Pontifícia Universidade Católica do Rio Grande do Sul

Hackatona AGES
Documento de Arquitetura de Software

Alessandro Medeiros, Chiara Paskulin, Gabriel Correa, João Nascimento

Versão <2.1>

Histórico de Revisão

Data	Versão	Descrição	Autor
<10/Mai/20>	<1.0>	Iniciada a estruturação do documento e seus componentes.	João Nascimento
<15/Mai/20>	<1.1>	Modificação de tópicos e adição dos diagramas de caso de uso e diagrama de pacotes.	João Nascimento
<24/Mai/20>	<1.2>	Adição dos diagramas para o *front-end* e ajustes na escrita do documento de arquitetura.	João Nascimento
<28/Jun/20>	<2.0>	Adição do diagrama de sequência para criar um novo time.	João Nascimento
<01/Jul/20>	<2.1>	Adição dos diagramas UML para os padrões de projeto desenvolvidos.	João Nascimento

Sumário

1. Introdução 4
   - 1.1 Propósito 4
   - 1.2 Escopo 4
   - 1.3 Definições, Acrônimos e Abreviações 4

2. Representação Arquitetural – *Back-End* 5

3. Representação Arquitetural – *Front-End* 5

4. Diagrama de Casos de Uso 6
   - 4.1 Visão Geral 8
   - 4.2 Pacotes importantes para a arquitetura 8
   - 4.3 Realizações de Caso de Uso 9

5. Tamanho e Performance 9

6. Qualidade 9

7. Banco de Dados 9

8. Padrões de Projeto 9
   - 8.1 Facade 9
   - 8.2 Observer 10
   - 8.3 Repository 12

9. Diagramas de sequência 12

Documento de Arquitetura de Software

1. Introdução

1.1 Propósito

Esse documento fornece uma visão compreensível e amigável sobre a arquitetura de software do sistema proposto, sendo este um sistema desenvolvido para administrar as *hackatons* promovidas pela Agência Experimental de Engenharia de Software. Apresentaremos neste documento a arquitetura adotada, as decisões sobre a arquitetura e framework utilizados, seguido da explicação sobre os módulos e componentes da aplicação. Apresentamos nas próximas seções o escopo do documento e seu conteúdo, algumas definições formais sobre arquitetura de software diagramas de pacote, de caso de uso, diagrama de classes e diagrama de implantação.

1.2 Escopo

Este documento tem como objetivo apresentar brevemente os diagramas de caso de uso, as *user stories* (US) elencadas para melhorar a visão sobre as atividades possibilitadas pelos diferentes usuários do sistema, explicar como os pacotes foram estruturados e a motivação por trás da tomada de decisão. Apresentaremos uma breve introdução sobre as tecnologias utilizadas e as arquiteturas implementadas por baixo de cada *framework* tanto para o módulo de interface com o usuário quanto a API construída para que o cliente pudesse consumir os dados de maneira otimizada e organizada do ponto de vista de arquitetura de software, mantendo as responsabilidades de lógica de negócio, manipulação de dados, acesso ao banco de dados e controle de requisições distribuídas entre os diferentes módulos da aplicação.

1.3 Definições, Acrônimos e Abreviações

API: *Application Programming Interface*. Abstração de métodos e rotinas desenvolvidas e disponíveis para uso de diferentes aplicações ou bibliotecas.

Back-End: A parte lógica onde o processamento sobre os dados da aplicação e as regras de negócio (cálculos, operações de armazenamento) são executados.

Front-End: A parte visual apresentada ao usuário final. Toda a interface gráfica utilizada para interagir, como componentes de botões, caixas de texto ou imagens de alerta sobre o comportamento do sistema. Envia e recebe informações ao back-end.

MVC: *Model-View-Controller*. Padrão de projeto de arquitetura de software que visa o reaproveitamento de código, separando a aplicação em três camadas lógicas distintas de entidade, visualização e controle.

US: *User Stories*. Especificação de requisitos de software (funcionalidades, comportamentos) apresentadas através de texto cotidiano e ações de trabalho para o usuário final.

SQL: *Structured Query Language*. Linguagem estruturada para banco de dados relacionais, definida para operar sobre seus respectivos dados.

Persona: uma representação fictícia de uma pessoa enquanto usuário final do sistema, utilizada para melhor visualização de um requisito funcional antes de sua implementação.

ORM: *Object Relational Mapping*. Abstração de banco de dados, onde toda a construção é códigos DDL (*Data Definition Language*) são informados em formatos de objetos em uma linguagem orientada a objetos, como Java.

2. Representação Arquitetural – *Back-End*

O padrão arquitetural aplicado pelo software atual é o MVC. As entidades e seus controladores foram desenvolvidas em linguagem Java, utilizando *Spring Boot*, um popular *framework* para aplicações web. O lado de modelagem de dados e controle de entrada e saída (as camadas M e C) são construídas a partir deste *framework.* A camada de visualização dos dados (a camada V), muitas vezes sendo a parte visível ao usuário final, é desenvolvida utilizando o *framework* VueJs. Como a interface com o usuário apresenta diversos botões, caixas de texto e tabelas, componentes semelhantes estão presentes em diferentes páginas da aplicação. O *framework* VueJs permite criar componentes reutilizáveis e adaptáveis através da sua arquitetura de componentes genéricos. Apresentaremos a arquitetura adotada para o lado do cliente (View) com VueJs e a arquitetura adotada para o lado do servidor (Model e Controller) com *Spring Boot.*

A Figura 1 apresenta uma breve visão e explicação sobre os módulos presentes no lado do cliente e a Figura 2 apresenta uma visão sobre o lado do servidor e como as duas partes estabelecem comunicação.

