

# **DropoutGuard: Projeto Detalhado**

**Eduardo Fillipe da Silva Reis<sup>1</sup>**

<sup>1</sup>Departamento de Computação – Universidade Federal de Sergipe (UFS)  
São Cristóvão – SE – Brasil

`eduardo.reis@dcomp.ufs.br`

**Abstract.** This document aims to detail the specification of the decision support system DropoutGuard. The class and sequence diagrams of the domain module related to the management of the system's knowledge base with its main operations will be presented.

**Resumo.** Este documento visa detalhar a especificação do sistema de apoio à decisão DropoutGuard. Serão apresentados os diagramas de classes e sequência do módulo de domínio relacionado ao gerenciamento da base de conhecimento do sistema com suas principais operações.

## **1. Introdução**

A evasão no ensino superior é tema de debate em todo o mundo e que acontece tanto em países desenvolvidos quanto em desenvolvimento, segundo dados do NCES [NCES 2020]. Além disso, é um problema que atinge instituições de ensino públicas e privadas, mostrando-se um obstáculo a ser superado por toda a sociedade.

É necessário as instituições de ensino atuem de forma ativa afim de reduzir os níveis de evasão no sistema educacional brasileiro. Assim, uma das possíveis abordagens é investir em estratégias que antecipem o problema, identificando individualmente quais estudantes têm maior tendência a abandonar o curso ou a instituição, antes que isso ocorra.

Assim, este projeto objetiva detalhar a especificação da arquitetura sugerida para o sistema DropoutGuard, com enfoque nas operações voltadas à base de conhecimento. Será apresentado o diagrama de classes, sequência e pacotes para as operações de atualização da base de conhecimento do sistema.

## **2. Especificação Detalhada**

Esta seção tem como objetivo definir em detalhes o caso de uso "Atualizar Base de Conhecimento". Dado que o sistema foi dividido em dois contêineres, cada um com responsabilidades distintas e comunicando-se de forma assíncrona, a documentação a seguir também será segmentada em duas partes, correspondentes a cada contêiner.

### **2.1. Solicitar Atualização base de conhecimento: DropoutGuard Api**

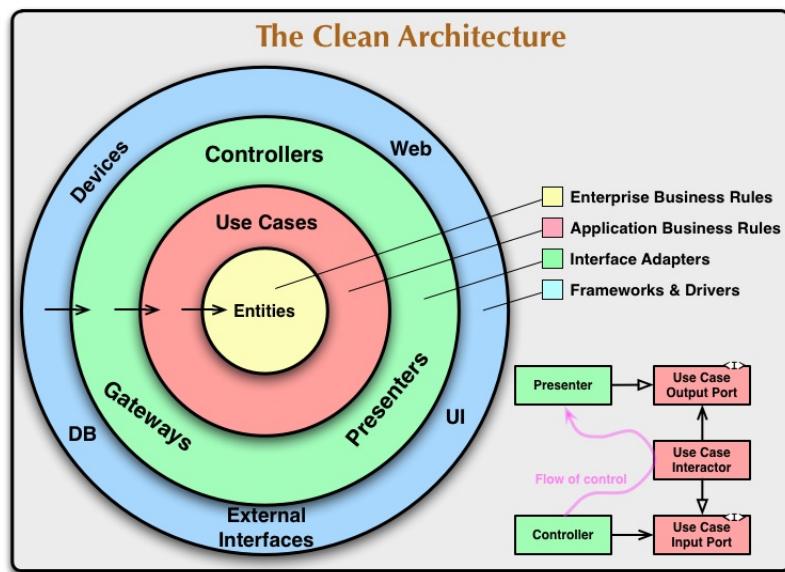
#### **2.1.1. Diagrama de Classes**

A Figura 2 mostra o diagrama de classes para a primeira etapa da operação *Atualizar Base de Conhecimento*, apresentada anteriormente no Diagrama de Casos de Uso. O diagrama é dividido em três grandes pacotes: *application*, *domain* e *infrastructure*. O

pacote *application* concentra os casos de uso da aplicação e é subdividido em pacotes que representam os subdomínios da aplicação, como *knowledgedatabase* e *report*. O pacote *domain* é responsável por manter as entidades do sistema e as interfaces e serviços necessários para que os casos de uso as manipulem. Já o pacote *infrastructure* cuida dos detalhes da aplicação, como as implementações concretas das interfaces do pacote *domain* e as configurações relacionadas às tecnologias empregadas nessas implementações.

