Padrões de projeto e micro serviços

Leonardo Bertechini Stábile 11/12/2021 Tecsoil Automação e Sistema S.A.

Sumário:

Todo o código e arquivos necessários em: https://github.com/solinftec/treinamento_microservicos

1. Introdução

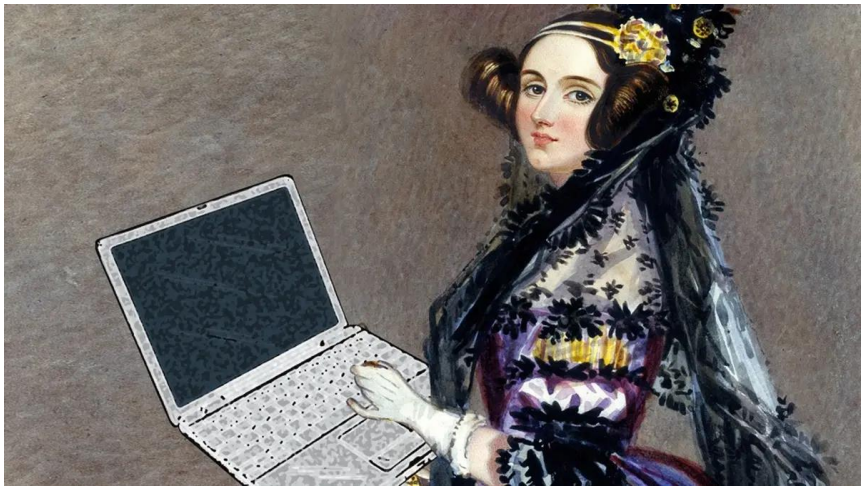
Bem-vindo ao curso de padrões de projeto, programar e resolver problemas realmente exige muito da nossa criatividade. Quanto mais complexo e difícil o problema, mais "criativo" você vai querer ser para resolve-lo, além disso, no ambiente corporativo estamos sempre buscando por produtividade, queremos sempre entregar nossas tarefas rapidamente e partir para a próxima para mostrar que somos produtivos.

Ser produtivo e criativo é muito bom, entretanto tem seu lado ruim, quando somente saímos a programar desenfreadamente com somente o objetivo de resolver o problema, acabamos criando um problema ainda maior no futuro.

Toda peça de software está fadada a um ciclo de vida, o programa que você escreveu hoje, que funciona e roda lindamente, amanhã já não vai mais atender as demandas da empresa, atingirá seu limite de escalonamento e provavelmente vai precisar cobrir mais casos de uso do que inicialmente foi projetado para atender.

Todo software sofre mudanças, mais cedo ou mais tarde.

Isso significa que mais do que resolver o problema e entregar a tarefa, devemos pensar nas inúmeras horas de manutenção que nós mesmos e outros colegas programadores irão empregar no software que estamos entregando.



Ada Lovelace (1815 - 1852), primeira mulher progamadora

Você não é o primeiro a mergulhar no mundo da programação, muitos outros já enfrentaram e resolveram muitos dos problemas que você vai ter pela frente **e isso é bom**. Utilizando padrões de projeto em nossos códigos fica mais fácil resolver problemas que outros já resolveram para nós no passado, isso tambem facilita que outros entendam nosso código e trabalhem nele depois de nós.

2. Decorador de objetos (Decorator pattern)

Quando estamos programando orientado a objetos é importante que tenhamos em mente o objetivo de criar objetos “fechados para alteração” e “abertos para extensão”. Não se preocupe, vamos entender mais sobre isso nas próximas páginas. Imagine que você construiu uma casa se, no futuro, você quiser construir um novo cômodo não deve ser necessário refazer a fundação do imóvel ou destruir cômodos existentes para construir um novo. A casa construída deve ser fechada para alteração e aberta para extensão, ou seja, deve ser fácil continuar construindo sobre o que já foi consolidado.

