Chapter 7: Simplifying Conditional Expressions

Conditional logic has a way of getting tricky, so here are a number of refactorings you can use to simplify it. The core refactoring here is *Decompose Conditional (136)* which talks about how to decompose the conditional into pieces. In many ways it is an obvious refactoring, but it's one that often makes a huge difference to the clarity of code.

The other refactorings here look into other important cases. Use *Consolidate Conditional Expression (139)* when you have several tests all with the same effect, use *Consolidate duplicate conditional fragments (142)* to remove any duplication within the conditional code.

If you are working with code developed in a one-exit point mentality you often find control flags which are there to allow the conditions to work with this rule. I don't follow the rule about one exit point from a method. Hence I'll use *Replace Nested Conditional with Guard Clauses (149)* to clarify special case conditionals and *Remove Control Flag (144)* to get rid of the awkward control flags.

Object-oriented programs often have less conditional behavior than procedural programs because much of the conditional behavior is handled by polymorphism. Polymorphism is better because the caller does not need to know about the conditional behavior and it is thus easier to extend the conditions. As a result object-oriented programs rarely have switch (case) statements. So any that show up are prime candidates for *Replace Conditional with Polymorphism (154)*.

One of the most useful, but less obvious, uses of polymorphism is to use *Introduce Null Object (159)* to remove checks for a null value.

Decompose Conditional

You have a complicated conditional (if-then-else) statement

Extract methods from the condition, then part, and else parts.

```
if (date.before (SUMMER_START) || date.after(SUMMER_END))
    charge = quantity * _winterRate + _winterServiceCharge;
else charge = quantity * _summerRate;
```

$$\Downarrow$$

```
if (notSummer(date))
    charge = winterCharge(quantity);
else charge = summerCharge (quantity);
```

Motivation

One of the most common areas of complexity in a program lies in complex conditional logic. As you write code to test conditions, and to do various things depending on various conditions, you quickly end up with a pretty long method. Length of a method is in itself a factor that makes it harder to read, but conditions increase the difficulty. The problem usually lies in the fact that the code, both in the condition checks and in the actions, tells you what happens but can easily obscure the why.

As with any large block of code, you can make your intention clearer by decomposing it, replacing chunks of code with a method call that is named after the intention of that block of code. With conditions you can get a further benefit by doing this for the conditional part and each of the alternatives. This way you highlight the condition and make it clearly what you are branching on, and why you are doing the branching.

Mechanics

- ❍ Extract the condition into its own method
- ❍ Extract the 'then part' and the 'else part' into their own methods

If I find a nested conditional I will usually first look to see if I should *Replace Nested Conditional with Guard Clauses (149).* If that does not make sense I will decompose each of the conditionals.

Example

Suppose you are calculating the charge for something that has a separate rate for winter and for summer

```
        if (date.before (SUMMER_START) || date.after(SUMMER_END))
            charge = quantity * _winterRate + _winterServiceCharge;
        else charge = quantity * _summerRate;
```

Extract the conditional and each leg into

```
        if (notSummer(date))
            charge = winterCharge(quantity);
        else charge = summerCharge (quantity);

    private boolean notSummer(Date date) {
        return date.before (SUMMER_START) || date.after(SUMMER_END);
    }

    private double summerCharge(int quantity) {
        return quantity * _summerRate;
    }

    private double winterCharge(int quantity) {
        return quantity * _winterRate + _winterServiceCharge;
    }
```

Here I've shown the result of the complete refactoring for clarity. In practice, however, I would do each extraction separately, compiling and testing after each one.

Many people don't extract the condition parts in situations like this. The conditions are often quite short, so it hardly seems worth it. But although the condition is often short, there is often a big gap between the intention of the code and its body. Even in this little case reading (notSummer(date)) conveys more clearly to me than the original code. With the original I have to look at the code and figure out what it is doing. It’s not difficult to do that here, but even so the extracted method reads more like a comment.

Consolidate Conditional Expression

You have a sequence of conditional tests with the same result

Combine them into a single conditional expression, and extract it

```
double disabilityAmount() {
    if (_seniority < 2) return 0;
    if (_monthsDisabled > 12) return 0;
    if (_isPartTime) return 0;
    // compute the disability amount
```

$$\Downarrow$$

```
double disabilityAmount() {
    if (isEligableForDisability()) return 0;
    // compute the disability amount
```

Motivation

Sometimes you see a series of conditional checks, where each check is different yet the resulting action is the same. When you see this you should consolidate them into a single conditional check, using ands and ors, with the single result.

