Universidade de Brasília
Centro de Informática
Unidade de Serviços de Sistemas de Informação - SSI

# Guia Rápido para Desenvolvedores Java na Arquitetura Erlangms

Everton de Vargas Agilar

**Manual de Arquitetura**

# Resumo

Este guia sintetiza os conhecimentos que são necessários para trabalhar com a nova arquitetura Erlangms no ambiente Java. Este trabalho é resultado dos esforços realizados no Mestrado em Computação Aplicada pelo Analista Everton de Vargas Agilar do CPD/UnB.

A abordagem proposta por esta arquitetura impõem o uso de um barramento de serviço desenvolvido na linguagem Erlang e um SDK (Software Development Kit) na linguagem que será utilizada para a implementação dos Web-Services, neste caso, o SDK ems-java para a linguagem Java.

De forma muito resumida, a arquitetura ErlangMS tem o intuíto de facilitar a criação e a integração de sistemas através de uma abordagem orientada a serviços no estilo arquitetural REST (Representational State Transfer).

**Palavras-chave:** Arquitetura Orientada a Serviços, REST, Erlangms.

# Sumário

# Lista de Figuras

# Lista de Tabelas

# Lista de Abreviaturas e Siglas

**API** *Application Programing Interface*. 16

**DDD** *Domain-Driven Design*. 10, 12–15

**JPA** *Java Persistence API*. 16

**JSON** *JavaScript Object Notation*. 11

**PNAES** Programa Nacional de Assistência Estudantil da Universidade. 15

**POJO** *Plain Old Java Objects*. 13

**REST** *Representational State Transfer*. 11

**SAE** Sistema de Assistência Estudantil. 10, 13

**SDK** *Software Development Kit*. 11, 13, 16

**SOA** *Service Oriented Architecture*. 10

**SQL** *Structured Query Language*. 16

**VO** *Value Object*. 13, 15

# Capítulo 1

# Introdução

Este guia sintetiza os conhecimentos que são necessários para trabalhar com a nova arquitetura Erlangms no ambiente Java. Este trabalho é resultado dos esforços realizados no Mestrado em Computação Aplicada pelo Analista Everton de Vargas Agilar do CPD/UnB.

Parta adiantar, a abordagem proposta por esta arquitetura impõem o uso de um barramento de serviço desenvolvido na linguagem Erlang e um SDK (Software Development Kit) na linguagem que será utilizada para a implementação dos Web-Services, neste caso, o SDK ems-java para a linguagem Java.

De forma muito resumida, a arquitetura ErlangMS tem o intuíto de facilitar a criação e a integração de sistemas através de uma abordagem orientada a serviços no estilo arquitetural REST (Representational State Transfer). Não se preocupe, este estilo não é mais do que um conjunto de restrições que devemos seguir do que uma nova tecnologia. Assim, usa-se os verbos HTTP para realizar as operações desejadas, como por exemplo, o verbo POST para incluir ou cadastrar alguma coisa, o verbo PUT para alterar, o verbo GET para pesquisar e o DELETE para excluir.

Em uma arquitetura como esta, existe a figura do Front-end (a parte visual do sistema) e o Back-end (a parte onde estão as regras de negócios providos através de serviços).

## 1.1 Componentes em Tempo de Execução

Os principais componentes da arquitetura em tempo de execução estão listados a seguir com uma breve descrição sobre o seu objetivo sem entrar em muitos detalhes:

### 1.1.1 Barramento de serviços (ems-bus)

Sua função é interligar os clientes (tipicamente os Front-ends) aos serviços que contém as regras de negócios da organização. Quando alguém faz uma requisição para um serviço, é

o barramento que intermedia o envio e o recebimento das mensagens, sendo que o cliente só precisa saber o endereço do barramento para isso.

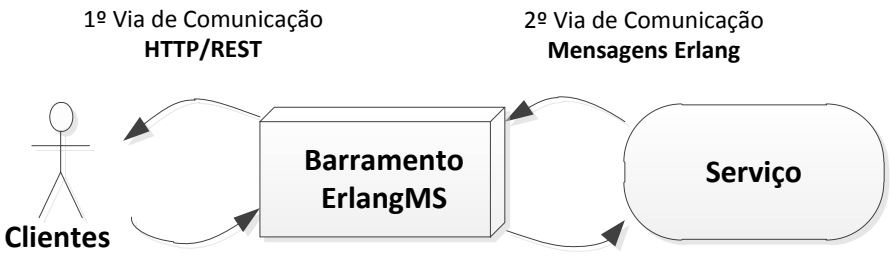


Figura 1.1: Esquema do roteamento das mensagens da arquitetura.

### 1.1.2 Catálogo de serviços

Representa o componente chave da arquitetura pois permite dar visibilidade aos serviços disponibilizados na organização. É no catálogo que os serviços são registrados. O exemplo a seguir expõem um serviço no catálogo para incluir uma resposta para um estudo preliminar do sistema SAE da UnB.

```
{
  "name": "/sae/estudo/preliminar/:id/resposta",
  "comment": "Cadastrar resposta do estudo preliminar",
  "owner": "sae",
  "service" : "br.unb.sae.facade.EstudoPreliminarFacade:insertResposta",
  "url": "/sae/estudo/preliminar/:id/resposta",
  "type": "POST"
}
```

Código 1.1: Exemplo de um serviço no catálogo de serviços.