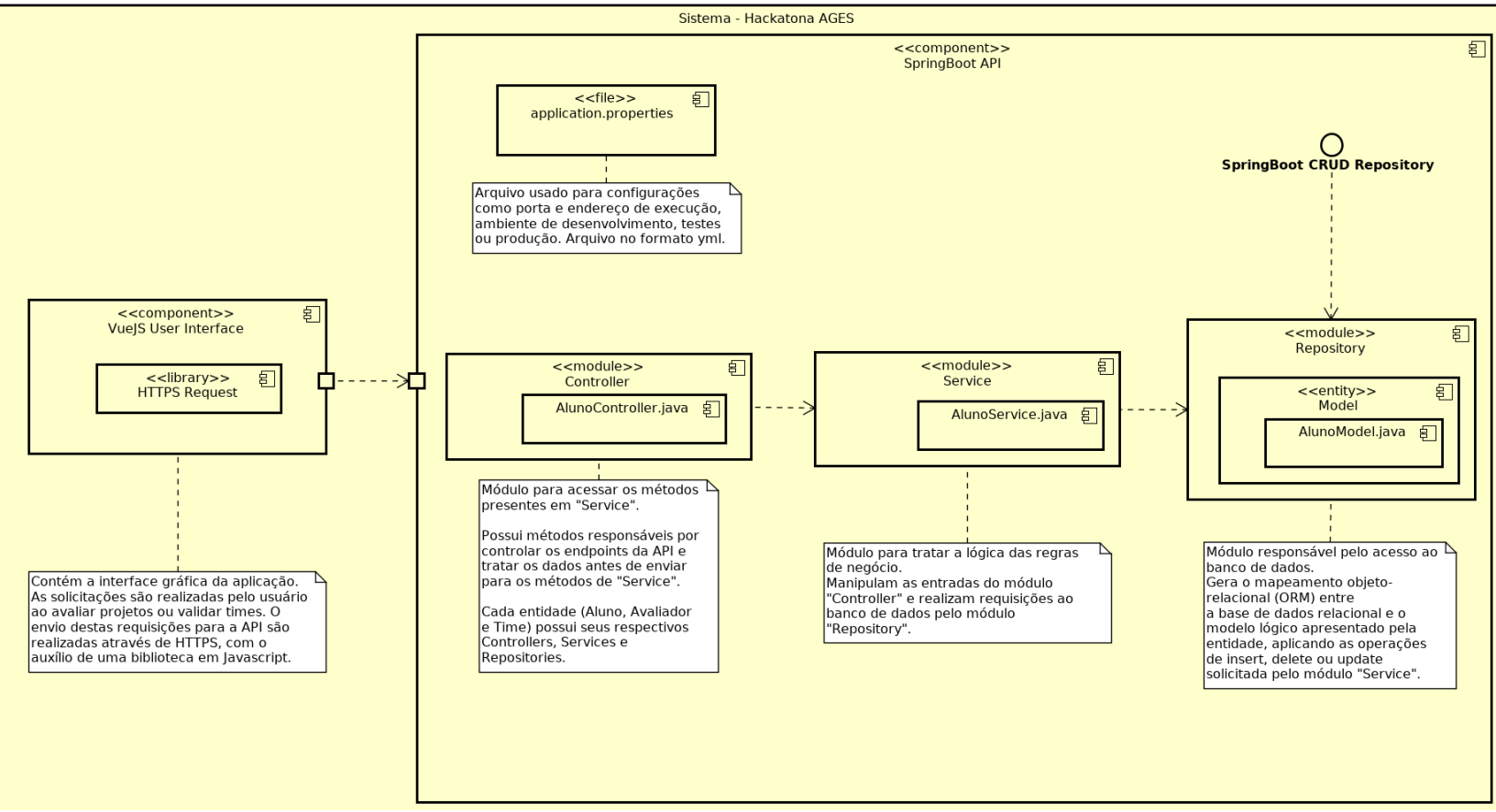

Figura 1: Diagrama de Componentes do Back-end

3. Representação Arquitetural – *Front-End*

Uma vez que os componentes de *front-end* possuem semelhanças visuais e apenas ajustes em suas funcionalidades,

nós optamos por uma arquitetura de componentes para reutilização de código. Dessa forma, o módulo de componentes guarda abstrações sobre itens do *front-end* como botões, caixas de texto ou caixa modal para informativos ao usuário. O módulo de *services* possui os métodos de acesso ao *back-end*.

