

Lásaro Curtinaz Dumer

Marcelo Schmitt Laser

**AutoREST**  
**Uma Ferramenta de**  
**Geração Automática de REST APIs**

Porto Alegre, Rio Grande do Sul, Brasil

2016

Lásaro Curtinaz Dumer  
Marcelo Schmitt Laser

# **AutoREST**

## **Uma Ferramenta de Geração Automática de REST APIs**

Relatório Final de Trabalho de Conclusão  
I, apresentado como requisito parcial para  
obtenção do grau de Bacharel em Ciência  
da Computação na Pontifícia Universidade  
Católica do Rio Grande do Sul

Pontifícia Universidade Católica do Rio Grande do Sul – PUCRS

Faculdade de Informática

Curso de Bacharelado em Ciência da Computação

Orientador: Eduardo Henrique P. de Arruda

Porto Alegre, Rio Grande do Sul, Brasil

2016

## Lista de ilustrações

## Lista de tabelas

# Lista de abreviaturas e siglas

REST	Representational State Transfer
BNF	Backus-Naur Form
UML	Unified Modeling Language
OMG	Object Management Group
W3C	World Wide Web Consortium
URI	Uniform Resource Identifier
CASE	Computer-aided software engineering
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
PUCRS	Pontifícia Universidade Católica do Rio Grande do Sul
CBSE	Component-Based Software Engineering
GP	Generative Programming
SPL	Software Product Line
MDD	Model-Driven Development
CRUD	Create, Read, Update, Delete
SGBD	Sistema Gerenciador de Banco de Dados
CMR	Configuration Model Reader
PFIS	Processing-Friendly Intermediate Structure

# Sumário

# 1 Introdução

Uma das principais preocupações da Engenharia de Software atualmente é a confiabilidade de sistemas e processos de desenvolvimento de software (??) (??). Definida como a prevenção de falhas na concepção, projeto e construção de software, além da capacidade de recuperação de falhas em operação e performance, a confiabilidade de sistemas é dita por Ross como uma preocupação "tardia" da Engenharia de Software, muitas vezes ignorada em favor de eficiência.

??) apresenta diversas razões para esta preocupação, dentre elas o crescimento dramático do tamanho e complexidade de sistemas computacionais e a forma como sistemas computacionais permeiam quase todas as áreas da sociedade. Como forma de medir a confiabilidade de sistemas, diversos autores (??)(??)(??) utilizam o quantificar *Mean Time to Failure* (MTTF), ou tempo médio até falha em português. Esta medida é, em todos os casos citados acima, realizada através do teste de software.

Independentemente das abordagens utilizadas, duas técnicas se destacam nos trabalhos de diversos autores (??)(??)(??)(??)(??)(??) como uma forma de diminuir consideravelmente o esforço de desenvolvimento e aumentar a confiabilidade de sistemas de software: o reuso de componentes de software testados e maduros, e; a geração automática de código a partir de modelos conceituais.

??) apresenta preocupação sobre o quão atrasada é a Engenharia de Software em buscar melhoras na confiabilidade e produtividade de software. O autor propõe a geração de código automática, a partir de modelos conceituais, como sendo uma mudança fundamental nos paradigmas de desenvolvimento de software, com o potencial de alterar a Engenharia de Software de formas que não são vistas desde a invenção de compiladores a mais de cinquenta anos.

Já ??) citam dentre as regras de restrição de um componente de software a composição por componentes de terceiros, ressaltando a importância da utilização de componentes de software maduros e testados na composição de sistemas confiáveis. ??) e ??) reforçam ainda mais a necessidade da criação de grupos de componentes em comum para que a evolução de cada componente seja propagada dentre diversos sistemas.

Os autores citados propõe várias abordagens e tecnologias para a prática dos conceitos propostos nas áreas de geração automática de código e reuso de componentes de software. Destas, se destacam neste estudo a Engenharia de Software Baseada em Componentes (CBSE - Component-Based Software Engineering) (??), Programação Generativa (GP - Generative Programming) (??) e o Desenvolvimento Dirigido por Modelos (MDD - Model-Driven Development) (??).

CBSE se destaca pela identificação de características de software e sua implementação na forma de componentes reutilizáveis, possibilitando a geração de sistemas a partir da composição por partes já mais maduras e confiáveis do que a construção de um sistema inteiramente novo ([?]). ([?]) citam a facilidade de manutenção de sistemas complexos baseados em CBSE e seu alto grau de uso em sistemas Web, e ([?]) mencionam que o grau de reuso possibilitado por práticas de CBSE varia desde o reuso de grandes componentes englobando diversas funcionalidades, até o reuso de pequenos blocos de código.

MDD se destaca pelo seu foco em uma documentação minuciosa de um produto de software, tendo a implementação em segundo plano e preferencialmente de forma automatizada, buscando permitir melhor visualização e manutenção do produto ([?]). ([?]) citam a importância de uma documentação completa de sistemas de software que possibilite o reuso de partes de sistemas ou até sistemas inteiros em vista do alto grau de compreensão de desenvolvedores que é obtido através do foco em documentação.

GP busca minimizar os trabalhos de programação através da automação de geração de código, realizada pela identificação e formalização das características comuns entre sistemas e a criação de geradores de componentes que os compõe a partir de subcomponentes pré-determinados ([?]). ([?]) menciona a abordagem GP como sendo o futuro da engenharia de software, ressaltando a importância de abordagens abrangentes para a geração de sistemas ao invés do uso de soluções *ad hoc*.

O estudo das abordagens de MDD e GP, além de publicações específicas demonstrando a aplicação de abordagens baseadas em modelos na área de desenvolvimento Web ([?])([?]), permitiu concluir que é possível a geração de APIs de acesso a estruturas de dados a partir de modelos conceituais que sirvam como modelos de configuração para a geração destas APIs. A aplicação de técnicas de CBSE permite a criação de subcomponentes reutilizáveis na geração destas APIs, na forma de blocos de código parametrizáveis, aumentando consideravelmente a confiabilidade dos sistemas gerados e a produtividade no seu processo de geração.

No que concerne às diferentes abordagens arquiteturais para o desenvolvimento de software, podemos destacar a arquitetura de cliente-servidor em ([?]) e a de sistemas em camadas de ([?]). Também enquadra-se dentro de arquiteturas a REST (*REpresentational State Transfer*) ([?]) que é uma arquitetura focada na comunicação de rede de um sistema, sendo uma das bases da Web.

O objetivo principal deste trabalho é propor: um paradigma baseado nas metodologias citadas acima (MDD, GP, CBSE) para a modelagem conceitual de APIs REST; métodos e algoritmos para a conversão destes modelos em formatos que permitam facilidade de compreensão humana e processamento por máquinas, e; o desenvolvimento de um protótipo de sistema de software que aplique este paradigma para a geração automática do código de APIs REST a partir de um esquema conceitual representado em um diagrama



de classes da UML. As classes e seus atributos serão complementados com anotações (*tags*) em JSON Schema a fim de permitir maior refinamento destes esquemas conceituais; estas servirão para representar restrições de integridade das estruturas de dados e permitir a geração de APIs com acesso a bases de dados de persistência. As APIs REST geradas suportam as operações básicas de manutenção dos documentos, comumente referidas como CRUD (*Create, Read, Update, Delete*), através da composição de blocos de código parametrizáveis representando os principais métodos HTTP.

O trabalho está organizado da seguinte forma: No capítulo ?? são apresentados os conceitos básicos sobre determinadas metodologias de desenvolvimento de software, seguidos de uma breve apresentação das linguagens UML e JSON. No capítulo ?? são apresentadas as arquiteturas Cliente/Servidor, REST e HTTP, que são a base das APIs que se busca gerar com a solução deste estudo. No capítulo ?? são apresentadas as tecnologias que foram utilizadas na solução do problema proposto.

No capítulo ?? é apresentada a solução proposta por este estudo, composta de uma arquitetura independente de tecnologia e a modelagem de uma ferramenta como prova de conceito. São apresentadas todas as regras de modelagem e de geração da ferramenta prova de conceito, além de uma apresentação da ferramenta e de seu funcionamento. O capítulo ?? apresenta a conclusão deste trabalho, além de trabalhos relacionados e a descrição de possíveis trabalhos futuros. Para a organização das contribuições deste trabalho no último capítulo, e para a orientação do leitor durante o estudo deste volume, são propostas as seguintes questões de pesquisa que são representativas dos desafios encontrados:

- QP1. Existe viabilidade na automação da geração de uma API REST utilizando uma abordagem baseada em MDD?
- QP2. Quais os elementos necessários para a representação completa de uma API REST em um modelo gráfico para que este possa ser utilizado como modelo de configuração em uma abordagem MDD para a geração de uma API com todas as funcionalidades CRUD fundamentais?
- QP3. Existe a necessidade de criação de um modelo intermediário, a partir de um modelo de configuração, que sirva como artefato de entrada para um gerador de APIs REST?
- QP4. As APIs REST são suficientemente modularizáveis a ponto de permitir a geração automática de código a partir de blocos de código parametrizáveis? Qual a complexidade de se criar um compilador de APIs REST que não se utiliza de uma abordagem GP?
- QP5. É possível a geração automática de um banco de dados auxiliar à API REST?

- QP6. Diagramas de Classes UML são uma notação adequada para a modelagem de APIs REST?
- QP7. Arquivos JSON Schema são adequados para a representação de APIs REST?
- QP8. O SGBD MongoDB, a linguagem Node.js e a biblioteca Mongoose são adequadas como tecnologias para a geração automática de APIs REST?

## 2 Modelagem Estrutural de Software

Este capítulo tem como objetivo contextualizar as metodologias de desenvolvimento e as linguagens e formatos de modelagem utilizados para a modelagem estrutural da solução proposta neste projeto.

### 2.1 Metodologias de Desenvolvimento

Dentre as metodologias de desenvolvimento que tem como foco central, ou tem entre suas principais contribuições, a automação da geração de código e aumento da confiabilidade de sistemas, foram identificadas três metodologias que se destacam neste sentido e são pertinentes ao tema deste estudo.

#### 2.1.1 Engenharia de Software Baseada em Componentes

??) oferece uma excelente definição de CBSE, que é a que utilizamos neste trabalho (tradução livre):

Engenharia de Software Baseada em Componentes (CBSE) se preocupa com o desenvolvimento de sistemas de software a partir de partes reutilizáveis (componentes), o desenvolvimento de componentes, e a manutenção e aprimoramento de sistemas através da substituição ou customização de componentes.

Esta definição é concisa e deixa em aberto o significado exato de certos termos, como o próprio termo “componente”. Para a clarificação disto, temos a definição de componente apresentada por ??). Os autores iniciam sua definição citando a definição do *Oxford Advanced Learners Dictionary* (tradução livre): “Um componente é qualquer parte da qual algo é feito”.

Como esta definição é insuficiente para a limitação do escopo do termo componente para a engenharia de software, os autores prosseguem listando quatro itens para a definição precisa de um componente reutilizável:

- Conter interfaces especificadas contratualmente;
- Dependências contextuais completamente explícitas;
- Entregas independentes, e;
- Composição por componentes de terceiros.

Mesmo com esses limitadores, os autores ainda mantêm que podem existir interpretações diferentes de componentes em diferentes tipos de projetos e sistemas. Para definir mais precisamente o termo, eles seguem com os seguintes axiomas, originalmente apresentados por ??):

- Um componente é capaz de realizar uma tarefa isoladamente; *i.e.* sem a necessidade de ser composto com outros componentes;
- Componentes podem ser desenvolvidos independentemente um do outro, e;
- O propósito da composição é permitir a cooperação entre os componentes constituintes.

Outras definições de componentes, incluindo apresentações de como utilizar componentes no contexto de padrões de projeto, podem ser encontradas nos trabalhos de ??) e ??).

Podemos concluir que uma das contribuições de CBSE é a diminuição do esforço de desenvolvimento, aqui medido em tempo, custo monetário e alocação de recursos humanos, e o aumento da confiabilidade de sistemas, na medida que estes são compostos por componentes mais maduros e robustos. Por esta razão, esta metodologia de desenvolvimento se apresenta como uma possível solução ao problema tratado por este trabalho.

??) explicitam a diferença entre CBSE e outros métodos de Engenharia de Software que fazem uso de componentes (tradução livre):

Enquanto o objetivo primário da arquitetura de software e da engenharia de sistemas é entender o sistema através da sua divisão em componentes e da identificação de componentes como unidades compostas que expressam determinadas funções e propriedades, CBSE parte de dadas propriedades dos componentes e define o sistema utilizando estas propriedades.

Partindo-se desta definição, apesar da abordagem CBSE não ser apropriada para o processo que é proposto neste trabalho para a geração de APIs REST, ele é apropriada para a construção da solução AutoREST. Após a identificação de características comuns entre APIs REST, é possível a representação destas características em componentes no formato de blocos de código parametrizáveis, que podem servir como base na geração das APIs. Portanto, CBSE mostra-se como uma metodologia apropriada para a modelagem da solução AutoREST.

### 2.1.2 Desenvolvimento Dirigido por Modelos

O uso de modelos para a representação de abstrações dentro das diversas especialidades de Engenharia já é uma prática comum a milênios (??). Em seu livro, ??) afirmam

que o uso de modelos no processo de desenvolvimento de software é uma tradição antiga, que se tornou mais comum após a popularização da UML. Entretanto, os autores afirmam que na maioria dos projetos de software, modelos são usados apenas como documentação, e que a relação entre modelos e software implementado é "intencional, mas não formal".

Em contraste a esta afirmação, os autores introduzem o conceito de Desenvolvimento de Software Dirigido por Modelos (MDD), em que os modelos de um projeto de software são considerados tão importantes quanto o código implementado, com a implementação sendo parcial ou totalmente automatizada. De acordo com ??), métodos MDD tem um enorme potencial dentro da área de engenharia de software que ainda não foi alcançado, em grande parte, devido a dificuldades em introduzir estes novos métodos ao contexto de projetos já existentes (diminuindo assim a barreira de adoção).

Para Selic, a base técnica do conceito de MDD são tecnologias de automação, que ele divide em duas partes: a geração completa de programas a partir de modelos, ao invés de fragmentos, e; a verificação automática de modelos. O autor afirma que uma das dificuldades na adoção de MDD seria a dificuldade de realizar a geração automática de modelos a partir de código, o que causa a atual prática de muitas ferramentas, de gerar apenas fragmentos de código, ineficiente, visto que alterações no código gerado não serão traduzidas em alterações nos modelos; ou seja, o código não é rastreável. O autor então propõe que a atual disseminação de padrões, como os administrados pela OMG, possibilita a criação de ferramentas que realizam a geração completa de programas a partir de modelos.

Para que isso seja possível, Selic propõe cinco qualidades essenciais de modelos, não apenas de software, mas em qualquer domínio:

- A primeira qualidade definida pelo autor é *Abstração*. Um modelo deve ser uma representação reduzida daquilo que está sendo representado.
- A segunda é *Compreensibilidade*. Um modelo deve ser compreensível por um ser humano rapidamente, sem a necessidade do tempo e esforço necessários para se compreender, por exemplo, código implementado.
- A terceira qualidade é *Precisão*. Um modelo deve ser uma representação correta daquilo que está sendo representado.
- A quarta qualidade é *Preditividade*. Um engenheiro deve ser capaz de prever detalhes sobre seu objeto de estudo através do estudo do modelo.
- Por fim, um modelo deve ser *Econômico*. Criar um modelo deve ser significativamente menos dispendioso do que construir o sistema em si.

Levadas em consideração as definições apresentadas, ficam claras as vantagens do uso de técnicas MDD neste projeto. A proposta de geração de código implementado a partir de modelos estruturais se beneficia dos conceitos já fundamentados de MDD, especialmente no que se refere ao uso de padrões industriais e representatividade dos modelos utilizados.

### 2.1.3 Programação Generativa

Em seu livro, ??) propõe a idéia de que a engenharia de software se encontra mais de um século atrás de outras engenharias mais maduras, e a compara a uma fábrica de tendas, construindo soluções únicas a mão ao invés de utilizar uma metodologia madura de engenharia. Tendo isto em vista, eles propõe a idéia de programação generativa (tradução livre):

Programação Generativa é a criação de produtos de software a partir de componentes de uma maneira automática, ou seja, da maneira como outras indústrias tem produzido produtos mecânicos, eletrônicos e outros a décadas.

Para que esta proposta seja possível, Czarnecki e Eisenecker apresentam dois requisitos fundamentais: a noção de famílias de sistemas, ao invés de sistemas únicos, e; a composição automática da implementação de componentes utilizando geradores.

Os autores descrevem os passos para que estes requisitos sejam alcançados: 1) o projeto da implementação de componentes deve se encaixar em uma arquitetura de linha de produto; 2) um modelo de configuração deve ser criado descrevendo como traduzir requisitos abstratos em constelações de componentes, e; 3) este modelo de configuração deve ser implementado utilizando geradores.

A metodologia de programação generativa é apropriada a este estudo na medida que são definidos pequenos blocos de código parametrizáveis que possam ser utilizados para a construção do código final gerado pelo tradutor desenvolvido. Desta forma, preenchem-se os três requisitos listados por Czarnecki e Eisenecker, visto que: 1) assume-se como domínio generativo as APIs REST de acesso a estruturas de dados; 2) assume-se como modelo de configuração a representação das estruturas de dados em diagramas de classes UML anotados, ou em definições JSON Schema; 3) assume-se como gerador o tradutor proposto, que irá gerar, a partir do modelo de configuração, os parâmetros necessários para o uso dos blocos de código na construção do sistema final.

### 2.1.4 Metodologias Utilizadas

As metodologias descritas nesta seção são utilizadas em momentos diferentes deste trabalho. Como descrito acima, CBSE é utilizada durante a fase de modelagem

da ferramenta AutoREST, na identificação das características comuns de APIs REST que possam ser reutilizadas, para que a AutoREST tenha uma base de componentes em comum que são utilizados durante sua execução.

MDD é utilizada na primeira metade do processo de execução da aplicação AutoREST, definindo regras de modelagem para a representação de APIs REST em alto nível de abstração, para posterior geração de código automatizada. Por fim, GP é utilizada na segunda metade do processo de execução da aplicação, utilizando os componentes parametrizáveis implementados durante o desenvolvimento da AutoREST para construir uma aplicação final, utilizando os modelos gerados na fase anterior como modelos de configuração.

## 2.2 Linguagem Unificada de Modelagem (UML)

Para a modelagem dos sistemas que são gerados pela solução AutoREST, é necessário o uso de uma linguagem de modelagem de software que permita a anotação de todos os dados necessários para a geração do código final. Além disso, é importante que a linguagem de modelagem utilizada seja de fácil compreensão, visto que para alcançar um nível de eficiência suficiente para a adoção da ferramenta em projetos reais, é necessário permitir a concepção de APIs REST de forma simples e abrangente.