### 1.1.3 Módulo Back-end

È um projeto Java Web típico do CPD/UnB mas desenvolvido com um padrão de design que será apresentado mais adiante neste guia e que contém a implementação dos serviços. No CPD, estes projetos são publicados em servidores de aplicação JBoss/Wildfly.

### 1.1.4 Módulo Front-end

Consiste na interface da aplicação bem como a parte que o usuário vê. O front-end é o responsável por coletar os dados de entrada do usuário e realizar as chamadas para os serviços por meio do barramento de serviços.

### 1.1.5 Servidor de Aplicação JBoss/Wildfly

O servidor de aplicação JBoss/Wildfly é onde são publicados (deployment) os projetos Java Web. Podem existir várias instâncias desses servidores para aumentar a escalabilidade dos serviços, sendo que o barramento despacha as requisições dos clientes utilizando um simples algoritmo round-robin.

### 1.1.6 Erlang Port Mapper Daemon

É um serviço que executa em segundo plano em cada nó de um cluster e que age como um servidor de nome. É importante salientar que tanto o barramento quanto os servidores de aplicação JBoss/Wildfly são vistos como nós na arquitetura. O cliente não faz parte do cluster pois é apenas um consumidor.

# Capítulo 2

# Criando um Novo Projeto

Para aliar a teoria com a prática, vamos começar desenvolvendo um novo projeto de back-end Java Web no Eclipse. Os projetos desenvolvidos no CPD/UnB na linguagem Java são do tipo Maven Project.

## 2.1 Usando o Wizard New Maven Project

Vá no menu File/New da IDE Eclipse e clique na opção Maven Project. Se a opção Maven Project não aparecer, localize-a dentro da opção Other.

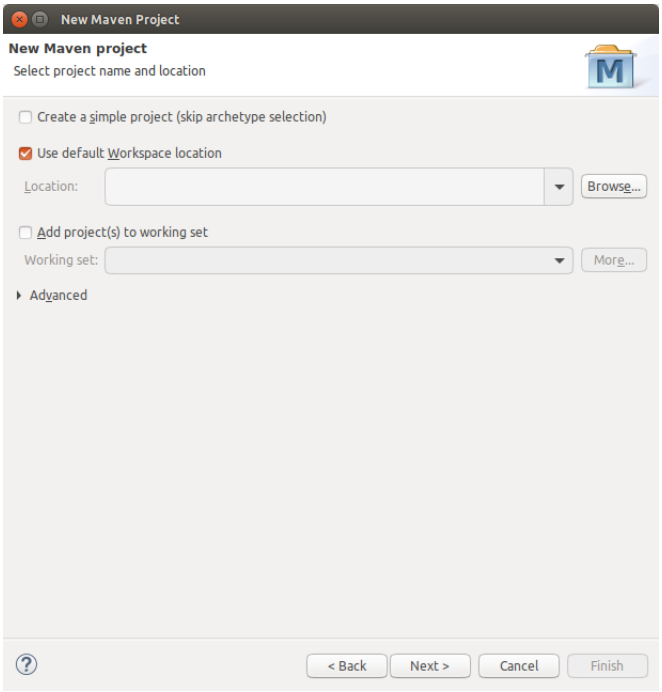


Figura 2.1: New Maven Project Wizard - Seleção local projeto.

A tela que se abre (Figura 2.1) é um Wizard para para a criação de um projeto Maven, que é o tipo de projeto Java EE que usamos no CPD/UnB. Nesta primeira etapa, deve-se escolher o local onde será armazenado o código fonte do projeto que por padrão é a workspace (marque a opção Use default Workspace location).

Na próxima tela, selecione o archetype da nova arquitetura para gerar um projeto com o esqueleto padrão da arquitetura Erlangms. Para isso, selecione o item archetype unb-modulo-erlangms-archetype da lista, conforme ilustra a Figura 2.2.