Para nosso exemplo de código, vamos assumir uma loja de carros, o primeiro objeto em que pensamos é o próprio carro. Vamos definir o conceito e as propriedades que um carro deve ter em uma interface. A interface é como se fosse a planta de um objeto concreto em java. Ela dita quais método a classe que a implementa deve possuir e pode ter várias implementações concretas.

```
public interface ICar {
    String getMake();
    void setMake(String make);
    String getModel();
    void setModel(String model);
    double getEngineSize();
    void setEngineSize(double size);
    String getTransmission();
    void setTransmission(String transmission);
    double getPrice();
    void setPrice(double price);
    boolean isAllTerrain();
    void setAllTerrain(boolean allTerrain);
```

```
    String print();
}
```

Também podemos chamar a interface ICar de **abstração** de um carro. Mas o que é uma abstração? Todo tempo que estamos programando estamos trazendo algo do mundo real, em que vivemos (também chamado de mundo concreto) para o mundo abstrato. O mundo abstrato é o mundo das ideias, trazer algo do mundo real para o mundo abstrato é isolá-lo e quebra-lo em pensamentos que podem ser expressos através de ideias. Quando eu lhe pergunto: o que é um carro? Antes de me responder, você vai criar na sua mente, mesmo que rapidamente, a imagem de um carro: que tem duas/quatro portas, quatro rodas, um motor, um porta malas e assim por diante, isso é uma abstração.

Aproveitando do conceito de abstração, vamos pegar todas essas características genéricas de um carro, e implantá-las em uma classe java, que servirá de base para o resto de nosso sistema.

```
public abstract class Car implements ICar {
    private static double TAX_PERCENTAGE = 0.01;
    private static final int PRINT_COLUMNS = 45;
    private static final int PRINT_PARAGRAPH_SIZE = 4;
    private static final DecimalFormat priceFormatter = new
DecimalFormat("0.00");

    protected String make;
    protected String model;
    protected double engineSize;
    protected String transmission;
    protected double price;
    protected boolean allTerrain;

    //getters and setters ommited.

    public String print() {
        var text = """
=============================================
  Congratulations on your new car purchase!
============================================="""" +
System.getProperty("line.separator");
        text += printDetailLine("Make", make);
        text += printDetailLine("Model", model);
        text += printDetailLine("Engine", "%.1f".formatted(engineSize) + "
liter(s)");
        text += printDetailLine("Transmission", transmission);
        if (allTerrain) {
            text += " AWD (All Wheel Drive) " +
System.getProperty("line.separator");
```

```
        }
        text += printDetailLine("Price", priceFormatter.format(price));
        text += printDetailLine("Taxes", priceFormatter.format(100 *
(getTaxPercentage()-1)) + "%");
        var final_price = priceFormatter.format(price * getTaxPercentage());
        text += System.getProperty("line.separator") + printDetailLine("Final
Price", final_price);
        for (int x = 0; x < PRINT_COLUMNS; x++) {
            text += "=";
        }
        return text;
    }

    protected String printDetailLine(String key, String value) {
        var line = "";
        for (int x = 0; x < PRINT_PARAGRAPH_SIZE; x++) {
            line += " ";
        }
        line += key + ": ";
        for (int x = 0, count = (PRINT_COLUMNS - line.length() -
value.length()); x < count; x++) {
            line += ".";
        }
        line += value;
        return line + System.getProperty("line.separator");
    }

    protected double getTaxPercentage(){
        return 1 + TAX_PERCENTAGE;
    }
}
```

Perceba que a definição da classe Car é precedida da palavra **abstract**, em java classes abstratas não podem ser instanciadas com a palavra **new**, elas apenas podem ser herdadas por outras classes abstratas ou implementações concretas. Isso faz sentido aqui pois, a classe Car é uma definição muito genérica de um carro, e os clientes quando vão a uma loja não estão procurando qualquer carro, as pessoas geralmente buscam por hatchbacks, sedans, coupes, SUVs, pickups e etc. Agora é hora das implementações concretas onde vamos criar **expecializações** da classe Car.

```
public class Hatchback extends Car {
    private static double TAX_PERCENTAGE = 0.04;

    @Override
    protected double getTaxPercentage() {
```

```
        return super.getTaxPercentage() + TAX_PERCENTAGE;
    }
}
```

Pronto, aproveitamos toda a implementação fechada, na classe Car e apenas a estendemos em uma especialização chamada de Hatchback. E o que muda? Em nosso sistema, todos os carros vendidos tem uma carga tributária. A taxa padrão para todo tipo de carro é de 1% (dividio por 100 é 0.01 que é o valor que consta na variável `TAX_PERCENTAGE` presente na classe Car), entretanto todos os hatchbacks tem um percentual adicional de impostos de 4%, por isso nosso método `getTaxPercentage` é sobrescrito na classe filha onde somamos a taxa adicional de impostos.

