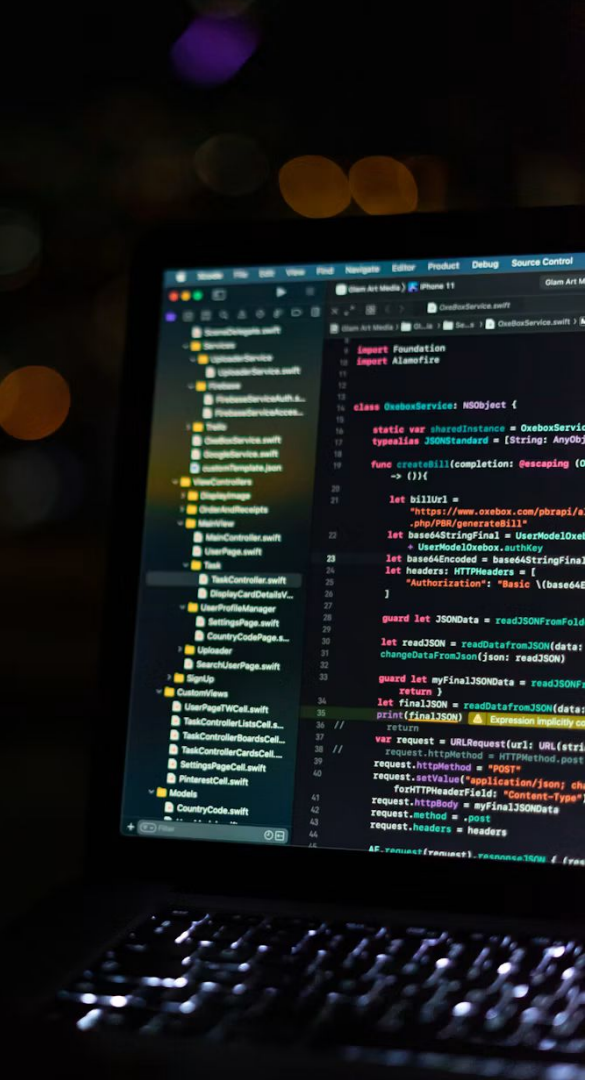


5954018 - Programação Orientada a Objetos

Aula - Princípios SOLID

Prof. Dr. Denis Martins

DCM | FFCLRP | USP

Cenário de Motivação

Como pequenas mudanças impactam o projeto de software

Cenário

Uma universidade implementou um sistema para que estudantes e servidores solicitem reservas de salas e recebam uma confirmação.

Como funciona?

1. O solicitante informa **sala, horário de início, horário de término e contato**.
2. O sistema valida os dados e verifica a **política de horário da sala**.
3. Se a solicitação for permitida, registra a reserva, gera um código e envia a confirmação.

Modelo de Dados (Reserva.java)

```
class Reserva {
 private String sala;
 private int inicio, fim;
 private String contato;
 Reserva(String s, int i, int f, String c) {
  this.sala = s;
  this.inicio = i;
  this.fim = f;
  this.contato = c;
 }
}
```

Instância:

```
new Reserva("A101", 10, 12, "aluna@...")
```

Regras de Negócio e Políticas de Horário

Dados Obrigatórios: Sala e contato devem estar devidamente preenchidos.

Restrição: Reservas nos limites exatos da política são permitidas.

Sala Comum: Permitido entre 8h e 18h.

Implementação Atual

ReservaService.java

```
class ReservaService {
  private Map<String, Reserva> dados = new HashMap<>();
  private int sequencia = 0;

  String reservar(Reserva r) {
    // 1. Validação de Dados
    if (r.sala == null || r.inicio >= r.fim ...)
      throw new IllegalArgumentException( ... );

    // 2. Regra de Negócio (Horário)
    if (r.inicio < 8 || r.fim > 18)
      throw new IllegalStateException( ... );

    // 3. Persistência
    String codigo = "RES-" + (++sequencia);
    dados.put(codigo, r);

    // 4. Notificação
    System.out.println("EMAIL " + r.contato ... );
    return codigo;
  }
}
```

Evolução

A universidade quer acrescentar **horários de laboratório**, consultar reservas pelo portal e trocar o **canal de confirmação**.

Nova restrição:
Laboratório: Permitido entre 10h e 16h (ex: 10h às 16h é válido).

Pergunta: Como modificar o código para acomodar essas e outras possíveis mudanças?