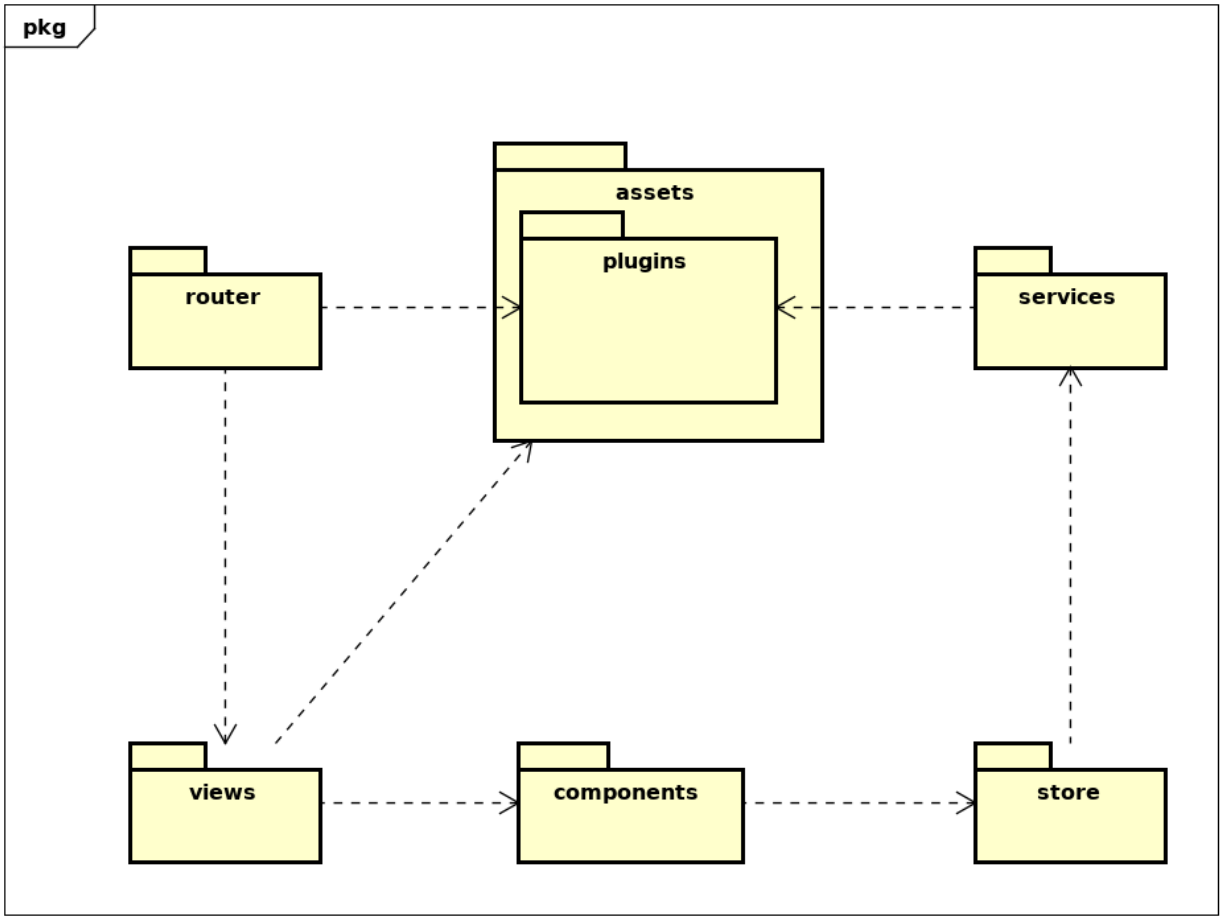

Figura 2 - Diagrama de pacotes do lado View

4. Diagrama de Casos de Uso

Apresentaremos nessa seção a lista dos principais casos de uso elencados através do enunciado do trabalho da disciplina e o diagrama gerado a partir dessas definições. A Figura 1 ilustra o diagrama de caso de uso com duas personas definidas pelo grupo. Anexado ao documento encontram-se os documentos de *Use-Case* para cara uma das funcionalidades apresentadas aqui.

- Moderador:
  - Eu, como moderador da Hackatona da AGES, gostaria de cadastrar integrantes a partir dos alunos inscritos, para conseguir formar equipes mistas dos cursos de computação;
  - Eu, como moderador da Hackatona da AGES, gostaria de cadastrar equipes com os integrantes do evento, para finalizar o processo de definição de equipes;
  - Eu, como moderador da Hackatona da AGES, gostaria de remover integrantes do evento e da equipe, em situações de desistência ou não concordância com as regras;

- Eu, como moderador da Hackatona da AGES, gostaria de remover uma equipe do evento, em situações de desistência ou não conformidade com as regras;
- Eu, como moderador da Hackatona da AGES, gostaria de validar os times cadastrados para verificar se os mesmos se encontram em conformidade com as regras do evento;

- Avaliador:
  - Eu, como avaliador da Hackatona da AGES, gostaria de avaliar os projetos apresentados com notas de 1 a 5, para que o vencedor seja eleito de maneira parcial.
  - Eu, como avaliador da Hackatona da AGES, gostaria de avaliar os pontos individuais por projeto, como pitch, inovação, processo de desenvolvimento e funcionamento do software para que todas as equipes tenham chances iguais de vencer independente do seu nível técnico e maturidade profissional;