Dentre as linguagens de especificação de sistemas, UML é considerada por vários autores como sendo a *lingua franca* (??) ou o padrão *de facto* para a descrição de modelos de design orientados a objetos (??). Mesmo ??), ao questionar a veracidade destas afirmações demonstrando que UML não é nem universalmente adotada, nem utilizada da mesma forma por toda a indústria, afirma que de fato UML se mantém como um padrão na área.

Definida formalmente pelo *Object Management Group* (OMG) (??), a UML tem como objetivo “provêr a arquitetos de sistemas, engenheiros de software, e desenvolvedores de software ferramentas para análise, design e implementação de sistemas baseados em software, assim como para a modelagem de negócios e processos similares”.

A linguagem é profundamente desenvolvida tanto em (??) como nos livros canônicos de ??), ??) e ??), além de possuir inúmeras extensões (perfis) que vão além do escopo deste trabalho.

A UML é uma excelente opção de linguagem de especificação para servir como base da AutoREST, visto que cobre o requisito citado acima de ser simples e abrangente, e como será apresentado a seguir, possui todas as características necessárias para a notação dos dados de representação de uma API REST.

### 2.2.1 Diagrama de Classes UML

Dentre os diagramas disponibilizados pela UML, o Diagrama de Classes serve particularmente bem na definição de estruturas de dados. Este diagrama permite a representação gráfica de classes, no sentido de orientação a objetos, mostrando seus atributos e métodos de forma bem definida.

Para a representação dos dados de uma API REST, esta representação de classes se mostra particularmente eficaz por permitir a definição de tipos de dados e relacionamentos entre classes. Além disso, como todos os diagramas da UML, o Diagrama de Classes é de fácil extensão através do uso de *Stereotypes* e *Tags*, que permitem a definição de regras mais refinadas para a composição dos atributos.

Neste trabalho, será usado como artefato de entrada da solução AutoREST um Diagrama de Classes em formato XML e anotado com certas definições derivadas da representação JSON Schema. Este diagrama servirá como modelo de configuração para a geração automática de APIs REST.

## 2.3 JavaScript Object Notation (JSON)

Apesar da UML ser uma excelente forma de representação para os modelos de configuração da AutoREST, ela provê apenas uma notação gráfica, o que é insuficiente para o processamento de um compilador por ser de difícil processamento por máquinas. Ainda que existam notações para a representação textual de UML, como determinadas formas de XML, estas notações são de difícil compreensão humana, o que vai contra o preceito de MDD de utilizar representações que possam ser lidas e mantidas facilmente pelos seus usuários.

Para cumprir os requisitos de MDD e a necessidade de uma representação textual para o processamento de um compilador, utilizamos a notação JSON, estabelecida por Crockford (??) (??) (??). Esta notação, criada para servir como um formato para troca de dados independente de linguagem, é baseada em um subconjunto da linguagem de programação JavaScript, e é considerada de fácil leitura por humanos e fácil processamento por máquinas.

Além das vantagens inerentes da notação JSON, ela também é tradicionalmente utilizada nas transações de APIs REST (??) (??), sendo por esta razão particularmente apropriada para a solução descrita neste trabalho.

### 2.3.1 JSON Schema

A notação JSON normalmente serve para a representação de dados propriamente ditos, e não para a representação de estruturas de dados. Visto que os modelos de



configuração da AutoREST são estruturas de dados, é necessária uma notação que descreva o formato de arquivos JSON para que possam ser abstraídas estruturas de dados. Para este propósito, utilizamos JSON Schema (??).

JSON Schema é uma linguagem de descrição de formato de dados baseada na linguagem JSON que, apesar de já ser amplamente aceita, ainda não está totalmente definida e continua em desenvolvimento por ??). Apesar de não possuir uma definição completa, ??) já disponibilizam uma ótima definição para JSON Schema, provendo uma especificação formal e uma gramática formal.

Como será apresentado na descrição da solução proposta neste estudo, o mapeamento de um diagrama de classes UML anotado para uma definição JSON Schema completa é possível, dadas certas limitações e o uso de um subconjunto pré-determinado da linguagem JSON Schema proposta por Pezoa.

## 3 Arquiteturas de Software

Neste capítulo são apresentados os modelos arquiteturais pertinentes a este trabalho, focando nas arquiteturas REST e HTTP, que são aplicadas diretamente na solução proposta.

### 3.1 Arquitetura Cliente/Servidor

A arquitetura cliente/servidor tem como fundamento a existência de duas camadas distintas de software, onde o servidor disponibiliza serviços que serão utilizados pelos clientes, conforme ??). ??) definem um cliente como sendo um computador que acessa recursos de outro computador. O computador que provê os serviços é, por sua vez, o servidor.

Um recurso pode ser considerado como qualquer função de software, hardware ou informação que um servidor disponibiliza. A disponibilização de um recurso é o que constitui um serviço. Este conceito arquitetural é importante para que se derive a compreensão da arquitetura REST.

### 3.2 Arquitetura REST

Nesta seção será apresentada e estudada a arquitetura de Transferência de Estado Representacional, do inglês *REpresentational State Transfer* (REST). Esta arquitetura foi criada por Roy T. Fielding e é definida em sua tese (??) da seguinte forma, em tradução livre:

O modelo *REpresentational State Transfer* (REST) é uma abstração dos elementos arquiteturais internos de um sistema de *hypermedia* distribuído. REST ignora os detalhes de implementação dos componentes e sintaxe de protocolo para focar na função dos componentes, as restrições a respeito das interações com outros componentes, e a interpretação de elementos significativos de dados. Ele abrange as restrições fundamentais sobre componentes, conectores e dados que definem a base da arquitetura Web, e por conseguinte a essência de seu comportamento como uma aplicação baseada em rede.

Por se tratar de um modelo arquitetural para Web, REST é uma arquitetura fundamentalmente restritiva. Das restrições da arquitetura REST, as principais são:

- **Cliente-servidor:** Esta restrição define que um cliente é um componente da arquitetura que utiliza os serviços disponibilizados por um servidor, conforme a Seção ??;
- **Stateless:** Define que não há estados na arquitetura, ou seja, todas as iterações são independentes e auto-contidas, não havendo assim a necessidade do servidor gerenciar estados internamente;
- **Cache:** Requer que os dados de resposta a uma requisição sejam implícita ou explicitamente categorizados como *cacheable* ou *non-cacheable*. Se uma resposta for *cacheable* o cliente poderá utilizá-la para requisições futuras;
- **Interface uniforme:** Define que os componentes irão utilizar uma interface uniforme, facilitando implementações, desacoplamento e integrações;
- **Sistema em camadas:** Restringe o sistema a ser organizado hierarquicamente em camadas, cada uma com visibilidade a suas camadas adjacentes, conforme (??).

### 3.2.1 Elementos Arquiteturais REST

Como citado no início desta seção, ??) define REST como um modelo abstrato de elementos arquiteturais. Estes elementos arquiteturais definem um padrão no formato de interfaces web que pode ser explorado para a geração automática destas interfaces.

#### 3.2.1.1 Elementos de dados

Os dados devem ser providos em um formato que satisfaça uma interface padronizada, ainda que isto seja diferente do formato de origem nas camadas inferiores. Para isto, qualquer informação que pode receber um nome é chamada de *resource* (recurso). Um *resource* define características em comum de um determinado conjunto de dados. Quando é necessário referenciar determinado *resource*, usa-se um *resource identifier* (identificador de recurso), que será usado nos vários conectores quando estiverem utilizando o mesmo *resource*.

Ações nos *resources* serão realizadas por componentes utilizando *representations* (representações) que capturam o estado atual ou desejado. Junto com a *representation*, também podem ser enviados metadados descrevendo o conteúdo que está sendo transportado. O formato dos dados de uma *representation* é conhecido como *media type* ou *MIME type*, conforme apresentado por ??).

Para que as ações em uma *representation* tenham significado entre os componentes, existem os dados de controle (*control data*). Conforme os dados de controle, uma *representation* pode significar o estado atual ou o estado desejado de um *resource*.

### 3.2.1.2 Conectores

Conectores são utilizados para encapsular atividades, possuindo uma interface abstrata de comunicação que oculta a implementação dos *resources* e mecanismos de comunicação, aplicando assim a restrição de camadas de ??). A utilização de um conector é similar a utilização de um método em programação estrutural (??), onde os parâmetros de entrada seriam os dados de controle, um *resource identifier* e, opcionalmente, uma *representation*. Já o resultado consistiria de dados de controle e, opcionalmente, metadados de *resource* e/ou uma *representation*.

Os principais conectores em REST são o cliente e o servidor. Um cliente iniciará comunicações através de *requests* (requisições) e um servidor ficará encarregado de esperar estas *requests* de forma a prover acesso a seus serviços.

Outros conectores, embora estejam fora do escopo deste trabalho, seriam:

- *cache*: para que clientes e servidores evitem envio e processamento, respectivamente, de *requests* com resultado já em *cache*;
- *resolver*: para resolução parcial ou total de endereços para o estabelecimento de conexões;
- *tunnel*: para que componentes REST troquem seu comportamento para o de um túnel, responsável apenas por repassar a *request* para outro destinatário.

### 3.2.1.3 Componentes

Os principais componentes REST são: *user agent*, que é um conector cliente e iniciará uma *request*, se tornando assim o recipiente final da *response*, e; *origin server*, um conector servidor e é a origem das *representations* dos *resources* e, portanto, o recipiente final de todas as *requests*. Também existem dois componentes secundários em REST: o *proxy* (??) e o *gateway*. Estes componentes possuem como objetivo agir como intermediários, encapsulando funcionalidades de rede como tradução de endereço e roteamento de *requests* e *responses*.

## 3.3 Hypertext Transfer Protocol (HTTP)

*Hypertext Transfer Protocol* (HTTP) é um protocolo de aplicação (??) usado amplamente na *World Wide Web* (WWW). A versão inicial, HTTP/1.1, foi publicada na RFC 2616 (??) e a maioria de seus conceitos são válidos até hoje. Uma versão mais atual, HTTP/2.0, foi publicada na RFC 7540 (??) mas não substitui a versão anterior, sendo ao invés disto uma alternativa para ela, compatível com a semântica do HTTP/1.1.

O protocolo HTTP possui vários parâmetros pertinentes a seu transporte pela Web e também dois tipos de mensagens, *request* e *response*. As mensagens possuem duas áreas principais: o *header* (cabeçalho), que é constituído de um conjunto de chaves e valores, e; o *message body* (corpo da mensagem). As partes mais importantes para o desenvolvimento da ferramenta AutoREST são as que constituem estas duas áreas.

Em uma *request* o *header* irá conter um *method* (método), uma *Uniform Resource Identifier* (URI) e outros valores que descrevem a *request* e o *entity-body* (ou a falta do mesmo). Os métodos HTTP são detalhados na Seção ???. Já na *response*, o *header* irá conter um *status code* e uma respectiva *reason phrase* e, assim como a *request*, outros valores que descrevem a *response* e o *entity-body* ou a falta do mesmo. O protocolo HTTP possui um considerável conjunto de *status codes*. Suas categorias e relevância para este trabalho são definidos na Seção ??.

A definição mais atual para URI está na RFC 3986, de autoria de ??). Conforme os autores, uma URI proporciona uma maneira simples e extensível para a identificação de *resources*. O principal uso para o protocolo HTTP é exatamente este: identificar os *resources* de um determinado servidor que um cliente quer utilizar.

Um item importante para as mensagens HTTP é o tipo do conteúdo sendo transportado. Para isto, os *headers* possuem duas configurações que são muito importantes para a arquitetura REST: *Content-Length* e *Content-Type*. *Content-Length* indica o tamanho da *entity* sendo transportada; se este valor for zero, não existe conteúdo sendo transportado. *Content-Type* indica o tipo de conteúdo sendo transportado, o que possibilita ao cliente saber como interpretá-lo. Os tipos de conteúdo aceitos em *Content-Type* foram definidos em várias publicações ao longo dos anos, iniciando em ??), e atualmente são mantidos pela Internet Assigned Numbers Authority (IANA)<sup>1</sup>.

### 3.3.1 Métodos

Os métodos HTTP servem para que o cliente especifique qual o objetivo da *request* que está sendo enviada; isto faz com que os métodos adquiram propriedades importantes na comunicação de cliente e servidor. Por convenção, um método não deve executar mais tarefas que sua definição. Como não é possível garantir que o servidor não as executará, o cliente é isento no caso de tarefas além do escopo do método.

Outra característica muito importante é a *idempotência*; isto é, a execução de uma mesma *request* várias vezes deve gerar sempre o mesmo resultado.

A seguir são definidos os métodos atualmente suportados pelo protocolo HTTP, sejam eles originalmente definidos pela versão HTTP/1.1 ou através de documentos posteriores (??).

---

<sup>1</sup> <<http://www.iana.org/assignments/media-types/media-types.xhtml>>

### 3.3.1.1 OPTIONS

Quando um cliente deseja saber as opções ou requisitos de comunicação de um *resource* ou servidor, o cliente pode utilizar o método OPTIONS, evitando assim a necessidade de informar uma ação. A *response* do servidor deve conter em seu *header* o máximo de informações possíveis sobre suas funcionalidades, sendo também possível enviar dados adicionais no *message body*.

### 3.3.1.2 GET

Indica que o cliente está requisitando o *resource* indicado na URI da *request*. O servidor deve enviar este *resource* no *message body* da *response*, além de seus metadados no *header*.

### 3.3.1.3 HEAD

É similar ao GET, porém não deve retornar um *message body*. Para uma *request* com HEAD, o servidor deve retornar a mesma *response* que seria esperada se usado o método GET, porém sem *message body* e *headers* relacionados ao mesmo.

### 3.3.1.4 POST

Tem o propósito de realizar o processamento de *resources* no servidor. Para realizar tal função, o *resource* deve ser enviado no *message body* da *request* para a URI desejada. O resultado do processamento deste *resource* no servidor é indicado pelo *status code* na *response*; é possível também que a *response* tenha o *resource* resultante no *message body* ou o local do *resource* no campo *Location* do *header*.

### 3.3.1.5 PUT

O método PUT define que o *resource* especificado na URI da *request* seja criado ou substituído usando a *entity* no *message body*, o que já permitiria um subsequente GET para a mesma URI. A *response* irá conter em seu *status code* o resultado da operação.

### 3.3.1.6 DELETE

Este método informa ao servidor que o *resource* indicado na URI da *request* deve ser removido. A *response* de um DELETE conterá dados em seu *header*, como *status code* referente a operação, mas não é obrigatório que contenha *message body*.

### 3.3.1.7 TRACE

Quando um servidor recebe uma *request* com o método TRACE, ele deverá enviar uma *response* com o mesmo conteúdo da *request*, porém com o *status code* de sucesso (??).

### 3.3.1.8 CONNECT

Um cliente envia um *request* de CONNECT quando deseja estabelecer um túnel utilizando o servidor de destino como *proxy*.

### 3.3.1.9 PATCH

O método PATCH é utilizado para que o cliente solicite a atualização parcial de um *resource*. Uma *request* que utiliza PATCH terá uma descrição das modificações, no *message body*, que deseja realizar no *resource* contido na URI. A definição mais atual para o método PATCH pode ser encontrada em (??). Também existe uma definição específica para PATCH em *resources* do tipo JSON (??).

## 3.3.2 Status Codes

Os *status codes* servem para indicar ao cliente o resultado de uma *request*. Eles fazem parte do *header* da *response* e são divididos nas categorias a seguir. Uma lista dos *status codes* (??) pode ser encontrada no Anexo ??.

- 1xx (Informação): A *request* foi recebida, processamento em andamento
- 2xx (Sucesso): A *request* foi recebida com sucesso, compreendida, e aceita
- 3xx (Redirecionamento): Ações adicionais precisam ser realizadas para completar a *request*
- 4xx (Erro do cliente): A *request* contém erros de sintaxe ou não pode ser completada
- 5xx (Erro do servidor): O servidor falhou em completar uma *request* aparentemente válida

## 4 Tecnologias e Ferramentas

Este capítulo descreve as tecnologias e ferramentas que são utilizadas na solução proposta neste estudo, desde a linguagem de programação que será utilizada na implementação da ferramenta AutoREST até as ferramentas de validação dos modelos de configuração e o sistema gerenciador de bases de dados utilizado pela API gerada pela solução.

### 4.1 Astah

A ferramenta de modelagem Astah<sup>1</sup> foi criada pela empresa *Change Vision* e tem como fundamento básico a facilidade de uso. O Astah Professional dispõe de diversas formas de modelagem, como múltiplos diagramas UML (Classes, Componentes, Casos de Uso, Atividades, entre outros) e modelos Entidade-Relacional.

Esta ferramenta será usada como base para a modelagem dos Diagramas de Classes UML que serão utilizados na solução proposta neste estudo. Esta escolha foi motivada tanto pela aceitação do Astah pela comunidade de usuários quanto pela disponibilidade da licença para o sistema completo oferecida a estudantes.

### 4.2 JSON Validators

Existem muitas soluções que se propõem à validação e geração de JSON Schemas. Nesta seção apresentamos algumas das ferramentas mais populares para a validação de estruturas JSON usando JSON Schema, e também uma ferramenta que utiliza como base de sua implementação uma definição formal de gramática BNF.

#### 4.2.1 Newtonsoft JSON Schema Validator

O validador da Newtonsoft é implementado em .NET (??) e é uma das escolhas padrões para programadores neste framework devido a sua confiabilidade adquirida. É contruído para fornecer suporte aos *Drafts* 3 e 4 de JSON Schema criados por ??). Para validações *online* de JSON é disponibilizado um site que utiliza o validador<sup>2</sup>.

---

<sup>1</sup> <<http://astah.net/>>

<sup>2</sup> <<http://www.jsonschemavalidator.net/>>



### 4.2.2 JSON Schema Validator

Conforme as definições de ??), o JSON Schema Validator foi desenvolvido em Java por um dos autores para realizar a validação de JSONs e JSON Schemas. Atualmente está com seu código aberto e livre<sup>3</sup>, tornando-o assim uma das melhores alternativas de mercado para as validações de JSON Schema na linguagem Java. Também é possível testá-lo *online* através de um site que o utiliza internamente<sup>4</sup>.

### 4.2.3 PUC/Chile JSON Validator

No trabalho de ??), os autores buscavam uma definição formal e que pudesse ser usada para pesquisas e automação de ferramentas utilizando JSON Schema. Após definirem formalmente uma gramática BNF, criaram um validador em Python utilizando a gramática em questão como base<sup>5</sup>.

