

# Padrões de Projeto

## Introdução



# Desenvolvendo com qualidade

Desenvolvimento de software é uma tarefa que exige criatividade e, em alguns casos, um pouco de arte. Mas para que o produto resultante seja de qualidade e, mais importante, para que o produto resultante seja passível de manutenção ao longo do tempo, é necessário o planejamento de seus componentes dentro de certos princípios.

O exercício que segue tenta mostrar a degradação de um produto de software à medida que sofre manutenções.

# Estrutura geral do exercício

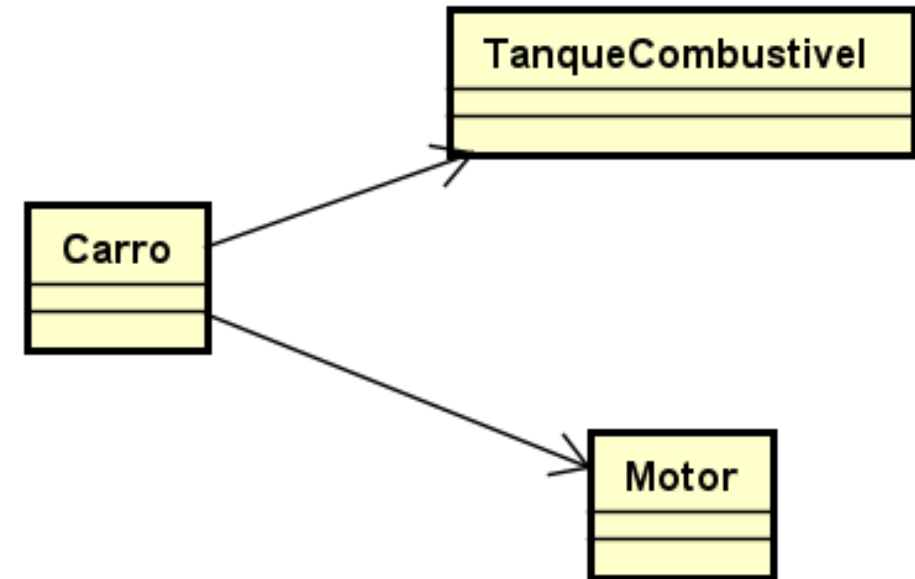
- O exercício apresenta um conjunto de classes que modelam um carro. Na sequência o exercício apresenta uma série de solicitações de manutenção. Para que o exercício alcance seu objetivo, é necessário que as solicitações sejam executadas na sequência, como se fosse solicitadas ao longo do tempo. Só assim seremos capazes de perceber a degradação do código.

# Analizando as classes originais

- O primeiro passo do exercício é analisar as classes originais e que irão sofrer diversas manutenções
- São 4 classes, a saber:
  - TipoCombustivel
  - Motor
  - TanqueCombustivel
  - Carro
- Você encontra o código destas classes no Moodle da disciplina.

# Diagrama de classes

- O diagrama de classes ao lado apresenta o relacionamento entre as 3 classes
- Abaixo um exemplo de criação de um “carro básico”



```
Carro basico = new Carro("Basico", TipoCombustivel.GASOLINA, 10, 55);
basico.abastece(TipoCombustivel.GASOLINA, 55);
basico.viaja(250);
System.out.println(basico);
```

# Execução do exercício

- O exercício demanda 5 solicitações diferentes
- Procure executar as 5 solicitações na ordem em que se apresentam, sem pular etapas ou bolar situações que antecipem a próxima solicitação.
- Execute o exercício como se cada solicitação em uma semana de trabalho diferente.

# Planejando a solução

- O exemplo apresentado demonstra a importância de planejarmos as soluções
- Só que nem sempre podemos prever o tipo de solicitação que será feita ao longo do tempo
- Então a saída é usar boas práticas e padrões conhecidos que buscam deixar o código mais fácil de manter

