Outro tipo de código digno de ser refatorado são códigos que contém muitos ifs. Ifs são difíceis de ler e tornam o código mais complexo do que deveria.

Veja, por exemplo, a classe `GeradorDeNotaFiscal`, que trabalhamos no primeiro capítulo. Ela tem um método privado, cheio de ifs:

```
    private NotaFiscal geraNf(Fatura fatura) {
        double valor = fatura.getValorMensal();
        double imposto = 0;
        if(valor < 200) {
            imposto = valor * 0.03;
        }
        else if(valor > 200 && valor <= 1000) {
            imposto = valor * 0.06;
        }
        else {
            imposto = valor * 0.07;
        }

        NotaFiscal nf = new NotaFiscal(valor, imposto);
        return nf;
    }
```

Veja que essas condições podem ficar complexas. Ifs encadeados, com condições cada vez mais complicadas. Uma refatoração possível para esse ponto é mover essa regra de negócio para o lugar certo; esse código faz muito mais sentido ficar dentro da própria classe `NotaFiscal`. Vamos então mover essa lógica de lugar:

```
public class NotaFiscal {

    private int id;
    private double valorBruto;

    public NotaFiscal(int id, double valorBruto) {
        this.id = id;
        this.valorBruto = valorBruto;
    }

    public NotaFiscal(double valorBruto) {
        this(0, valorBruto);
    }

    public double getImpostos() {

        double imposto = 0;
        if(valorBruto < 200) {
            imposto = valorBruto * 0.03;
        }
        else if(valorBruto > 200 && valorBruto <= 1000) {
            imposto = valorBruto * 0.06;
        }
        else {
            imposto = valorBruto * 0.07;
        }

        return imposto;

    }

    public double getValorLiquido() {
        return this.valorBruto - this.getImpostos();
    }
}
```

Já está melhor. Veja então que podemos mover comportamentos de um lado para outro. O objetivo disso é encapsular melhor as regras de negócio, e até esconder a "sujeira" desses ifs aninhados.

Outro if que é bem digno de uma refatoração é aquele if com uma condição complexa demais. Veja, por exemplo, o código abaixo, que representa uma `Matricula`:

```
public class Matricula {

    private boolean trial;
    private Calendar expiracao;

    public Matricula(boolean ehTrial, Calendar expiracao) {
        this.trial = ehTrial;
        this.expiracao = expiracao;
    }

    public boolean isTrial() {
        return trial;
    }

    public Calendar getExpiracao() {
        return expiracao;
    }
}
```

Uma matrícula é válida quando não é um "trial" e a data de expiração ainda não passou. Ou seja, se quisermos fazer alguma lógica cuja matrícula deve estar nesse estado, faríamos:

```
if(!matricula.isTrial() && matricula.getExpiracao().after(Calendar.getIns
  // faz algo aqui...
}
```

Veja só que essa condição é difícil de ser entendida de primeira. Isso acontece sempre quando temos um if com diversas condições (AND, OR, etc). Nesses casos, a melhor solução é refatorar, e esconder essa condição dentro de algum método, cujo nome dê uma semântica à condição:

```
public class Matricula {
    public boolean estaValida() {
        return !trial && expiracao.after(Calendar.getInstance());
    }
}

if(matricula.estaValida()) {
  // faz algo
}
```

Evite ao máximo ifs complicados espalhados pelo código. Além de ser difícil de ler, ainda furam o encapsulamento! Veja que o método `estaValida()` resolve bem esse problema. É fácil de saber o que ele faz, e a regra (os ifs) estão bem escondidos.

Uma outra maneira bem interessante de eliminar essas condições, é apelar para padrões de projeto. Padrões de projeto, como Strategy, State, Decorator, e etc, ajudam a resolver o problema desses diversos ifs em sequência. Estude-os. Nós temos um curso online focado neles.