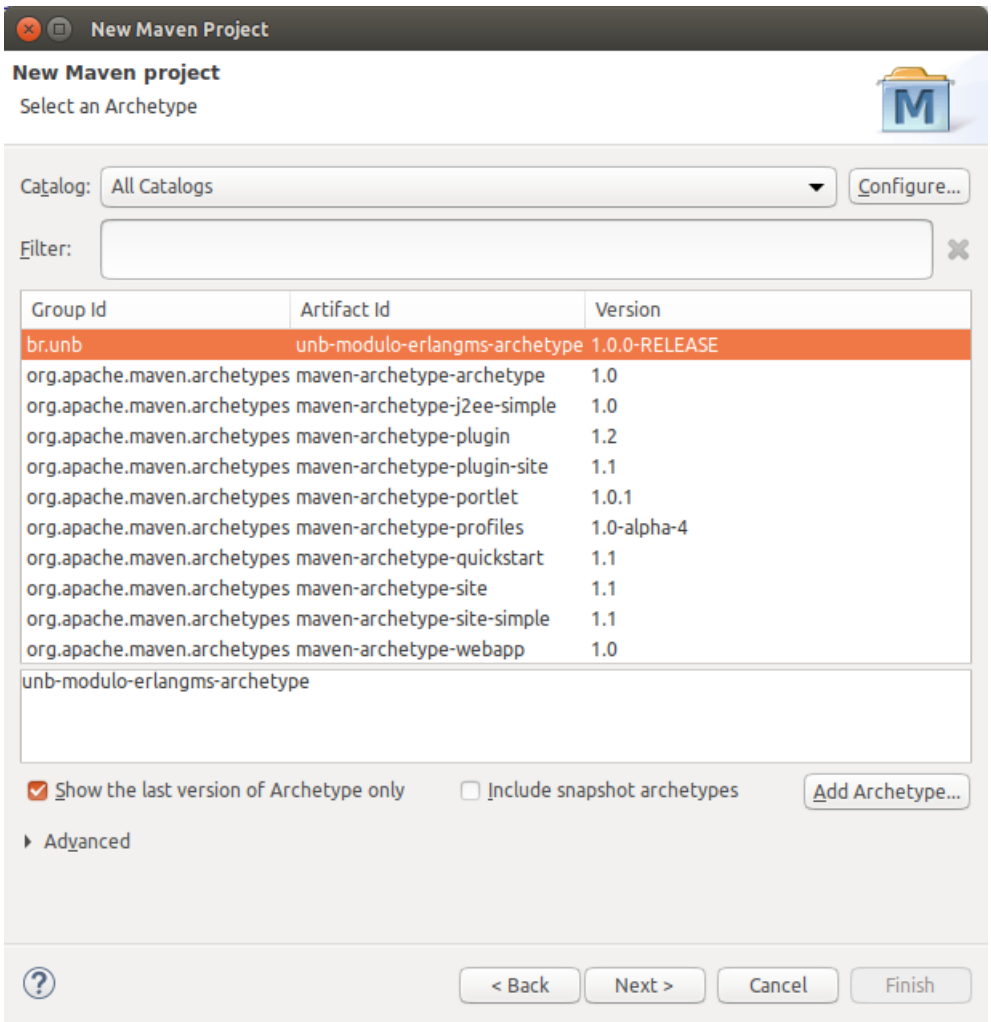


Figura 2.2: New Maven Project Wizard - Seleção archetype.

A partir de agora, devemos informar algumas informações importantes para o *New Maven Project* criar nossa aplicação Java Web:

(a) GroupId – namespace base do projeto;

(b) Artifact Id – nome do projeto;

(c) Version – versão do pacote do projeto;

(d) Package – namespace completo do package.

Assim, no campo GroupId informe br.unb; no campo Artifact Id informe o nome do seu projeto (por exemplo, unb_aula neste guia); no campo Version informe a versão do projeto. O último campo (Package) é gerado automaticamente pela IDE. A Figura 2.3 ilustra esta etapa do processo de criação da aplicação.