Consolidating the conditional code is important for two reasons. Firstly it makes the check clearer by showing that you are really making a single check that's or'ing the other checks together. The sequence has the same effect but it communicates carrying out a sequence of separate checks that just happen to be done together. The second reason for this refactoring is that it often sets you up to do *Extract Method (114)*. Extracting a condition is one of the most useful things you can do to clarify your code, for it replaces a statement of what you are doing with why you are doing it.

Those reasons in favor of consolidating conditionals also point to when you shouldn't do it. If you think the checks are really independent, and shouldn't be thought of as a single check; then don't do the refactoring. Your code already communicates your intention.

Mechanics

❍ Replace the string of conditionals with a single conditional statement using logical operators

❍ Compile and test

❍ Consider using *Extract Method (114)* on the condition.

Example: ors

The state of the code for this is along the lines of the following.

```
double disabilityAmount() {
    if (_seniority < 2) return 0;
    if (_monthsDisabled > 12) return 0;
    if (_isPartTime) return 0;
    // compute the disability amount
    ...
```

Here we see a sequence of conditional checks that all result in the same thing. With sequential code like this they are the equivalent of an or statement.

```
double disabilityAmount() {
    if ((_seniority < 2) || (_monthsDisabled > 12) || (_isPartTime)) return 0;
    // compute the disability amount
    ...
```

Now I can look at the condition and use *Extract Method (114)* to communicate what the condition is looking for.

```
double disabilityAmount() {
    if (isEligableForDisability()) return 0;
    // compute the disability amount
    ...
}

boolean isEligableForDisability() {
    return ((_seniority < 2) || (_monthsDisabled > 12) || (_isPartTime));
}
```

Example: ands

That example showed ors, but you can do the same with ands. Here the set up is something like

```
if (onVacation())
    if (lengthOfService() > 10)
        return 1;
return 0.5;
```

This would be changed to

```
if (onVacation() && lengthOfService() > 10) return 1;
else return 0.5;
```

You may well find you get a combination of these that yield an expression with ands ors and nots.

If the routine you are looking at only tests the condition and returns a value; then you can turn the routine into a single return statement using the tertiary operator. So

```
if (onVacation() && lengthOfService() > 10) return 1;
else return 0.5;
```

becomes

```
return (onVacation() && lengthOfService() > 10) ? 1 : 0.5;
```

In this case the condition speaks for itself as clearly as any method name would — so I won't extract it.

Consolidate duplicate conditional fragments

The same fragment of code is in all branches of a conditional expression

Move it outside of the expression

```
if (isSpecialDeal()) {
    total = price * 0.95;
    send();
}
else {
    total = price * 0.98;
    send();
}
```

⇓

```
if (isSpecialDeal())
    total = price * 0.95;
else
    total = price * 0.98;
send();
```

Motivation

Sometimes you will find the same code executed in all legs of a conditional. In that case you should move the code to outside the conditional. This makes it clearer as it what varies, and what stays the same.

Mechanics

- ❍ Identify code that gets executed the same regardless of the condition
- ❍ If the common code is at the beginning, then move it to before the conditional
- ❍ If it is at the end, move it to after
- ❍ If it is in the middle look to see if the code before or after it changes

anything. If so you can move the common code forwards or backwards to the ends you can then move it as above

❍ If there is more than a single statement then you should extract that code into a method.

Example

The kind of situation you find this is with code like

```
if (isSpecialDeal()) {
    total = price * 0.95;
    send();
}
else {
    total = price * 0.98;
    send();
}
```

Since the send method is executed in either case you should move it out of the conditional.

```
if (isSpecialDeal())
    total = price * 0.95;
else
    total = price * 0.98;
send();
```

The same situation can apply to exceptions. If code is repeated after an exception causing statement in the try block and all the catch blocks, then you can move it to the finally block.

Remove Control Flag

You have a control flag that is acting as a control flag cfor a series of boolean expression.

Use a break or return instead

Motivation

When you have a series of conditional expressions, you often see a control flag used to determine when to stop looking.

```
set done to false
while not done
  if (condition)
      do something
      set done to true
  next step of loop
```

Such control flags are more trouble than they are worth. They come from the rules of structured programming that call for routines with one entry and one exit point. I agree with (and modern languages enforce) one entry point, but the one exit point rule leads you to very convoluted conditionals with these awkward flags in the code. This is why languages have the break and continue statements to get out of a complex conditional. It is often surprising what you can do when you get rid of a control flag, the real purpose of the conditional becomes so much more clear.

Mechanics

The obvious way to deal with this is using the break or continue statements present in Java.

❍ Find the value of the control flag that get's you out of the logic statement

❍ Replace assignments of the break out value with a break or continue statement

  ☞ Break is used to end processing in that code fragment, continue is used to take another trip round a loop.