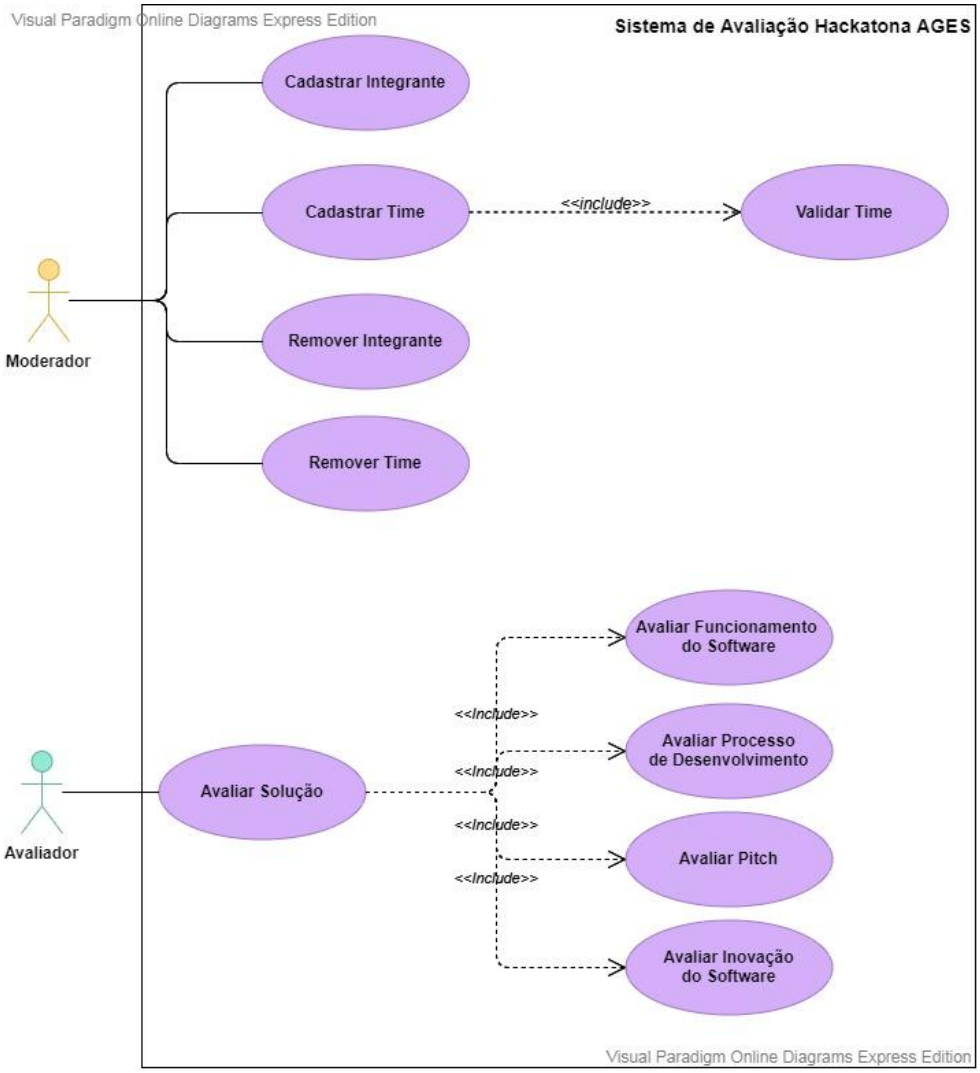

Figura 3: Diagrama de Caso de Uso

4.1 Visão Geral

4.2 Pacotes importantes para a arquitetura

A arquitetura da API conta com quatro principais pacotes, sendo estes: *controller, model, repository* e *service*. Cada um dos pacotes é responsável por armazenar as classes que possuem responsabilidades específicas sobre a API.

Controller: contém os métodos disponíveis para o cliente realizar suas requisições de consumo de dados. Cada *endpoint* é mapeado para uma função específica em Java, que recebe os parâmetros necessários e os converte para objetos e envia então para a camada de serviços.

Service: contém os métodos de lógica de negócio. Possui métodos que manipulam os dados enviados pelas controladoras, seja para operar sobre eles ou requisitar informações ao banco de dados. Cada tipo de entidade de banco de dados possui sua própria classe de serviço. Por exemplo, a entidade "Avaliador" possui sua classe de serviço que é responsável por operar atividades realizadas por um avaliador, como avaliar um time. As consultas ao banco de dados são realizadas através dos seus respectivos repositórios.

Repository: contém interfaces Java que tem como funcionalidade operar sobre os dados do banco de dados, abstraindo as consultas clássicas em SQL (*insert, delete, update* e *select),* através de métodos criados nessas interfaces. Por exemplo, para evitar a construção de uma *query* de seleção, a interface Java pode conter apenas um método chamado *findById(Integer id)* e o *framework* ficará encarregado de construir uma busca pela tabela de sua respectiva entidade, filtrando pelo identificador passado por parâmetro. Estão diretamente acopladas ao pacote de entidades.

Model: contém as abstrações lógicas de cada entidade do banco de dados. As tabelas e relações entre colunas são mapeadas de maneira automática através de anotações Java, fornecendo a abstração necessária para o repositório conseguir realizar as operações de banco de dados apenas com essas informações. Cada classe Java indica uma tabela no banco de dados e seus atributos indicam suas respectivas colunas.

A Figura 3 representa o diagrama de pacotes da API:

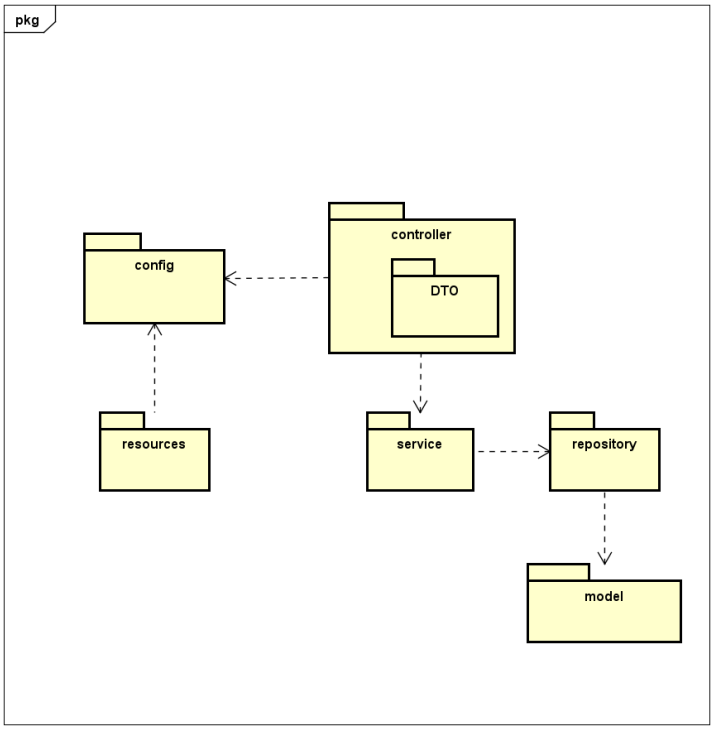

Figura 4 - Diagrama de pacotes da API

4.3 Realizações de Caso de Uso

Elencamos alguns principais casos de uso através da especificação fornecida no enunciado do trabalho. Para tanto, as principais personas definidas foram “aluno”, “avaliador” e “moderador”. O tópico 4 apresenta detalhadamente algumas das US elaboradas pelo grupo iniciar o desenvolvimento desta solução.

