Code Principles (Princípios de código)

Pricípios de código servem para organizar e padronizar código desenvolvido. Fundamentalmente a organização e a padronização de código leva a várias melhorias no processo de desenvolvimento de software.

- Legibilidade
- Testabilitade

Porém todas essas melhorias serve apenas para um ajudar em particularmente um aspecto do desenvolvimento de softaware: **Adaptação a mudança**. Um sistema de software **precisa ser adaptado a mudança**. Requisitos do usuário, expecificações, hardware, escalabilidade, tudo isso mudam constantemente na vida de um software e é dever do programador garantir que cada mudança no sistema **não irá impactar em mudanças futuras ou diminuir o fator de entrega do time.**

Princípio do ETC (Easier to Change)

O principal princípio que devemos levar durante o desenvolvimento de qualquer funcionalidade em software é o **ETC (Easier to Change)**. A facilidade de modificar o sistema deve ser a principal preocupação na hora que a funcionalidade está sendo desenvolvida.

Porém garantir que um sistema de software seja fácil de ser modificado não é uma tarefa simples. Várias decisões durante o desenvolvimento podem levar a um resultado que impacta em futuras implementações e por assim deixam o sistema mais complexo e menos sucetível a mudanças. Por isso existem vários outros princípios que ajudam em pontos específicos e quando aplicados em conjunto levam a um sistema ETC.

Clean Code

Código limpo é um das principais formas de garantir que um sistema seja mais fácil de modificiar.

Garantir uma boa nomenclatura de métodos, variáveis, classes, funções ajudam na legibilidade do código. **Quanto mais legível** um código mais fácil de compreendê-lo e então **mais simples de ser modificado.**

Modularidade

> Modularidade é definida como, "o grau cujo componentes de um sistema podem ser separados e recombinados, com o objetivo de flexibilizar e variar seu uso."

Modularidade é um fator primordial para deixar um sistema mais simples para modificar. Um sistema modular reaproveita seus componentes e adiciona vários casos de uso para cada componente.

Além de permitir reutilizar componentes para múliplos usos componentes modulares garante a flexibilidade do sistema. É possível trocar componentes e comportamentos facilmente com componentes que apresentam a mesma interface. Em algumas linguagems podemos sendo possível fazê-lo em tempo de compilação, como caso de C# ou Java e seus tipos genéricos, ou em tempo de execução como linguagens interpretadas.

```
// Bad Modularity
function consoleLogCart(){
```

```
    var cart = Json.parse(localStorage.getItem("cart"));

    if(cart)
        throw "No items on cart"

    let str = ""
    for(const item of cart){
        str += `Product: ${item.name}\n`
        str += `Price: ${item.price}\n`

        const releaseDateYear = new Date(item.releaseDate).fullYear;
        str += `Release year: ${releaseDateYear}\n`
        str += "\n"
    }

    console.log(str)
}
```

O método `consoleLogCart()` não é nada modular. Todas as operações são dependentes umas das outras e não conseguimos reutilizar nenhuma parte desse código. Temos também vários níveis de abstrações implemetadas no mesmo escopo. Operações baixo nível como recuperar as informações do cart do localStorage ou criar uma data para buscar o ano de lançamento.

Outro problema de modularidade é em relação ao formato do cart. Qual quer que seja o sistema que armazena o cart no localStorage o faz com um formato próprio. Assim qualquer alteração nesse formato irá quebrar todos os outros lugares que utilizam do cart.

```
// Good Modularity
function consoleLogCart(cart){
    let str = ""
    for(const item of cart){
        str += `Product: ${item.product}\n`
        str += `Price: ${item.price}\n`
        str += `Release year: ${item.releaseYear}\n`
        str += "\n"
    }

    console.log(str)
}

function getCart(mapper){
    var cart = Json.parse(localStorage.getItem("cart"));

    if(cart)
        throw "No items on cart"

    return mapper(cart)
}

function mapCart(cart) {
    var mappedCart = {items: []}
```

```
    for(const item of cart){
        mappedCart.items.push({
            product: item.name,
            price: item.price,
            releaseYear: getFullYear(dateStr)
        })
    }

    return mappedCart
}

function getFullYear(dateStr){
    return `${new Date(dateStr).fullYear}`
}

// Chamada das funções
const cart = getCart(mapCart)
consoleLogCart(cart)
```

A segunda forma do código é muito mais eficiente em **definir fronteiras entre os componentes do código**. O método `consoleLogCart(cart)` agora é responsável exatamente por fazer o que se propõe, printar no console as informações do cart.

