

© Pedro Alves 2020

Object Oriented Design

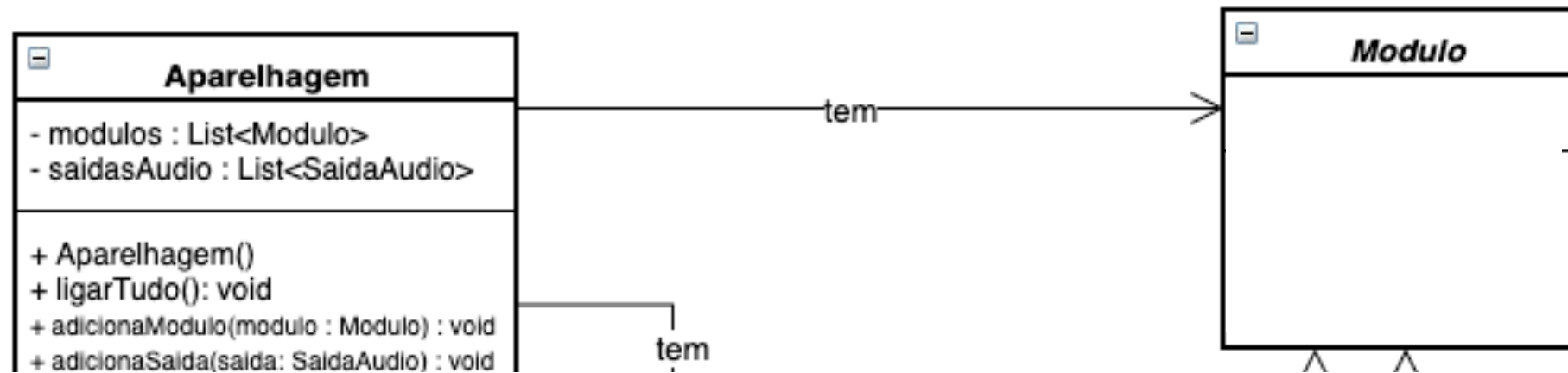


TPC teórico 3

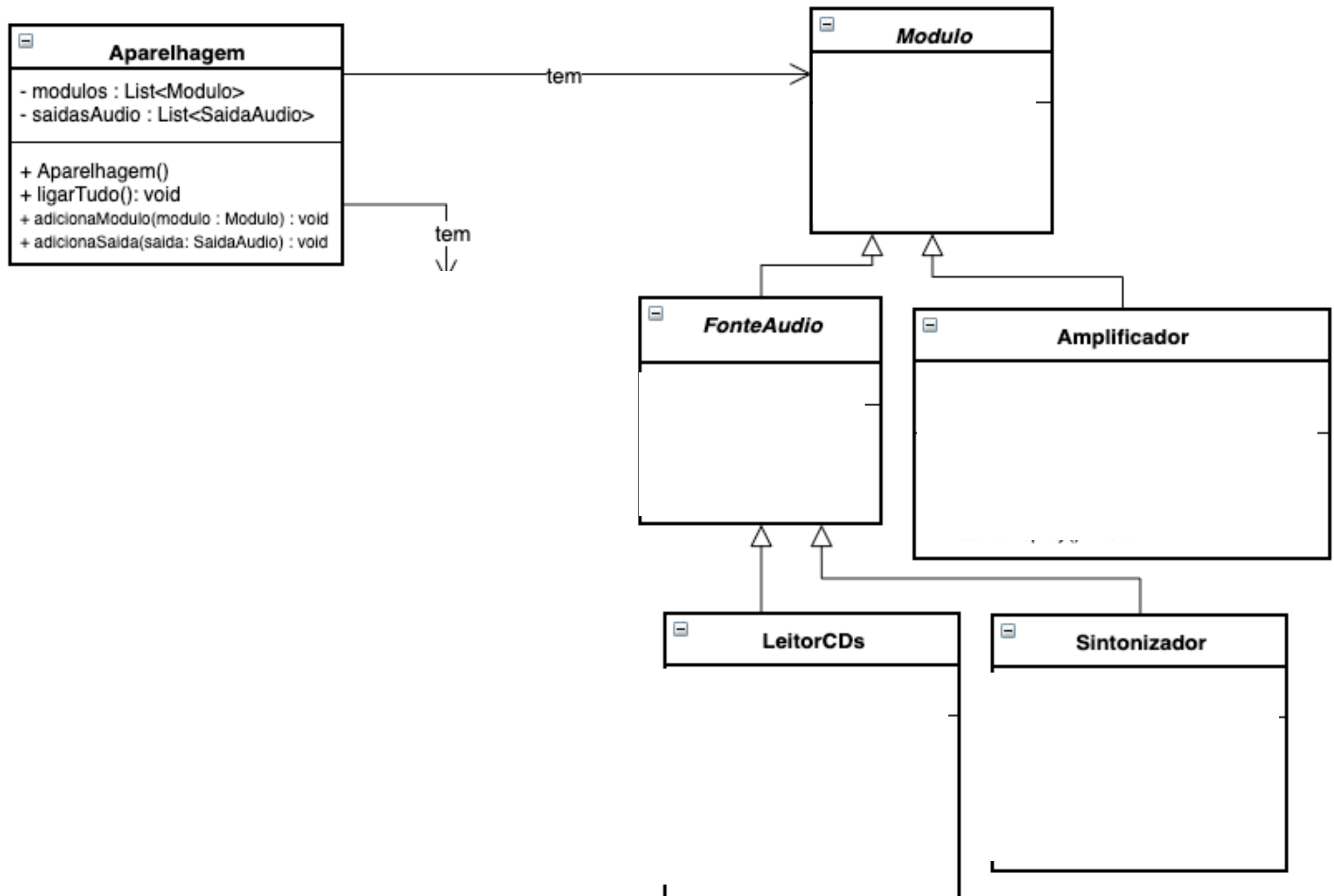
Desenhe o diagrama de classes de uma Aparelhagem constituída por módulos independentes
(deverá usar herança, pelo menos uma classe deverá ser abstracta e usar os mecanismos de visibilidade, ver cábula)