Vamos testar nossa implementação?

```
public class TemplateMethodApp
{
    public static void main( String[] args )
    {
        var golf = new SportsHatchback();
        golf.setMake("Volkswagen");
        golf.setModel("Golf");
        golf.setEngineSize(2.0);
        golf.setPrice(120000);
        golf.setTransmission("6 speed manual");
        golf.setAllTerrain(false);
        System.out.println(golf.print());
    }
}
```

O resultado de golf.print() é":

```
 ================================================
  Congratulations on your new car purchase!
================================================
    Make: ........................Volkswagen
    Model: ..............................Golf
    Engine: .....................2.0 liter(s)
    Transmission: ..............6 speed manual
    Price: ........................120000.00
    Taxes: ..............................5.00%

    Final Price: ...................126000.00
================================================
```

Decorar objetos é a arte de abstraí-los de forma que tenhamos uma classe que represente o conceito mais genérico possível sobre objeto discutido e que permita gerar especializações a partir desse conceito genérico. Dessa maneira estamos estendendo o conceito de carro no conceito de hatchback, aproveitando a fundação da casa que já existia e adicionando apenas um novo cômodo, mudando apenas o necessário, que no nosso caso, foi a taxa de impostos diferenciada para esse tipo de carro. O padrão decorator é flexível e permite inúmeras especializações, assim como podemos criar agora Caminhonetes,

Sedans e outros também podemos nos espcializar ainda mais no conceito de Hatchbacks Criando um Hatchback Esportivo:

```
public class SportsHatchback extends Hatchback  {
    private static double TAX_PERCENTAGE = 0.1;

    @Override
    protected double getTaxPercentage() {
        return super.getTaxPercentage() + TAX_PERCENTAGE;
    }
}
```

Perceba que um hatchback esportivo tem uma taxação adicional de 10%, que somada a taxação de um hatchback normal (4%) e à de qualquer carro (1%) nos deixa com uma alíquota total de 15%. Um exemplo de saída do método print dessa classe seria:

```
================================================
  Congratulations on your new car purchase!
================================================
    Make: ........................Volkswagen
    Model: .........................Golf GTI
    Engine: ......................2.0 liter(s)
    Transmission: ......7 speed automatic DSG
    Price: .........................250000.00
    Taxes: ............................15.00%

    Final Price: ..................287500.00
================================================
```

Um pouco mais sobre classes abstratas.

Você acha que já viu tudo que havia para saber sobre classes abstratas? Achou errado!

Pois bem, imagine que todo carro tem opcionais de fábrica e eles tem que sair no recibo de compra. É impossível saber esses opcionais enquanto ainda estamos na classe Car, eles não são genéricos para todos os carros, fazem parte das classes especialistas. Isso quer dizer que teríamos que sobrescrever o método print da classe pai, copiar o código de impressão para carros genéricos, e apenas alterar o necessário para imprimir também os opcionais?

Isso sem dúvida quebraria o conceito de aberto e fechado e nos esforçamos tanto para seguir até agora, entretanto os métodos abstratos, da classe Car podem nos ajudar com isso. Métodos abstratos fazem parte de classes abstratas, e não possuem definição de seu corpo, apenas de seu cabeçalho, assim como em interfaces e são de implementação obrigatória para as classes filhas.

Vamos criar um novo método `printDetails()` e chamá-lo no método de impressão antes de imprimir o preço final do carro:

```
protected abstract String printDetails();

public String print() {
        var text = """
==========================================
  Congratulations on your new car purchase!
==========================================""" +
System.getProperty("line.separator");
        text += printDetailLine("Make", make);
        text += printDetailLine("Model", model);
        text += printDetailLine("Engine", "%.1f".formatted(engineSize) + "
liter(s)");
        text += printDetailLine("Transmission", transmission);
        if (allTerrain) {
            text += " AWD (All Wheel Drive) " +
System.getProperty("line.separator");
        }
        text += printDetails();
        text += printDetailLine("Price", priceFormatter.format(price));
        text += printDetailLine("Taxes", priceFormatter.format(100 *
(getTaxPercentage()-1)) + "%");
        var final_price = priceFormatter.format(price * getTaxPercentage());
        text += System.getProperty("line.separator") + printDetailLine("Final
Price", final_price);
        for (int x = 0; x < PRINT_COLUMNS; x++) {
            text += "=";
        }
        return text;
    }
```