## 4.3 Java

A linguagem de programação Java é amplamente conhecida e será a linguagem utilizada na implementação da ferramenta proposta neste estudo. Esta linguagem possui diversas vantagens, como por exemplo: ser construída com base em uma máquina virtual que permite que programas escritos em Java sejam executados em múltiplas plataformas; ser primariamente orientada a objetos, o que permite maior facilidade de uso em conjunto com a notação UML; ser uma linguagem de licença pública; possuir ambientes de desenvolvimento maduros, e; ser ensinada em diversas universidades ao redor do mundo. Informações específicas da linguagem podem ser encontradas em diversos livros, como os de ??), ??) e ??).

## 4.4 Node.js

Node.js é um *runtime* de JavaScript orientado a eventos (??) e com capacidade de operações de leitura e escrita assíncronas. Estas características permitem uma alta escalabilidade para as aplicações, tornando Node.js altamente usado em aplicações Web.

### 4.4.1 NPM e Módulos

No ambiente Node.js cada arquivo é considerado um módulo; assim, as funções e classes contidas em um arquivo de código podem ser facilmente utilizadas em outros, tornando necessário apenas “carregar” o módulo desejado. O ambiente Node.js possui

<sup>3</sup> <<https://github.com/daveclayton/json-schema-validator>>

<sup>4</sup> <<http://json-schema-validator.herokuapp.com/>>

<sup>5</sup> <[https://github.com/CSWR/json\\_schema\\_validator](https://github.com/CSWR/json_schema_validator)>

uma comunidade de desenvolvedores ampla e ativa. Para que funcionalidades necessárias a todos possam ser distribuídas, vista a ubiquidade de uso do ambiente, é utilizado o NPM (*Node Package Manager*, Gerenciador de Pacotes Node) que mantém módulos criados e publicados pela própria comunidade.

Apesar das várias opções disponíveis no NPM, muitos módulos acabam sendo amplamente adotados e se tornando parte do ferramental diário de desenvolvedores Node.js. Um destes é o Express, um módulo robusto para a criação de servidores HTTP.

#### 4.4.2 Padrões de Projeto

Conforme apresentado por ??), Node.js possui muitas peculiaridades quanto a aplicação dos padrões de projeto clássicos da orientação a objetos (?). Devido a estas peculiaridades, muitos padrões de projeto sofrem algumas alterações quando aplicados a Node.js; estas peculiaridades também permitiram o surgimento de alguns padrões no ecossistema de Node.js.

Entre estes padrões, um dos que se destaca é o *Padrão Middleware*. Nele, há uma série de funções que devem ser executadas em um determinado processo; porém, estas funções podem ser substituídas por outras que tenham a mesma entrada e a mesma saída. Assim, um *middleware* seria um *software do meio*, que pode ser trocado por outro com o mesmo propósito mas outra implementação.

### 4.5 MongoDB

MongoDB (??) é um banco de dados *NoSQL* (??) orientado a documentos. Diferente dos bancos de dados relacionais, que armazenam linhas em uma tabela, o MongoDB armazena documentos similares com o formato JSON em uma *collection*. As *collections* não possuem um *schema* (esquema) como as tabelas, o que torna possível o armazenamento de documentos com as mais variadas propriedades.

#### 4.5.1 Mongoose

Para a utilização do MongoDB em aplicações, usam-se *drivers*, que são programas para o estabelecimento de conexão e realização de operações no banco de dados. O Mongoose é um módulo de Node.js que serve exatamente este propósito; ele também proporciona a criação de *models* para as *collections* do MongoDB. Desta forma, é possível realizar validação de tipos e valores dos dados, além de adicionar regras de negócio próprias.

No Mongoose, é possível a adição de *middlewares* e *plugins* para novos recursos e validações.

## 5 AutoREST: Geração Automática de APIs REST para Funcionalidades CRUD

Neste capítulo, iremos apresentar a solução proposta para as questões levantadas neste estudo. A solução proposta, chamada AutoREST, é composta por um processo de modelagem de estruturas de dados e uma ferramenta de geração automática de APIs utilizando estes modelos como entrada. Serão apresentados os pontos motivadores deste estudo seguidos dos requisitos para a validação da solução proposta. Em seguida, a arquitetura AutoREST é proposta para cumprir com estes requisitos e, finalmente, a modelagem de uma instância desta arquitetura é apresentada como prova de conceito, utilizando as linguagens de modelagem UML e JSON Schema, o framework de programação Node.js e o sistema gerenciador de bancos de dados (SGBD) MongoDB através da biblioteca Mongoose.

### 5.1 Motivação

É parte essencial de sistemas Web que lidem com dados de persistência a implementação de APIs para a recepção e encaminhamento de requisições do tipo CRUD. Dentre as formas de implementação deste tipo de serviço o estilo arquitetural REST se destaca como a base fundamental da Internet e, portanto, é particularmente apropriada como base para o desenvolvimento de APIs de comunicação e acesso a recursos (??).

A geração automática de código a partir de modelos é uma maneira de garantir a confiabilidade destas APIs e aumentar a produtividade do processo de geração de sistemas (??). Visto que a arquitetura REST tem entre suas restrições a presença de interfaces uniformes, com interações auto-contidas, a geração de APIs REST se apresenta particularmente bem para a aplicação de métodos de representação conceitual e geração de código automática.

O objetivo deste trabalho é a proposta de um paradigma para a representação conceitual do fornecimento de objetos por uma API REST, baseado na abordagem MDD. Além disso, buscamos uma maneira de proporcionar a geração automática de APIs REST para funcionalidades de manutenção de documentos de persistência a partir destes modelos conceituais, baseada na abordagem GP. Finalmente, a solução é apresentada utilizando modelos arquiteturais baseados nos preceitos da abordagem CBSE.

## 5.2 Requisitos

Nesta seção, apresentamos os requisitos que foram identificados ao longo deste estudo para que o processo de modelagem conceitual e de geração automática de código para APIs REST seja possível. Foram levadas fortemente em consideração as metodologias apresentadas no Capítulo ?? e os padrões arquiteturais apresentados no Capítulo ?. Estes requisitos foram formulados para servir como base para o projeto de uma arquitetura de software altamente independente de tecnologias de implementação.

- RQ01: A solução apresentada deve proporcionar orientações exatas para a modelagem conceitual de uma estrutura de dados contendo todas as entidades, associações, atributos e restrições de integridade necessárias para a geração automática de uma API REST. Este modelo conceitual deve utilizar de linguagem gráfica que seja considerada de fácil compreensão e manutenção pela literatura de referência, conforme preceitos da abordagem MDD;
- RQ02: O sistema deve ser capaz de converter um modelo conceitual criado em linguagem gráfica para uma estrutura intermediária de fácil processamento por máquinas. Esta estrutura deve conter todos os dados presentes no modelo conceitual original;
- RQ03: O sistema deve gerar automaticamente, a partir de um modelo conceitual conforme RQ01, uma API REST que implemente as operações básicas de manutenção das entidades componentes do esquema conceitual (CRUD). Esta geração deve se valer de blocos de código parametrizáveis para a padronização das APIs geradas, conforme preceitos da abordagem GP;
- RQ04: O código fonte gerado deverá ser implementado de forma a permitir a persistência e manipulação dos recursos acessados durante a operação da API REST através de um SGBD;
- RQ05: A arquitetura da solução deverá se valer de interfaces com contratos bem definidos de forma a permitir a implementação de novos componentes para a extensão da aplicação, conforme preceitos da abordagem CBSE.

## 5.3 Decisões de Projeto

Nesta seção são apresentadas as decisões de projeto referentes a uma instância da arquitetura AutoREST, que será projetada e desenvolvida como prova de conceito da arquitetura e como embasamento para responder as perguntas de pesquisa propostas nesse trabalho. Estas decisões de projeto foram desenvolvidas com base nas tecnologias e linguagens apresentadas no Capítulo ?.

- DD01 (referente ao RQ01): Serão criadas orientações para a modelagem de um Diagrama de Classes UML da ferramenta Astah para servir como modelo de configuração. Este diagrama irá representar um esquema conceitual de dados sobre o qual a API REST será gerada, com suas classes, relacionamentos, atributos e domínios, e anotado com um subconjunto da linguagem JSON Schema para representar as restrições de integridade das entidades.
- DD02 (referente ao RQ02): O sistema irá dispor de algoritmos e regras de conversão de um Diagrama de Classes UML anotado conforme DD01 para uma representação de definições JSON Schema, que irá servir como PFIS (Processing-Friendly Intermediate Structure) do sistema.
- DD03 (referente ao RQ03): O sistema irá realizar a geração automática de uma API REST em código na linguagem Node.js a partir das definições JSON Schema geradas conforme DD02. Este código deverá desempenhar as operações básicas de manutenção de documentos de persistência (CRUD).
- DD04 (referente ao RQ04): A API REST gerada irá persistir os dados através do SGBD MongoDB, utilizando-se da biblioteca Mongoose.
- DD05 (referente ao RQ03): O sistema irá realizar a geração de código a partir de fragmentos de código pré-implementados representando cada um dos métodos HTTP (GET, HEAD, POST, PUT, PATCH, DELETE).
- DD06 (referente ao RQ05): O sistema será desenvolvido em quatro partes independentes, como representadas na Figura ??, sendo estas: um leitor de arquivos XML representativos de Diagramas de Classes UML da ferramenta Astah, conforme DD01; um conversor de Diagramas de Classes UML anotados para definições em JSON Schema, conforme DD02; um gerador de código Node.js para APIs REST a partir de definições JSON Schema, conforme DD03, e; um componente de interface com usuário e controle de execução de sistema.

## 5.4 Proposta de Solução

Para que todos os requisitos apresentados acima sejam cumpridos, é proposto o modelo arquitetural AutoREST, uma arquitetura independente de tecnologia, com seus modelos estruturais e comportamentais em linguagem UML. Exemplos de possíveis concretizações dos componentes conceituais são apresentadas para motivos de contextualização.

Projetada com base nos preceitos da abordagem CBSE, utilizando como domínio de solução as aplicações para a geração automática de APIs REST a partir de esquemas conceituais, a AutoREST é composta de quatro subsistemas na forma de componentes

que operam de forma autônoma e cinco interfaces de comunicação para permitir a cooperação entre estes quatro subsistemas. Cada subsistema possui uma dependência ligada às interfaces ao qual está relacionado, mas se mantém independente dos outros subsistemas. Desta forma, a arquitetura obedece o requisito RQ05 listado na Seção ??.

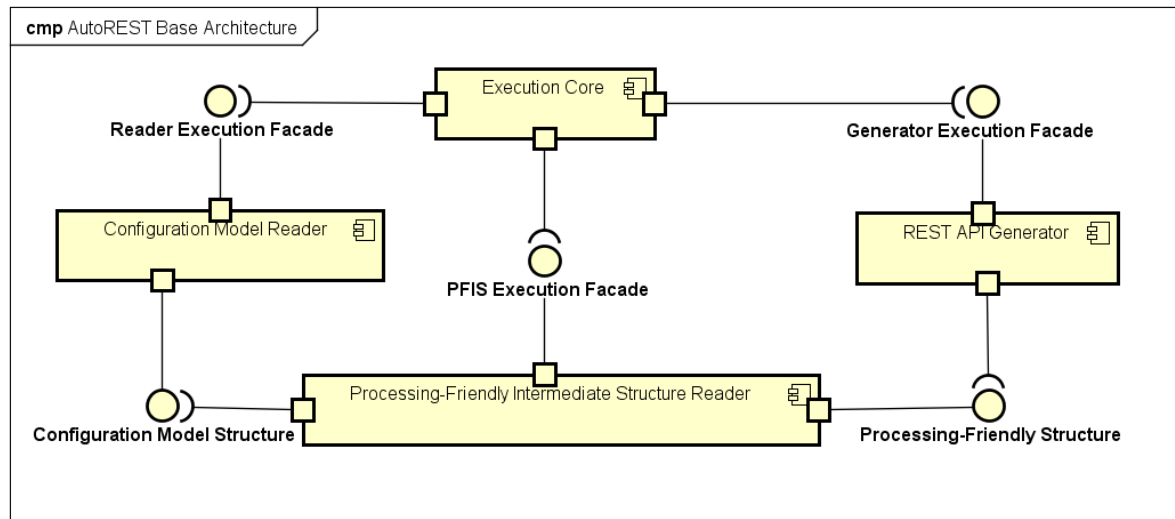


Figura 1 – AutoREST - Arquitetura Base

A Figura ?? apresenta um diagrama de componentes UML representando os quatro subsistemas e suas interfaces. As seções a seguir descrevem cada um destes subsistemas e interfaces em maior detalhe.

- *Execution Core*: é o subsistema encarregado de controlar a execução da ferramenta, provendo as interfaces de usuário e gerenciando a comunicação dos outros subsistemas;
- *Configuration Model Reader* (CMR): é o subsistema encarregado da leitura do modelo conceitual em linguagem gráfica, criado conforme orientações dadas pela documentação do componente utilizado na implementação da arquitetura;
- *Processing-Friendly Intermediate Structure Reader* (PFIS-R): é o subsistema encarregado de realizar a tradução do modelo conceitual em linguagem gráfica para uma estrutura intermediária de fácil processamento;
- *REST API Generator*: é o subsistema encarregado da geração de uma implementação da API REST representada no esquema conceitual de entrada do sistema;
- *Execution Facades*: são as interfaces utilizadas para a comunicação do subsistema de controle com os outros subsistemas e definidas utilizando o padrão de projeto Fachada (??);

- *Configuration Model Structure*: é a interface de comunicação entre os subsistemas CMR e PFIS-R, servindo como ponto de transferência do modelo conceitual em linguagem gráfica entre os dois subsistemas;
- *Processing-Friendly Structure*: é a interface de comunicação entre os subsistemas PFIS-R e *REST API Generator*, servindo como ponto de transferência da estrutura intermediária de fácil processamento entre os dois subsistemas.

#### 5.4.1 *Execution Core e Execution Facades*

O subsistema *Execution Core* é responsável pelo controle da aplicação através de requisições às *Execution Facades*. Estas requisições devem ser procedimentos ativadores das funções básicas de cada um dos subsistemas, transferindo pacotes de dados entre eles. Estes pacotes de dados serão estruturados conforme as interfaces providas pelos subsistemas CMR, PFIS-R e Generator.

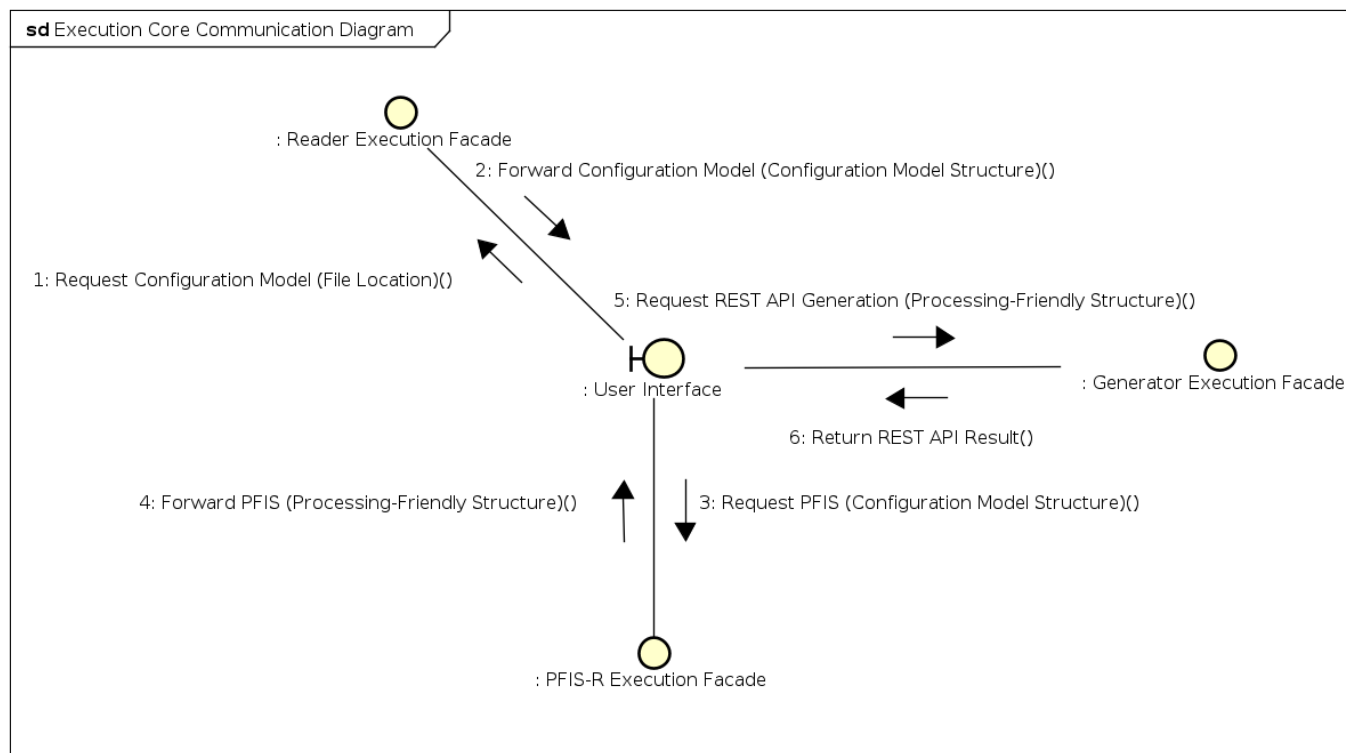


Figura 2 – Diagrama de Comunicação do Subsistema *Execution Core*

A Figura ?? apresenta um diagrama de comunicação entre a interface de usuário provida pelo *Execution Core* e as interfaces de *Execution Facade*. A comunicação ocorre através de seis mensagens básicas:

1. É realizada uma requisição ao *Reader Execution Facade* para que o modelo conceitual, contido em um arquivo cuja localização é enviada como argumento, seja carregado em memória;
2. Após a execução dos processos internos do subsistema CMR, o modelo conceitual é retornado ao *Execution Core* para que este dê continuidade ao processo;
3. É realizada uma requisição ao *PFIS-R Execution Facade* para que o modelo conceitual em memória, contido em um pacote de dados enviado como argumento, seja convertido em uma PFIS;
4. Após a execução dos processos internos do subsistema PFIS-R, a PFIS é retornada ao *Execution Core* para que este dê continuidade ao processo;
5. É realizada uma requisição ao *Generator Execution Facade* para que a API REST seja gerada a partir do PFIS em memória, contido em um pacote de dados enviado como argumento;
6. O código gerado é retornado ao *Execution Core* e entregue ao usuário.

