Documentação do Projeto - Sistema de Gerenciamento de Produtos

1. Visão Geral

O sistema implementa um CRUD de produtos utilizando padrões de projeto modernos e técnicas avançadas de programação Java, incluindo reflexão e injeção de dependências.

2. Anotações Personalizadas

2.1. @Rota

```
package br.com.ucsal.annotations;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Rota {
    String value();
    String method() default "GET";
}
```

Propósito: Mapeia métodos para URLs específicas no sistema

Uso: Aplicada em métodos dos servlets

Atributos:

- `value`: Define o caminho da URL
- `method`: Define o método HTTP (padrão "GET")

2.2. @Inject

```
package br.com.ucsal.annotations;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
public @interface Inject {
}
```

Propósito: Marca campos para injeção automática de dependências

Uso: Utilizada principalmente em serviços para injetar repositórios

Exemplo prático:

```
    @Inject
    private ProdutoRepository<Produto, Integer> produtoRepository;
```

2.3. @Singleton

```
package br.com.ucsal.annotations;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface Singleton {
}
```

Propósito: Marca classes que devem ter apenas uma instância

Uso: Aplicada em repositórios em memória

Exemplo:

```
@Singleton
public class MemoriaProdutoRepository implements ProdutoRepository<Produto, Integer> {
```

3. Reflexão e Carga Dinâmica

3.1. Carregamento de Rotas

```
  private void carregarRotas() throws ServletException {
      try {
          Reflections reflections = new Reflections("br.com.ucsal.controller",
              new MethodAnnotationsScanner());

          Set<Method> metodos = reflections.getMethodsAnnotatedWith(Rota.class);

          for (Method method : metodos) {
              Rota rota = method.getAnnotation(Rota.class);
              rotas.put(rota.value(), new RouteInfo(method, rota.method()));
          }
          System.out.println("Rotas carregadas: " + rotas.keySet());
      } catch (Exception e) {
          throw new ServletException("Erro ao carregar rotas", e);
      }
  }
```

O sistema utiliza reflexão para:

1. Escanear classes no pacote `br.com.ucsal.controller`
2. Identificar métodos anotados com `@Rota`
3. Registrar automaticamente os mapeamentos URL → método

3.2. Injeção de Dependências

```
  public void processInject(Object object) {
      Class<?> clazz = object.getClass();
      for (Field field : clazz.getDeclaredFields()) {
          if (field.isAnnotationPresent(Inject.class)) {
              field.setAccessible(true);
              try {
                  Object dependency = null;
                  if (ProdutoRepository.class.isAssignableFrom(field.getType())) {
                      dependency = PersistenciaFactory.createProdutoRepository();
                  } else {
                      dependency = dependencies.get(field.getType());
                  }

                  if (dependency == null) {
                      throw new RuntimeException("Dependência não encontrada para: " +
field.getType());
                  }
```

```
                field.set(object, dependency);
            } catch (Exception e) {
                throw new RuntimeException("Erro ao injetar dependência em " +
                    clazz.getSimpleName() + "." + field.getName(), e);
            }
        }
    }
}
```

O processo de injeção:

1. Analisa campos da classe usando reflexão
2. Identifica campos marcados com `@Inject`
3. Resolve e injeta as dependências apropriadas

4. Padrões de Projeto Implementados

4.1. Factory Method

```
public static ProdutoRepository<Produto, Integer> createProdutoRepository() {
        switch (tipoPersistenciaAtual) {
          case MEMORIA:
          return SingletonManager.getInstance()
          .getSingleton(MemoriaProdutoRepository.class);
    case HSQLDB:
     return new HSQLProdutoRepository();
          default:
  throw new IllegalStateException("Tipo de persistência não suportado");
                    }
        }
```

Propósito: Criar instâncias de repositórios

Tipos suportados:

MEMORIA: Repositório em memória (Singleton)

HSQLDB: Repositório em banco de dados

4.2. Singleton

```
   public <T> T getSingleton(Class<T> clazz) {
       if (!clazz.isAnnotationPresent(Singleton.class)) {
           throw new IllegalArgumentException("Classe não é um Singleton");
       }

       @SuppressWarnings("unchecked")
       T singleton = (T) singletons.get(clazz);
```

```
        if (singleton == null) {
            synchronized (singletons) {
                singleton = (T) singletons.get(clazz);
                if (singleton == null) {
                    try {
                        singleton = clazz.getDeclaredConstructor().newInstance();
                        singletons.put(clazz, singleton);
                    } catch (Exception e) {
                        throw new RuntimeException("Erro ao criar singleton", e);
                    }
                }
            }
        }
        return singleton;
    }
```