Agora nosso hatchback esportivo é obrigado a implementar esse método:

```
@Override
    protected String printDetails() {
        return printDetailLine("Optional", "Recaro seats");
    }
```

E nosso output passa a ser:

```
==========================================
  Congratulations on your new car purchase!
==========================================
    Make: ........................Volkswagen
    Model: .........................Golf GTI
    Engine: ....................2.0 liter(s)
    Transmission: ......7 speed automatic DSG
    Optional: ..................Recaro seats
    Price: .......................250000.00
    Taxes: ............................15.00%

    Final Price: ..................287500.00
==========================================
```

Conseguimos alterar o comportamento do código na classe pai, sem copiá-lo nem o sobrescrever, apenas deixando-o aberto para expansão.

3. Padrão construtor (Builder pattern)

Nossa impressora de recibo de compra para carros está pronta e está funcionando obedecendo aos requisitos do cliente, entretanto, você percebeu como o código de impressão é confuso, como ele é difícil de entender a princípio? Ele foi codificado de maneira procedural, ou seja, um grande bloco de código sequencial, e muitas partes desse código poderiam ser reaproveitadas para imprimir outros tipos de recibo dentro da loja, como por exemplo recibos de peças e serviços.

Para melhorar a compreensão desse código e facilitar manutenções futuras bem como a expansão do produto podemos utilizar o padrão builder. O padrão builder é indicado quando se quer construir objetos muito complexos que dependem de muitas variáveis. Neste tipo de padrão você começa com um objeto builder que contém várias funções que vão contruindo o objeto final em fazes, esses métodos retornam a própria instância do builder, o que permite que suas chamadas possam ser encadeadas.

```
public class CarReceiptBuilder {
    private static final String LINE_BREAK =
System.getProperty("line.separator");
    private static final DateTimeFormatter dateFormatter =
DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss.SSS");

    private final StringBuilder receipt;
    private int printColumns = 50;
    private int printParagraphSize = 3;

    public CarReceiptBuilder(StringBuilder receipt) {
        this.receipt = receipt;
    }

    public CarReceiptBuilder() {
        this(new StringBuilder());
    }

    public CarReceiptBuilder setPrintColumns(int printColumns) {
        this.printColumns = printColumns;
        return this;
    }

    public CarReceiptBuilder setPrintParagraphSize(int printParagraphSize) {
        this.printParagraphSize = printParagraphSize;
        return this;
    }

    public CarReceiptBuilder appendHeader() {
        return this.appendSeparatorLine()
```

```
            .appendParagraphLine("Congratulations on your new car purchase!")
            .appendSeparatorLine();
    }

    public CarReceiptBuilder appendDateLine() {
        return this.appendDetailLine("Date",
dateFormatter.format(LocalDateTime.now()), ' ');
    }

    public CarReceiptBuilder appendSeparatorLine() {
        receipt.append(StringUtils.repeat("=",
printColumns)).append(LINE_BREAK);
        return this;
    }

    public CarReceiptBuilder appendLineBreak() {
        receipt.append(LINE_BREAK);
        return this;
    }

    public CarReceiptBuilder appendDetailLine(String key, String value) {
        return appendDetailLine(key, value, '.');
    }

    public CarReceiptBuilder appendDetailLine(String key, String value, char
fillerChar) {
        var line = "";
        line += key + ":";
        for (int x = 0, count = (printColumns - line.length() -
value.length()); x < count; x++) {
            line += fillerChar;
        }
        line += value;
        receipt.append(line).append(LINE_BREAK);
        return this;
    }

    public CarReceiptBuilder appendParagraphLine(String paragraphText) {
        var line = "";
        //Adiciona espaços em branco representando o início do parágrafo
        paragraphText = StringUtils.repeat(" ", printParagraphSize) +
paragraphText;
        var printedCount = 0;
```

```
        var totalCharsToPrint = paragraphText.length();
        while(printedCount < totalCharsToPrint){
            line += paragraphText.charAt(printedCount);
            /*Se a linha atingir o tamanho máximo, adiciona a
            linha existente ao recibo e reseta a variabel "line"*/
            if (line.length() == printColumns){
                receipt.append(line).append(LINE_BREAK);
                line = "";
            }
            printedCount++;
        }
        if (line.length() > 0){
            receipt.append(line).append(LINE_BREAK);
        }
        return this;
    }

    public CarReceiptBuilder appendFooter() {
        return this.appendSeparatorLine();
    }

    public String build() {
        return receipt.toString();
    }
}
```

