Iremos começar nosso estudo prático de boas práticas de desenvolvimento a partir dessa seção. O conhecimento a ser adquirido pela proposta dessa unidade vai ser amadurecido com o aprendizado concomitante das subunidades Refatoração, Clean Code e Teste em algoritmos. E *Refatoração* começará a ser discutido com o algoritmo abaixo:

function permitirSaque(quantia) {

var saldo = 500.00;

const imposto = 0.010;

const chequeEspecial = 100.00

// permite sacar sempre que o saque for menor ou igual ao saldo + chequeEspecial E saque menor que a metade do saldo

if ((quantia <= (chequeEspecial + saldo)) && (quantia <= (saldo / 2))){

saldo = saldo - quantia

console.log('Saque da quantia de ' + quantia + ' autorizado! Novo saldo: R$ ' + saldo)

// permite sacar sempre que o saque for menor ou igual ao saldo + chequeEspecial E saque maior que a metade do saldo com imposto

} else if ((quantia <= (chequeEspecial + saldo)) && (quantia > (saldo / 2))) {

var novoSaldo = saldo - quantia

var saldoDescontadoOImposto = novoSaldo - saldo \* imposto

console.log('Saque da quantia de ' + quantia + ' autorizado! Novo saldo: R$ ' + saldoDescontadoOImposto)

}else if ((quantia > (chequeEspecial + saldo))) {

console.log('Saque não autorizado!')

}

}

O algoritmo acima se for invocado através de permitirSaque(300)exibirá a seguinte mensagem como saída:

Saque da quantia de 300 autorizado! Novo saldo: R$ 195

Certamente essa informação é bem sugestiva quando lida tanto por um usuário quanto por um programador, isso quer dizer que para entendê-la, não é necessário que o algoritmo seja compreendido. O usuário sairá feliz por ter seu saque autorizado, será? E o programador, estará feliz porque fez o usuário feliz? Bem, para entender essa nova unidade precisamos enxergar sob duas óticas diferentes, a do programador e a do usuário.

**A ótica do usuário**

Já falamos sobre o princípio da responsabilidade única inúmeras vezes assim como também já exploramos o uso de funções. Uma função bem desenvolvida por um programador cumprirá com o princípio da responsabilidade única, isso já sabemos, mas o quanto dessa premissa interessa ao usuário que irá interagir com o software que está disponível?

Não é objeto dessa disciplina discutirmos sobre projeto de software, mas desde o início do nosso aprendizado sobre algoritmos nós estamos levando em conta que um programador programa alguma coisa que é exigido por uma demanda, ou seja: Um requisito de software gera uma necessidade de programação.

Os requisitos de software são baseados em cenários do mundo real que representam a realidade de interação entre o usuário e o software. Por exemplo: Um jogador de futebol pode realizar inúmeras façanhas com uma bola de futebol, mas será que ele consegue fazer a mesma coisa com uma bolinha de gude? Por favor não procure no Youtube, você vai acabar encontrando e atrapalhando meu exemplo =)

Um cenário é reflexo da realidade e direciona o programador a desenvolver exatamente o que o usuário espera em cada interação com o software. Se o motorista aperta a buzina de um carro, ele espera um barulho, se pisa em um acelerador, espera que o carro acelere, se pisa no freio espera que o carro freie, se puxa o freio de mão, espera que o carro pare. Simples assim: Para cada cenário uma resposta.

**A ótica do programador**

Muito mais complexa que a ótica do usuário, a ótica do programador precisa cumprir algumas façanhas para garantir um usuário satisfeito, vamos nos ater a duas específicas. São elas:

* Entregar a funcionalidade que o usuário espera
* Garantir a manutenção e evolução das funcionalidades

O que o usuário espera

*\* Cenários de um saque em conta corrente sem as regras de negócio definidas pela apostila.*

Cenário 1) Usuário possui R$ 300,00 de saldo e solicita o saque de R$ 100,00. Espera que o saldo de R$ 200,00 permaneça disponível em sua conta

Cenário 2) Usuário possui R$ 300,00 de saldo e solicita o saque R$ 300,00. Espera que o saldo de R$ 0,00 seja o novo valor de saldo para sua conta

Cenário 3) Usuário possui R$ 300,00 de saldo e solicita o saque de R$ 350,00. Espera que fique devendo ao banco R$ 50,00

Cenário 4) Usuário com R$ 50,00 negativos na conta faz um depósito de R$ 100,00, espera que o novo saldo em conta seja de R$ 50,00

Enfim, poderíamos criar tantos outros cenários baseados em requisitos de software, mas julgo ser suficiente para o entendimento sobre o que são cenários e como eles guiam o desenvolvedor em sua árdua tarefa de programar exatamente o que o usuário espera.