5. Tamanho e Performance

Considerando que a aplicação deve rodar em diferentes dispositivos (notebooks ou smartphones), a performance quanto ao tempo de resposta e envio das informações e o tratamento das mesmas deve ser considerado. Dessa forma,

6. Qualidade

A qualidade do desenvolvimento desta solução foi notável durante a execução do projeto, uma vez que tínhamos uma boa definição dos requisitos e de como ficaria a divisão de entidades e suas responsabilidades dentro da API. Por exemplo, uma vez desenvolvida a funcionalidade de cadastro de alunos, cadastrar um avaliador ou time apresentava um comportamento semelhante necessitando apenas de algumas alterações no código. A manutenção do código também apresentou grande facilidade uma vez que as camadas estavam desacopladas, permitindo descobrir se um método não funcionava por dados não tratados, ausentes ou por lógica de implementação.

7. Banco de Dados

O grupo optou por utilizar um banco local e em memória, chamado H2.O H2 é um banco de dados escrito em Java e permite ser embutido a diversos sistemas. Ele pode ser utilizado por pequenas aplicações para persistir dados em disco/arquivos ou validar algumas modelagens de maneira rápida. É ideal para testar a modelagem utilizando a estrutura ORM. Por já estarmos utilizando o *framework* Spring, a integração com o banco H2 é praticamente automática, uma vez que o arquivo JAR do banco de dados já está embutido nas dependências utilizadas pelo *framework.* Como o banco já é embutido ao *framework*, sua configuração e conexão é realizada apenas alterando alguns parâmetros no arquivo global de configuração, o *application.properties.*

8. Padrões de Projeto

Nessa seção serão apresentados os três padrões de projetos implementados pelo grupo na segunda etapa de desenvolvimento do trabalho. O desenvolvimento ocorreu a partir dos padrões *Facade*, *Observer* e *Repository*. Cada um dos supracitados padrões será brevemente explicado no contexto do projeto, junto do seu diagrama UML com as classes e interfaces utilizadas.

8.1 Facade

O padrão de projeto *Facade* é comumente utilizado para abstrair grandes trechos de código (muitas vezes necessários em execuções sequenciais) em apenas uma chamada de método, deixando invisível ao cliente que o consome quais as atividades lá realizadas. Muitas das implementações do *Facade* consistem na criação de um método com nome indicando um comando (*private void realizarCalculo(objeto))*. Dessa forma, o cliente que consumir o método para a realização do cálculo precisará apenas fornecer o dado para a função e aguardar seu resultado. Bibliotecas e *frameworks* trabalham nessa mesma linha lógica, fornecendo apenas uma classe com toda a implementação e totalmente abstrata ao consumidor. Esse padrão é utilizado pelo *front-end* da aplicação ao estabelecer comunicação com o *back-end*, através de uma interface comum com os endereços de acesso à API, onde o cliente sabe apenas quais os métodos chamados e os objetos necessários, não tendo acesso a quaisquer implementações a nível de protocolo HTTP/HTTPS, se há um *token* de autenticação, quais os *headers* necessários ao estabelecer conexão ou qual o endereço está sendo utilizado, se é para o ambiente local, ambiente de desenvolvimento ou de produção. A Figura 5 ilustra a forma atual como o *front-end* realiza a implementação do *Facade.* Utilizamos a biblioteca AXIOS para realizar as

abstrações de chamadas HTTP/HTTPS para o *back-end.* A lógica de geração das chamadas, modificação de cabeçalhos e parâmetros é feita por cada um dos serviços (Aluno.js, Time.js e Avaliacao.js). O módulo de *view* só consome os métodos disponíveis pelos serviços, informando os dados necessários através de objetos JSON.