O método `getCart(mapper)` pode ser reutilizado para qualquer outra parte do sistema. E como ele recebe um mapper como parâmetro o formato que os dados estão armazenados não importa para quem chama o método, já que o formato utilizado será determinado pelo `mapper`.

Por fim ganhamos um método auxiliar para buscar o ano dada uma data em formato string. Para o sistema que chama o método `getFullYear()` não importa se a forma para a obtensão do ano de uma data. Essa implementação pode ser alterada a qualquer momento adicionando novos casos se necessário (Por exemplo, o DateConstructor não aceita uma data no formato ptBR como parâmetro).

Coesão

> Kent Beck define coesão como "Pull the things that are unrelated further apart, and put the things that are related closer together".

O programador deve se preocupar o tempo todo a criar um sistema coeso. Cada componente de um sistema deve ter um **significado** e deve ser implementado de acordo com esse significado. **O principal objetivo de um código é comunicar ideias para humanos.**

Coesão é mais dos que todas as outras ferramentas para gerenciar complexidade, **contextual**. Dependendo do contexto, coisas podem ou não ser relacionadas.

Uma visão mais ingênua de coesão é definir que tudo que se relaciona a uma coisa deve ser implementado de forma acoplado (implementado junto). Isso não é bem verdade já que devemos separar a implementação de forma a ter um código mais desacoplado.

Uma boa forma de aumentar a coesão e diminuir acoplamento é separar complexidade acidental da complexidade essencial.

O exemplo abaixo nos mostra um caso de um código muito acoplado e pouco coeso, já que vários conceitos estão se misturando e deixam o **código mais difícil de ser modificado**.

```
// Bad cohesion
function addToCart(item){
    var user = Json.parse(localStorage.getItem("user"))
    if(!user)
        throw "User must be logged in to add item to cart"

    var cart = Json.parse(localStorage.getItem("cart"));

    cart.push(item)

    localStorage.setItem("cart", Json.stringfy(cart))

    let totalCost = 0
    for(const cartItem in cart){
        totalCost += cartItem.price
    }

    return totalCost
}
```

O método addToCart apresenta várias responsabilidades o deixando com uma complexidade alta. Ele é responsável por:

- Validar que o item pode ser adicionado
- Carregar as informações prévias do cart
- Adicionar o item ao cart **(função descrita pelo nome da função)**
- Persistir as informações do cart
- Calcular o valor final do cart

Isso tudo para um método que chama apenas `addToCart`.

Uma primeira refatoração para esse método seria separar as complexidades acidentiais das complexidades essenciais.

```
// Improved from bad cohesion
function addToCart(item){
    var user = getUser()
    if(!user)
        throw "User must be logged in to add item to cart"

    var cart = getCart();

    cart.push(item)

    setCart(cart)

    let totalCost = 0
```

```
    for(const cartItem in cart){
        totalCost += cartItem.price
    }

    return totalCost
}

function getUser(){ return Json.parse(localStorage.getItem("user")) }

function getCart(){ return Json.parse(localStorage.getItem("cart")) }

function setCart(cart) { localStorage.setItem("cart", Json.stringfy(cart)) }
```

Esse código já está ordens de grandeza mais coeso que o anterior. Conseguimos ter uma leitura muito mais clara do que está acontecendo e muito mais simples de ser modificado. Porém ainda é um código pouco coeso já que vários conceitos estão acoplados no mesmo local. Ainda o método `addToCart` apresenta muita responsabilidade.

```
// Good cohesion
function addToCart(cart, item, next){
    // Validação de usuário deve ser feito em outro local

    cart.push(item)
    next(item) // Chama o próximo comando a ser executado
}
```

Esse último exemplo mostra um exemplo de um método com grande coesão. O método é responsável apenas pelo que ele é incumbido de fazer. Depois de fazer o que é necessário ele passa para o próximo comando na cadeia de execução (Chain of Responsability Pattern) para fazer a próxima função do sistema.

Separation of Concerns (Separação de responsabilidades)

Isolar responsabilidades aumenta a flexibilidade do sistema. Dentro de um sistema temos vários outros subsistemas que conversam entre si. Separar esses subsistemas facilita o teste do sistema já que é possível testar subsistemas independentemente, o que **aumenta a robustez** do sistema como um todo. Assim quando um subsistema é alterado o impacto dessa alteração é isolado o que deixa o sistema **mais simples de ser modificado.**

Desacoplamento

Coming Soon

Construção iterativa e incremental

Coming Soon

DRY

O princípio DRY significa *Don't repeat yourself*. Muitas pessoas entendem isso como evitar cópia de código em multiplos lugares, porém esse princípio nos diz algo muito mais poderoso que isso. O DRY tem a **intensão de garantir que um conceito dentro de um sistema esteja restrito apenas a um único ponto**. Ou seja, qualquer mudança no sistema deve ser feita em apenas um único lugar.