Implementado de duas formas:

1. Gerenciamento centralizado: Através do `SingletonManager`
2. Double-checked locking: Para thread safety

5. Fluxo de Inicialização

```
    public void contextInitialized(ServletContextEvent sce) {
        try {
            // Configura o tipo de persistência
            PersistenciaFactory.setTipoPersistencia(TipoPersistencia.HSQLDB);

            // Configura o repositório
            DependencyManager dm = DependencyManager.getInstance();
            ProdutoRepository<Produto, Integer> repository =
                PersistenciaFactory.createProdutoRepository();
            dm.register(ProdutoRepository.class, repository);

            // Registra e configura o ProdutoService
            ProdutoService produtoService = new ProdutoService();
            dm.processInject(produtoService);
            dm.register(ProdutoService.class, produtoService);

            System.out.println("Sistema inicializado com sucesso");
        } catch (Exception e) {
            System.err.println("Erro ao inicializar o sistema: " + e.getMessage());
            throw new RuntimeException(e);
        }
    }
```

1. Configuração do tipo de persistência
2. Registro do repositório no DependencyManager
3. Configuração e registro do ProdutoService
4. Inicialização do banco de dados (se necessário)

6. Estrutura do Banco de Dados

```
        stmt.executeUpdate("CREATE TABLE produtos (" +
                "id INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY, " +
                "nome VARCHAR(50), " +
                "preco DOUBLE" +
                ")");
```

Tabela `produtos`:

- `id`: Chave primária auto-incrementada
- `nome`: VARCHAR(50)
- `preco`: DOUBLE

7. Interface do Usuário

O sistema oferece duas views principais:

Lista de Produtos (`produtolista.jsp`)

Formulário de Produto (`produtoformulario.jsp`)

8. Detalhamento Técnico Adicional

8.1. Processo de Escaneamento de Rotas

O sistema utiliza a biblioteca Reflections para escanear classes em três etapas:

1. Configuração do Scanner:

```
Reflections reflections = new Reflections("br.com.ucsal.controller",
    new MethodAnnotationsScanner());
```

2. Identificação de Métodos:
- Usa MethodAnnotationsScanner para encontrar métodos com @Rota
- Armazena em um Set<Method> para processamento

3. Mapeamento URL → Método:

java:src/main/java/br/com/ucsal/controller/ProdutoController.java
startLine: 45
endLine: 48

8.2. Ciclo de Vida das Dependências

8.2.1. Processo de Injeção

java:src/main/java/br/com/ucsal/controller/InicializadorListener.java
startLine: 19
endLine: 31

8.2.2. Resolução de Dependências Circulares

O sistema implementa as seguintes estratégias:

1. Lazy Loading:
- Dependências são carregadas apenas quando necessário
- Implementado no DependencyManager:

8.3. Factory Method e Troca de Repositórios

8.3.1. Configuração de Persistência

```java:src/main/java/br/com/ucsal/persistencia/PersistenciaFactory.java
startLine: 7
endLine: 9
```

8.3.2. Processo de Troca

1. Configuração Inicial:
```java
PersistenciaFactory.setTipoPersistencia(TipoPersistencia.MEMORIA);
```

2. Efeito nas Dependências:
- O DependencyManager detecta a mudança através do PersistenciaFactory
- Novas instâncias de serviços receberão o novo repositório
- Serviços existentes mantêm suas dependências até serem reinicializados

8.3.3. Garantias de Thread Safety

1. Para Repositórios em Memória:
```java:src/main/java/br/com/ucsal/persistencia/MemoriaProdutoRepository.java
startLine: 22
endLine: 31
```

2. Para o Gerenciador de Singletons:
```java:src/main/java/br/com/ucsal/config/SingletonManager.java
startLine: 34
endLine: 46
```

8.4. Exemplo de Uso Completo

```java
// 1. Configurar tipo de persistência
PersistenciaFactory.setTipoPersistencia(TipoPersistencia.MEMORIA);

// 2. Obter nova instância do repositório
ProdutoRepository<Produto, Integer> repository =
    PersistenciaFactory.createProdutoRepository();

// 3. Registrar no DependencyManager
DependencyManager.getInstance().register(ProdutoRepository.class, repository);

// 4. Processar injeções pendentes
ProdutoService produtoService = new ProdutoService();
DependencyManager.getInstance().processInject(produtoService);
```