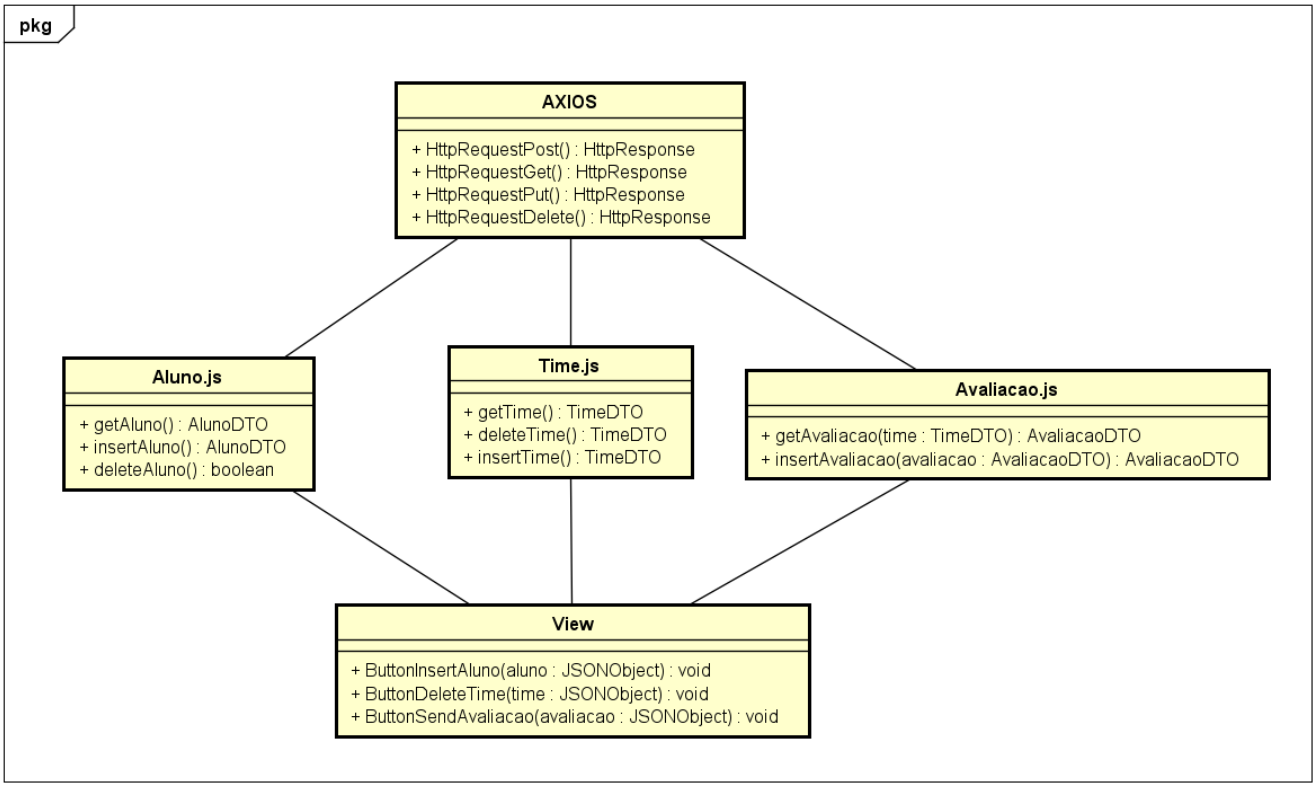

Figura 5: Diagrama do padrão Facade no front-end

8.2 Observer

O padrão *Observer* é implementado no nível do *back-end*, sendo aplicado no momento de exclusão de um aluno do sistema. O agente responsável por observar os eventos que envolvam a remoção de um aluno é também responsável por iniciar o método que remove esse mesmo aluno de um time. Utilizamos para isso uma lista de eventos que é passado ao *observer*, ficando este encarregado de atender as demandas dessa lista e agir de acordo. Na situação apresentada é a remoção de um aluno e logo depois sua remoção de um time. A Figura 6 ilustra o diagrama UML para a implementação desse padrão.