❍ Compile and test after each replacement

Another approach, also usable in languages without break and continue

- ❍ Extract the logic into a method
- ❍ Find the value of the control flag that get's you out of the logic statement
- ❍ Replace assignments of the break out value with a return
- ❍ Compile and test after each replacement

Indeed even in languages with a break or continue, I usually prefer the use of a extraction and the use of return. The return clearly signals that no more code in the method gets executed. Often if you have that kind of code, you need to extract that piece anyway.

Keep an eye on whether the control flag also indicates some result information. If so you still need it if you use the break, or you can return the value if you have extracted a method.

Example - simple control flag replaced with break

The following function checks to see if a list of people contains a couple of hard coded suspicious characters.

```
void checkSecurity(String[] people) {
    boolean found = false;
    for (int i = 0; i < people.length; i++) {
        if (! found) {
           if (people[i].equals ("Don")){
              sendAlert();
              found = true;
           }
           if (people[i].equals ("John")){
              sendAlert();
              found = true;
           }
        }
    }
}
```

In a case like this it is easy to see the control flag, it's the piece that sets the `found` variable to true. I can introduce the breaks one at a time

```
void checkSecurity(String[] people) {
    boolean found = false;
    for (int i = 0; i < people.length; i++) {
        if (! found) {
           if (people[i].equals ("Don")){
              sendAlert();
```

```
                  break;
               }
               if (people[i].equals ("John")){
                  sendAlert();
                  found = true;
               }
           }
       }
   }
```

Until I have them all.

```
   void checkSecurity(String[] people) {
       boolean found = false;
       for (int i = 0; i < people.length; i++) {
           if (! found) {
               if (people[i].equals ("Don")){
                  sendAlert();
                  break;
               }
               if (people[i].equals ("John")){
                  sendAlert();
                  break;
               }
           }
       }
   }
```

Then I can remove all references to the control flag

```
   void checkSecurity(String[] people) {
       for (int i = 0; i < people.length; i++) {
           if (people[i].equals ("Don")){
               sendAlert();
               break;
           }
           if (people[i].equals ("John")){
               sendAlert();
               break;
           }
       }
   }
```

Example - using return with a control flag result

The other style of refactoring uses a return. I'll illustrate this with a variant that uses the control flag as a result value.

```
   void checkSecurity(String[] people) {
       String found = "";
       for (int i = 0; i < people.length; i++) {
            if (found.equals("")) {
               if (people[i].equals ("Don")){
                  sendAlert();
                  found = "Don";
               }
               if (people[i].equals ("John")){
                  sendAlert();
                  found = "John";
               }
            }
        }
        someLaterCode(found);
    }
```

Here found is doing two things, it is both indicating a result and acting as a control flag. When I see this I like to extract the code that is determining found into its own method.

```
    void checkSecurity(String[] people) {
        String found = foundMiscreant(people);
        someLaterCode(found);
    }

    String foundMiscreant(String[] people){
        String found = "";
        for (int i = 0; i < people.length; i++) {
            if (found.equals("")) {
               if (people[i].equals ("Don")){
                  sendAlert();
                  found = "Don";
               }
               if (people[i].equals ("John")){
                  sendAlert();
                  found = "John";
               }
            }
        }
        return found;
    }
```

Then I can replace the control flag with a return.

```
    String foundMiscreant(String[] people){
        String found = "";
        for (int i = 0; i < people.length; i++) {
            if (found.equals("")) {
               if (people[i].equals ("Don")){
                  sendAlert();
                  return "Don";
               }
               if (people[i].equals ("John")){
                sendAlert();
                found = "John";
             }
          }
       }
       return found;
    }
```

Until I have removed the control flag.

```
    String foundMiscreant(String[] people){
       for (int i = 0; i < people.length; i++) {
          if (people[i].equals ("Don")){
             sendAlert();
             return "Don";
          }
          if (people[i].equals ("John")){
             sendAlert();
             return "John";
          }
       }
       return "";
    }
```

You can also use the return style when you're not returning a value. Just use return without the argument.

Of course this has the problem of a function with side-effects. So I want to *Separate Query from Modifier (237)*. You'll find this example continued there....

