getMonth(), arg.getDate() + 1)  
9 }  
10 }
```

## 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** 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.

## Similar refactorings

### § Introduce Local Extension

Move all extension methods to a separate class, which is wrapper or a subclass of some service class.

## Eliminates smell

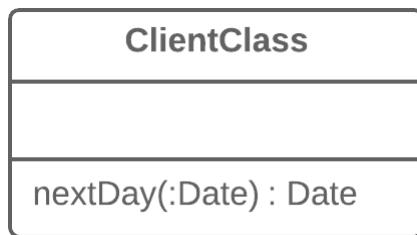
### § Incomplete Library Class



# Introduce Local Extension

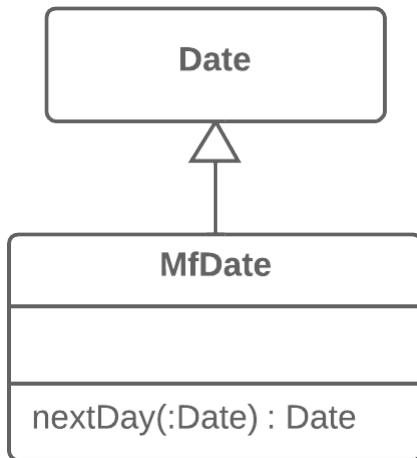
## Problem

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.

## Similar refactorings

### § Introduce Foreign Method

If you only want one special method, which doesn't exist in service class, and you can't extend it, move it to the client class and pass the object of a service class as a parameter.

## Eliminates smell

### § Incomplete Library Class

# Organizing Data

These refactoring techniques help with data handling, replacing primitives with rich class functionality.

Another important result is untangling of class associations, which makes classes more portable and reusable.

## § Self Encapsulate Field

**Problem:** You use direct access to private fields inside a class.

**Solution:** Create a getter and setter for the field, and use only them for accessing the field.

## § Replace Data Value with Object

**Problem:** A class (or group of classes) contains a data field. The field has its own behavior and associated data.

**Solution:** Create a new class, place the old field and its behavior in the class, and store the object of the class in the original class.

## § Change Value to Reference

**Problem:** So you have many identical instances of a single class that you need to replace with a single object.

**Solution:** Convert the identical objects to a single reference object.

## § Change Reference to Value

**Problem:** You have a reference object that's too small and infrequently changed to justify managing its life cycle.

**Solution:** Turn it into a value object.

## § Replace Array with Object

**Problem:** You have an array that contains various types of data.

**Solution:** Replace the array with an object that will have separate fields for each element.

## § Duplicate Observed Data

**Problem:** Is domain data stored in classes responsible for the GUI?

**Solution:** Then it's a good idea to separate the data into separate classes, ensuring connection and synchronization between the domain class and the GUI.

## § Change Unidirectional Association to Bidirectional

**Problem:** You have two classes that each need to use the features of the other, but the association between them is only unidirectional.

**Solution:** Add the missing association to the class that needs it.

## § Change Bidirectional Association to Unidirectional

**Problem:** You have a bidirectional association between classes, but one of the classes doesn't use the other's features.

**Solution:** Remove the unused association.

## § Replace Magic Number with Symbolic Constant

**Problem:** Your code uses a number that has a certain meaning to it.

**Solution:** Replace this number with a constant that has a human-readable name explaining the meaning of the number.

## § Encapsulate Field

**Problem:** You have a public field.

**Solution:** Make the field private and create access methods for it.

## § Encapsulate Collection

**Problem:** A class contains a collection field and a simple getter and setter for working with the collection.

**Solution:** Make the getter-returned value read-only and create methods for adding/deleting elements of the collection.

## § Replace Type Code with Class

**Problem:** A class has a field that contains type code. The values of this type aren't used in operator conditions and don't affect the behavior of the program.

**Solution:** Create a new class and use its objects instead of the type code values.

## § **Replace Type Code with Subclasses**

**Problem:** You have a coded type that directly affects program behavior (values of this field trigger various code in conditionals).

**Solution:** Create subclasses for each value of the coded type. Then extract the relevant behaviors from the original class to these subclasses. Replace the control flow code with polymorphism.

## § **Replace Type Code with State/Strategy**

**Problem:** You have a coded type that affects behavior but you can't use subclasses to get rid of it.

**Solution:** Replace type code with a state object. If it's necessary to replace a field value with type code, another state object is "plugged in".

## § **Replace Subclass with Fields**

**Problem:** You have subclasses differing only in their (constant-returning) methods.

**Solution:** Replace the methods with fields in the parent class and delete the subclasses.



# Self Encapsulate Field

Self-encapsulation is distinct from ordinary Encapsulate Field: the refactoring technique given here is performed on a private field.

## Problem

You use direct access to private fields inside a class.

```
1 class Range {
2     private int low, high;
3     boolean includes(int arg) {
4         return arg >= low && arg <= high;
5     }
6 }
```

## Solution

Create a getter and setter for the field, and use only them for accessing the field.

```
1  class Range {  
2      private int low, high;  
3      boolean includes(int arg) {  
4          return arg >= getLow() && arg <= getHigh();  
5      }  
6      int getLow() {  
7          return low;  
8      }  
9      int getHigh() {  
10         return high;  
11     }  
12 }
```

## Why Refactor

Sometimes directly accessing a private field inside a class just isn't flexible enough. You want to be able to initiate a field value when the first query is made or perform certain operations on new values of the field when they're assigned, or maybe do all this in various ways in subclasses.

## Benefits

- *Indirect access to fields* is when a field is acted on via access methods (getters and setters). This approach is much more flexible than *direct access to fields*.
  - First, you can perform complex operations when data in the field is set or received. *Lazy initialization* and *validation of*

*field values* are easily implemented inside field getters and setters.

- Second and more crucially, you can redefine getters and setters in subclasses.
- You have the option of not implementing a setter for a field at all. The field value will be specified only in the constructor, thus making the field unchangeable throughout the entire object lifespan.

## Drawbacks

When *direct access to fields* is used, code looks simpler and more presentable, although flexibility is diminished.

## How to Refactor

1. Create a getter (and optional setter) for the field. They should be either `protected` or `public`.
2. Find all direct invocations of the field and replace them with getter and setter calls.

## Similar refactorings

### § Encapsulate Field

Hide public fields, provide getters and setters.

## Helps other refactorings

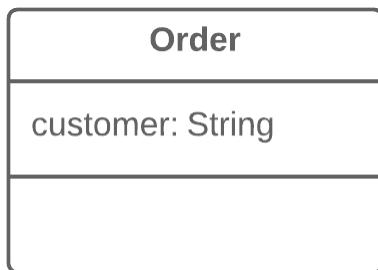
- § Duplicate Observed Data
- § Replace Type Code with Subclasses
- § Replace Type Code with State/Strategy



# Replace Data Value with Object

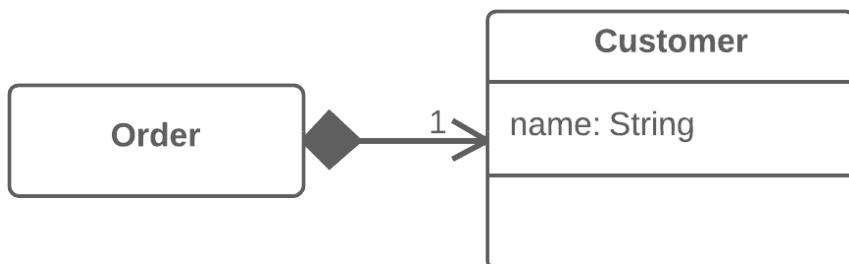
## Problem

A class (or group of classes) contains a data field. The field has its own behavior and associated data.



## Solution

Create a new class, place the old field and its behavior in the class, and store the object of the class in the original class.



## Why Refactor

This refactoring is basically a special case of Extract Class. What makes it different is the cause of the refactoring.

In Extract Class, we have a single class that's responsible for different things and we want to split up its responsibilities.

With replacement of a data value with an object, we have a primitive field (number, string, etc.) that's no longer so simple due to growth of the program and now has associated data and behaviors. On the one hand, there's nothing scary about these fields in and of themselves. However, this fields-and-behaviors family can be present in several classes simultaneously, creating duplicate code.

Therefore, for all this we create a new class and move both the field and the related data and behaviors to it.

## Benefits

Improves relatedness inside classes. Data and the relevant behaviors are inside a single class.

## How to Refactor

Before you begin with refactoring, see if there are direct references to the field from within the class. If so, use Self Encapsulate Field in order to hide it in the original class.

1. Create a new class and copy your field and relevant getter to it. In addition, create a constructor that accepts the simple value of the field. This class won't have a setter since each new field value that's sent to the original class will create a new value object.
2. In the original class, change the field type to the new class.
3. In the getter in the original class, invoke the getter of the associated object.
4. In the setter, create a new value object. You may need to also create a new object in the constructor if initial values had been set there for the field previously.

## Next Steps

After applying this refactoring technique, it's wise to apply **Change Value to Reference** on the field that contains the object. This allows storing a reference to a single object that corresponds to a value instead of storing dozens of objects for one and the same value.

Most often this approach is needed when you want to have one object be responsible for one real-world object (such as users, orders, documents and so forth). At the same time, this approach won't be useful for objects such as dates, money, ranges, etc.

## Similar refactorings

- § **Extract Class**
- § **Introduce Parameter Object**
- § **Replace Array with Object**
- § **Replace Method with Method Object**

Does the same with method's code.

## Eliminates smell

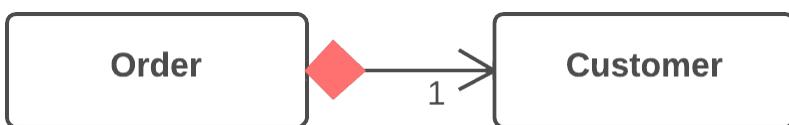
- § **Duplicate Code**



# Change Value to Reference

## Problem

So you have many identical instances of a single class that you need to replace with a single object.



## Solution

Convert the identical objects to a single reference object.



## Why Refactor

In many systems, objects can be classified as either values or references.

- **References:** when one real-world object corresponds to only one object in the program. References are usually user/order/product/etc. objects.
- **Values:** one real-world object corresponds to multiple objects in the program. These objects could be dates, phone numbers, addresses, colors, and the like.

The selection of reference vs. value isn't always clear-cut. Sometimes there's a simple value with a small amount of unchanging data. Then it becomes necessary to add changeable data and pass these changes every time the object is accessed. In this case it becomes necessary to convert it to a reference.

## Benefits

An object contains all the most current information about a particular entity. If the object is changed in one part of the program, these changes are accessible from the other parts of the program that make use of the object.

## Drawbacks

References are much harder to implement.

## How to Refactor

1. Use **Replace Constructor with Factory Method** on the class from which the references are to be generated.
2. Determine which object will be responsible for providing access to references. Instead of creating a new object, when you need one you now need to get it from a storage object or static dictionary field.
3. Determine whether references will be created in advance or dynamically as necessary. If objects are created in advance, make sure to load them before use.
4. Change the factory method so that it returns a reference. If objects are created in advance, decide how to handle errors when a non-existent object is requested. You may also need to use **Rename Method** to inform that the method returns only existing objects.

## Anti-refactoring

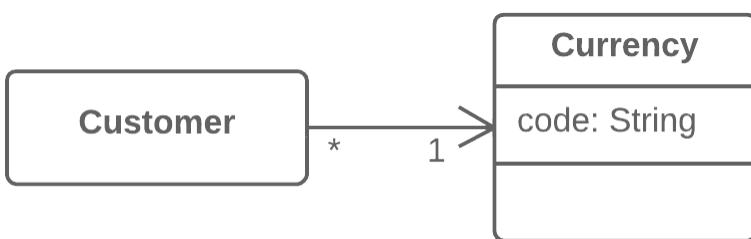
### § Change Reference to Value



# Change Reference to Value

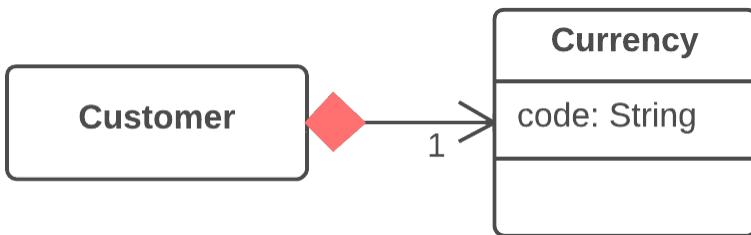
## Problem

You have a reference object that's too small and infrequently changed to justify managing its life cycle.



## Solution

Turn it into a value object.



## Why Refactor

Inspiration to switch from a reference to a value may come from the inconvenience of working with the reference. References require management on your part:

- They always require requesting the necessary object from storage.
- References in memory may be inconvenient to work with.
- Working with references is particularly difficult, compared to values, on distributed and parallel systems.

Values are especially useful if you would rather have unchangeable objects than objects whose state may change during their lifetime.

## Benefits

- One important property of objects is that they should be unchangeable. The same result should be received for each query that returns an object value. If this is true, no problems arise if there are many objects representing the same thing.
- Values are much easier to implement.

## Drawbacks

If a value is changeable, make sure if any object changes that the values in all the other objects representing the same entity are updated. This is so burdensome that it's easier to create a reference for this purpose.

## How to Refactor

1. Make the object unchangeable. The object shouldn't have any setters or other methods that change its state and data ([Remove Setting Method](#) may help here). The only place where data should be assigned to the fields of a value object is a constructor.
2. Create a comparison method to be able to compare two values.
3. Check whether you can delete the factory method and make the object constructor public.

## Anti-refactoring

### § Change Value to Reference



# Replace Array with Object

This refactoring technique is a special case of [Replace Data Value with Object](#).

## Problem

You have an array that contains various types of data.

```
1 String[] row = new String[2];  
2 row[0] = "Liverpool";  
3 row[1] = "15";
```

## Solution

Replace the array with an object that will have separate fields for each element.