O resultado da composição do sistema através deste padrão de comportamento é um desacoplamento dos outros subsistemas componentes da arquitetura AutoREST, permitindo que estes sejam construídos de forma semi-autônoma, tendo como dependência apenas as estruturas de dados providas por interfaces de outros subsistemas.

#### 5.4.2 *Configuration Model Reader (CMR) e Configuration Model Structure*

O subsistema *Configuration Model Reader* (CMR) é responsável por definir a estrutura interna de sistema do modelo de configuração (representado por um modelo conceitual em linguagem gráfica) da ferramenta AutoREST. Além disto, é responsável por prover as funcionalidades de IO relacionadas a estes modelos. Este subsistema é composto de elementos de código e artefatos de documentação, visto que é parte dele também a definição de orientações e restrições para a construção dos modelos de configuração, conforme RQ01 apresentada na Seção ??.

Minimamente, a estrutura de dados definida pelo subsistema CMR deve conter representações de entidades com identificadores únicos, associações entre estas entidades, seus atributos e suas restrições de integridade. Exemplos de linguagens de modelagem que proveem estes elementos são Diagramas de Classes UML anotados (utilizado na prova de conceito apresentada na Seção ??) e Modelos Entidade-Relacionamento (??) (??).

Algumas restrições de integridade que se mostraram importantes na representação de uma RESTful API são: restrições de tipo de atributos, tipos de acesso aceitos para cada associação, notação de mandatório ou multiplicidade e notação de herança. Em relação



à questão de tipos de acesso aceitos por associação, se quer dizer especificamente quais os atributos que podem ser utilizados em requisições HTTP a partir das entidades que compõe a associação.

Nosso estudo não identificou utilidade prática de determinadas restrições de integridade na geração de RESTful APIs. São estas: notações de classes internas, modificadores de acesso, entidades com identificadores compostos (aridade maior do que um) e atributos com tipos de dado não primitivos. Também não conseguimos identificar uma forma de implementar associações sem atributo relacionado, visto que todo recurso REST deve ser acessado através de um identificador.

### 5.4.3 *Processing-Friendly Intermediate Structure Reader (PFIS-R) e Processing-Friendly Structure*

O subsistema *Processing-Friendly Intermediate Structure Reader* (PFIS-R) é responsável pela tradução do modelo de configuração em um modelo textual equivalente para fácil processamento no estágio de geração da API REST, no subsistema *REST API Generator*. Este subsistema é composto por três componentes como representado na Figura ??.

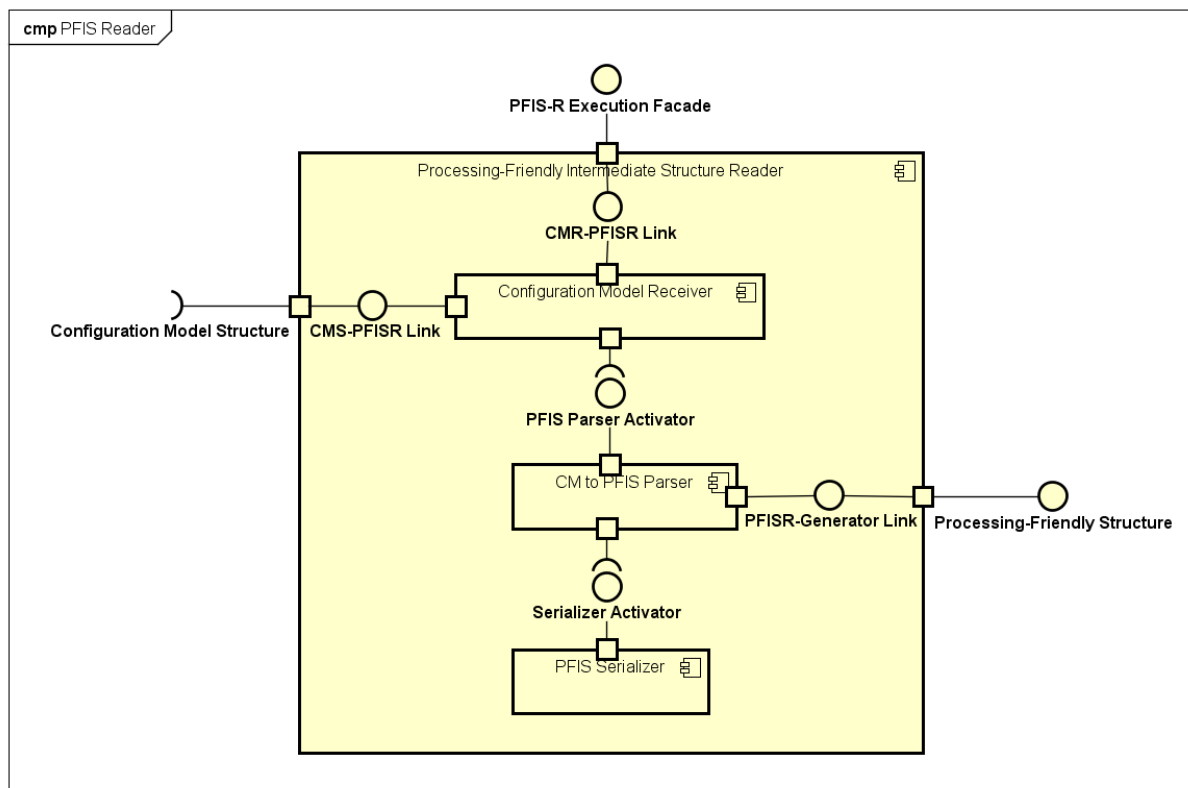


Figura 3 – Diagrama de Componentes do Subsistema PFIS-R

As interfaces internas do subsistema, que realizam a comunicação entre os compo-

nentes, permitem um grau de autonomia entre os componentes do subsistema PFIS-R. A execução deste subsistema é descrita pelo Diagrama de Sequência apresentado nas Figuras ?? e ?. A interface CMS-PFISR Link é omitida para reduzir o tamanho da figura.

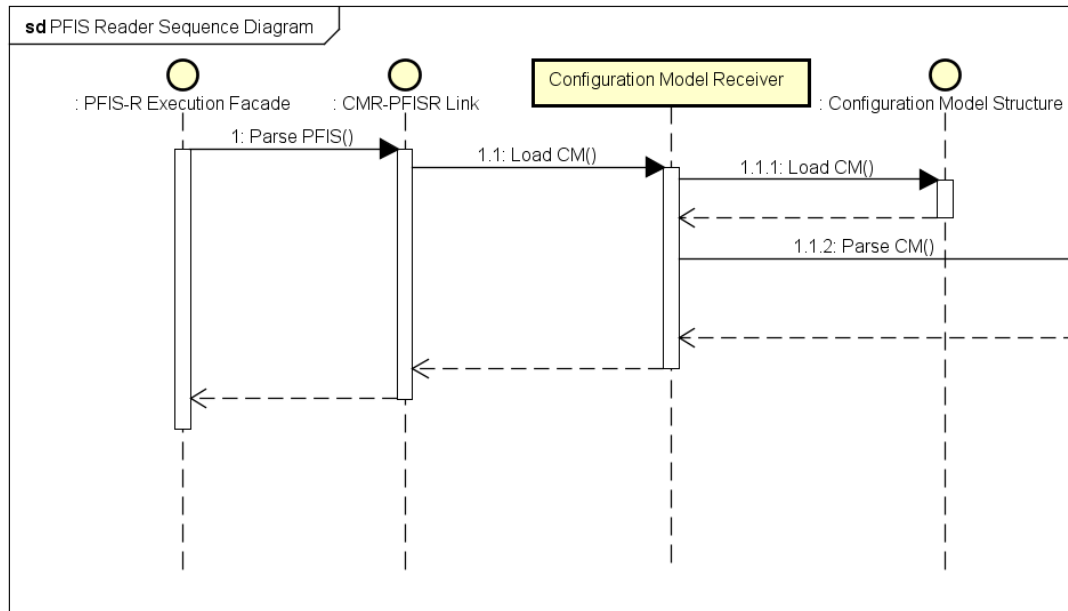


Figura 4 – Diagrama de Sequência do Subsistema PFIS-R - Parte 1

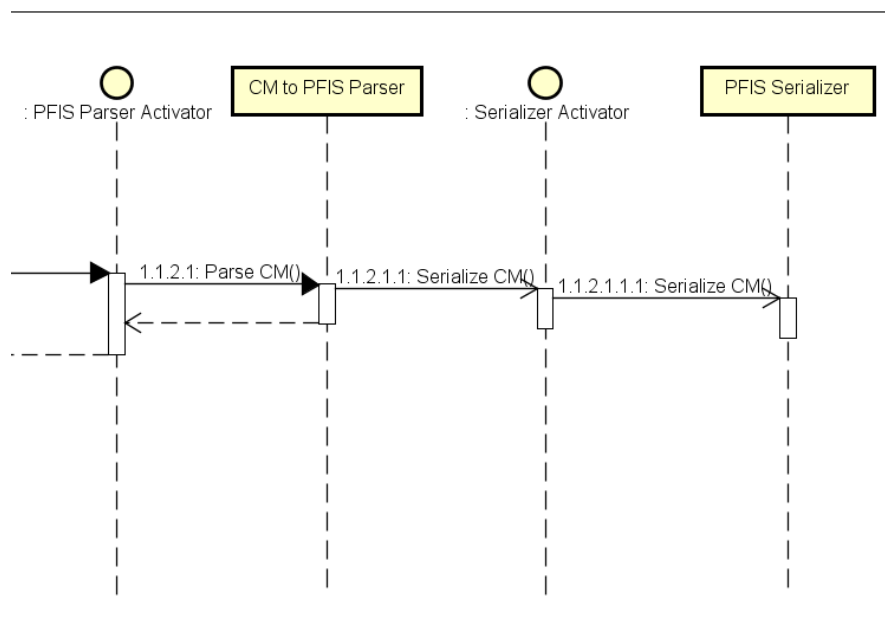


Figura 5 – Diagrama de Sequência do Subsistema PFIS-R - Parte 2

Este subsistema também é responsável por definir a estrutura interna de sistema da PFIS. Esta representação deve ser feita através de uma linguagem textual bem definida

por terceiros, em conformidade com as definições de CBSE de ??). Também é importante ressaltar mais uma vez que deve haver equivalência entre as representações CM e PFIS. Exemplos de linguagens que podem ser utilizadas para esta representação são RAML (??), Swagger (??), WSDL (??) e JSON Schema (??), este último sendo utilizado na prova de conceito apresentada na Seção ??.

#### 5.4.4 REST API Generator

O subsistema *REST API Generator* é responsável pela fase final da operação de um sistema AutoREST: a geração do código fonte de uma API REST. Este subsistema é composto por dois componentes, como apresentado na Figura ??: *PFIS Compiler* e *HTTP Method Stub Library*. O *PFIS Compiler* é responsável pela execução deste subsistema, recebendo requisições através da interface *Generator Execution Facade* e realizando a leitura de uma PFIS através da interface provida pelo subsistema PFIS-R.

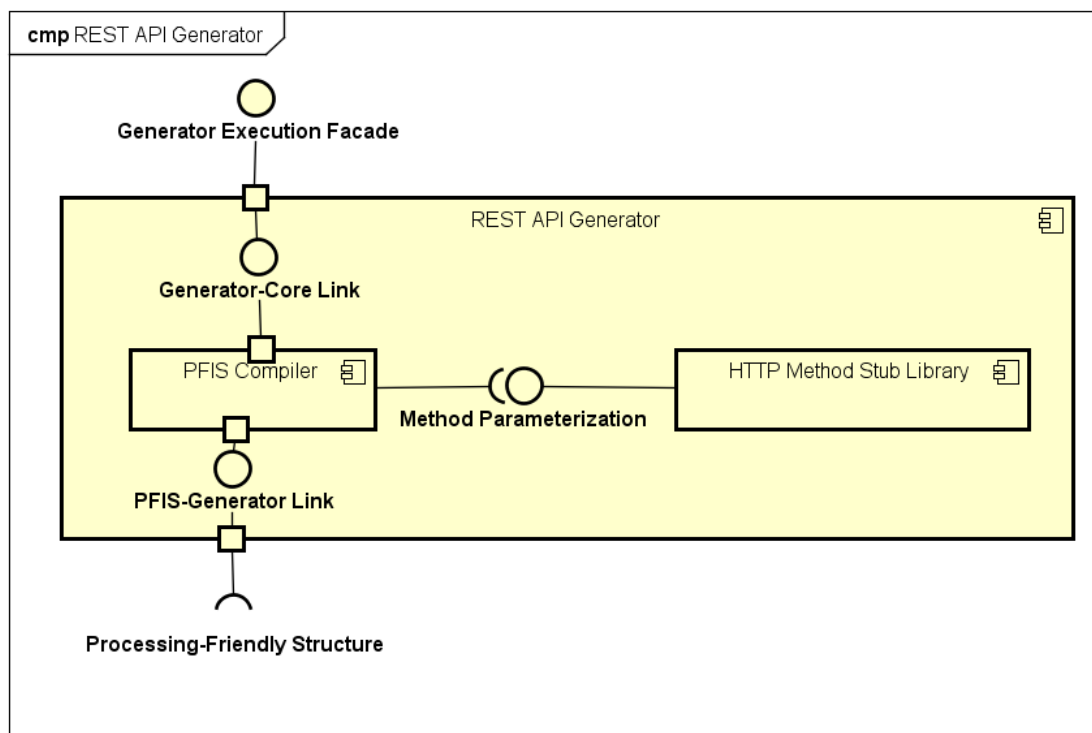


Figura 6 – Diagrama de Componentes do Subsistema *REST API Generator*

A partir da leitura de uma PFIS, o subsistema realiza a geração de uma API REST baseada no código parametrizável implementado na *HTTP Method Stub Library*, que serve como base para todas as APIs REST geradas por uma implementação deste subsistema, oferecendo código fonte base em determinada linguagem de programação. A forma utilizada para a leitura dos artefatos de entrada pelo gerador pode variar entre implementações, o

que motivou a estruturação deste subsistema de forma a permitir o reuso de uma mesma *HTTP Method Stub Library* em diversas implementações deste subsistema.

## 5.5 Prova de Conceito - Instanciando a Arquitetura AutoREST

Como prova de conceito da arquitetura AutoREST, foi desenvolvido um sistema de geração de APIs REST baseado na arquitetura proposta<sup>1</sup>, conforme as decisões de projeto delineadas na Seção ???. Este sistema utiliza as tecnologias de Diagramas de Classes UML, JSON Schema, Java, Node.js e MongoDB.

Os componentes de sistema foram implementados na linguagem de programação Java, com exceção do subsistema Execution Core, que foi desenvolvido em JavaScript. Cada um dos subsistemas foi implementado de forma autônoma, e suas interações são realizadas através de arquivos, garantindo o cumprimento da RQ05.

A notação utilizada para modelos de configuração foi a de Diagramas de Classes UML anotados conforme Subseção ??, utilizando um subconjunto da linguagem JSON Schema conforme a definição de ??). O artefato de entrada do subsistema instância de *Configuration Model Reader* é um XML gerado pela ferramenta Astah. A utilização de UML como linguagem gráfica para o modelo de configuração garante o cumprimento da RQ01 (??).

A notação utilizada como PFIS foi um conjunto de definições JSON Schema, conforme Subseção ?? (??). Este modelo é gerado automaticamente pelo sistema a partir do modelo de configuração, garantindo o cumprimento da RQ02.

A API REST é gerada utilizando a linguagem Node.js. Um compilador foi criado utilizando a ferramenta BYACC/J (??), utilizando chamadas de função a uma biblioteca que contém blocos de código parametrizáveis representando os métodos HTTP. Além disto, a API REST fará uso da biblioteca Mongoose para permitir a persistência de objetos a partir do SGBD MongoDB. Desta forma, é garantido o cumprimento das RQ03 e RQ04.

### 5.5.1 Modelo de Configuração - Orientações de Modelagem

Para que seja possível a conversão automática de um modelo de configuração, é necessário que este tenha uma notação formal estabelecida de acordo com regras específicas (conforme Seção ??). Nesta seção serão apresentadas as orientações para a modelagem de um Diagrama de Classes UML anotado, utilizando a ferramenta Astah, para que este sirva como modelo de configuração para a ferramenta AutoREST implementada como prova de conceito. O uso de JSON Schema é majoritariamente baseado em ??) e ??). Todas as Regras de Modelagem serão indexadas como “RM##” e as Regras de Conversão

<sup>1</sup> Disponível em <<https://autoREST-site.herokuapp.com/>>

serão indexadas como “RC##”. A quebra de qualquer uma das RMs definidas nesta seção invalida o resultado da execução da ferramenta AutoREST implementada como prova de conceito, visto que implica na não-conformidade com um pré-requisito do uso da ferramenta.

#### 5.5.1.1 Classe e Atributos

Um modelo simples que pode ser utilizado como exemplo é uma única classe, sem associações, contendo apenas atributos não anotados. A Figura ?? apresenta uma classe contendo atributos de todos os tipos que são suportados pela ferramenta.

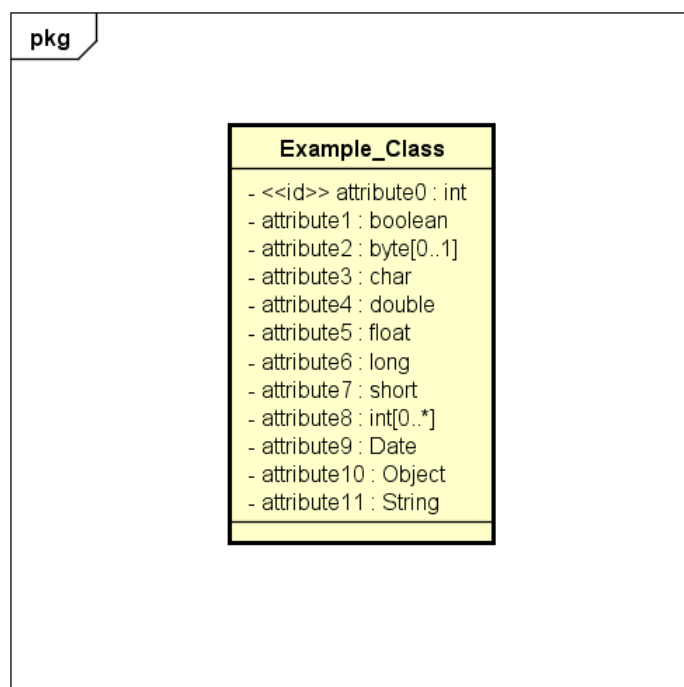


Figura 7 – Classe UML - Astah

Esta classe contém doze atributos numerados de 0 a 11, sendo o primeiro (*attribute0*) o identificador da classe. As seguintes regras de modelagem são estabelecidas a partir desta figura:

- RM01: Todo elemento Classe deve ter um nome único dentro do escopo do diagrama, maior do que um caractere, sem espaços (utilizar CamelCase ou `__underscore`).
- RM02: Todo elemento Classe deve ter um e apenas um atributo identificado com o Esteriótipo «id», representando o atributo identificador da classe. Caso nenhum atributo seja identificado com este esteriótipo, um atributo identificador será gerado pela ferramenta AutoREST durante a conversão para JSON Schema.

