UNIVERSIDADE FEDERAL FLUMINENSE
Instituto de Ciência e Tecnologia
Bacharelado em Ciência da Computação

Paulo Cézar Diniz Losano Figueiredo da Rocha

REST in Safety (RiS): Um Framework para desenvolvimento de APIs RESTful com foco em segurança pública

Rio das Ostras – RJ
2021

Ficha catalográfica automática - SDC/BRO
Gerada com informações fornecidas pelo autor

R672r Rocha, Paulo Cézar Diniz Losano Figueiredo da
REST in Safety (RiS) : Um Framework para desenvolvimento de APIs RESTful com foco em segurança pública / Paulo Cézar Diniz Losano Figueiredo da Rocha ; Sérgio Crespo Coelho da Silva Pinto, orientador. Niterói, 2021.
89 f. : il.

Trabalho de Conclusão de Curso (Graduação em Ciência da Computação)-Universidade Federal Fluminense, Instituto de Ciência e Tecnologia, Rio das Ostras, 2021.

1. Framework orientado a objeto. 2. Engenharia de software. 3. Seguranca publica. 4. Aplicacão web. 5. Produção intelectual. I. Pinto, Sérgio Crespo Coelho da Silva, orientador. II. Universidade Federal Fluminense. Instituto de Ciência e Tecnologia. III. Título.

CDD -

Bibliotecário responsável: Debora do Nascimento - CRB7/6368

Paulo Cézar Diniz Losano Figueiredo da Rocha

REST in Safety (RiS): Um Framework para desenvolvimento de APIs RESTful com foco em segurança pública

Trabalho de Conclusão de Curso submetido ao Curso de Ciência da Computação da Universidade Federal Fluminense como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

Orientador:
Prof. DSc. SÉRGIO CRESPO COELHO DA SILVA PINTO

Rio das Ostras – RJ
2021

Paulo Cézar Diniz Losano Figueiredo da Rocha

REST in Safety (RiS): Um Framework para desenvolvimento de APIs RESTful com foco em segurança pública

Trabalho de Conclusão de Curso submetido ao Curso de Ciência da Computação da Universidade Federal Fluminense como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

Aprovado em 24 de setembro de 2021.

BANCA EXAMINADORA

Prof. DSc. SÉRGIO CRESPO COELHO DA SILVA PINTO – Orientador
UFF -Universidade Federal Fluminense

Profª. DSc. ADRIANA PEREIRA DE MEDEIROS - Avaliadora
UFF -Universidade Federal Fluminense

Prof. DSc. LEANDRO SOARES DE SOUSA - Avaliador
UFF -Universidade Federal Fluminense

Rio das Ostras – RJ
2021

Dedico este trabalho a minha Mãe, pelos seus inúmeros sacrifícios para me dar essa oportunidade, e ao meu Pai que se foi, mas foi o principal responsável pela minha paixão pelos computadores desde cedo.

AGRADECIMENTOS

Agradeço principalmente a minha Mãe e a minha família, por me fornecerem todo o suporte para ter a oportunidade e o privilégio de estudar em uma Universidade Pública distante de casa.

Agradeço aos professores do ICT e do RCM que generosamente compartilharam comigo e com tantos outros colegas seus conhecimentos, experiências e sabedorias.

Um agradecimento em especial ao meu orientador Professor Sérgio Crespo, que além do suporte neste trabalho, junto com a Professora Adriana Medeiros foram fundamentais para desenvolver minha capacidade de abstrair e compreender abstrações, através de suas aulas, trabalhos e exercícios, serei eternamente grato por isso.

Aos grandes amigos Pedro Henrique Antonellini e Cauã Rezende, com os quais tive o privilégio de compartilhar não apenas uma casa, mas conhecimento, pensamentos, sonhos, alegrias e tristezas.

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Este trabalho apresenta o projeto e implementação de um framework orientado a objetos para suportar a criação de APIs *RESTful* com foco em segurança pública. Observa-se uma forte demanda por novos softwares e integração entre sistemas para um efetivo apoio e planejamento das atividades de segurança pública. Os custos do desenvolvimento de novos softwares são muito altos, nas últimas décadas a engenharia de software tem trabalhado para propor técnicas e métodos para diminuir esses custos através do reuso, duas técnicas populares de reuso são o desenvolvimento de frameworks e a computação orientada a serviços. Diante deste cenário foi proposto a criação de um framework para criação de serviços em uma arquitetura REST, e que também permita capturar conhecimento do domínio e reutilizá-lo em outras aplicações. Para isso foi feito um breve estudo sobre essas técnicas de reuso, sobre a arquitetura REST e uma análise do domínio da segurança pública, e então com o auxílio de uma metodologia baseada em UML, foi projetado o framework proposto, por fim as decisões de implementação e as instancias de teste são comentadas.

Palavras-chave: Framework, REST, Web Services, API, Segurança Pública.

ABSTRACT

This paper presents the design and implementation of an object-oriented framework to support the development of RESTful API's focused on public safety. There is an crescent demand for new software and integration between systems in order to get an effective support and planning of public safety activies. The costs of developing new software are very high, in the last decades software engineering has been working in techniques and methods to reduce these costs through reuse, two popular reuse techniques are framework development and service-oriented computing. Given this scenario, it was proposed to create a framework for creating services based on an REST architecture, which also allows capturing domain knowledge and reusing it in other applications. For this, a brief study on these reuse techniques, on the REST architecture and an analysis of the domain of public safety was done, and then with the aid of a methodology based on UML, the proposed framework was designed, finally the implementation decisions and test instances are commented.

Keywords: Framework, REST, Web Services, API, Public Security.

SUMÁRIO

1 INTRODUÇÃO

O Brasil é um país com muitos problemas sociais, e certamente a segurança pública é um dos problemas conhecido por todos os brasileiros, desde os efeitos devastadores da violência até a insegurança do cidadão com medo de ter seus bens furtados.

Assim como qualquer outro problema social, é bastante complexo e não tem uma solução simples, pois envolve muitos fatores, mas principalmente o contexto social e econômico do momento. Embora o Brasil tenha aumentado os investimentos em segurança pública nas últimas décadas, percebe-se que estes recursos não são bem utilizados, foram feitos investimentos em tecnologia, mas sem um devido planejamento de como usá-la de forma eficaz [1].

Em 2003, a Secretária Nacional de Segurança Pública (Senasp) reconheceu a dificuldade de estruturar políticas públicas de combate à violência e a criminalidade, principalmente devido à falta de informações qualitativas e de estudos sobre a situação da segurança pública no país. Essa análise gerou uma série de iniciativas para investimento em pesquisa aplicada e na reestruturação tecnológica do setor. Mas embora existam alguns canais de apoio e investimento ao desenvolvimento tecnológico da área de segurança pública, falta um projeto nacional para estruturar e definir prioridades na pesquisa e desenvolvimento de tecnologias [2].

Um exemplo é o Sistema Nacional de Informações de Segurança Pública, Prisional e sobre Drogas (Sinesp), em que a falta de clareza nos objetivos, e em quais dados coletar tornou o projeto do sistema extremamente abrangente, de forma que além das informações necessárias para a gestão das políticas públicas, o sistema também suporte informações para inteligência, planejamento e até atuação operacional de órgãos de segurança pública. [3]

1.1 Problema

Existem obstáculos na cooperação e na coordenação de estados no fornecimento de informações para a União, muitas vezes por questões específicas de cada estado e seus sistemas, que nem sempre estão preparados ou então faltam profissionais para se trabalhar no compartilhamento desses dados e estatísticas [3].

Uma pesquisa realizada em 2014 levantou que 74% dos 26 estados brasileiros e Distrito Federal disponibilizavam dados sobre criminalidade na internet, e desses que disponibilizavam, 70% forneciam essas informações no formato texto (PDF ou HTML) [4]. Embora a lei nº 12.527 de 2011, conhecida como Lei de Acesso à Informação, estabeleça que órgãos e entidades públicas disponibilizem informações de interesse coletivo em sítios oficiais na internet, e que

estes sítios, entre outros requisitos deveriam disponibilizar dados estruturados em formatos abertos e legíveis por máquina para possibilitar o acesso por sistemas externos [5].

Entre 2002 e 2010 a Financiadora de Estudos e Projetos (Finep) financiou 53 projetos da área de segurança pública, 34 desses projetos são da área de tecnologias da informação e comunicação, sendo principalmente o desenvolvimento de softwares para treinamento e apoio as atividades de segurança pública. E esses 34 projetos respondem por R$ 73 milhões dos aproximadamente R$ 77 milhões investidos nos 53 projetos [2].

Todos esses exemplos reforçam como o desenvolvimento de softwares e sistemas para apoio da segurança pública e compartilhamento de informações é extremamente necessário, mas também caro. O próprio desenvolvimento do Sinesp, já gastou mais de R$ 200 milhões desde 2012 [1].

1.2 Proposta

Uma das estratégias da engenharia de software para se diminuir o custo de desenvolvimento e manutenção de sistemas é o reuso de software. Essa é uma abordagem que visa reutilizar softwares existentes de diversos tipos, desde sistemas e aplicações inteiras até componentes e funcionalidades isoladas. Além da redução dos custos, o reuso de software traz outras vantagens como redução do tempo de desenvolvimento, facilidade na adoção de padrões, aumento na qualidade do software e permite encapsular o conhecimento de especialistas do domínio em componentes reusáveis [6].

Existem duas abordagens de reuso de software que são especialmente úteis para se compreender a proposta desse trabalho, a primeira é a utilização de frameworks que são abstrações genéricas utilizadas para compor aplicações, uma definição mais precisa seria que ”*um framework é um conjunto integrado de artefatos de software (como classes, objetos e componentes) que colaboram para fornecer uma arquitetura reutilizável para uma família de aplicações relacionadas*" (Schmidt; Gokhale; Natarajan, 2004) [7].

E a outra abordagem é a de sistemas orientados a serviços, onde sistemas são criados utilizando funcionalidades distribuídas chamadas **serviços**, que são a representação de algum recurso computacional ou de informação que pode ser utilizado (consumido) por outra aplicação, independente das linguagens de programação que estejam sendo utilizadas. [6].

Dessa forma o objetivo desse trabalho é o projeto e implementação de um framework que fornecerá uma arquitetura reutilizável para encapsular funcionalidades úteis ao domínio da segurança pública e disponibilizá-las na forma de serviços na internet. Estes serviços criados

pelo framework proposto, poderão ainda ser reutilizados para compor vários sistemas e aplicações para a segurança pública.

2 REFERENCIAL TEÓRICO

Nesta seção serão apresentados com mais detalhes os conceitos relacionados a reuso de software, frameworks de aplicação, computação orientada a serviços, arquiteturas do tipo *RESTful*, e uma visão geral sobre o domínio da segurança pública.

