Objetivo:

I. Coesão;
II. Acoplamento:
   a. Fraco;
   b. Médio;
   c. Forte;
III. Exercícios

Observação: antes de começar, crie um projeto para reproduzir os exemplos:

1) Crie a pasta exemplo (ou qualquer outro nome sem caracteres especiais) no local de sua preferência do computador;

2) Abra a pasta no VS Code e acesse o terminal do VS Code;

3) No terminal, execute o comando **npm init -y** para criar o arquivo fundamental de um projeto Node (arquivo package.json);

4) No terminal, execute o comando **npm i -D ts-node typescript** para instalar os pacotes ts-node e typescript como dependências de desenvolvimento;

5) No terminal, execute o comando **tsc --init** para criar o arquivo de opções e configurações para o compilador TS (arquivo tsconfig.json);

6) Crie a **pasta src** na raiz do projeto;

7) Crie o arquivo **index.ts** na **pasta src**;

8) Adicione na propriedade scripts, do package.json, o comando para executar o arquivo index.ts. Ao final o arquivo package.json terá o seguinte conteúdo:

```
{
  "name": "TPII_Aula02",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "index": "ts-node ./src/index",
    "ex01": "ts-node ./src/ex01",
    "ex02": "ts-node ./src/ex02",
    "ex03": "ts-node ./src/ex03",
    "ex04": "ts-node ./src/ex04",
    "ex05": "ts-node ./src/ex05",
    "teste": "ts-node ./src/teste",
    "teste1": "ts-node ./src/teste1",
    "teste2": "ts-node ./src/teste2"
  },
  "keywords": [],
  "author": "Prof.Henrique Louro",
  "license": "ISC",
  "devDependencies": {
    "ts-node": "^10.9.2",
    "typescript": "^5.4.2"
```

```
    }
}
```

Observação: se o comando `ts-node` não funcionar, então use o programa `npx` para executar o comando `tsnode`:

```
"scripts": {
"start": "npx ts-node ./src/index"
        },
```

I. Coesão:

Segundo o dicionário "Oxford Languages" algo coeso é aquilo que é unido por coesão, ou que, no sentido figurado: apresenta harmonia; ajustado; concorde. Assim, coesão como parte das técnicas de Programação Orientada a Objetos, está, na verdade, ligada ao princípio da responsabilidade única, que foi introduzido por Robert C. Martin no início dos anos 2000 e diz que **uma classe deve ter apenas uma única responsabilidade** e **realizá-la de maneira satisfatória**, ou seja, uma **classe não deve assumir responsabilidades que não são suas**.

Portanto, um código coeso é um código onde as classes e/ou métodos tem uma única responsabilidade. Ou seja, um método com o nome **imprimeSoma()** deve **IMPRIMIR A SOMA** e não **CALCULAR A SOMA E IMPRIMI-LA**. Por quê?

a) Se quisermos mudar a maneira de como imprimir a soma: mudamos **apenas** o método de imprimir a soma.
b) Se quisermos mudar a maneira de calcular a soma: mudamos **apenas** o método que calcula a soma.

Porém, se um mesmo método calcula e imprime, podemos ter problemas ao mudá-lo, já que mexeremos em partes do código que não precisariam estar sob sua responsabilidade, caso o conceito de coesão, ou responsabilidade única, estivesse sendo implementado corretamente.

Assim, podemos inferir que a coesão é uma técnica importante para a manutenção e evolução dos softwares.

Resumindo, coesão se refere a quão estreitamente todas as rotinas em uma classe ou módulo estão relacionadas umas às outras. Uma classe coesa teria um único propósito ou responsabilidade.

Aqui está um exemplo de uma classe coesa em TypeScript:

```
class Calculadora {
    somar(a: number, b: number): number {
        return a + b;
    }

    subtrair(a: number, b: number): number {
        return a - b;
    }

    multiplicar(a: number, b: number): number {
        return a * b;
    }

    dividir(a: number, b: number): number {
        if (b === 0) {
            throw new Error("Divisão por zero não é permitida.");
```

```
        }
        return a / b;
    }
}
```

Neste exemplo, a classe Calculadora é coesa porque todas as suas funções (métodos) estão relacionadas à realização de operações matemáticas.