A estrutura de classes, seguindo o Princípio da Inversão de Dependências [Martin 2017], respeita a Regra da Dependência [Martin 2017], ou seja, as dependências são sempre direcionadas para o núcleo do domínio do problema. Portanto, no diagrama, não há qualquer dependência ou associação que se estenda de uma camada interna para uma camada externa da aplicação, conforme a Figura 1. Em outras palavras, não há dependências que partem do pacote *domain* para os pacotes *infrastructure* ou *application*.

**Figura 1. Regra das dependências da Arquitetura Limpa de [Martin 2017]**



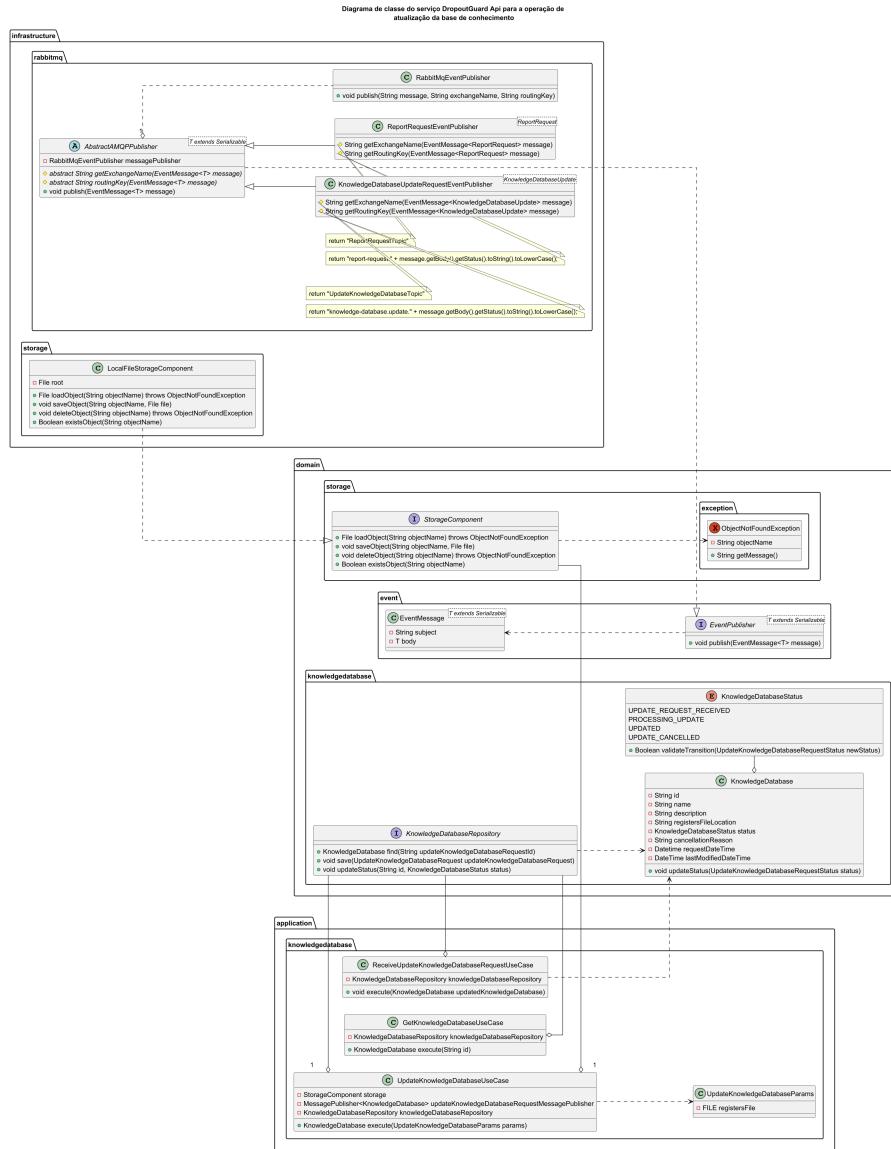
Como já mencionado, o objetivo deste caso de uso é atualizar a base de conhecimento utilizada pelos modelos de suporte à decisão disponíveis no sistema. O contrato dessa operação é estabelecido na classe *UpdateKnowledgeDatabaseUseCase*, localizada no pacote *application*. Esse caso de uso recebe a lista de matrículas de alunos, que será utilizada para preencher a base de conhecimento, salvando-a em um sistema de arquivos que a torne disponível para o módulo IntelligenceHub e notificando sobre a chegada da solicitação em um tópico de domínio específico.