Replace Nested Conditional with Guard Clauses

A method has conditional behavior that does not make clear what the normal path of execution is

Use Guard Clauses for all the special cases

```
double getPayAmount() {
  double result;
  if (_isDead) result = deadAmount();
  else {
      if (_isSeparated) result = separatedAmount();
      else {
          if (_isRetired) result = retiredAmount();
          else result = normalPayAmount();
      };
  }
return result;
};
```

$$\Downarrow$$

```
double getPayAmount() {
  if (_isDead) return deadAmount();
  if (_isSeparated) return separatedAmount();
  if (_isRetired) return retiredAmount();
  return normalPayAmount();
};
```

Motivation

I often find that conditional expressions come in two forms. The first form is a check whether either course is part of the normal behavior, the second case is where one answer from the conditional indicates normal behavior and the other indicates an unusual condition.

These kinds of conditionals have a different intention, and this intention should come through in the code. If both are part of normal behavior, then use a condition with an if and an else leg. However if the condition is an unusual condition then check the condition and return if the condition is true. This kind of check is often called a *Guard Clause* [Beck].

The key point about this refactoring is one of emphasis. If you are using an if-then-else construct you are giving equal weight to the if leg and the else leg. This communicates to the reader that they are equally likely and important. Instead the guard clause says “this is rare, and if it happens do something and get out”.

I often find I use this refactoring when I’m working with a programmer who has been taught to have only one entry and one exit point from a method. One entry point is enforced by modern languages, and one exit point is really not a useful rule. Clarity is the key principle: if it is clearer with one exit point then use one exit point, otherwise don’t.

Mechanics

- ❍ For each check put the guard clause in.
  - ☞ The guard clause will either return, or throw an exception.
- ❍ Compile and test after each check is replaced with a guard clause.
  - ☞ If all the guard clauses yield the same result then *Consolidate the Conditional Expressions*.

Example

Imagine a run of a payroll system where you have special rules for dead, separated, and retired employees. Such cases are unusual but they do happen from time to time.

If you write the code like this

```
double getPayAmount() {
  double result;
  if (_isDead) result = deadAmount();
  else {
      if (_isSeparated) result = separatedAmount();
      else {
          if (_isRetired) result = retiredAmount();
          else result = normalPayAmount();
      };
  }
return result;
};
```

Then the checking is masking the normal course of action behind the checking. So instead it is clearer to use guard clauses. I can introduce these one at a time. I like to go from the top.

```
double getPayAmount() {
  double result;
  if (_isDead) return deadAmount();
  if (_isSeparated) result = separatedAmount();
  else {
      if (_isRetired) result = retiredAmount();
      else result = normalPayAmount();
  };
  return result;
};
```

I then continue one at a time

```
double getPayAmount() {
  double result;
  if (_isDead) return deadAmount();
  if (_isSeparated) return separatedAmount();
  if (_isRetired) result = retiredAmount();
  else result = normalPayAmount();
  return result;
};
```

and then

```
double getPayAmount() {
  double result;
  if (_isDead) return deadAmount();
  if (_isSeparated) return separatedAmount();
  if (_isRetired) return retiredAmount();
  result = normalPayAmount();
  return result;
};
```

By this point the result temp isn't pulling its weight so I nuke it.

```
double getPayAmount() {
  if (_isDead) return deadAmount();
  if (_isSeparated) return separatedAmount();
  if (_isRetired) return retiredAmount();
  return normalPayAmount();
};
```

Example: reversing the conditions

In reviewing this, Joshua Kerievsky pointed out that you often do this refactoring by reversing the conditional expressions. He kindly came up with an example to save further taxing my imagination.

```
public double getAdjustedCapital() {
  double result = 0.0;
  if (_capital > 0.0) {
    if (_intRate > 0.0 && _duration > 0.0) {
      result = (_income / _duration) * ADJ_FACTOR;
    }
  }
  return result;
}
```

Again I do this one at a time, but this time I reverse the conditional as I put in the guard clause.

```
public double getAdjustedCapital() {
    double result = 0.0;
    if (_capital <= 0.0) return result;
    if (_intRate > 0.0 && _duration > 0.0) {
      result = (_income / _duration) * ADJ_FACTOR;
    }
    return result;
}
```

As the next conditional is a bit more complicated, I can reverse it in two steps. First I just add a not.

```
public double getAdjustedCapital() {
    double result = 0.0;
    if (_capital <= 0.0) return result;
    if (!(_intRate > 0.0 && _duration > 0.0)) return result;
    result = (_income / _duration) * ADJ_FACTOR;
    return result;
}
```

Leaving nots in a conditional like that twists my mind around at a painful angle, so I simplify it.

```
public double getAdjustedCapital() {
    double result = 0.0;
    if (_capital <= 0.0) return result;
    if (_intRate <= 0.0 || _duration <= 0.0) return result;
    result = (_income / _duration) * ADJ_FACTOR;
    return result;
}
```

In these situations I prefer to put an explicit value on the returns from the guards. That way you can easily see the result of the guard failing. (I would also consider *Replace Magic Number with Symbolic Constant (217)* here.)

```
public double getAdjustedCapital() {
    double result = 0.0;
    if (_capital <= 0.0) return 0.0;
    if (_intRate <= 0.0 || _duration <= 0.0) return 0.0;
    result = (_income / _duration) * ADJ_FACTOR;
    return result;
}
```

With that done I can also remove the temp.