Para tornar a classe Calculadora **não coesa**, podemos adicionar responsabilidades que não estão relacionadas às operações matemáticas. Aqui está um exemplo:

```
class Calculadora {
    somar(a: number, b: number): number {
        return a + b;
    }

    subtrair(a: number, b: number): number {
        return a - b;
    }

    multiplicar(a: number, b: number): number {
        return a * b;
    }

    dividir(a: number, b: number): number {
        if (b === 0) {
            throw new Error("Divisão por zero não é permitida.");
        }
        return a / b;
    }

    // Adicionando responsabilidades não relacionadas à classe
Calculadora

    imprimirDataAtual(): void {
        const data = new Date();
        console.log(`Data atual: ${data.toLocaleDateString()}`);
    }

    saudacao(nome: string): void {
        console.log(`Olá, ${nome}!`);
    }
}
```

Neste exemplo, adicionamos os métodos 'imprimirDataAtual()' e 'saudacao()', que não têm relação com as responsabilidades de uma calculadora. Isso torna a **classe não coesa**, pois agora ela tem mais de uma responsabilidade extra. No entanto, é importante notar que a coesão é geralmente desejável em design de software, pois torna o código mais fácil de entender, manter e reutilizar. Portanto, este exemplo serve apenas para fins ilustrativos e não é uma prática recomendada.

II. Acoplamento

Acoplamento em POO significa o quanto uma classe depende de outra para funcionar. Quanto maior for esta dependência entre ambas, dizemos que estas classes estão fortemente acopladas.

Quando uma classe está fortemente acoplada a outras classes, dificulta-se o gerenciamento do sistema, pois quando precisamos efetuar uma alteração em uma das classes, temos que alterar o código em outras classes também.

Assim, Acoplamento se refere ao grau em que uma classe ou módulo depende de outras classes ou módulos.

a) **Acoplamento Fraco**: aqui o objetivo é ter o menor acoplamento possível.

Aqui está um exemplo de baixo (fraco) acoplamento em TypeScript:

```
class Motor {
    ligar(): void {
        console.log("Motor ligado.");
    }
}

class Carro {
    private motor: Motor;

    constructor(motor: Motor) {
        this.motor = motor;
    }

    ligarMotor(): void {
        this.motor.ligar();
    }
}
```

Neste exemplo, a classe Carro tem uma dependência de Motor, mas essa dependência é minimizada porque Carro não precisa saber os detalhes de como Motor funciona. Isso é conhecido como baixo ou fraco acoplamento. Além disso, se decidirmos mudar a implementação de Motor, não precisaremos alterar a classe Carro, desde que a interface de Motor permaneça a mesma.

b) **Acoplamento Médio**: O termo “acoplamento médio” não é comumente usado na literatura de design de software. Geralmente, falamos em termos de “alto acoplamento” e “baixo acoplamento”. No entanto, se quisermos considerar um “acoplamento médio”, poderíamos pensar em um cenário onde uma classe depende de outra classe, mas não de todos os detalhes da outra classe.

Aqui está um exemplo em TypeScript que pode ilustrar essa ideia:

```
class Motor {
    ligar(): void {
        console.log("Motor ligado.");
    }

    desligar(): void {
        console.log("Motor desligado.");
    }
}

class Carro {
    private motor: Motor;

    constructor(motor: Motor) {
```

```
        this.motor = motor;
    }

    ligarMotor(): void {
        this.motor.ligar();
    }
}
```

Neste exemplo, a classe Carro depende da classe Motor, mas apenas do método ligar(). Ela não depende do método desligar(). Portanto, poderíamos dizer que Carro está moderadamente acoplada a Motor: ela depende da Motor, mas não de todos os detalhes da Motor.

No entanto, é importante notar que o objetivo do design de software é geralmente minimizar o acoplamento tanto quanto possível. Mesmo o "acoplamento médio" pode tornar o código mais difícil de manter e modificar. Portanto, é melhor se esforçar para baixo acoplamento sempre que possível.

c) **Acoplamento Forte**: Isso ocorre quando uma classe ou módulo depende fortemente de outra classe ou módulo. Significa que uma mudança em uma classe pode afetar a outra classe. O acoplamento forte é geralmente indesejável porque torna o código mais difícil de modificar e manter.

Aqui está um exemplo de acoplamento forte em TypeScript:

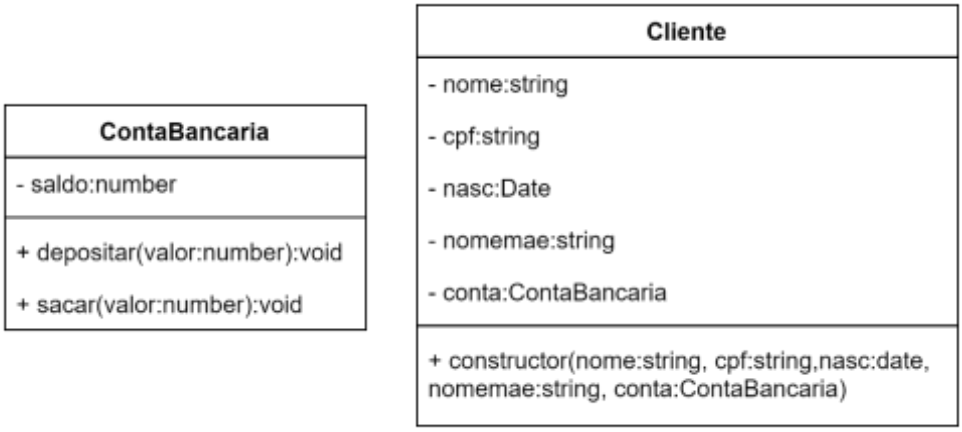
```
class Motor {
    ligar(): void {
        console.log("Motor ligado.");
    }

    desligar(): void {
         console.log("Motor desligado.");
    }

}

class Carro {
    private motor: Motor;

    constructor() {
        this.motor = new Motor();
    }

    ligarMotor(): void {
        this.motor.ligar();
    }

    desligarMotor(): void {
        this.motor.desligar();
    }

}
```

Neste exemplo, a classe Carro está fortemente acoplada à classe Motor porque está diretamente instanciando um novo Motor dentro do seu construtor, além de utilizar os métodos ligar e desligar. Isso significa que qualquer mudança na classe Motor pode afetar a classe Carro. Além disso, se quisermos usar um tipo diferente de Motor na Carro, teríamos que mudar o código da classe Carro. Isso torna o código

menos flexível e mais difícil de manter. É por isso que geralmente preferimos baixo acoplamento em design de software.

III. Exercícios

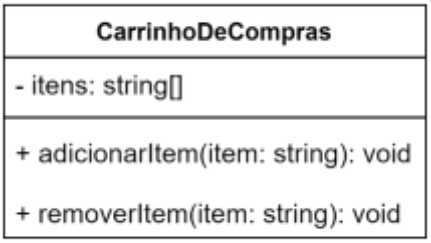
1. **Conta Bancária**: Crie uma classe chamada "ContaBancaria" que represente uma conta bancária com saldo e métodos para depósito e saque,
   Cliente: Crie uma classe chamada Cliente que implemente um objeto da Classe ContaBancaria.
   Siga os requisitos descritos nos diagramas de Classe UML, abaixo.



   Tarefas de Implementação:
   - Implemente os métodos "depositar" e "sacar";
   - Garanta que o saldo não fique negativo após um saque;
   - Instancie um objeto da classe ContaBancaria;
   - Instancie um objeto da classe Cliente com os seguintes dados:
     - nome: Seu nome
     - cpf: seu CPF
     - nasc: sua data de nascimento no formato "DD/MM/AAAA"
     - nomemae: primeiro nome da sua mãe
     - conta: objeto criado anteriormente.
   - Faça um depósito de R$ 100,00 via conta do cliente;
   - Faça um saque de R$ 50,00 via conta do cliente;
   - Tente fazer um saque de R$ 60,00 via conta do cliente.

Nesse exercício a classe Cliente implementa um objeto da classe ContaBancaria e outro da classe Date, causando uma forte dependência, estando assim fortemente acopladas.

2. **Carrinho de Compras**: Crie uma classe "CarrinhoDeCompras" que represente um carrinho de compras com itens e métodos para adicioná-los e removê-los.



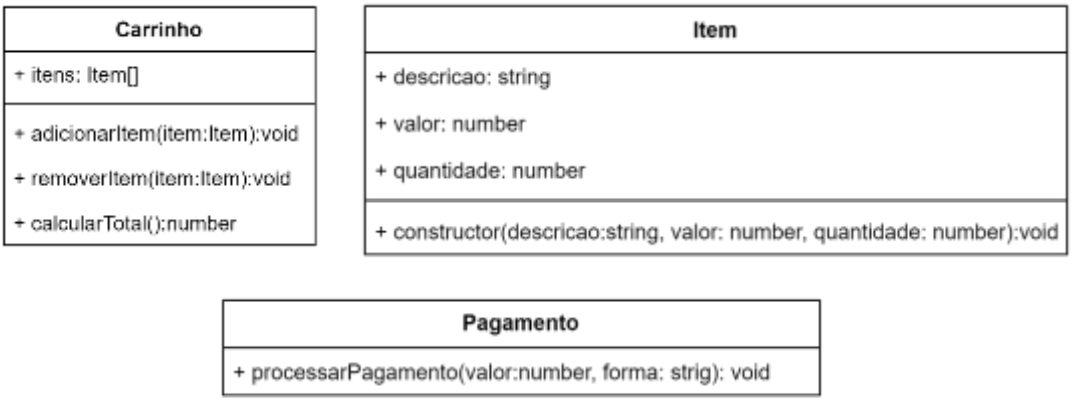
   Tarefas de Implementação:
   - Implemente os métodos "adicionarItem" e "removerItem";

- Mantenha a lista de itens atualizada;
- Crie um método imprimir():void, que liste no console todos os itens do carrinho;
- Use o código a seguir para testar sua implementação:

```
const carrinho = new CarrinhoDeCompras();
carrinho.adicionarItem("Camiseta");
carrinho.adicionarItem("Calça");
carrinho.adicionarItem("Meia");
carrinho.removerItem("Camiseta");
console.log(carrinho.imprimir());
```

Nesse exercício, a classe CarrinhoDeCompras têm responsabilidades bem definidas e não está fortemente acoplada a outras classes. Isso facilita a manutenção e reutilização do código.

