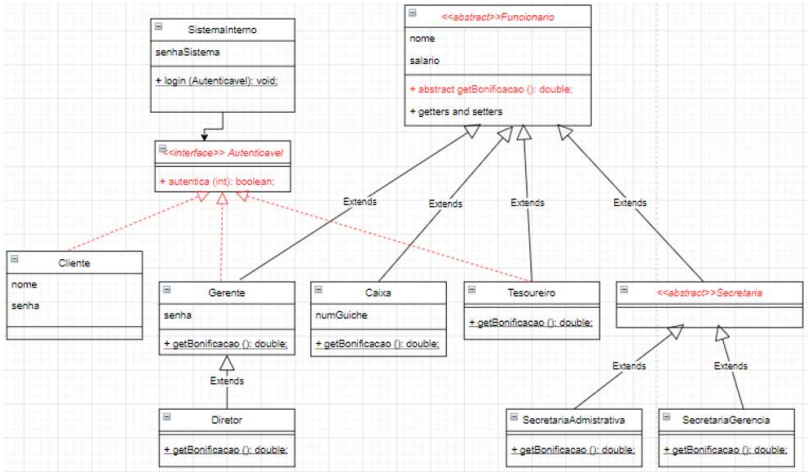
Aula 10
Exercício projeto bancov012. Exercício para ser feito em classe.

1) A partir de bancov011 crie o projeto bancov012. Abra o projeto e feche todos os outros.
2) Observe bem a hierarquia abaixo.



3) Cliente, Tesoureiro, Gerente e, consequentemente, Diretor podem se autenticar no SistemaInterno e possuem uma espécie de lógica de autenticação identica como podemos ver abaixo.

```
private int senha;

public int getSenha() {
    return this.senha;
}
public void setSenha(int senha) {
    this.senha = senha;
}
public boolean autentica(int senha) {
    if (this.senha == senha)
        return true;
    return false;
}
```

4) O código acima se repete nas classes Gerente, Cliente e Tesoureiro. Isso não chega a ser um grande problema, mas, como nosso objetivo é escrever sistemas escaláveis e evitar o espalhamento de regras, podemos fazer algo a respeito. Se pararmos para pensar, essas classes são compostas por seus atributos e por uma lógica de autenticação. Percebam que essa lógica **compõe** essas classes. Podemos concentrar esse código em um único lugar revisitando um assunto do início de nossas aulas. Podemos fazer uso de composição, ou seja, podemos criar uma classe que contenha toda a lógica de autenticação e fazer com que essa classe componha Cliente, Gerente e Tesoureiro. Utilizaremos de composição para evitar o espalhamento de regras.
5) Extraia toda a lógica de autenticação das classes citadas acima e concentre essa lógica em uma classe chamada AutenticadorLogica.

```
3  public class AutenticadorLogica {
4      private int senha;
5
6      public int getSenha() {
7          return this.senha;
8      }
9      public void setSenha(int senha) {
10         this.senha = senha;
11     }
12     public boolean autentica(int senha) {
13         if (this.senha == senha)
14             return true;
15         return false;
16     }
17 }
```

6) As 3 classes estarão acusando erro neste momento. Passo-a-passo resolveremos isso com uma solução bem elegante. Faça com que as 3 classes em questão tenham um atributo chamado autenticador do tipo AutenticadorLogica.

```
3     private AutenticadorLogica autenticador = new AutenticadorLogica();
```

7) As três classes continuam acusando erro. O problema é fácil de resolver. As classes em questão precisam ter a implementação do método autentica uma vez que implementam a interface Autenticavel. Veja o código contido em Gerente e que serve também para Cliente e Tesoureiro. Se você observar bem o código, o autentica dessas 3 classes passará a invocar o autentica de AutenticadorLogica. Com isso, toda a regra de autenticação estará concentrada em um único lugar. O método autentica de Cliente, Gerente e Tesoureiro será como uma espécie de "casca" para a invocação do verdadeiro e único método autentica do nosso sistema.

```
3  public class Cliente implements Autenticavel{
4      private String nome;
5      private String cpf;
6      private String email;
7      private AutenticadorLogica autenticador = new AutenticadorLogica();
8      @Override
9      public boolean autentica(int senha) {
10         //Invocando o método autentica de AutenticadorLogica
11         return this.autenticador.autentica(senha);
12     }
13     //Construtores
```

8) Ao fazer as modificações, as 3 classes deixarão de acusar erro. No entanto, a classe TestaSistemaInterno acusa erros na invocação do método setSenha das 3 classes. Para resolver isso, não basta simplesmente implementar o setSenha nessas classes. Precisamos fornecer a garantia de que Cliente, Gerente e Tesoureiro possuem um método setSenha. Vamos fornecer essa garantia através da interface Autenticavel. Agora, para ser autenticável, além de fornecer uma implementação para o método autentica, temos que fornecer uma implementação de setSenha. Veja:

```
3  public interface Autenticavel {
4      boolean autentica(int senha);
5      void setSenha(int senha);
6  }
```