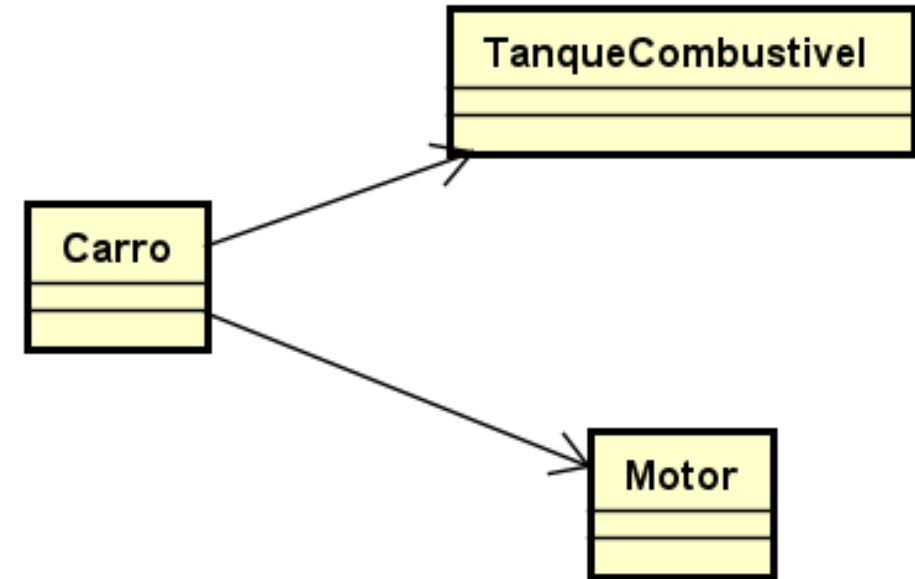
# Analizando a solução

- Não passe desse slide antes de ter tentado solucionar o exercício



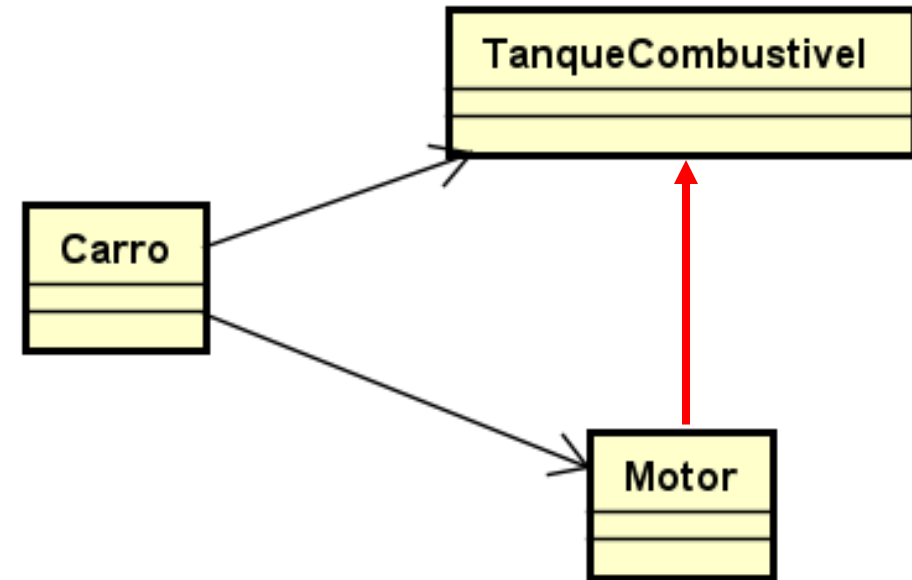
# Tornando as dependências explícitas

- Pelo diagrama de classes podemos ver que a classe *Carro* depende das classes *Motor* e *TanqueCombustivel* para funcionar.
- O padrão de injeção de dependências poderá nos ajudar muito nesta situação



# Cada classe deve ter uma responsabilidade

- Quem sabe que combustível foi usado no abastecimento é o tanque de combustível.
- Então o motor precisa conhecer o tanque de maneira a poder calcular quanto irá gastar para o carro percorrer uma certa distância (mais uma dependência).



# Refazendo a solução

- O principal aspecto a ser considerado neste problema em particular é que as classes tem dependências entre si
- O padrão de injeção de dependências diz que se uma classe depende de outras para funcionar, então essas dependências devem estar explícitas.
- Isso significa que a classe *Carro* deve receber pelo construtor ou por um método *set* referências para as classes das quais depende.

# Tornando as dependências de *Carro* explícitas

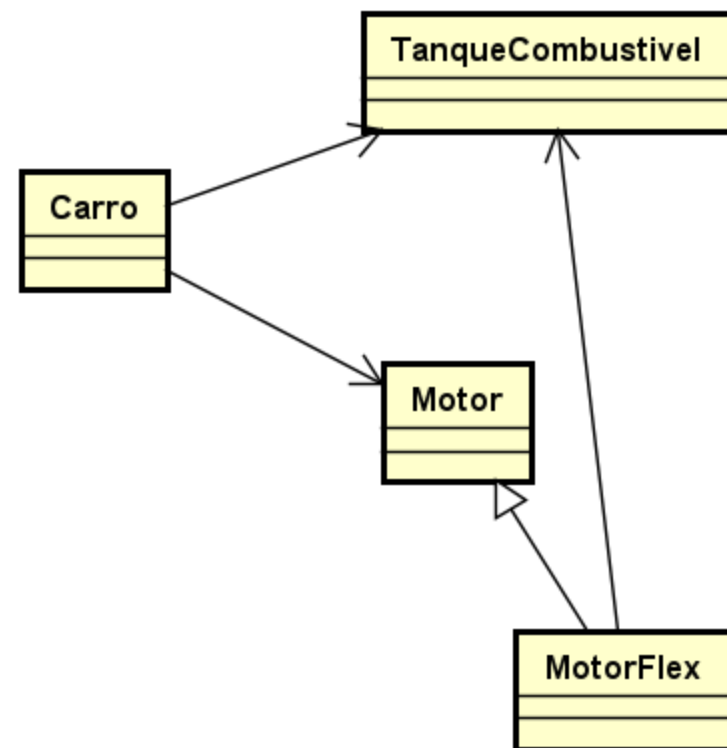
- No código ao lado, observe a aplicação do padrão de injeção de dependências na classe *Carro*.
- O fato dos diferentes papéis estarem explícitos, facilita até em pensarmos em subclasses para algumas das solicitações vistas anteriormente.
- Analise o código disponibilizado pelo professor com a solução.

```
public class Carro {  
    private String modelo;  
    private IMotor motor;  
    private ITanqueCombustivel tanqueCombustivel;  
  
    public Carro(String modelo,  
                  IMotor motor,  
                  ITanqueCombustivel tanqueCombustivel) {  
        this.modelo = modelo;  
        this.motor = motor;  
        this.tanqueCombustivel = tanqueCombustivel;  
    }  
}  
  
var t1 = new TanqueCombustivel('GASOLINA',45);  
var m1 = new Motor('GASOLINA',6);  
var c1 = new Carro('Basico',m1,t1);
```