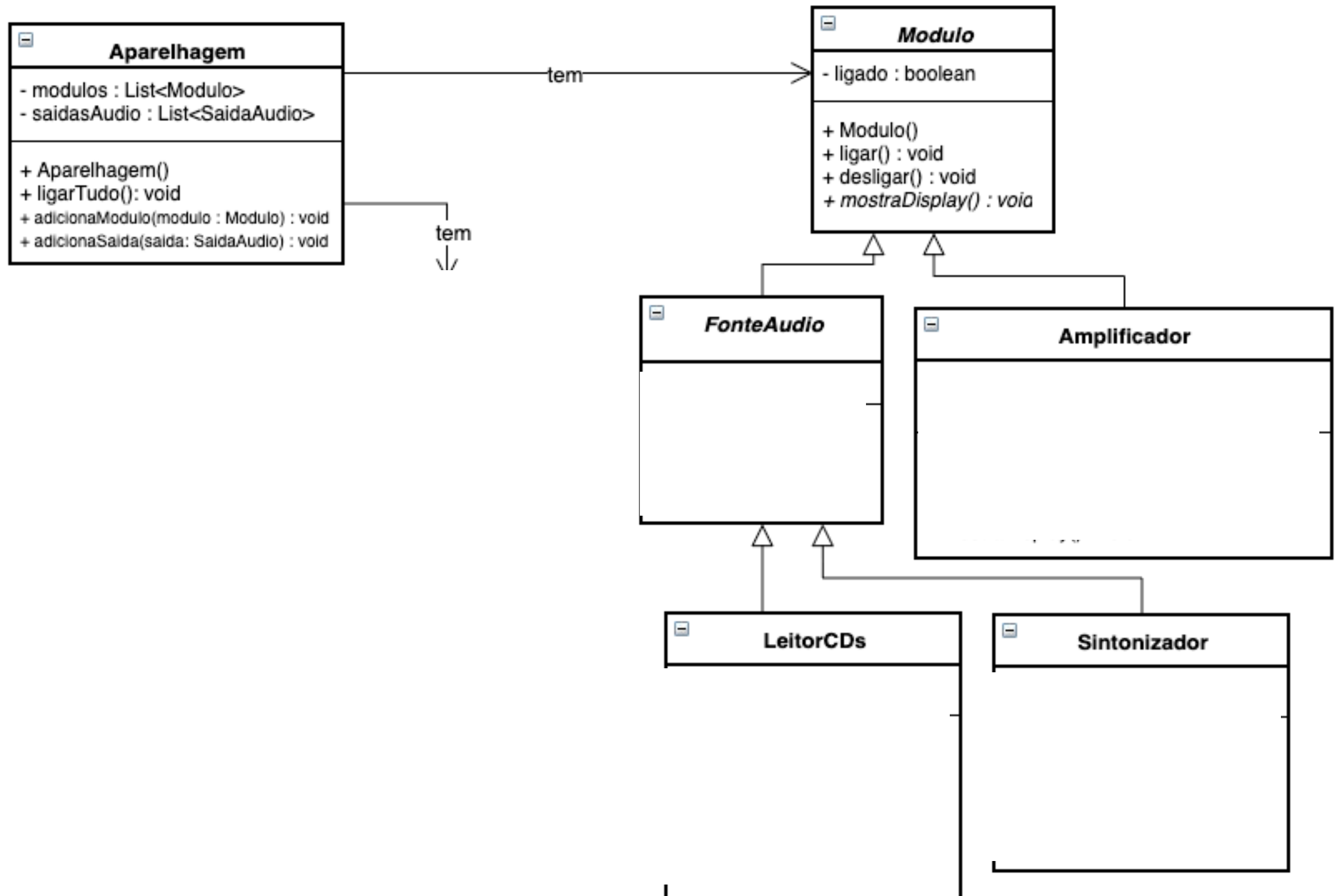
Resolução



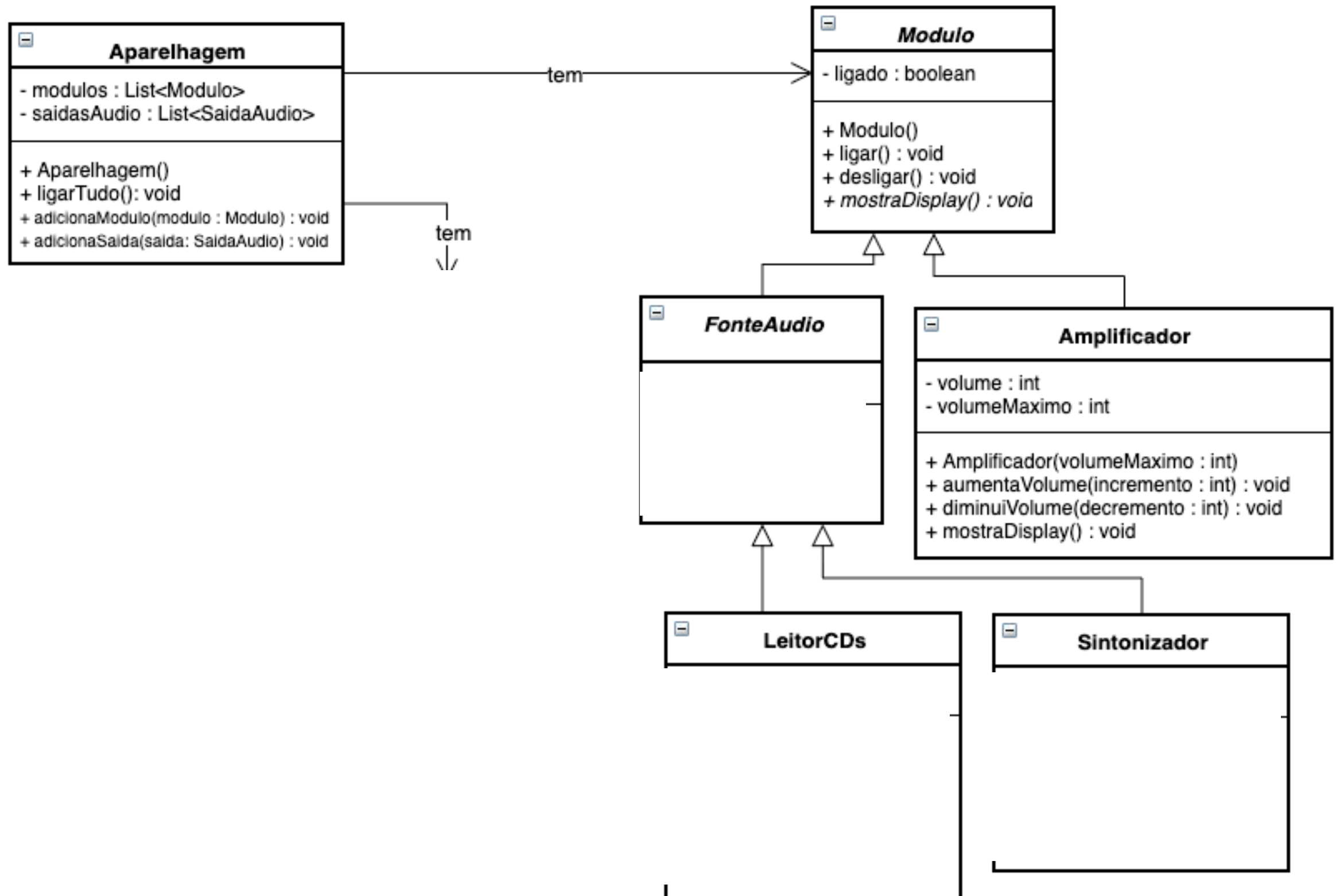
Resolução



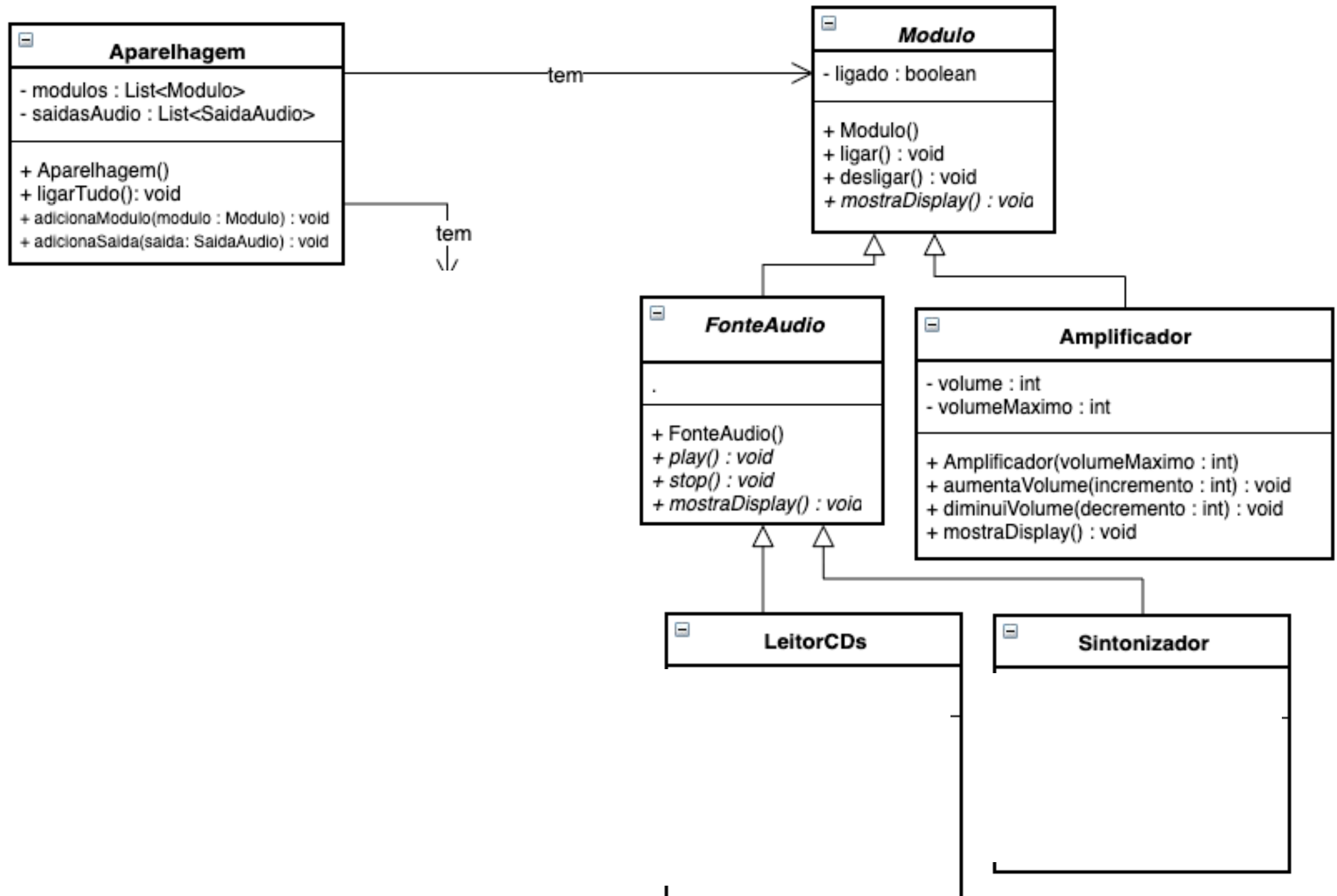
Resolução



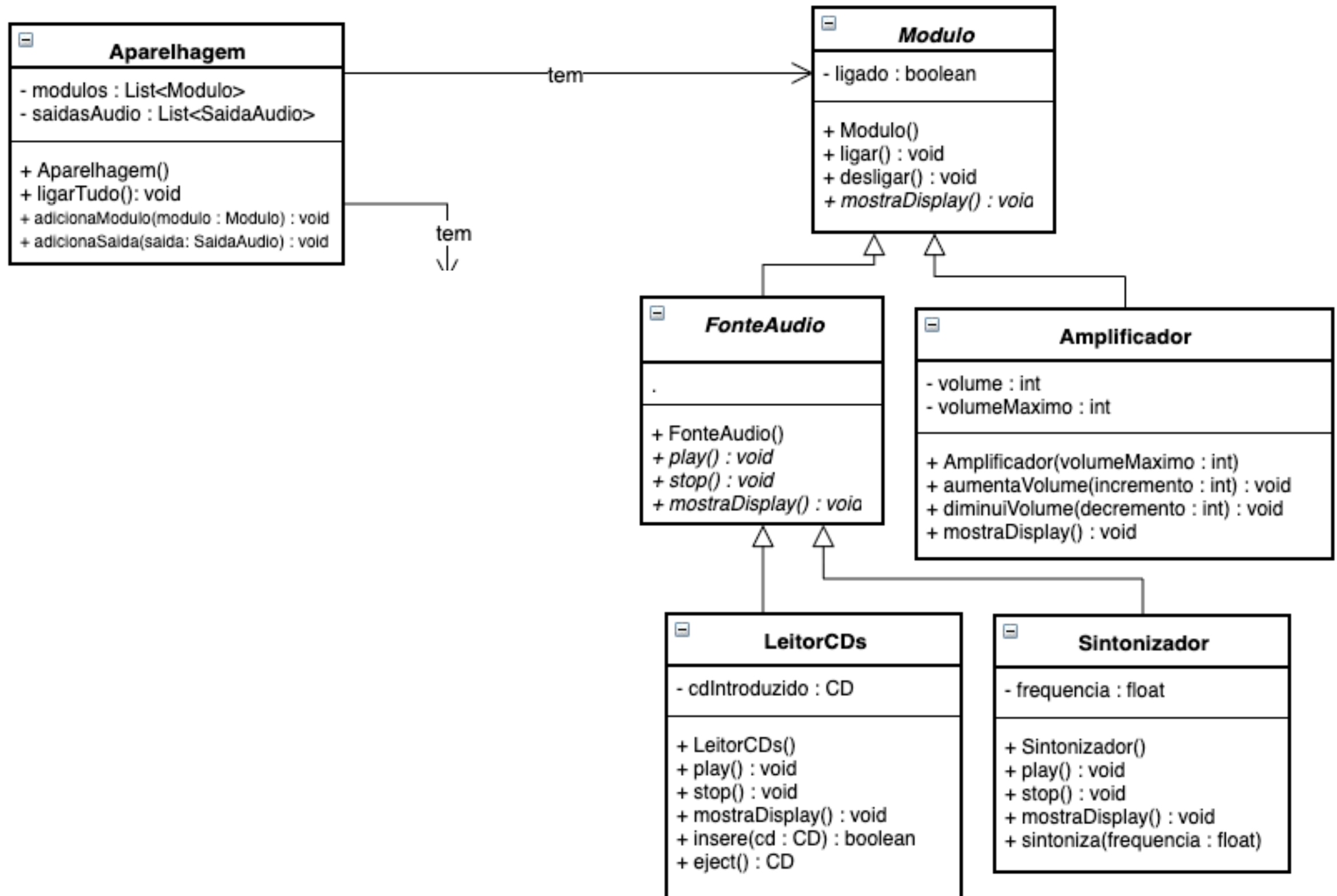
Resolução



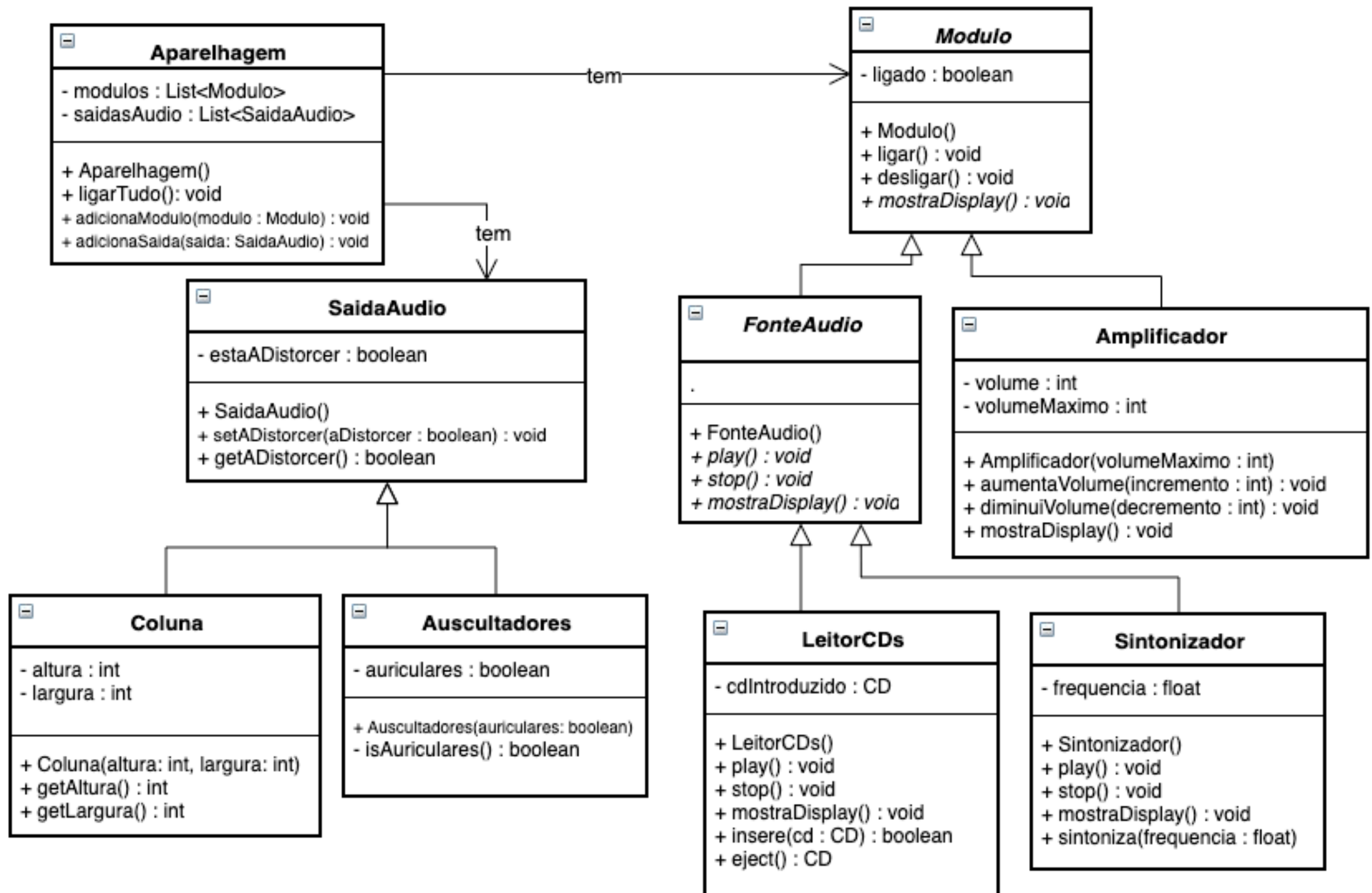
Resolução



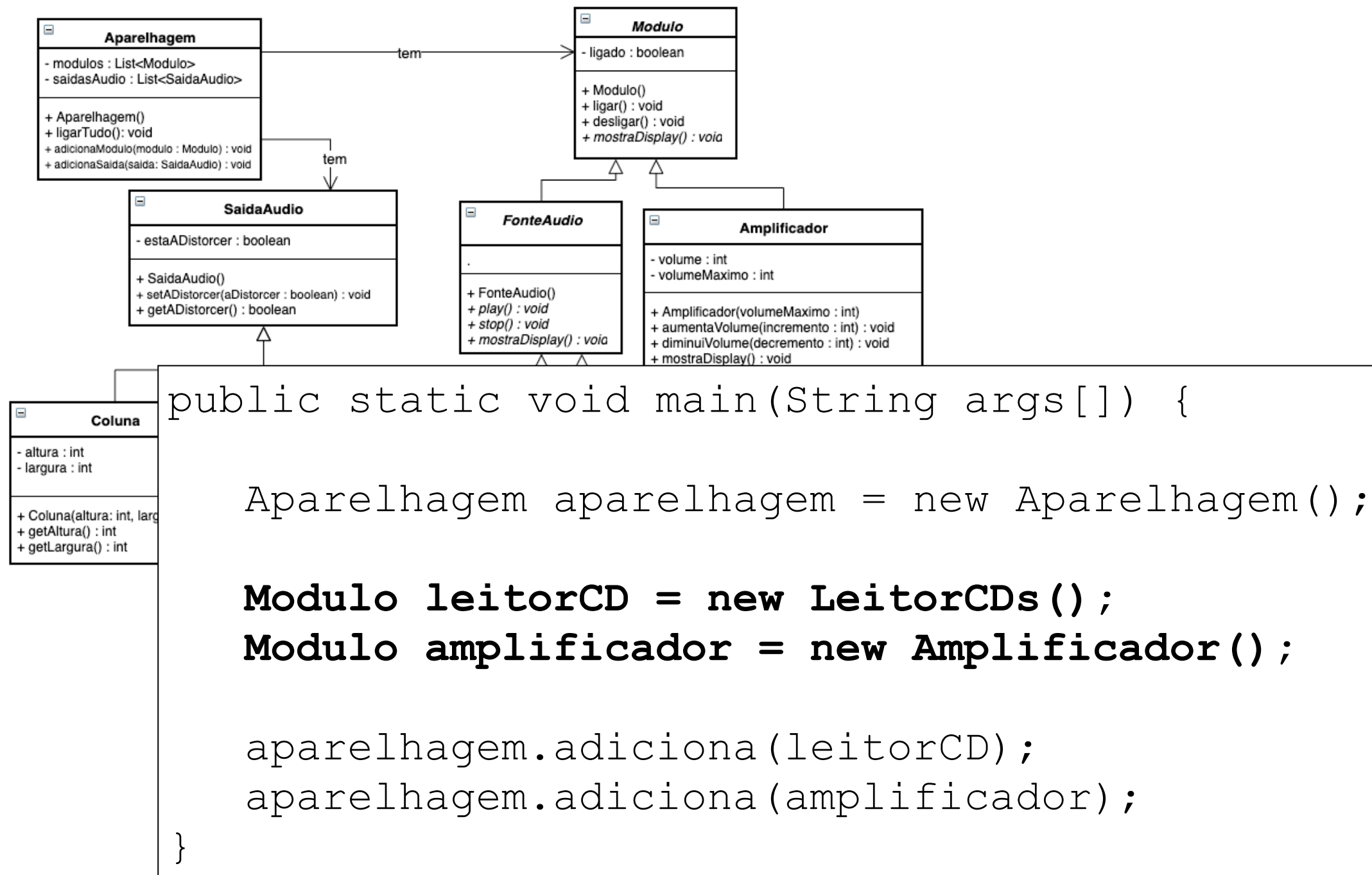
Resolução



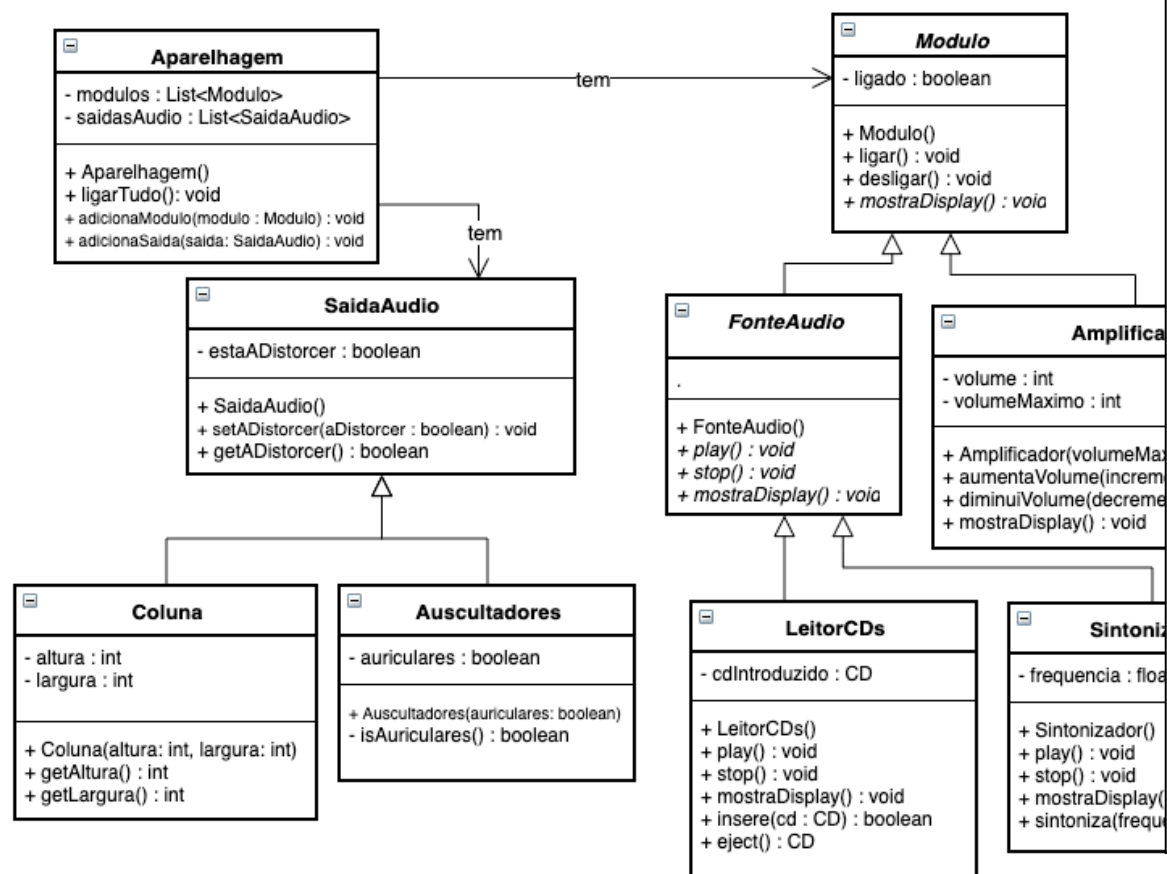
Resolução



Polimorfismo



Polimorfismo



```

class Aparelhagem {

    private List<Modulo> modulos;

    ...

    public void ligarTudo() {

        for (Modulo modulo: modulos) {
            modulo.ligar();
        }

    }

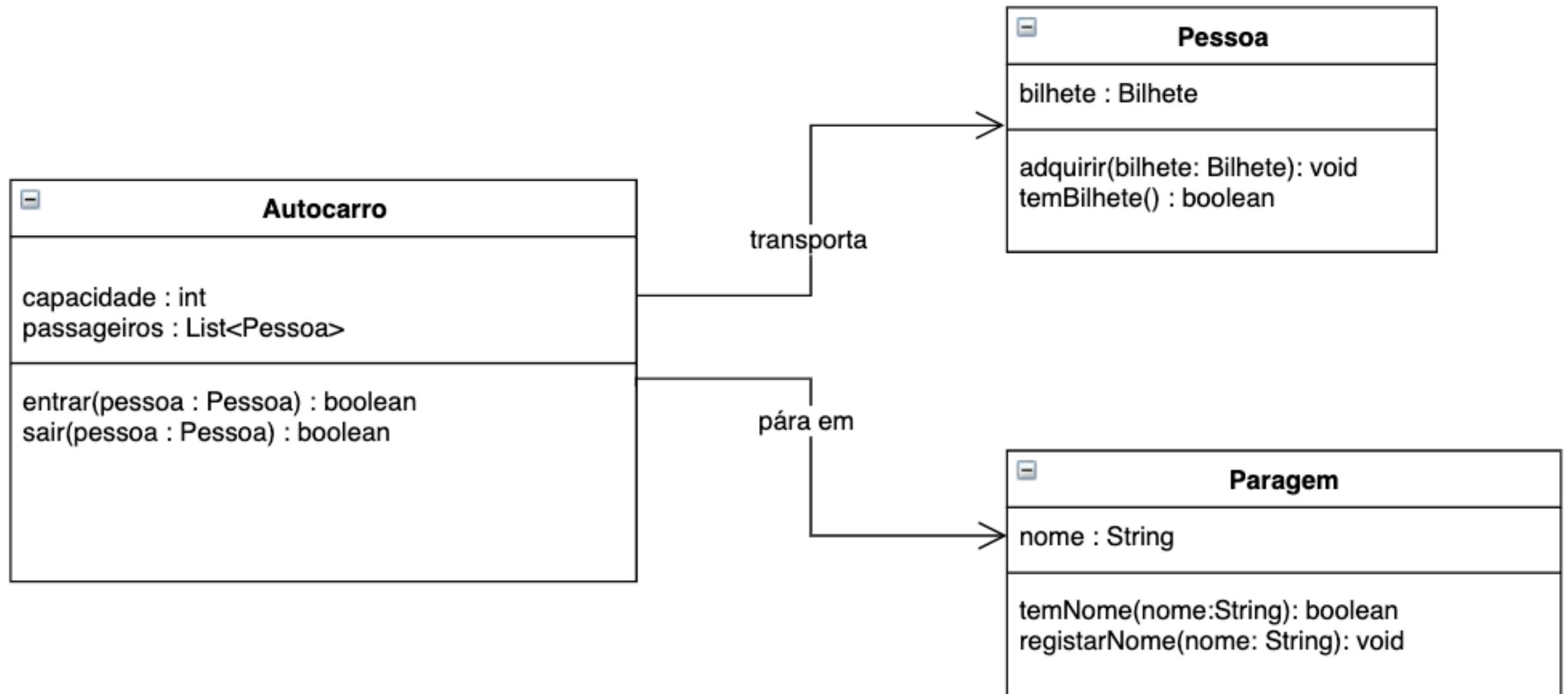
}
    
```

Se algum dos módulos redefinisse o método “ligar()”, seria esse o método que era chamado

Unit tests (again!)



Exercício do Autocarro



Testes unitários - exercício

estado inicial	acções	retorno esperado	estado final