O processamento da solicitação é realizado de forma assíncrona pelo módulo *IntelligenceHub*, que será descrito nas próximas seções. Após o término do processamento, o *IntelligenceHub* informa sobre a conclusão da operação emitindo um evento. Esse evento é capturado por uma fila e tratado pelo caso de uso *ReceiveUpdateKnowledge-DatabaseUseCase*, que finaliza o processo atualizando a base de dados local.

Durante o processamento no *IntelligenceHub*, clientes do DropoutGuardAPI podem querer consultar o status da solicitação enviada anteriormente. Isso pode ser feito

utilizando o caso de uso *GetKnowledgeDatabaseUseCase*.

**Figura 2. Diagrama de classes para o módulo Dropout Guard API**



### 2.1.2. Diagrama de Sequência

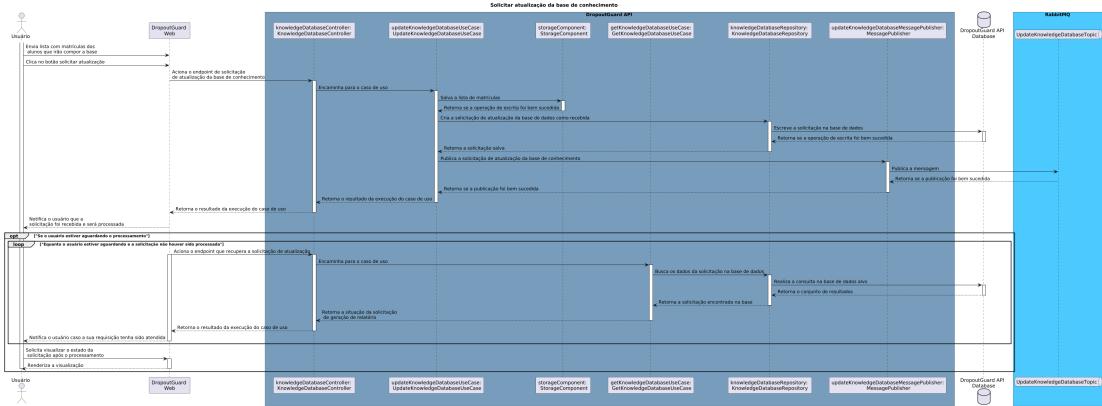
A Figura 3 ilustra como se dão as relações entre as classes da Figura 2 e os demais componentes do sistema ao longo do tempo.

## 2.2. Processar solicitação da base de conhecimento: Intelligence Hub

### 2.2.1. Diagrama de Classes

O diagrama da Figura 4 mostra as classes e relações necessárias para realizar o processamento da atualização da base de conhecimento do sistema. O diagrama possui uma estrutura de pacotes semelhante à da Figura 2, com três pacotes principais: *application*, *do-*

**Figura 3. Diagrama de sequência para o módulo DropoutGuard API**



*main* e *infrastructure*. Além disso, como ambos publicam em canais AMQP/RabbitMQ, as implementações do componente *EventPublisher* são idênticas.