# Discutindo a solução

- A figura ao lado apresenta o diagrama de classes da solução
- Abaixo exemplos de instanciamento de carros

```
ITanqueCombustivel t3 = new TanqueCombustivel('FLEX',55);  
IMotor m3 = new Motor('GASOLINA',8);  
Carro c3 = new Carro('SUV',m3,t3);  
c3.abastece('GASOLINA',40);  
c3.viaja(100);
```



# Introdução aos padrões de projeto

- Uma forma efetiva de evitar falhas durante o desenvolvimento de software é reusar código.
- Código existente, normalmente em bibliotecas, costuma estar exaustivamente testado, de maneira que a grande maioria dos bugs já foi detectada e consertada.
- Entretanto o reuso de código nem sempre é possível porque pode ser muito custoso adaptar o código existente as necessidades atuais.
- Um outro tipo de reuso possível é reusar ideias e conceitos ao invés de código.
- Padrões de projeto são uma forma de descrever boas práticas em POO
- Descrevem soluções confiáveis para problemas recorrentes.

# Definição

- **Padrões de projeto** são soluções reusáveis para problemas comuns que costumam ocorrer em certos contextos de projeto de software
  - São orientados à objetos e descrevem soluções em termos de objetos e classes
  - Descrevem a estrutura para a solução de um problema que deve ser adaptada para um caso e uma linguagem de programação particulares

# Princípios norteadores

- Separação de objetivos:
  - Cada abstração do programa (classe, método, interface etc) deve lidar com um único propósito ou objetivo e todos os aspectos relativos aquele objetivo devem ser tratados por aquela abstração
- Separação do “o que” do “como”:
  - Se um componente fornece um serviço específico, ele deve expor apenas à informação necessária para que possa ser usado. Detalhes de implementação não são de interesse de seus usuários



# Seguindo os princípios

- Se você segue os princípios seu código fica menos complexo e menos propenso à erros
- A complexidade aumenta a chance de cometermos erros e inserirmos defeitos em nossos programas
- Padrões de projeto foram desenvolvidos em diferentes áreas, mas os mais conhecidos ainda são os desenvolvidos pela “Gangue dos quatro” no livro de 1995 → Design Patterns: Elements of Reusable Object-Oriented Software escrito por Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides

# Classificação segundo catálogo GOF

- **Padrões de criação:** preocupam-se com a criação de classes e objetos. Definem maneiras de instanciar e inicializar objetos e classes que são mais abstratos que os mecanismos básicos definidos pelas linguagens de programação.
- **Padrões estruturais:** preocupam-se com a maneira como classes e objetos podem ser compostos. Descrevem como classes e objetos podem ser combinados para formar estruturas maiores.
- **Padrões comportamentais:** preocupam-se com a forma como os objetos e classes se comunicam. Definem como os objetos interagem trocando mensagens, as atividades em um processo e como estas são distribuídas entre os objetos participantes.

# Considerações gerais

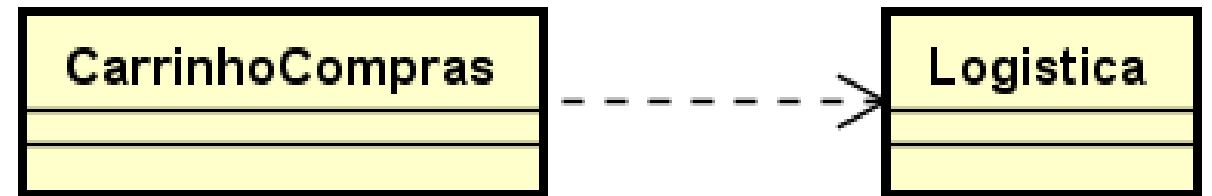
- Padrões de Projeto provêm um vocabulário comum para o entendimento e discussão de projetos de software
- Padrões de Projeto podem melhorar a flexibilidade ou performance do código
- Padrões de Projeto podem aumentar a complexidade do código
  - Neste caso, devem ser utilizados quando as vantagens do uso de um determinado padrão são maiores que as desvantagens
- Padrões de Projeto devem ser utilizados de modo a preservar o Princípio de Substituição