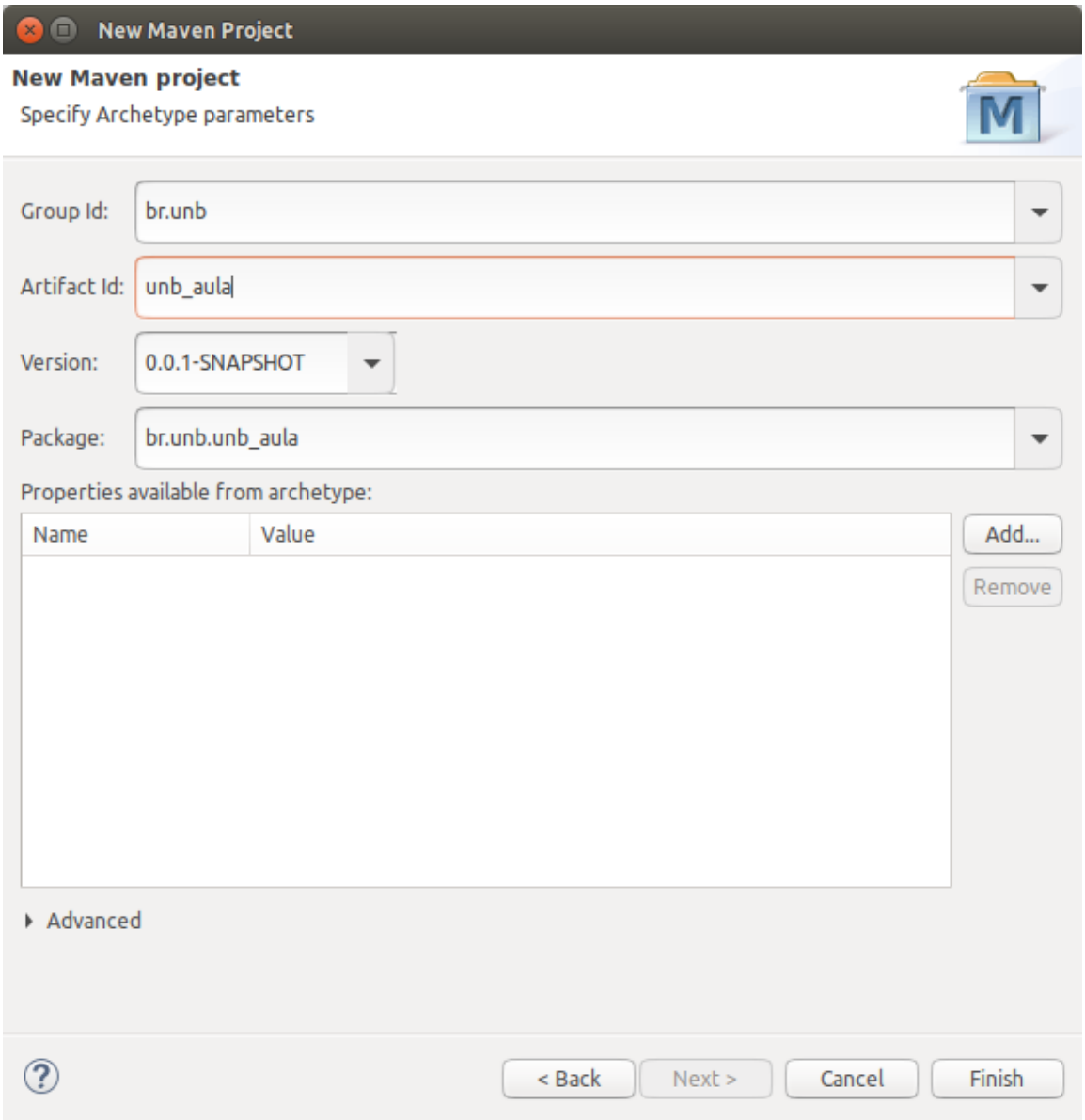


Figura 2.3: New Maven Project Wizard - Informações sobre o projeto.

Pronto, podemos clicar em Finish para criar o projeto. A Figura 2.4 ilustra o painel Project Explorer com a estrutura do projeto na arquitetura Erlangms.

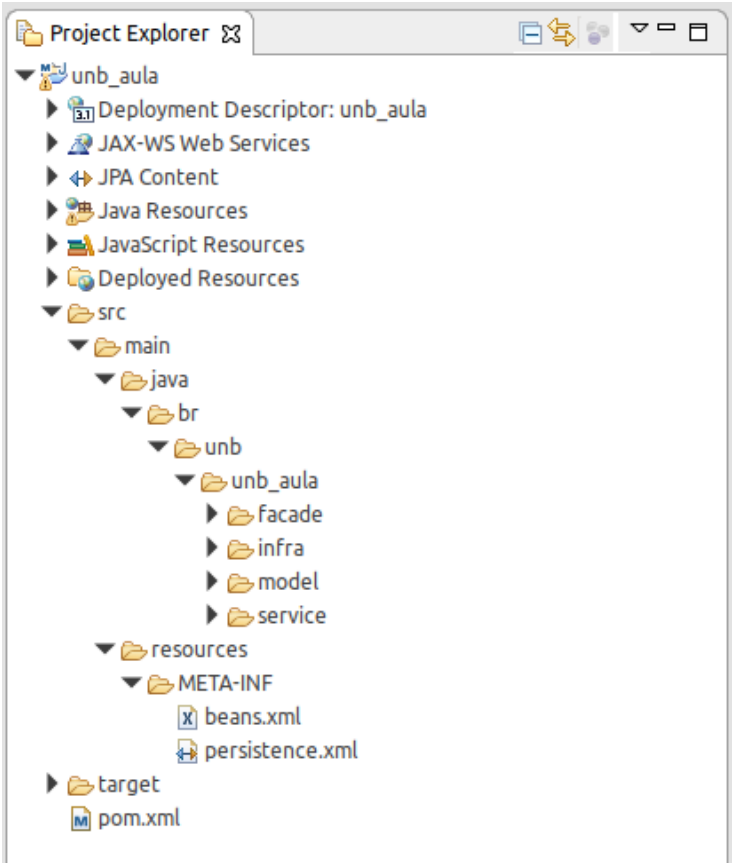


Figura 2.4: Estrutura do Novo Projeto na arquitetura Erlangms.

## 2.2 Dependências do Projeto

Em um projeto Java Web do tipo Maven, existe um arquivo especial chamado *pom.xml* onde são informadas as dependências do projeto. As dependências geralmente são bibliotecas ou frameworks que oferecem algo útil para as aplicações. O Código 2.1 exibe as dependências do projeto.

```
<dependencies>
    <dependency>
      <groupId>br.erlangms</groupId>
      <artifactId>ems_java</artifactId>
      <version>1.0.0</version>
    </dependency>

    <dependency>
      <groupId>javax</groupId>
      <artifactId>javaee-api</artifactId>
      <version>7.0</version>
      <scope>provided</scope>
    </dependency>
```

```
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-entitymanager</artifactId>
      <version>4.3.11.Final</version>
      <scope>provided</scope>
    </dependency>
</dependencies>
```

Código 2.1: Dependências na nova arquitetura.

A seguir, uma breve descrição sobre o objetivo de cada dependência:

- **ems_java** é o SDK Java do Erlangms para comunicação com o barramento ems-bus. Contém um conjunto mínimo de classes para construir Web-services agnósticos, ou seja, de forma mais neutra possível. O código fonte do SDK está disponível em `https://github.com/erlangMS/sdk`.
- **Java EE 7** é a base na qual as aplicações Java Edição Enterprise são construídas. Convém salientar que na arquitetura anterior (Fast), a versão utilizada é a 6.
- **hibernate-entitymanager** é o framework de persistência utilizado para a camada de persitência.