2.1 Reuso de software

A engenharia de software baseada em reuso, é uma abordagem onde o processo de desenvolvimento é direcionado para o reuso de artefatos de software já existentes, com o objetivo de atingir custos mais baixos, entregas mais rápidas, e maior qualidade. Embora seja um conceito antigo, essa estratégia só se popularizou a partir dos anos 2000 quando as empresas começaram a enxergar o software como um ativo valioso e buscaram no reuso uma forma de aumentar o retorno sobre seus investimentos. Outros dois fatores que também contribuíram para a extensa adoção dessa estratégia foram o movimento de código aberto, que acabou por disponibilizar uma imensa base de código pronta para ser reutilizada, e os padrões de *Web Services*, que facilitaram o desenvolvimento e o reuso de serviços de software [6].

A ideia de reuso consiste em encapsular conhecimento e experiencia, organizando-os e criando mecanismos e estruturas que permitam reaproveitá-los em novas aplicações [8]. Segundo (Ezran et al., 2002) [9], as três características fundamentais do reuso de software são [10]:

Reuso é uma prática sistemática de desenvolvimento de software, o que significa que o reuso precisa ser totalmente integrado no processo de desenvolvimento. Para se atingir máximo valor do reuso é preciso definir estratégias e garantir suporte técnico, orçamentário e gerencial para que a equipe tenha competência e motivação, além disso é importante monitorar e controlar a performance de reuso [10].

O reuso explora similaridades nos requisitos e/ou na arquitetura entre aplicações, as oportunidades de se reutilizar software entre aplicações geralmente acontece quando essas aplicações têm semelhanças nos requisitos, na arquitetura ou em ambos. Um processo de desenvolvimento focado no reuso, deve aproveitar ao máximo essas oportunidades durante as fases em que os requisitos são identificados e as decisões sobre a arquitetura são tomadas [10].

Reuso oferece benefícios na produtividade, qualidade e na performance do negócio, de um modo geral não apenas na engenharia de software, reutilizar qualquer forma de experiência comprovada desde produtos, processos até modelos, ajuda a aumentar a qualidade e a produtividade. As melhorias na performance do negócio incluem a redução dos custos, menor tempo para o lançamento de produtos no mercado e aumento na satisfação do cliente como consequência dos aprimoramentos na qualidade e produtividade [10].

É importante ressaltar que o reuso de software vai além de apenas reutilizar códigos e implementações, praticamente qualquer artefato de software pode ser reutilizado incluindo requisitos, estruturas de projeto, modelos, especificações e documentações [11].

Os artefatos reutilizáveis que encapsulam conhecimento do domínio e tem um alto valor para a organização, são chamados de "ativos de software", e são classificados em dois tipos: ativos verticais, que são específicos do domínio de negócios, ou ativos horizontais, inerentes à arquitetura da aplicação, podendo ser reutilizados independente do domínio [10].

A granularidade dos ativos de software varia, desde sistemas inteiros sendo reutilizados dentro de um sistema maior, ou então uma aplicação que é configurada e customizada para diferentes clientes, até unidades menores como componentes que encapsulam uma série de funcionalidades para resolver um problema específico, ou até mesmo objetos e funções que implementam uma única funcionalidade [6].

2.1.1 Técnicas de reuso de software

Muitas técnicas e abordagens foram desenvolvidas para apoiar o reuso de software, essas técnicas apresentam ações, estratégias e padrões, que podem ser adotados durante o processo de desenvolvimento a fim de se implementar o reuso efetivamente [12]. Duas dessas técnicas já foram citadas anteriormente, que são frameworks e sistemas orientados a serviços, essas abordagens serão melhore apresentadas nas próximas seções. Algumas outras técnicas de reuso de software interessantes são:

Bibliotecas de classes, que são grandes repositórios de código contendo classes e funções que são reutilizadas frequentemente durante o desenvolvimento [6].

Engenharia de software baseada em componentes, sistemas são desenvolvidos compondo e integrando componentes de software, esses componentes são coleções de objetos reutilizáveis com interfaces, interações e dependências bem definidas [6].

Linhas de produtos de software, é uma técnica em que as aplicações são generalizadas em torno de uma arquitetura comum, e assim permitir que essa aplicação genérica possa ser adaptada para os requisitos de diversos clientes [6].

Padrões de projeto, que são abstrações de soluções para problemas comuns que ocorrem frequentemente no desenvolvimento de software, essas soluções já foram exaustivamente testadas e reutilizadas em vários sistemas e se tornaram padrões de melhores práticas [6].

2.1.2 Benefícios e Dificuldades do reuso de software

Existem várias vantagens do reuso de software, a mais óbvia é a redução dos custos do desenvolvimento, já que os artefatos reutilizados permitem que menos componentes precisem ser especificados, projetados, implementados e validados, entretanto um obstáculo não tão óbvio é que pode haver um custo significativo durante o processo de desenvolvimento para se compreender quais dos componentes disponíveis são mais adequados para cada caso [6].

Na Tabela 1 são mostrados os principais benefícios e as principais dificuldades encontradas ao se reutilizar software.

Tabela 1- Benefícios e dificuldades da engenharia de software baseada em reuso [6].

Benefícios	Dificuldades
Desenvolvimento Acelerado: Com o reuso de software é possível reduzir o tempo de desenvolvimento e validação.	**Criar, manter e usar uma biblioteca de componentes:** Desenvolver uma biblioteca de componentes e garantir que eles podem reusáveis, é um investimento caro.
Uso eficaz de especialistas: É possível encapsular o conhecimento de especialistas em componentes reusáveis, em vez de repetirem a mesma tarefa várias vezes.	**Encontrar, entender e adaptar componentes reusáveis:** Os componentes de software disponíveis em um repositório precisam ser descobertos, compreendidos e depois adaptados para cada situação.
Maior confiança e qualidade: O ativo de software que já foi reutilizado e testado em inúmeras situações é mais confiável.	**Falta de suporte da ferramenta:** Algumas ferramentas de software não possuem suporte ao desenvolvimento com reuso.

Custos de desenvolvimento mais baixos: Como menos linhas de código precisam ser implementadas espera-se que os custos diminuam também.	**Síndrome do "não foi inventado aqui":** Alguns engenheiros preferem reescrever novos componentes a reutilizar existentes, pois acreditam que é mais desafiador e que podem desenvolver componentes melhores.
Menor risco para o processo: O custo de reutilizar software já é conhecido enquanto o um novo desenvolvimento é incerto, essa maior certeza sobre os custos é importante do ponto de vista gerencial, pois diminui a margem de erro nas estimativas.	**Maiores custos de manutenção:** Caso o código fonte de um componente reutilizado não estiver disponível, o custo de manutenção aumenta muito quando esse componente precisa ser atualizado ou alterado.
Conformidade com os padrões: O reuso de software facilita alguns padrões como por exemplo os elementos de uma interface gráfica que podem ser reutilizados para padronizar a aparência de uma família de aplicações.	

2.2 Frameworks

Um dos principais benefícios apresentado pelo desenvolvimento orientado a objetos é que os objetos poderiam ser facilmente reutilizados em diferentes aplicações, porém com a maturidade dessa abordagem, percebeu-se que objetos eram especializados demais e por isso muitas vezes era mais trabalhoso adaptar uma classe existente do que implementá-la novamente, e utilizando-se abstrações mais genéricas era possível reutilizar de forma mais eficiente os objetos criados, essa técnica deu origem aos frameworks [6].

(Mattison, 1996) [16] define framework como "*uma arquitetura projetada para o máximo reuso, representada como uma coleção de classes abstratas e concretas que encapsulam comportamentos com alto potencial de especialização por subclasses"*.

Para (Johnson e Foote, 1988) [13], um framework é "*um design abstrato de uma família particular de aplicações, geralmente composto por classes*". Além de ser uma forma de reuso mais resistente do que as convencionais, pois permite de uma forma fácil reutilizar toda a estrutura que conecta os componentes, mantendo a consistência entre eles, mesmo quando os requisitos estão em constante mudança.

Segundo (Fayad e Schmidt, 1997) [14], "*framework é uma aplicação semiacabada, reutilizável que pode ser especializada para gerar aplicações customizadas"*. Afirmam também que a utilização de frameworks tem como benefícios principais, a modularidade, reusabilidade, extensibilidade e a inversão de controle que eles fornecem aos desenvolvedores.

Em (Gamma et al., 1994) [15] framework é definido como "*um conjunto de classes que colaboram para compor um design reutilizável para uma família específica de software"*. Ou seja é o framework que irá definir a arquitetura da aplicação, a estrutura geral, as responsabilidades e interações entre os objetos. Muitas decisões de projeto são capturadas e predefinidas pelo framework, permitindo que desenvolvedor se concentre apenas nos parâmetros específicos da aplicação. Portanto frameworks devem ter maior ênfase em reutilizar design do que código, embora geralmente um framework terá classes concretas que já fornecem funcionalidades imediatas.

Baseando-se nas definições apresentadas, podemos considerar framework como uma estrutura reutilizável, semiacabada e abstrata o suficiente para ser adaptada a vários projetos de software de um mesmo domínio de problema.

Algumas vezes é possível confundir frameworks com bibliotecas de classes, porque muitas bibliotecas apresentam comportamento parecido com um framework, e muitos frameworks podem ser usados apenas como uma biblioteca de classes [17], mas as principais diferenças entre frameworks e bibliotecas de classes, são o grau de reuso e o impacto na arquitetura da aplicação, frameworks fornecem um grau de reuso maior que bibliotecas de classes, pois além de cobrir o reuso das funcionalidades, eles também capturam as características da arquitetura, e como consequência a utilização de um framework tem um impacto muito maior na arquitetura da aplicação do que uma biblioteca de classes [16].

2.2.1 Características de frameworks

Frameworks precisam gerar várias aplicações para um determinado domínio de problema, isso exige uma certa flexibilidade na estrutura para que a aplicação possa customizar o design fornecido. Esses pontos de flexibilidade de um framework são chamados de **hot spots**, e normalmente são classes ou métodos abstratos, que precisam ser implementados com o código específico da aplicação que está instanciando o framework [18].

Assim como existem pontos flexíveis em um framework, existem outro pontos que não são mutáveis, e não podem ser alterados facilmente, esses pontos imutáveis constituem o kernel de

um framework, e são chamados de **frozen spots**. Normalmente são classes concretas já implementadas que chamam os hot spots desenvolvidos pela aplicação [18].

Todas as aplicações geradas pelo framework utilizam o mesmo kernel, ele que decide quando e qual hot spot da aplicação deve ser chamado durante a execução do programa [18], normalmente em outras técnicas de reuso, como bibliotecas de classes por exemplo, esse controle é feito pela aplicação, que decide quais classes da biblioteca irá utilizar durante seu fluxo de execução. Essa característica da arquitetura dos frameworks é chamada de **inversão de controle**, ou algumas vezes também chamado de princípio Hollywood, "Não nos chame, nós chamamos você" [16].

2.2.2 Vantagens e desvantagens do reuso por frameworks

Embora a utilização de frameworks traga claras vantagens na redução de código e em manter o conhecimento do domínio encapsulado na organização. É preciso ressaltar que frameworks focam seu reuso em um domínio de problema, e não em uma aplicação específica, por isso toda vez que existir um conflito entre o geral e o específico é possível identificar vantagens e desvantagens de framework [16].

Na Tabela 2 são apresentadas as vantagens e desvantagens mais comuns encontrados na utilização de frameworks orientados a objeto.