- RM03: Todo elemento Atributo deve ter um nome único dentro do escopo da classe, maior do que um caractere, sem espaços (utilizar CamelCase ou `__underscore`).
- RM04: A visibilidade (modificador de acesso) de um elemento Atributo será ignorado pela ferramenta, e portanto pode ser demarcado com quaisquer opções disponíveis.
- RM05: Um elemento Atributo pode assumir os seguintes tipos: *int*; *boolean*; *byte*; *char*; *double*; *float*; *long*; *short*; *Date*; *Object*, e; *String*. Também pode ser assumido como tipo quaisquer outras Classes disponíveis no diagrama. Os tipos *double* e *float* serão considerados como o mesmo tipo; da mesma forma, os tipos *int*, *long* e *short* serão considerados como o mesmo tipo.
- RM06: Um elemento Atributo com multiplicidade 0..1 será considerado como opcional.
- RM07: Um elemento Atributo com multiplicidade diferente de 0..1 ou 1 será considerado um vetor do tipo especificado, tal que o primeiro valor representa o tamanho mínimo do vetor (utilizar 0 para valor mínimo não definido) e o segundo valor representa o valor máximo do vetor (utilizar \* para valor máximo não definido).

A partir destas regras de modelagem, podem ser estabelecidas as seguintes regras de conversão para notação JSON Schema, utilizada como PFIS:

- RC01: Cada elemento Classe será convertido em um elemento de tipo *object* no campo *definitions*, com o mesmo nome da Classe em questão.
- RC02: Cada elemento Atributo será convertido em um elemento no campo *properties* da classe relacionada, com o mesmo nome do Atributo em questão.
- RC03: Atributos de tipo *int*, *short*, *long* e *byte* serão representados pelo tipo *integer*. Adicionalmente, Atributos de tipo *byte* serão marcados com as restrições de integridade de valor mínimo 0 e valor máximo 255.
- RC04: Atributos de tipo *boolean* mantém o mesmo tipo na representação em PFIS.
- RC05: Atributos de tipo *char*, *String* e *Date* serão representados pelo tipo *string*. Adicionalmente, atributos de tipo *char* serão marcados com a restrição de integridade de tamanho máximo 1, e atributos de tipo *Date* serão marcados com a restrição de integridade *“pattern”* : *“ $\wedge d\d\d d-(0?[1-9]/1[0-2])-(0?[1-9]/[12][0-9]/3[01]) (00/[0-9]/1[0-9]/2[0-3]):([0-9]/[0-5][0-9]):([0-9]/[0-5][0-9])$$ ”*<sup>2</sup>.

<sup>2</sup> Esta funcionalidade, apesar de especificada, não foi implementada na ferramenta prova de conceito.

- RC06: Atributos de tipo *Object* ou atributos tipados com outras Classes disponíveis no diagrama serão representados pelo tipo JSON Schema *object*. Adicionalmente, atributos tipados com outras Classes disponíveis no diagrama terão como propriedade a referência da definição da respectiva Classe.
- RC07: Todos os atributos serão definidos como dependentes do atributo identificador.
- RC08: Atributos com multiplicidade mínima maior do que 0 serão definidos como requeridos.
- RC09: Atributos com multiplicidade máxima maior do que 1 serão definidos como vetores.

#### 5.5.1.2 Restrições de integridade de atributos

Restrições de integridade de atributos devem ser anotados diretamente nos atributos com a utilização de *tags* no formato JSON Schema. Todas as restrições de integridade aqui definidas seguem a semântica definida por ??). As restrições de integridade válidas são ditadas pelas regras de construção a seguir:

- RM08: Atributos de tipos *int*, *short* e *long* podem ser anotados com as *tags*: *min*, *max*, *exMin* e *exMax*.
- RM09: Atributos de tipo *byte* e *Date* não podem ser anotados, tendo em vista que estes tipos já consideram restrições de integridade próprias.
- RM10: Atributos de tipo *boolean* não podem ser anotados, dada a natureza primitiva deste tipo.
- RM11: Atributos de tipo *char* e *String* podem ser anotados com a *tag*: *pattern*. Adicionalmente, atributos de tipo *String* também podem ser anotados com as *tags*: *minLength* e *maxLength*.
- RM12: Atributos de tipo *Object* não podem ser anotados, e servem como *wildcard*. Para definições de objetos mais complexos, utilizar elementos do tipo Classe.
- RM13: Atributos tipados com outras Classes disponíveis no diagrama não podem ser anotados, visto que estes atributos farão referência às restrições de integridade da Classe referenciada.
- RM14: Atributos com multiplicidade diferente de 0..1 ou 1 (tipo vetor) podem ser anotados com a *tag*: *uniqueItems*, de valor booleano (*true* ou *false*), representando a restrição de que todos os itens do vetor sejam diferentes entre si.

Estas *tags* serão convertidas diretamente para as restrições de integridade equivalentes em JSON Schema, gerando uma única regra de conversão:

- RC10: Todas as *tags* de elementos do tipo Atributo serão convertidas em suas respectivas restrições de integridade JSON Schema <sup>3</sup>.

A Figura ?? apresenta a parte da tela do Astah referente às *tags*, com anotações referentes ao *attribute11* da classe representada na Figura ??.

Base	Stereotype	Constraint
Language	TaggedValue	Hyperlink
Name	Value	
minLength	3	
maxLength	140	
pattern	/^[a-zA-Z]*\$/	
Add		Delete
Up		Down

Figura 8 – *Tags* relativas ao *attribute11* da Figura ?? - Astah

A partir das regras de conversão apresentadas até este ponto, a Classe representada na Figura ?? terá o formato em JSON Schema apresentado no Anexo ??.

### 5.5.1.3 Multiplicidade de Associações e Navegabilidade

Associações simples entre duas classes podem ser representadas utilizando as notações de multiplicidade e navegabilidade. As restrições descritas nesta seção são detalhadas conforme ??).

- RM15: Associações podem ser navegáveis ou não-navegáveis. Associações anotadas como tendo navegabilidade não-definida serão considerados não-navegáveis.
- RM16: Associações podem ter multiplicidade 1, 0..1, 0..\*, 1..\* ou \*. Associações de multiplicidade não definida serão consideradas como tendo multiplicidade 1.

<sup>3</sup> Esta funcionalidade, apesar de especificada, não foi implementada na ferramenta prova de conceito.



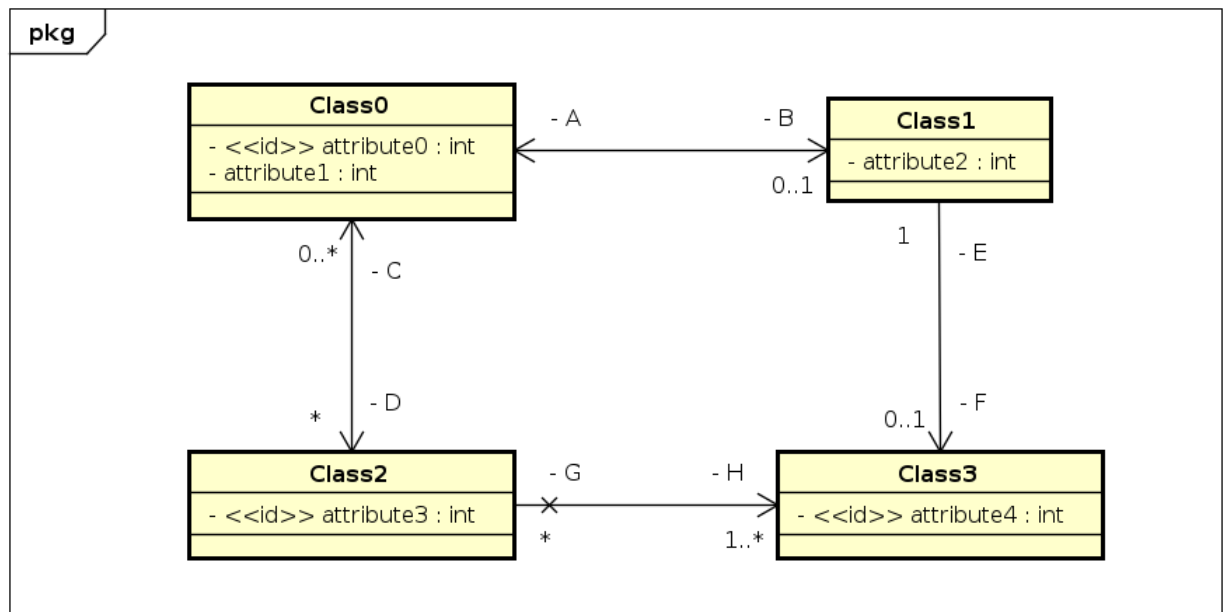


Figura 9 – Exemplo de Diagrama de Classes UML com Associações Simples - Astah

- RM17: Vértices não-navegáveis de associações são considerados como tendo multiplicidade 1, independentemente da multiplicidade anotada no diagrama.

A partir destas regras de modelagem, podem ser estabelecidas as seguintes regras de conversão para PFIS (um vértice é considerado como tendo a mesma navegabilidade e multiplicidade da ponta oposta de uma associação na qual está relacionado):

- RC11: Classes localizadas em vértices não-navegáveis não terão referência à classe associada.
- RC12: Classes localizadas em vértices navegáveis terão referência à classe associada conforme multiplicidade do vértice.
- RC13: Vértices de multiplicidade 1 ou 0..1 serão representados por uma referência ao atributo identificador da classe associada à sua especificação. Adicionalmente, vértices de multiplicidade 0..1 terão este atributo identificador representado como opcional.
- RC14: Vértices de multiplicidade 0..\*, \* ou 1..\* serão representados por um vetor de referências ao atributo identificador da classe associada à sua especificação. Adicionalmente, vértices de multiplicidade 1..\* terão adicionada a este vetor a restrição de integridade de tamanho mínimo 1.

A Figura ?? apresenta um exemplo de diversas possíveis aplicações de associações simples à um diagrama. O resultado da conversão deste diagrama é apresentado no Anexo

???. A seguir é apresentada a aplicação das regras de conversão descritas até este ponto sobre o diagrama da Figura ??:

- Cada uma das classes tem um elemento criado em *definitions*, conforme RCs 1-10.
- *Class1* tem um atributo identificador criado, visto que não possui nenhum atributo anotado com o esteriótipo «id», conforme RM02.
- *Class0* tem adicionado as suas propriedades o atributo identificador de *Class1* anotado como opcional, dado o vértice navegável B de multiplicidade 0..1, conforme RC13.
- *Class0* tem adicionado as suas propriedades um vetor do atributo identificador de *Class2* anotado como opcional, dado o vértice navegável D de multiplicidade \*, conforme e RC14.
- *Class1* tem adicionado as suas propriedades o atributo identificador de *Class0* anotado como opcional, dado o vértice navegável A de multiplicidade não definida, conforme RM16 e RC13.
- *Class1* tem adicionado as suas propriedades o atributo identificador de *Class3* anotado como opcional, dado o vértice navegável F de multiplicidade 0..1, conforme RC13.
- *Class2* tem adicionado as suas propriedades um vetor do atributo identificador de *Class0* anotado como opcional, dado o vértice navegável C de multiplicidade 0..\*, conforme RM15 e RC14.
- *Class2* tem adicionado as suas propriedades um vetor do atributo identificador de *Class3* com tamanho mínimo 1, dado o vértice navegável H de multiplicidade 1..\*, conforme RC14.
- *Class3* não tem adicionado as suas propriedades um vetor do atributo identificador de *Class2*, dado o vértice não navegável G de multiplicidade \*, conforme RMs 15 e 17.
- *Class3* não tem adicionado as suas propriedades o atributo identificador de *Class1*, dado o vértice de navegabilidade não definida E de multiplicidade 1, conforme RM 15.

#### 5.5.1.4 Associações de Generalização/Especialização

Associações de Generalização/Especialização (Herança) permitem o reuso e a extensão de classes representadas no modelo de configuração. As restrições aplicadas a este tipo de associação são:

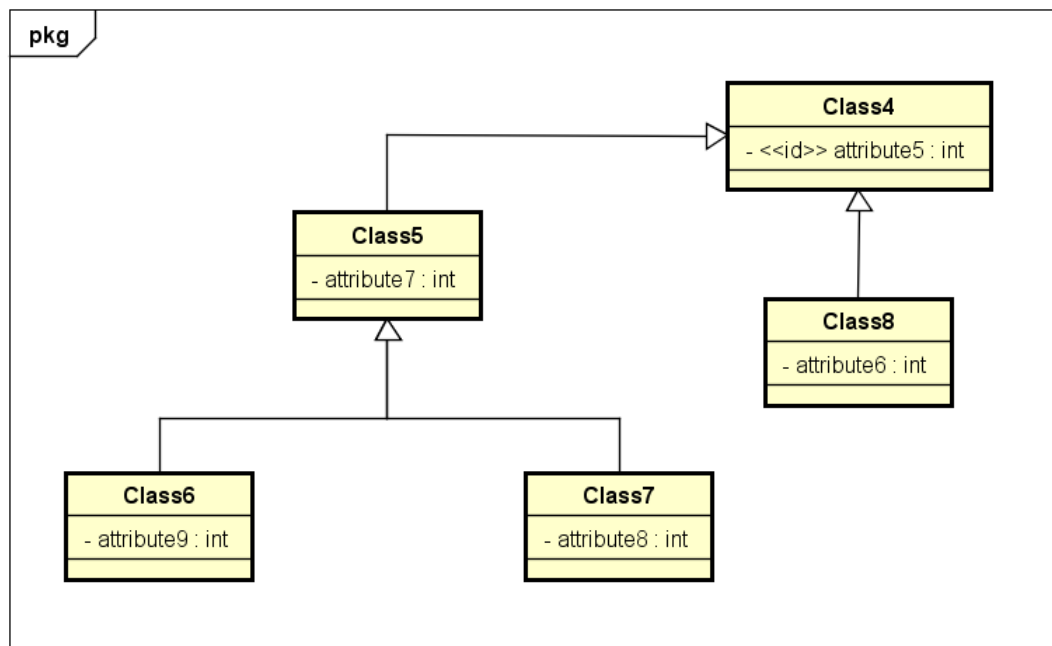


Figura 10 – Exemplo de Diagrama de Classes UML com Associações de Generalização/Especialização - Astah

- RM18: Uma classe pode ser uma Especialização de uma e apenas uma outra classe, não sendo permitida a modelagem de herança múltipla para fins desta ferramenta.
- RM19: Todas as associações de Generalização/Especialização são consideradas Parciais e Não-exclusivas. Isto se dá devido a complexidade de modelar estas características de forma explícita na ferramenta Astah.
- RM20: Uma classe “filha” não pode ter atributos identificadores, sendo estes necessariamente herdados da classe “pai”.

É necessária uma única regra de conversão para que este tipo de associação seja contemplado:

- RC15: A classe “filha” de uma associação de Realização/Especialização será representada na PFIS pela composição de seus atributos e uma referência à classe “pai”.

A Figura ?? apresenta um exemplo válido de diagrama de classes utilizando associações de Especialização/Generalização. O JSON Schema relacionado é apresentado no Anexo ??.

### 5.5.1.5 Associações de Agregação

Associações de Agregação permitem a representação de posse de uma classe sobre outra. Dois tipos de agregação são permitidos: Agregação Simples e Agregação por Composição. As restrições aplicadas a este tipo de associação são:

- RM21: Associações de Agregação Simples representam a posse de uma classe sobre outra, porém mantendo a independência funcional de cada uma das classes associadas.
- RM22: Associações de Agregação por Composição representam a posse de uma classe sobre outra, sendo a classe “possuída” uma definição interna da classe “possessora”. Desta forma, a classe possuída só pode existir no contexto da classe possensora.
- RM23: O vértice possuído de Associações de Agregação pode ter multiplicidade 1, 0..1, 0..\*, \* ou 1..\*.
- RM24: O vértice possessor de Associações de Agregação deve ter multiplicidade 1.
- RM26: Não é comportada navegabilidade direcionada da classe possuída para a classe possensora, por limitação da notação gráfica da ferramenta Astah. Por esta razão, assume-se sempre navegabilidade unidirecional para associações de agregação.

São estabelecidas as seguintes regras de conversão para que este tipo de associação seja contemplado:

- RC16: O vértice possuído de Associações de Agregação não será representado.
- RC17: O vértice possessor de Associações de Agregação Simples será representado por uma referência à classe possuída, ou um vetor de referências à classe possuída, definido de acordo com a multiplicidade do vetor, de forma similar a RCs 13-14.
- RC18: O vértice possessor de Associações de Agregação por Composição será representado por uma definição interna de classe, seguindo as regras de multiplicidade das RCs 13-14 para decisão de uso de definição única ou vetor.
- RC19: Classes possuídas de Associações de Agregação por Composição não serão representadas nas definições base.

A Figura ?? apresenta um diagrama de classes que utiliza os dois tipos de associação de agregação, além de um tipo de associação simples para que seja possível visualizar a diferença entre estes. O Anexo ?? apresenta o JSON Schema resultante da conversão deste diagrama.