Implementação Atual

ReservaService.java

```
class ReservaService {
  private Map<String, Reserva> dados = new HashMap<>();
  private int sequencia = 0;

  String reservar(Reserva r) {
    // 1. Validação de Dados
    if (r.sala == null || r.inicio >= r.fim ...)
      throw new IllegalArgumentException( ... );

    // 2. Regra de Negócio (Horário)
    if (r.inicio < 8 || r.fim > 18)
      throw new IllegalStateException( ... );

    // 3. Persistência
    String codigo = "RES-" + (++sequencia);
    dados.put(codigo, r);

    // 4. Notificação
    System.out.println("EMAIL " + r.contato ... );
    return codigo;
  }
}
```

Perguntas de Motivação

1. Quais decisões estão no ReservaService?

A classe acumula validação, política de horários, geração de IDs, persistência em memória e envio de e-mail.

2. Onde alterar para atender um laboratório?

Diretamente nos comandos if do método reservar, correndo o risco de quebrar regras existentes para salas comuns.

3. Por que editar a classe para trocar o canal?

Porque o envio de e-mail está acoplado de forma rígida (System.out.println) junto com a lógica central de reserva.

Implementação Atual

ReservaService.java

```
class ReservaService {
  private Map<String, Reserva> dados = new HashMap<>();
  private int sequencia = 0;

  String reservar(Reserva r) {
    // 1. Validação de Dados
    if (r.sala == null || r.inicio >= r.fim ...)
      throw new IllegalArgumentException( ... );

    // 2. Regra de Negócio (Horário)
    if (r.inicio < 8 || r.fim > 18)
      throw new IllegalStateException( ... );

    // 3. Persistência
    String codigo = "RES-" + (++sequencia);
    dados.put(codigo, r);

    // 4. Notificação
    System.out.println("EMAIL " + r.contato ... );
    return codigo;
  }
}
```

Perguntas de Motivação

1. Quais decisões estão no ReservaService?

A classe acumula validação, política de horários, geração de IDs, persistência em memória e envio de e-mail.

2. Onde alterar para atender um laboratório?

Diretamente nos comandos if do método reservar, correndo o risco de quebrar regras existentes para salas comuns.

3. Por que editar a classe para trocar o canal?

Porque o envio de e-mail está acoplado de forma rígida (System.out.println) junto com a lógica central de reserva.

Fundamentos: Coesão, Acoplamento e Refatoração

Conceito	Definição	No sistema de reservas
Coesão	Relação entre os elementos de um módulo	Formatar a confirmação reúne decisões de apresentação
Acoplamento	Interdependência entre módulos	O serviço conhece diretamente o canal de e-mail
Refatoração	Reorganização interna que preserva o comportamento previsto	Extrair a validação sem mudar as entradas aceitas

Como proceder: Localizar mudanças e preservar contratos.

Importante: Extrair validação preserva comportamento (refatoração). Acrescentar política cria comportamento novo.

O modelo e o fluxo que vamos evoluir

Modelo & Regras de Validação

Dados da solicitação:

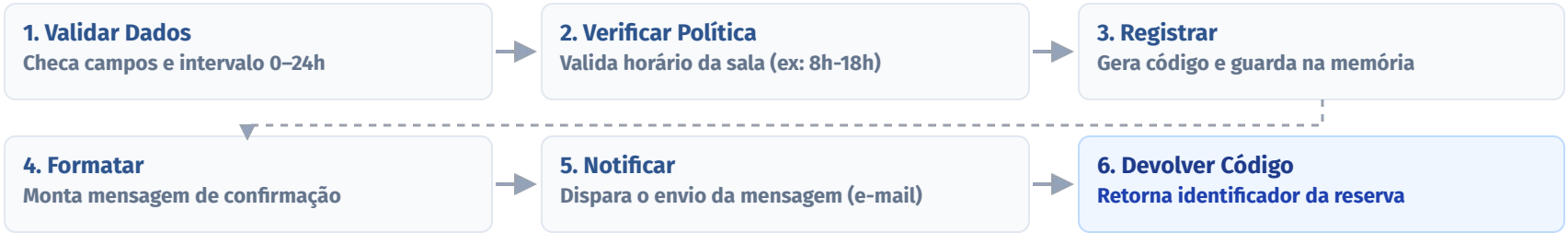
```
class Reserva {
  final String sala, contato;
  final int inicio, fim;
}
```

Dados válidos (Estrutural)

- sala e contato preenchidos
- 0 <= inicio < fim <= 24

Tipo de Sala	Horário Permitido
Comum	**8h – 18h**
Laboratório	**10h – 16h**

Fluxo do Processo de Reserva (6 Etapas)



S Single Responsibility Principle: SRP

Princípio da Responsabilidade Única

"Uma classe deve reunir uma responsabilidade coesa, associada a um **motivo principal para mudar**."