## 2.3 Persistência do Projeto

Outro importante arquivo em um projeto Java Web é o persistence.xml onde está configurado o banco de dados que será utilizado. Inicialmente, este arquivo está configurado para utilizar um banco de dados em memória.

Assim, pode-se desenvolver o back-end e até mesmo fazer deployment no JBoss/Wildfly pois as tabelas são automaticamente geradas de acordo com os modelos (ou POJOs) encontrados no projeto.

O Código 2.2 exibe a configuração de persistência do projeto.

```
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
      http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
  version="2.0" xmlns="http://java.sun.com/xml/ns/persistence">
  <persistence-unit name="service_context" transaction-type="JTA">
 <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <properties>
      <property name="javax.persistence.jdbc.driver"
          value="org.apache.derby.jdbc.EmbeddedDriver" />
```

```
      <property name="javax.persistence.jdbc.url"
          value="jdbc:derby://localhost:1527/database;create=true" />
      <property name="javax.persistence.jdbc.user" value="test" />
      <property name="javax.persistence.jdbc.password" value="test" />

    <!-- <property name="jboss.entity.manager.factory.jndi.name"
        value="java:/service_context" /> -->
    <property name="hibernate.show_sql" value="true" />
      <property name="hibernate.format_sql" value="false"/>
      <property name="hibernate.jdbc.use_scrollable_resultset"
          value="false"/>
      <property name="hibernate.hbm2ddl.auto" value="create" />
     </properties>
  </persistence-unit>
</persistence>
```

Código 2.2: Arquivo persistence.xml.

# Capítulo 3

# Design de Implementação

O design de implementação para os módulos back-end foi proposto no trabalho de dissertação *Uma Abordagem Orientada a Serviços para a Modernização de Sistemas Legados*, como uma recomendação de uso e baseia-se fortemente nos padrões de design DDD e SOA, descritos em [**?**, **?**, **?**, **?**, **?**].

Para demonstrar o seu uso, descreve-se a seguir a arquitetura de um módulo do SAE, contendo exemplos em código Java real.

## 3.1 Anatomia do módulo back-end

A Figura 3.1 ilustra a anatomia de um módulo de serviços de acordo com o padrão *Service Layer* descrito em [**?**]. A primeira observação que se pode fazer é a organização interna sob uma arquitetura em *layers* (ou arquitetura em camadas), cujo objetivo é separar os vários tipos de artefatos de um software de forma lógica e coesa [**?**].

O interessante deste design é a centralização dos códigos que tem relação com o negócio da aplicação em uma camada específica (a camada de serviços) e o isolamento dos outros tipos de artefatos, como a persistência dos dados, em outra camada, o que pode, segundo [**?**], simplificar o desenvolvimento do sistema ao evitar (ou talvez minimizar) que diferentes tipos de códigos fiquem misturados.

## 3.2 Finalidade de cada camada

Com base nestes aspectos, apresenta-se a seguir uma breve descrição da finalidade de cada camada, no modelo de arquitetura proposto para os módulos de serviços.