Nossa classe builder, apesar de possuir muitos métodos, é bem simples e cada um desses métodos tem um objetvo muito específico o que torna o código bastante reaproveitável. Agora contamos com o novo métdodo appendParagraphLine que nos permite escrever textos longos, quebrando automaticamente de linha quando o texto atingir o limite de colunas especificadas para impressão. Veja como fica a chamada do método print na classe Car utilizando o padrão builder:

```
public String print() {
        var builder = new CarReceiptBuilder()
            .appendHeader()
            .appendDateLine()
            .appendDetailLine("Make", make)
            .appendDetailLine("Model", model)
            .appendDetailLine("Engine", "%.1f".formatted(engineSize) + "
liter(s)")
            .appendDetailLine("Transmission", transmission);

        if (allTerrain) {
            builder.appendParagraphLine("Your vehicle is equipped with AWD (All
```

```
Wheel Drive capability) "
                + "please be aware that in order to flat tow your vehicle you
have to disengage the 4x4 "
                + "features first or else it will cause permanent damage to the
car transmission.");
        }

        return builder
            .appendDetailLine("Price", priceFormatter.format(price))
            .appendDetailLine("Taxes", priceFormatter.format(100 *
(getTaxPercentage() - 1)) + "%")
            .appendLineBreak()
            .appendDetailLine("Final Price", priceFormatter.format(price *
getTaxPercentage()))
            .appendFooter()
            .build();
    }
```

E o resultado final da impressão:

```
================================================
   Congratulations on your new car purchase!
================================================
Date:                      2021-12-28 15:11:08.640
Make:.................................Volkswagen
Model:................................Golf GTI
Engine:...............................2.0 liter(s)
Transmission:.................7 speed automatic DSG
   Your vehicle is equipped with AWD (All wheel Dr
ive capability) please be aware that in order to f
lat tow your vehicle you have to disengage the 4x4
features first or else it will cause permanent da
mage to the car transmission.
Price:................................250000.00
Taxes:................................15.00%

Final Price:..........................287500.00
================================================
```

4. Fábrica de objetos (Factory pattern)

Uma loja de automóveis não vende somente hatchbacks não é mesmo? Ao construir nossa aplicação precisamos pensar que um dia ela vai crescer e se tornar mais complexa, isso se chama escalabilidade. A escalabilidade é uma qualidade que permite que o software seja flexível para expandir e é algo que atinge vários aspectos do produto como recursos de hardware que ocupa, custos de deploy na nuvem, trafego de rede e etc, entretanto neste capítulo vamos abordar o código. O que acontece quando nossa loja passar a vender 20, 50 ou mais de 100 modelos de carros diferentes, sem contar peças e serviços? Quando temos uma variedade muito grande de classes filhas é comum que se use o padrão

pattern para gerenciar a criação dessas classes juntamente com qualquer lógica de negócio que possa existir nesse momento, dessa maneira, essa lógica não fica espalhada entre diversas classes do nosso código e facilita a manutenção.

Vamos partir do princípio que nossa loja oferece agora 5 tipos de carro para o consumidor:

```
Choose a car, press 0 to end purchase:
1 - Hyunday Azera
2 - Toyota Hilux
3 - Mitsubishi Pajero Sport
4 - Chevrolet Onix
5 - Lincoln Town Car
```

E que o cliente possa escolher mais de uma em sua compra. Para gerenciar a criação desses carros, selecionando corretamente todas suas especificações, vamos criar a classe CarSimpleFactory exemplificada pelo modelo abaixo:

```
public class CarSimpleFactory {
    public ICar makeCar(int carType) throws InvalidCarException {
        ICar car = null;
        switch(carType){
            case 1:
                car = new Sedan();
                car.setEngineSize(3.0);
                car.setMake("Hyunday");
                car.setModel("Azera");
                car.setPrice(200000);
                car.setTransmission("8 speed automatic");
                break;
            case 2:
                car = new Pickup();
                car.setEngineSize(2.8);
                car.setMake("Toyota");
                car.setModel("Hilux");
                car.setPrice(250000);
                car.setTransmission("6 speed manual");
                break;
            case 3:
                car = new SUV();
                car.setEngineSize(2.4);
                car.setMake("Mitsubishi");
                car.setModel("Pajero Sport");
                car.setPrice(350000);
                car.setTransmission("8 speed automatic");
                break;
            case 4:
                car = new Hatchback();
```

```
                car.setEngineSize(1.0);
                car.setMake("Chevrolet");
                car.setModel("Onix");
                car.setPrice(90000);
                car.setTransmission("6 speed manual");
                break;
            case 5:
                car = new Sedan();
                car.setEngineSize(4.6);
                car.setMake("Lincoln");
                car.setModel("Town car");
                car.setPrice(500000);
                car.setTransmission("7 speed automatic");
                break;
            default:
                throw new InvalidCarException();
        }
        return car;
    }
}
```