Tabela 2 - Vantagens e desvantagens de frameworks orientados a objeto [16].

Vantagens	Desvantagens
Real diminuição de linhas de código quando as funcionalidades exigidas na aplicação forem similares as funcionalidades que o framework captura.	Documentação e um framework é um ponto crucial para seu usuário, um framework que não possui um bom suporte de documentação para o usuário será pouco utilizado.
O código do framework já foi escrito e depurado.	É difícil desenvolver um bom framework, é necessário experiência no domínio.

O framework reusa design e não somente código, esse tipo de reuso faz com que conhecimento e experiência sejam transferidos para o usuário do framework	Retro compatibilidade pode ser difícil de ser mantida, conforme o framework evolui com o tempo e se torna mais maduro, as aplicações construídas com ele devem evoluir junto.
Alguns aspectos de eficiência do software podem ser atingidos com o framework, mas que normalmente não seriam feitos dentro do orçamento e prazo normal do projeto.	O processo de depuração pode ser complicado porque as vezes é difícil distinguir se um defeito é do framework ou da aplicação.
A manutenção do software é aprimorada, isso acontece pois quando um defeito é corrigido no framework, ele também é corrigido na aplicação.	A flexibilidade e generalização de um framework, pode atrapalhar a eficiência de alguma aplicação em particular.

2.2.3 Classificação de Frameworks

Frameworks podem ser classificados por diferentes dimensões, as mais importantes são em relação ao seu escopo, a estrutura interna do framework e como ele foi projetado para ser utilizado. [16]

Do ponto de vista do **escopo** os frameworks são classificados pelo tipo de problema que eles atendem, divididos em três categorias:

- **Frameworks de aplicação:** simplificam o desenvolvimento cobrindo funcionalidades uteis para vários domínios de aplicação, por exemplo a interface gráfica com o usuário (GUI) e ferramentas de processamento de linguagens [13].
- **Frameworks de domínio:** capturam o conhecimento e experiência de um domínio de aplicação em particular, exemplos são frameworks para sistemas de engenharia, finanças e telecomunicações. [13].
- **Frameworks de suporte:** oferecem serviços de baixo nível do sistema, como por exemplos drivers de dispositivos [16].

A **estrutura** interna de um framework é descrita pelos conceitos de arquitetura de software nos quais ele se baseia, por isso os frameworks podem ser classificados de acordo com sua arquitetura. (Buschman et al. 1996) [19] descrevem as principais arquiteturas de softwares:

- **Arquitetura em camadas:** a aplicação é estruturada de forma em que suas funcionalidades são separadas em grupos de subtarefas, cada grupo representa uma camada de abstração da aplicação.
- **Arquitetura de pipes and filters:** é uma estrutura onde o processamento é feito em etapas, cada etapa de processamento é encapsulada em um *filter*, e os dados são passados entre os filters através de pipes.
- **Arquitetura Blackboard:** utilizada em sistemas que trabalham com problemas muito complexos, nesta arquitetura vários subsistemas especializados em diferentes domínios cooperam para chegar a uma melhor solução para esses problemas.
- **Broker:** nessa arquitetura um componente chamado broker é responsável por coordenar a comunicação, encaminhando requisições, transmitindo resultados e exceções.
- **Model-View-Controller (MVC):** estrutura a aplicação em três tipos de componentes, os models que que contém os dados e as funcionalidades principais, as *views* que exibem a informação para o usuário e os *controllers* que cuidam das interações do usuário.
- **Presentation-Abstraction-Controller (PAC):** a aplicação é estruturada em pequenas subfunções independentes que cooperam para fornecer a funcionalidade completa.
- **Mircokernel:** o sistema é divido entre um núcleo funcional mínimo, e as suas outras funcionalidades estendidas ou especificas do cliente.
- **Arquitetura Reflexiva:** é uma arquitetura que suporta modificações dinamicamente em aspectos fundamentais como, tipos de estruturas e chamadas de função.

E por fim, frameworks podem ser classificados em relação a forma como ele foi projetado par ser **utilizado**, que basicamente pode ser de duas maneiras:

- **Architecture-driven:** nessa abordagem o usuário adapta o framework derivando as classes e sobrescrevendo seus métodos, utilizando principalmente o mecanismo de herança [16]. Também conhecidos como frameworks White-box, já que seus usuários precisam conhecer suas estruturas internas para adaptar suas funcionalidades [14].
- **Data-driven:** é uma abordagem em que as especializações são feitas através da composição de objetos existentes [16]. São considerados frameworks Black-box, pois seus

usuários precisam apenas conhecer as interfaces dos componentes disponíveis e não suas estruturas internas.

2.2.4 Desenvolvimento de um framework

O desenvolvimento de um framework é diferente do desenvolvimento de uma aplicação comum, um framework tenta cobrir os conceitos relevantes de um domínio, enquanto a aplicação precisa apenas cobrir seus requisitos [20]. Algumas metodologias foram propostas para apoiar o processo de desenvolvimento de frameworks, a seguir serão apresentadas algumas delas.

Processo de desenvolvimento baseado na experiência de aplicações passadas, onde inicialmente são desenvolvidas algumas aplicações (no mínimo duas), no domínio do problema, a partir delas uma série de funcionalidades comuns entre essas aplicações são identificadas e capturadas no desenvolvimento da primeira versão do framework. As aplicações iniciais utilizadas como base são desenvolvidas novamente, mas utilizando agora a primeira versão do framework. Baseando-se na experiência obtida durante o redesenvolvimento dessas aplicações, o framework é aprimorado gerando uma segunda versão. A partir desse ponto um processo iterativo é iniciado, onde novas aplicações são construídas com a versão aprimorada do framework, e as novas funcionalidades em comum identificadas junto com as experiências obtivas no desenvolvimento dessas novas aplicações são utilizadas no aprimoramento e manutenção das versões seguintes do framework. O processo completo é exibido na Figura 1 [16].

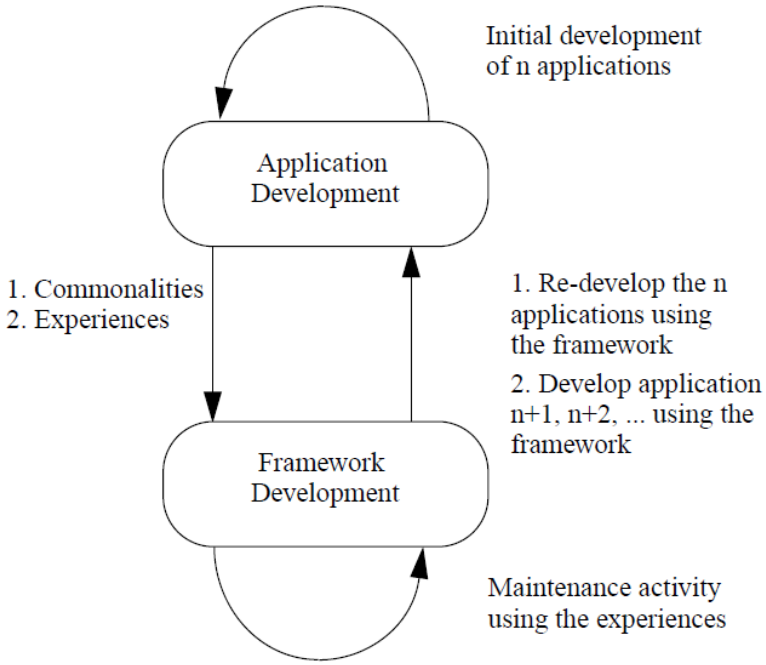

Figura 1- Processo de desenvolvimento baseado na experiência de aplicações [16]

Processo de desenvolvimento baseado na análise do domínio, a primeira etapa desse processo é analisar o domínio do problema, buscando identificar e entender as abstrações mais comuns. Essa é uma tarefa difícil, pois exige que a organização tenha experiência prévia no desenvolvimento de aplicações dessa área. Depois de identificadas as abstrações necessárias uma versão inicial do framework é desenvolvida junto com uma aplicação de teste. O próximo passo é desenvolver novas aplicações utilizando essa versão inicial, e então de acordo com as necessidades encontradas durante o desenvolvimento, o framework é modificado conforme necessário, e as aplicações desenvolvidas anteriormente são revalidadas para garantir que irão funcionar com as novas mudanças. A partir disso um processo de evolução do framework é estabelecido através de atividades de manutenção e aprimoramento periódicos com base nas experiências obtidas no desenvolvimento de novas aplicações, este processo completo é ilustrado na Figura 2 [16].

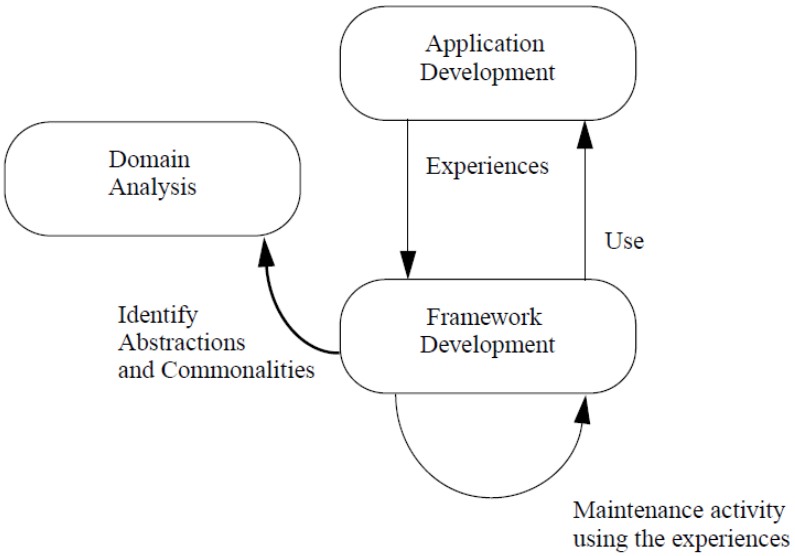

Figura 2- Processo de desenvolvimento baseado na análise do domínio [16]

Processo de desenvolvimento utilizando padrões de projeto, neste processo inicia-se criando uma aplicação no domínio do problema, e então padrões de projeto são aplicados sistematicamente nessa aplicação até conseguir se atingir a estrutura inicial do framework. E assim como nas outras abordagens, um processo iterativo é iniciado para se manter e aprimorar o framework com base nas experiências obtidas com o desenvolvimento de novas aplicações. A Figura 3 mostra esse processo [16].