O IntelligenceHub é responsável por processar as solicitações de atualização da base de conhecimento, realizando todos os procedimentos necessários para a obtenção dos dados. Atualmente, há apenas uma maneira de acessar esses dados, que é extraíndo-os dos currículos dos estudantes. Portanto, para cada matrícula de estudante, o processo consiste em acessar o sistema acadêmico, baixar o histórico escolar e extrair as informações necessárias. O trabalho de atualização foi dividido em tarefas, onde cada tarefa representa o processamento de uma matrícula escolar. No diagrama de classes, essas tarefas são enfileiradas numa fila de tarefas pelo caso de uso *ExecuteKnowledgeDatabaseUpdateUseCase*, e cada matrícula é processada individualmente e, possivelmente, em paralelo por instâncias da classe *UpdateKnowledgeDatabaseWorker*. Ao final do processamento, um evento é enviado informando sobre a conclusão do processamento. O processo completo é descrito no diagrama de sequência da Figura 5.

## 2.2.2. Diagrama de Sequência

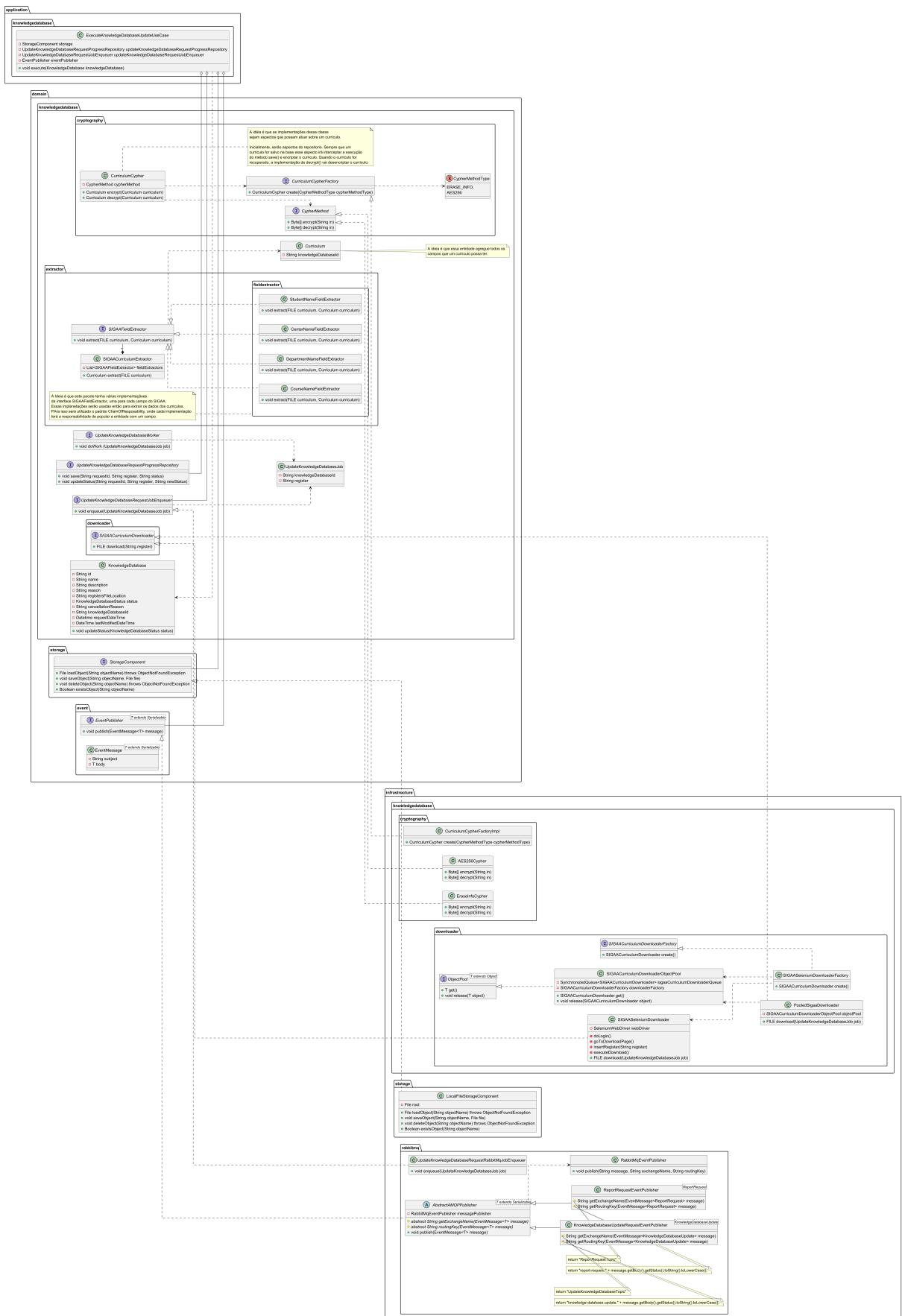
A Figura 5 apresenta o diagrama de sequência que detalha a ordem das interações entre as classes descritas na Figura 4.