capacidade = 4, passageiros = [pessoaC,pessoaD],	entrar(pessoaA)	true	capacidade = 4, passageiros = [pessoaC,pessoaD,pessoaA]
capacidade = 4, ??	sair(pessoaB)	true	??
capacidade = 4, ??	entrar(pessoaC)	false	??
capacidade = 4, ??	entrar(pessoaC)	false	??
capacidade = 4, ??	sair(pessoaC)	false	??

Nota: [A,B] significa uma lista com os objectos A e B

Testes unitários - resolução

estado inicial	acções	retorno esperado	estado final
capacidade = 4, passageiros = [pessoaC,pessoaD]	entrar(pessoaA)	true	capacidade = 4, passageiros = [pessoaC,pessoaD,pessoaA]
capacidade = 4, passageiros = [pessoaB]	sair(pessoaB)	true	capacidade = 4, passageiros = []
capacidade = 4, passageiros = [pessoaC]	entrar(pessoaC)	false	capacidade = 4, passageiros = [pessoaC]
capacidade = 4, passageiros = [pessoaA,pessoaB, pessoaD, pessoaE]	entrar(pessoaC)	false	capacidade = 4, passageiros = [pessoaA,pessoaB, pessoaD, pessoaE]
capacidade = 4, passageiros = []	sair(pessoaC)	false	capacidade = 4, passageiros = []

Testes unitários - implementação

estado inicial	acções	retorno esperado	estado final
capacidade = 4, passageiros = []	sair(pessoaC)	false	capacidade = 4, passageiros = []

```
@Test
public void testSairVazio() {

    Pessoa pessoaC = new Pessoa();
    Autocarro autocarro = new Autocarro(4); // autocarro começa vazio

    boolean success = autocarro.sair(pessoaC);

    if (success) {
        fail("O retorno devia ter sido false");
    }

    if (!autocarro.getPassageiros().isEmpty()) {
        fail("A lista de passageiros devia ser vazia");
    }
}
```

Testes unitários


estado inicial	acções	retorno esperado	estado final
capacidade = 4, passageiros = []	sair(pessoaC)	false	capacidade = 4, passageiros = []
capacidade = 4, passageiros = []	sair(pessoaD)	false	capacidade = 4, passageiros = []

Vale a pena fazer estes dois testes?