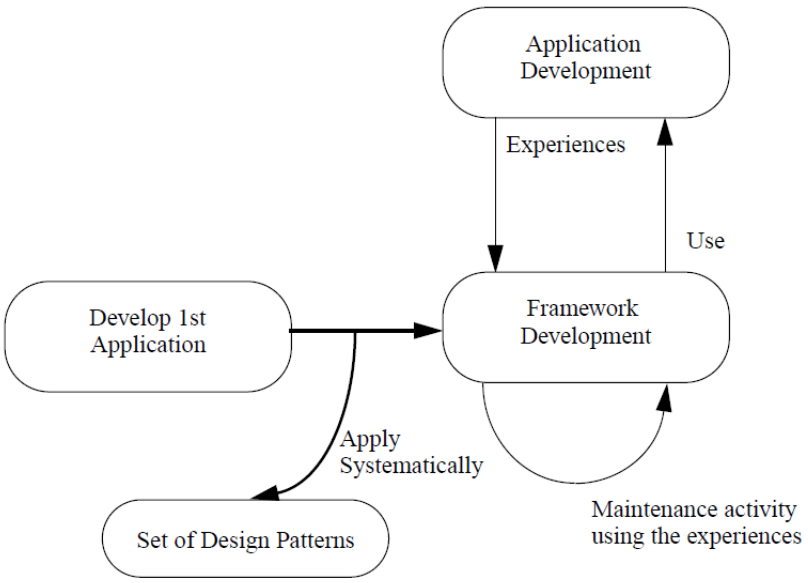

Figura 3 - Processo de desenvolvimento utilizando padrões de projeto [16]

Analisando as abordagens apresentadas, percebe-se pontos em comum entre elas, dessa forma é possível descrever um **processo geral para desenvolvimento de frameworks,** esse processo inclui as seguintes etapas [16]:

- **Análise do domínio**: feita sistematicamente ou através do desenvolvimento de uma ou mais aplicações, o objetivo é identificar as abstrações chaves no domínio de interesse.
- **Desenvolvimento inicial do framework:** a primeira versão do framework é desenvolvida com base nas abstrações identificadas.
- **Desenvolvimento de aplicações:** Uma ou possivelmente algumas aplicações são desenvolvidas utilizando a versão inicial do framework.
- **Aprimoramento:** Os problemas e experiências encontrados no desenvolvimento de aplicações são capturados e resolvidos em uma nova versão do framework.
- **Evolução:** Este ciclo de se criar aplicações e aprimorar o framework é repetido um número de vezes até o framework atingir níveis de maturidade mais altos.

A Figura 4 mostra esse processo generalizado para desenvolvimento de frameworks.

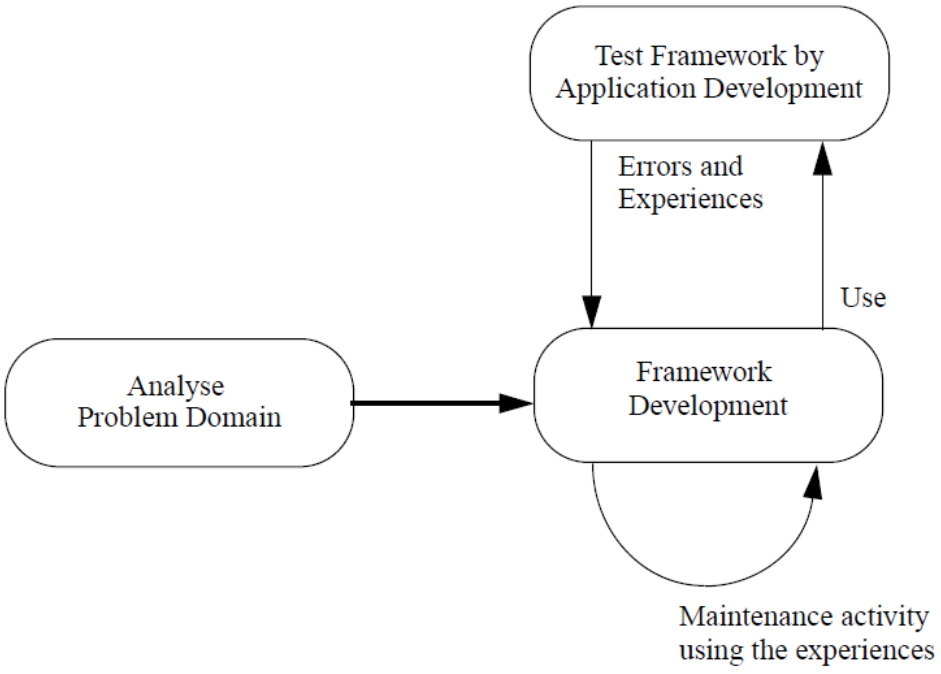

Figura 4 - Processo geral de desenvolvimento de frameworks [16].

2.3 Computação Orientada a Serviços

Um dos maiores desafios durante as etapas do processo de desenvolvimento de software é lidar com a heterogeneidade dos ambientes e as mudanças constantes nos requisitos. Muitas companhias hoje utilizam uma grande quantidade de sistemas, aplicações e arquiteturas de diferentes idades e tecnologias, integrar todos esses produtos de vários vendedores e entre diferentes plataformas quase sempre é uma tarefa complicada. Além disso, com a aceleração da globalização e a modernização dos negócios online, a competição entre as empresas aumentou muito, fazendo com que as necessidades e requisitos do cliente mudem cada vez mais rápido [21].

Conforme exibido na Figura 5, as arquiteturas de software foram evoluindo para tentar amenizar esses problemas de heterogeneidade, interoperabilidade e constante mudança dos requisitos. Atualmente acredita-se que respostas satisfatórias são obtidas com a utilização da computação orientada a serviços [21].

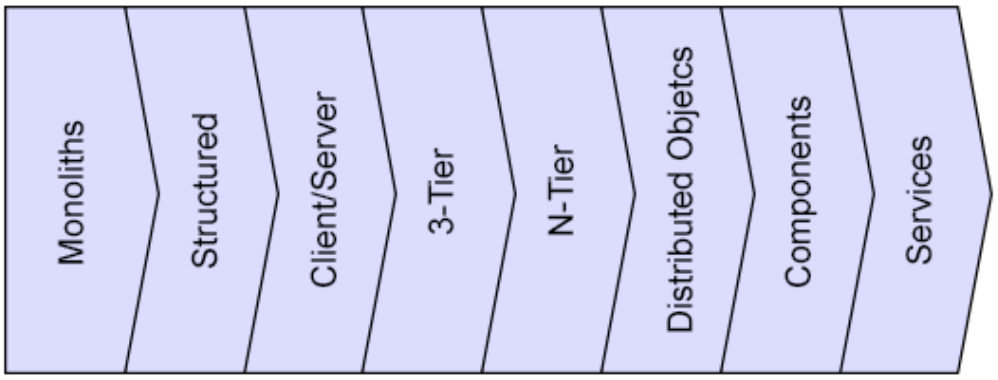

Figura 5 - A evolução das arquiteturas de software [21]

A computação orienta a serviços é uma forma de desenvolver sistemas distribuídos nos quais os componentes do sistema são serviços independentes, que não estão vinculados a nenhum tipo de plataforma ou de linguagem de programação específica [6]. Ela aprimora as plataformas passadas de computação distribuída à medida que adiciona novas camadas ao design e um amplo conjunto de tecnologias de implementação disponíveis [22].

2.3.1 Conceitos da Computação Orientada a Serviços

Para se compreender a computação orientada a serviços é necessário conhecer seus elementos primários, e o mais fundamental é o **serviço**. Empresas têm implementado o conceito

de serviço muito antes de ele ser utilizado na engenharia de software, elas já estão habituadas a consumir serviços de limpeza, serviços de entrega, serviços de segurança, serviços jurídicos e vários outros tipos de serviços [24].

Dentro do contexto da engenharia de software, (ERL, 2008) [22] define serviço como:

> *Um programa de software fisicamente independente, com características de design distintas para apoiar a computação orientada a serviços. Cada serviço recebe seu próprio contexto funcional distinto e possui um conjunto de capacidades relacionadas a esse contexto. Essas capacidades adequadas para a invocação de programas externos são comumente expressas em um contrato de serviços público, semelhante a uma Interface de Programação de Aplicação (API) tradicional.*

Portanto serviço é uma coleção de capacidades relacionadas a um contexto funcional, totalmente independente e que disponibiliza essas capacidades através de trocas de mensagens definidas em um contrato de serviços. Essa é uma definição bastante importante, pois posiciona o conceito de serviço de forma agnóstica a qualquer plataforma ou tecnologia. Normalmente serviços são diretamente associados a *Web Services* que é a plataformas mais popular para implementação de computação orientada a serviços [22].

Geralmente os vários serviços de um contexto relacionado disponibilizados por uma empresa ou um segmento significativo de uma organização, sejam padronizados e organizados em um catálogo chamado **inventário de serviços** [22].

Como serviços são granulares e atendem a um contexto funcional específico, para se automatizar processos de negócio mais complexos muitas vezes é necessário utilizar a **composição de serviços**, agregando e coordenando vários serviços para atender aos objetivos da tarefa. Por isso é fundamento o serviço ser independente do processo de negócios, e assim permitir ser reutilizado em múltiplas composições [22].

A computação orientada a serviços possui seu próprio paradigma de princípios de **design orientado a serviços** para modelar e especificar serviços e composições de baixo acoplamento, que são os elementos principais utilizados pela **lógica orientada a serviços** para se projetar a solução de um problema através de serviços [22].

Uma **arquitetura orientada a serviços** (SOA) é um modelo arquitetônico que define a criação, evolução, composições e inventários de serviços durante todo seu ciclo de vida, além

de descrever como é possível fornecer um ambiente adequado para a realização da lógica que foi projetada baseando-se nos princípios do design orientado a serviços [22].

Na Figura 6 é possível ver como todos esses conceitos se relacionam para forma a computação orientada a serviços.

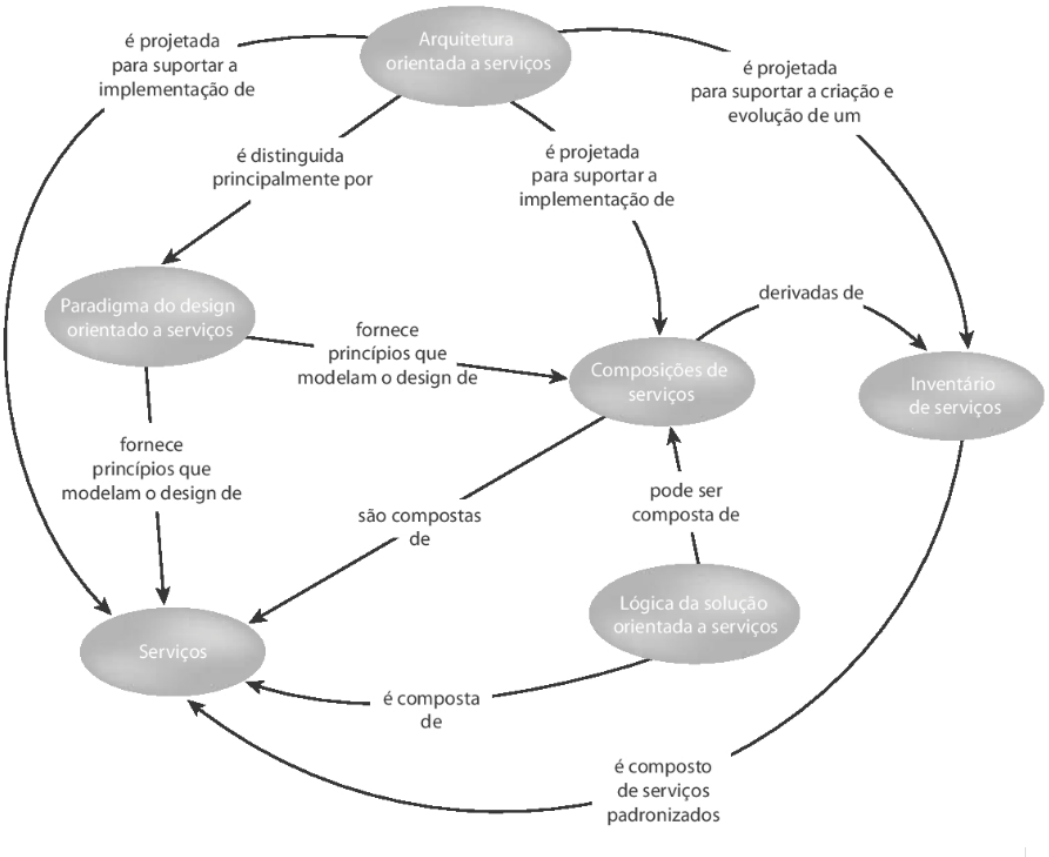

Figura 6 - Relacionamento dos conceitos da computação orientada a serviços [22]