## 2.3. Receber o resultado da atualização da base de Conhecimento: DropoutGuard API

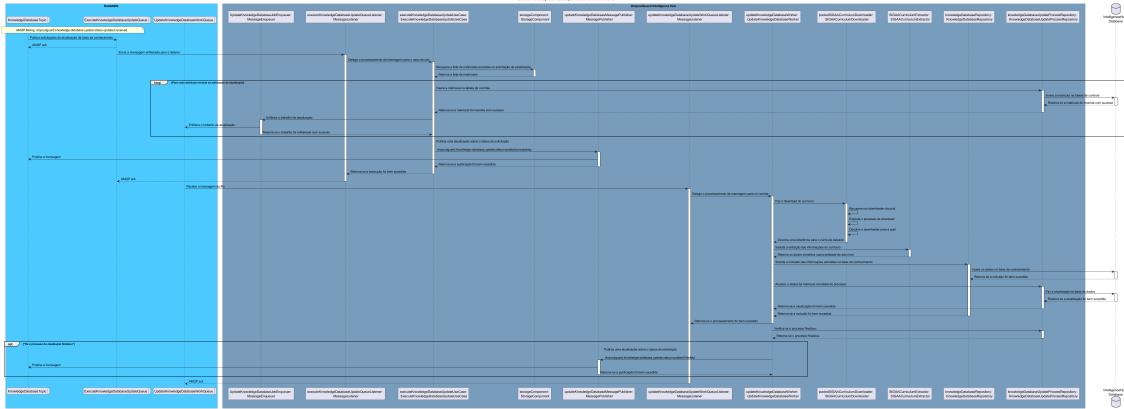
### 2.3.1. Diagrama de Classe

Após o término do processamento, o módulo *IntelligenceHub* emite um evento informando sobre a conclusão da atualização da base de conhecimento. Esse evento é consumido pelo *DropoutGuard API*, que atualiza sua base de dados local. As classes envolvidas nessa operação e suas respectivas relações são apresentadas na Figura 2. A classe responsável por realizar o procedimento é o caso de uso *ReceiveUpdateKnowledgeDatabaseUseCase*.