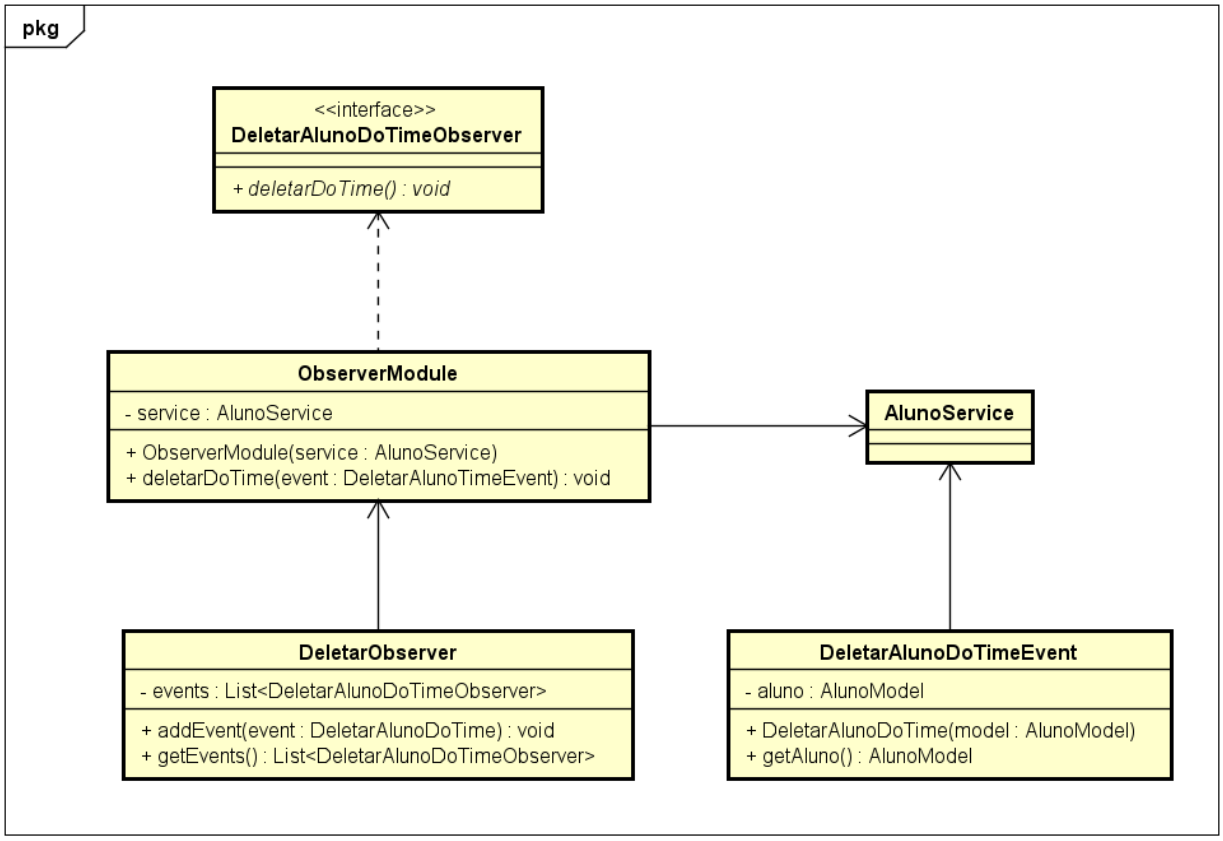

Figura 6: Padrão Observer para deletar um aluno

8.3 Repository

Para o padrão *Repository*, criamos uma interface *Repository* para cada entidade do nosso projeto: Aluno, Avaliação, Avaliador e Time. Essas interfaces fazem a mediação entre o domínio e as camadas de mapeamento de dados, agindo como uma coleção de objetos de domínio em memória. O repositório encapsula os objetos persistidos em um armazenamento de dados e as operações que são realizadas em cima dos mesmos. Dessa maneira, essa interface faz com que a lógica de negócio não tenha acesso aos detalhes da lógica de acesso a dados, abstraindo esses detalhes. Essa implementação garante que as entidades de domínio e a lógica de acesso aos dados só se comuniquem por meio das interfaces, tornando a manutenção de código mais fácil, evitando a duplicação de código, facilitando a realização de testes unitários, evitando dependências do banco de dados na lógica de negócio e deixando as responsabilidades das classes mais definidas.
A Figura 7 exemplifica a aplicação do *Repository* com uma das entidades.

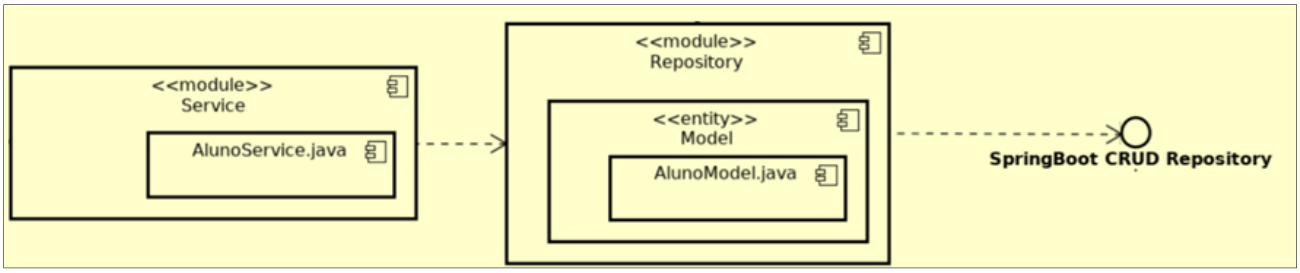

Figura 7: Padrão Repository para o acesso ao banco de dados

9. Diagramas de sequência

Apresentamos nessa seção os diagramas de sequência desenvolvidos para validar as atividades de criação de um novo time e avaliação de um time para eleger o vencedor da competição. No diagrama de sequência para avaliar um time não foram consideradas as opções de verificar o resultado da competição.