```
1 Performance row = new Performance();  
2 row.setName("Liverpool");  
3 row.setWins("15");
```

## Why Refactor

Arrays are an excellent tool for storing data and collections of a single type. But if you use an array like post office boxes, storing the username in box 1 and the user's address in box 14, you will someday be very unhappy that you did. This approach leads to catastrophic failures when somebody puts something in the wrong "box" and also requires your time for figuring out which data is stored where.

## Benefits

- In the resulting class, you can place all associated behaviors that had been previously stored in the main class or elsewhere.
- The fields of a class are much easier to document than the elements of an array.

## How to Refactor

1. Create the new class that will contain the data from the array. Place the array itself in the class as a public field.
2. Create a field for storing the object of this class in the original class. Don't forget to also create the object itself in the place where you initiated the data array.
3. In the new class, create access methods one by one for each of the array elements. Give them self-explanatory names that

indicate what they do. At the same time, replace each use of an array element in the main code with the corresponding access method.

4. When access methods have been created for all elements, make the array private.
5. For each element of the array, create a private field in the class and then change the access methods so that they use this field instead of the array.
6. When all data has been moved, delete the array.

## Similar refactorings

### § Replace Data Value with Object

### Eliminates smell

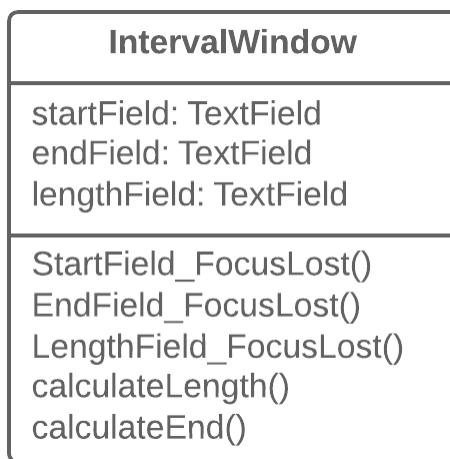
### § Primitive Obsession



# Duplicate Observed Data

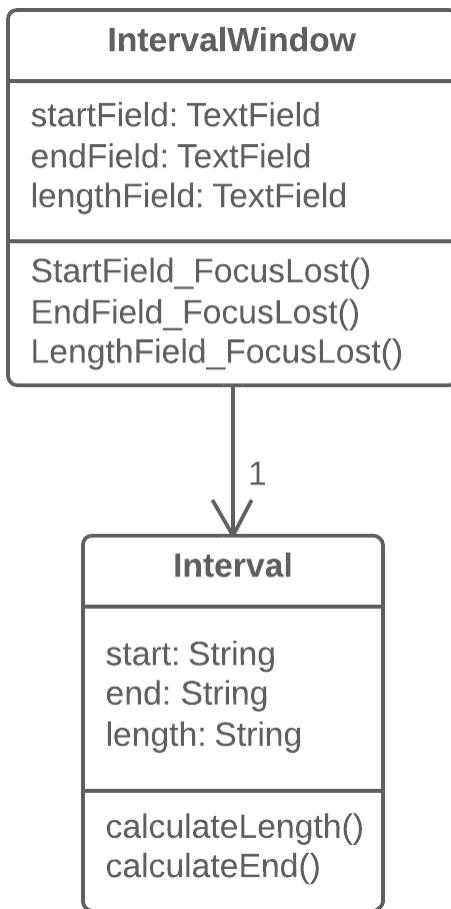
## Problem

Is domain data stored in classes responsible for the GUI?



## Solution

Then it's a good idea to separate the data into separate classes, ensuring connection and synchronization between the domain class and the GUI.



## Why Refactor

You want to have multiple interface views for the same data (for example, you have both a desktop app and a mobile app). If you fail to separate the GUI from the domain, you will have a very hard time avoiding code duplication and a large number of mistakes.

## Benefits

- You split responsibility between business logic classes and presentation classes (cf. the *Single Responsibility Principle*), which makes your program more readable and understandable.
- If you need to add a new interface view, create new presentation classes; you don't need to touch the code of the business logic (cf. the *Open/Closed Principle*).
- Now different people can work on the business logic and the user interfaces.

## When Not to Use

- This refactoring technique, which in its classic form is performed using the **Observer** template, isn't applicable for web apps, where all classes are recreated between queries to the web server.
- All the same, the general principle of extracting business logic into separate classes can be justified for web apps as well. But this will be implemented using different refactoring techniques depending on how your system is designed.

## How to Refactor

1. Hide direct access to domain data in the *GUI class*. For this, it's best to use **Self Encapsulate Field**. So you create the getters and setters for this data.
2. In handlers for *GUI class* events, use setters to set new field values. This will let you pass these values to the associated *domain object*.
3. Create a domain class and copy necessary fields from the *GUI class* to it. Create getters and seters for all these fields.
4. Create an Observer pattern for these two classes:
  - In the *domain class*, create an array for storing observer objects (*GUI objects*), as well as methods for registering, deleting and notifying them.
  - In the *GUI class*, create a field for storing references to the *domain class* as well as the `update()` method, which will be reacting to changes in the object and update the values of fields in the *GUI class*. Note that value updates should be established directly in the method, in order to avoid recursion.
  - In the *GUI class* constructor, create an instance of *domain class* and save it in the field you have created. Register the *GUI object* as an observer in the *domain object*.

- In the setters for *domain class* fields, call the method for notifying the observer (in other words, method for updating in the *GUI class*), in order to pass the new values to the GUI.
- Change the setters of the *GUI class* fields so that they set new values in the domain object directly. Watch out to make sure that values aren't set through a *domain class* setter – otherwise infinite recursion will result.

## Implements design pattern

§ Observer

## Eliminates smell

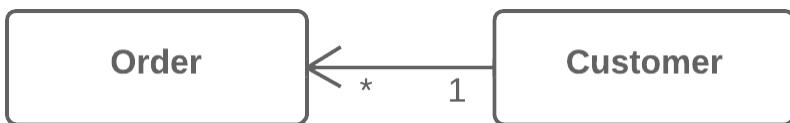
§ Large Class



# Change Unidirectional Association to Bidirectional

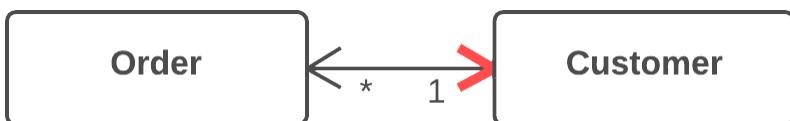
## Problem

You have two classes that each need to use the features of the other, but the association between them is only unidirectional.



## Solution

Add the missing association to the class that needs it.



## Why Refactor

Originally the classes had a unidirectional association. But with time, client code needed access to both sides of the association.

## Benefits

If a class needs a reverse association, you can simply calculate it. But if these calculations are complex, it's better to keep the reverse association.

## Drawbacks

- Bidirectional associations are much harder to implement and maintain than unidirectional ones.
- Bidirectional associations make classes interdependent. With a unidirectional association, one of them can be used independently of the other.

## How to Refactor

1. Add a field for holding the reverse association.
2. Decide which class will be “dominant”. This class will contain the methods that create or update the association as elements are added or changed, establishing the association in its class

and calling the utility methods for establishing the association in the associated object.

3. Create a utility method for establishing the association in the “non-dominant” class. The method should use what it’s given in parameters to complete the field. Give the method an obvious name so that it isn’t used later for any other purposes.
4. If old methods for controlling the unidirectional association were in the “dominant” class, complement them with calls to utility methods from the associated object.
5. If the old methods for controlling the association were in the “non-dominant” class, create the methods in the “dominant” class, call them, and delegate execution to them.

## Anti-refactoring

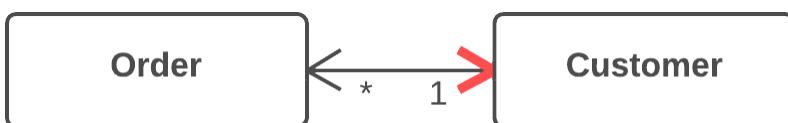
### § Change Bidirectional Association to Unidirectional



# Change Bidirectional Association to Unidirectional

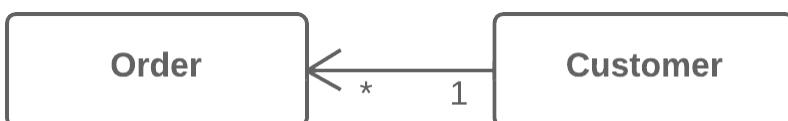
## Problem

You have a bidirectional association between classes, but one of the classes doesn't use the other's features.



## Solution

Remove the unused association.



## Why Refactor

A bidirectional association is generally harder to maintain than a unidirectional one, requiring additional code for properly creating and deleting the relevant objects. This makes the program more complicated.

In addition, an improperly implemented bidirectional association can cause problems for garbage collection (in turn leading to memory bloat by unused objects).

Example: the garbage collector removes objects from memory that are no longer referenced by other objects. Let's say that an object pair `User - Order` was created, used, and then abandoned. But these objects won't be cleared from memory since they still refer to each other. That said, this problem is becoming less important thanks to advances in programming languages, which now automatically identify unused object references and remove them from memory.

There's also the problem of interdependency between classes. In a bidirectional association, the two classes must know about each other, meaning that they can't be used separately. If many of these associations are present, different parts of the program become too dependent on each other and any changes in one component may affect the other components.

## Benefits

- Simplifies the class that doesn't need the relationship. Less code equals less code maintenance.
- Reduces dependency between classes. Independent classes are easier to maintain since any changes to a class affect only that class.

## How to Refactor

1. Make sure that one of the following is true for your classes:
  - No association is used.
  - There's another way to get the associated object, such through a database query.
  - The associated object can be passed as an argument to the methods that use it.
2. Depending on your situation, use of a field that contains an association with another object should be replaced by a parameter or method call for getting the object in a different way.
3. Delete the code that assigns the associated object to the field.
4. Delete the now-unused field.

## Anti-refactoring

### § Change Unidirectional Association to Bidirectional

**Eliminates smell**

### § Inappropriate Intimacy



# Replace Magic Number with Symbolic Constant

## Problem

Your code uses a number that has a certain meaning to it.

```
1 double potentialEnergy(double mass, double height) {  
2     return mass * height * 9.81;  
3 }
```

## Solution

Replace this number with a constant that has a human-readable name explaining the meaning of the number.

```
1 static final double GRAVITATIONAL_CONSTANT = 9.81;  
2  
3 double potentialEnergy(double mass, double height) {  
4     return mass * height * GRAVITATIONAL_CONSTANT;  
5 }
```

## Why Refactor

A magic number is a numeric value that's encountered in the source but has no obvious meaning. This “anti-pattern” makes it harder to understand the program and refactor the code.

Yet more difficulties arise when you need to change this magic number. Find and replace won't work for this: the same number may be used for different purposes in different places, meaning that you will have to verify every line of code that uses this number.

## Benefits

- The symbolic constant can serve as live documentation of the meaning of its value.
- It's much easier to change the value of a constant than to search for this number throughout the entire codebase, without the risk of accidentally changing the same number used elsewhere for a different purpose.
- Reduce duplicate use of a number or string in the code. This is especially important when the value is complicated and long (such as `3.14159` or `0xCAFEBAE` ).

## Good to Know

**Not all numbers are magical.**

If the purpose of a number is obvious, there's no need to replace it. A classic example is:

```
1 for (i = 0; i < count; i++) { ... }
```

## Alternatives

1. Sometimes a magic number can be replaced with method calls. For example, if you have a magic number that signifies the number of elements in a collection, you don't need to use it for checking the last element of the collection. Instead, use the standard method for getting the collection length.
2. Magic numbers are sometimes used as type code. Say that you have two types of users and you use a number field in a class to specify which is which: administrators are 1 and ordinary users are 2 .

In this case, you should use one of the refactoring methods to avoid type code:

- [Replace Type Code with Class](#)
- [Replace Type Code with Subclasses](#)

- **Replace Type Code with State/Strategy**

## How to Refactor

1. Declare a constant and assign the value of the magic number to it.
2. Find all mentions of the magic number.
3. For each of the numbers that you find, double-check that the magic number in this particular case corresponds to the purpose of the constant. If yes, replace the number with your constant. This is an important step, since the same number can mean absolutely different things (and replaced with different constants, as the case may be).



# Encapsulate Field

## Problem

You have a public field.

```
1 class Person {  
2     public String name;  
3 }
```

## Solution

Make the field private and create access methods for it.