Podemos utilizar uma instância da classe CarSimpleFactory para auxiliar nosso loop de seleção de carros clase Main:

```
public static void main( String[] args )
    {
        int choice = 0;
        try (var inputScanner = new Scanner(System.in)) {
            var carArray = new ArrayList<ICar>();
            var carFactory = new CarSimpleFactory();
            while(true) {
                System.out.println("""
                Choose a car, press 0 to end purchase:
                1 - Hyunday Azera
                2 - Toyota Hilux
                3 - Mitsubishi Pajero Sport
                4 - Chevrolet Onix
                5 - Lincoln Town Car""");
                choice = inputScanner.nextInt();
                if(choice != 0) {
                    try {
                        carArray.add(carFactory.makeCar(choice));
                    } catch (InvalidCarException e) {
                        System.out.println("You have chosen a wrong type of
```

```
car. Try again");
                    }
                } else {
                    break;
                }
            }
            carArray.forEach(t -> System.out.println(t.print()));
        }
    }
```

5. Emissores e consumidores (*Observer pattern*)

Hoje considerado um padrão depreciado por muitos, foi o precursor dos padrões reativos de hoje: Data binding e RX java são exemplos de padrões onde a programação consiste em tratar eventos de maneira não procedural. Nesse padrão um objeto A, que necessita realizar uma tarefa toda vez que um determinado valor for alterado no objeto B, implementa uma interface que lhe permite receber notificações quando esses valores mudarem no objeto B. Dentro dessa dinâmica o objeto A é chamado de Observer e o B de Observable.

Agora, trazendo esses princípios para nosso aplicativo de vendas de carros, queremos que, enquanto o cliente estiver fazendo um pedido, adicionando e removendo carros da sua compra, que o valor total da compra seja atualizado e mostrado na tela. Para isso vamos transferir a lógica da seleção de carros para uma classe chamada CarSale que permite que o cliente compre mais de um carro. Dentro dessa classe também existe uma interface chamada ICarSaleObserver que deve ser implementada pelos objetos que queiram receber notificações do objeto CarSale.

```
public class CarSale {
    private List<ICar> carArray = null;
    private Set<ICarSaleObserver> observers = new HashSet<>();

    public void startSale(){
        carArray = new ArrayList<>();
        int choice = 0;
        try (var inputScanner = new Scanner(System.in)) {
            var carFactory = new CarSimpleFactory();
            System.out.println("""
                Choose a car, press
                    0 to finish sale
                    9 to remove last car:
                1 - Hyunday Azera
                2 - Toyota Hilux
                3 - Mitsubishi Pajero Sport
                4 - Chevrolet Onix
                5 - Lincoln Town Car""");
```

```
            while (true) {
                choice = inputScanner.nextInt();
                if (choice == 9){
                    if(carArray.size() > 0){
                        notifyCarRemoved(carArray.remove(carArray.size() - 1));
                    }
                } else if (choice == 0){
                    break;
                } else {
                    try {
                        var car = carFactory.makeCar(choice);
                        carArray.add(car);
                        notifyCarAdded(car);
                    } catch (InvalidCarException e) {
                        System.out.println("You have chosen a wrong type of
car. Try again");
                    }
                }
            }
        }
    }

    public void addObserver(ICarSaleObserver observer){
        observers.add(observer);
    }

    private void notifyCarAdded(ICar itemAdded) {
        observers.forEach(t -> t.onAddItem(itemAdded));
    }

    private void notifyCarRemoved(ICar itemRemoved) {
        observers.forEach(t -> t.onItemRemoved(itemRemoved));
    }

    public Double getSaleTotal(){
        return carArray.stream().map(ICar::getPriceWithTaxes).reduce(0D,
Double::sum);
    }

    public String getSaleReceipt(){
        return carArray.stream().map(t ->
t.print()).collect(Collectors.joining(System.getProperty("line.separator")));
    }
```

```
    public static interface ICarSaleObserver {
        void onAddItem(ICar itemAdded);
        void onItemRemoved(ICar itemRemoved);
    }
}
```