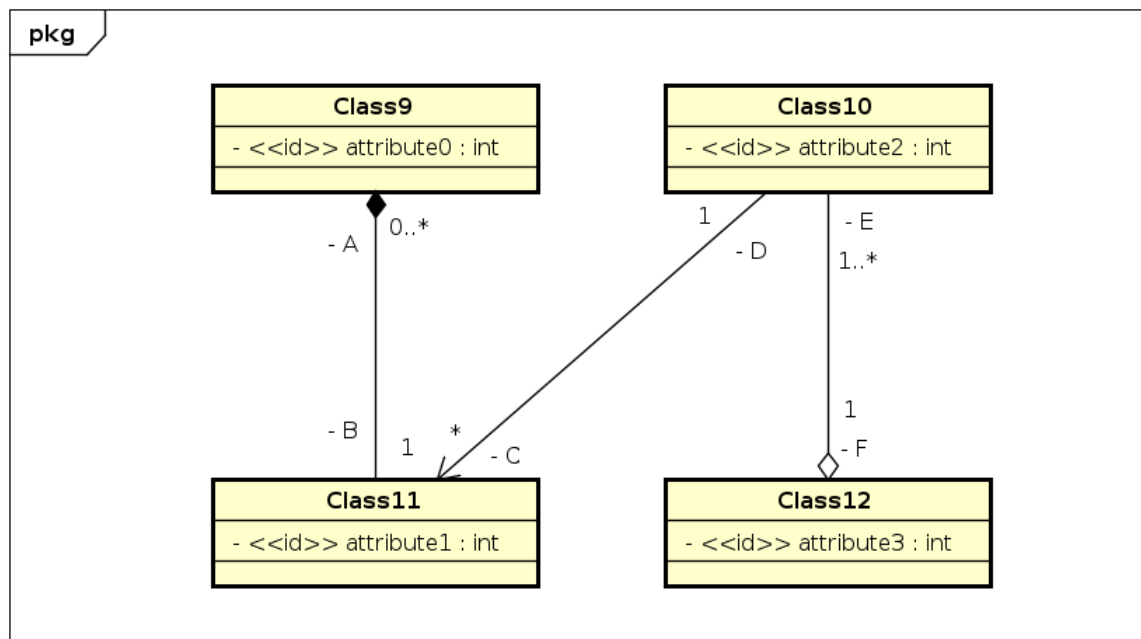


Figura 11 – Exemplo de Diagrama de Classes UML com Associações de Agregação - Astah

#### 5.5.1.6 Classes Associativas

Classes Associativas podem ser utilizadas para representar associações entre classes que contém atributos próprios. A seguinte restrição se aplica:

- RM27: Classes Associativas não podem possuir atributo identificador; os identificadores das classes associadas são utilizados.

As regras de conversão referentes a esta restrição são:

- RC20: Cada uma das classes associadas irá receber um vetor de referências à objetos da classe associativa.
- RC21: Um objeto será criado em *definitions* para a classe associativa, utilizando como identificador a composição dos identificadores das classes associadas.

A Figura ?? apresenta um diagrama de classes exemplo. O Anexo ?? apresenta o JSON Schema resultante da conversão deste diagrama.

### 5.5.2 Gramática BNF para Geração de Código a partir de JSON Schema

No trabalho de ??) é definida e validada formalmente uma gramática na forma de Backus-Naur (BNF) para a estrutura JSON Schema (??). Nesta seção, serão exploradas

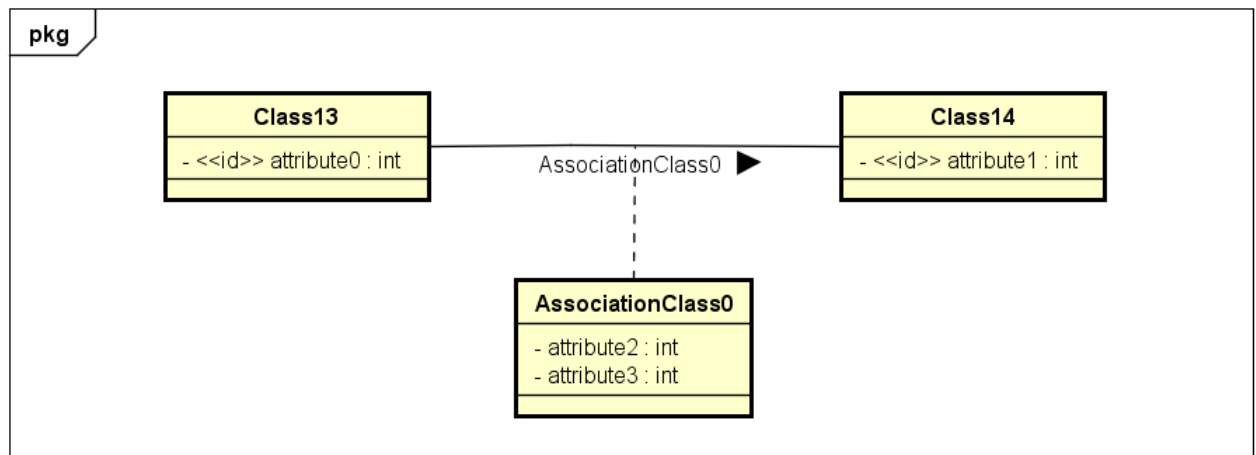


Figura 12 – Exemplo de Diagrama de Classes UML com Classe Associativa - Astah

as partes desta gramática utilizadas no subsistema *REST API Generator* e seu significado semântico.

#### 5.5.2.1 Sintaxe

A gramática BNF foi usada como entrada para a geração de um *parser* na ferramenta BYACC/J (??). O subconjunto de definições que serão relevantes para o subsistema está contido na definição *defs*, que corresponde à propriedade *definitions* de um JSON Schema. Cada *defs* possui no mínimo um *kSch*, que é composto de um nome (*kword*) e uma ou mais restrições (*res*). Conforme segmento extraído de (??):

```

defs := “definitions”: { kSch (, kSch)* }
kSch := kword: { JSch }
JSch := ( res (, res)* )
res := type | strRes | numRes | arrRes | objRes | multRes | refSch | title |
description
  
```

A gramática utilizada serve para a validação sintática completa de um JSON Schema; porém somente algumas das definições terão significado semântico para o subsistema, conforme seção a seguir. A gramática completa está no Anexo ??.

#### 5.5.2.2 Semântica

Nesta seção, serão apresentadas as derivações *kSch*, que deriva de *defs*, consideradas no processo de criação da API REST, assim como suas derivações utilizadas. Em sequência, serão apresentados os processos de geração do código fonte final, subseções ?? e ??.

#### 5.5.2.2.1 Derivações de *kSch*

Cada *kSch* será considerada a definição de um *resource* a ser fornecido pela API. Logo, a derivação *kword* se tornará o nome deste *resource*. A derivação *JSch* será utilizada para a geração de métodos HTTP disponibilizados para o *resource*, assim como para a criação de um *model* do módulo Mongoose de Node.js, que subsequentemente irá fazer as interações com o MongoDB. As derivações utilizadas para um *JSch* de um *resource* são:

- **type**: restringe o tipo do objeto sendo definido. Para um *resource* o valor deverá ser *object*; outros valores serão considerados inválidos
- **objRes**: restrições do *resource*; serão consideradas as definições na Subseção ??

#### 5.5.2.2.2 Derivações de *objRes*

A diretiva *objRes* representa as restrições relacionadas a definição de um objeto. Ela pode ser usada tanto para definição de *resources* quanto para a definição de propriedades do tipo objeto. As derivações consideradas válidas são as seguintes:

- **properties**: irá conter as propriedades do objeto sendo definido; será melhor detalhada na Seção ??.
- **required**: define quais propriedades são obrigatórias
- **dependencies**: será considerada nesta diretiva somente a derivação contendo um *kArr*, que é o nome de uma propriedade e suas dependências. Todas as propriedades serão dependentes da propriedade definida como chave primária, para que a chave primária seja composta será necessário que as propriedades sejam dependentes de mais de uma propriedade

#### 5.5.2.2.3 Derivações de *properties*

As propriedades de um objeto serão todas as derivações *kSch* que fazem parte da diretiva *properties* relacionada ao mesmo. Porém, as derivações de um *kSch* terão um significado diferente quando aplicadas a uma propriedade do que quando aplicadas a um *resource*. A derivação *kword* será o nome da propriedade e para a derivação *JSch* serão consideradas válidas somente as derivações listadas a seguir:

- **type**: restringe o tipo da propriedade sendo definida. Caso nenhum valor seja informado assume-se *object*. Valores considerados válidos são: *string*, *integer*, *number*, *boolean*, *object* e *array*.

- **strRes**: será aceita quando o *type* for *string* e serão consideradas válidas as seguintes derivações:
  - **minLength**: tamanho mínimo do valor da propriedade
  - **maxLength**: tamanho máximo do valor da propriedade
  - **pattern**: expressão regular para validação do valor<sup>4</sup>
- **numRes**: será aceita quando o *type* for *integer* ou *number* e serão consideradas válidas as seguintes derivações:
  - **minimum**: valor mínimo da propriedade
  - **exclusiveMinimum**: valor booleano. Indica se o valor mínimo está incluído ou não no intervalo de validação
  - **maximum**: valor máximo da propriedade
  - **exclusiveMaximum**: valor booleano. Indica se o valor máximo está incluído ou não no intervalo de validação
- **arrRes**: será aceita quando o *type* for *array*, será melhor detalhada na Subseção ??
- **objRes**: será aceita quando o *type* for *object*. Serão aplicadas as definições da Subseção ??
- **refSch**: referências a outras definições no arquivo JSON Schema que está sendo processado. Estas referências possuirão significados diferentes, de acordo com seu contexto, na disponibilização dos *resources* pela API, conforme Seções ?? e ??

#### 5.5.2.2.4 Derivações de *arrRes*

As restrições desta diretiva são relacionadas ao *type array*. Serão aceitas as seguintes derivações:

- **items**: definição dos tipos do *array*. Pode ser derivada de duas formas:
  - **sameitems**: determina que todos os itens do *array* possuem a mesma definição, que é determinada por um *JSch*. Para esta definição são aceitas as mesmas derivações de *JSch* da Subseção ??
  - **varitems**: determina que o *array* pode ter itens de definições variadas. Esta derivação é considerada inválida pelo gerador
- **minitems**: indica o numero mínimo de itens do *array*
- **maxitems**: indica o numero máximo de itens do *array*

<sup>4</sup> Esta funcionalidade apesar de especificada, não está presente na ferramenta prova de conceito



#### 5.5.2.2.5 Geração dos *models* Mongoose

Para a criação dos *models* de cada *resource* são usadas as seguintes regras de criação:

- Cada propriedade do *resource* no JSON Schema irá resultar em uma propriedade do *model* Mongoose
- Uma propriedade do tipo *integer* no JSON Schema resultará em uma propriedade Mongoose com propriedades *type: Number* e *integer: true*
- Uma propriedade do tipo *number* no JSON Schema resultará em uma propriedade Mongoose com a propriedade *type: Number*
- Uma propriedade do tipo *boolean* no JSON Schema resultará em uma propriedade Mongoose com a propriedade *type: Boolean*
- Uma propriedade do tipo *string* no JSON Schema resultará em uma propriedade Mongoose com a propriedade *type: String*
- Uma propriedade do tipo *array* no JSON Schema resultará em uma propriedade Mongoose com uma propriedade *any* que será um *array* contendo o *SchemaType* Mongoose dos elementos
- Uma propriedade do tipo *object* no JSON Schema resultará em uma propriedade Mongoose com a propriedade *type: Object*. Se esta propriedade *object* possuir definições JSON Schema, esta serão criadas no Mongoose conforme estas mesmas regras
- Todas as propriedades do JSON Schema especificadas como *required* receberão uma propriedade Mongoose *required: true*
- Todas as propriedades do JSON Schema dos tipos *number* e *integer* que possuírem definições de máximo e mínimo receberão propriedades Mongoose correspondentes
- Propriedades do JSON Schema do tipo *string* que possuírem definições de tamanho máximo, tamanho mínimo e expressão regular receberão propriedades Mongoose correspondentes<sup>5</sup>

As definições *dependencies* do JSON Schema são interpretadas como se referindo a chave primária de um *resource*. Logo, a propriedade da qual todas as outras dependem será considerada a chave primária. O MongoDB considera como chave primária de suas *collections* a propriedade *\_\_id* e este comportamento é assimilado pelo Mongoose.

<sup>5</sup> Esta funcionalidade apesar de especificada, não está presente na ferramenta prova de conceito

Se for identificado que um *resource* possui uma chave primária composta por mais de uma propriedade, ele receberá um *index unique* que contenha todas estas propriedades que compõem a chave, fazendo-o assim respeitar a restrição de unicidade da chave primária.

Para *arrays* e propriedades que possuam referências (**refSch**) a outros *resources*, a criação do *model* sofrerá alterações. Se um *array* possuir itens que são referências (**refSch**) a outro *resource*, então o que será armazenado no MongoDB serão os valores das chaves primárias deste *resource*, e o *model* Mongoose possuirá propriedades correspondentes. Quando uma propriedade for referência a outro *resource* isto implicará em uma herança. Neste caso o *model* que contém esta propriedade será armazenado na mesma *collection* no MongoDB e possuirá seu *Schema* de Mongoose representando o relacionamento de herança.

Para que os *resources* fornecidos pela API não contenham propriedades não desejadas, e ainda seja possível utilizar a chave primária com seu nome original, serão utilizados métodos de acesso virtuais do Mongoose. Como exemplo temos no Listing ?? o atributo *attribute0* da classe apresentada na Figura ?. O código completo para o *model* desta classe está no Anexo ??

---

**Listing 1** Exemplo métodos virtuais Mongoose

---

```
1 Example_ClassSchema.virtual('attribute0').get(function() {
2   return this._id;
3 });
4 Example_ClassSchema.virtual('attribute0').set(function (value) {
5   this._id = value;
6 });
```

---

Além da propriedade *\_id* do MongoDB, o Mongoose também adiciona outras propriedades que não estariam de acordo com o JSON Schema do *resource*. Para evitar o envio destas propriedades, todos os *models* possuirão o método *cleanObject*, que terá a função de retornar um objeto sem estas propriedades. Um exemplo aplicado a classe *Example\_Class* pode ser visto no Listing ??

#### 5.5.2.2.6 Geração dos métodos HTTP

Cada *resource* terá seus métodos HTTP dentro de um *middleware* do tipo *Router*, que será utilizado pelo módulo *Express* do *Node.js* para gerenciar as *requests*.

Os métodos HTTP gerados para um *resource* serão todos os identificados como necessários para realizar as operações CRUD, que são: GET, HEAD, POST, PUT, PATCH e DELETE.

**Listing 2** Exemplo método *cleanObject*

```

1  Example_ClassSchema.methods.cleanObject = function() {
2      var doc = this.toObject({ virtuals: true });
3      delete doc.__v;
4      delete doc._id;
5      delete doc.id;
6      return doc;
7  };

```

Método	URI	Função
GET	/resourceName/[?propName=valueX]	Retornar todos os <i>resources</i> da <i>collection</i>
GET	/resourceName/:identifier	Retornar o <i>resource</i> com o <i>identifier</i> igual ao parâmetro na URI
HEAD	/resourceName/:identifier	Verificar se o <i>resource</i> com o <i>identifier</i> igual ao parâmetro indicado existe, sem retorná-lo
POST	/resourceName/	Inserir ou modificar totalmente o <i>resource</i> contido no <i>message body</i> da <i>request</i>
PUT	/resourceName/:identifier	Inserir ou modificar totalmente o <i>resource</i> indicado pelo parâmetro da URI, utilizando os dados contidos no <i>message body</i> da <i>request</i>
PATCH	/resourceName/:identifier	Modificar parcialmente o <i>resource</i> indicado pelo parâmetro da URI, utilizando os dados contidos no <i>message body</i> da <i>request</i>
DELETE	/resourceName/:identifier	Excluir o <i>resource</i> com o <i>identifier</i> igual ao parâmetro indicado

Tabela 1 – *Resource* Simples - Métodos HTTP e URIs

Um *resource* com uma chave primária simples possuirá os *endpoints* apresentados na Tabela ??, onde: *resourceName* é o nome do *resource*, e; *identifier* é a chave primária do *resource*. As *query strings* nas URIs indicadas são opcionais e os parâmetros esperados seriam *propName*, que é o nome de uma propriedade, e *valueX*, que é o valor desejado na propriedade; é possível mais de um parâmetro de filtro na *query string*.

Se um *resource* possuir uma chave composta, seus *endpoints* ao invés de aceitarem um *identifier* irão esperar que os valores das chaves sejam enviados via *query string*; e o comportamento gerado para o caso de nem todas as chaves serem enviadas na *query string* será de um erro retornado ao cliente. Estas alterações nos *endpoints* são apresentadas na Tabela ??; a função dos métodos permanecem as mesmas.

Método	URI	Query string
GET	/resourceName/[?propName=valueX]	Opcional, utilizado somente para filtro, aplicável a todas as propriedades
HEAD	/resourceName/[?propName=valueX]	Devem ser informados os valores da chave primária
POST	/resourceName/	Não aplicável (mantém comportamento utilizando o <i>message body</i> )
PUT	/resourceName/[?propName=valueX]	Devem ser informados os valores da chave primária
PATCH	/resourceName/[?propName=valueX]	Devem ser informados os valores da chave primária
DELETE	/resourceName/[?propName=valueX]	Devem ser informados os valores da chave primária

Tabela 2 – *Resource* com chave composta - Métodos HTTP e URIs

Quando um *resource* com chave primária simples possuir propriedades complexas ou do tipo *array*, estas poderão ser acessadas diretamente através de *endpoints* dedicados a elas se o *endpoint* já possuir o *identifier* do *resource*. Estes *endpoints* serão formados utilizando o *endpoint* do *resource* e um sufixo com o nome da propriedade, conforme Tabela ???. Propriedades complexas que estão contidas dentro de outras propriedades complexas possuirão seus *endpoints* também, sendo estes criados com o mesmo procedimento: adicionando um sufixo com o nome da propriedade aos *endpoints* já existentes.

### 5.5.3 Funcionamento da API REST gerada

A API gerada poderá ser utilizada através de um arquivo “*api.js*” que será gerado após todos os *models* e *routers* serem gerados. Neste arquivo, o servidor *Express* será criado e iniciado e os *routers* gerados serão adicionados como *middleware* do servidor. Também constarão no arquivo a criação da conexão com o MongoDB e outras utilidades que permitam uma melhor utilização dos recursos da linguagem.

Ao final da geração da API, é possível utilizá-la através do arquivo *api.js*, usando-o como parâmetro do *runtime* de Node.js em um terminal e estando na pasta onde o arquivo se encontra; por exemplo: **node api.js**

Após executar este comando o usuário será informado de que a API está executando e também os *endpoints* de GET de cada *resource*, conforme exemplo da Figura ??.