2.3.2 Princípios da orientação a Serviços

Ao se construir soluções baseadas em uma lógica distribuída, normalmente as abordagens de design giram em torno do conceito de separação de preocupações (*separation of concerns*), que busca resolver um problema decompondo-o em problemas menores, ou em preocupações menores. Essa ideia permite dividir a lógica em funcionalidades independentes, em que cada funcionalidade é focada em resolver apenas uma preocupação. A principal vantagem de se solucionar problemas dessa forma, é que essas funcionalidades projetadas para resolver o problema imediato, podem ser também totalmente agnósticas em relação ao contexto do problema, e isso permite maximizar a reutilização.

Baseado nesse conceito de separação de preocupações a orientação a serviços estabelece os seguintes princípios de design [22]:

- **Contratos de serviço padronizados:** é por meio de contratos públicos que os serviços disponibilizam suas funcionalidades, seus modelos de dados e suas políticas. Por isso é importante assegurar que os contratos tenham granularidade adequada e sejam padronizados para manter a consistência com outros serviços relacionados, e tornar inventários de serviços mais governáveis [22].
- **Baixo acoplamento de serviço:** este princípio foca em diminuir a dependência entre o contrato, a implementação e os consumidores do serviço. Isto permite que o design e a implementação do serviço possam evoluir de forma independente, ao mesmo tempo em quem mantem a interoperabilidade básica entre os consumidores daquele serviço [22].
- **Abstração de serviço:** quanto mais detalhes internos do serviço puderem ser abstraídos, mais fácil será manter o baixo acoplamento descrito no princípio anterior, além de facilitar o design de composições de serviços [22].
- **Capacidade de reuso do serviço:** um esforço considerável durante design deve assegurar que os serviços projetados tenham um contexto funcional agnóstico e sejam tratados como recursos corporativos, a fim de maximizar a reutilização de lógica [22].
- **Autonomia de serviço:** para que os serviços realizem suas capacidades de modo consistente, eles precisam ser independentes ter um certo grau de controle sobre seu ambiente e seus recursos, durante o design várias considerações devem ser feitas sobre o nível de isolamento e normalização de serviço para se atingir um nível conveniente de autonomia [22].
- **Independência de estado do serviço:** idealmente ao se projetar serviços, deve-se evitar o gerenciamento excessivo de informações de estado, o serviço deve manter somente os estados necessários e assim evitar comprometer sua disponibilidade e autonomia. Sempre que possível, tentar utilizar sua arquitetura tecnológica para delegar esse gerenciamento de estados [22].
- **Visibilidade do serviço:** Para que o investimento em serviços se justifique, eles precisam ser facilmente identificados e entendidos, assim que houver oportunidades de reuso dentro da organização. Então é preciso levar em conta os mecanismos de descoberta e descrição das funcionalidades o serviço, ainda durante o projeto [22].

- **Composição de serviços:** a capacidade de compor serviços com efetividade é um dos aspectos cruciais da orientação a serviços. E é importante ressaltar que a média que a soluções orientadas a serviços continuam aumentando a complexidade das configurações de composição também aumenta. E por isso é importante durante o design considerar como serão feitas as composições complexas de serviços e assim evitar possíveis esforços de readaptação de serviços [22].

Um último item que não é exatamente um princípio, mas uma característica de serviços é que ele são interoperáveis por natureza, isso é uma consequência de todos os princípios descritos acima, cada um deles contribui de alguma forma para manter serviços interoperáveis mesmo em ambientes mesmo em ambientes mais heterogêneos [22].

2.3.3 Benefícios e desafios da orientação a serviços

Além dos benefícios óbvios de alto grau no reuso de software e uma arquitetura fracamente acoplada, adotar uma abordagem orientada a serviços traz uma série de benefícios importantes, porém vale ressaltar que sua aplicação traz alguns desafios. A seguir na Tabela 3 são apresentados os principais benefícios e alguns desafios da computação orientada a serviços.

Tabela 3- Benefícios e desafios da orientação a serviços

Benefícios	**Desafios**
Serviços podem ser oferecidos por vários fornecedores, permitindo as organizações desenvolverem aplicações interligando serviços de uma gama de fornecedores [6].	Criar serviços verdadeiramente reutilizáveis e agnósticos pode introduzir uma alta complexidade no design tanto da arquitetura quanto dos serviços em si [22].
A informações sobre o serviço são públicas aos seus consumidores autorizados, e não é necessário nenhum tipo de negociação ou implantação para integrar o serviço a uma aplicação [6].	Para se atingir serviços consistentes e compatibilidade, é necessário criar e utilizar padrões de design, incorporar esses padrões na cultura da pode encontrar resistência por parte dos desenvolvedores e arquitetos, ou desafios para se propagar esses padrões pela organização [22].

A aplicação consumidora de um serviço tem liberdade para trocar dinamicamente o serviço consumido, inclusive utilizando um serviço de outro fornecedor para realizar determinada tarefa [6].	Uma estratégia comum para desenvolvimento de serviços é conceituar inicialmente um inventário de serviços, definindo e projetando os serviços individuais, seus modelos e relacionamentos, o que pode trazer uma quantidade considerável de esforço para durante a etapa de análise [22].
É possível construir novos serviços apenas compondo serviços existentes de maneira inovadora [6].	Para se atingir o estado de agilidade no atendimento a novas demandas, é necessário que a orientação a serviços já tenha sido implementada com sucesso na organização [22].
Pode se utilizar um modelo em que o usuário paga pelos serviços utilizados de acordo com seu uso, em vez de comprar um componente caro que raramente é utilizado [6].	Conforme vários inventários de serviços vão sendo criados, maiores serão as exigências na governança desses inventários para mantê-los e evolui-los [22].

2.3.4 Web Services

Uma das tecnologias mais populares para o desenvolvimento de serviços são os padrões de *Web Services*. A definição utilizada pelo grupo de trabalho em arquitetura de *Web Services* do *World Wide Web Consortium* (W3C) é a seguinte [23]:

> *Web Service é um sistema de software projetado para suportar interoperabilidade, e interação máquina-a-máquina pela rede. Possui uma interface descrita em formato processável por máquina (especificamente WSDL). Outros sistemas interagem com o Web Service de maneira prescrita pela descrição utilizando mensagens SOAP, que são transmitidas por HTTP com serialização XML em conjunto de outros padrões relacionados a web.*

Outra definição é a de (SOMERVILLE, 2019) [6], que apresenta Web Service como:

> A definição de *web service* é *"um componente de software reutilizável, fracamente acoplado, que encapsula funcionalidade discreta, e que pode ser distribuído e acessado por meio de programação. Um web service é um serviço acessado usando protocolos padrões da internet e baseados em XML.*

Ou seja, é um conjunto de tecnologias, protocolos, padrões e formatos que juntos fornecem uma estrutura interoperável para construção de serviços. Na Figura 7 é possível ver a arquitetura padrão de um *Web Service*, que normalmente é composta por:

- **Contrato de serviço** fisicamente desacoplado, que consiste em um arquivo de definição no formato *Web Service Description Language* (WSDL), um outro arquivo contendo a definição do schema XML dos dados contidos nas mensagens seguindo o formato de XML Schema Definition Language (XSD) e possivelmente algumas outras definições de extensões que aumentam a qualidade e segurança do serviço.
- **Lógica central** das funcionalidades que esse Web Service implementa, podendo ser uma lógica personalizada implementada para este serviço, ou é possível reutilizar a funcionalidade de uma aplicação legada, com objetivo de empacotar e disponibilizar essa lógica através de padrões de comunicação de Web Services.
- **Lógica de processamento de mensagens** que requer uma série de parsers, processadores e agentes de serviço que tratam os eventos e tarefas relacionados a troca de mensagens. Normalmente a maior parte dessa lógica é fornecida através do ambiente de execução, mas pode ser customizada também.

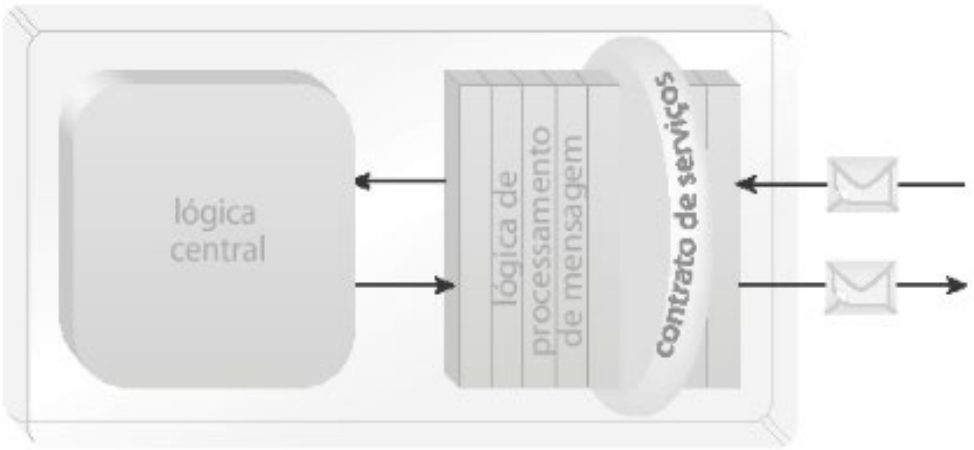

Figura 7 - Arquitetura padrão de um Web Service [22]

Além do Web Service em si, uma tecnologia importante nesta plataforma é o *Service Oriented Architecture Protoco*l (SOAP), que provê um padrão extensível para empacotar e trocar mensagens baseadas em XML. As mensagens SOAP podem ser transmitidas por uma grande variedade de protocolos de rede como HTTP, SMTP, FTP, inclusive protocolos proprietários [23].