**Figura 4. Diagrama de classe para o módulo IntelligenceHub**



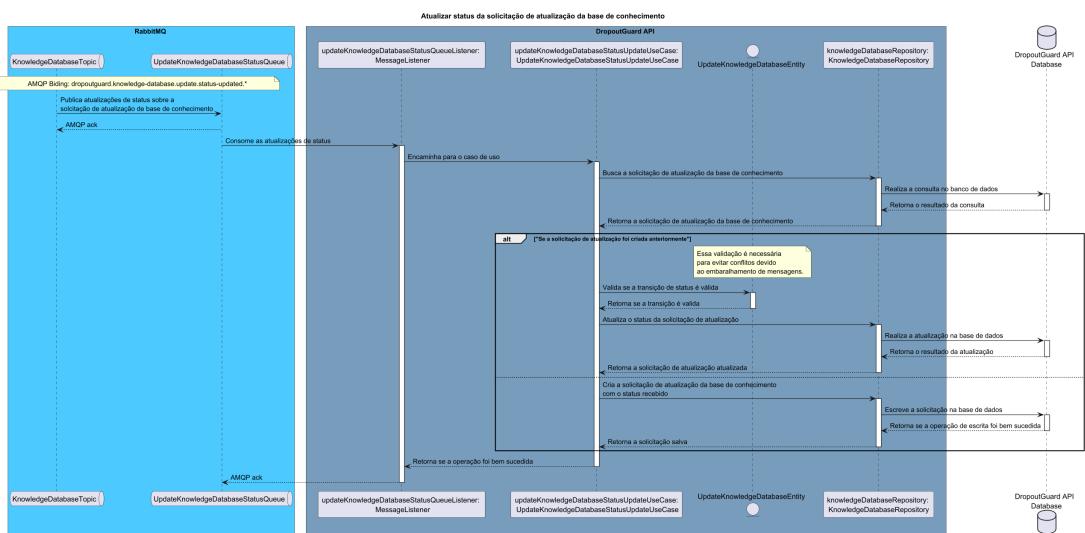
**Figura 5. Diagrama de Sequência para Processamento de Solicitação da base de conhecimento**



### 2.3.2. Diagrama de Sequência

A Figura 6 descreve de maneira detalhada o passo a passo realizado pelo módulo *DropoutGuard API* para receber atualizações sobre o processo de atualização da base de conhecimento do sistema.

**Figura 6. Diagrama de sequência da operação Receber atualizações sobre o processo de Atualização da base de conhecimento**



### 2.4. Padrões de Projeto Aplicados

1. **Builder[Gamma et al. 1994]:** Para a criação de objetos complexos e com muitos parâmetros, como o agregado *Curriculum*, evitando o uso de construtores com muitos parâmetros. Em geral, esse padrão produz implementações repetitivas (*boilerplate*), por isso a estrutura desse padrão não foi delineada. Entretanto, é importante que, durante o desenvolvimento da aplicação, esse padrão seja explorado pelos motivos supracitados.

2. **Chain of Responsibility[Gamma et al. 1994]:** Aplicado na classe *SIGAACCurriculumExtractor*. A classe possui uma lista de extratores. Durante a execução, cada extrator será responsável por extrair uma informação do currículo e inseri-la no agregado *Curriculum*.
3. **Bridge[Gamma et al. 1994]:** Aplicado na implementação concreta da interface *EventPublisher*. Tendo como abstração a classe *AbstractAMQPPublisher*, como implementação a classe *RabbitMQPublisher* e como abstrações refinadas as classes *KnowledgeArticleEventMessagePublisher* e *ReportRequestEventPublisher*.
4. **Factory Method[Gamma et al. 1994]:** Utilizado, por exemplo, no pacote *infrastructure.downloaders* para criar objetos na pool de objetos, quando necessário.
5. **Singleton[Gamma et al. 1994]:** Devido à natureza da aplicação, em geral, as classes que não são entidades do negócio podem possuir o ciclo de vida de um Singleton, não necessitando ser criadas a cada requisição efetuada pelo usuário. Apesar de não ser explícito nos diagramas de classe, é importante levar isso em conta na implementação do projeto, principalmente se forem utilizadas ferramentas de Injeção de Dependências, evitando o consumo desnecessário de memória pela aplicação durante picos de utilização.
6. **Repository & Gateway[Richardson 2018]:** Para romper as fronteiras do domínio do sistema. A aplicação utiliza Repositórios para acessar dados dentro do seu domínio e Gateways para realizar operações fora do seu domínio.
7. **Object Pooling[Kircher and Jain 2002]:** Na classe *SIGAAPooledDownloader* é utilizado este padrão, pois a criação e a manutenção dos estados dos *downloaders* são custosas. Dessa forma, diversos workers podem compartilhar esses *downloaders* sem a necessidade de acoplar seus ciclos de vida.