# Primeiros padrões

- Vamos começar apresentando um padrão e um princípio
  - Padrão de injeção de dependência (Dependency Injection)
  - Princípio da inversão de controle
- Depois iremos mostrar como framework Spring-Boot utiliza os conceitos apresentados

# Injeção de dependência

- Existem classes que dependem de outras para funcionar. Exemplo:
  - A classe *CarrinhoDeCompras* depende da classe “Logistica” para cálculo do frete dos produtos (veja a figura ao lado)
- Neste caso diz-se que a classe *CarrinhoDeCompras* depende da classe *Logistica* para funcionar. Então *CarrinhoDeCompras* é dependente de *Logistica*.
- De que maneira esta relação pode ser implementada?



# Implementando a dependência

```
class CarrinhoCompras{  
    private Logistica log;  
  
    public CarrinhoCompras(){  
        log = new Logistica();  
    }  
  
    ...  
}
```

A classe “CarrinhoCompras” cria uma instancia da classe “Logistica” no construtor.

```
class CarrinhoCompras{  
    private Logistica log;  
  
    public CarrinhoCompras(){  
        log=Logistica.getInstance();  
    }  
  
    ...  
}
```

A classe “CarrinhoCompras” acessa a instância única de “Logistica” através de um singleton (será visto mais adiante).

# Quais os problemas?

- Nas implementações apresentadas a dependência entre *CarrinhoDeCompras* e *Logistica* não é explícita.
- Isso pode acarretar uma série de problemas:
  - Dificulta o teste de *CarrinhoCompras*:
    - É preciso garantir a existência da classe “Logistica”
    - Fica difícil testar apenas *CarrinhoCompras*. Só se consegue testar o par *CarrinhoCompras* + *Logística*.
  - Se *Logistica* necessita de parâmetros de configuração para sua correta execução (por exemplo: definição do país ou região) pode ser que isso ainda não tenha sido feito (no caso do *Singleton*).
  - **Alterações em *Logistica* podem ter impacto em *CarrinhoCompras* e isso não está explícito**

# Solução

- Deixar as dependências explícitas:
  - Neste caso a dependência é explicitamente informada no construtor
  - Facilita o teste unitário: permite informar um dublê no construtor (será visto mais adiante)
  - Facilita a compreensão do que deve ser corretamente configurado
  - Reduz a probabilidade de alterações em *Logistica* terem impacto em *CarrinhoCompras*
  - É o primeiro passo em direção ao uso do padrão “inversão de controle” e aos diversos frameworks de injeção de dependência

```
class CarrinhoCompras{  
    private Logistica log;  
  
    public CarrinhoCompras(Logistica log){  
        this.log = log;  
    }  
  
    ...  
}
```





## Dinâmica

Analise o código da classe candidato ao lado. Aplique o padrão de injeção de dependência de maneira que possamos ter diferentes critérios de aceitação dos candidatos. Ex: nascidos em qualquer país do Mercosul antes de 2005; nome com pelo menos duas palavras, nascidos antes de 2001 no Brasil etc

```
class Candidato{
    private String nome;
    private int anoNascimento;
    private String paisNascimento;

    public Candidato(String nome,int anoNasc,String paisNasc){
        if (!valida(nome,anoNasc,paisNasc)){
            throw new RuntimeException("Dados nao validaram!");
        }
        this.nome = nome;
        this.anoNascimento = anoNasc;
        this.paisNascimento = paisNasc;
    }

    public boolean valida(String nome,int anoNasc,String paisNasc){
        if (nome.length() <= 3){ return false; }
        if (anoNasc >= 2005){ return false; }
        if (!paisNasc.toUpperCase().equals("BRASIL")){ return false; }
        return true;
    }
    ...
}
```

# Revisando o exemplo da biblioteca

- Anteriormente apresentamos um exemplo de “backend” de um sistema de biblioteca.
- Neste todas as requisições eram atendidas pela própria classe “controller” (*AppController*)
- Isso significa que a classe *AppController* tem mais de uma responsabilidade:
  - Monitorar e responder as requisições HTTP
  - Manter os dados sobre os livros
  - Executar consultas sobre os livros
- A classe *AppController*, nitidamente, viola o princípio que diz “Cada abstração do programa deve lidar com um único propósito ou objetivo “