```
1 class Person {  
2     private String name;  
3  
4     public String getName() {  
5         return name;  
6     }  
7     public void setName(String arg) {  
8         name = arg;  
9     }  
10 }
```

## Why Refactor

One of the pillars of object-oriented programming is *Encapsulation*, the ability to conceal object data. Otherwise, all objects would be public and other objects could get and modify the data of your object without any checks and balances! Data is separated from the behaviors associated with this data, modularity of program sections is compromised, and maintenance becomes complicated.

## Benefits

- If the data and behavior of a component are closely interrelated and are in the same place in the code, it's much easier for you to maintain and develop this component.
- You can also perform complicated operations related to access to object fields.

## When Not to Use

In some cases, encapsulation is ill-advised due to performance considerations. These cases are rare but when they happen, this circumstance is very important.

Say that you have a graphical editor that contains objects possessing x- and y-coordinates. These fields are unlikely to change in the future. What's more, the program involves a great many different objects in which these fields are present.

So accessing the coordinate fields directly saves significant CPU cycles that would otherwise be taken up by calling access methods.

As an example of this unusual case, there's the **Point** class in Java. All fields of this class are public.

## How to Refactor

1. Create a getter and setter for the field.
2. Find all invocations of the field. Replace receipt of the field value with the getter, and replace setting of new field values with the setter.
3. After all field invocations have been replaced, make the field private.

## Next Steps

*Encapsulate Field* is only the first step in bringing data and the behaviors involving this data closer together. After you create simple methods for access fields, you should recheck the places where these methods are called. It's quite possible that the code in these areas would look more appropriate in the access methods.

## Similar refactorings

### § Self Encapsulate Field

Create getters and setters for a field instead of direct access *within the class' methods.*

## Eliminates smell

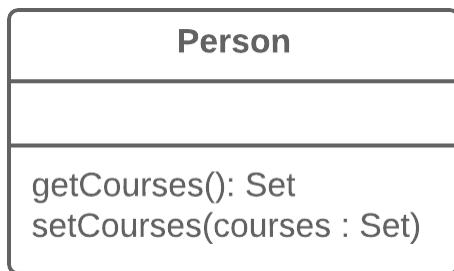
### § Data Class



# Encapsulate Collection

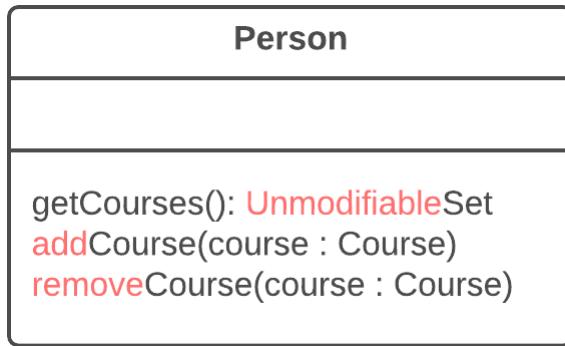
## Problem

A class contains a collection field and a simple getter and setter for working with the collection.



## Solution

Make the getter-returned value read-only and create methods for adding/deleting elements of the collection.



## Why Refactor

A class contains a field that contains a collection of objects. This collection could be an array, list, set or vector. A normal getter and setter have been created for working with the collection.

But the collections should be used by a protocol that's a bit different from the one used by other data types. The getter method shouldn't return the collection object itself, since this would let clients change collection contents without the knowledge of the owner class. In addition, this would show too much of the internal structures of the object data to clients. The method for getting collection elements should return a value that doesn't allow changing the collection or disclose excessive data about its structure.

In addition, there shouldn't be a method that assigns a value to the collection. Instead, there should be operations for adding and deleting elements. Thanks to this, the owner

object gains control over addition and deletion of collection elements.

Such a protocol properly encapsulates a collection, which ultimately reduces the degree of association between the owner class and the client code.

## Benefits

- The collection field is encapsulated inside a class. When the getter is called, it returns a copy of the collection, which prevents accidental changing or overwriting of the collection elements without the knowledge of the class that contains the collection.
- If collection elements are contained inside a primitive type, such as an array, you create more convenient methods for working with the collection.
- If collection elements are contained inside a non-primitive container (standard collection class), by encapsulating the collection you can restrict access to unwanted standard methods of the collection (such as by restricting addition of new elements).

## How to Refactor

1. Create methods for adding and deleting collection elements. They must accept collection elements in their parameters.

2. Assign an empty collection to the field as the initial value if this isn't done in the class constructor.
3. Find the calls of the collection field setter. Change the setter so that it uses operations for adding and deleting elements, or make these operations call client code.

Note that setters can be used only to replace all collection elements with other ones. Therefore it may be advisable to change the setter name (**Rename Method**) to `replace`.

4. Find all calls of the collection getter after which the collection is changed. Change the code so that it uses your new methods for adding and deleting elements from the collection.
5. Change the getter so that it returns a read-only representation of the collection.
6. Inspect the client code that uses the collection for code that would look better inside of the collection class itself.

## Eliminates smell

### § Data Class

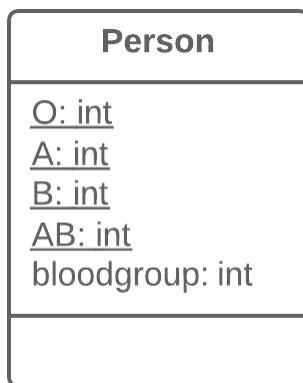


# Replace Type Code with Class

**What's type code?** Type code occurs when, instead of a separate data type, you have a set of numbers or strings that form a list of allowable values for some entity. Often these specific numbers and strings are given understandable names via constants, which is the reason for why such type code is encountered so much.

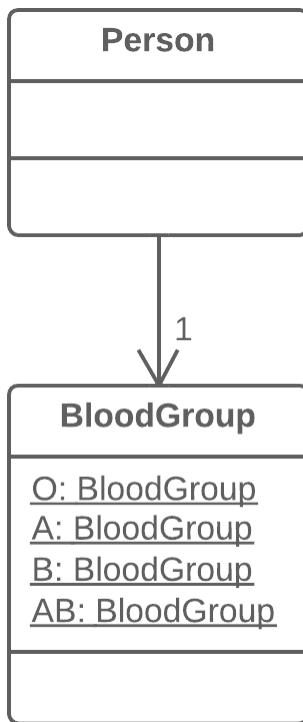
## Problem

A class has a field that contains type code. The values of this type aren't used in operator conditions and don't affect the behavior of the program.



## Solution

Create a new class and use its objects instead of the type code values.



## Why Refactor

One of the most common reasons for type code is working with databases, when a database has fields in which some complex concept is coded with a number or string.

For example, you have the class `User` with the field `user_role`, which contains information about the access priv-

ileges of each user, whether administrator, editor, or ordinary user. So in this case, this information is coded in the field as A , E , and U respectively.

What are the shortcomings of this approach? The field setters often don't check which value is sent, which can cause big problems when someone sends unintended or wrong values to these fields.

In addition, type verification is impossible for these fields. It's possible to send any number or string to them, which won't be type checked by your IDE and even allow your program to run (and crash later).

## Benefits

- We want to turn sets of primitive values – which is what coded types are – into full-fledged classes with all the benefits that object-oriented programming has to offer.
- By replacing type code with classes, we allow type hinting for values passed to methods and fields at the level of the programming language.

For example, while the compiler previously didn't see difference between your numeric constant and some arbitrary number when a value is passed to a method, now when data that doesn't fit the indicated type class is passed, you're warned of the error inside your IDE.

- Thus we make it possible to move code to the classes of the type. If you needed to perform complex manipulations with type values throughout the whole program, now this code can “live” inside one or multiple type classes.

## When Not to Use

If the values of a coded type are used inside control flow structures (`if`, `switch`, etc.) and control a class behavior, you should use one of the two refactoring techniques for type code:

- Replace Type Code with Subclasses
- Replace Type Code with State/Strategy

## How to Refactor

1. Create a new class and give it a new name that corresponds to the purpose of the coded type. Here we'll call it *type class*.
2. Copy the field containing type code to the *type class* and make it private. Then create a getter for the field. A value will be set for this field only from the constructor.
3. For each value of the coded type, create a static method in *type class*. It'll be creating a new *type class* object corresponding to this value of the coded type.

4. In the original class, replace the type of the coded field with *type class*. Create a new object of this type in the constructor as well as in the field setter. Change the field getter so that it calls the *type class* getter.
5. Replace any mentions of values of the coded type with calls of the relevant *type class* static methods.
6. Remove the coded type constants from the original class.

## Similar refactorings

§ Replace Type Code with Subclasses

§ Replace Type Code with State/Strategy

## Eliminates smell

§ Primitive Obsession



# Replace Type Code with Subclasses

**What's type code?** Type code occurs when, instead of a separate data type, you have a set of numbers or strings that form a list of allowable values for some entity. Often these specific numbers and strings are given understandable names via constants, which is the reason for why such type code is encountered so much.

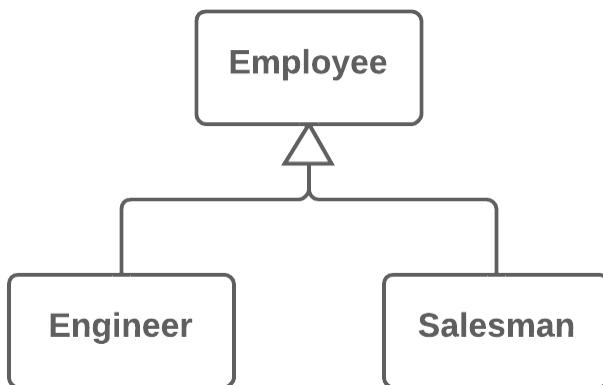
## Problem

You have a coded type that directly affects program behavior (values of this field trigger various code in conditionals).



## Solution

Create subclasses for each value of the coded type. Then extract the relevant behaviors from the original class to these subclasses. Replace the control flow code with polymorphism.



## Why Refactor

This refactoring technique is a more complicated twist on Replace Type Code with Class.

As in the first refactoring method, you have a set of simple values that constitute all the allowed values for a field. Although these values are often specified as constants and have understandable names, their use makes your code very error-prone since they're still primitives in effect. For example, you have a method that accepts one of these values in the parameters. At a certain moment, instead of the constant `USER_TYPE_ADMIN` with the value `"ADMIN"`, the method receives the same string

in lower case ( "admin" ), which will cause execution of something else that the author (you) didn't intend.

Here we're dealing with control flow code such as the conditionals `if`, `switch` and `?:`. In other words, fields with coded values (such as `$user->type === self::USER_TYPE_ADMIN`) are used inside the conditions of these operators. If we were to use **Replace Type Code with Class** here, all these control flow constructions would be best moved to a class responsible for the data type. Ultimately, this would of course create a type class very similar to the original one, with the same problems as well.

## Benefits

- Delete the control flow code. Instead of a bulky `switch` in the original class, move the code to appropriate subclasses. This improves adherence to the *Single Responsibility Principle* and makes the program more readable in general.
- If you need to add a new value for a coded type, all you need to do is add a new subclass without touching the existing code (cf. the *Open/Closed Principle*).
- By replacing type code with classes, we pave the way for type hinting for methods and fields at the level of the programming language. This wouldn't be possible using simple numeric or string values contained in a coded type.

## When Not to Use

- This technique isn't applicable if you already have a class hierarchy. You can't create a dual hierarchy via inheritance in object-oriented programming. Still, you can replace type code via composition instead of inheritance. To do so, use Replace Type Code with State/Strategy.
- If the values of type code can change after an object is created, avoid this technique. We would have to somehow replace the class of the object itself on the fly, which isn't possible. Still, an alternative in this case too would be Replace Type Code with State/Strategy.

## How to Refactor

1. Use Self Encapsulate Field to create a getter for the field that contains type code.
2. Make the superclass constructor private. Create a static factory method with the same parameters as the superclass constructor. It must contain the parameter that will take the starting values of the coded type. Depending on this parameter, the factory method will create objects of various subclasses. To do so, in its code you must create a large conditional but, at least, it'll be the only one when it's truly necessary; otherwise, subclasses and polymorphism will do.

3. Create a unique subclass for each value of the coded type. In it, redefine the getter of the coded type so that it returns the corresponding value of the coded type.
4. Delete the field with type code from the superclass. Make its getter abstract.
5. Now that you have subclasses, you can start to move the fields and methods from the superclass to corresponding subclasses (with the help of **Push Down Field** and **Push Down Method**).
6. When everything possible has been moved, use **Replace Conditional with Polymorphism** in order to get rid of conditions that use the type code once and for all.

## Anti-refactoring

### § Replace Subclass with Fields

## Similar refactorings

### § Replace Type Code with Class

### § Replace Type Code with State/Strategy

## Eliminates smell

### § Primitive Obsession



# Replace Type Code with State/Strategy

**What's type code?** Type code occurs when, instead of a separate data type, you have a set of numbers or strings that form a list of allowable values for some entity. Often these specific numbers and strings are given understandable names via constants, which is the reason for why such type code is encountered so much.

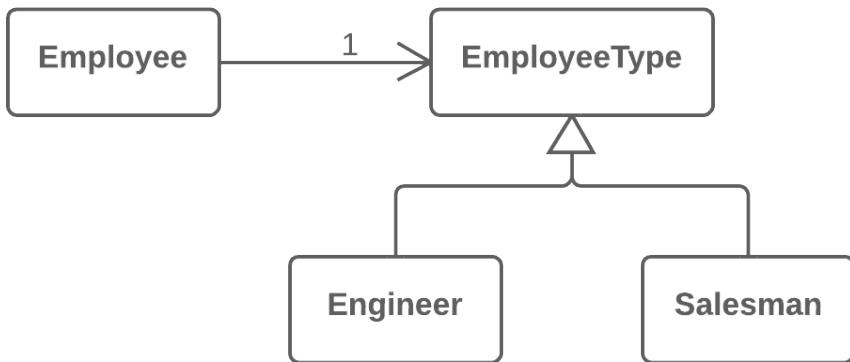
## Problem

You have a coded type that affects behavior but you can't use subclasses to get rid of it.



## Solution

Replace type code with a state object. If it's necessary to replace a field value with type code, another state object is "plugged in".



## Why Refactor

You have type code and it affects the behavior of a class, therefore we can't use [Replace Type Code with Class](#).

Type code affects the behavior of a class but we can't create subclasses for the coded type due to the existing class hierarchy or other reasons. Thus means that we can't apply [Replace Type Code with Subclasses](#).

## Benefits

- This refactoring technique is a way out of situations when a field with a coded type changes its value during the object's

lifetime. In this case, replacement of the value is made via replacement of the state object to which the original class refers.

- If you need to add a new value of a coded type, all you need to do is to add a new state subclass without altering the existing code (cf. the *Open/Closed Principle*).

## Drawbacks

If you have a simple case of type code but you use this refactoring technique anyway, you will have many extra (and unneeded) classes.

## Good to Know

Implementation of this refactoring technique can make use of one of two design patterns: **State** or **Strategy**. Implementation is the same no matter which pattern you choose. So which pattern should you pick in a particular situation?

If you're trying to split a conditional that controls the selection of algorithms, use Strategy.

But if each value of the coded type is responsible not only for selecting an algorithm but for the whole condition of the class, class state, field values, and many other actions, State is better for the job.

## How to Refactor

1. Use **Self Encapsulate Field** to create a getter for the field that contains type code.
2. Create a new class and give it an understandable name that fits the purpose of the type code. This class will be playing the role of *state* (or *strategy*). In it, create an abstract coded field getter.
3. Create subclasses of the state class for each value of the coded type. In each subclass, redefine the getter of the coded field so that it returns the corresponding value of the coded type.
4. In the abstract state class, create a static factory method that accepts the value of the coded type as a parameter. Depending on this parameter, the factory method will create objects of various states. For this, in its code create a large conditional; it'll be the only one when refactoring is complete.
5. In the original class, change the type of the coded field to the state class. In the field's setter, call the factory state method for getting new state objects.
6. Now you can start to move the fields and methods from the superclass to the corresponding state subclasses (using **Push Down Field** and **Push Down Method**).

7. When everything moveable has been moved, use **Replace Conditional with Polymorphism** in order to get rid of conditionals that use type code once and for all.

## Similar refactorings

- § **Replace Type Code with Class**
- § **Replace Type Code with Subclasses**

## Implements design pattern

- § **State**
- § **Strategy**

## Eliminates smell

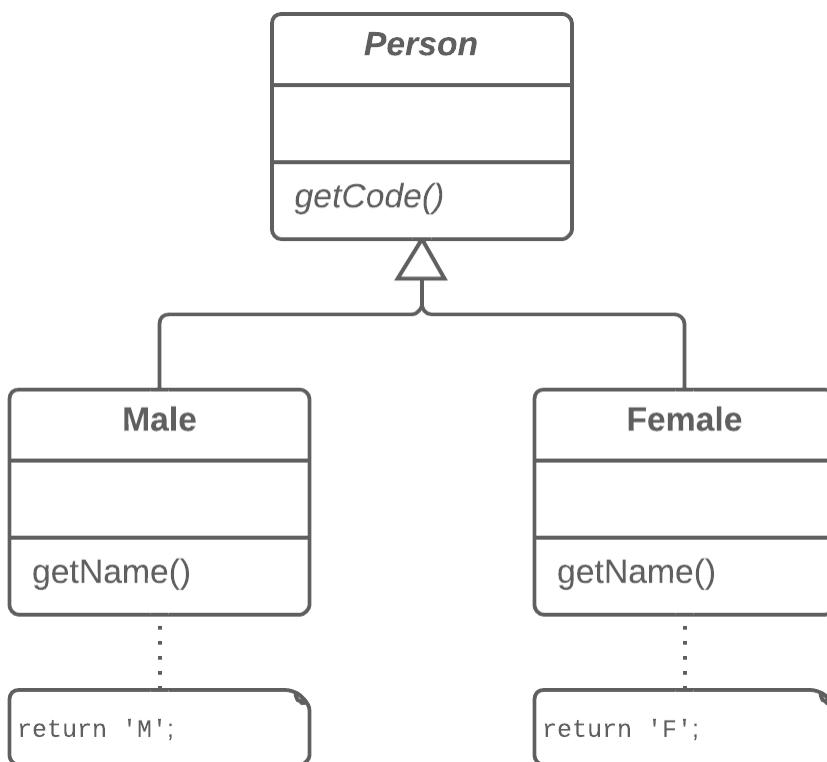
- § **Primitive Obsession**



# Replace Subclass with Fields

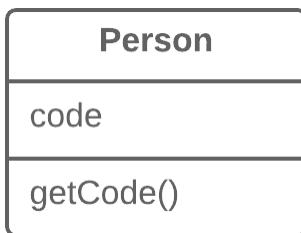
## Problem

You have subclasses differing only in their (constant-returning) methods.



## Solution

Replace the methods with fields in the parent class and delete the subclasses.



## Why Refactor

Sometimes refactoring is just the ticket for avoiding type code.

In one such case, a hierarchy of subclasses may be different only in the values returned by particular methods. These methods aren't even the result of computation, but are strictly set out in the methods themselves or in the fields returned by the methods. To simplify the class architecture, this hierarchy can be compressed into a single class containing one or several fields with the necessary values, based on the situation.

These changes may become necessary after moving a large amount of functionality from a class hierarchy to another place. The current hierarchy is no longer so valuable and its subclasses are now just dead weight.

## Benefits

Simplifies system architecture. Creating subclasses is overkill if all you want to do is to return different values in different methods.

## How to Refactor

1. Apply Replace Constructor with Factory Method to the subclasses.
2. Replace subclass constructor calls with superclass factory method calls.
3. In the superclass, declare fields for storing the values of each of the subclass methods that return constant values.
4. Create a protected superclass constructor for initializing the new fields.
5. Create or modify the existing subclass constructors so that they call the new constructor of the parent class and pass the relevant values to it.
6. Implement each constant method in the parent class so that it returns the value of the corresponding field. Then remove the method from the subclass.

7. If the subclass constructor has additional functionality, use **Inline Method** to incorporate the constructor into the superclass factory method.
8. Delete the subclass.

## Anti-refactoring

### § Replace Type Code with Subclasses

# Simplifying Conditional Expressions

Conditionals tend to get more and more complicated in their logic over time, and there are yet more techniques to combat this as well.

## § Decompose Conditional

**Problem:** You have a complex conditional (`if-then` / `else` or `switch`).

**Solution:** Decompose the complicated parts of the conditional into separate methods: the condition, `then` and `else`.

## § Consolidate Conditional Expression

**Problem:** You have multiple conditionals that lead to the same result or action.

**Solution:** Consolidate all these conditionals in a single expression.

## § Consolidate Duplicate Conditional Fragments

**Problem:** Identical code can be found in all branches of a conditional.

**Solution:** Move the code outside of the conditional.

## § Remove Control Flag

**Problem:** You have a boolean variable that acts as a control flag for multiple boolean expressions.

**Solution:** Instead of the variable, use `break`, `continue` and `return`.

## § Replace Nested Conditional with Guard Clauses

**Problem:** You have a group of nested conditionals and it's hard to determine the normal flow of code execution.

**Solution:** Isolate all special checks and edge cases into separate clauses and place them before the main checks. Ideally, you should have a “flat” list of conditionals, one after the other.

## § Replace Conditional with Polymorphism

**Problem:** You have a conditional that performs various actions depending on object type or properties.

**Solution:** Create subclasses matching the branches of the conditional. In them, create a shared method and move code from the corresponding branch of the conditional to it. Then replace the conditional with the relevant method call. The result is that the proper implementation will be attained via polymorphism depending on the object class.

## § Introduce Null Object

**Problem:** Since some methods return `null` instead of real objects, you have many checks for `null` in your code.

**Solution:** Instead of `null`, return a null object that exhibits the default behavior.

## § Introduce Assertion

**Problem:** For a portion of code to work correctly, certain conditions or values must be true.

**Solution:** Replace these assumptions with specific assertion checks.



# Decompose Conditional

## Problem

You have a complex conditional (`if-then` / `else` or `switch`).