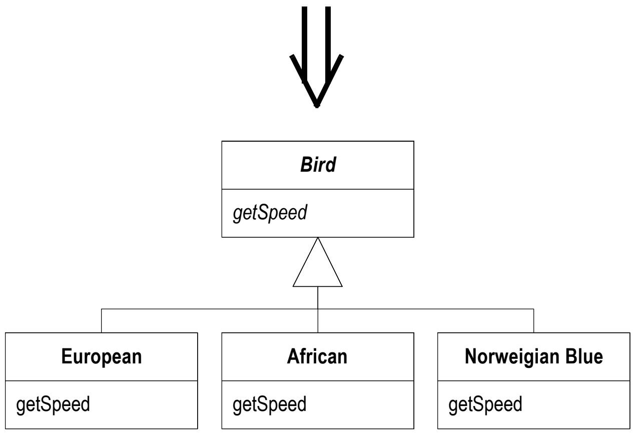
```
public double getAdjustedCapital() {
    if (_capital <= 0.0) return 0.0;
    if (_intRate <= 0.0 || _duration <= 0.0) return 0.0;
    return (_income / _duration) * ADJ_FACTOR;
}
```

Replace Conditional with Polymorphism

You have a conditional that chooses different behavior depending on the type of an object

Move each leg of the conditional to a subclass

```
double getSpeed() {
    switch (_type) {
        case EUROPEAN:
           return getBaseSpeed();
        case AFRICAN:
           return getBaseSpeed() - getLoadFactor() * _numberOfCoconuts;
        case NORWEIGIAN_BLUE:
           return (_isNailed) ? 0 : getBaseSpeed(_voltage);
    }
    throw new RuntimeException ("Should be unreachable");
}
```



Motivation

One of the grandest sounding words in object jargon is “polymorphism”. The essence of polymorphsim is that it allows you to avoid writing an explicit conditional when you have objects whose behavior varies depending on their types.

As a result you find that switch statements that switch on type codes, or if-then-else statements that switch on type strings, are much less common in an object-oriented program.

Polomorphism gives you many advantages. The biggest gain occurs when this same set of conditions appears in many places in the program. If you want to add a new type you have to find and update all the conditionals. But with subclasses you just create a new subclass and provide the appropriate methods. Clients of the class don't need to know about the subclasses, which reduces the dependencies in your system, making it easier to update.

Mechanics

Before you can begin with this refactoring you need to have the necessary inheritance structure. You may already have this structure from previous refactorings. If you don't have the structure you will need to create it.

To create the inheritance structue you have two options: *Replace Type Code with Subclasses (235)* and *Replace Type Code with State/Strategy (239)*. Subclasses are the simplest option so you should use it if you can. If you update the type code after the object is created, however, you cannot use subclassing and have to use the state / strategy pattern instead. You also need to use the state / strategy pattern if you are already subclassing this class for some other reason. Remember that if you have several case statements switching on the same type code you only need to create one inheritance structure for that type code.

You can now attack the conditional. The code you target may be a switch (case) statement or an if statement.

- ❍ If the conditional statement is one part of a larger method then take the conditional statement part and use *Extract Method (114)*
- ❍ If necessary use *Move Method (160)* to get the conditional onto the top of the inheritance structure
- ❍ Pick one of the subclasses. Create a subclass method that overrides the conditional statement method. Copy the body of that leg of the conditional statement into the subclass method, and adjust it to fit.
  - ☞ You may need to make some private members of the superclass protected in

order to do this.

- ❍ Compile and test
- ❍ Remove that leg of the conditional statement
- ❍ Compile and test.
- ❍ Repeat with each leg of the conditional statement until all legs are turned into subclass methods.
- ❍ Make the superclass method abstract

Example

I'll use the tedious and simplistic employee pay example. I'm using the classes after the using *Replace Type Code with State/Strategy (239)* so the objects look like this (see the example there for how we got here) .