# Relembre o código

- Observe que neste exemplo o controller mantém os dados e faz as consultas conforme a demanda dos endpoints.

```
@RestController
@RequestMapping("/biblioteca")
public class DemoController{
    private List<Livro> livros;

    public DemoController(){
        livros = new LinkedList<>();

        livros.add(new Livro(10,"Introdução ao Java","Huguinho Pato",2022));
        livros.add(new Livro(20,"Introdução ao Spring-Boot","Zezinho Pato",2020));
        livros.add(new Livro(15,"Principios SOLID","Luizinho Pato",2023));
        livros.add(new Livro(17,"Padroes de Projeto","Lala Pato",2019));
    }

    @GetMapping("/")
    public String getSaudacao() { return "Bem vindo as biblioteca central!"; }

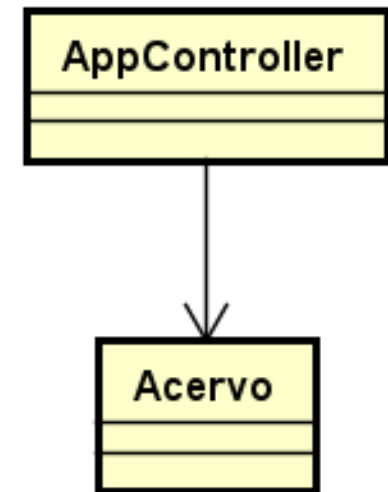
    @GetMapping("/livros")
    public List<Livro> getLivros() { return livros; }

    @GetMapping("/titulos")
    public List<String> getTitulos() {
        return livros.stream()
            .map(livro->livro.titulo())
            .toList();
    }
}
```

...

# Resolvendo o problema

- Para resolver o problema da violação do princípio do objetivo único quebramos a classe `DemoController` em duas:
  - A classe `DemoController` fica responsável apenas por tratar as requisições HTTP e ativar os métodos da classe *Acervo*.
  - A classe *Acervo* cuida das operações relativas ao acervo (coleção) de livros da biblioteca
- Em resumo a classe *DemoController* passa a depender da classe *Acervo* para funcionar.



# Como ficaria o código?

- Observe que *AppController* esta usando o padrão de injeção de dependências:
  - *AppController* depende de *Acervo* para funcionar
  - *AppController* recebe uma instancia de *Acervo* por injeção de dependência através do método construtor
  - No “módulo principal” todas as instancias são criadas e as dependências corretamente informadas

```
class DemoController{  
    private Acervo acervo
```

```
    constructor(acervo){  
        this.acervo = acervo;  
    }
```

```
    ...  
}
```

```
// Módulo principal
```

```
Acervo acervo = new Acervo();
```

```
DemoController controller =  
    new AppController(acervo);
```

# Qual a grande questão?

- Quando usamos o framework Spring-Boot não temos acesso ao “módulo principal”. Dessa forma não temos como instanciar a classe *DemoController* e nem fazer a injeção da dependência.
- Ao iniciar a execução do programa (ver código ao lado) a função *bootstrap* cria uma instancia da classe principal e passa a monitorar a porta designada. A partir disso ativa os “endpoints” correspondentes conforme a requisição recebida
- Em resumo, o Spring-Boot assume o controle da aplicação: **inversão de controle**

```
@SpringBootApplication
public class Endpointsdemo3Application {
    public static void main(String[] args) {
        SpringApplication.run(Endpointsdemo3Application.class, args);
    }
}
```

# O princípio da inversão de controle

- O princípio da inversão de Controle está relacionado com o fato do “framework” ou outros mecanismos que estivermos usando assumirem o controle do fluxo da aplicação.
- No caso o Spring assume o comando desde o início para poder monitorar as requisições que chegam na porta designada. Isso faz, porém, que ele tenha de assumir também a responsabilidade pela criação das instâncias e pelas injeções de dependências
- Então as classes que devem ser “injetadas” são indicadas para um componente “injetor” que se encarrega de fazer a “injeção” propriamente dita.