Mudança na Classe Inicial	Responsabilidade Envolvida
Novas exigências para os dados	Validação
Outra forma de guardar reservas	Armazenamento
Outro canal de confirmação	Comunicação

Pergunta: Essas decisões precisam ficar na mesma classe?

Uma classe pode possuir vários métodos coesos, e a coordenação do fluxo também é uma responsabilidade válida. A análise do SRP considera as razões de mudança e a separação de papéis, não a simples contagem de linhas de código ou métodos.

Evolução 1: Extrair Responsabilidades

Extrair da versão inicial as classes de validação, repositório e notificação. A regra de horário permanece no serviço.

```
class ValidadorReserva {
  void validar(Reserva r) {
    if (r.sala == null || r.sala.trim().isEmpty() ||
     r.contato == null || r.contato.trim().isEmpty() ||
     r.inicio < 0 || r.inicio >= r.fim || r.fim > 24) {
     throw new IllegalArgumentException( ... );
    }
  }
}

class RepositorioMemoria {
  private Map<String, Reserva> dados = ... ;
  private int sequencia = 0;

  public String salvar(Reserva r) {
    String codigo = "RES-" + (++sequencia);
    dados.put(codigo, r);
    return codigo;
  }
}

class EmailNotificador {
  public void enviar(String destino, String msg) {
   System.out.println("EMAIL " + destino + ": " + msg);
  }
}
```

1. ValidadorReserva

- Isolamento total da validação estrutural dos dados.
- Lança excepção caso os campos obrigatórios estejam inválidos.

2. RepositorioMemoria

- Centraliza a geração do código de reserva e persistência em memória.
- Encapsula o mapa interno e o controle da sequência de IDs.

3. EmailNotificador

- Separa a infraestrutura e o mecanismo de comunicação externa.
- Retira do serviço principal a responsabilidade de formatação/envio.

Como Fica o ReservaService?

O serviço agora coordena os colaboradores extraídos, mantendo a regra de horário localmente.

```
String reservar(Reserva r) {
  validador.validar(r);

  if (r.inicio < 8 || r.fim > 18) {
    throw new IllegalStateException(
     "Horário não permitido"
    );
  }

  String codigo = repositorio.salvar(r);

  notificador.enviar(
    r.contato,
    "Reserva " + codigo
  );

  return codigo;
}
```

ValidadorReserva

- Chamada delegada: `validador.validar(r)`
- Isola a verificação de campos obrigatórios.

RepositorioMemoria

- Chamada delegada: `repositorio.salvar(r)`
- Registra os dados e devolve o código gerado.

EmailNotificador

- Chamada delegada: `notificador.enviar( ... )`
- Simula o envio externo da notificação.

Regra de Horário (Local)

- Permanece no serviço até a **próxima etapa**.

Exercício 1: A confirmação ganha um formato

Em duplas

```
String mensagem = "Reserva " + codigo + " | Sala: " + r.sala
                + " | Horário: " + r.inicio + "-" + r.fim;
notificador.enviar(r.contato, mensagem);
```

Tarefas a realizar:

1. Criar `FormatadorConfirmacao` com a assinatura: `String formatar(String codigo, Reserva r){}`
2. Substituir a montagem da mensagem por uma chamada ao formatador.
3. Identificar o motivo de mudança retirado do serviço.

Exemplo de mensagem esperada: `"Reserva RES-1 | Sala: A101 | Horário: 10-12"`

O Open/Closed Principle: OCP

Princípio Aberto/Fechado

"Uma unidade de software deve permitir extensão de comportamento e proteger sua lógica estável contra modificações recorrentes."

Parte Estável

- Coordenação da solicitação.
- Estrutura principal do fluxo de execução mantida intacta sem alterações a cada nova regra.

Parte Variável

- Decisão sobre o horário permitido.
- Nova demanda: laboratório permite reservas entre 10h e 16h.

Pergunta: Como evitar um novo condicional no serviço a cada política?