Testes unitários

estado inicial	acções	retorno esperado	estado final
capacidade = 4, passageiros = []	sair(pessoaC)	false	capacidade = 4, passageiros = []
capacidade = 4, passageiros = []	sair(pessoaD)	false	capacidade = 4, passageiros = []

Não, porque ambos executam o mesmo código. Se funciona para um também vai funcionar para o outro




```
public boolean sair(Pessoa p) {  
    if (!this.passageiros.contains(p)) {  
        return false;  
    }  
    this.passageiros.remove(p);  
    return true;  
}
```

Testes unitários

estado inicial	acções	retorno esperado	estado final
capacidade = 4, passageiros = []	sair(pessoaC)	false	capacidade = 4, passageiros = []
capacidade = 4, passageiros = []	sair(pessoaD)	false	capacidade = 4, passageiros = []

E este trecho de código?
Será que algum teste
executa isto?

```
public boolean sair(Pessoa p) {  
    if (!this.passageiros.contains(p)) {  
        return false;  
    }  
    this.passageiros.remove(p);  
    return true;  
}
```

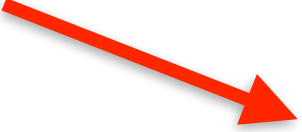


Testes unitários

Se temos código que nenhum teste executa, estamos numa situação de risco pois pode ser precisamente nesse código que estão os bugs da aplicação

Perigo! Nenhum teste executa isto!!

```
public boolean sair(Pessoa p) {  
    if (!this.passageiros.contains(p)) {  
        return false;  
    }  
    this.passageiros.remove(p);  
    return true;  
}
```



Cobertura

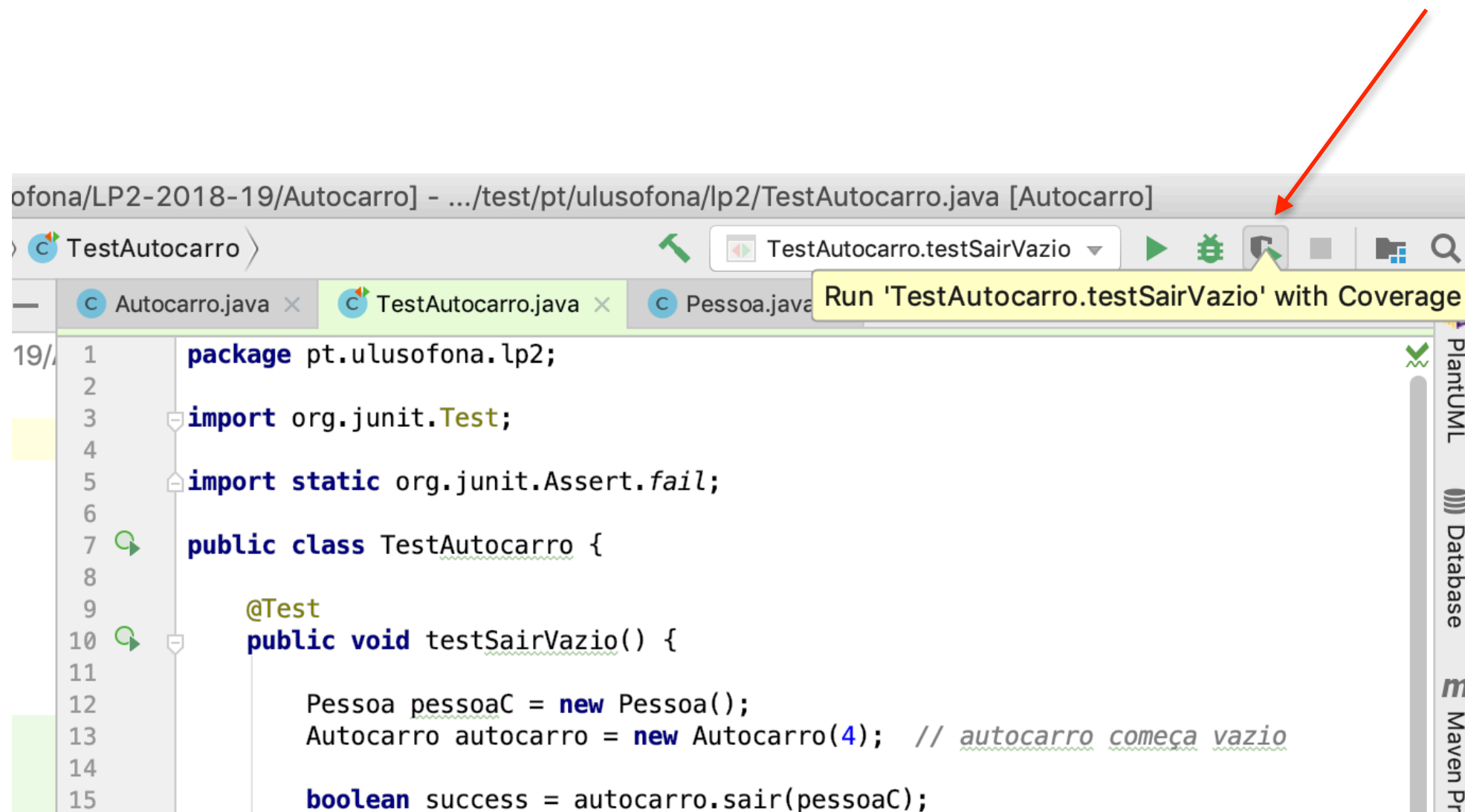
A cobertura dos testes é um indicador da percentagem de código que está coberto pelos testes.

Ex: A cobertura do projecto X é 73%.

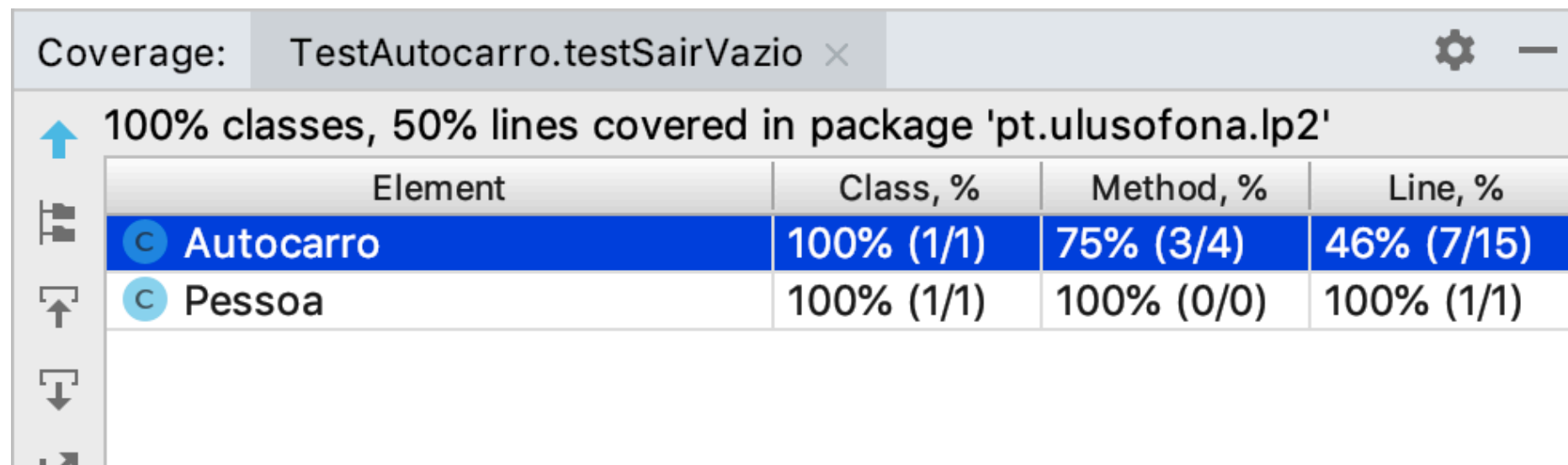
Significa que 73% das linhas de código do projecto são executadas quando se corre toda a bateria de testes

Cobertura

É possível calcular a cobertura dos testes no IntelliJ utilizando este botão



Cobertura



Coverage: TestAutocarro.testSairVazio x

100% classes, 50% lines covered in package 'pt.ulusofona.lp2'

Element	Class, %	Method, %	Line, %
Autocarro	100% (1/1)	75% (3/4)	46% (7/15)
Pessoa	100% (1/1)	100% (0/0)	100% (1/1)

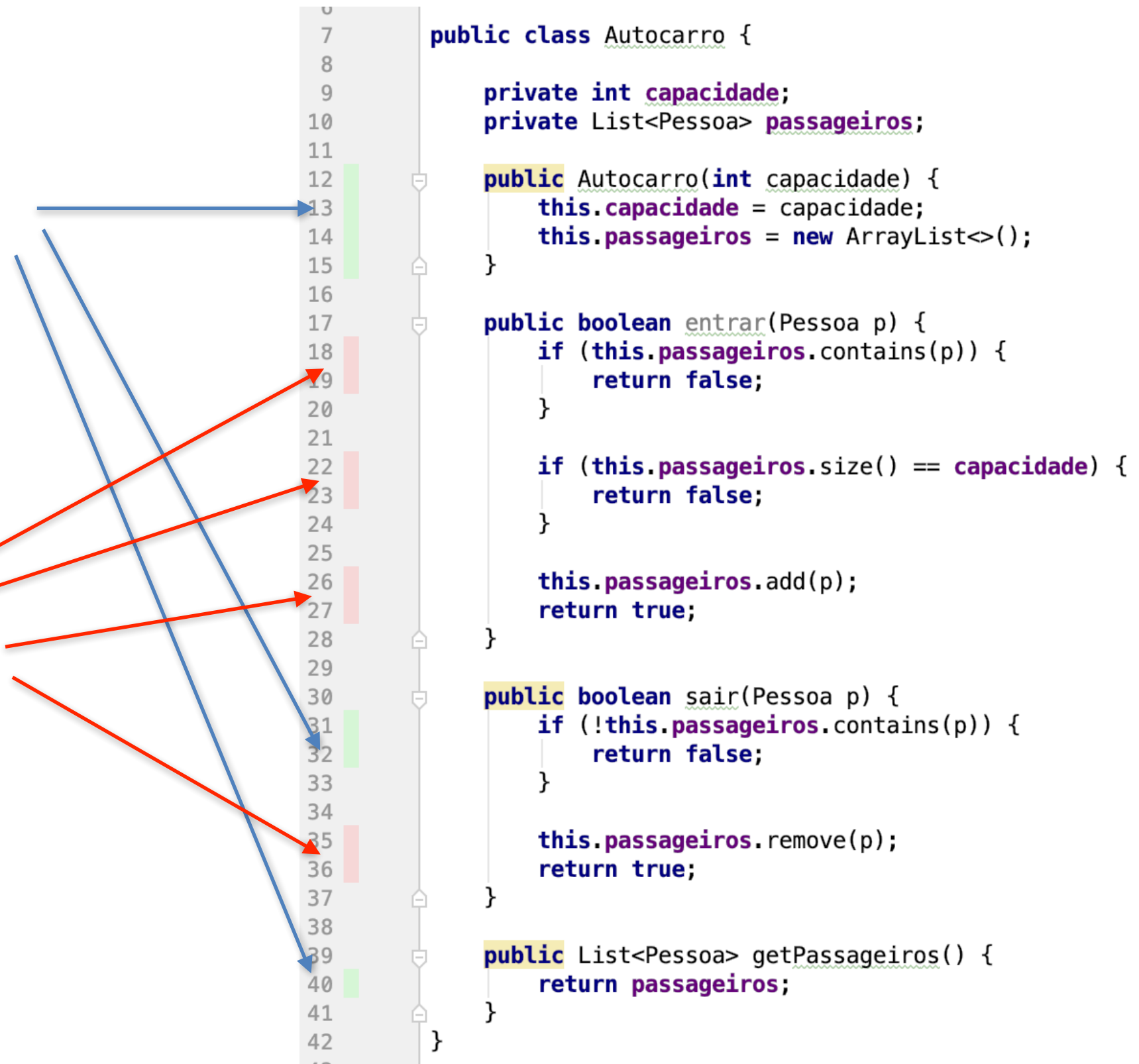
Para cada classe do Projecto, aparece a sua cobertura:

- Cobertura de classes: Para saber quais as classes que não são executados de todo durante os testes
- Cobertura de métodos: Para saber quais os métodos que não são executados
- Cobertura de linhas: Para saber quais as linhas que não são executadas

Cobertura

A verde aparece o
que foi executado

A vermelho
aparece o que não
foi executado



Padrões de desenho

Um padrão de desenho é uma solução genérica para um certo tipo de problemas

Se soubermos alguns padrões de desenho, evitamos estar sempre a “**reinventar a roda**”

Padrões de desenho

Na prática, um padrão de desenho consiste num conjunto de uma ou mais classes (com variáveis e métodos) implementados de uma forma genérica

Exemplos: Singleton, Factory, Observer-Observable, etc.

Problema

Pegando novamente na Conta Bancária....

```
class Conta {  
  
    private long saldo;  
    private String iban;  
  
    public void deposita(int montante) {  
        saldo += montante;  
    }  
  
    public void levanta(int montante) {  
        if (montante <= saldo) {  
            saldo -= montante;  
        }  
    }  
}
```

Problema

Pegando novamente na Conta Bancária....

```
class Conta {  
  
    private long saldo;  
    private String iban;  
  
    public void deposita(int montante) {  
        saldo += montante;  
    }  
  
    public void levanta(int montante) {  
        if (montante <= saldo) {  
            saldo -= montante;  
        }  
    }  
}
```

Queremos registrar todas as operações para efeitos de auditoria

Problema

Pegando novamente na Conta Bancária....

```
class Conta {  
  
    private long saldo;  
    private String iban;
```

Queremos registrar todas as operações para efeitos de auditoria

Log:

02-12-2020 10:34:53 - Depositados 5 euros na conta PT5034543534534534345462

02-12-2020 11:17:01 - Levantados 10 euros da conta PT5099993432343221112111

...

```
public void levanta(int montante) {  
    if (montante <= saldo) {  
        saldo -= montante;  
    }  
}
```

```
}
```

Resolução (1)

```
class Conta {  
  
    private long saldo;  
    private String iban;  
  
    public void deposita(int montante) {  
        saldo += montante;  
        System.out.println(new Date() + " - Depositados " +  
                             montante + " euros na conta " + iban);  
    }  
  
    public void levanta(int montante) {  
        if (montante <= saldo) {  
            saldo -= montante;  
            System.out.println(new Date() + " - Levantados " +  
                                 montante + " euros da conta " + iban);  
        }  
    }  
}
```

Resolução

```
class Conta {
```

```
    private long saldo;  
    private String iban;
```

```
    public void deposita(int montante) {
```

```
        saldo += montante;
```

```
        System.out.println(new Date() + " - Depositados " +  
                             montante + " euros na conta " + iban);
```

```
    }
```

```
    public void levanta(int montante) {
```

```
        if (montante <= saldo) {
```

```
            saldo -= montante;
```

```
            System.out.println(new Date() + " - Levantados " +  
                                 montante + " euros da conta " + iban);
```

```
        }
```

```
    }
```

```
}
```

Código semi-duplicado!

Resolução (2)

```
class Conta {  
  
    private long saldo;  
    private String iban;  
  
    public void log(String message) {  
        System.out.println(new Date() + " - " + message);  
    }  
  
    public void deposita(int montante) {  
        saldo += montante;  
        log("Depositados " + montante + " euros na conta " + iban);  
    }  
  
    public void levanta(int montante) {  
        if (montante <= saldo) {  
            saldo -= montante;  
            log("Levantados " + montante + " euros da conta " + iban);  
        }  
    }  
}
```


Resolução (2)

```
class Conta {  
  
    private long saldo;  
    private String iban;  
  
    public void log(String message) {  
        System.out.println(new Date() + " - " + message);  
    }  
  
    public void deposita(int montante) {  
        saldo += montante;  
        log("Depositados " + montante + " euros na conta " + iban);  
    }  
  
    public void levanta(int montante) {  
        if (montante <= saldo) {  
            saldo -= montante;  
            log("Levantados " + montante + " euros da conta " + iban);  
        }  
    }  
}
```

Este método não devia estar na classe Conta. E se eu quiser registrar operações noutras classes?

Resolução (3)

```
class Logger {  
  
    public void log(String message) {  
        System.out.println(new Date() + " - " + message);  
    }  
}
```

```
class Conta {  
  
    private long saldo;  
    private String iban;  
    private Logger logger;  
  
    public Conta(String iban, Logger logger) {  
        this.iban = iban;  
        this.logger = logger;  
        this.saldo = 0;  
    }  
  
    public void deposita(int montante) {  
        saldo += montante;  
        logger.log("Depositados " + montante + " euros na conta " + iban);  
    }  
  
    ...  
}
```

Resolução (3)

```
class Logger {  
  
    public void log(String message) {  
        System.out.println(new Date() + " - " + message);  
    }  
}
```

Esta classe não tem atributos??

```
class Conta {  
  
    private long saldo;  
    private String iban;  
    private Logger logger;  
  
    public Conta(String iban, Logger logger) {  
        this.iban = iban;  
        this.logger = logger;  
        this.saldo = 0;  
    }  
  
    public void deposita(int montante) {  
        saldo += montante;  
        logger.log("Depositados " + montante + " euros na conta " + iban);  
    }  
  
    ...  
}
```

Resolução (4)

```
class Logger {
```

```
    private boolean enabled;
```

```
    public void log(String message) {
```



```
        if (enabled) {
```

```
            System.out.println(new Date() + " - " + message);
```

```
        }
```

```
    }
```

```
    // métodos para ligar/desligar
```

```
}
```

Resolução (4)

```
class Logger {  
    private boolean enabled;  
  
    public void log(String message) {  
        → if (enabled) {  
            System.out.println(new Date() + " - " + message);  
        }  
    }  
  
    // métodos para ligar/desligar  
}
```

Como garantir que há apenas um Logger global de forma a que quando ligo, ligo para toda a aplicação?

Solução

Singleton pattern

Padrão usado para garantir que existe apenas uma instância do objeto em toda a aplicação

Solução

Passo 1 - Tornar o construtor private de forma a que ninguém consiga instanciar

```
class Logger {  
    private Logger() {  
    }  
  
    private boolean enabled;  
  
    public void log(String message) {  
        if (enabled) {  
            System.out.println(new Date() + " - " + message);  
        }  
    }  
  
    // métodos para ligar/desligar  
}
```

Solução

Passo 1 - Tornar o construtor private de forma a que ninguém consiga instanciar

```
class Logger {  
    private Logger() {  
    }  
  
    private boolean enabled;  
  
    public void log(String message) {  
        if (enabled) {  
            System.out.println(new Date() + " - " + message);  
        }  
    }  
  
    // métodos para ligar/desligar  
}
```


Solução

Passo 2 - Criar uma variável global que guarde a única instância da aplicação

```
class Logger {  
  
    public static Logger instance = new Logger();  
  
    private Logger() {  
    }  
  
    private boolean enabled;  
  
    public void log(String message) {  
        if (enabled) {  
            System.out.println(new Date() + " - " + message);  
        }  
    }  
  
    // métodos para ligar/desligar  
}
```

Solução

```
class Conta {  
  
    private long saldo;  
    private String iban;  
    private Logger logger = Logger.instance;  
  
    public Conta(String iban) {  
        this.iban = iban;  
        this.saldo = 0;  
    }  
  
    public void deposita(int montante) {  
        saldo += montante;  
        logger.log("Depositados " + montante + " euros na conta " + iban);  
    }  
}
```

Factory

O factory é uma classe cuja principal finalidade é instanciar objectos (uma fábrica de objectos)



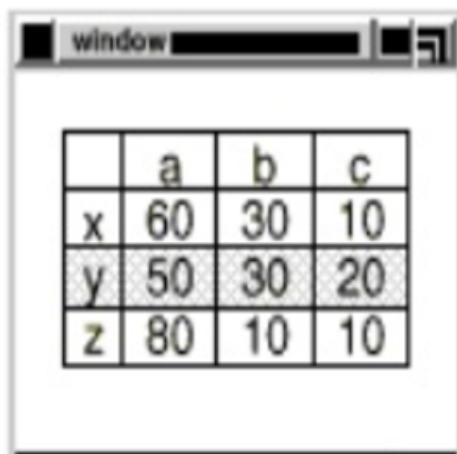
Factory

```
public class FigurasGeometricasFactory {  
  
    public static FiguraGeometrica criaFiguraAleatoria(String nomeFigura) {  
  
        int altura = ..., largura = ...; // gera uma altura e largura aleatória  
        int x = ..., y = ...; // gera um par de coordenadas aleatórias  
  
        FiguraGeometrica figura = null;  
        switch (nomeFigura) {  
            case "Quadrado":  
                return new Quadrado(x, y, altura, getCorAleatoria());  
            case "Retângulo":  
                return new Rectangulo(x, y, altura, largura, getCorAleatoria());  
            default:  
                throw new IllegalArgumentException("Nome da figura desconhecido: " +  
                    nomeFigura);  
        }  
    }  
}
```