9) Por fim, basta implementar o método setSenha nas 3 classes. Assim como fizemos com o método autentica, o método setSenha dessas 3 classes será apenas uma "casca" para o verdadeiro e único setSenha do nosso sistema. Esse setSenha e suas eventuais regras de validação continuará apenas na classe AutenticadorLogica, ou seja, essa regra também ficará concentrada em um único local. Veja como ficará o setSenha dessas classes.

```
@Override
public void setSenha(int senha) {
    //Invocando o setSenha de AutenticadorLogica
    this.autenticador.setSenha(senha);
}
```

10) Pronto! Problema resolvido. Agora, Cliente, Tesoureiro, Gerente e, consequentemente, Diretor possuem um atributo autenticador do tipo AutenticadorLogica e por implementarem a interface Autenticavel possuem um método autentica e um método setSenha. Veja como ficaram as classes.

Cliente

```java
public class Cliente implements Autenticavel{
    private String nome;
    private String cpf;
    private String email;
    private AutenticadorLogica autenticador = new AutenticadorLogica();
    @Override
    public boolean autentica(int senha) {
        //Invocando o método autentica de AutenticadorLogica
        return this.autenticador.autentica(senha);
    }
    @Override
    public void setSenha(int senha) {
        //Invocando o setSenha de AutenticadorLogica
        this.autenticador.setSenha(senha);
    }
    //Restante do código....
```

Tesoureiro

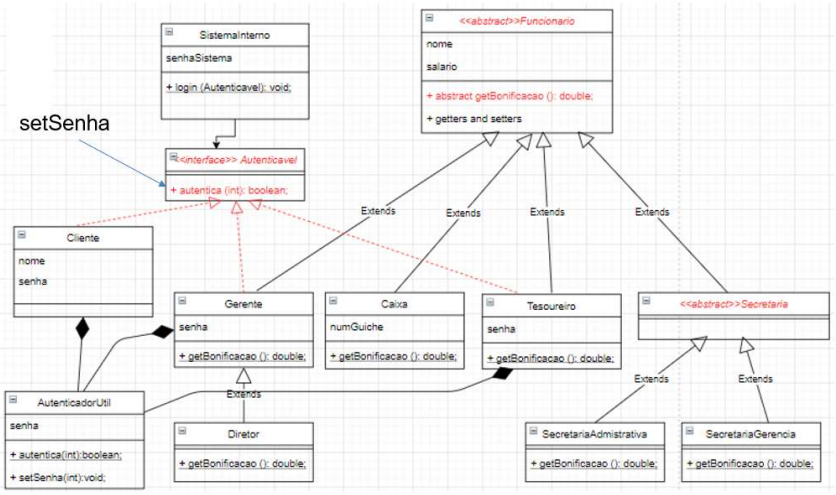
```java
public class Tesoureiro extends Funcionario implements Autenticavel{
    private AutenticadorLogica autenticador = new AutenticadorLogica();
    @Override
    public boolean autentica(int senha) {
        //Invocando o método autentica de AutenticadorLogica
        return this.autenticador.autentica(senha);
    }
    @Override
    public void setSenha(int senha) {
        //Invocando o setSenha de AutenticadorLogica
        this.autenticador.setSenha(senha);
    }
    @Override
    public double getBonificacao() {
        //super é mais descritivo uma vez que o atributo está na superclasse
        return super.salario * 0.20;
    }
}
```

Gerente

```java
public class Gerente extends Funcionario implements Autenticavel{
    private AutenticadorLogica autenticador = new AutenticadorLogica();
    @Override
    public boolean autentica(int senha) {
        //Invocando o método autentica de AutenticadorLogica
        return this.autenticador.autentica(senha);
    }
    @Override
    public void setSenha(int senha) {
        //Invocando o setSenha de AutenticadorLogica
        this.autenticador.setSenha(senha);
    }
    @Override
    public double getBonificacao() {
        return this.salario * 0.30;
    }
}
```

11) Agora basta rodar a classe SistemaInterno e verá que tudo voltou a funcionar normalmente, mas agora com um código muito mais elegante, bem escrito e escalável. Afinal, agora, caso seja necessário efetuar qualquer mudança na lógica de autenticação, as regras estarão em um único lugar. Na classe AutenticadorLogica. Utilizamos composição junto com interface e ganhamos reutilização de código. Veja o diagrama de classes.

Novo diagrama de classes



12) Essa técnica de concentrar determinadas regras de uma lógica em uma única classe (um único objeto) também é conhecida como Value Objects. AutenticadorLogica é um value Object.