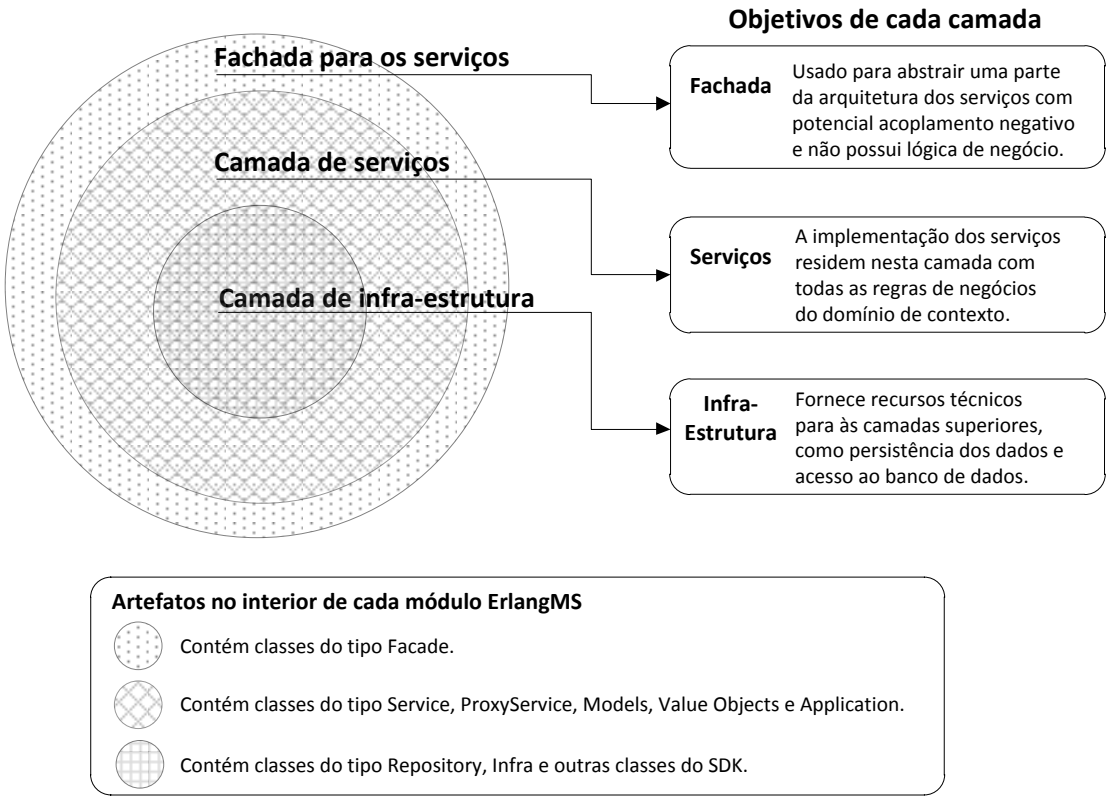


Figura 3.1: Anatomia de um módulo da camada de serviços.

### 3.2.1 Primeira Camada: De fachada

Implementa o padrão *ServiceFacade* identificado em [**?**]. Salienta-se que esta camada não deve conter lógica de negócio e o seu objetivo é evitar um possível acoplamento indesejado ao impedir que a lógica de negócio localizada na camada de serviços seja exposta diretamente ao barramento. Assim, quando o cliente requisita um serviço ao barramento, na verdade, é invocado um método da fachada.

Em termos práticos, a finalidade desta camada é possibilitar que os parâmetros da requisição REST sejam lidos e o método do serviço correspondente seja invocado com os dados que ele necessita. E, no caso do serviço retornar uma resposta ao cliente, a fachada pode realizar algum processamento opcional (no estudo de caso não foi necessário) ou simplesmente retornar o resultado ao cliente[1].

O Código 3.1 mostra um exemplo de fachada para o serviço *QuestionarioService* do módulo unb_questionario, onde há três métodos declarados que ao serem invocados, ocorrem os seguintes eventos nessa ordem: os dados da requisição são lidos a partir do *IEmsRequest request* (interface que contém os dados da requisição); após, é obtido a

[1]O desenvolvedor não precisa preocupar-se com a transformação dos dados de/e para JSON, isso é de responsabilidade do SDK e do barramento.

referência ao objeto *QuestionarioApplication*, responsável pelo acesso a camada de serviços do módulo; depois, a referência para o objeto do serviço correspondente é recuperada, e finalmente; o método do serviço solicitado é invocado com os parâmetros requeridos.

```
package br.unb.questionario.facade;

import br.unb.questionario.service.QuestionarioApplication;

public class QuestionarioFacade extends EmsServiceFacade {
  public Questionario findById(IEmsRequest request){
    Integer id = request.getParamAsInt("id");
    return QuestionarioApplication.getInstance()
      .getQuestionarioService()
      .findById(id);
  }

  public Questionario insert(IEmsRequest request){
    Questionario questionario = (Questionario)
        request.getObject(Questionario.class);
    return QuestionarioApplication.getInstance()
      .getQuestionarioService()
      .insert(questionario);
  }

  public boolean vinculaPerguntaAoQuestionario(IEmsRequest request){
    int questionario_id = request.getParamAsInt("id");
    int pergunta_id = request.getPropertyAsInt("pergunta");
    QuestionarioApplication.getInstance()
      .getQuestionarioService()
      .vinculaPerguntaAoQuestionario(questionario_id, pergunta_id);
    return true;
  }
  // outros métodos omitidos...
}
```

Código 3.1: Exemplo de fachada para o serviço QuestionarioService.

### 3.2.2 Segunda Camada: De serviços

A camada de serviços consiste na parte mais importante da arquitetura mostrada na Figura 3.1, pois na visão de [?, ?], é o local onde concentra-se a lógica de domínio do negócio do sistema. Por conta disso, convém destacar que parte do foco do DDD, como identificado em [?, ?], situa-se nessa camada, razão pela qual o design da arquitetura

proposta tenta (na medida do possível) não focar em tecnologia, mas, em vez disso, entender as regras do negócio e como refleti-las no código fonte da aplicação de maneira agnóstica.

Nesse sentido, ao modelar a camada de serviços, pode-se fazer uso de diversos *blocos de construção* definidos no DDD, que são, essencialmente, os artefatos que constituem esta camada. Para exemplificar e explicar alguns conceitos desses artefatos, segue uma descrição dos principais elementos utilizados na camada de serviços dos módulos desenvolvidos no SAE.

- Entidades. São as classes de objetos que possuem identidade, comportamentos (ou métodos) e um ciclo de vida [**?**]. Por exemplo, um aluno do SAE que se autentica no sistema e realiza a avaliação socioeconômica. Em sistemas Java típicos, como os desenvolvidos pelo CPD/UnB, as classes de entidades podem ser comparados a classes *Plain Old Java Objects* (POJO), embora um POJO siga algumas definições de design que uma Entidade não precisa (por exemplo, ter um construtor padrão sem argumentos e métodos *getters* e *setters* para os atributos), de acordo com [**?**].

- *Value Object* (VO). São as classes de objetos que são modeladas geralmente para carregar dados e possuem o sufixo *Vo* no SAE por convenção. Uma classe desse tipo é o *CampusVo* que possui apenas dois atributos (id e nome) e foi implementada para que os serviços do módulo unb_sae possam carregar a lista de campus a partir de um serviço que está no módulo unb_sitab. Existem outras características sobre VO que não serão tratadas aqui por simplicidade mas em [**?**] é discutido em mais detalhes.