Evolução 2: Uma Política Substitui a Condição Fixa

O serviço pergunta à política se a reserva é permitida, eliminando condicionais diretas.

```
interface PoliticaReserva {
  boolean permite(Reserva r);
}

class PoliticaSalaComum
  implements PoliticaReserva {
  public boolean permite(Reserva r) {
    return r.inicio ⩾ 8 && r.fim ⩽ 18;
  }
}

class PoliticaLaboratorio
  implements PoliticaReserva {
  public boolean permite(Reserva r) {
    return r.inicio ⩾ 10 && r.fim ⩽ 16;
  }
}
```

```
// Alterações em ReservaService:
private PoliticaReserva politica;

ReservaService(PoliticaReserva politica){
  this.politica = politica;
}

if (!politica.permite(r)) {
  throw new IllegalStateException(
       "Horário não permitido");
}
```

Seleção da Política (Injeção)

O serviço passa a receber a abstração via construtor:

```
new ReservaService(
  new PoliticaLaboratorio()
);
```

Exercício 2: Política para a Sala de Estudos

Objetivo: Estender o sistema sem modificar a coordenação principal.

Em duplas

Nova Política de Reserva:
Sala de estudos permite reservas entre **6h** e **22h**.

```
class PoliticaSalaEstudos
  implements PoliticaReserva {

  public boolean permite(Reserva r) {

  // Completar lógica
  // (verificar horário 6h-22h)

  }

}
```

1. Implementar a classe `PoliticaSalaEstudos`.
2. Selecioná-la ao montar o serviço (Injeção).
3. Prever a decisão para uma reserva de 7h–9h.
4. Explicar qual parte do serviço permanece igual.

L Liskov Substitution Principle: LSP

Princípio da Substituição de Liskov

“Uma implementação deve poder substituir outra preservando as propriedades prometidas pelo contrato aos clientes.”

Contrato de `PoliticaReserva.permite`:

- **Entrada:** reserva **não nula** e **aprovada** pelo validador.
- **Saída:** `true` se **permite**, `false` se **recusa**.
- **Para toda entrada válida: sem exceção**.
- **Efeitos colaterais: sem registro** de reserva ou **envio** de mensagem.

Definição: *O contrato especifica* ***condições de uso*** *e* ***garantias da operação****.*

Importante: Assinaturas compatíveis não garantem comportamento compatível.

Evolução 3: A Recusa Precisa Preservar o Contrato

✕ Quebra o contrato (Violação LSP)

```
public boolean permite(Reserva r) {
  if (r.inicio < 10 || r.fim > 16) {
    throw new IllegalArgumentException(
      "Laboratório fechado");
  }
  return true;
}
```

✓ Preserva o contrato (Correto)

```
public boolean permite(Reserva r) {
  return r.inicio ⩾ 10 &&
         r.fim ⩽ 16;
}
```

Entrada de teste (Solicitação válida): Reserva das 9h–11h.
Resultado esperado da política: Retornar `false` (sem lançar exceções para entradas válidas).

Responsabilidade do Serviço: Ao receber `false` da política, o `ReservaService` deve recusar o caso de uso lançando `IllegalStateException`.

I Interface Segregation Principle: ISP

Princípio da Segregação de Interfaces

“Clientes não devem ser obrigados a depender de operações que não utilizam.”

O sistema ganha novos clientes:

Cliente	Operação necessária
Serviço de solicitação	`Registrar`
Portal do estudante	`Consultar`
Secretaria	`Cancelar`

Pergunta: Todos precisam depender de salvar, consultar e cancelar?

Evolução 4: Contratos Específicos de Armazenamento

✕ Contrato Amplo Proposto

```
interface GestaoReservas {
 String salvar(Reserva r);
 Reserva consultar(String cod);
 void cancelar(String cod);
}
```

✓ Contratos Específicos por Cliente

```
interface RegistroReserva {
 String salvar(Reserva r);
}
```

```
interface ConsultaReserva {
 Reserva consultar(String cod);
}
```

```
interface CancelamentoReserva {
 void cancelar(String cod);
}
```

Implementação compartilhada: `RepositorioMemoria` **implementa os três contratos.**

Importante: Um único objeto (ex: RepositorioMemoria) pode atender a múltiplos contratos sem forçar clientes a dependerem de métodos desnecessários.

Evolução 4: Uma Única Classe, Múltiplos Contratos

Implementação Concreta (RepositorioMemoria)

```
class RepositorioMemoria implements RegistroReserva,
      ConsultaReserva, CancelamentoReserva {
  private final Map<String, Reserva> dados = new HashMap<>();
  private int sequencia = 0;
  public String salvar(Reserva r) {
    String cod = "RES-" + (++sequencia);
    dados.put(cod, r); return cod;
  }
  public Reserva consultar(String cod) {
    return dados.get(cod);
  }
  public void cancelar(String cod) {
    dados.remove(cod);
  }
}
```

Por que importa?

Segregação Total: Cada cliente enxerga apenas o contrato que precisa.

- **Portal:** dependerá apenas de `ConsultaReserva`.
- **Serviço de Solicitação:** dependerá apenas de `RegistroReserva`.

