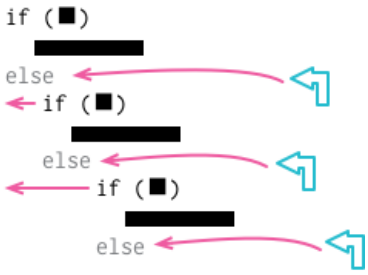
Replace Nested Conditional with Guard Clauses

previous:
Consolidate Conditional Expression

next:
Replace Conditional with Polymorphism




```
function getPayAmount() {
  let result;
  if (isDead)
    result = deadAmount();
  else {
    if (isSeparated)
      result = separatedAmount();
    else {
      if (isRetired)
        result = retiredAmount();
      else
        result = normalPayAmount();
    }
  }
  return result;
}
```

```
function 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 styles. In the first style, both legs of the conditional are part of normal behavior, while in the second style, one leg is normal and the other indicates an unusual condition.

These kinds of conditionals have different intentions—and these intentions should come through in the code. If both are part of normal behavior, I use a condition with an `if` and an `else` leg. If the condition is an unusual condition, I check the condition and return if it's true. This kind of check is often called a **guard clause**.

The key point of Replace Nested Conditional with Guard Clauses is emphasis. If I'm using an if-then-else construct, I'm giving equal weight to the `if` leg and the `else` leg. This communicates to the reader that the legs are equally likely and important. Instead, the guard clause says, “This isn't the core to this function, and if it happens, do something and get out.”

I often find I use Replace Nested Conditional with Guard Clauses when I’m working with a programmer who has been taught to have only one entry point and one exit point from a method. One entry point is enforced by modern languages, but one exit point is really not a useful rule. Clarity is the key principle: If the method is clearer with one exit point, use one exit point; otherwise don’t.

Mechanics

- Select outermost condition that needs to be replaced, and change it into a guard clause.
- Test.
- Repeat as needed.
- If all the guard clauses return the same result, use Consolidate Conditional Expression.

Example

Here's some code to calculate a payment amount for an employee. It's only relevant if the employee is still with the company, so it has to check for the two other cases.

```
function payAmount(employee) {
  let result;
  if(employee.isSeparated) {
    result = {amount: 0, reasonCode: "SEP"};
  }
  else {
    if (employee.isRetired) {
      result = {amount: 0, reasonCode: "RET"};
    }
    else {
      // logic to compute amount
      lorem.ipsum(dolor.sitAmet);
      consectetur(adipiscing).elit();
      sed.do.eiusmod = tempor.incididunt.ut(labore) && dolore(magna.aliqua);
      ut.enim.ad(minim.veniam);
      result = someFinalComputation();
    }
```

```
  }
  return result;
}
```

Nesting the conditionals here masks the true meaning of what it going on. The primary purpose of this code only applies if these conditions aren't the case. In this situation, the intention of the code reads more clearly with guard clauses.

As with any refactoring change, I like to take small steps, so I begin with the topmost condition.

```
function payAmount(employee) {
  let result;
  if (employee.isSeparated) return {amount: 0, reasonCode: "SEP"};
  if (employee.isRetired) {
    result = {amount: 0, reasonCode: "RET"};
  }
  else {
    // logic to compute amount
    lorem.ipsum(dolor.sitAmet);
    consectetur(adipiscing).elit();
    sed.do.eiusmod = tempor.incididunt.ut(labore) && dolore(magna.aliqua);
    ut.enim.ad(minim.veniam);
    result = someFinalComputation();
  }
  return result;
}
```

I test that change and move on to the next one.

```
function payAmount(employee) {
  let result;
  if (employee.isSeparated) return {amount: 0, reasonCode: "SEP"};
  if (employee.isRetired)   return {amount: 0, reasonCode: "RET"};
  // logic to compute amount
  lorem.ipsum(dolor.sitAmet);
  consectetur(adipiscing).elit();
  sed.do.eiusmod = tempor.incididunt.ut(labore) && dolore(magna.aliqua);
  ut.enim.ad(minim.veniam);
  result = someFinalComputation();
  return result;
}
```

At which point the result variable isn't really doing anything useful, so I remove it.

```
function payAmount(employee) {
  let result;
  if (employee.isSeparated) return {amount: 0, reasonCode: "SEP"};
  if (employee.isRetired)   return {amount: 0, reasonCode: "RET"};
  // logic to compute amount
  lorem.ipsum(dolor.sitAmet);
  consectetur(adipiscing).elit();
  sed.do.eiusmod = tempor.incididunt.ut(labore) && dolore(magna.aliqua);
  ut.enim.ad(minim.veniam);
  return someFinalComputation();
}
```

The rule is that you always get an extra strawberry when you remove a mutable variable.

Example: Reversing the Conditions

When reviewing the manuscript of the first edition of this book, Joshua Kerievsky pointed out that we often do Replace Nested Conditional with Guard Clauses by reversing the conditional expressions. Even better, he gave me an example so I didn't have to further tax my imagination.

```
function adjustedCapital(anInstrument) {
  let result = 0;
  if (anInstrument.capital > 0) {
    if (anInstrument.interestRate > 0 && anInstrument.duration > 0) {
      result = (anInstrument.income / anInstrument.duration) * anInstrument.adjustmentFactor;
    }
  }
  return result;
}
```

Again, I make the replacements one at a time, but this time I reverse the condition as I put in the guard clause.

```
function adjustedCapital(anInstrument) {
  let result = 0;
  if (anInstrument.capital <= 0) return result;
  if (anInstrument.interestRate > 0 && anInstrument.duration > 0) {
    result = (anInstrument.income / anInstrument.duration) * anInstrument.adjustmentFactor;
  }
  return result;
}
```

The next conditional is a bit more complicated, so I do it in two steps. First, I simply add a `not`.

```
function adjustedCapital(anInstrument) {
  let result = 0;
  if (anInstrument.capital <= 0) return result;
  if (!(anInstrument.interestRate > 0 && anInstrument.duration > 0)) return result;
  result = (anInstrument.income / anInstrument.duration) * anInstrument.adjustmentFactor;
  return result;
}
```

Leaving `not`s in a conditional like that twists my mind around at a painful angle, so I simplify it:

```
function adjustedCapital(anInstrument) {
  let result = 0;
  if (anInstrument.capital <= 0) return result;
  if (anInstrument.interestRate <= 0 || anInstrument.duration <= 0) return result;
  result = (anInstrument.income / anInstrument.duration) * anInstrument.adjustmentFactor;
  return result;
}
```

Both of those lines have conditions with the same result, so I apply Consolidate Conditional Expression.

```
function adjustedCapital(anInstrument) {
  let result = 0;
  if (   anInstrument.capital      <= 0
      || anInstrument.interestRate <= 0
      || anInstrument.duration     <= 0) return result;
  result = (anInstrument.income / anInstrument.duration) * anInstrument.adjustmentFactor;
  return result;
}
```

The `result` variable is doing two things here. Its first setting to zero indicates what to return when the guard clause triggers; its second value is the final computation. I can get rid of it, which both eliminates its double usage and gets me a strawberry.

```
function adjustedCapital(anInstrument) {
  if (   anInstrument.capital      <= 0
      || anInstrument.interestRate <= 0
      || anInstrument.duration     <= 0) return 0;
  return (anInstrument.income / anInstrument.duration) * anInstrument.adjustmentFactor;
}
```


previous:
Consolidate Conditional Expression

next:
Replace Conditional with Polymorphism