2.4 REST

Conforme novos Web Services foram sendo desenvolvidos, percebeu-se que a maioria deles eram serviços simples, de uma única função, que não utilizavam os vários recursos de qualidade de serviço definidos nos padrões de trocas de mensagens, outro obstáculo comum é que a implementação de todos esses padrões requer uma quantidade razoável de processamento para criar, transmitir e interpretar mensagens baseadas em XML. Como uma alternativa, APIs REST (*REpresentational State Transfer*) tem sido escolhida como uma abordagem mais leve para se desenvolver serviços na web [6].

REST é um estilo arquitetural formalizado por (FIELDING, 2000) [25] para sistemas distribuídos de hipermídia, que foi derivado a partir da arquitetura da web. Um estilo arquitetural é uma forma de classificar arquiteturas de software, cada estilo define os requisitos e características necessárias que uma arquitetura deve seguir para pertencer a essa categoria [25].

Isso significa que REST não é uma arquitetura, não define padrões e nem abordagens para resolver os problemas e questões de implementação, mas define princípios para se identificar e classificar um determinado tipo de sistema como REST, sistemas que seguem os princípios desse estilo também são chamados de "*RESTful*" [26].

Qualquer sistema pode ser considerado RESTful, desde que sua arquitetura esteja em conformidade com os seis requisitos REST definidos por (FIELDING, 2000) [25], o que fornece uma grande liberdade no projeto de serviços desse tipo [27].

2.4.1 Requisitos de arquiteturas do tipo REST

REST define formalmente seis requisitos básicos que a arquitetura do software deve seguir, cada requisitos na verdade é uma decisão de design pré-determinada, através da aplicação desse conjunto de regras de design é possível obter uma arquitetura de sistema muito parecida com a

da web, incluindo suas melhores características de escalabilidade e interoperabilidade. Esses seis requisitos de uma arquitetura do estilo REST são:

Comunicação cliente servidor, esse é um requisito fundamental de qualquer arquitetura distribuída, e é totalmente baseado no princípio de separação de preocupações [27]. Essa separação melhora a portabilidade da interface com o usuário entre múltiplas plataformas, melhora a escalabilidade da arquitetura simplificando os componentes do servidor e principalmente permite que a lógica do lado do cliente e a lógica do lado servidor evoluam de forma independente [25].

Comunicação sem estado, cada requisição do cliente para o servidor precisa conter toda a informação necessária para ser compreendida e processada, as chamadas do cliente não podem depender de nenhum estado armazenado no servidor. Essa restrição permite melhorar a escalabilidade dos componentes do servidor, já que não é necessário armazenar o estado entre chamadas, além de economizar recursos. Outros benefícios importantes é uma melhor visibilidade e confiabilidade da comunicação, uma vez que todo o contexto da requisição está disponível na chamada, isso facilita ferramentas de monitoramento e a recuperação de falhas parciais [25].

Cache, as mensagens de resposta do servidor para os clientes definem explicitamente se seu conteúdo pode ou não ser armazenado em *cache* pelo cliente ou por qualquer outro componente de *middleware* que esteja participando da interação [27]. Essa é uma característica adicionada para melhorar a eficiência da rede, permitindo reduzir parcialmente a quantidade de interações entre cliente e servidor, o que melhora consideravelmente a performance percebida pelo usuário, mas em contrapartida, permitir o uso do cache em uma determinada resposta pode diminuir a confiabilidade, uma vez que os dados armazenados no cache podem ficar desatualizados [25].

Interface Uniforme, essa é uma das principais características do REST, ao se generalizar a interface dos componentes a arquitetura geral do sistema é simplificada e otimizada para sistemas distribuídos de larga escala, além de permitir desacoplar as implementações dos componentes dos serviços que ele fornece. A interface dos componentes em arquiteturas *RESTful* precisa se focar em quatro requisitos para se atingir a uniformidade, são eles, identificação de recursos, a manipulação de recursos através de representações, mensagens auto descritivas e o princípio de *Hypermedia as the engine of application state* (HATEOAS) [25]. Essas características serão abordadas em detalhes nas próximas sessões, assim como a principal abstração do estilo REST, que é o recurso.

Sistema em camadas, arquiteturas REST devem ser construídas sob os princípios de arquitetura em camadas, onde cada camada apenas tem apenas visibilidade das camadas adjacentes as quais ela está interagindo, dessa forma é possível estabelecer um limite sobre a complexidade total do sistema, e promover independência entre os componentes. As camadas também são úteis para encapsular serviços legados e implementação de camadas intermediárias (*middlewares* [27]) para compartilhar funcionalidades ou melhorar a escalabilidade do sistema fornecendo distribuição de carga entre redes, redundância e segurança dos serviços. A maior desvantagem de um sistema em camadas é a degradação da performance pelo aumento da latência, mas com uma utilização eficaz do *cache* é possível sobrepor esse problema [25].

Código sob demanda, esse é o único requisito opcional, e seu propósito é fornecer extensibilidade ao sistema, permitindo aperfeiçoar as funcionalidades do cliente apenas requisitando e executando código sob demanda na forma de *applets* ou scripts [25]. Como esse é um requisito opcional, arquiteturas que não possuem essa característica ainda são consideradas *RESTful*.

2.4.2 Interface Uniforme através de Recursos

A principal abstração da informação em uma arquitetura do estilo REST é o recurso, segundo (FIELDING, 2000) [25]:

> *Qualquer informação que possa ser nomeada, pode ser um recurso: um documento ou imagem, um serviço temporal (como por exemplo "O tempo hoje em Los Angeles"), uma coleção de outros recursos, um objeto não virtual (como uma pessoa), e assim por diante.*

Essa definição de recurso é especialmente útil porque abrange várias fontes de informação sem nenhum tipo de distinção por tipo ou implementação, que eleva o grau de generalidade com o qual os componentes lidam. Também porque recurso é o conceito central de alguns elementos necessários para satisfazer o requisito REST de interface uniforme entre os componentes que deve considerar a **identificação de recursos**, a **manipulação de recursos através de representações** e **mensagens auto descritivas** [25].

Uma arquitetura *RESTful* precisa definir um **identificador de recurso**, para localizar um recurso específico envolvido na interação entre componentes [25]. A identificação de recursos em REST não significa simplesmente seguir um padrão para criar identificadores

únicos, mas definir uma sintaxe padrão que permita expressar identificadores de uma forma uniforme [27], e que represente a real natureza dos recursos que se está identificando [25].

Os componentes REST **manipulam os recursos através de representações** que capturam o estado atual ou o estado que se desejado para um determinado recurso, essas representações de estado são transferidas entre os componentes, compostas basicamente de dados que são apenas uma sequência de bytes (um arquivo, um documento, uma mensagem HTTP), junto com metadados que descrevem esses dados, como por exemplo o formato da representação (também chamado de *media type*), seu tipo de codificação etc. [25].

(FIELDING, 2000) [25] descreve bem como deve funcionar a comunicação entre os componentes REST para a transferência de estados, que é parecido com uma invocação procedural, onde nos parâmetros de entrada são enviados: dados de controle da requisição, um identificador de recursos indicando o alvo da requisição, e opcionalmente uma representação. Os parâmetros de retorno dessa invocação são os dados de controle de resposta, uma representação opcional e metadados sobre o recurso também opcional [25].

Os **dados de controle**, definem principalmente o propósito da mensagem entre os componentes, ou seja, qual ação está sendo requisitada, ou então o significado da resposta devolvida. São os dados de controle, que indicam se a representação contida em uma requisição é o estado atual de um recurso que foi requisitado, ou se é um novo estado que se deseja atualizar daquele recurso, ou ainda outro tipo de representação, como a representação de uma condição de erro ao se acessar o recurso requisitado. Dados de controle são essencialmente informações sobre a requisição, alguns exemplos são dados de controle de cache, informações de controle de acesso, ou condições que devem ser atendidas pela requisição [25].

Já os **metadados sobre o recurso**, são dados que descrevem atributos específicos do recurso, e são independentes de sua representação, como por exemplo a versão do recurso ou a data de sua última alteração [25].

É através dessa estrutura de comunicação, que as **mensagens auto descritivas** são trocadas por componentes REST, e são esses elementos juntos que fornecem todas as informações necessárias para que mensagens possam ser entendidas por qualquer componente participante da comunicação. Esses são os requisitos necessários para se projetar uma interface uniforme entre componentes [25].

Outro elemento importante que uma arquitetura precisa atender para ser considerada *RESTFul* é o princípio de HATEOAS, que estabelece o uso de hipertexto (mais precisamente *hyperlinks* [27]), para se navegar através de uma implementação [25]. Segundo (FIELDING, 2008) [28], uma API REST deveria ser acessada sem nenhum conhecimento além de seu

identificador de recurso inicial e um conjunto de *media types* padronizado que são apropriados para audiência destinada (ou seja, que possam ser entendidos por qualquer cliente que use a API). A partir disso, todas as transições de estado da aplicação devem ser feitas pelo cliente, selecionando entre escolhas que estão presentes nas representações fornecidas pelo servidor [28].

2.4.3 SOA baseado em REST

Tanto Web Services, como APIs *RESTful* são uma das várias formas de se implementar sistemas orientados a serviços, mas diferente dos Web Services baseados em SOAP, não existem padrões, nem formatos pré-definidos para se especificar os contratos e as mensagens trocadas por serviços REST, porém existem algumas tecnologias que se tornaram padrão da instruída para desenvolver APIs RESTful [27].

Um serviço REST pode ser considerado uma combinação de um método de contrato uniforme, um identificador de recurso (onde recurso pode ser um conjunto de dados, processamento de lógica, arquivos e qualquer outra coisa que um serviço possa ter acesso), junto com a definição dos *media types* suportados [27].

A sintaxe mais comum utilizada para expressar os identificadores dos recursos expostos por serviços REST é o padrão de *Uniform Resource Identifier* (URI) utilizado pela web e definido pelo *Internet Engineering Task Force* (IETF) [29]. Esse padrão fornece uma sintaxe incluindo uma lista de caracteres permitidos e proibidos além de uma estrutura genérica para os identificadores [27]. A Figura 8 exibe essa sintaxe.

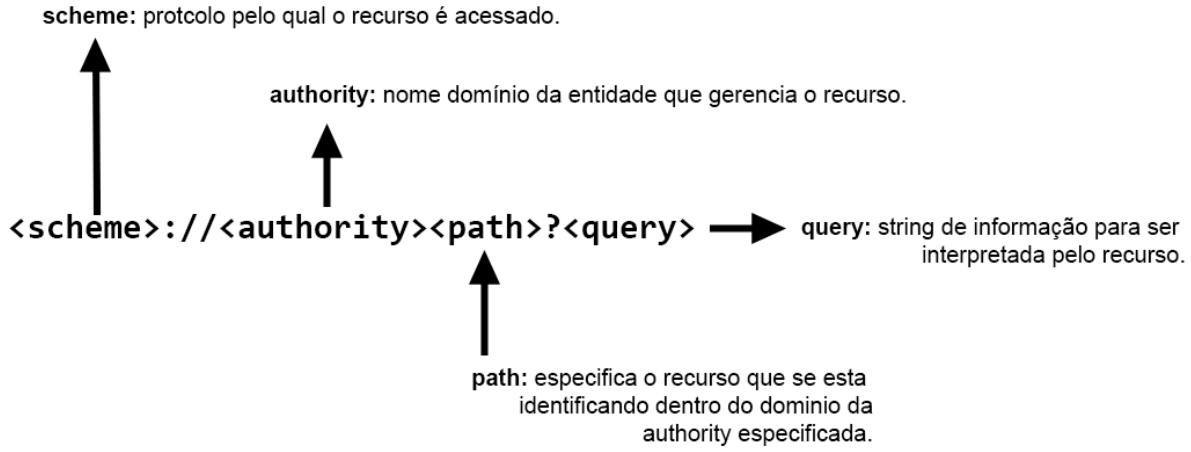

Figura 8 - A sintaxe genérica de uma URI

No contexto de serviços REST, método é um tipo de função fornecida por um contrato uniforme para processar dados e identificadores de recursos. O protocolo HTTP tem sido a tecnologia mais utilizada para se implementar contratos uniformes em REST, ele fornece um conjunto genérico de métodos pré-definidos na especificação do protocolo como GET, PUT, POST, DELETE, além de fornecer um conjunto de códigos de respostas e uma série de parâmetros de cabeçalho que permitem implementar mensagens auto descritivas [27].