O servidor *Express* criado irá processar as *requests* que forem recebidas, irá identificar a URI solicitada e o método HTTP e então irá encaminhar a *request* para o *router middleware* que, por sua vez, irá processar a *request* e realizar a operação CRUD correspondente utilizando seu(s) respectivo(s) *model(s)*. Esta sequência de processamento é

Método	URI	Query string
GET	/resourceName/:identifier/propName	Retornar o conteúdo da propriedade <i>propName</i> do <i>resource</i> com o <i>identifier</i> igual ao parâmetro na URI
HEAD	/resourceName/:identifier/propName	Verificar se o <i>resource</i> com o <i>identifier</i> igual ao parâmetro indicado existe e possui a propriedade <i>propName</i> , sem retornar seu valor
PUT	/resourceName/:identifier/propName	Inserir ou modificar totalmente o conteúdo da propriedade <i>propName</i> no <i>resource</i> indicado pelo parâmetro da URI, utilizando os dados contidos no <i>message body</i> da <i>request</i>
PATCH	/resourceName/:identifier/propName	Modificar parcialmente o conteúdo da propriedade <i>propName</i> do <i>resource</i> indicado pelo parâmetro da URI, utilizando os dados contidos no <i>message body</i> da <i>request</i> . Não aplicável a propriedades <i>array</i>
DELETE	/resourceName/:identifier/propName	Excluir a propriedade <i>propName</i> do <i>resource</i> com o <i>identifier</i> igual ao parâmetro indicado

Tabela 3 – Endpoints de propriedades complexas e arrays

```

Listening on http://localhost:5000
Available resources at:
GET http://localhost:5000/api/example_class/
GET http://localhost:5000/api/avaliacao/
Connection to MongoDB established

```

Figura 13 – Exemplo de servidor de API iniciado

mostrada no diagrama de sequências nas Figuras ?? e ??.

#### 5.5.4 Demonstração de resultados

Para demonstrar a instanciación da solução AutoREST, uma aplicação *web*, disponível no endereço <<https://autoREST-site.herokuapp.com/>>, foi criada para a geração de APIs usando os componentes AutoREST, que são os seguintes:

- **astah-xml-parser**<sup>6</sup>: responsável por realizar a conversão de XMLs em JSON Schemas. Também tem a funcionalidade de extrair os pacotes presentes no XML, para

<sup>6</sup> <<https://github.com/AutoREST/astah-xml-parser>>

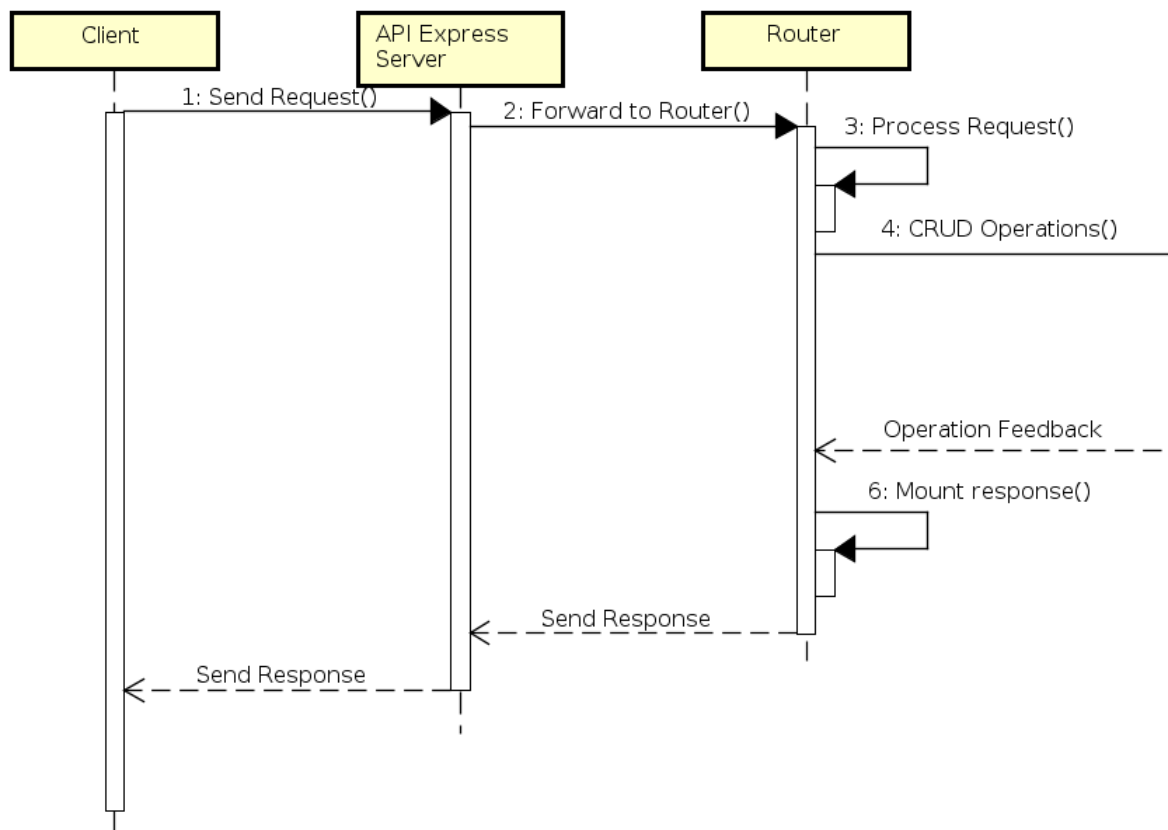


Figura 14 – Exemplo de processamento de *request* no servidor da API - Parte 1

que seja possível a escolha de qual pacote contém as informações sobre a qual a API deve ser gerada;

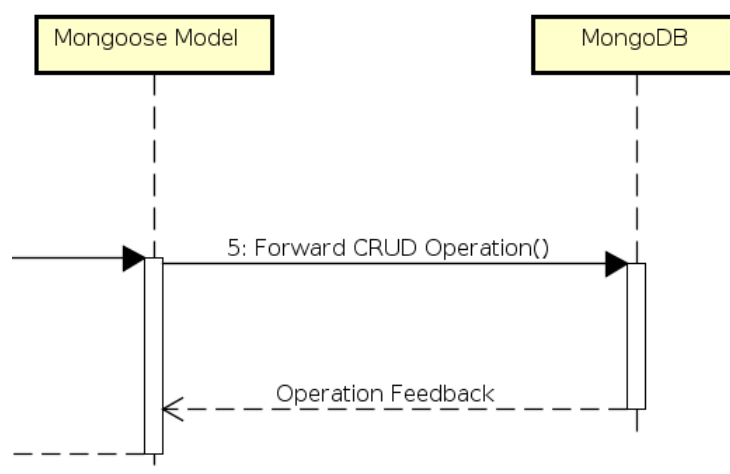
- ***rest-api-generator***<sup>7</sup>: responsável por gerar as APIs usando um JSON Schema e algumas opções informadas na interface com o usuário;
- ***autorest-site***<sup>8</sup>: aplicação *web* responsável por receber o arquivo de entrada para geração e algumas opções adicionais.

A interface com o usuário tem duas telas principais que são baseadas nos *mockups* das Figuras ?? e ??. Na tela da Figura ??, temos a entrada básica para a geração da API, onde:

- *API Name*: é o nome da API; é usado para dar um nome em alguns pontos da API, assim como no arquivo de retorno final;
- *Data Base Name*: nome do banco de dados usado na conexão com o MongoDB;

<sup>7</sup> <<https://github.com/AutoREST/rest-api-generator>>

<sup>8</sup> <<https://github.com/AutoREST/autorest-site>>

Figura 15 – Exemplo de processamento de *request* no servidor da API - Parte 2

API Name	<input type="text"/>
Data Base Name	<input type="text"/>
Repository URL	<input type="text"/>
Source File	<input type="button" value="Choose file"/>
<input type="button" value="Generate API"/>	

Figura 16 – Tela de entrada de dados para geração de API

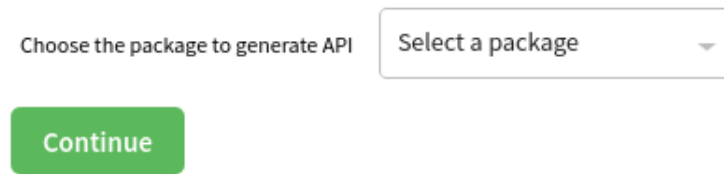


Figura 17 – Tela para seleção de *package* da API

- *Repository URL*: a URL onde a API será guardada; é somente para fins de documentação no *package.json* da API;
- *Source File*: arquivo contendo a especificação da API; aceita-se um XML da ferramenta Astah modelado conforme as regras de modelagem AutoREST ou um arquivo JSON Schema com restrições aceitas e coerentes com as regras de sintaxe e semântica AutoREST
- *Generate API*: botão que inicia a geração da API

Caso o *Source File* seja um XML que contenha mais de um *package* de classes, o componente *astah-xml-parser* irá retornar a lista de *packages* disponíveis para que o usuário escolha qual deve ser a fonte de sua API; este é o comportamento apresentado na tela da Figura ??.



## 6 Considerações Finais

Neste trabalho, foi proposta uma solução para o problema de geração automática de APIs REST que provejam as funcionalidades necessárias para que sejam executadas as operações básicas de manutenção de documentos (CRUD - *Create, Read, Update, Delete*), com o objetivo de utilizar a geração automática de código como uma forma de aumentar a confiabilidade e produtividade do processo de criação de software.

A solução AutoREST aplica os conceitos das metodologias de Engenharia de Software Baseada em Componentes, Desenvolvimento Dirigido por Modelos e Programação Generativa, concretizando heurísticas de cada uma destas metodologias em uma proposta de arquitetura de software. Além disto, propomos uma ferramenta como prova de conceito, que serve como instanciación desta arquitetura.

Este trabalho foi motivado pelo formalismo da arquitetura REST e a representatividade das notações JSON Schema e Diagrama de Classes UML. Através destas notações, foi possível a proposição de regras de modelagem que permitem a representação conceitual de uma API REST. Além disto, dada a estrutura da notação JSON Schema conforme ??), que é baseada na Backus-Naur Form (BNF), foi possível a proposição de um método para a conversão de um documento JSON Schema em código fonte da API representada.

A arquitetura proposta, além dos componentes descritos para a instanciación desta arquitetura em uma prova de conceito, foram construídos a partir do estudo da arquitetura REST, do *Hypertext Transfer Protocol* (HTTP) e das notações JSON Schema e UML, esta última particularmente no que concerne o conjunto da UML que é disponibilizado pela ferramenta Astah. Além destas tecnologias, foram estudadas a linguagem Node.js, o sistema gerenciador de bancos de dados MongoDB e o módulo Node.js Mongoose, que são utilizados pelas APIs REST geradas pela ferramenta proposta.

### 6.1 Trabalhos Relacionados

Dentre os trabalhos encontrados que tratam de geração de código automática, foram selecionados alguns que tratam especificamente de ferramentas para a geração automática de código a partir de definições em formato JSON Schema ou similar. Se destacaram em nossa pesquisa as ferramentas Heroics, Schematic, Swagger e Eve. Além disso, damos destaque as ferramentas DaemonX e Prmd, que utilizam alterações na especificação de APIs para gerar a documentação e versionamento delas automaticamente.

### 6.1.1 Heroics and Schematic

Heroics é um gerador que utiliza um JSON Schema que representa uma API para gerar clientes HTTP em Ruby para esta API (??). Outro gerador que utiliza uma entrada similar, em JSON Hyper-Schema (??), para gerar clientes HTTP em Go é o Schematic (??).

A principal diferença destas ferramentas para o AutoREST é o fato de gerarem apenas código cliente. Porém, o fato de utilizarem JSON Schema em seus processos pode significar que uma interação com elas seria algo que acrescentaria valor a solução AutoREST, permitindo assim que os clientes da API já fossem criados também de forma automática.

### 6.1.2 Prmd

Prmd (??) é uma ferramenta que permite realizar verificação e geração de documentação para APIs utilizando um JSON Schema que a descreve. O JSON Schema utilizado deve possuir algumas características descritas na documentação do Prmd.

Esta é outra ferramenta que utiliza o JSON Schema para um propósito diferente do AutoREST, pois gera documentação para uma API. Como as ferramentas anteriores, esta funcionalidade poderia ser alcançada com uma possível integração de um JSON Schema que descrevesse a REST API gerada.

### 6.1.3 Swagger

Swagger (??) é uma representação de REST APIs com um grande conjunto de ferramentas de apoio ao desenvolvimento e suporte, em várias das linguagens de programação atuais. Com uma definição Swagger, é possível a geração de documentação e geração de clientes. A especificação Swagger atualmente é gerenciada pela Open API Initiative (OAI) (??).

As definições Swagger e o grande número de ferramentas de geração disponibilizadas pela comunidade proporcionariam o escopo restante no processo de criação de APIs que não é explorado no AutoREST. Uma especificação Swagger gerada pelo AutoREST poderia proporcionar tanto a criação de documentações quanto de códigos clientes da API.

### 6.1.4 Eve

Eve (??) é um *framework* em Python para a disponibilização de REST APIs. É utilizada uma definição similar a JSON Schema que será interpretada e validada pela ferramenta Cerberus (??). Esta definição determina o *schema* do *resource* disponibilizado e os métodos HTTP disponíveis para o mesmo.

O *framework* do Eve proporciona uma experiência com uma interação menos explícita com o sistema gerenciador de banco de dados, deixando para o desenvolvedor a tarefa de especificar na gramática da ferramenta Cerberus todos os seus *resources* e artefatos relacionados.

A principal diferença para o AutoREST é que a única coisa que seria necessária criar seria o diagrama de classes, e com este a especificação dos *resources* seria criada e então a API seria gerada com esta especificação sem exigir intervenção no código gerado para que a API comece a ser utilizada.

### 6.1.5 DaemonX

DaemonX é uma plataforma extensível para o gerenciamento de evolução de documentos representativos de estruturas de dados. Dentre os usos desta plataforma está o de gerenciar a evolução de APIs REST com uma notação baseada no princípio de Arquiteturas Dirigidas por Modelos (MDA - *Model-Driven Architecture*) (??).

Similar ao Prmd, esta aplicação da plataforma DaemonX apresenta uma forma de representar e gerenciar a evolução de APIs REST, uma funcionalidade não disponível na solução AutoREST. Entretanto, por utilizar uma representação específica, apesar de possivelmente atingir maior detalhamento das APIs, não seria de fácil integração com soluções mais gerais como a AutoREST.

## 6.2 Conclusão

Nesta seção, são apresentadas respostas para as perguntas de pesquisa propostas no início deste documento utilizando a experiência obtida durante o desenvolvimento deste trabalho e implementação da ferramenta prova de conceito.

- QP1. Existe viabilidade na automação da geração de uma API REST utilizando uma abordagem baseada em MDD?

Esta automação é viável, desde que sejam compreendidos seus pontos positivos e negativos, que no decorrer deste trabalho podemos assumir como sendo:

#### – Positivos

- \* Para APIs não customizadas após a geração, há alto nível de compreensão e custo muito baixo após alterações no modelo
- \* Fácil transição de conhecimento sobre o funcionamento da API, sem a necessidade de olhar código fonte e outras estruturas não gráficas

#### – Contrás

- \* APIs customizadas tem seu código customizado perdido no momento de uma nova geração a partir do modelo
  - \* É necessário uma boa compreensão das regras de modelagem a serem utilizadas
- QP2. Quais os elementos necessários para a representação completa de uma API REST em um modelo gráfico para que este possa ser utilizado como modelo de configuração em uma abordagem MDD para a geração de uma API com todas as funcionalidades CRUD fundamentais?
    - identificadores
    - navegabilidade entre *resources*

- QP3. Existe a necessidade de criação de um modelo intermediário, a partir de um modelo de configuração, que sirva como artefato de entrada para um gerador de APIs REST?

Existe se o modelo de configuração for muito abstrato e o gerador de APIs não compreender o mesmo nível de abstração. No caso da solução apresentada neste trabalho tomou-se a decisão de usar um modelo intermediário que possibilitasse uma validação sintática automática, e uma leitura humana e computacional rápida

- QP4. As APIs REST são suficientemente modularizáveis a ponto de permitir a geração automática de código a partir de blocos de código parametrizáveis? Qual a complexidade de se criar um compilador de APIs REST que não se utiliza de uma abordagem GP?

Considerando que a API gerada pela solução apresentada neste trabalho é em Node.js, nós conseguimos uma boa utilização dos blocos de código parametrizáveis (*snippets*). O fato da arquitetura REST impor a divisão em camadas e reforçar o uso de um formato de *resource* no sistema contribuiu para esta modularização. Sem a abordagem GP acreditamos que a implementação poderia ficar maior e mais difícil de manter, pois a necessidade de atualizar um *snippet* geraria um trabalho mais manual.

- QP5. É possível a geração automática de um banco de dados auxiliar à API REST?

O trabalho nos leva a concluir que sim, é possível gerar um banco de dados auxiliar a uma API REST. Algumas incompatibilidades entre a especificação MDD e o banco de dados podem atrapalhar esta geração. Neste trabalho estas incompatibilidades foram endereçadas gerando uma API mais robusta, e utilizando algumas funções do bando de dados de uma forma semantica especifica, por exemplo a relação de herança entre *resources* foi tratada adicionando um atributo oculto de tipo (`__type`) e armazenando todas as classe filhas na mesma coleção da classe pai.

- QP6. Diagramas de Classes UML são uma notação adequada para a modelagem de APIs REST?

Muitos fatores podem influenciar a resposta para esta pergunta, um dos mais importante deles seria a respeito do tipo de banco de dados utilizado. Se o banco de dados for relacional então algumas definições de dados serão levemente alteradas, porem as restrições de relacionamento entre os *resources* serão melhor reforçadas. No caso do banco de dados não for relacional as restrições de relacionamento podem ficar um pouco mais livres, podendo levar a referencias inválidas, porem as definições dos dados possam ser mais fieis ao diagrama de classes.

- QP7. Arquivos JSON Schema são adequados para a representação de APIs REST?
- QP8. O SGBD MongoDB, a linguagem Node.js e a biblioteca Mongoose são adequadas como tecnologias para a geração automática de APIs REST?

## 6.3 Trabalhos Futuros

Diversos trabalhos podem ser realizados expandindo o tema deste estudo. Nesta seção, definimos alguns temas que surgiram ao longo da realização do estudo, mas que por estar além do escopo proposto ou requerir um dispêndio de tempo não disponível, não foram explorados.

- A ferramenta Astah é extensível através do uso de *plugins*. A implementação de um *plugin* Astah para a geração de modelos de configuração segundo as especificações deste trabalho seria de grande ajuda na modelagem de APIs REST e execução de ferramentas AutoREST. Dentre os requisitos que este *plugin* deverá cumprir estão a criação automática de *tags* conforme as especificações da Seção ??, a limitação dos tipos de atributos disponíveis, a limitação dos tipos de associação disponíveis e uma interface de ajuda que apresente as regras listadas na Seção ?? para fácil acesso a referência.
- Algumas das ferramentas listadas na Seção ?? permitem: a geração automática de código para funcionalidades diferentes das propostas pela solução AutoREST, e; o gerenciamento da documentação e versionamento de APIs REST. Visto que algumas destas ferramentas utilizam especificações em notações derivadas de JSON, seria útil a adequação das especificações destas diversas ferramentas e a integração de seus ambientes para execução em conjunto.
- Devido a limitações da ferramenta Astah, determinados tipos de associação disponíveis na UML não puderam ser definidos para modelagem neste estudo. A definição de regras para a modelagem e conversão destes tipos de associação seria útil para aumentar a versatilidade da solução proposta. Dentre estas

associações, se destacam: as associações de Generalização/Especialização para representar o conceito de herança múltipla; o uso dos esteriótipos *Disjoint* e *Overlapping* para associações de Generalização/Especialização, e; a definição de navegabilidade em associações de agregação.