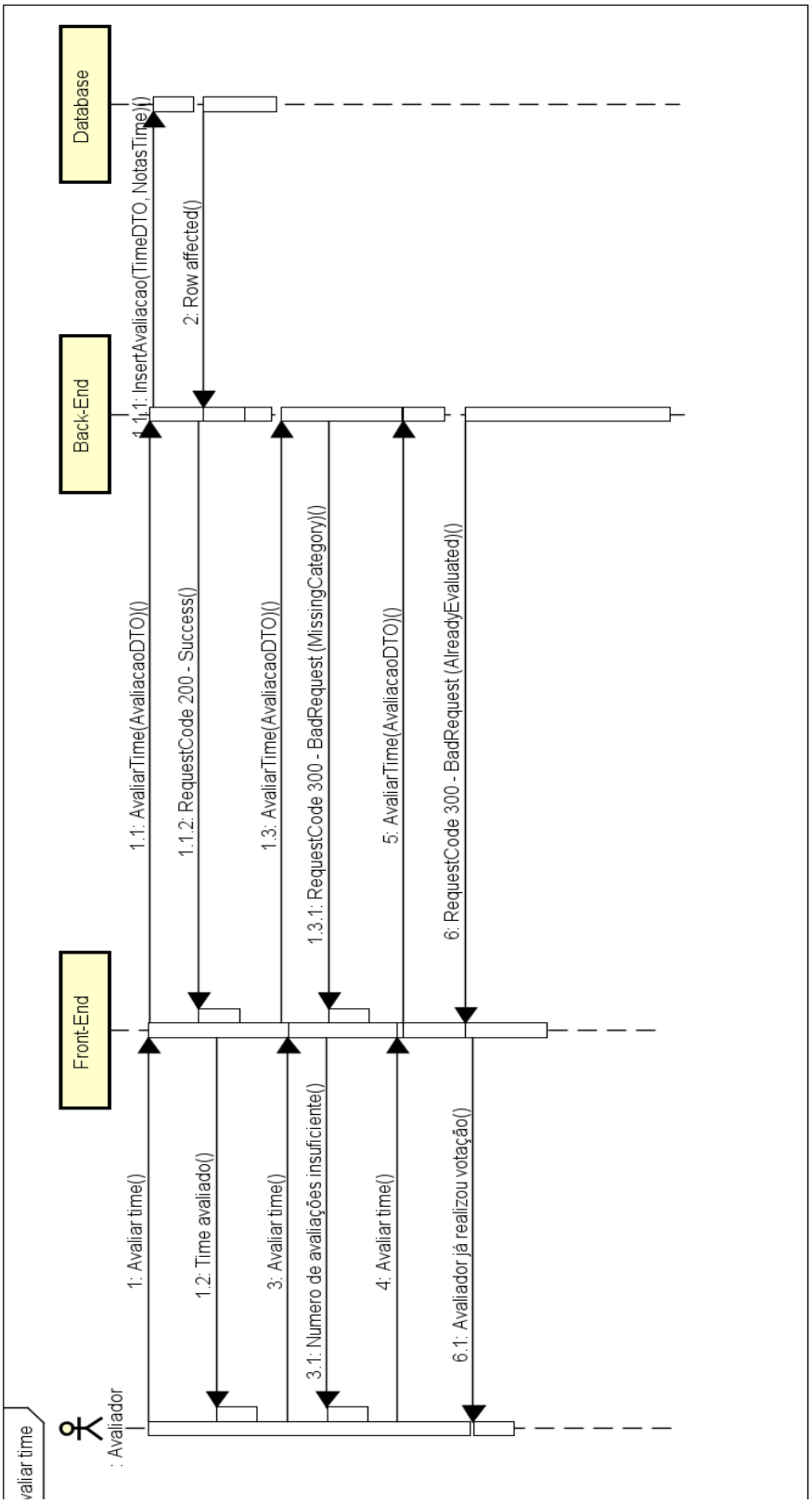

Figura 8: Diagrama de sequência para avaliar um time

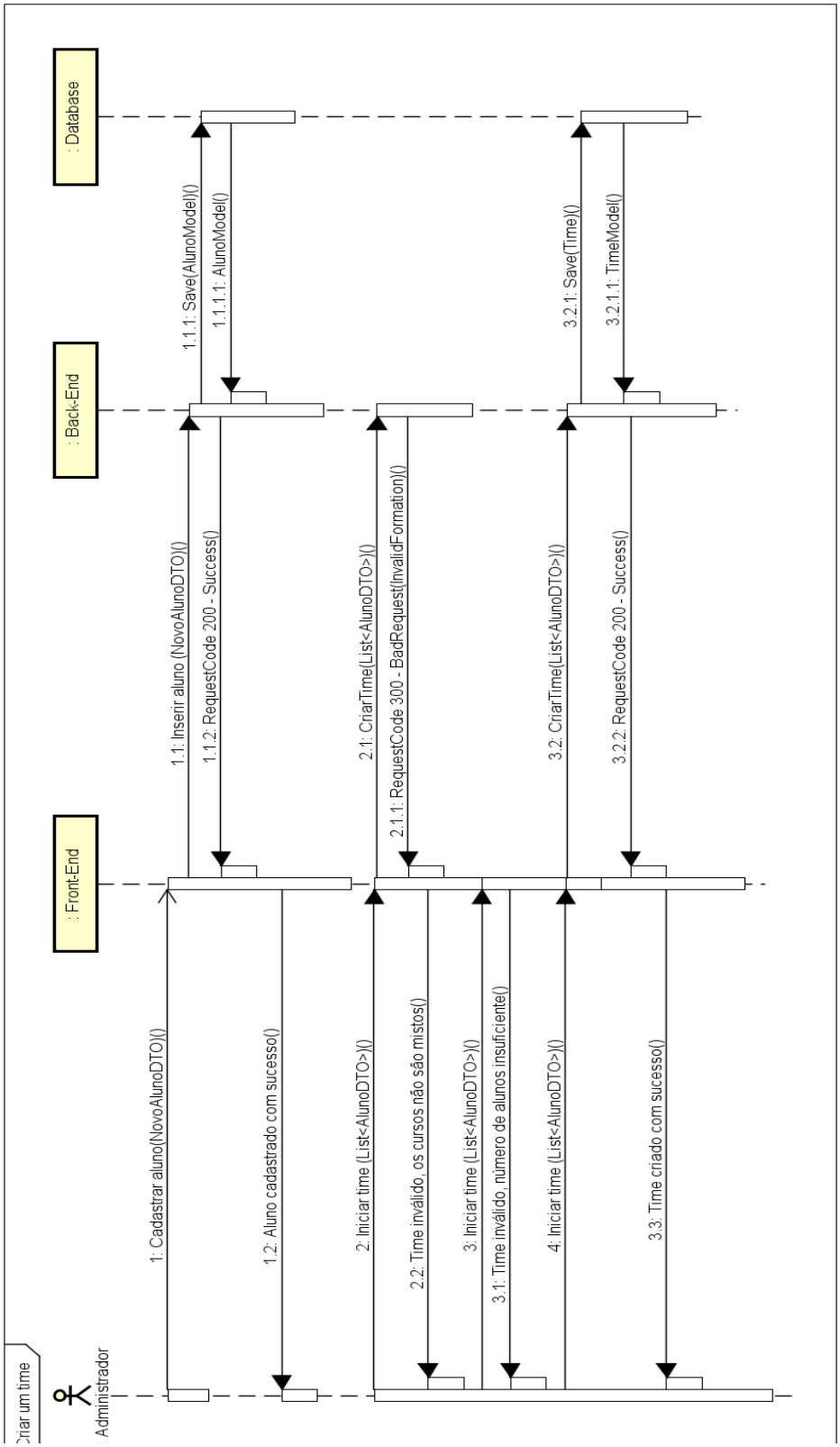

Figura 9: Diagrama de sequência para criar um time