Para receber as notificações na classe main vamos utilizar uma funcionalidade do java que nos permite implementar uma interface sem precisar criar uma nova classe, esse método se chama **implementação anônima**. Nesses casos podemos instanciar uma interface com o palavra-chave **new** e temos que implementar o corpo dos métodos dessa interface no momento da criação dessa nova instância.

```
public static void main( String[] args )
    {
        var carSale = new CarSale();
        carSale.addObserver(new ICarSaleObserver() {
            @Override
            public void onAddItem(ICar itemAdded) {
                System.out.println(
                    String.format("%s %s added. Total sale value is $%.2f",
                        itemAdded.getMake(),
                        itemAdded.getModel(),
                        carSale.getSaleTotal()
                    )
                );
            }

            @Override
            public void onItemRemoved(ICar itemRemoved) {
                System.out.println(
                    String.format("%s %s removed. Total sale value is $%.2f",
                        itemRemoved.getMake(),
                        itemRemoved.getModel(),
                        carSale.getSaleTotal()
                    )
                );
            }
        });
        carSale.startSale();
        System.out.println(carSale.getSaleReceipt());
        System.out.println(String.format("Total sale value is $%.2f",
carSale.getSaleTotal()))
}
```

Dessa maneira, um exemplo de saída desse programa poderia ser:

```
Choose a car, press
    0 to finish sale
    9 to remove last car:
1 - Hyunday Azera
2 - Toyota Hilux
3 - Mitsubishi Pajero Sport
4 - Chevrolet Onix
5 - Lincoln Town Car
1
Hyunday Azera added. Total sale value is $202000.00
2
Toyota Hilux added. Total sale value is $454500.00
3
Mitsubishi Pajero Sport added. Total sale value is $808000.00
9
Mitsubishi Pajero Sport removed. Total sale value is $454500.00
0
================================================
   Congratulations on your new car purchase!
================================================
Date:                      2021-12-30 16:32:49.286
Make:.....................................Hyunday
Model:....................................Azera
Engine:...............................3.0 liter(s)
Transmission:.....................8 speed automatic
Price:....................................200000.00
Taxes:....................................1.00%

Final Price:.............................202000.00
================================================

================================================
   Congratulations on your new car purchase!
================================================
Date:                      2021-12-30 16:32:49.286
Make:.....................................Toyota
Model:....................................Hilux
Engine:...............................2.8 liter(s)
Transmission:.......................6 speed manual
         Your vehicle is equipped with AWD (All wheel Dr

ive capability) please be aware that in order to f
lat tow your vehicle you have to disengage the 4x4
features first or else it will cause permanent da
mage to the car transmission.
Price:....................................250000.00
Taxes:....................................1.00%

Final Price:.............................252500.00
================================================

Total sale value is $454500.00
```

Arquitetura de microsserviços

Na segunda seção do nosso treinamento vamos entender o que é a arquitetura de microsserviços e quando ela deve ser utilizada. O modelo antagonista a essa arquitetura é chamado de monolito, existe a muitos anos e sempre foi utilizado pela indústria de software, nele todos os casos de uso são implementados em uma única API. Um exemplo clássico desse modelo são aplicações feitas com PHP ou o já ultrapassado Java Entreprise.

Em um servidor PHP reside o código que acessa o banco de dados, que faz os processamentos necessários, executa todas as regras de negócio e também responde às solicitações por páginas web que são geradas utilizando processamento do próprio servidor, ou seja, não são estáticas.

O primeiro passo no sentido de se afastar desse modelo foi a adoção de PWAs (Progressive Web Apps), que são servidores de páginas http estáticas, ou seja, não são realizados processamentos do lado do servidor, somente no lado do cliente (utilizando javascript). Essas páginas, por sua vez, acessam o conteúdo do banco de dados através de uma API Restful que pode ser codificada em qualquer linguagem para servidores de backend.