# Indicando o ponto de injeção no Spring-Boot

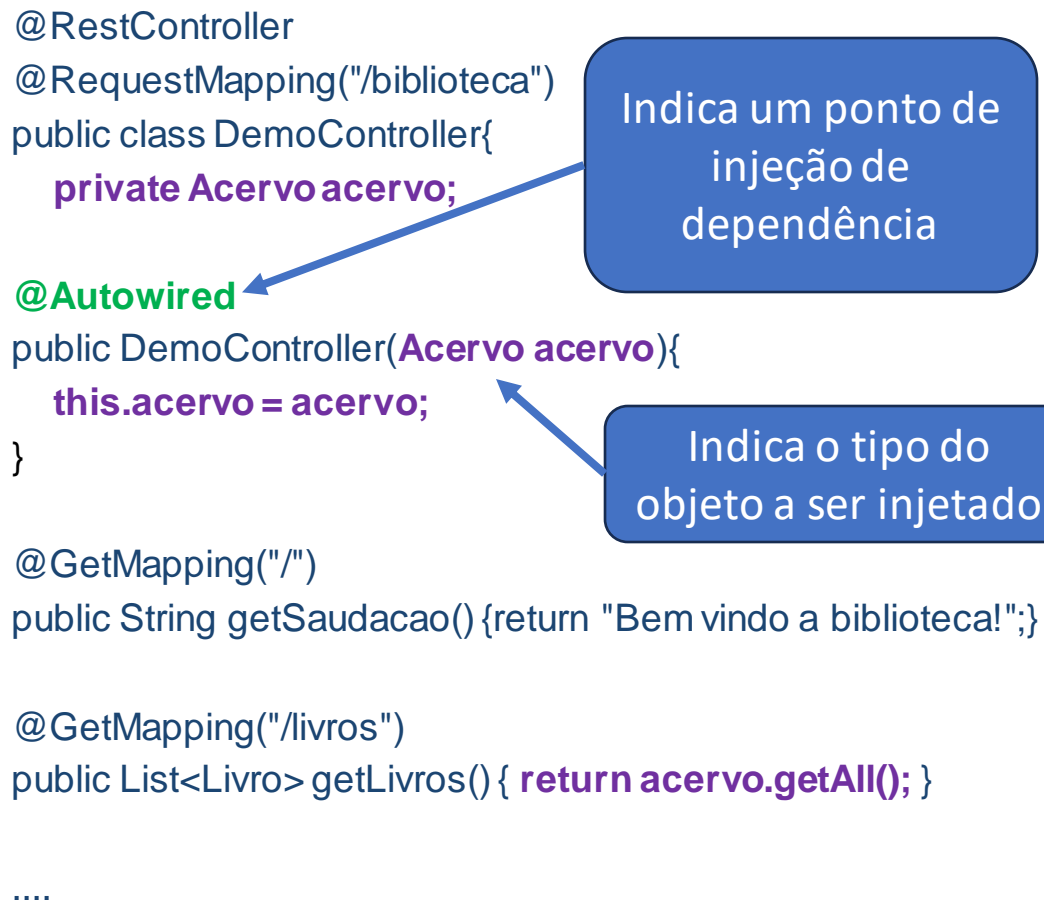
```
@RestController
@RequestMapping("/biblioteca")
public class DemoController{
    private Acervo acervo;

    @Autowired
    public DemoController(Acervo acervo){
        this.acervo = acervo;
    }

    @GetMapping("/")
    public String getSaudacao() {return "Bem vindo a biblioteca!";}

    @GetMapping("/livros")
    public List<Livro> getLivros() { return acervo.getAll(); }

    ....
```



```
@Component
public class Acervo {
    private List<Livro> livros;

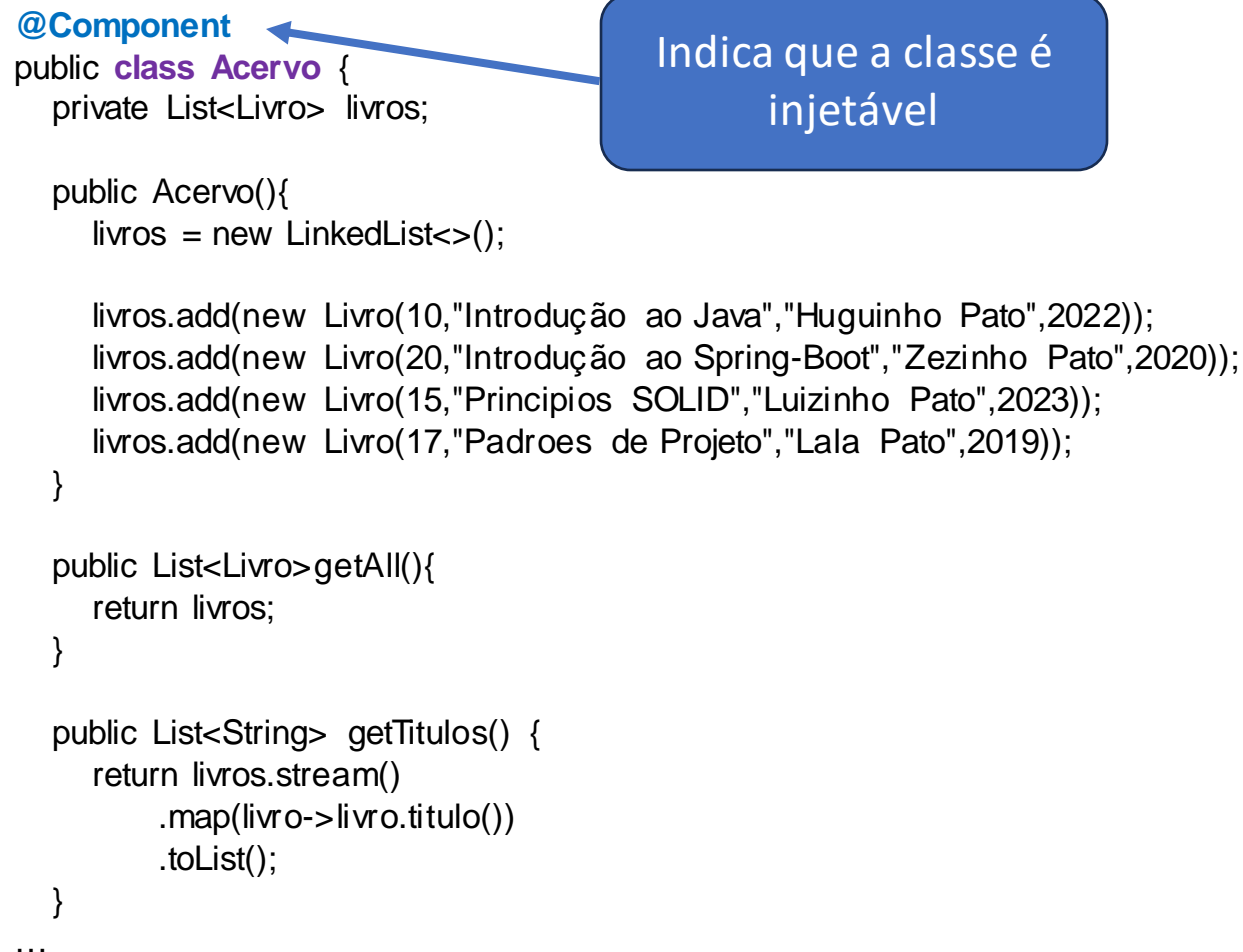
    public Acervo(){
        livros = new LinkedList<>();

        livros.add(new Livro(10,"Introdução ao Java","Huguinho Pato",2022));
        livros.add(new Livro(20,"Introdução ao Spring-Boot","Zezinho Pato",2020));
        livros.add(new Livro(15,"Principios SOLID","Luizinho Pato",2023));
        livros.add(new Livro(17,"Padroes de Projeto","Lala Pato",2019));
    }

    public List<Livro> getAll(){
        return livros;
    }

    public List<String> getTitulos() {
        return livros.stream()
            .map(livro->livro.titulo())
            .toList();
    }

    ...
```