Esclarecimentos Importantes

- Há 3 interfaces porque existem **3 clientes com necessidades distintas**.
- O ISP **não exige** que toda interface possua apenas um único método.
- A divisão de interfaces define contratos de uso, **não autorização/segurança**.

Exercício 3: O Portal Precisa Apenas Consultar

Objetivo: Escolher o contrato usado por um cliente. A dependência do portal deve expressar somente sua necessidade de consulta.

Em duplas

```
class PortalConsulta {
   private ______ reservas;
   PortalConsulta(______ reservas) {
      this.reservas = reservas;
   }
   String mostrarSala(String codigo) {
      Reserva r = reservas.______(codigo);
      return r == null ? "Reserva não encontrada" : r.sala;
   }
}
```

Tarefas

- Completar as três lacunas no código.
- Fornecer um repositório já utilizado para registro.
- Justificar o tipo escolhido.

D Dependency Inversion Principle: DIP

Princípio da Inversão de Dependência
"Módulos de alto nível e detalhes de baixo nível devem depender de abstrações. Os detalhes se adequam aos contratos da aplicação."

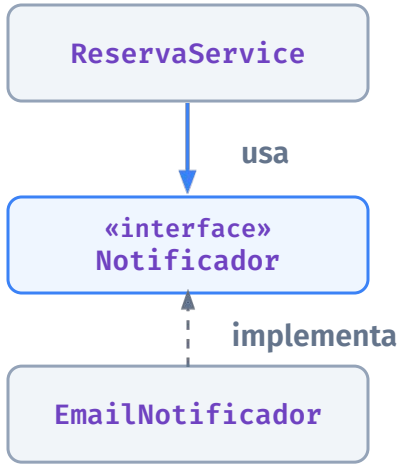
Estrutura no Sistema:

- **Alto nível:** Coordenação em `ReservaService`.
- **Detalhes variáveis:** `RepositorioMemoria` e `EmailNotificador`.
- **Contratos:** `RegistroReserva` e `Notificador`.

Injeção de Dependência:

Fornecer uma dependência ao objeto, por exemplo pelo construtor. Os serviços deixam de conhecer detalhes concretos.

Diagrama de Relações (DIP)



Evolução 5: Contratos no Serviço, Escolhas na Aplicação

Dependências Variáveis do Serviço

```
private PoliticaReserva politica;
private RegistroReserva repositorio;
private Notificador notificador;

ReservaService(PoliticaReserva politica,
  RegistroReserva repositorio,
  Notificador notificador) {
  this.politica = politica;
  this.repositorio = repositorio;
  this.notificador = notificador;
}
```

Contrato de Envio (Notificador):

```
void enviar(String destino, String mensagem);
```

Fluxo Consolidado (método reservar)

```
validador.validar(r);
if (!politica.permite(r)) {
  throw new IllegalStateException(
      "Horário não permitido");
}
String codigo = repositorio.salvar(r);
String mensagem =
      formatador.formatar(codigo, r);
notificador.enviar(r.contato, mensagem);
return codigo;
```

Composição na Aplicação: a aplicação escolhe as implementações concretas e injeta no serviço.

```
new ReservaService(new PoliticaLaboratorio(), repo, new EmailNotificador())
```

Estudo: Três Questões Teóricas

1. **Responsabilidade e Extensão**
   A confirmação terá outro texto e uma sala de reunião permitirá reservas de 9h a 20h. Onde fazer cada alteração? O que permanece igual no serviço?

2. **Substituição e Contrato**
   A política de reunião lança exceção para uma reserva válida de 8h–10h. Qual garantia ela quebra? Como deve representar a recusa?

3. **Interfaces e Dependências**
   O portal precisa consultar e o serviço precisa salvar. Quais contratos devem receber? Por que compartilhar a instância do repositório?

SOLID: Resumo e Conclusão

S

Responsabilidade Única

Separar validação, registro, formatação e notificação.

O

Aberto/ Fechado

Adicionar políticas de horário sem modificar o fluxo do serviço.

L

Substituição de Liskov

Garantir que todas as políticas respeitem o contrato de `permite`.

I

Segregação de Interfaces

Fazer o portal depender apenas de `ConsultaReserva`.

D

Inversão de Dependência

Fazer o serviço depender de contratos, com implementações fornecidas pela aplicação.

Resultado: responsabilidades coesas, mudanças mais localizadas e contratos confiáveis.

Antes de criar uma abstração, pergunte: qual responsabilidade, contrato ou necessidade de mudança ela representa?

Dúvidas e Discussão