Exemplo

Considerando um método que imprime o saldo de um client em relação a sua conta bancária. Exemplo adaptado do livro The Pragmatic Programmer.

```
function printBalance(account){

    console.log(`Debits: ${account.debits.toFixed(2)}\n`)
    console.log(`Credits: ${account.debits.toFixed(2)}\n`)

    console.log("        -----------\n")

    if(account.fees < 0){
        console.log(`Fees: ${-account.fees.toFixed(2)}`)
    }
    else {
        console.log(`Fees: ${account.fees.toFixed(2)}`)
    }

    if(account.balance < 0){
        console.log(`Balance: ${-account.balance.toFixed(2)}`)
    }
    else {
        console.log(`Balance: ${account.balance.toFixed(2)}`)
    }

}
```

Esse método fere o princípio DRY pelo menos umas 3 vezes.

Primeiramente caso a formatação do valores seja alterada precisamos mudar em 4 lugares no nosso código. Para resolver essa inconsistência extrairmos esse comportamento para ele ser definido como um único ponto do nosso sistema.

```
function printBalance(account){

    console.log(`Debits: ${formatAmount(account.debits)}\n`)
    console.log(`Credits: ${formatAmount(account.credits)}\n`)

    console.log("        -----------\n")

    if(account.balance < 0){
        console.log(`Fees: -${formatAmount(account.fees)}`)
    }
```

```
    else {
        console.log(`Fees: ${formatAmount(account.fees)}`)
    }

    if(account.balance < 0){
        console.log(`Balance: -${formatAmount(account.debits)}`)
    }
    else {
        console.log(`Balance: ${formatAmount(account.debits)}`)
    }
}

function formatAmount(amount) {
    return amount.toFixed(2)
}
```

Com essa refatoração garantimos que a formatação do valor sempre irá respeitar a mesma regra e em caso de mudança, apenas um ponto do nosso sistema deve ser alterado, assim o princípio ETC é respeitado.

Porém ainda existem pontos que ferem o príncipio DRY. O segundo é o que diz respeito a formatação de valores negativos. Temos dois pontos do código que precisam ser alterados em caso de mudança.

```
function printBalance(account){
    console.log(`Debits: ${formatAmount(account.debits)}\n`)
    console.log(`Credits: ${formatAmount(account.credits)}\n`)
    console.log("        -----------\n")
    console.log(`Fees: ${formatAmount(account.fees)}\n`)
    console.log(`Balance: ${formatAmount(account.debits)}\n`)
}

function formatAmount(amount) {
    result = Math.abs(amount).toFixed(2)

    if(amount < 0){
        return "-" + result
    }

    return " " + result
}
```

Por último temos apenas a quantidade de espaços em branco de cada um dos seguimentos. Caso no futuro a exibição do saldo fosse alterada precisaria ser alterada em todos os pontos to código que são exibidos.

```
function printBalance(account){
    console.log(reportLine("Debits", account.debits))
    console.log(reportLine("Credits", account.credits))
    console.log(printLine("", "-----------"))
    console.log(reportLine("Fees", account.fees))
```

```
    console.log(reportLine("Balance", account.balance))
    console.log(reportLine("Debits", account.debits))
}

function reportLine(label, amount){
    return printLine(label + ":", formatAmount(amount))
}

function formatAmount(amount) {
    result = Math.abs(amount).toFixed(2)

    if(amount < 0){
        return "-" + result
    }

    return " " + result
}

function printLine(label, value){
    return `${label} ${value}\n`
}
```

Após toda essa refatoração o código repeita o princípio do DRY e é ordens de grandeza mais fácil de ser modificado no futuro.

KISS

Coming Soon

YAGNI

Coming Soon

SOLID

Os postulados SOLID foram apresentados por Robert C. Martin em um artigo publicado no ano 2000 cujo título, em tradução livre, é "Postulados de Projeto e Padrões de Projeto". O acrônimo SOLID propriamente dito teria sido cunhado mais tarde por Michael Feathers.

Single Responsability

Coming Soon

Open Close

Coming Soon

Liskov Substitution Principal

Coming Soon

Interface Segregation

Coming Soon

Dependency inversion

Coming Soon

Exemplos

Bibliografia

Pragmatic Programmer, The: Your journey to mastery, 20th Anniversary Edition (English Edition)

Modern Software Engineering

SOLID Principles summary