```

1 if (date.before(SUMMER_START) || date.after(SUMMER_END)) {
2     charge = quantity * winterRate + winterServiceCharge;
3 }
4 else {
5     charge = quantity * summerRate;
6 }
```

## Solution

Decompose the complicated parts of the conditional into separate methods: the condition, `then` and `else`.

```

1 if (isSummer(date)) {
2     charge = summerCharge(quantity);
3 }
4 else {
5     charge = winterCharge(quantity);
```

```
6 }
```

## Why Refactor

The longer a piece of code is, the harder it's to understand. Things become even more hard to understand when the code is filled with conditions:

- While you're busy figuring out what the code in the `then` block does, you forget what the relevant condition was.
- While you're busy parsing `else`, you forget what the code in `then` does.

## Benefits

- By extracting conditional code to clearly named methods, you make life easier for the person who'll be maintaining the code later (such as you, two months from now!).
- This refactoring technique is also applicable for short expressions in conditions. The string `isSalaryDay()` is much prettier and more descriptive than code for comparing dates.

## How to Refactor

1. Extract the conditional to a separate method via [Extract Method](#).

2. Repeat the process for the `then` and `else` blocks.

## Eliminates smell

### § Long Method



# Consolidate Conditional Expression

## Problem

You have multiple conditionals that lead to the same result or action.

```
1 double disabilityAmount() {  
2     if (seniority < 2) {  
3         return 0;  
4     }  
5     if (monthsDisabled > 12) {  
6         return 0;  
7     }  
8     if (isPartTime) {  
9         return 0;  
10    }  
11    // compute the disability amount  
12    //...  
13 }
```

## Solution

Consolidate all these conditionals in a single expression.

```
1 double disabilityAmount() {  
2     if (isNotEligibleForDisability()) {  
3         return 0;  
4     }  
5     // compute the disability amount  
6     //...  
7 }
```

## Why Refactor

Your code contains many alternating operators that perform identical actions. It isn't clear why the operators are split up.

The main purpose of consolidation is to extract the conditional to a separate method for greater clarity.

## Benefits

- Eliminates duplicate control flow code. Combining multiple conditionals that have the same “destination” helps to show that you’re doing only one complicated check leading to one action.

- By consolidating all operators, you can now isolate this complex expression in a new method with a name that explains the conditional's purpose.

## How to Refactor

Before refactoring, make sure that the conditionals don't have any "side effects" or otherwise modify something, instead of simply returning values. Side effects may be hiding in the code executed inside the operator itself, such as when something is added to a variable based on the results of a conditional.

1. Consolidate the conditionals in a single expression by using `and` and `or`. As a general rule when consolidating:
  - Nested conditionals are joined using `and`.
  - Consecutive conditionals are joined with `or`.
2. Perform **Extract Method** on the operator conditions and give the method a name that reflects the expression's purpose.

## Eliminates smell

### § Duplicate Code



# Consolidate Duplicate Conditional Fragments

## Problem

Identical code can be found in all branches of a conditional.

```
1 if (isSpecialDeal()) {  
2     total = price * 0.95;  
3     send();  
4 }  
5 else {  
6     total = price * 0.98;  
7     send();  
8 }
```

## Solution

Move the code outside of the conditional.

```
1  if (isSpecialDeal()) {  
2      total = price * 0.95;  
3  }  
4  else {  
5      total = price * 0.98;  
6  }  
7  send();
```

## Why Refactor

Duplicate code is found inside all branches of a conditional, often as the result of evolution of the code within the conditional branches. Team development can be a contributing factor to this.

## Benefits

Code deduplication.

## How to Refactor

1. If the duplicated code is at the beginning of the conditional branches, move the code to a place before the conditional.
2. If the code is executed at the end of the branches, place it after the conditional.
3. If the duplicate code is randomly situated inside the branches, first try to move the code to the beginning or end of the

branch, depending on whether it changes the result of the subsequent code.

4. If appropriate and the duplicate code is longer than one line, try using **Extract Method**.

## Eliminates smell

### § Duplicate Code



# Remove Control Flag

## Problem

You have a boolean variable that acts as a control flag for multiple boolean expressions.

## Solution

Instead of the variable, use `break`, `continue` and `return`.

## Why Refactor

Control flags date back to the days of yore, when “proper” programmers always had one entry point for their functions (the function declaration line) and one exit point (at the very end of the function).

In modern programming languages this style tic is obsolete, since we have special operators for modifying the control flow in loops and other complex constructions:

- `break` : stops loop
- `continue` : stops execution of the current loop branch and goes to check the loop conditions in the next iteration

- `return` : stops execution of the entire function and returns its result if given in the operator

## Benefits

Control flag code is often much more ponderous than code written with control flow operators.

## How to Refactor

1. Find the value assignment to the control flag that causes the exit from the loop or current iteration.
2. Replace it with `break` , if this is an exit from a loop; `continue` , if this is an exit from an iteration, or `return` , if you need to return this value from the function.
3. Remove the remaining code and checks associated with the control flag.



# Replace Nested Conditional with Guard Clauses

## Problem

You have a group of nested conditionals and it's hard to determine the normal flow of code execution.

```
1 public double getPayAmount() {  
2     double result;  
3     if (isDead){  
4         result = deadAmount();  
5     }  
6     else {  
7         if (isSeparated){  
8             result = separatedAmount();  
9         }  
10        else {  
11            if (isRetired){  
12                result = retiredAmount();  
13            }  
14            else{  
15                result = normalPayAmount();  
16            }  
17        }  
18    }  
19}
```

```
17      }
18  }
19  return result;
20 }
```

## Solution

Isolate all special checks and edge cases into separate clauses and place them before the main checks. Ideally, you should have a “flat” list of conditionals, one after the other.

```
1 public double getPayAmount() {
2     if (isDead){
3         return deadAmount();
4     }
5     if (isSeparated){
6         return separatedAmount();
7     }
8     if (isRetired){
9         return retiredAmount();
10    }
11    return normalPayAmount();
12 }
```

## Why Refactor

Spotting the “conditional from hell” is fairly easy. The indentations of each level of nestedness form an arrow, pointing to the right in the direction of pain and woe:

```
1
2  if () {
3      if () {
4          do {
5              if () {
6                  if () {
7                      if () {
8                          ...
9                      }
10                     }
11                     ...
12                 }
13                 ...
14             }
15             while ();
16             ...
17         }
18     else {
19         ...
20     }
21 }
```

It's difficult to figure out what each conditional does and how, since the "normal" flow of code execution isn't immediately obvious. These conditionals indicate helter-skelter evolution, with each condition added as a stopgap measure without any thought paid to optimizing the overall structure.

To simplify the situation, isolate the special cases into separate conditions that immediately end execution and return a

null value if the guard clauses are true. In effect, your mission here is to make the structure flat.

## How to Refactor

Try to rid the code of side effects – **Separate Query from Modifier** may be helpful for the purpose. This solution will be necessary for the reshuffling described below.

1. Isolate all guard clauses that lead to calling an exception or immediate return of a value from the method. Place these conditions at the beginning of the method.
2. After rearrangement is complete and all tests are successfully completed, see whether you can use **Consolidate Conditional Expression** for guard clauses that lead to the same exceptions or returned values.



# Replace Conditional with Polymorphism

## Problem

You have a conditional that performs various actions depending on object type or properties.