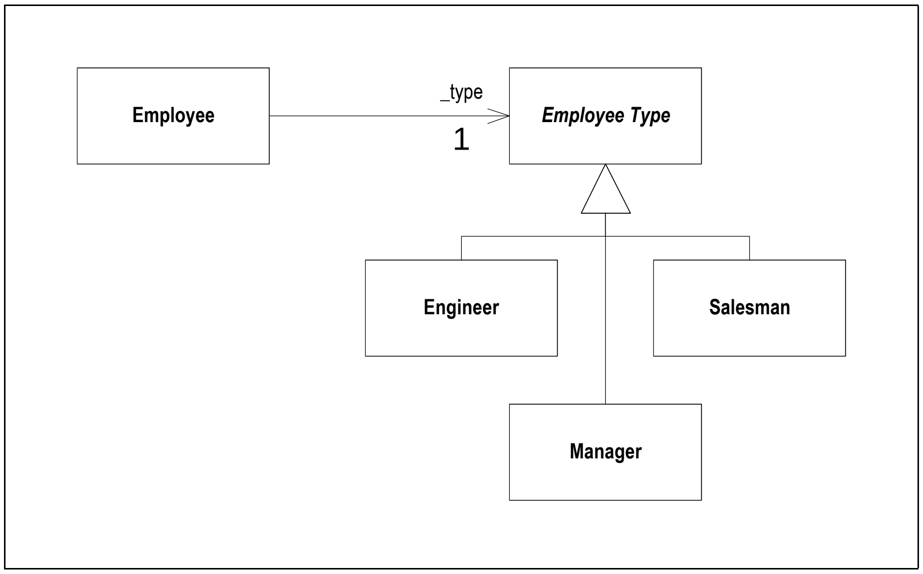


Figure 7.1: The inheritance structure

```
class Employee…
  int payAmount() {
      switch (getType()) {
          case EmployeeType.ENGINEER:
             return _monthlySalary;
          case EmployeeType.SALESMAN:
             return _monthlySalary + _commission;
          case EmployeeType.MANAGER:
             return _monthlySalary + _bonus;
```

```
            default:
               throw new RuntimeException("Incorrect Employee");
        }
    }

    int getType() {
        return _type.getTypeCode();
    }
    private EmployeeType _type;

  abstract class EmployeeType…
    abstract int getTypeCode();

  class Engineer extends EmployeeType…
    int getTypeCode() {
        return Employee.ENGINEER;
    }

  … and other subclasses
```

The case statement is already nicely extracted, so there is nothing to do there. I do need to move it into the employee type, as that is the class that is being subclassed.

```
  class EmployeeType...
    int payAmount(Employee emp) {
        switch (getTypeCode()) {
            case ENGINEER:
               return emp.getMonthlySalary();
            case SALESMAN:
               return emp.getMonthlySalary() + emp.getCommission();
            case MANAGER:
               return emp.getMonthlySalary() + emp.getBonus();
            default:
               throw new RuntimeException("Incorrect Employee");
        }
    }
```

Since I need data from the employee, I need to pass in the employee as an argument. Some of this data might be moved to the employee type object, but that is an issue for another refactoring.

When this compiles I change the payAmount method in employee to delegate to the new class.

```
  class Employee...
    int payAmount() {
        return _type.payAmount(this);
    }
```

Now I can go to work on the case statement. It's rather like the way small boys kill insects – I remove one leg at a time. First I copy the engineer leg of the case statement onto the engineer class.

```
class Engineer…
  int payAmount(Employee emp) {
      return emp.getMonthlySalary();
  }
```

This new method will override the whole case statement for engineers. If you want to be sure of that you can put in a trap in the case statement.

```
class EmployeeType…
  int payAmount(Employee emp) {
      switch (getTypeCode()) {
          case ENGINEER:
             throw new RuntimeException ("Should be being overridden");
          case SALESMAN:
             return emp.getMonthlySalary() + emp.getCommission();
          case MANAGER:
             return emp.getMonthlySalary() + emp.getBonus();
          default:
             throw new RuntimeException("Incorrect Employee");
      }
  }
```

Carry on until all the legs are removed.

```
class Salesman…
  int payAmount(Employee emp) {
      return emp.getMonthlySalary() + emp.getCommission();
  }

class Manager…
  int payAmount(Employee emp) {
      return emp.getMonthlySalary() + emp.getBonus();
  }
```

And then declare the superclass method as abstract.

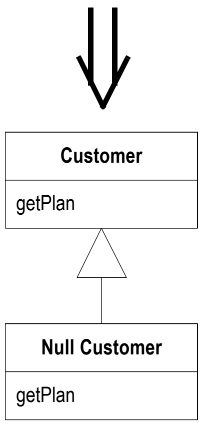
```
class EmployeeType…
  abstract int payAmount(Employee emp);
```

Introduce Null Object

You have repeated checks for a null value

Replace the null value with a null object

```
if (customer == null) plan = BillingPlan.basic();
else plan = customer.getPlan();
```



Motivation

The essence of polymorphism is that instead of asking an object what type it is and then invoking some behavior based on the answer; you just invoke the behavior. The object, depending on its type, does the right thing. One of the less intuitive places to do this is when you have a null value in a field. I'll let Ron Jeffries tell the story.