Esse modelo cliente servidor já ajuda muito a escalabilidade e separação de responsabilidades do produto, entretanto isso não é o bastante. Ainda temos todas nossas regras de negócio e processos vivendo dentro da mesma API, a situação fica pior quando é necessário fazer o deploy de uma instancia da API para cada cliente.

Quando pensamos em microsserviços pensamos em separação de responsabilidades, ou seja, identificar quais os principais casos de uso do produto que poderiam funcionar em uma API por si só e que possam ser utilizados em mais de um contexto ou por mais de um produto. Exemplo:

Um Web Broker é uma aplicação web para investimentos na bolsa de valores e deve possuir as seguintes premissas:

- Uma interface web para interação do usuário.
- O usuário deve poder consultar o valor atual de cada ação.
- A tela de consulta de ações deve ser atualizada em tempo real.
- O usuário deve poder criar ordens de compra para uma ação.
- O usuário deve poder criar ordens de venda para uma ação.
- O usuário deve estar autenticado para acessar o sistema.

Olhe para essas especificações e pense por algum tempo em quais são as responsabilidades desse sistema e em quantos microsserviços ele poderia ser dividido para garantir escalabilidade...

O desafio de desenvolvimento será codificar o sistema descrito acima e ele dever ser dividido em:

- Uma API para autenticação.
- Uma API responsável pelos cadastros das ordens de compra e venda e por realizar a compra ou a venda da ação quando houver o cadastro de uma ordem de compra compatível com uma ordem de venda já existente.
- Uma API responsável por armazenar, atualizar e permitir consulta dos preços atuais de cada ação, conforme ordens de compra e venda forem sendo criadas e fechadas. Essa API também será responsável por atualizar em tempo real os dados das ações no frontend.
- Uma aplicação PWA onde residirá todo o frontend da aplicação.

A linguagem preferencial para o desenvolvimento das APIs é o Java utilizando o framework Spring Boot. Somente vou permitir o uso de outras linguagens de programação se houver uma boa justificativa para isso.

A aplicação PWA deve ser desenvolvida com VUE JS.

A versão do java é obrigatoriamente a 17.

Dicas:

Você não precisa criar uma API de autenticação do zero, eu recomendo que você utiliza o Okta. Um exemplo bem completo de como implementar e como fazer funcionar está nesse artigo: https://developer.okta.com/blog/2021/10/04/spring-boot-spa

No nosso caso o Okta trabalhará como "authorization server" e nossas apis serão os “resource servers”. Para mais detalhes de como o padrão Oauth 2.0 de autenticação leia esse artigo: https://developer.okta.com/docs/concepts/oauth-openid/

Tanto a API de ordens de compra como a que atualiza o valor das ações devem ter bancos de dados SEPARADOS, justamente porque esse é um dos principais problemas que enfrentamos quando partimos para a arquitetura de microsserviços. Para facilitar um pouco as coisas eu recomendo que vocês utilizem o FLYWAY para realizar a migração de DDL, os primeiros arquivos de criação das tabelas já estão presentes na pasta “helper_files”.

Não precisa focar na beleza do frontend. Pode fazer o mais simples possível o foco é nas funcionalidades do backend.

A parte de atualizar o frontend em tempo real é particularmente difícil, por isso vai ser opcional, entretanto, se você decidir fazer isso, eu recomendo utilizar o PostgreSQL como banco de dados que possui uma funcionalidade de notificação PG_NOTIFY (https://jdbc.postgresql.org/documentation/81/listennotify.html). Recomendo também que crie uma API com spring webflux só para receber essas notificações e mandar para o fronted. Lembre-se você é livre para implementar como quiser, isso tudo são apenas dicas.

Funcionalidade	Pontuação
Para cada padrão de projeto implementado, independentemente de estar ou não no conteúdo do treinamento	2
Api de autenticação com padrão Oauth2.0	10
Api de cadastros de ordem de serviço	10
Api de atualização e consulta de Ações	10
Frontend em vuejs	10
Notificação para o front em tempo real	20
Diagrama que represente como as diferentes partes do sistema se comunicam	5
Apresentação powerpoint explicando o que foi implementado no projeto	5
Utilizar no projeto as novas funcionalidades do java 17 e explicar no power point quais são elas	5
Um programa em java que gere randomicamente ordens de compra e venda para alimentar o sistema e simular vários usuários usando o sistema.	15