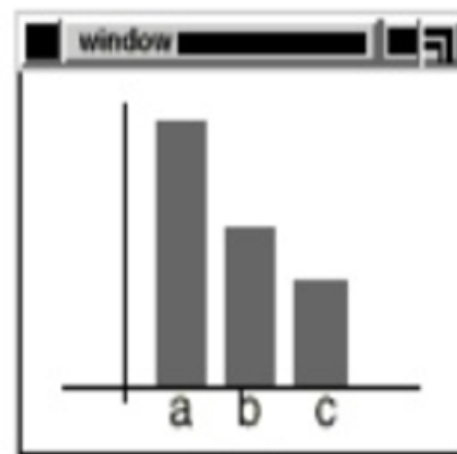
Observer-Observable

O Observer-Observable é um padrão usado quando certos objectos (observadores) querem ser “avisados” de que houve alterações noutros objectos (observados)

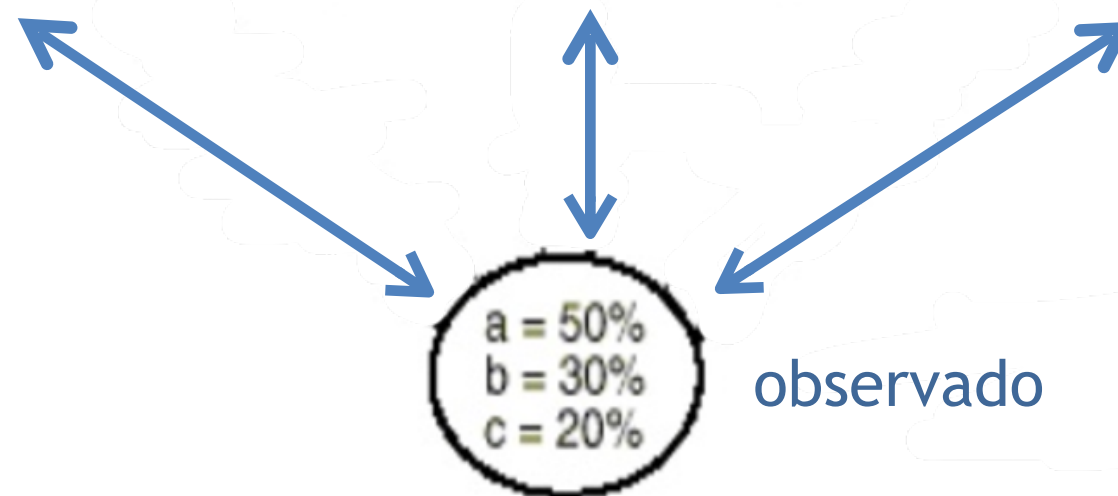
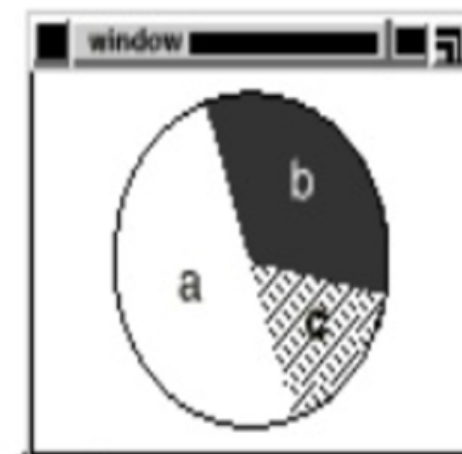
observador 1



observador 2

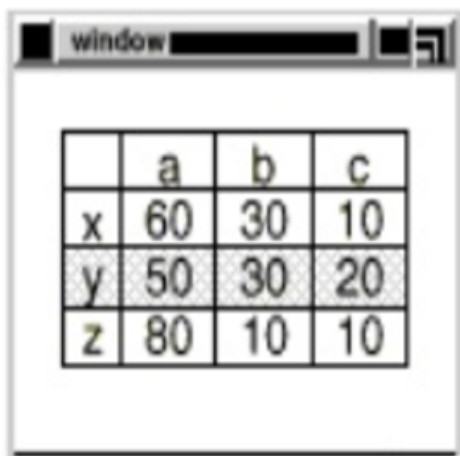


observador 3

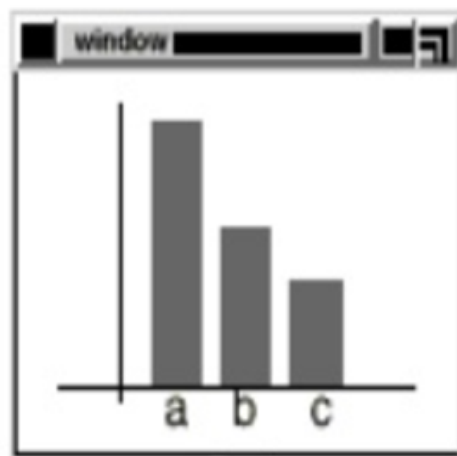


Observer-Observable

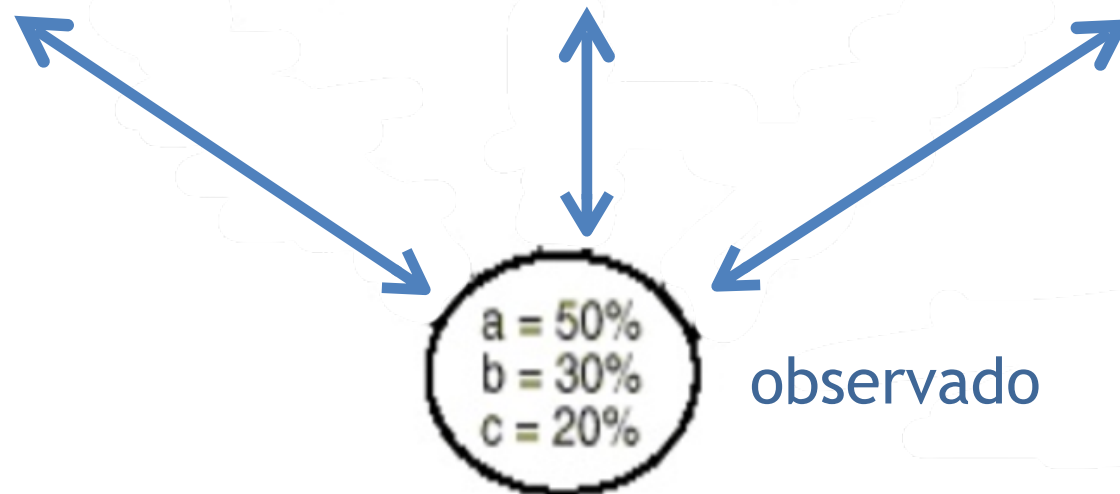
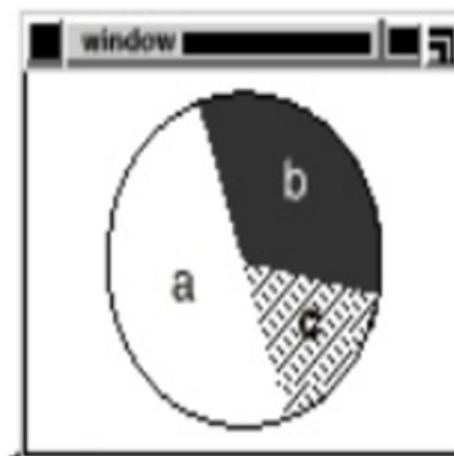
observador 1



observador 2



observador 3



Quando uma das variáveis muda, os observadores são avisados (por exemplo para redesenharem o gráfico)

```
class Estatistica {  
    private int a, b, c;  
  
    // setters e getters  
}
```

Observer-Observable

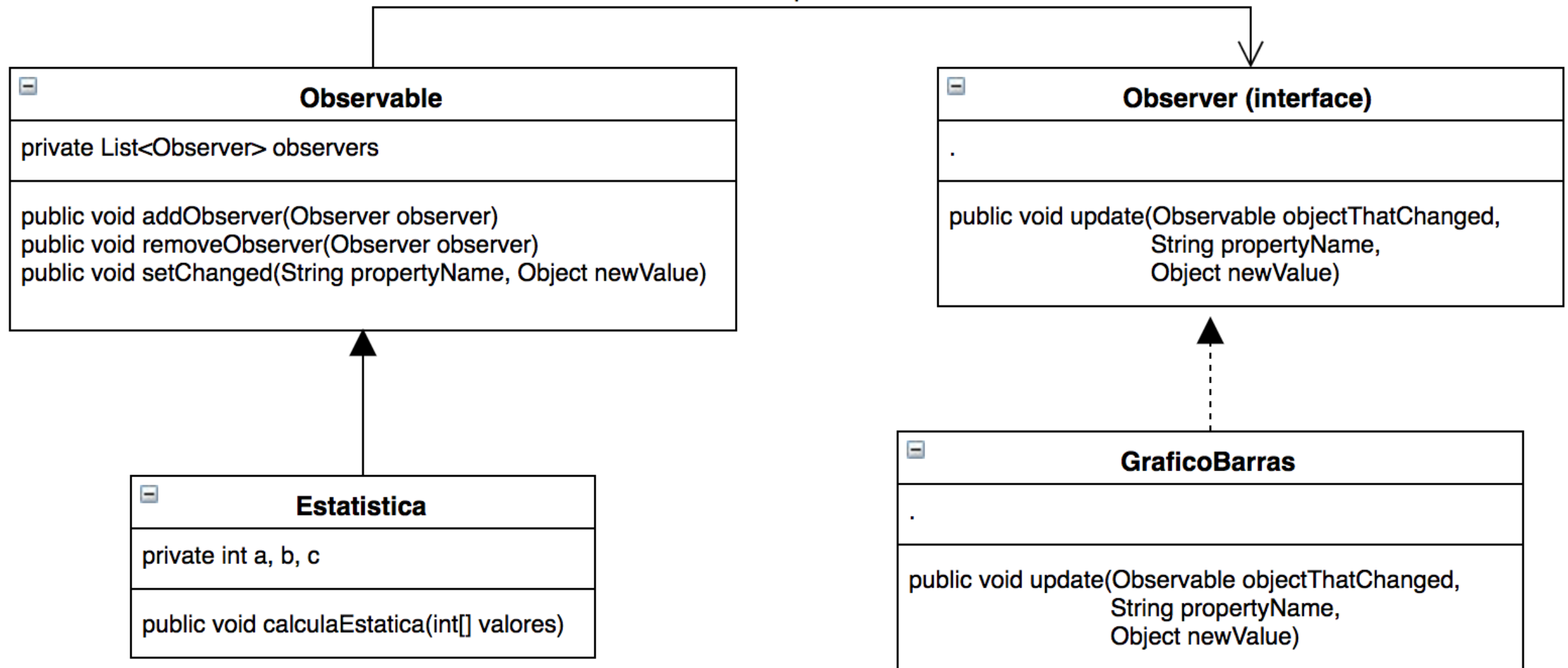
```
public static void main(String[] args) {  
  
    Estatistica estatica = new Estatistica();  
    GraficoBarras graficoBarras = new GraficoBarras();  
  
    // registo o grafico de barras como observador da estatística  
    estatica.addObserver(graficoBarras);  
  
    // mudo os dados da estatística  
    estatica.calculaEstatistica(new int[] { 3, 7, 4, 5, 1, 1 });  
  
    // automaticamente o grafico de barras é avisado e pode-se redesenhar  
}
```