- A modelagem e implementação da ferramenta usada como prova de conceito da arquitetura proposta está limitada as tecnologias determinadas em sua especificação. A criação de outras possíveis implementações da arquitetura AutoREST serviria para validar a sua proposta de independência de tecnologia. Tecnologias que poderiam ser utilizadas para diferentes implementações são: para modelos de configuração, Diagramas de Classes UML gerados por ferramentas diferentes do Astah, ou Modelos Entidade-Relacionais; para PFIS, estruturas em linguagem RAML, Swagger ou WSDL, e; para geração de código, linguagens como ASP.NET ou php.

## Anexos

## ANEXO A – Lista de *Status Codes*

<i>Code</i>	<i>Reason-Phrase</i>
100	Continue
101	Switching Protocols
200	OK
201	Created
202	Accepted
203	Non-Authoritative Information
204	No Content
205	Reset Content
206	Partial Content
300	Multiple Choices
301	Moved Permanently
302	Found
303	See Other
304	Not Modified
305	Use Proxy
307	Temporary Redirect
400	Bad Request
401	Unauthorized
402	Payment Required
403	Forbidden
404	Not Found
405	Method Not Allowed
406	Not Acceptable
407	Proxy Authentication Required
408	Request Timeout
409	Conflict
410	Gone
411	Length Required
412	Precondition Failed
413	Payload Too Large
414	URI Too Long
415	Unsupported Media Type
416	Range Not Satisfiable
417	Expectation Failed
426	Upgrade Required
500	Internal Server Error
501	Not Implemented
502	Bad Gateway
503	Service Unavailable
504	Gateway Timeout
505	HTTP Version Not Supported



# ANEXO B – Gramática BNF de JSON Schema

Gramática BNF conforme ??).

```

JSDoc := { ( id , )? ( defs , )? JSch }
id := "id": "uri"
defs := "definitions": { kSch ( , kSch)* }
kSch := kword: { JSch }
JSch := ( res ( , res)* )
res := type | strRes | numRes | arrRes | objRes | multRes
      | refSch | title | description
type := "type": ([typename ( , typename)*] | typename)
typename := "string" | "integer" | "number" | "boolean" | "null"
          | "array" | "object"
title := "title": string
description := "description": string
strRes := minLen | maxLen | pattern
minLen := "minLength": n
maxLen := "maxLength": n
pattern := "pattern": "regExp"
numRes := min | max | multiple
min := "minimum": r ( ,exMin)?
exMin := "exclusiveMinimum": bool
max := "maximum": r ( ,exMax)?
exMax := "exclusiveMaximum": bool
multiple := "multipleOf": r (r >= 0)
arrRes := items | additems | minitems | maxitems | unique
items := ( sameitems | varitems )
sameitems := "items": { JSch }
varitems := "items": [{ JSch } ( , { JSch } ) * ]
additems := "additionalItems": (bool | { JSch })
minitems := "minItems": n
maxitems := "maxItems": n
unique := "uniqueItems": bool
objRes := prop | addprop | req | minprop | maxprop | dep | pattprop
prop := "properties": { kSch ( , kSch)* }

```

```

kSch := kword: { JSch }
addprop := "additionalProperties": (bool | { JSch })
req := "required": [ kword (, kword)*]
minprop := "minProperties": n
maxprop := "maxProperties": n
dep := "dependencies": { kDep (, kDep)*}
kDep := (kArr | kSch)
kArr := kword: [ kword (, kword)*]
pattprop := "patternProperties": { patSch (, patSch)*}
patSch := "regExp": { JSch }
multRes := allOf | anyOf | oneOf | not | enum
anyOf := "anyOf": [ { JSch } (, { JSch }) * ]
allOf := "allOf": [ { JSch } (, { JSch }) * ]
oneOf := "oneOf": [ { JSch } (, { JSch }) * ]
not := "not": { JSch }
enum := "enum": [ Jval (, Jval)*]
refSch := "\$ref": "uriRef"
uriRef := ( address )? ( \# / JPointer )?
JPointer := ( / path )
path := ( unescaped | escaped )
escaped := ~0 | ~1
address = (scheme : )? hier-part (? query )

```

## ANEXO C – JSON Schema relativo a Figura ??

**Listing 3** JSON Schema criado a partir da Figura ?? - Parte 1

```
1  {
2      "definitions": {
3          "Example_Class": {
4              "type": "object",
5              "additionalProperties": false,
6              "properties": {
7                  "attribute0": {
8                      "type": "integer"
9                  },
10                 "attribute1": {
11                     "type": "boolean"
12                 },
13                 "attribute2": {
14                     "type": "integer",
15                     "minimum": 0,
16                     "maximum": 255
17                 },
18                 "attribute3": {
19                     "type": "string",
20                     "maxLength": 1
21                 },
22                 "attribute4": {
23                     "type": "number"
24                 },
25                 "attribute5": {
26                     "type": "number"
27                 },
28                 "attribute6": {
29                     "type": "integer"
30                 },
31                 ...
32             }
33         }
34     }
35 }
```

**Listing 4** JSON Schema criado a partir da Figura ?? - Parte 2

```

1  {
2      {
3          {
4              {
5                  ...
6                  "attribute7": {
7                      "type": "integer"
8                  },
9                  "attribute8": {
10                     "type": "array"
11                     "items":{
12                         "type": "integer"
13                     }
14                 },
15                 "attribute9": {
16                     "type": "string",
17                     "format": "date-time"
18                 },
19                 "attribute10": {
20                     "type": "object"
21                 },
22                 "attribute11": {
23                     "type": "string",
24                     "minLength": 3,
25                     "maxLength": 140,
26                     "pattern": "/^[a-zA-Z]*$/ "
27                 }
28             },
29             "dependencies": {
30                 "attribute1": [ "attribute0" ],
31                 "attribute2": [ "attribute0" ],
32                 "attribute3": [ "attribute0" ],
33                 "attribute4": [ "attribute0" ],
34                 "attribute5": [ "attribute0" ],
35                 "attribute6": [ "attribute0" ],
36                 "attribute7": [ "attribute0" ],
37                 "attribute8": [ "attribute0" ],
38                 "attribute9": [ "attribute0" ],
39                 "attribute10": [ "attribute0" ],
40                 "attribute11": [ "attribute0" ]
41             }
42             "required": {
43                 [ "attribute0", "attribute1", "attribute3",
44                 ↪ "attribute4", "attribute5", "attribute6", "attribute7",
45                 ↪ "attribute8", "attribute9", "attribute10", "attribute11" ]
46             }
47         }
48     }
49 }

```

## ANEXO D – JSON Schema relativo a Figura ??

**Listing 5** JSON Schema criado a partir da Figura ?? - Parte 1

```
1  {
2      "definitions": {
3          "Class0": {
4              "type": "object",
5              "additionalProperties": false,
6              "properties": {
7                  "attribute0": {
8                      "type": "integer"
9                  },
10                 "attribute1": {
11                     "type": "integer"
12                 },
13                 "Class1_created_identifier": {
14                     "type": "integer"
15                 },
16                 "Class2_attribute3": {
17                     "type": "array",
18                     "items": {
19                         "type": "integer"
20                     }
21                 }
22             },
23             "dependencies": {
24                 "attribute1": [ "attribute0" ],
25                 "Class1_created_identifier": [ "attribute0" ],
26                 "Class2_attribute3": [ "attribute0" ]
27             },
28             "required": {
29                 [ "attribute0" ]
30             }
31         },
32         ...
33     }
34 }
```

**Listing 6** JSON Schema criado a partir da Figura ?? - Parte 2

```
1  {
2    {
3      ...,
4      "Class1": {
5        "type": "object",
6        "additionalProperties": false,
7        "properties": {
8          "created_identifier": {
9            "type": "integer"
10         },
11         "attribute2": {
12           "type": "integer"
13         },
14         "Class0_attribute0": {
15           "type": "integer"
16         },
17         "Class3_attribute4": {
18           "type": "integer"
19         }
20       },
21       "dependencies": {
22         "attribute2": [ "created_identifier" ],
23         "Class0_attribute0": [ "created_identifier" ],
24         "Class3_attribute4": [ "created_identifier" ]
25       },
26       "required": {
27         [ "created_identifier", "attribute0" ]
28       }
29     },
30     ...
31   }
32 }
```

**Listing 7** JSON Schema criado a partir da Figura ?? - Parte 3

```
1  {
2      {
3          ...,
4          "Class2": {
5              "type": "object",
6              "additionalProperties": false,
7              "properties": {
8                  "attribute3": {
9                      "type": "integer"
10                 },
11                 "Class0_attribute0": {
12                     "type": "array",
13                     "items": {
14                         "type": "integer"
15                     }
16                 },
17                 "Class3_attribute4": {
18                     "type": "array",
19                     "items": {
20                         "type": "integer"
21                     }
22                 "minLength": 1
23             }
24         },
25         "dependencies": {
26             "Class0_attribute0": [ "attribute3" ],
27             "Class3_attribute4": [ "attribute3" ]
28         },
29         "required": {
30             [ "attribute3", "attribute4" ]
31         }
32     },
33     ...
34 }
35 }
```

**Listing 8** JSON Schema criado a partir da Figura ?? - Parte 4

```
1  {
2      {
3          ...,
4          "Class3": {
5              "type": "object",
6              "additionalProperties": false,
7              "properties": {
8                  "attribute4": {
9                      "type": "integer"
10                 }
11             },
12             "dependencies": {
13
14             },
15             "required": {
16                 [ "attribute4" ]
17             }
18         }
19     }
20 }
```



## ANEXO E – JSON Schema relativo a Figura ??

---

**Listing 9** JSON Schema criado a partir da Figura ?? - Parte 1

---

```
1  {
2      "definitions": {
3          "Class4": {
4              "type": "object",
5              "additionalProperties": false,
6              "properties": {
7                  "attribute5": {
8                      "type": "integer"
9                  }
10             },
11             "dependencies": {
12
13             },
14             "required": {
15                 [ "attribute5" ]
16             }
17         },
18         ...,
19     }
20 }
```

---

**Listing 10** JSON Schema criado a partir da Figura ?? - Parte 2

```
1  {
2      {
3          ...,
4          "Class5": {
5              "type": "object",
6              "additionalProperties": false,
7              "properties": {
8                  "Class4_Inherited": {
9                      "\$ref": "#/definitions/Class4"
10                 },
11                 "attribute7": {
12                     "type": "integer"
13                 }
14             },
15             "dependencies": {
16
17             },
18             "required": {
19
20             }
21         },
22         "Class8": {
23             "type": "object",
24             "additionalProperties": false,
25             "properties": {
26                 "Class4_Inherited": {
27                     "\$ref": "#/definitions/Class4"
28                 },
29                 "attribute6": {
30                     "type": "integer"
31                 }
32             },
33             "dependencies": {
34
35             },
36             "required": {
37
38             }
39         },
40         ...,
41     }
42 }
```

**Listing 11** JSON Schema criado a partir da Figura ?? - Parte 3

```
1  {
2      {
3          ...,
4          "Class6": {
5              "type": "object",
6              "additionalProperties": false,
7              "properties": {
8                  "Class5_Inherited": {
9                      "\$ref": "#/definitions/Class4"
10                 },
11                 "attribute9": {
12                     "type": "integer"
13                 }
14             },
15             "dependencies": {
16
17             },
18             "required": {
19
20             }
21         },
22
23         "Class7": {
24             "type": "object",
25             "additionalProperties": false,
26             "properties": {
27                 "Class5_Inherited": {
28                     "\$ref": "#/definitions/Class4"
29                 },
30                 "attribute8": {
31                     "type": "integer"
32                 }
33             },
34             "dependencies": {
35
36             },
37             "required": {
38
39             }
40         }
41     }
42 }
```

## ANEXO F – JSON Schema relativo a Figura ??

**Listing 12** JSON Schema criado a partir da Figura ?? - Parte 1

```
1  {
2      "definitions": {
3          "Class10": {
4              "type": "object",
5              "additionalProperties": false,
6              "properties": {
7                  "attribute2": {
8                      "type": "integer"
9                  },
10                 "Class11_attribute1": {
11                     "type": "array",
12                     "items": {
13                         "type": "integer"
14                     }
15                 },
16                 "Class12_attribute3": {
17                     "type": "array",
18                     "items": {
19                         "type": "integer"
20                     }
21                 }
22             },
23             "dependencies": {
24                 "Class11_attribute1": [ "attribute2" ],
25                 "Class12_attribute3": [ "attribute2" ]
26             },
27             "required": {
28                 [ "attribute2" ]
29             }
30         },
31         ...
32     }
33 }
```

**Listing 13** JSON Schema criado a partir da Figura ?? - Parte 2

```
1  {
2    {
3      ...,
4      "Class11": {
5        "type": "object",
6        "additionalProperties": false,
7        "properties": {
8          "attribute1": {
9            "type": "integer"
10         },
11         "Class9": {
12           "type": "object",
13           "additionalProperties": false,
14           "properties": {
15             "attribute0": {
16               "type": "integer"
17             }
18           },
19           "dependencies": {
20
21           },
22           "required": {
23             [ "attribute0" ]
24           }
25         }
26       },
27       "dependencies": {
28         "Class9": [ "attribute1" ]
29       },
30       "required": {
31         [ "attribute1" ]
32       }
33     },
34     ...
35   }
36 }
```

**Listing 14** JSON Schema criado a partir da Figura ?? - Parte 3

```
1  {
2    {
3      ...,
4      "Class12": {
5        "type": "object",
6        "additionalProperties": false,
7        "properties": {
8          "attribute3": {
9            "type": "integer"
10         },
11         "Class10_Refs": {
12           "type": "array",
13           "items": {
14             "$ref": "#/definitions/Class10"
15           },
16           "minLength": 1
17         }
18       },
19       "dependencies": {
20         "Class10_Refs": [ "attribute3" ]
21       },
22       "required": {
23         [ "attribute3" ]
24       }
25     }
26   }
27 }
```

## ANEXO G – JSON Schema relativo a Figura ??

**Listing 15** JSON Schema criado a partir da Figura ?? - Parte 1

```

1  {
2      "definitions": {
3          "AssociationClass0": {
4              "type": "object",
5              "additionalProperties": false,
6              "properties": {
7                  "Class13_attribute0": {
8                      "type": "integer"
9                  },
10                 "Class14_attribute1": {
11                     "type": "integer"
12                 },
13                 "attribute2": {
14                     "type": "integer"
15                 },
16                 "attribute3": {
17                     "type": "integer"
18                 }
19             },
20             "dependencies": {
21                 "attribute2": [ "Class13_attribute0",
↪ "Class14_attribute1" ],
22                 "attribute3": [ "Class13_attribute0",
↪ "Class14_attribute1" ],
23                 "Class13_attribute0": [ "Class14_attribute1" ],
24                 "Class14_attribute1": [ "Class13_attribute0" ]
25             },
26             "required": {
27                 [ "Class13_attribute0", "Class14_attribute1" ]
28             }
29         },
30         ...
31     }
32 }

```

**Listing 16** JSON Schema criado a partir da Figura ?? - Parte 2

```
1  {
2    {
3      ...,
4      "Class13": {
5        "type": "object",
6        "additionalProperties": false,
7        "properties": {
8          "attribute0": {
9            "type": "integer"
10         },
11         "AssociationClass0_Link": {
12           "type": "array",
13           "items": {
14             "type": "object",
15             "properties": {
16               "\$ref":
17 ↪ "#/definitions/AssociationClass0"
18             }
19           }
20         },
21         "dependencies": {
22           "AssociationClass0_Link": [ "attribute0" ]
23         },
24         "required": {
25           [ "attribute0" ]
26         }
27       },
28       ...
29     }
30 }
```



**Listing 17** JSON Schema criado a partir da Figura ?? - Parte 3

```
1  {
2    {
3      ...,
4      "Class14": {
5        "type": "object",
6        "additionalProperties": false,
7        "properties": {
8          "attribute1": {
9            "type": "integer"
10         },
11         "AssociationClass0_Link": {
12           "type": "array",
13           "items": {
14             "type": "object",
15             "properties": {
16               ↪ "\$ref":
17                 "#/definitions/AssociationClass0"
18             }
19         },
20       },
21       "dependencies": {
22         "AssociationClass0_Link": [ "attribute1" ]
23       },
24       "required": {
25         [ "attribute1" ]
26       }
27     }
28   }
29 }
```

## ANEXO H – Model Mongoose para a classe da Figura ??

```
1  var mongoose = require('mongoose'),
2      integerValidator = require('mongoose-integer'),
3      Schema = mongoose.Schema;
4
5  var Example_ClassSchema = new Schema({
6      _id: {
7          type: Number,
8          integer: true,
9          required: true,
10         unique: true
11     },
12     attribute1: {
13         type: Boolean, required: true
14     },
15     attribute2: {
16         type: Number,
17         integer: true,
18         minimum: 0,
19         maximum: 255
20     },
21     attribute3: {
22         type: String,   maxlength: 1,   required: true
23     },
24     attribute4: {
25         type: Number,   required: true
26     },
27     attribute5: {
28         type: Number,   required: true
29     },
30     attribute6: {
31         type: Number,
32         integer: true,
33         required: true
34     },
35     ...
36 }
```

**Listing 18** Model Mongoose - Parte 2

```
1  {
2    ...
3    attribute7: {
4      type: Number,
5      integer: true,
6      required: true
7    },
8    attribute8: {
9      any: [{ type: Number, integer: true }]
10   },
11   attribute9: {
12     type: Date,
13     required: true
14   },
15   attribute10: {
16     type: Object, required: true
17   },
18   attribute11: {
19     type: String,
20     minlength: 3,
21     maxlength: 140,
22     match: /^[a-zA-Z]*$/,
23     required: true
24   }
25 };

26
27 Example_ClassSchema.virtual('attribute0').get(function() {
28   return this._id;
29 });
30 Example_ClassSchema.virtual('attribute0').set(function (value) {
31   this._id = value;
32 });
33
34 Example_ClassSchema.plugin(integerValidator);
35
36 Example_ClassSchema.methods.cleanObject = function() {
37   var doc = this.toObject({ virtuals: true });
38   delete doc.__v;
39   delete doc._id;
40   delete doc.id;
41   return doc;
42 };
43
44 module.exports = mongoose.model('Example_Class',
  ⇐ Example_ClassSchema);
```