- *Factory*. São as classes responsáveis por *fabricar* os objetos e usadas na fabricação dos serviços nessa camada. Essas classes tem o sufixo *Application* por convenção (QuestionarioApplication, por exemplo) e apenas por curiosidade, classes desse tipo são também conhecidas pelo padrão *ApplicationService* no guia de padrões de design Java EE (Core J2EE Patterns) [**?**].

  Como é possível observar no Código 3.1 de exemplo, a fachada somente acessa um objeto de serviço mediante o QuestionarioApplication. O principal motivo para o uso desse design no SAE é esconder a forma como se criam ou injetam os objetos (com injeção de dependência), uma vez que é desejável que o SDK em outras linguagens de programação façam o mesmo, independente da tecnologia empregada para isso. Outro motivo importante é minimizar as dependências e o acoplamento entre os objetos. Note que isso foi obtido, pois, é possível verificar que os métodos da fachada acessam somente a interface do serviço e não precisam importar a

classe de implementação do serviço, somente a classe QuestionarioApplication em br.unb.questionario.service.

- *Services*. São as classes que contém as regras de negócios da aplicação e a implementação do serviço de acordo com a sua especificação no catálogo de serviços do projeto. Note que os objetos do tipo Entidades também contém regras de negócios porque o DDD trabalha com objetos com comportamentos e não somente atributos (ou seja, objetivos não anêmicos).

  Além disso, uma observação importante sobre os serviços identificado em [**?**], é que as classes de serviço não devem ter estado, obedecendo dessa forma, a restrição REST *Stateless*, discutida no Capítulo **??**.

Para finalizar a descrição dos conceitos discutidos sobre a camada de serviços da arquitetura proposta, o Código 3.2 mostra a implementação parcial do código fonte da classe do serviço QuestionarioService com os 3 métodos invocados pela fachada demonstrada anteriormente no Código 3.1. Perceba que o serviço faz uso da camada de infraestrutura apresentada a seguir.

```
package br.unb.questionario.service;

// imports omitidos para facilitar a visualização

public class QuestionarioService {
  public Questionario findById(Integer id) {
    return QuestionarioInfra.getInstance()
      .getQuestionarioRepository()
      .findById(id);
  }

  public Questionario insert(Questionario questionario) {
    questionario.validar();
    return QuestionarioInfra.getInstance()
      .getQuestionarioRepository()
      .insert(questionario);
  }

  public void vinculaPerguntaAoQuestionario(
    int questionario_id, int pergunta_id) {
    Questionario questionario = findById(questionario_id);
    Pergunta pergunta = QuestionarioApplication.getInstance()
                .getPerguntaService()
                .findById(pergunta_id);
```

```
        questionario.vinculaPergunta(pergunta);
    }
    // outros métodos omitidos...
}
```

Código 3.2: Exemplo de implementação do serviço QuestionarioService.

### 3.2.3 Terceira Camada: Infraestrutura

É muito comum, parte do software não estar diretamente relacionado ao domínio do negócio, mas a sua infraestrutura de apoio [**?**]. Assim, a finalidade desta camada é prover os recursos técnicos necessários para as camadas superiores do módulo, como o acesso, a persistência e a consulta dos objetos em um banco de dados, a escrita de logs para o registro de eventos relevantes, entre outros recursos.

No entanto, [**?**, **?**] afirmam que esta camada deve prover os recursos tecnológicos de forma isolada, não expondo os detalhes internos da infraestrutura, pois podem comprometer a camada de serviços com aspectos técnicos do software misturados com as regras de negócios. Nesse caso, [**?**] sugere que a camada de infraestrutura seja exposta através de interfaces simples.

Assim como na camada de serviços, existem alguns artefatos para modelar esta camada, previstos no DDD [**?**]. Na arquitetura proposta, sugere-se utilizar os artefatos *Repository* (Repositório) e o *Factory*, para lidar com os desafios discutidos anteriormente. Segue uma breve descrição de alguns conceitos desses artefatos com exemplos de utilização em código Java.

- *Repository*. Tradicionalmente, a maioria das aplicações precisam persistir ou recuperar os objetos em algum banco de dados. Sendo assim, o objetivo das classes *Repository* é basicamente gerenciar o ciclo de vida dos objetos, como as Entidades ou VO, centralizando as operações de criação, modificação, exclusão e consulta de objetos em um banco de dados [**?**].

  Para exemplificar, o Código 3.3 mostra a classe de repositório *OcorrenciaRepository* do módulo *unb_sae*, cujo objetivo é gerenciar o ciclo de vida das ocorrências de um estudante, um tipo de evento ocorrido em um determinado período que pode acarretar na suspensão do auxílio alimentação, mediante justificativa, de acordo com as normas do Programa Nacional de Assistência Estudantil da Universidade (PNAES). As classes *Repository* na arquitetura proposta possuem o sufixo Repository por convenção.