```
1  class Bird {  
2      //...  
3      double getSpeed() {  
4          switch (type) {  
5              case EUROPEAN:  
6                  return getBaseSpeed();  
7              case AFRICAN:  
8                  return getBaseSpeed() - getLoadFactor() * number_of_coconuts;  
9              case NORWEGIAN_BLUE:  
10                  return (isNailed) ? 0 : getBaseSpeed(voltage);  
11          }  
12          throw new RuntimeException("Should be unreachable");  
13      }  
14  }
```

## Solution

Create subclasses matching the branches of the conditional. In them, create a shared method and move code from the corresponding branch of the conditional to it. Then replace the conditional with the relevant method call. The result is that the proper implementation will be attained via polymorphism depending on the object class.

```
1 abstract class Bird {
2     //...
3     abstract double getSpeed();
4 }
5
6 class European extends Bird {
7     double getSpeed() {
8         return getBaseSpeed();
9     }
10 }
11 class African extends Bird {
12     double getSpeed() {
13         return getBaseSpeed() - getLoadFactor() * number0fCoconuts;
14     }
15 }
16 class NorwegianBlue extends Bird {
17     double getSpeed() {
18         return (isNailed) ? 0 : getBaseSpeed(voltage);
19     }
20 }
21
22 // Somewhere in client code
```

```
23 speed = bird.getSpeed();
```

## Why Refactor

This refactoring technique can help if your code contains operators performing various tasks that vary based on:

- Class of the object or interface that it implements
- Value of an object's field
- Result of calling one of an object's methods

If a new object property or type appears, you will need to search for and add code in all similar conditionals. Thus the benefit of this technique is multiplied if there are multiple conditionals scattered throughout all of an object's methods.

## Benefits

- This technique adheres to the *Tell-Don't-Ask* principle: instead of asking an object about its state and then performing actions based on this, it's much easier to simply tell the object what it needs to do and let it decide for itself how to do that.
- Removes duplicate code. You get rid of many almost identical conditionals.

- If you need to add a new execution variant, all you need to do is add a new subclass without touching the existing code (*Open/Closed Principle*).

## How to Refactor

### Preparing to Refactor

For this refactoring technique, you should have a ready hierarchy of classes that will contain alternative behaviors. If you don't have a hierarchy like this, create one. Other techniques will help to make this happen:

- **Replace Type Code with Subclasses.** Subclasses will be created for all values of a particular object property. This approach is simple but less flexible since you can't create subclasses for the other properties of the object.
- **Replace Type Code with State/Strategy.** A class will be dedicated for a particular object property and subclasses will be created from it for each value of the property. The current class will contain references to the objects of this type and delegate execution to them.

The following steps assume that you have already created the hierarchy.

## Refactoring Steps

1. If the conditional is in a method that performs other actions as well, perform **Extract Method**.
2. For each hierarchy subclass, redefine the method that contains the conditional and copy the code of the corresponding conditional branch to that location.
3. Delete this branch from the conditional.
4. Repeat replacement until the conditional is empty. Then delete the conditional and declare the method abstract.

## Eliminates smell

### § Switch Statements



# Introduce Null Object

## Problem

Since some methods return `null` instead of real objects, you have many checks for `null` in your code.

```
1 if (customer == null) {  
2     plan = BillingPlan.basic();  
3 }  
4 else {  
5     plan = customer.getPlan();  
6 }
```

## Solution

Instead of `null`, return a null object that exhibits the default behavior.

```
1 class NullCustomer extends Customer {  
2     booleanisNull() {  
3         return true;  
4     }
```

```
5     Plan getPlan() {  
6         return new NullPlan();  
7     }  
8     // Some other NULL functionality.  
9 }  
10  
11 // Replace null values with Null-object.  
12 customer = (order.customer != null) ?  
13     order.customer : new NullCustomer();  
14  
15 // Use Null-object as if it's normal subclass.  
16 plan = customer.getPlan();
```

## Why Refactor

Dozens of checks for `null` make your code longer and uglier.

## Drawbacks

The price of getting rid of conditionals is creating yet another new class.

## How to Refactor

1. From the class in question, create a subclass that will perform the role of null object.
2. In both classes, create the method `isNull()`, which will return `true` for a null object and `false` for a real class.

3. Find all places where the code may return `null` instead of a real object. Change the code so that it returns a null object.
4. Find all places where the variables of the real class are compared with `null`. Replace these checks with a call for `isNull()`.
5.
  - If methods of the original class are run in these conditionals when a value doesn't equal `null`, redefine these methods in the null class and insert the code from the `else` part of the condition there. Then you can delete the entire conditional and differing behavior will be implemented via polymorphism.
  - If things aren't so simple and the methods can't be redefined, see if you can simply extract the operators that were supposed to be performed in the case of a `null` value to new methods of the null object. Call these methods instead of the old code in `else` as the operations by default.

## Similar refactorings

- § Replace Conditional with Polymorphism

Implements design pattern

- § Null-object

Eliminates smell

- § Switch Statements

- § Temporary Field



# Introduce Assertion

## Problem

For a portion of code to work correctly, certain conditions or values must be true.

```
1 double getExpenseLimit() {  
2     // should have either expense limit or a primary project  
3     return (expenseLimit != NULL_EXPENSE) ?  
4         expenseLimit:  
5         primaryProject.getMemberExpenseLimit();  
6 }
```

## Solution

Replace these assumptions with specific assertion checks.

```
1 double getExpenseLimit() {  
2     Assert.isTrue(expenseLimit != NULL_EXPENSE || primaryProject != null)  
3  
4     return (expenseLimit != NULL_EXPENSE) ?  
5         expenseLimit:  
6         primaryProject.getMemberExpenseLimit();  
7 }
```

## Why Refactor

Say that a portion of code assumes something about, for example, the current condition of an object or value of a parameter or local variable. Usually this assumption will always hold true except in the event of an error.

Make these assumptions obvious by adding corresponding assertions. As with type hinting in method parameters, these assertions can act as live documentation for your code.

As a guideline to see where your code needs assertions, check for comments that describe the conditions under which a particular method will work.

## Benefits

If an assumption isn't true and the code therefore gives the wrong result, it's better to stop execution before this causes fatal consequences and data corruption. This also means that you neglected to write a necessary test when devising ways to perform testing of the program.

## Drawbacks

- Sometimes an exception is more appropriate than a simple assertion. You can select the necessary class of the exception and let the remaining code handle it correctly.

- When is an exception better than a simple assertion? If the exception can be caused by actions of the user or system and you can handle the exception. On the other hand, ordinary unnamed and unhandled exceptions are basically equivalent to simple assertions – you don't handle them and they're caused exclusively as the result of a program bug that never should have occurred.

## How to Refactor

When you see that a condition is assumed, add an assertion for this condition in order to make sure.

Adding the assertion shouldn't change the program's behavior.

Don't overdo it with use of assertions for **everything** in your code. Check for only the conditions that are necessary for correct functioning of the code. If your code is working normally even when a particular assertion is false, you can safely remove the assertion.

## Eliminates smell

### § Comments

# 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

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

**Solution:** Rename the method.

## § 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

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

**Solution:** Remove the unused parameter.

## § 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

**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

**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

**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

**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

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

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

## § 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

**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

**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

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

**Solution:** Throw an exception instead.

## § **Replace Exception with Test**

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

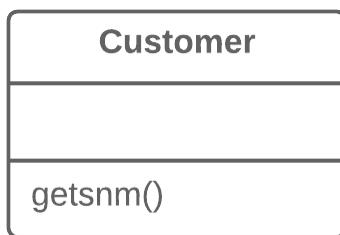
**Solution:** Replace the exception with a condition test.



# Rename Method

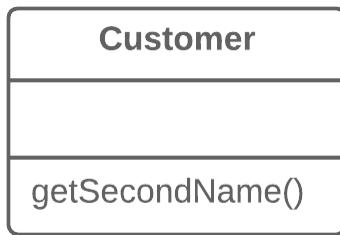
## Problem

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.

## Similar refactorings

- § Add Parameter
- § Remove Parameter

### Eliminates smell

- § Alternative Classes with Different Interfaces
- § Comments



# Add Parameter

## Problem

A method doesn't have enough data to perform certain actions.



## Solution

Create a new parameter to pass the necessary data.



## Why Refactor

You need to make changes to a method and these changes require adding information or data that was previously not available to the method.

## Benefits

The choice here is between adding a new parameter and adding a new private field that contains the data needed by the method. A field is preferable when you need some occasional or frequently changing data for which there's no point in holding it in an object all of the time. In this case, a new parameter will be a better fit than a private field and the refactoring will pay off. Otherwise, add a private field and fill it with the necessary data before calling the method.

## Drawbacks

- Adding a new parameter is always easier than removing it, which is why parameter lists frequently balloon to grotesque sizes. This smell is known as the **Long Parameter List**.
- If you need to add a new parameter, sometimes this means that your class doesn't contain the necessary data or the existing parameters don't contain the necessary related data. In both cases, the best solution is to consider moving data to the main class or to other classes whose objects are already accessible from inside the method.

## How to Refactor

1. See whether the method is defined in a superclass or subclass. If the method is present in them, you will need to repeat all the steps in these classes as well.

2. The following step is critical for keeping your program functional during the refactoring process. Create a new method by copying the old one and add the necessary parameter to it. Replace the code for the old method with a call to the new method. You can plug in any value to the new parameter (such as `null` for objects or a zero for numbers).
3. Find all references to the old method and replace them with references to the new method.
4. Delete the old method. Deletion isn't possible if the old method is part of the public interface. If that's the case, mark the old method as deprecated.

## Anti-refactoring

### § Remove Parameter

## Similar refactorings

### § Rename Method

## Helps other refactorings

### § Introduce Parameter Object



# Remove Parameter

## Problem

A parameter isn't used in the body of a method.



## Solution

Remove the unused parameter.



## Why Refactor

Every parameter in a method call forces the programmer reading it to figure out what information is found in this parameter.

And if a parameter is entirely unused in the method body, this “noggin scratching” is for naught.

And in any case, additional parameters are extra code that has to be run.

Sometimes we add parameters with an eye to the future, anticipating changes to the method for which the parameter might be needed. All the same, experience shows that it's better to add a parameter only when it's genuinely needed. After all, anticipated changes often remain just that – anticipated.

## Benefits

A method contains only the parameters that it truly requires.

## When Not to Use

If the method is implemented in different ways in subclasses or in a superclass, and your parameter is used in those implementations, leave the parameter as-is.

## How to Refactor

1. See whether the method is defined in a superclass or subclass. If so, is the parameter used there? If the parameter is used in one of these implementations, hold off on this refactoring technique.

2. The next step is important for keeping the program functional during the refactoring process. Create a new method by copying the old one and delete the relevant parameter from it. Replace the code of the old method with a call to the new one.
3. Find all references to the old method and replace them with references to the new method.
4. Delete the old method. Don't perform this step if the old method is part of a public interface. In this case, mark the old method as deprecated.

## Anti-refactoring

§ Add Parameter

## Similar refactorings

§ Rename Method

## Helps other refactorings

§ Replace Parameter with Method Call

## Eliminates smell

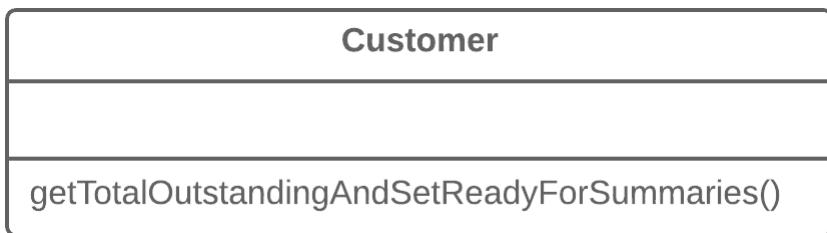
§ Speculative Generality



# Separate Query from Modifier

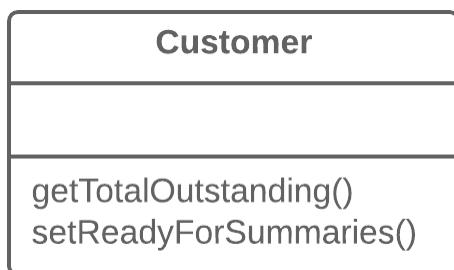
## Problem

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.



## Why Refactor

This factoring technique implements *Command and Query Responsibility Segregation*. This principle tells us to separate code responsible for getting data from code that changes something inside an object.

Code for getting data is named a *query*. Code for changing things in the *visible state* of an object is named a *modifier*. When a *query* and *modifier* are combined, you don't have a way to get data without making changes to its condition. In other words, you ask a question and can change the answer even as it's being received. This problem becomes even more severe when the person calling the query may not know about the method's "side effects", which often leads to runtime errors.

But remember that side effects are dangerous only in the case of *modifiers* that change the **visible** state of an object. These could be, for example, fields accessible from an object's public interface, entry in a database, in files, etc. If a *modifier* only caches a complex operation and saves it within the private field of a class, it can hardly cause any side effects.

## Benefits

If you have a *query* that doesn't change the state of your program, you can call it as many times as you like without having to worry about unintended changes in the result caused by the mere fact of you calling the method.

## Drawbacks

In some cases it's convenient to get data after performing a command. For example, when deleting something from a database you want to know how many rows were deleted.

## How to Refactor

1. Create a new *query method* to return what the original method did.
2. Change the original method so that it returns only the result of calling the new *query method*.
3. Replace all references to the original method with a call to the *query method*. Immediately before this line, place a call to the *modifier method*. This will save you from side effects in case if the original method was used in a condition of a conditional operator or loop.
4. Get rid of the value-returning code in the original method, which now has become a proper *modifier method*.

## Helps other refactorings

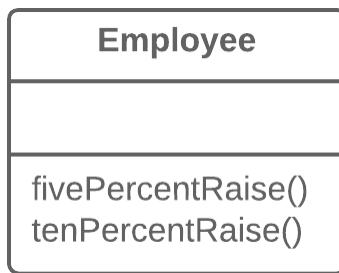
### § Replace Temp with Query



# Parameterize Method

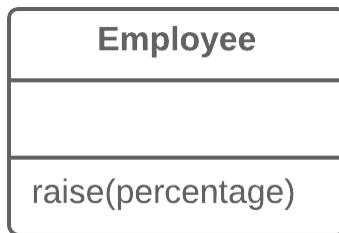
## Problem

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.



## Why Refactor

If you have similar methods, you probably have duplicate code, with all the consequences that this entails.

What's more, if you need to add yet another version of this functionality, you will have to create yet another method. Instead, you could simply run the existing method with a different parameter.

## Drawbacks

- Sometimes this refactoring technique can be taken too far, resulting in a long and complicated common method instead of multiple simpler ones.
- Also be careful when moving activation/deactivation of functionality to a parameter. This can eventually lead to creation of a large conditional operator that will need to be treated via Replace Parameter with Explicit Methods.

## How to Refactor

1. Create a new method with a parameter and move it to the code that's the same for all classes, by applying Extract Method. Note that sometimes only a certain part of methods is actually the same. In this case, refactoring consists of extracting only the same part to a new method.

2. In the code of the new method, replace the special/differing value with a parameter.
3. For each old method, find the places where it's called, replacing these calls with calls to the new method that include a parameter. Then delete the old method.

## Anti-refactoring

### § Replace Parameter with Explicit Methods

### Similar refactorings

#### § Extract Method

#### § Form Template Method

### Eliminates smell

#### § Duplicate Code



# Replace Parameter with Explicit Methods

## Problem

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