### 3. Planos de Teste

#### 3.1. Casos de testes de Unidade

- **KnowledgeDatabaseTest**
  1. Deve alterar o status para *UPDATE\_CANCELLED* ao cancelar a atualização, verificando também a razão de cancelamento.
  2. Deve lançar uma exceção se a transição para cancelar a atualização não for permitida.
  3. Deve alterar o status para um status final válido (*UPDATED*, *UPDATE\_ERROR* ou *UPDATE\_CANCELLED*) ao finalizar a atualização.
  4. Deve lançar uma exceção se o status final ao tentar finalizar for inválido (*UPDATING*).
  5. Deve lançar uma exceção se a transição para um status final ao finalizar não for permitida após cancelamento.
  6. Deve alterar o status para "UPDATING" ao iniciar uma atualização.
  7. Deve lançar uma exceção se a transição para iniciar uma atualização não for permitida.
- **ReceiveKnowledgeDatabaseStatusUpdateUseCaseTest**
  1. Não deve salvar quando o status da base de conhecimento for o mesmo.
  2. Não deve salvar quando a transição de estado para a base de conhecimento não for permitida.
  3. Deve salvar quando a transição de estado para a base de conhecimento for permitida.

- **ExecuteKnowledgeDatabaseUpdateUseCaseTest**
  1. Deve executar a atualização com sucesso, salvando os trabalhos no repositório, enfileirando-os e publicando um evento com o status "UPDATING".
  2. Deve publicar um evento com o status "UPDATE\_ERROR" em caso de qualquer exceção durante o carregamento do arquivo de registros.
  3. Deve lançar uma exceção se falhar ao publicar um evento após um erro durante o processo de atualização.
- **CurriculumCypherTest**
  1. Deve criptografar um currículo utilizando o método de criptografia fornecido.
  2. Deve descriptografar um currículo, restaurando as informações criptografadas.
- **KnowledgeDatabaseUpdateWorkerTest**
  1. Deve ignorar o processamento quando o trabalho de atualização não for encontrado.
  2. Deve ignorar o processamento quando o trabalho de atualização já tiver sido processado.
  3. Deve atualizar o trabalho com erro quando o currículo não for encontrado.
  4. Deve atualizar o trabalho com erro quando o currículo for inválido.
  5. Deve enviar eventos quando todos os trabalhos de atualização tiverem sido processados.

### 3.2. Casos de testes de Integração

- **GetKnowledgeDatabaseTest**
  1. Deve recuperar corretamente uma base de conhecimento existente e validar o resultado retornado.
  2. Deve lançar uma exceção ao tentar recuperar uma base de conhecimento inexistente.
- **ReceiveKnowledgeDatabaseStatusUpdateIntegrationTest**
  1. Deve verificar se a atualização de status de uma base de conhecimento é recebida corretamente, garantindo a persistência do novo status no repositório.
- **RequestKnowledgeDatabaseUpdateUseCaseTest**
  1. Deve validar o fluxo de solicitação de atualização com sucesso e verificar:
    - A existência física do arquivo de registro gerado.
    - A persistência da atualização no repositório de base de conhecimento.
    - O envio de mensagens para a fila com os dados corretos e o status esperado.
  2. Deve lançar uma *NullPointerException* ao tentar executar a lógica principal com uma entrada nula.
- **SIGAACurriculumExtractorTest**
  1. Deve extrair corretamente os campos do currículo de um arquivo fornecido, validando os campos extraídos.
- **Extratores de campos**
  1. Testar se cada extrator está extraíndo os campos do histórico corretamente.

- **CryptographyAspectTest**

1. Deve criptografar corretamente o currículo antes de salvá-lo no repositório, validando:
  - A execução do método de criptografia no aspecto.
  - A equivalência entre o objeto salvo e o original (considerando as diferenças de criptografia).
  - A recuperação correta do currículo criptografado no repositório.