O método utilizado em uma requisição especifica qual operação se deseja realizar sobre um recurso, um método GET normalmente é utilizado para se ler dados, enquanto o PUT para se atualizar um recurso existente, operações POST normalmente criam recursos e por fim o método DELETE é usado em situações que se deseja excluir um recurso [30].

Assim como durante o projeto de serviços REST se define quais métodos um recurso irá aceitar, também se define quais media types serão aceitos no processamento de um determinado método sobre um recurso especifico. Por exemplo, pode se definir que uma operação GET sobre o recurso de uma fatura poderá devolver a representação no formato XML, JSON ou PDF, ou que atualização via PUT do endereço de um cliente só aceitará representações no formato JSON. Normalmente em uma mensagem HTTP, os *media types* aceitos são especificados no parâmetro *Accept*, e o *media type* da representação contida no corpo da mensagem é especificado no parâmetro *Content-Type* do cabeçalho [27].

Um exemplo de mensagem de requisição e resposta HTTP é apresentado na Figura 9:

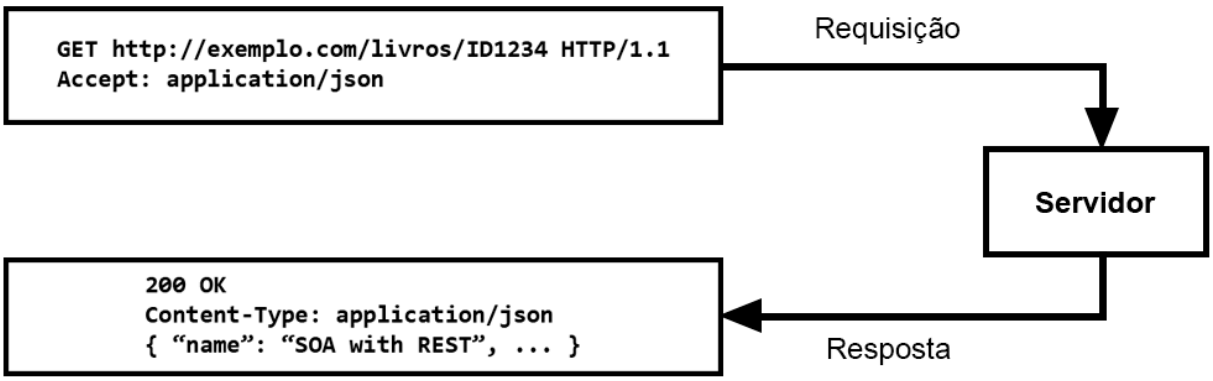

Figura 9- Comunicação com mensagens HTTP, exemplo de requisição e resposta.

2.5 Domínio da Segurança Pública

Para fornecer um contexto sobre as principais atividades de segurança pública. (FURTADO, 2002) apresenta uma modelagem bastante abrangente do domínio, baseada em sua experiencia durante o processo de modernização da Secretaria de Segurança Pública e Defesa da Cidadania do Estado do Ceará. O autor opta por uma descrição baseada em atores e diagramas de casos de uso da UML (*Unified Modeling Language*) para descrever a atividade geral de segurança pública, de forma independente de qualquer estruturação organizacional [31].

Os principais atores que representam a segurança pública apresentados por (FURTADO, 2002) e que são responsáveis pela tarefa de reduzir a ocorrência de crimes, são os cinco papéis exercidos pela polícia: **policial preventivo**, **policial investigativo**, **perito**, **policial de inteligência** e **corregedor**, suas principais ações são, impedir que crimes ocorram, e caso ocorram identificar, apreender os culpados, para que o poder judiciário (representado pelo ator **justiça**) possa puni-los, e assim dissuadir de que novos crimes aconteçam. A principal interação dos atores policiais é com o cidadão, cujos atores são representados pelos três papéis que o ele pode assumir, que são o de **vítima**, o de **suspeito ou indiciado** e o de **cidadão** comum [31].

A Figura 10 apresenta o diagrama de casos de uso com as interações realizadas entre esses atores, e em seguida esses casos de uso são descritos na Tabela 4.

Tabela 4 - Descrição dos casos de uso da atividade de segurança pública [31]

Nº	Caso de Uso	Ator	Descrição
1	Pede Socorro	Cidadão	O cidadão entra em contato com o atendente do centro de emergência para solicitar socorro em uma situação envolvendo os órgãos de segurança.
2	Despacha	Atendente de Emergência	O atendente de emergência, ao receber a solicitação de socorro, envia uma comunicação à polícia para que o atendimento seja efetuado
3	Realiza Patrulhamento	Policial Preventivo	O policial preventivo realiza rondas para fazer o patrulhamento e evitar ocorrências delituosas.

4	Presta Socorro	Policial Preventivo	O policial preventivo presta socorro a um cidadão atendendo a um chamado.
5	Realiza Prisão	Policial Preventivo	O policial preventivo, ao realizar um atendimento ou durante uma patrulha, pode efetuar uma prisão.
6	Presta Queixa	Vítima	O cidadão pode também se dirigir à polícia para presta queixa sobre algum fato que ele considera delituoso.
7	Presta Informação	Cidadão	O cidadão pode igualmente fornecer informações a polícia sobre qualquer fato que ele considera relevante. O setor de inteligência da polícia é responsável por receber estas informações, como denúncias e tratá-las.
8	Investiga	Policial Investigativo	O policial investigativo ao se deparar com um evento delituoso, realiza investigações para identificar os culpados.
9	Colhe Provas	Policial Investigativo / Perito	O policial investigativo procura provas para identificar culpados do evento criminoso. O perito também o faz, no que concerne a identificação de provas materiais.
10	Captura	Policial Investigativo	O policial investigativo captura um suspeito de crime ou já indiciado e ordenado de prisão pela justiça
11	Realiza Procedimento	Policial Investigativo	O policial investigativo realiza os procedimentos legais, como boletins e termos de ocorrências, inquéritos etc., com vistas a preparar o processo que será enviado à justiça para julgamento.
12	Indicia	Policial Investigativo	O policial investigativo depois da realização de um procedimento, pode vir a inculpar alguém por um crime e indicá-lo por ele.

13	Produz Conhecimento	Inteligência	O policial de inteligência, de posse de informações coletadas em diversas fontes, produz conhecimento que pode ser útil aos diversos setores policiais nas suas atividades.
14	Ordena Prisão	Justiça	Após análise pelo juiz, um mandado de prisão pode ser expedido para cumprimento pela autoridade local.
15	Identifica civil e criminalmente	Perito	O perito realiza procedimentos técnicos para registrar o cidadão e, no caso deste praticar algum ilícito, identificá-lo como criminalmente responsável. Esses registros serão úteis também como auxílio na identificação de suspeitos de realização de crimes.
16	Avalia Polícia	Corregedor	O corregedor avalia o trabalho policial, seja investigativo ou preventivo, zelando para que suas atividades sejam desenvolvidas em perfeita consonância com a legislação em vigor.

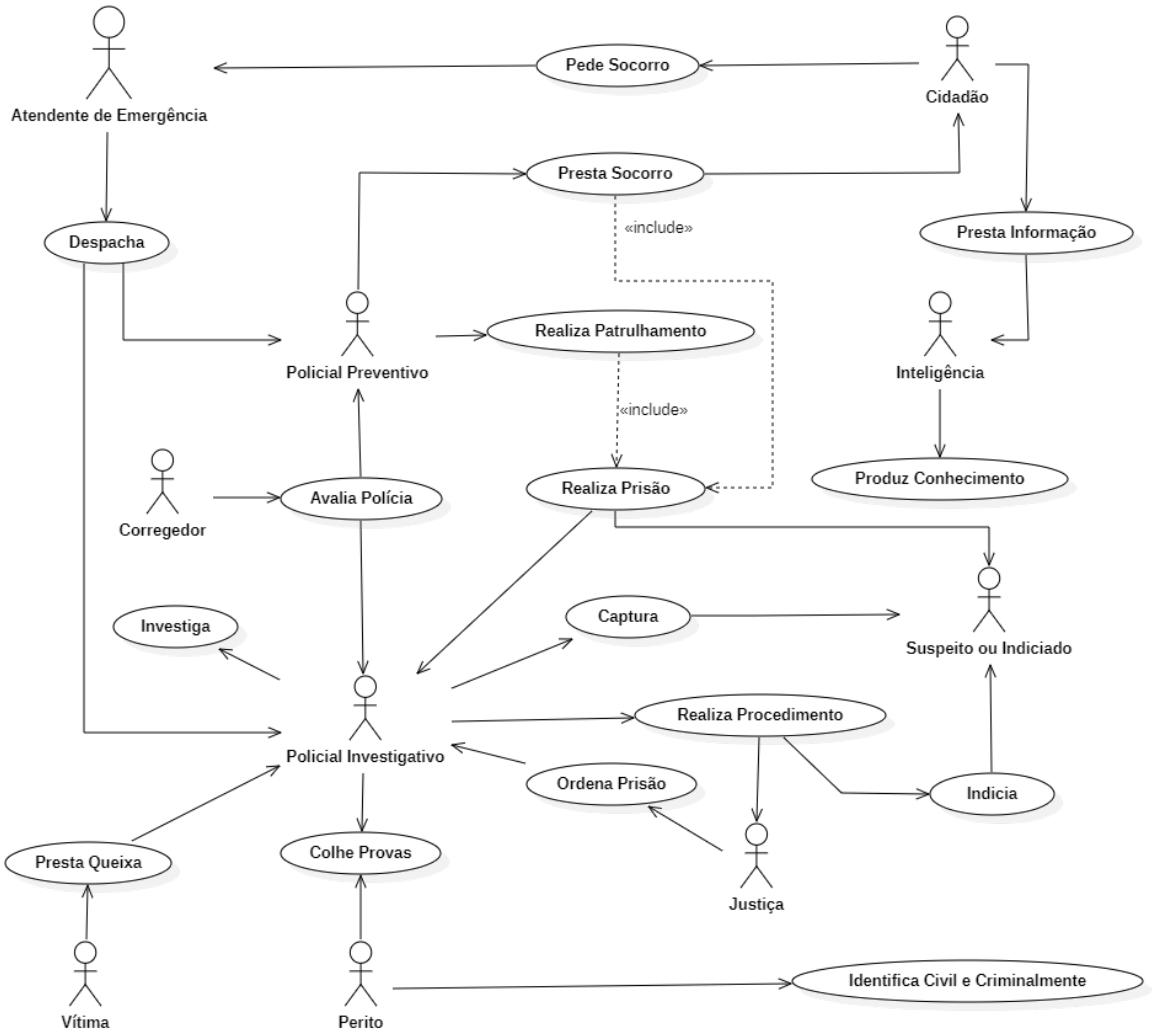

Figura 10- Diagrama de casos de uso da atividade de segurança pública [31]

Baseado nesse contexto descrito acima, (FURTADO, 2002) agrupa os casos de uso relacionados em cinco outros contextos menores, que serão a base dos sistemas de informação apresentados em seu trabalho, esses sistemas são descritos desde sua abstração inicial, até as funcionalidades de seus módulos, requisitos de hardware e software, assim como estratégias de gestão. As questões relacionadas a hardware, software, gestão e outros detalhes são muito específicos aos sistemas adotados na secretaria de segurança pública do Ceará, porém as abstrações fornecidas pelo autor, são extremamente valiosas para este trabalho. Os cinco contextos identificados por (FURTADO, 2002), são listados abaixo, eles serão apresentados nas próximas seções.

