**Decompose Conditional**

**Problem**

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

**if** (date.before(SUMMER\_START) || date.after(SUMMER\_END)) {

charge = quantity \* winterRate + winterServiceCharge;

}

**else** {

charge = quantity \* summerRate;

}

**Solution**

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

**if** (isSummer(date)) {

charge = summerCharge(quantity);

}

**else** {

charge = winterCharge(quantity);

}

**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**](https://refactoring.guru/extract-method).
2. Repeat the process for the then and else blocks.

**Consolidate Conditional Expression**

**Problem**

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

**double** **disabilityAmount**() {

**if** (seniority < 2) {

**return** 0;

}

**if** (monthsDisabled > 12) {

**return** 0;

}

**if** (isPartTime) {

**return** 0;

}

// Compute the disability amount.

// ...

}

**Solution**

Consolidate all these conditionals in a single expression.

**double** **disabilityAmount**() {

**if** (isNotEligibleForDisability()) {

**return** 0;

}

// Compute the disability amount.

// ...

}

**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**](https://refactoring.guru/extract-method) on the operator conditions and give the method a name that reflects the expression’s purpose.

**Consolidate Duplicate Conditional Fragments**

**Problem**

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

**if** (isSpecialDeal()) {

total = price \* 0.95;

send();

}

**else** {

total = price \* 0.98;

send();

}

**Solution**

Move the code outside of the conditional.

**if** (isSpecialDeal()) {

total = price \* 0.95;

}

**else** {

total = price \* 0.98;

}

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

**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.

**public** **double** **getPayAmount**() {

**double** result;

**if** (isDead){

result = deadAmount();

}

**else** {

**if** (isSeparated){

result = separatedAmount();

}

**else** {

**if** (isRetired){

result = retiredAmount();

}

**else**{

result = normalPayAmount();

}

}

}

**return** result;

}

**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.

**public** **double** **getPayAmount**() {

**if** (isDead){

**return** deadAmount();

}

**if** (isSeparated){

**return** separatedAmount();

}

**if** (isRetired){

**return** retiredAmount();

}

**return** normalPayAmount();

}

**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:

if () {

if () {

do {

if () {

if () {

if () {

...

}

}

...

}

...

}

while ();

...

}

else {

...

}

}

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**](https://refactoring.guru/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**](https://refactoring.guru/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.

**class** **Bird** {

// ...

**double** getSpeed() {

**switch** (type) {

**case** EUROPEAN:

**return** getBaseSpeed();

**case** AFRICAN:

**return** getBaseSpeed() - getLoadFactor() \* numberOfCoconuts;

**case** NORWEGIAN\_BLUE:

**return** (isNailed) ? 0 : getBaseSpeed(voltage);

}

**throw** **new** RuntimeException("Should be unreachable");

}

}

### 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.

**abstract** **class** **Bird** {

// ...

**abstract** **double** getSpeed();

}

**class** **European** **extends** Bird {

**double** getSpeed() {

**return** getBaseSpeed();

}

}

**class** **African** **extends** Bird {

**double** getSpeed() {

**return** getBaseSpeed() - getLoadFactor() \* numberOfCoconuts;

}

}

**class** **NorwegianBlue** **extends** Bird {

**double** getSpeed() {

**return** (isNailed) ? 0 : getBaseSpeed(voltage);

}

}

// Somewhere in client code

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**](https://refactoring.guru/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**](https://refactoring.guru/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**](https://refactoring.guru/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.

**Introduce Null Object**

**Problem**

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

**if** (customer == **null**) {

plan = BillingPlan.basic();

}

**else** {

plan = customer.getPlan();

}

**Solution**

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

[**Java**](https://refactoring.guru/introduce-null-object#java)[C#](https://refactoring.guru/introduce-null-object#csharp)[PHP](https://refactoring.guru/introduce-null-object#php)[Python](https://refactoring.guru/introduce-null-object#python)[TypeScript](https://refactoring.guru/introduce-null-object#typescript)

**class** **NullCustomer** **extends** Customer {

**boolean** isNull() {

**return** **true**;

}

Plan getPlan() {

**return** **new** NullPlan();

}

// Some other NULL functionality.

}

// Replace null values with Null-object.

customer = (order.customer != **null**) ?

order.customer : **new** NullCustomer();

// Use Null-object as if it's normal subclass.

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().
   * 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.

**Introduce Assertion**

**Problem**

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

**double** **getExpenseLimit**() {

// Should have either expense limit or

// a primary project.

**return** (expenseLimit != NULL\_EXPENSE) ?

expenseLimit :

primaryProject.getMemberExpenseLimit();

}

**Solution**

Replace these assumptions with specific assertion checks.

**double** **getExpenseLimit**() {

Assert.isTrue(expenseLimit != NULL\_EXPENSE || primaryProject != **null**);

**return** (expenseLimit != NULL\_EXPENSE) ?

expenseLimit:

primaryProject.getMemberExpenseLimit();

}

**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.