```
1 void setValue(String name, int value) {  
2     if (name.equals("height")) {  
3         height = value;  
4         return;  
5     }  
6     if (name.equals("width")) {  
7         width = value;  
8         return;  
9     }  
10    Assert.shouldNeverReachHere();  
11 }
```

## Solution

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

```
1 void setHeight(int arg) {  
2     height = arg;  
3 }  
4 void setWidth(int arg) {  
5     width = arg;  
6 }
```

## 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.

## Anti-refactoring

### § Parameterize Method

## Similar refactorings

### § Replace Conditional with Polymorphism

## Eliminates smell

### § Switch Statements

### § Long Method



# Preserve Whole Object

## Problem

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

```
1 int low = daysTempRange.getLow();
2 int high = daysTempRange.getHigh();
3 boolean withinPlan = plan.withinRange(low, high);
```

## Solution

Instead, try passing the whole object.

```
1 boolean withinPlan = plan.withinRange(daysTempRange);
```

## Why Refactor

The problem is that each time before your method is called, the methods of the future parameter object must be called. If these methods or the quantity of data obtained for the method

are changed, you will need to carefully find a dozen such places in the program and implement these changes in each of them.

After you apply this refactoring technique, the code for getting all necessary data will be stored in one place – the method itself.

## Benefits

- Instead of a hodgepodge of parameters, you see a single object with a comprehensible name.
- If the method needs more data from an object, you won't need to rewrite all the places where the method is used – merely inside the method itself.

## Drawbacks

Sometimes this transformation causes a method to become less flexible: previously the method could get data from many different sources but now, because of refactoring, we're limiting its use to only objects with a particular interface.

## How to Refactor

1. Create a parameter in the method for the object from which you can get the necessary values.

2. Now start removing the old parameters from the method one by one, replacing them with calls to the relevant methods of the parameter object. Test the program after each replacement of a parameter.
3. Delete the getter code from the parameter object that had preceded the method call.

## Similar refactorings

- § Introduce Parameter Object
- § Replace Parameter with Method Call

## Eliminates smell

- § Primitive Obsession
- § Long Parameter List
- § Long Method
- § Data Clumps



# 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.

```
1 int basePrice = quantity * itemPrice;  
2 double seasonDiscount = this.getSeasonalDiscount();  
3 double fees = this.getFees();  
4 double finalPrice = discountedPrice(basePrice, seasonDiscount, fees);
```

## Solution

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

```
1 int basePrice = quantity * itemPrice;  
2 double finalPrice = discountedPrice(basePrice);
```

## Why Refactor

A long list of parameters is hard to understand. In addition, calls to such methods often resemble a series of cascades, with winding and exhilarating value calculations that are hard to navigate yet have to be passed to the method. So if a parameter value can be calculated with the help of a method, do this inside the method itself and get rid of the parameter.

## Benefits

We get rid of unneeded parameters and simplify method calls. Such parameters are often created not for the project as it's now, but with an eye for future needs that may never come.

## Drawbacks

You may need the parameter tomorrow for other needs... making you rewrite the method.

## How to Refactor

1. Make sure that the value-getting code doesn't use parameters from the current method, since they'll be unavailable from inside another method. If so, moving the code isn't possible.
2. If the relevant code is more complicated than a single method or function call, use **Extract Method** to isolate this code in a new method and make the call simple.

3. In the code of the main method, replace all references to the parameter being replaced with calls to the method that gets the value.
4. Use **Remove Parameter** to eliminate the now-unused parameter.



# Introduce Parameter Object

## Problem

Your methods contain a repeating group of parameters.

Customer
amountInvoicedIn (start : Date, end : Date) amountReceivedIn (start : Date, end : Date) amountOverdueIn (start : Date, end : Date)

## Solution

Replace these parameters with an object.

Customer
amountInvoicedIn (date : DateRange) amountReceivedIn (date : DateRange) amountOverdueIn (date : DateRange)

## 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.

## 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**, 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** or **Extract Method**.

## Similar refactorings

### § Preserve Whole Object

## Eliminates smell

### § Long Parameter List

### § Data Clumps

### § Primitive Obsession

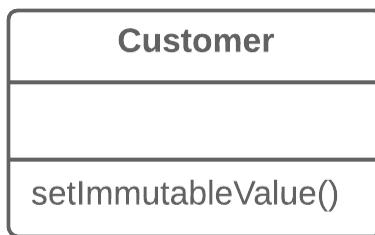
### § Long Method



# Remove Setting Method

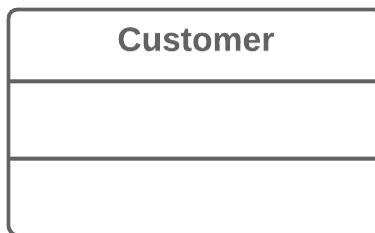
## Problem

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.



## Why Refactor

You want to prevent any changes to the value of a field.

## How to Refactor

1. The value of a field should be changeable only in the constructor. If the constructor doesn't contain a parameter for setting the value, add one.
2. Find all setter calls.
  - If a setter call is located right after a call for the constructor of the current class, move its argument to the constructor call and remove the setter.
  - Replace setter calls in the constructor with direct access to the field.
3. Delete the setter.

## Helps other refactorings

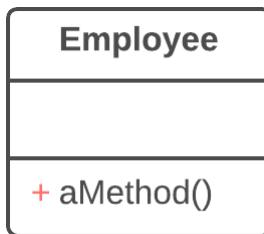
### § Change Reference to Value



# Hide Method

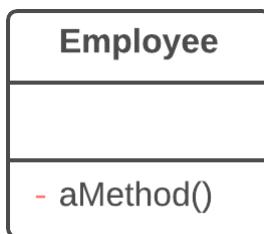
## Problem

A method isn't used by other classes or is used only inside its own class hierarchy.



## Solution

Make the method private or protected.



## Why Refactor

Quite often, the need to hide methods for getting and setting values is due to development of a richer interface that provides

additional behavior, especially if you started with a class that added little beyond mere data encapsulation.

As new behavior is built into the class, you may find that public getter and setter methods are no longer necessary and can be hidden. If you make getter or setter methods private and apply direct access to variables, you can delete the method.

## Benefits

- Hiding methods makes it easier for your code to evolve. When you change a private method, you only need to worry about how to not break the current class since you know that the method can't be used anywhere else.
- By making methods private, you underscore the importance of the public interface of the class and of the methods that remain public.

## How to Refactor

1. Regularly try to find methods that can be made private. Static code analysis and good unit test coverage can offer a big leg up.
2. Make each method as private as possible.

## Eliminates smell

### § Data Class



# Replace Constructor with Factory Method

## Problem

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

```
1 class Employee {  
2     Employee(int type) {  
3         this.type = type;  
4     }  
5     //...  
6 }
```

## Solution

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

```
1 class Employee {  
2     static Employee create(int type) {  
3         employee = new Employee(type);  
4         // do some heavy lifting.  
5         return employee;
```

```
6      }
7  //...
8 }
```

## Why Refactor

The most obvious reason for using this refactoring technique is related to Replace Type Code with Subclasses.

You have code in which a object was previously created and the value of the coded type was passed to it. After use of the refactoring method, several subclasses have appeared and from them you need to create objects depending on the value of the coded type. Changing the original constructor to make it return subclass objects is impossible, so instead we create a static factory method that will return objects of the necessary classes, after which it replaces all calls to the original constructor.

Factory methods can be used in other situations as well, when constructors aren't up to the task. They can be important when attempting to Change Value to Reference. They can also be used to set various creation modes that go beyond the number and types of parameters.

## Benefits

- A factory method doesn't necessarily return an object of the class in which it was called. Often these could be its subclasses, selected based on the arguments given to the method.
- A factory method can have a better name that describes what and how it returns what it does, for example `Troops::GetCrew(myTank)`.
- A factory method can return an already created object, unlike a constructor, which always creates a new instance.

## How to Refactor

1. Create a factory method. Place a call to the current constructor in it.
2. Replace all constructor calls with calls to the factory method.
3. Declare the constructor private.
4. Investigate the constructor code and try to isolate the code not directly related to constructing an object of the current class, moving such code to the factory method.

## Helps other refactorings

- § Change Value to Reference
- § Replace Type Code with Subclasses

## Implements design pattern

- § Factory Method



# Replace Error Code with Exception

## Problem

A method returns a special value that indicates an error?

```
1 int withdraw(int amount) {  
2     if (amount > _balance) {  
3         return -1;  
4     }  
5     else {  
6         balance -= amount;  
7         return 0;  
8     }  
9 }
```

## Solution

Throw an exception instead.

```
1 void withdraw(int amount) throws BalanceException {  
2     if (amount > _balance) {  
3         throw new BalanceException();  
4     }
```

```
5     balance -= amount;  
6 }
```

## Why Refactor

Returning error codes is an obsolete holdover from procedural programming. In modern programming, error handling is performed by special classes, which are named exceptions. If a problem occurs, you “throw” an error, which is then “caught” by one of the exception handlers. Special error-handling code, which is ignored in normal conditions, is activated to respond.

## Benefits

- Freed code from a large number of conditionals for checking various error codes. Exception handlers are a much more succinct way to differentiate normal execution paths from abnormal ones.
- Exception classes can implement their own methods, thus containing part of the error handling functionality (such as for sending error messages).
- Unlike exceptions, error codes can't be used in a constructor, since a constructor must return only a new object.

## Drawbacks

An exception handler can turn into a goto-like crutch. Avoid this! Don't use exceptions to manage code execution. Exceptions should be thrown only to inform of an error or critical situation.

## How to Refactor

Try to perform these refactoring steps for only one error code at a time. This will make it easier to keep all the important information in your head and avoid errors.

1. Find all calls to a method that returns error codes and, instead of checking for an error code, wrap it in `try / catch` blocks.
2. Inside the method, instead of returning an error code, throw an exception.
3. Change the method signature so that it contains information about the exception being thrown (`@throws` section).



# Replace Exception with Test

## Problem

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

```
1 double getValueForPeriod(int periodNumber) {  
2     try {  
3         return values[periodNumber];  
4     } catch (ArrayIndexOutOfBoundsException e) {  
5         return 0;  
6     }  
7 }
```

## Solution

Replace the exception with a condition test.

```
1 double getValueForPeriod(int periodNumber) {  
2     if (periodNumber >= values.length) {  
3         return 0;  
4     }
```