Observer-Observable

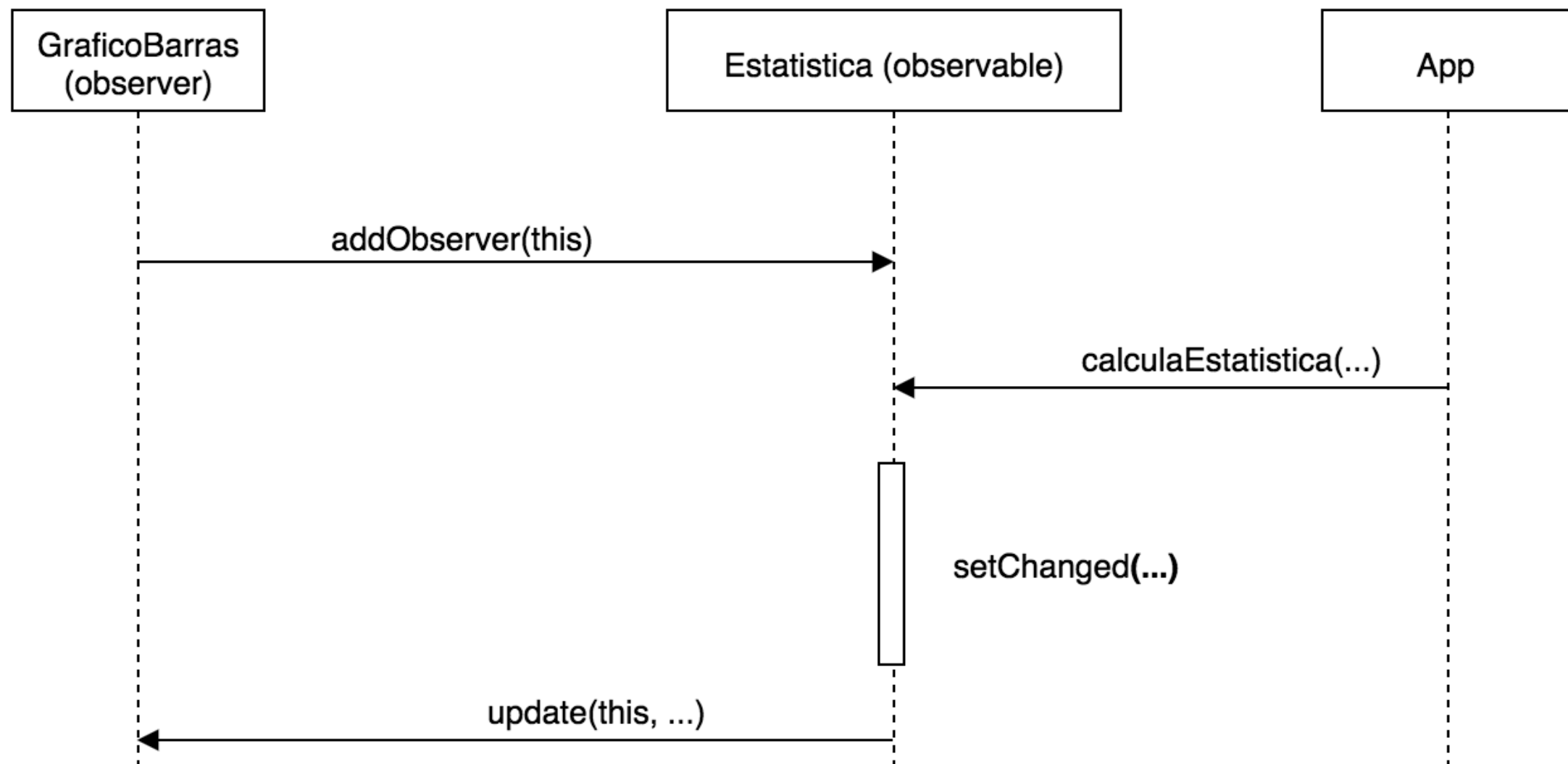
- Muito usado em aplicações gráficas
- Permite separar os objectos que guardam e processam dados dos objectos que lidam com a apresentação gráfica (no écran)
- Os objectos responsáveis pelos dados não sabem como é que estes vão ser apresentados
- Os objectos responsáveis pela apresentação não sabem processar os dados

Observer-Observable

é observado por

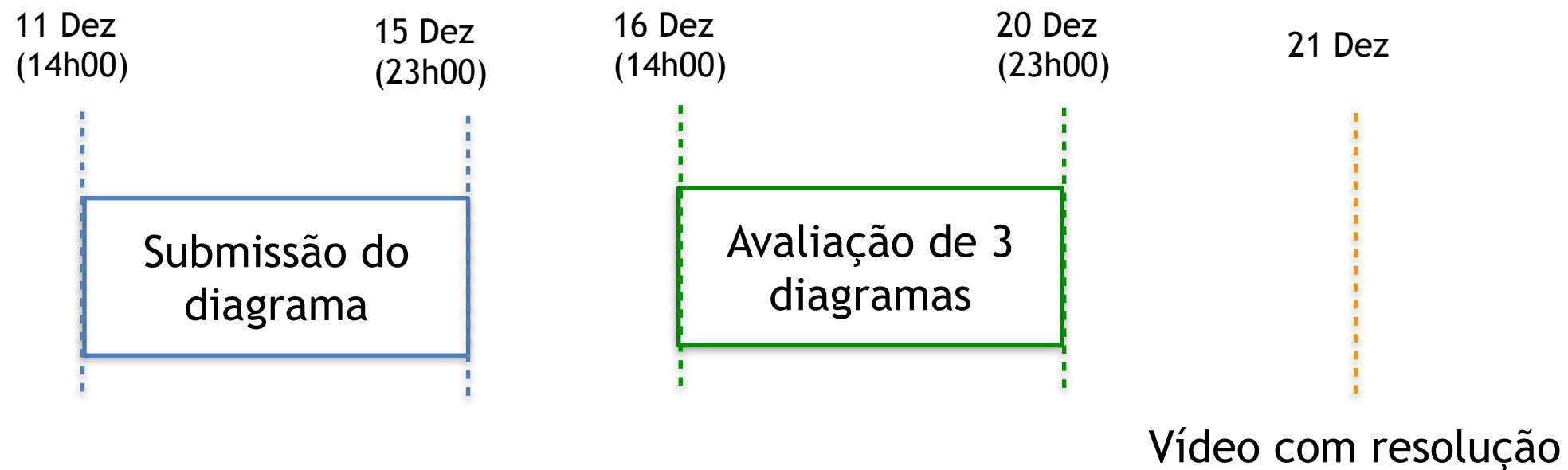


Observer-Observable



TPC teórico 4

Submissão de um diagrama de classes
e avaliação de 3 colegas



TPC teórico 4

Considere uma aplicação para gerir os salários dos funcionários de uma empresa.

Cada funcionário é caracterizado por um nome e por um número. O número é sequencial, atribuído automaticamente e é único entre todas as empresas.

O funcionário apenas recebe salário nos dias úteis. No entanto, dependendo do que acontece nesses dias, podem corresponder a dias de trabalho, dias de férias ou ausências. Nos dias de trabalho, o funcionário recebe uma remuneração que consiste no salário diário e no subsídio de almoço. Nos dias de férias, recebe apenas o salário diário. Nas ausências recebe um valor fixo. Para simplificar, assuma que a aplicação lida apenas com o ano atual e que contempla quer o passado quer o futuro (ex: pode-se marcar dias de férias para daqui a uns meses). Lembre-se que o funcionário pode começar a trabalhar na empresa a meio do ano e também pode sair a meio do ano.

Esta aplicação disponibilizará duas acções:

- Calcular a remuneração anual prevista de cada funcionário
- Calcular a remuneração total anual prevista de todos os funcionários da empresa

1. Desenhe o diagrama de classes que satisfaz todos estes requisitos.

Terá que obrigatoriamente identificar relações de herança e relações “normais”. Pelo menos uma classe deve ser abstrata. Lembre-se de respeitar as regras da cábula.

2. Implemente, em Java, os métodos correspondentes às 2 acções indicadas

Tire partido do polimorfismo e não use a instrução “if”