We first started using the Null Object pattern when Rich Garzaniti found that there was lots of code in the system that would check objects for presence before sending a message to it. We might ask an object for its person, then ask the result whether it was null, then if it was there, ask it for its rate. We were doing this in several places and resulting duplicate code was getting annoying.

So we implemented a missing person object that answered a zero rate (we call our null objects missing objects). Soon missing person knew a lot of methods like rate. Now we have over eighty null object classes.

Our most common use of these objects is in the display of information. When we display, say, a person, the object may or may not have any of perhaps twenty instance variables. If these were allowed to be null, the printing of a person would be very complex. Instead, we plug in various null objects, all of which know how to display themselves in an orderly way. This got rid of huge amounts of procedural code.

Our most clever use of null object was the missing Gemstone session. We use the Gemstone database for production, but we prefer to develop without it and push the new code to Gemstone every week or so. There are various points in the code where we have to log into a Gemstone session. When we are running without Gemstone we simply plug in a missing Gemstone session. It looks the same as the real thing, but allows us to develop and test without realizing the database isn t there.

Another very helpful use was the missing bin. A bin is a collection of payroll values that often need to be summed or looped over. If a particular bin doesn't exist, we answer a missing bin, which acts just like an empty bin. The missing bin knows it has zero balance and no values. By using this approach, we eliminated the creation of tens of empty bins for each of our thousands of employees.

An interesting characteristic of using null objects is that things almost never blow up. Since the null object responds to all the same messages as a real one, the system generally behaves normally. This can sometimes make it difficult to detect or find a problem, because nothing ever breaks. Of course, as soon as you being inspecting the objects, you'll find the null object somewhere where it shouldn't be.

Remember, null objects are always constant: nothing about them ever changes. Accordingly, we implement them using the Singleton pattern [Gang of Four]. Whenever you ask for, say, a missing person, you always get the single instance of that class.

- - Ron Jeffries

You can find more details about the Null Object pattern in [Woolf].

Mechanics

❍ Create a subclass of the source class to act is a null version of the class. Create an `isNull` operation on the source class and the null class. For the source class it should return false, for the null class it should return true.

☞ You may find it useful to create an explicit Nullable interface for the isNull method.

☞ As an alternative you can use a testing interface to test for nullness.

❍ Compile

❍ Find all places that can give out a null when asked for a source object. Replace them to give out a null object instead.

❍ Find all places that compare a variable of the source type to null, and replace them with a call is `isNull`.

☞ You may be able to do this by replacing one source and its clients at a time, and compiling and testing between working on sources.

☞ A few assertions that check for null in places where you should no longer see it can be useful.

❍ Compile and test

❍ Look for cases where clients invoke an operation if not null and do some alternative behavior if null.

❍ For each of these cases override the operation in the null class with the alternative behavior.

❍ Remove the condition check for those that use the overriden behavior, compile and test.

Example

A utility company knows about Sites: the houses and apartments that use the utility's services. At any time a site will have a customer.

```
class Site...
  Customer getCustomer() {
      return _customer;
  }
  Customer _customer;
```

There are various features of a customer. I'll look at three of them.

```
class Customer...
  public String getName() {...}
  public BillingPlan getPlan() {...}
  public PaymentHistory getHistory() {...}
```

The payment history has its own features.

```
public class PaymentHistory...
  int getWeeksDelinquentInLastYear()
```

The getters I've shown allow clients to get at this data. However sometimes we don't have a customer for a site. Someone may have moved out and we don't know who has moved in yet. Because this can happen we have to ensure any code that uses the customer can handle nulls. Here are a few example fragments.

```
          Customer customer = site.getCustomer();
          BillingPlan plan;
          if (customer == null) plan = BillingPlan.basic();
          else plan = customer.getPlan();
...
          String customerName;
          if (customer == null) customerName = "occupant";
          else customerName = customer.getName();
...
          int weeksDelinquent;
          if (customer == null) weeksDelinquent = 0;
          else weeksDelinquent = customer.getHistory().getWeeksDelinquentInLastYear();
```

In these situations you may have many clients of site and customer, all of which have to check for nulls and all of which do the same thing when it finds one. Sounds like its time for a null object.