- **CurriculumJpaServiceTest**

1. Deve localizar um currículo existente por seu identificador, validando que o currículo recuperado corresponde ao salvo.
2. Deve retornar vazio ao tentar localizar um currículo utilizando um identificador inexistente.
3. Deve salvar corretamente um currículo no repositório, verificando que ele pode ser recuperado com sucesso.
4. Deve reutilizar um estudante existente no sistema ao salvar um novo currículo onde o documento do estudante já existe, validando que o identificador do estudante permanece o mesmo.

- **RabbitMQPublisherIntegrationTest**

1. Deve publicar uma mensagem no RabbitMQ, verificando:
  - Que o método de envio do *RabbitTemplate* foi chamado uma vez com os parâmetros corretos.
  - Que a mensagem publicada é corretamente consumida da fila, preservando seu conteúdo.

- **KnowledgeDatabaseUpdateWorkerTest**

1. Deve executar corretamente a atualização de um currículo na base de dados de conhecimento, verificando:
  - Que o currículo salvo no repositório possui os mesmos valores esperados do arquivo armazenado.
  - Que o identificador da base de conhecimento no currículo salvo está correto.
  - Que uma mensagem de evento de atualização é publicada no RabbitMQ após a conclusão da tarefa.
  - Que a mensagem contém as informações corretas sobre o identificador e o status da base de dados de conhecimento.

## 4. Considerações Finais

Conforme as documentações apresentadas, este projeto especificou a arquitetura de um sistema de software capaz de atender às necessidades funcionais e não funcionais do módulo responsável por gerenciar a base de conhecimento do sistema.

Também é possível afirmar que as fronteiras do domínio do problema foram respeitadas no projeto detalhado do módulo descrito neste documento. Através do uso de inversão de dependências, foi possível cruzar tais fronteiras, garantindo a capacidade de expansão do sistema e sua manutenibilidade. Além disso, essa estratégia propiciou o desenvolvimento dos testes no protótipo apresentado, através do uso de dublês de testes que puderam ser facilmente injetados em objetos de mais alto nível.

Assim como planejado na especificação arquitetural, as tecnologias sugeridas não foram acopladas à visão detalhada do sistema. Dessa forma, componentes de baixo nível, como bancos de dados e protocolos de comunicação, foram tratados como detalhes de implementação, permitindo postergar tais decisões.

Com relação ao formato em que a documentação foi apresentada, é possível afirmar que o modelo C4 [Ford and Richards 2019] facilita a manutenção e o entendimento de alto nível da arquitetura ao custo de tornar implícitos detalhes arquiteturais de mais baixo nível. Por exemplo, as relações assíncronas entre os módulos *DropoutGuard API* e *Intelligence Hub* podem ser observadas já nos primeiros diagramas. Entretanto, a capacidade de processamento paralelo que as filas de trabalho propiciam fica implícita no modelo de comunicação.

Esse tipo de detalhe pode passar despercebido por programadores que não possuam conhecimento suficiente sobre os padrões arquiteturais utilizados, podendo levar a erros ou à não exploração total das capacidades que a arquitetura oferece. Dessa forma, é importante incorporar novas documentações de acordo com as necessidades do time de desenvolvimento, mantendo esse conjunto de documentos atualizado conforme o projeto se desenvolve.

## Referências

- Ford, N. and Richards, M. (2019). *Fundamentals of Software Architecture: An Engineering Approach*. O'Reilly Media, Sebastopol, CA.
- Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional.
- Kircher, M. and Jain, P. (2002). Pooling pattern. In *EuroPLoP 2002*.
- Martin, R. C. (2017). *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Pearson, Boston, MA.
- NCES (2020). What are the dropout rates of high school students?
- Richardson, C. (2018). *Microservices Patterns: With Examples in Java*. Manning Publications, Shelter Island, NY.