Manutenção e evolução das funcionalidades

Esse quesito é inteiramente voltado para o programador. Quando um programador desenvolve algum algoritmo ele tem a responsabilidade de escrever códigos que sejam legíveis tanto para o próprio quanto para outro programador. Muitas das vezes o trabalho será em equipe, o que torna mais complexo o desenvolvimento de um software, mas não há como fugir dessa realidade. Na verdade, é muito melhor para um projeto que mais de um programador esteja envolvido, pois um programador consegue validar o que o outro fez e cria uma formalística de trabalho que só contribui para a agilidade do desenvolvimento do projeto e para a qualidade.

Para que um programador garanta que o software desenvolvido cumpre com os requisitos para uma boa manutenção e evolução das funcionalidades, os conceitos a seguir precisam ser bem empregados:

=> Refatoração:

*Alteração feita na estrutura interna do software para torná-lo mais fácil de ser entendido e menos custoso de ser modificado sem alterar seu comportamento observável - Martin Fowler*

var saldo = 500.00;

function permitirSaque(quantia) {

const chequeEspecial = 100;

if (quantia <= (chequeEspecial + saldo)){

console.log('Saque da quantia de ' + quantia + ' autorizado!') // Não sabe ainda o novo saldo

descontarSaqueDoSaldo(quantia)

}else{

console.log('Saque não autorizado!')

}

}

function descontarSaqueDoSaldo(quantia) {

var novoSaldo = saldo - quantia // O novo saldo deve ser descoberto aqui

var saldoFinal = descontarImpostoDoSaldo(novoSaldo, quantia)

console.log('Novo saldo: R$ ' + saldoFinal)

}

function descontarImpostoDoSaldo(quantia, saldo){

const imposto = 0.010;

if (quantia > (saldo / 2)){

var saldoDescontadoOImposto = novoSaldo - (saldo \* imposto)

return saldoDescontadoOImposto

}else{

return novoSaldo

}

}

Começamos nossa aula apresentando um algoritmo que ganhou o selo padrão *Go Horse.* O Código acima refatora o algoritmo inicial e nos apresenta uma nova forma de escrita para a mesma funcionalidade. Isso é a refatoração. O objetivo é mudar a forma de escrita sem mudar o comportamento, pois nesse momento o que nos importa é o entendimento da funcionalidade através da linguagem do programador e não na linguagem do usuário. Ou seja, um programador se comunica com ele e com outro programador através do código escrito, e para isso deve estar bem legível. Um programador se comunica com o usuário através da funcionalidade desenvolvida, que deve cumprir com o requisito de software solicitado.

=> Clean Code:

* Variável dentro do contexto

Uma variável que tenha um nome condizente com o contexto em que se está inserida. Observe como não há contexto nas variáveis criadas abaixo:

var valor1 = 500.00;

var valor2 = 0.010;

*valor1* e *valor2* não se referem a contexto algum. Um programador que leia essas duas variáveis precisará ler outras linhas de código para saberem o que elas representam. Dessa forma, é fundamental que os nomes das variáveis estejam de acordo com a área de domínio (contexto) em que estão inseridas.

var saldo = 500.00;

var imposto = 0.010;

Agora sim, *saldo* representa *valor1* e *imposto* representa valor2.

* Quantos menos Else If melhor

Vulgarmente conhecido como “*Else less”* (menos Else), essa abordagem é interessante, pois além de diminuir linhas de código, na maioria dos casos, faz com o que o programador programe de forma clara e fácil de entender o propósito de uma função. No código refatorado podemos observar a ausência da condicional alternativa *Else* implementando uma técnica simples: Negar a lógica de decisão e retornar como valor padrão o que o propósito do algoritmo exige que seja retornado*.* Um algoritmo denominado “Descontar imposto do saldo” tem o propósito de retornar um saldo descontado o imposto, logo, espera-se como padrão que seja retornado *saldoDescontatoOImposto* e tudo o que não for padrão seja retornado em lógicas condicionais específicas, nesse caso, *novoSaldo* é uma alternativa retornada quando não é preciso aplicar o imposto ao valor do saldo.

function descontarImpostoDoSaldo(quantia, saldo){

const imposto = 0.010;

if (quantia > (saldo / 2)){

var saldoDescontadoOImposto = novoSaldo - (saldo \* imposto)

return saldoDescontadoOImposto

}else{

return novoSaldo

}

}

var saldo = 500.00;

function descontarImpostoDoSaldo(novoSaldo, quantia){

const imposto = 0.010;

var novoSaldoDescontadoOImposto = novoSaldo - (saldo \* imposto)

if (!quantia > (saldo / 2)){

return novoSaldo

}

return novoSaldoDescontadoOImposto

}

Lembrando que uma estrutura *if … else* que conhecemos é uma estrutura de desvio de fluxo, ou seja, é natural que na ausência de uma estrutura de desvio de fluxo, o fluxo “de cima para baixo” ocorra sem ela, retornando algum resultado padrão. Para esses casos, quando uma função tem um comportamento padrão e a condicional vem para “fugir” do padrão, deixemos a condicional para a “fuga” enquanto a estrutura da função permite que seja retornado o que se espera.