  A primeira observação nessa classe é quanto a sua interface. Por motivos já discutidos, as classes *Repository* foram projetadas para terem uma interface simples. Note

que a classe OcorrenciaRepository herda de *EmsRepository* (veja a Figura 3.2) fornecido pelo SDK e não precisa implementar nenhum método público nesse exemplo, uma vez que as operações herdadas são suficientes. Em outras classes, poderá ser necessária a criação de outros métodos públicos conforme a necessidade. Pode-se notar também alguns métodos protegidos e que na implementação em Java do SDK, os repositórios usam o *Java Persistence API* (JPA), um *framework* de persistência de dados. Este é um detalhe técnico que importa somente aos repositórios e não devem ser expostos, como verificou-se em [**?**, **?**]. Por curiosidade, o diagrama de classe na Figura 3.2 exibe as operações comuns a todas as classes *Repository* na arquitetura.

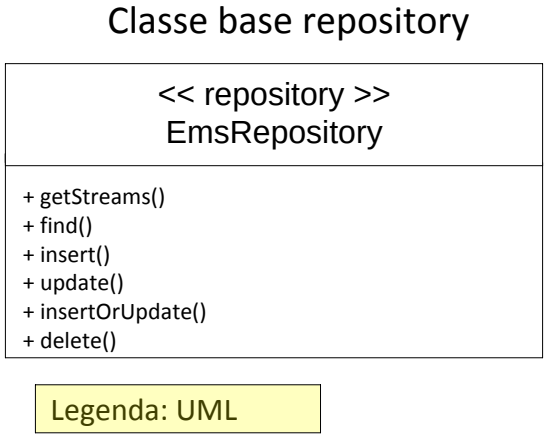


Figura 3.2: Classe base repository.

Como pode-se ver na Figura 3.2, há uma operação denominada *getStreams()*, que merece uma explicação. A ideia desse método surgiu no estudo de caso (não fazia parte da proposta inicial da arquitetura) para fornecer uma interface comum para consulta de objetos. Ou seja, como a maioria das operações geralmente são consultas (nos sistemas do CPD/UnB), buscou-se uma forma de expor uma pequena API para que o desenvolvedor não precise implementar um novo método de consulta sempre que for necessário pesquisar objetos por determinada condição (contando que a pesquisa seja simples). O Código 3.4 ilustra o uso desta API em alguns métodos da entidade Aluno fazendo uso do método *getStreams()*, onde é possível notar uma certa praticidade com o uso desta funcionalidade, além de possibilitar emitir as consultas usando uma linguagem tipificada em vez de usar as linguagens de consultas tradicionais, como a *Structured Query Language* (SQL).

- *Factory.* São as classes responsáveis por fabricar os repositórios (ou outros objetos) da camada de infraestrutura, tendo por convenção, o sufixo *Infra* neste trabalho. No

Código 3.4, é possível observar que a classe Aluno acessa os repositórios requeridos por meio da classe *SaeInfra*, o *Factory* da camada de infraestrutura do módulo unb_sae. O principal motivo do uso desse design, como discutido anteriormente, é esconder a forma como se criam os objetos.

```
package br.unb.sae.infra;

// imports omitidos para facilitar a visualização

public class OcorrenciaRepository extends EmsRepository<Ocorrencia>{

  @PersistenceContext(unitName = "service_context")
  protected EntityManager saeContext;

  @Override
  protected Class<Ocorrencia> getClassOfModel() {
    return Ocorrencia.class;
  }

  @Override
  protected EntityManager getEntityManager() {
    return saeContext;
  }

}
```

Código 3.3: Exemplo de implementação do repositório QuestionarioRepository.

```
package br.unb.sae.model;

// imports omitidos para facilitar a visualização

public class Aluno{

  private boolean assinouTermoConcessaoValeAlimentacao(
    String periodo) {
      int this_aluno = getId();
      return SaeInfra.getInstance()
        .getAssinaturaTermoBaRepository()
        .getStreams()
        .anyMatch(a -> a.getAluno() == this_aluno &&
                  a.getPeriodo().equals(periodo));
  }

  private boolean existeOcorrenciaAberto(
```

```
    String periodo, Date dataInicio) {
      int this_aluno = getId();
      return SaeInfra.getInstance()
        .getOcorrenciaRepository()
        .getStreams()
        .anyMatch(a -> a.getAluno() == this_aluno &&
                  a.getPeriodo().equals(periodo) &&
                  a.getDataInicio().equals(dataInicio));
  }

  public List<Ocorrencia> getListaOcorrencia() {
    int aluno_id = getId();
    return SaeInfra.getInstance()
        .getOcorrenciaRepository()
        .getStreams()
        .where(a -> a.getAluno() == aluno_id)
        .toList();
  }

  public Ocorrencia findOcorrenciaById(Integer idOcorrencia) {
    return SaeInfra.getInstance()
      .getOcorrenciaRepository()
      .findById(idOcorrencia);
  }

  public List<AssinaturaTermoBa>
     getListaAssinaturaTermoConcessaoValeAlimentacao() {
    int this_aluno = getId();
    return SaeInfra.getInstance()
        .getAssinaturaTermoBaRepository()
        .getStreams()
        .where(a -> a.getAluno() == this_aluno)
        .toList();
  }
  // outros métodos omitidos
}
```

Código 3.4: Exemplo de uso do método getStream() dos repositórios para consulta.