3. **Carrinho e Pagamento**: Implemente as classes dos diagramas UML a seguir:




```
calcularTotal(): number {
        // Lógica para calcular o total dos itens no carrinho
        // Depende diretamente dos itens
        // Alto acoplamento entre CarrinhoDeCompras e Itens
}

class Pagamento {
    processarPagamento(total: number, forma: string): void {
        // Lógica para processar o pagamento
       // Imprimir "Pagamento de R$ 'total' em 'forma',
       processado com sucesso!"
        // Depende diretamente do total
        // Alto acoplamento entre Pagamento e CarrinhoDeCompras
    }
}
```

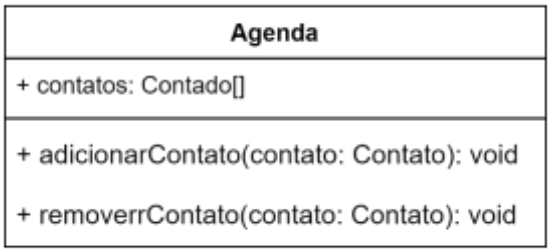
Execute as linhas a seguir para testar suas classes:

```
const carrinhoc = new Carrinho();
let item = new Item("Camiseta",50,2);
carrinhoc.adicionarItem(item);
item = new Item("Calça",130,1);
carrinhoc.adicionarItem(item);
item = new Item("Meia",20,3);
carrinhoc.adicionarItem(item);
const total = carrinhoc.calcularTotal();
```

```
console.log(total);
const pagamento = new Pagamento();
pagamento.processarPagamento(total,"dinheiro");
```

Nesse exercício, a classe Carrinho está altamente acoplada a classe Item e a classe Pagamento à classe Carrinho, pois depende do total da soma de cada Item. Assim, qualquer mudança na Item ou no Carrinho, impactarão todo o sistema.

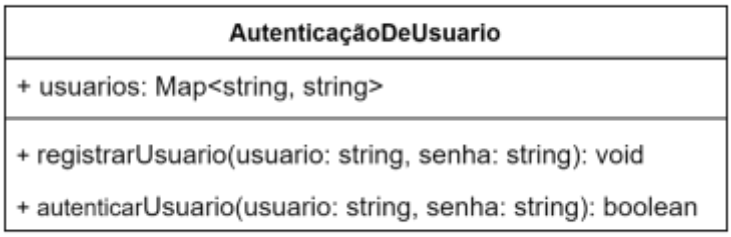
4. **Agenda de Contatos**: Crie uma classe "Agenda" que armazena informações de contatos (nome, telefone, e-mail):



Tarefas de Implementação:
- Implemente os métodos "adicionarContato" e "removerContato".
- Crie uma classe "Contato" para representar cada contato e poder inseri-lo na agenda.
- Crie linhas de código para testar as classes.

Nesse exercício as classes Agenda possui um alto acoplamento com a classe Contato, pois depende dela para registrar sua lista de contatos.

5. **Autenticação de Usuário**: Crie um sistema de autenticação de usuário com métodos para autenticar, registrar e gerenciar usuários:



Execute as linhas a seguir para testar suas classes:

```
const autenticacao = new AutenticacaoDeUsuario();
autenticacao.registrarUsuario("alice", "senha123");
autenticacao.registrarUsuario("bob", "outrasenha");

const usuarioAutenticado = autenticacao.autenticarUsuario("alice", "senha123");

if(usuarioAutenticado){
  console.log("Usuário autenticado com sucesso!");
} else {
  console.log("Falha na autenticação do Usuário!");
}
```

Nesse exercício, a classe tem responsabilidades bem definidas e não esta fortemente acoplada a outras classes. Isso facilita a manutenção e reutilização do código.

Referências:

https://www.devmedia.com.br/entendendo-coesao-e-acoplamento/18538
https://www.ateomomento.com.br/acoplamento-e-coesao/
https://devbty.com.br/blog/coesao-e-acoplamento-software
https://learn.microsoft.com/pt-br/visualstudio/code-quality/code-metrics-class-coupling?view=vs-2022