The first step is to create the null customer class and modify the customer class to support a query for a null test.

```
class NullCustomer extends Customer {
  public boolean isNull() {
      return true;
  }
}

class Customer...
  public boolean isNull() {
      return false;
  }

  protected Customer() {} //needed by the NullCustomer
```

(If you aren't able to modify the Customer class you can use a testing interface, see page 165.)

If you like you can signal the use of null object by an interface

```
interface Nullable {
  boolean isNull();
}

class Customer implements Nullable
```

I like to add a factory method to create null customers. That way clients don't have to know about the null class.

```
class Customer...
  static Customer newNull() {
      return new NullCustomer();
  }
```

Now comes the difficult bit. Now we have to return this new null object whenever we expect a null, and replace the tests of the form `foo == null` with tests of the form `foo.isNull()`. I find it useful to look for all the places where you ask for a customer, and modify them so that they return a null customer rather than null.

```
class Site...
  Customer getCustomer() {
      return (_customer == null) ?
          Customer.newNull():
          _customer;
  }
```

I also have to alter all uses of this value so that they test with `isNull()` rather than `== null`.

```
      Customer customer = site.getCustomer();
      BillingPlan plan;
      if (customer.isNull()) plan = BillingPlan.basic();
      else plan = customer.getPlan();
...
      String customerName;
      if (customer.isNull()) customerName = "occupant";
      else customerName = customer.getName();
...
      int weeksDelinquent;
      if (customer.isNull()) weeksDelinquent = 0;
      else weeksDelinquent = customer.getHistory().getWeeksDelinquentInLastYear();
```

There's no doubt that this is the trickiest part of this refactoring. For each source of a null you replace, you have to find all the times it is tested for nullness and replace them. If the object is widely passed around, these can be hard to track. You have to find every variable of type customer and find everywhere it is used. It is hard to break this into small steps. Sometimes you find one source that is only used in a few places, and you can replace that source only. But most of the time you have to make many widespread changes. The changes aren't too difficult to back out, since you can find calls of isNull without too much difficulty, but this is still a messy step.

Once this step is done, and I've compiled and tested, I can smile. Now the fun begins. As it stands I gain nothing from using `isNull` rather than `== null`. The gain comes as I move behavior to the null customer and remove conditionals. I can do these moves one at a time. I begin with the name. Currently I have client code that says

```
        String customerName;
        if (customer.isNull()) customerName = "occupant";
        else customerName = customer.getName();
```

I add a suitable name method to the null customer.

```
  class NullCustomer...
    public String getName(){
        return "occupant";
    }
```

Now I can make the conditional code go away.

```
        String customerName = customer.getName();
```

I can do the same for any other method where there is a sensible general response to a query. I can also do appropriate action for modifiers. So client code such as

```
        if (! customer.isNull())
            customer.setPlan(BillingPlan.special());
```

can be replaced by

```
  customer.setPlan(BillingPlan.special());

  class NullCustomer...
    public void setPlan (BillingPlan arg) {}
```

Remember that this movement of behavior makes sense only when most clients want the same response. Notice I said "most" not "all". Any clients who want a different response to the standard one can still test using `isNull`. You benefit when many clients want to do the same thing, they can just rely on the default null behavior.

The example contains a slightly different case. Client code that uses the result of a call to customer.

```
        if (customer.isNull()) weeksDelinquent = 0;
        else weeksDelinquent = customer.getHistory().getWeeksDelinquentInLastYear();
```

I can handle this by creating a null payment history.

```
class NullPaymentHistory extends PaymentHistory...
  int getWeeksDelinquentInLastYear() {
      return 0;
  }
```

I modify the null customer to return it when asked.

```
class NullCustomer...
  public PaymentHistory getHistory() {
      return PaymentHistory.newNull();
  }
```

And again I can remove the conditional code

```
      int weeksDelinquent = customer.getHistory().getWeeksDelinquentInLastYear();
```

You often find that null objects return other null objects.

When carrying out this refactoring remember that you can have several kinds of null. Often there is a difference between there is no customer (new building and not yet moved in) and a unknown customer (we think there is someone there but we don’t know who it is). If that is the case you can build separate classes for the different null cases. Sometimes null objects can actually carry data: such as usage records for the unknown customer so that we can bill them when we find out who they are.

Example - Testing Interface

The testing interface is an alternative to defining an isNull method. In this approach you create a Null interface with no methods defined.

```
interface Null {}
```

You then implement null in your null objects.

```
class NullCustomer extends Customer implements Null...
```

You then test for nullness with the instanceof operator

```
aCustomer instanceof Null
```

I normally run away screaming from the instanceof operator, but in this case it is okay to use it. It has the particular advantage that you don’t need to change the customer class. This allows you to use the null object even when you don’t have access to customer’s source code.