* Código auto explicativo

Muito comum um programador colocar comentários em suas linhas de código para explicar para outro programador (pra ele mesmo, muitas vezes) o que determinado código faz. Isso não é uma boa prática, pois, como já estamos aprendendo aqui, um código por si só deve ser claro o suficiente para ser bem compreendido. Nomes de funções devem condizer com seu propósito, exemplo: function retirarImpostoDoSaldo(). Isso lembra muito com a idéia de escolhermos os nomes corretos para as variáveis.

// permite sacar sempre que o saque for menor ou igual ao saldo + chequeEspecial E saque menor que a metade do saldo

if ((quantia <= (chequeEspecial + saldo)) && (quantia <= (saldo / 2))){

saldo = saldo - quantia

console.log('Saque da quantia de ' + quantia + ' autorizado! Novo saldo: R$ ' + saldo)

O código acima já foi refatorado anteriormente, e no momento que se concluiu a refatoração, o entendimento do código foi suficiente e o comentário pode ser apagado.

Claro que nem todo comentário é completamente desnecessário. Quando alguma coisa que está transparente para o programador, como uma trigger de banco de roda quando determinada ação é executada, por exemplo.

// Trigger de banco *executarSaque* é executada na função

executarSaque(100.00)

* Teste em algoritmos

Quantos testes somos capazes de realizarmos no nosso dia a dia? Por exemplo, quando você toma banho em um dia frio, é comum a gente por a mão ou o pé na água antes de colocar todo o corpo para o caso de a água estar muito gelada. Ou, ao invés de tomar um bom gole de café, damos apenas uma bicada para saber se não está muito quente e aos poucos acostumarmos nossa boca a sentir a temperatura do líquido. Enfim, a gente testa bastante as coisas antes de determinadas ações, só não estamos muito acostumados a associar os movimentos que fazemos antes das ações ao nome “teste”.

Basicamente “teste” existe para que algo não dê errado. Seria uma forma de mitigar os riscos de alguma ação indesejada. Pois então, o assunto sobre testes em software é extremamente necessário que seja bem discutido nessa disciplina.

Imagine um software que acompanha os batimentos cardíacos de um paciente que não dá os resultados precisos a cada pulsação do coração. Ou então, imagine um software que meça a altura de um avião na hora da aterrissagem. Nossa, já deu pra perceber a seriedade que muitos softwares possuem por atuar em ações muito críticas para a vida humana. Por esse motivo, o primeiro passo em qualquer teste é saber o que esperar do funcionamento de determinada coisa. E esse funcionamento a gente consegue prever baseado em um requisito, ou seja, baseado naquilo que quem entende sobre o assunto espera como um correto funcionamento. Dessa forma aprenderemos aqui a testar o comportamento das coisas nos baseando no nosso entendimento sobre elas.

Desenvolvimento Orientado ao Comportamento (BDD - Behaviour Driven Development)

Um requisito de software é baseado em um comportamento que é definido pelo cliente. O comportamento desse requisito será testado seguindo o seguinte entendimento:

Dado xxxxxxxxxxxxxxxxxx Quando xxxxxxxxxxxxx Espero xxxxxxxxxxxxxxxxxx

Esse modelo de orientação será útil para induzir o que recebemos como requisito a uma escrita de teste que será desenvolvida como a seguir:

1. Entender o problema que motiva a criação do algoritmo

*aplicar desconto de 5% para compras acima de R$ 200,00*

1. Preencher o modelo apresentado com as entradas compatíveis com o problema

Dado *valorDaCompra = 210.00* Quando *finalizar a compra* Espero *valorDaCompra = 199.50*

1. Desenvolver uma função de teste vazia

function adicionarDescontoAoValorDaCompra(){

// sem código

}

1. Chamar a função com os parâmetros de entrada a serem utilizados como teste

adicionarDescontoAoValorDaCompra(200.00)

1. Acompanhar o retorno
2. Desenvolver o conteúdo da função para que a chamada da função não mais retorne o erro

function adicionarDescontoAoValorDaCompra(valorDaCompra){

if (valorDaCompra > 200.00){

return (valorDaCompra = valorDaCompra - (valorDaCompra \* 5)/100)

}else{

return valorDaCompra

}

}

1. Refatorar o algoritmo caso o teste tenha retornado exatamente o que se esperava

function adicionarDescontoAoValorDaCompra(valorDaCompra){

var desconto = (valorDaCompra \* 5)/100;

var valorFinalComDesconto = valorDaCompra - desconto

if (valorDaCompra <= 200){

return valorDaCompra

}

return valorFinalComDesconto

}