## Dinâmica

1) Recupere o código do sistema de backend da biblioteca e faça as modificações de maneira a separar o controller do acervo conforme demonstrado nos slides anteriores.

2) Crie um serviço de coleta de estatísticas sobre o acervo que tenha funções para:

1. Contar quantas são as obras de determinado autor
2. Contar quantas são as obras mais recentes que determinado ano
3. Calcula o número médio de obras por autor

Crie o serviço e acrescente endpoints que demandem estes serviços. Experimente acrescentar esses novos endpoints em um controller separado.

# Padrão *Repository*

Um repositório executa as tarefas de um intermediário entre lógica da aplicação e a persistência dos dados, funcionando de maneira semelhante a um conjunto de objetos na memória. Os objetos de clientes criam consultas de forma declarativa e enviam-nas para os repositórios buscando respostas. Conceitualmente, um repositório encapsula um conjunto de objetos armazenados em um banco de dados – ou outro tipo de mecanismo de persistência – e as operações que podem ser executadas sobre eles.

# Padrão *Repository*: contexto

- Um sistema necessita trabalhar com coleções de entidades (objetos de domínio) que estão armazenadas em algum tipo de mecanismo de persistência
- Deseja-se manter o acoplamento baixo, mantendo os objetos de domínio ignorantes em relação aos mecanismos de persistência utilizados

# Padrão *Repository*: solução

- Definir uma interface que abstrai as operações básicas de acesso a dados (adicionar, atualizar, remover, consultar) com operações semelhantes a de uma coleção em memória (lista, dicionário etc)
- Implementar a interface definida usando a tecnologia mais adequada
- No modelo mais simples usar um “Repository” para cada objeto de domínio

# Padrão Repository: exemplo

```
public interface IAcervoRepository {  
    List<Livro> getAll();  
    Livro getPorId(int id);  
    List<Livro> getAutor(String autor);  
    List<Livro> getTitulo(String titulo);  
    List<Livro> getAno(int ano);  
    boolean cadastraLivroNovo(Livro livro);  
    boolean removeLivro(int id);  
}
```

- A definição da interface permite que se criem diferentes implementações para o repositório conforme a necessidade.
- A versão ao lado mantém o cadastro em memória, porém, outra versão pode manter o cadastro em um banco de dados.
- Construindo a aplicação ao redor da interface a alteração da tecnologia de persistência não traz impacto sobre o restante da aplicação.

```
@Component  
public class AcervoMemorialImpl implements IAcervoRepository {  
    private List<Livro> livros;  
  
    public AcervoMemorialImpl(){  
        livros = new LinkedList<>();  
  
        livros.add(new Livro(10,"Introdução ao Java","Huguinho Pato",2022));  
        livros.add(new Livro(20,"Introdução ao Spring-Boot","Zezinho Pato",2020));  
        livros.add(new Livro(15,"Principios SOLID","Luizinho Pato",2023));  
        livros.add(new Livro(17,"Padroes de Projeto","Lala Pato",2019));  
    }  
  
    @Override  
    public List<Livro>getAll(){ return livros; }  
  
    ...  
  
    @Override  
    public boolean cadastraLivroNovo(Livro livro) {  
        livros.add(livro);  
        return true;  
    }  
  
    @Override  
    public boolean removeLivro(int codigo) {  
        List<Livro> tmp = livros.stream()  
            .filter(livro->livro.codigo() == codigo)  
            .toList();  
        return tmp.removeAll(tmp);  
    }  
}
```

# Acessando o banco de dados usando JDBC no Spring-Boot

- JDBC é uma biblioteca de classes e interfaces escritas na linguagem Java que possibilitam se conectar, através de um driver específico, com o SGBD desejado. Através de uma conexão JDBC pode-se executar instruções SQL de qualquer tipo de banco de dados relacional.