```
5     return values[periodNumber];  
6 }
```

## Why Refactor

Exceptions should be used to handle irregular behavior related to an unexpected error. They shouldn't serve as a replacement for testing. If an exception can be avoided by simply verifying a condition before running, then do so. Exceptions should be reserved for real errors.

For instance, you entered a minefield and triggered a mine there, resulting in an exception; the exception was successfully handled and you were lifted through the air to safety beyond the mine field. But you could have avoided this all by simply reading the warning sign in front of the minefield to begin with.

## Benefits

A simple conditional can sometimes be more obvious than exception handling code.

## How to Refactor

1. Create a conditional for an edge case and move it before the try/catch block.

2. Move code from the `catch` section inside this conditional.
3. In the `catch` section, place the code for throwing a usual unnamed exception and run all the tests.
4. If no exceptions were thrown during the tests, get rid of the `try / catch` operator.

## Similar refactorings

### § Replace Error Code with Exception

# 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

**Problem:** Two classes have the same field.

**Solution:** Remove the field from subclasses and move it to the superclass.

## § 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.

## § Pull Up Constructor Body

**Problem:** Your subclasses have constructors with code that's mostly identical.

**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.

## § **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.

## § **Push Down Field**

**Problem:** Is a field used only in a few subclasses?

**Solution:** Move the field to these subclasses.

## § **Extract Subclass**

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

**Solution:** Create a subclass and use it in these cases.

## § **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.

## § 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.

## § 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.

## § 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.

## § 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.

## § Replace Delegation with Inheritance

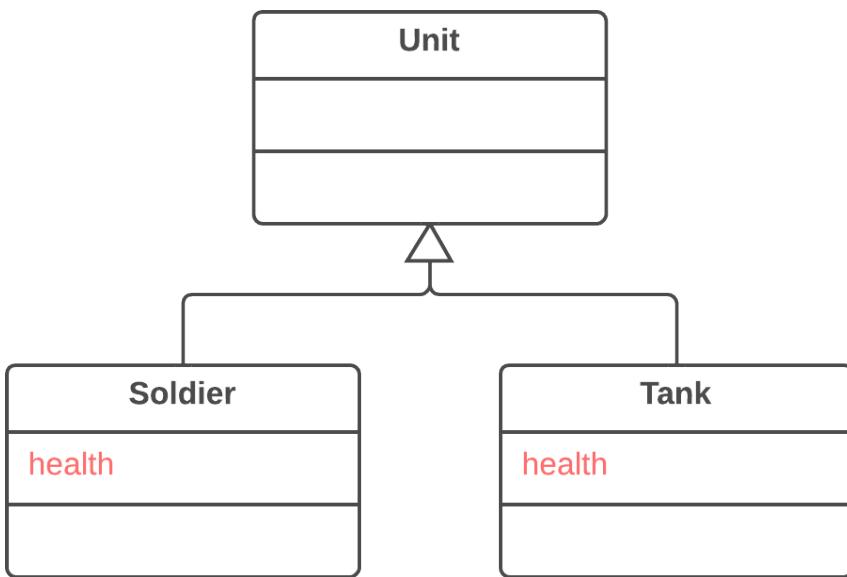
**Problem:** A class contains many simple methods that delegate to all methods of another class.

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

# Pull Up Field

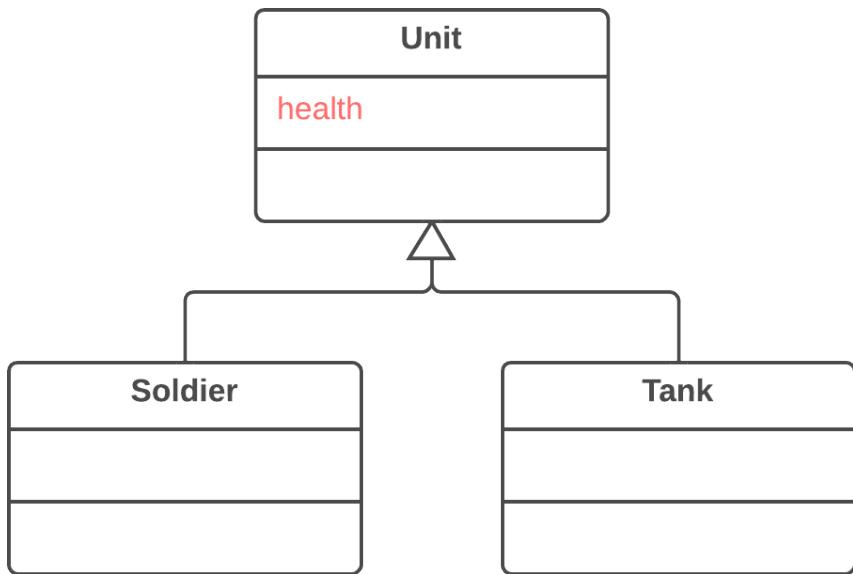
## Problem

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** for the new field, in order to hide it behind access methods.

## Anti-refactoring

### § Push Down Field

## Similar refactorings

### § Pull Up Method

## Eliminates smell

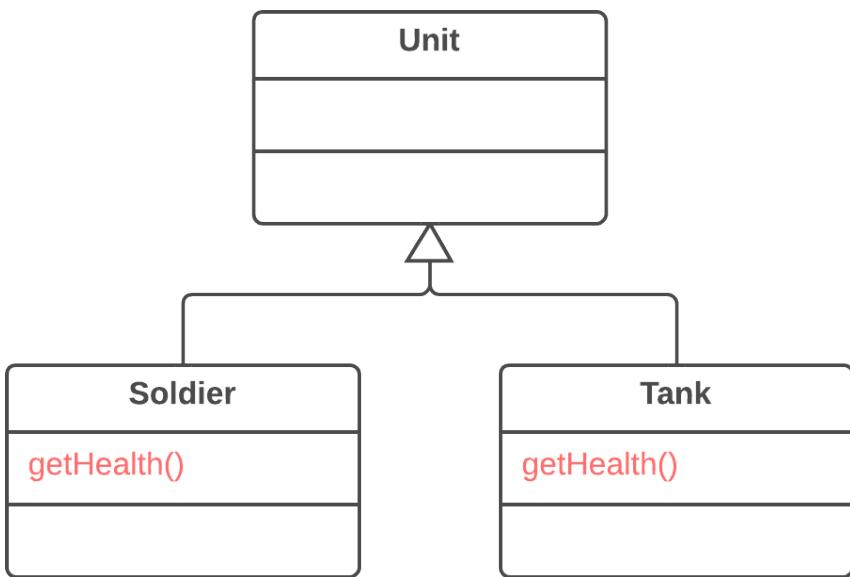
### § Duplicate Code



# Pull Up Method

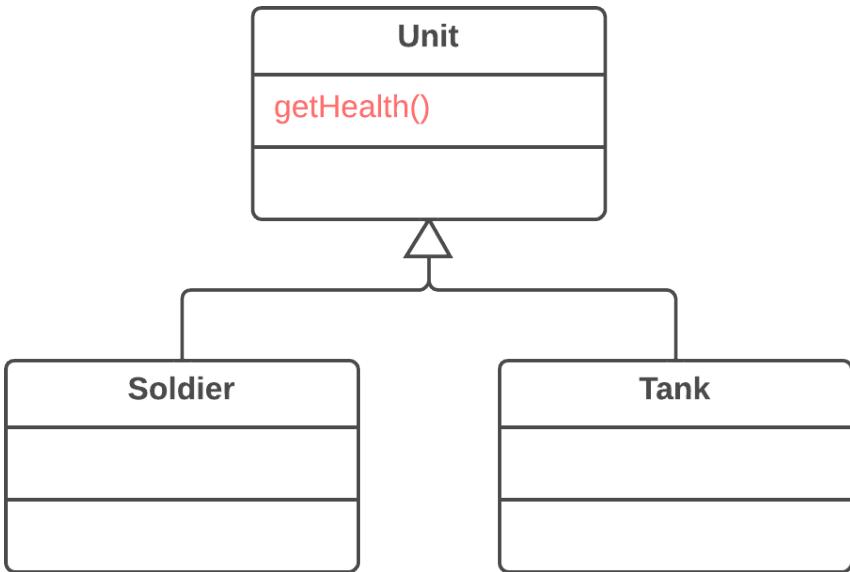
## Problem

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** or **Self-Encapsulate Field** to create getters and setters in subclasses; then declare these getters abstractly in the superclass.
  - For methods: use either **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.

## Anti-refactoring

### § Push Down Method

Similar refactorings

### § Pull Up Field

Helps other refactorings

### § Form Template Method

Eliminates smell

### § Duplicate Code



# Pull Up Constructor Body

## Problem

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

```
1 class Manager extends Employee {  
2     public Manager(String name, String id, int grade) {  
3         this.name = name;  
4         this.id = id;  
5         this.grade = grade;  
6     }  
7     //...  
8 }
```

## 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.