# Configurando as dependências

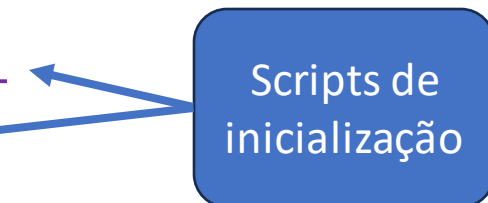
- Para utilizar a biblioteca JDBC no Spring-boot é necessário configurar dois tipos de dependências no arquivo “pom.xml”:
  - A biblioteca JDBC
  - O gerenciador de banco de dados: neste exemplo iremos usar o H2, SGBD que possui opção de rodar em memória de forma “embeded”. Muito usado em protótipos e testes.
- A figura ao lado apresenta estas dependências

```
<dependency>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-starter-jdbc</artifactId>  
</dependency>
```

```
<dependency>  
  <groupId>com.h2database</groupId>  
  <artifactId>h2</artifactId>  
  <scope>runtime</scope>  
</dependency>
```

# Arquivos de inicialização

- Para inicializar o banco de dados vamos precisar de um script de inicialização e outro para inserir registros iniciais no banco (se for o caso). Além disso precisamos de um arquivo de propriedades que descreve onde estão os demais.
- Todos estes arquivos devem ficar localizados na pasta “../main/resources”
- Exemplo de conteúdo do arquivo “application.properties”:
  - `spring.sql.init.mode=always`
  - `spring.sql.init.schema-locations=classpath*:./create.sql`
  - `spring.sql.init.data-locations=classpath*:./insert.sql`
  - `spring.sql.init.encoding=UTF-8`



Scripts de  
inicialização



# Scripts de inicialização

- Arquivo “create.sql”:

```
DROP TABLE livros IF EXISTS;  
CREATE TABLE livros (codigo long,  
                        titulo VARCHAR(255),  
                        autor VARCHAR(255),  
                        ano int,  
                        PRIMARY KEY(codigo)  
);
```

- Exemplo de arquivo “insert.sql”:

```
INSERT INTO livros (codigo,titulo,autor,ano) VALUES  
(10,'Introdução ao Java','Huguinho Pato',2022);  
INSERT INTO livros (codigo,titulo,autor,ano) VALUES  
(20,'Introdução ao Spring-Boot','Zezinho Pato',2020);  
INSERT INTO livros (codigo,titulo,autor,ano) VALUES  
(15,'Principios SOLID','Luizinho Pato',2023);  
INSERT INTO livros (codigo,titulo,autor,ano) VALUES  
(17,'Padroes de Projeto','Lala Pato',2019);
```

# JDBC: exemplo

- Ao lado o código parcial da implementação do “Repository” usando JDBC.
- A anotação “@Primary” indica que esta implementação é a que deve ser prioritariamente em caso da interface *IAcervoRepository* ser usada em um ponto de injeção de dependência
- Note que a classe recebe uma instancia de *JdbcTemplate* por injeção de dependência. Esta é inicializada pelo framework de maneira a conectar no banco de dados.

@Component

@Primary

```
public class AcervoJDBCImpl implements IAcervoRepository {  
    private JdbcTemplate jdbcTemplate;
```

@Autowired

```
public AcervoJDBCImpl(JdbcTemplate jdbcTemplate) { this.jdbcTemplate = jdbcTemplate; }
```

@Override

```
public List<Livro> getAll() {  
    List<Livro> resp = this.jdbcTemplate.query("SELECT * from livros",  
        (rs, rowNum) ->  
            new Livro(rs.getLong("codigo"),rs.getString("titulo"),rs.getString("autor"), rs.getInt("ano")));  
    return resp;  
}
```

@Override

```
public boolean removeLivro(long codigo){  
    String sql = "DELETE FROM livros WHERE id = "+codigo;  
    this.jdbcTemplate.batchUpdate(sql);  
    return true;  
}
```

@Override

```
public boolean cadastraLivroNovo(Livro livro){  
    this.jdbcTemplate.update(  
        "INSERT INTO livros(codigo,titulo,autor,ano) VALUES (?, ?, ?, ?)",  
        livro.codigo(),livro.titulo(),livro.autor(),livro.ano());  
    return true;  
}
```

...



## Dinâmica

3) Recupere o código dos exercícios 1 e 2 e transforme a classe *Acervo* em um repositório (use o padrão *Repository*). Crie duas versões: em memória e usando JDBC. Pesquise os recursos da biblioteca JDBC para implementar todas as operações previstas para este repositório. Garanta também que o repositório implemente todas operações CRUD para livros além das consultas já previstas. Por fim pesquise a utilidade da anotação `@Primary`.