```
1 class Manager extends Employee {
```

```
2 public Manager(String name, String id, int grade) {  
3     super(name, id);  
4     this.grade = grade;  
5 }  
6 //...  
7 }
```

## Why Refactor

How is this refactoring technique different from Pull Up Method?

1. In Java, subclasses can't inherit a constructor, so you can't simply apply 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.

## Similar refactorings

### § Pull Up Method

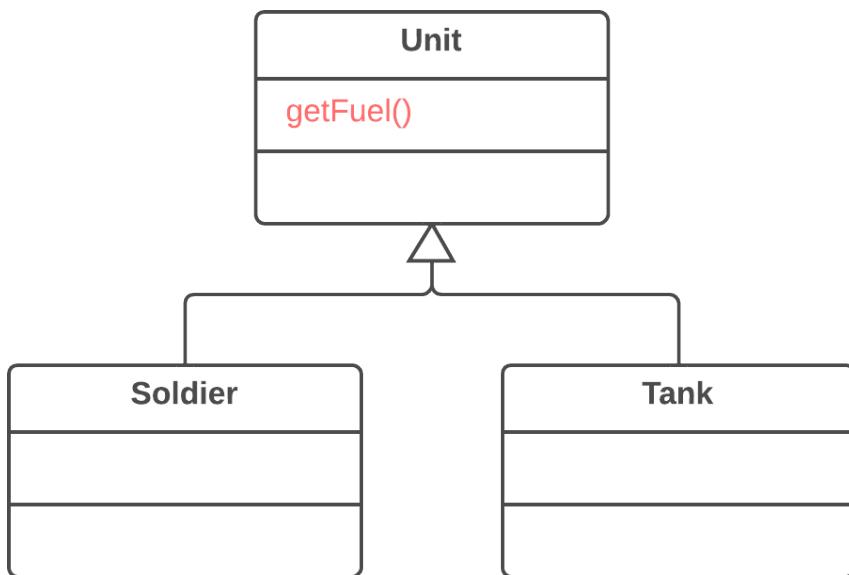
### Eliminates smell

### § Duplicate Code

# Push Down Method

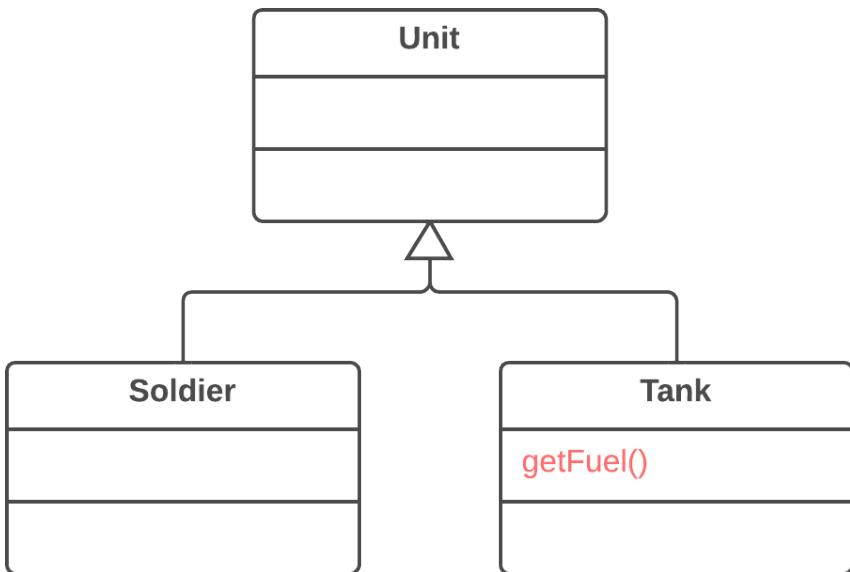
## Problem

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.

## Anti-refactoring

### § Pull Up Method

Similar refactorings

### § Push Down Field

Helps other refactorings

### § Extract Subclass

Eliminates smell

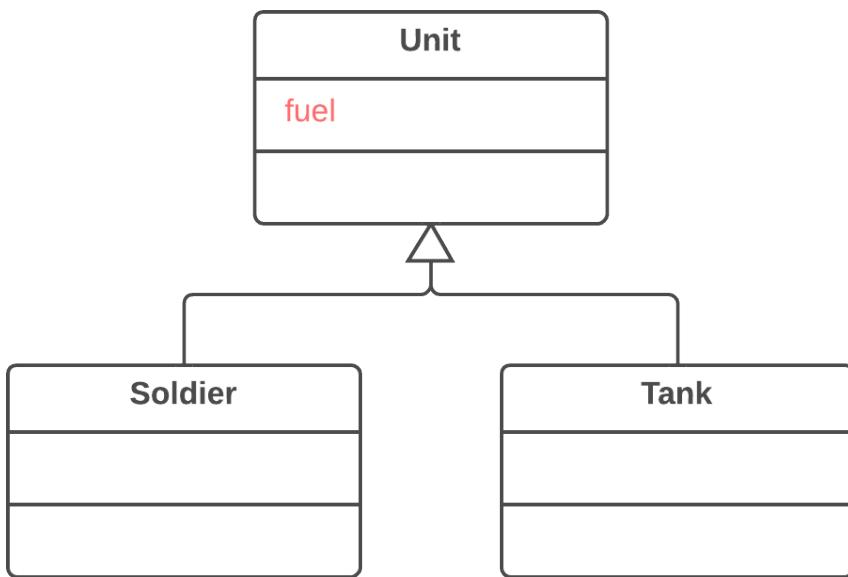
### § Refused Bequest



# Push Down Field

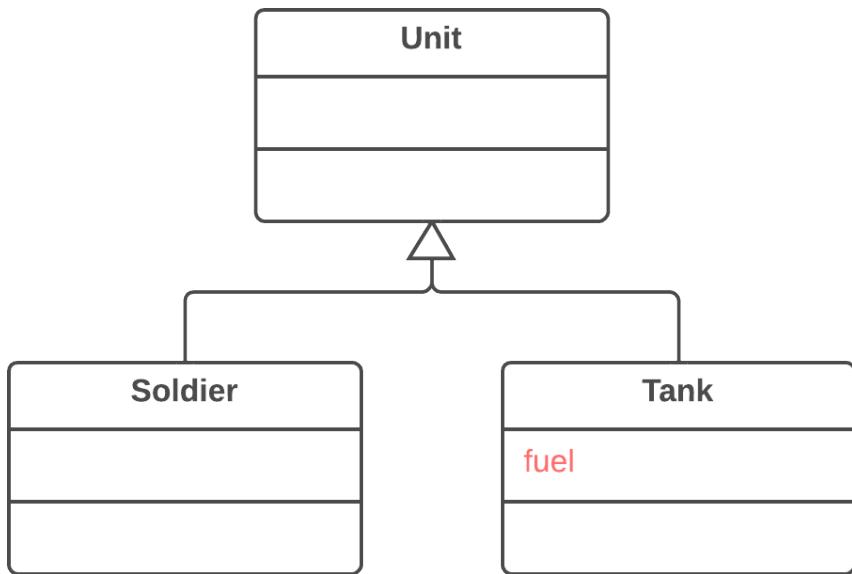
## Problem

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.

## Anti-refactoring

### § Pull Up Field

## Similar refactorings

### § Push Down Method

## Helps other refactorings

### § Extract Subclass

## Eliminates smell

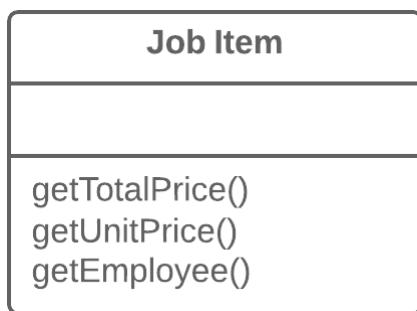
### § Refused Bequest



# Extract Subclass

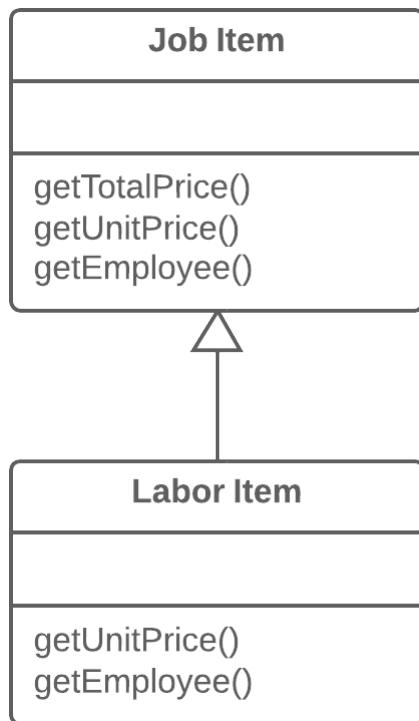
## Problem

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** 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** and **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.

## Similar refactorings

§ Extract Class

**Eliminates smell**

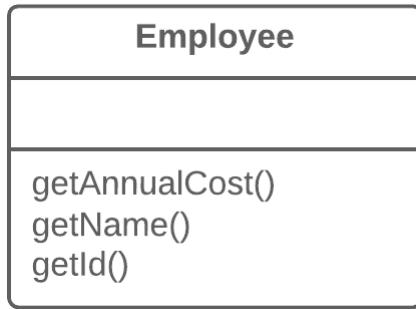
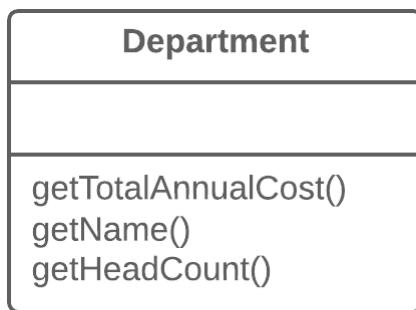
§ Large Class



# Extract Superclass

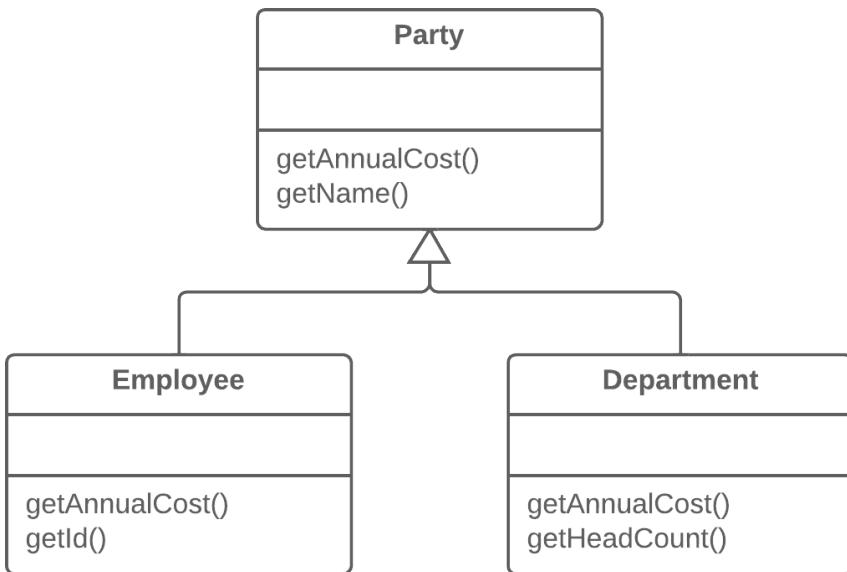
## Problem

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**, **Pull Up Method**, and **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).

## Similar refactorings

### § Extract Interface

## Eliminates smell

### § Duplicate Code

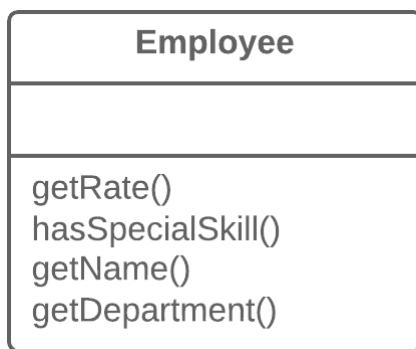


# Extract Interface

## Problem

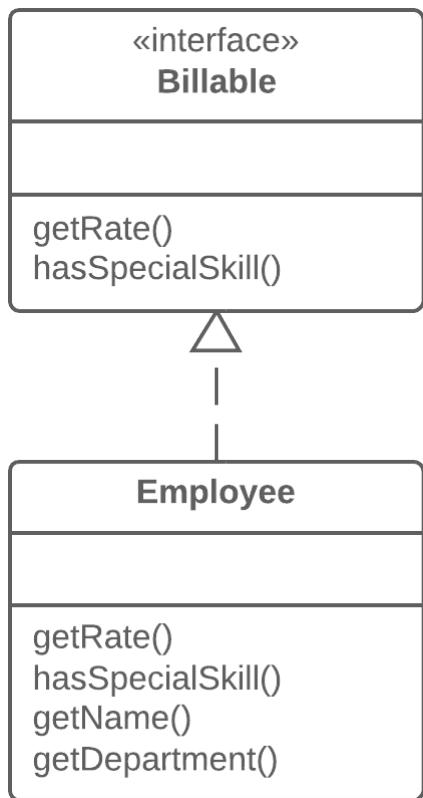
Multiple clients are using the same part of a class interface.

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** 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** and **Extract Interface**.

Extracting an interface allows isolating only common interfaces, not common code. In other words, if classes contain **Duplicate Code**, extracting the interface won't help you to deduplicate.

All the same, this problem can be mitigated by applying **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**. 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.

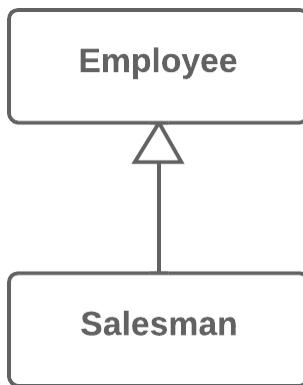
## Similar refactorings

### § Extract Superclass

# 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** and **Pull Up Method** if you decide to get rid of the subclass. If you choose to eliminate the superclass, go for **Push Down Field** and **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.

## Similar refactorings

### § Inline Class

*Collapse Hierarchy* is a variation of Inline Class, where the code moves to superclass or subclass.

## Eliminates smell

### § Lazy Class

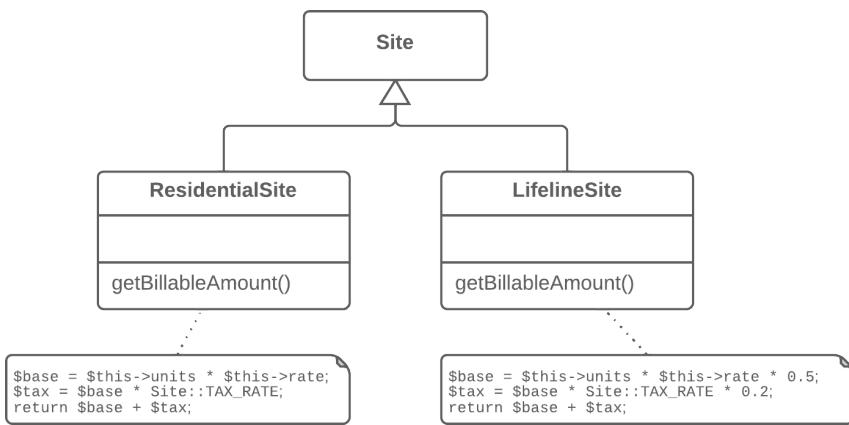
### § Speculative Generality



# Form Template Method

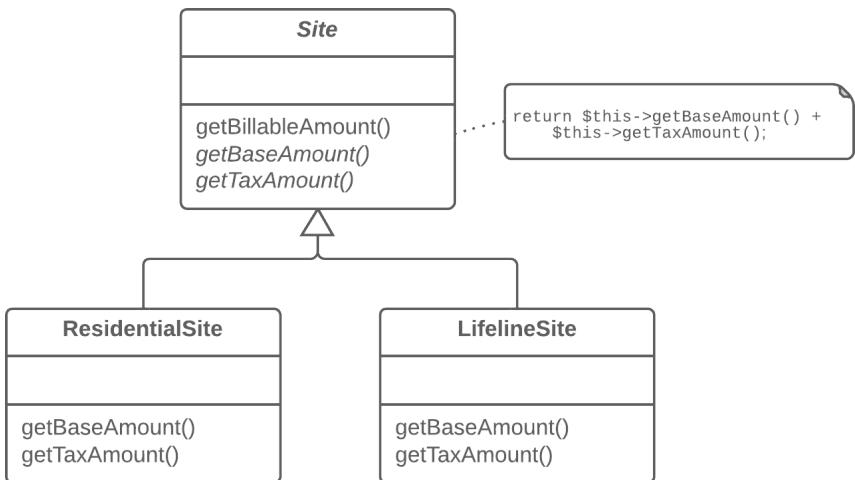
## Problem

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** can help with this.
2. The resulting methods that are identical for all subclasses can be moved to a superclass via **Pull Up Method**.
3. The non-similar methods can be given consistent names via **Rename Method**.
4. Move the signatures of non-similar methods to a superclass as abstract ones by using **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.

## Implements design pattern

§ Template Method

## Eliminates smell

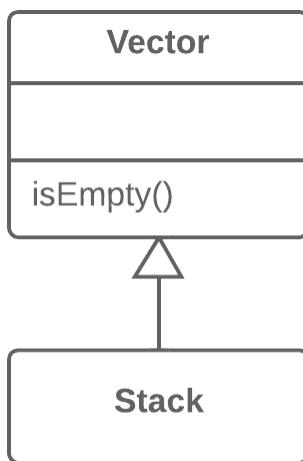
§ Duplicate Code



# Replace Inheritance with Delegation

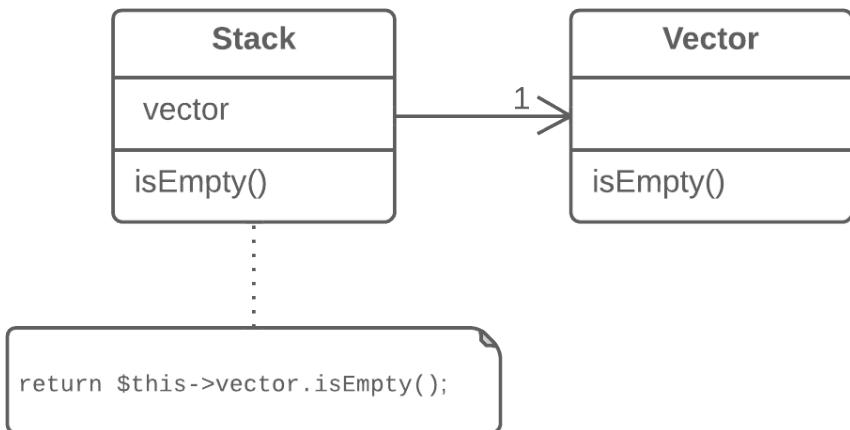
## Problem

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** 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.

## Anti-refactoring

### § Replace Delegation with Inheritance

Implements design pattern

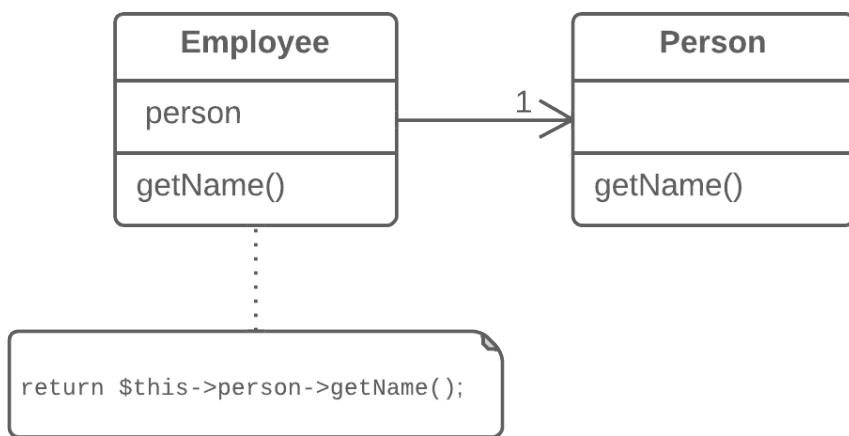
### § Strategy



# Replace Delegation with Inheritance

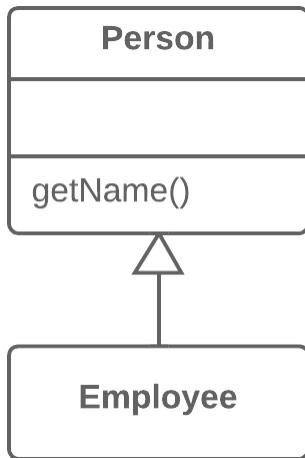
## Problem

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 stop 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** 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.

## Anti-refactoring

### § Replace Inheritance with Delegation

#### Similar refactorings

### § Remove Middle Man

#### Eliminates smell

### § Inappropriate Intimacy

# Afterword

## Congrats on reaching the last page!

Now that you know so much about refactoring, what are you planning to do? Here's a bunch of ideas, if you still haven't decided:

-  Read Joshua Kerievsky's book "[\*\*Refactoring to Patterns\*\*](#)".
-  What? Don't you know what are the design patterns? [Read our section about design patterns](#)
-  Print out the [\*\*refactoring cheat sheet\*\*](#) and pin it somewhere you'll always see it.
-  [\*\*Submit your feedback\*\*](#) about this book and the full course. I'd be glad to hear any feedback, even if it's negative 😊