- **Suporte ao atendimento de emergências**, para auxiliar no recebimento de chamadas e atendimento ao cidadão.
- **Suporte à polícia investigativa,** para registro e tratamento de ocorrências policiais;
- **Suporte à identificação civil e criminal,** para tratamento das informações relativas à identificação civil, criminal e de medicina-legal.
- **Suporte à inteligência policial,** para registro e tratamento de informações coletadas por agentes de inteligência.

2.5.1 Suporte ao atendimento de emergências

O primeiro sistema que (FURTADO, 2002) apresenta é para suportar as operações do Centro Integrado de Operações de Segurança esse sistema é baseado no contexto do atendimento de emergências, esse contexto envolve os casos de uso 1 a 5 descritos na Tabela 4.

Os principais atores desse contexto são o atendente telefônico e o despachante. O atendente telefônico realiza o primeiro atendimento ao cidadão, e registra as informações iniciais da ocorrência no sistema, ele confere e valida a veracidade das informações fornecidas utilizando dados e internos, ou fornecidos por outras fontes, como por exemplo pela companhia telefônica, e caso necessário e assim enviar o pedido aos despachantes dos respectivos órgãos de segurança pública.

Os despachantes são responsáveis por localizar viaturas próximas ao local da ocorrência e comunicar a ocorrência através de uma mensagem, solicitando o pronto atendimento e fornecendo todas a informações relevantes que já foram coletadas. E assim os órgãos de segurança responsáveis podem prestar o atendimento necessário ao cidadão.

A Figura 11 apresenta o diagrama de casos de uso com as interações realizadas por estes atores, e em seguida esses casos de uso são descritos na Tabela 5.

Tabela 5 - Descrição dos casos de uso no atendimento de emergências [31]

Nº	Caso de Uso	Ator	Descrição
1	Solicita Atendimento	Cidadão	O cidadão entra em contato com o atendente do centro de emergência para solicitar atendimento em uma

			situação envolvendo os órgãos de segurança.
2	Localiza Chamada	Atendente Telefonista	O atendente de emergência ao receber a solicitação de socorro envia comunicação à polícia para que o atendimento seja efetuado e cria um evento no sistema caracterizando a ocorrência.
3	Solicita Informações	Atendente Telefonista	O atendente telefonista, durante o atendimento da chamada telefônica, solicita informações para caracterizar a ocorrência e identificar se a solicitação se refere a um fato verídico e que necessita de atendimento emergencial dos órgãos de segurança.
4	Repassa a Chamada	Atendente Telefonista	O atendente telefonista, após colher as informações que caracterizam o tipo de ocorrência, repassa a mesma para o(s) despachante(s) do órgão/área onde ela ocorreu.
5	Monitora Atendimento	Supervisor Telefonista	O supervisor telefonista acompanha o trabalho do atendente para zelar pela qualidade do serviço e o auxiliando em situações que isso for necessário.
6	Monitora Despacho	Supervisor Despacho	O supervisor de despacho supervisiona e monitora todas as atividades da estação do operador de despacho da área.

7	Localiza a Viatura	Despachante	Cada despachante, ao receber a solicitação de atendimento, identifica a(s) viatura(s) mais próxima(s) e adequada(s) para realizar o atendimento.
8	Despacha	Despachante	O despachante da área onde ocorreu o evento, ao identificar a(s) viaturas(s) a ser(em) despachada(s), fornece informações obre a ocorrência e despacha o atendimento pela polícia de campo.
9	Presta Atendimento	Policial Preventivo/Policial Investigativo/Bombeiro/Perito	O policial/bombeiro presta atendimento a um cidadão atendendo a um chamado.

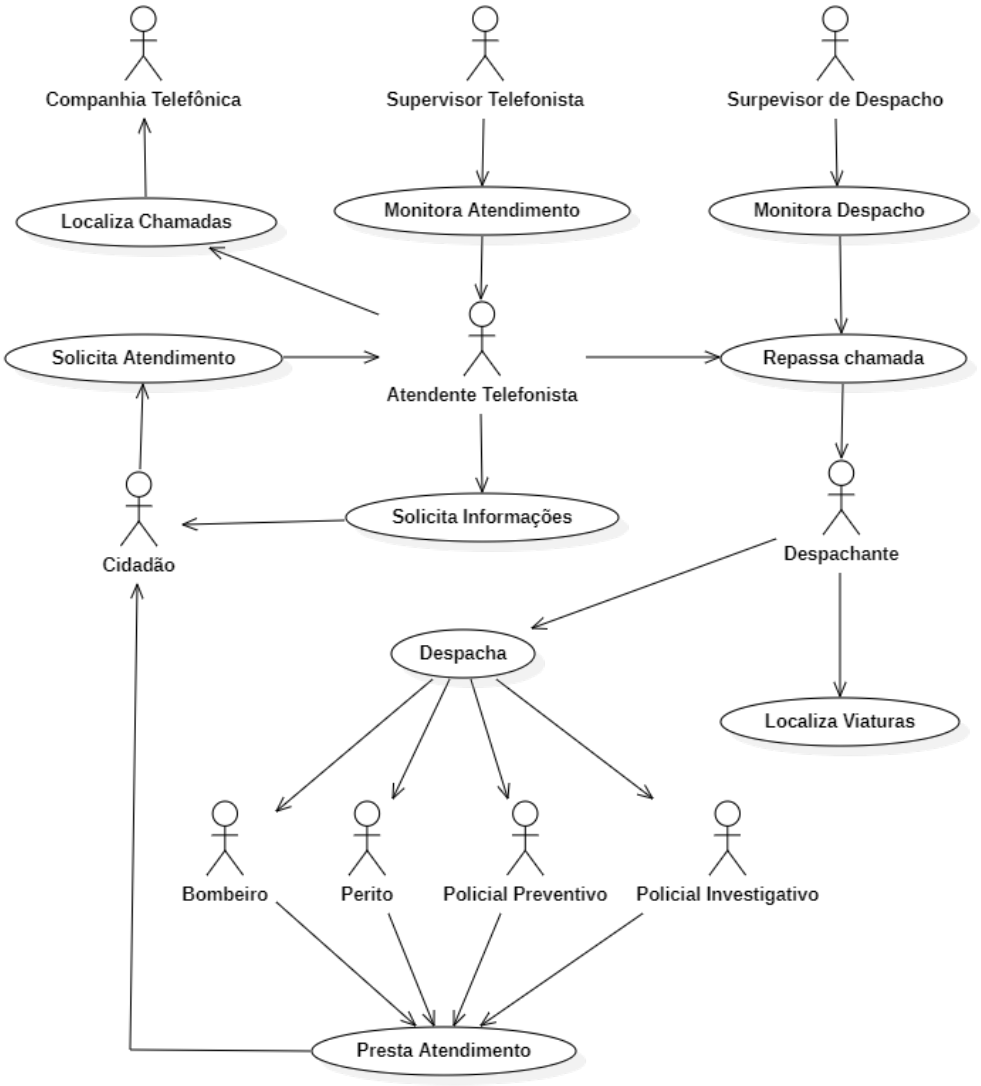

Figura 11 - Diagrama de casos de uso no atendimento de emergência [31]

Grande parte das funcionalidades apresentadas para este sistema, são dependentes de tecnologias especificas como comunicação via rádio, centrais telefônicas PABX, sistemas de comunicação com redes móveis, sistemas de informações geográficas, GPS, sistemas de localização automática de veículos. Uma outra parte são funcionalidades mais relacionadas a software, como acesso a bases dados tanto locais para registro e posterior geração de estatísticas e relatórios das ocorrências, como acesso a bases de dados remotas, normalmente bases de dados sobre informações do cidadão, veículos, registro de armas, endereços e outros dados ficam localizadas em outros órgãos [31].

2.5.2 Suporte à polícia investigativa

O segundo contexto apresentado, é o que diz respeito as atividades da polícia investigativa, os casos de uso envolvidos nesse contexto são o 6, 8, 10, 11, 12 e 14 da Tabela 4. Os sistemas de informação que visam apoiar esse contexto, devem suportar funcionalidades para abranger as atividades típicas de uma delegacia da polícia civil (polícia investigativa).

O trabalho típico de uma delegacia, normalmente envolve o cadastro das informações relacionadas aos procedimentos policiais, como boletins de ocorrência (BOs), Termos Circunstanciados de Ocorrência (TCOs), inquéritos policiais e atos infracionais, assim como a confecção de documentos oficiais utilizados nesses procedimentos.

Os principais atores envolvidos no contexto da polícia civil são o policial investigativo (delegado, escrivão e inspetor), o policial preventivo, o perito (criminal e médico-legista), a justiça e o cidadão (vítima/indiciado). O delegado é o responsável pela investigação policial, é ele quem determina o registro do BO, instaura o inquérito policial, determina que se lavre o TCO, ou indicie alguém.

Ao tomar conhecimento de um ato delituoso, o delegado determina ao escrivão que faça o registro do BO como medida preliminar, caso o autor do delito não tenha sido preso em flagrante ou sua identidade ainda não é conhecida. Após a comprovação do delito, por investigação, testemunhas ou flagrante, o BO pode ser convertido em um TCO para o caso de crimes de menor potencial ofensivo, com pena de até dois anos, ou em um inquérito policial caso seja um crime de maior gravidade com pena maior que dois anos. Outro procedimento é o ato infracional, usado para registrar crimes onde o autor do delito é menor de idade.

A Figura 12 apresenta o diagrama de casos de uso com as interações realizadas no contexto da polícia investigativa, e em seguida esses casos de uso são descritos na Tabela 6.

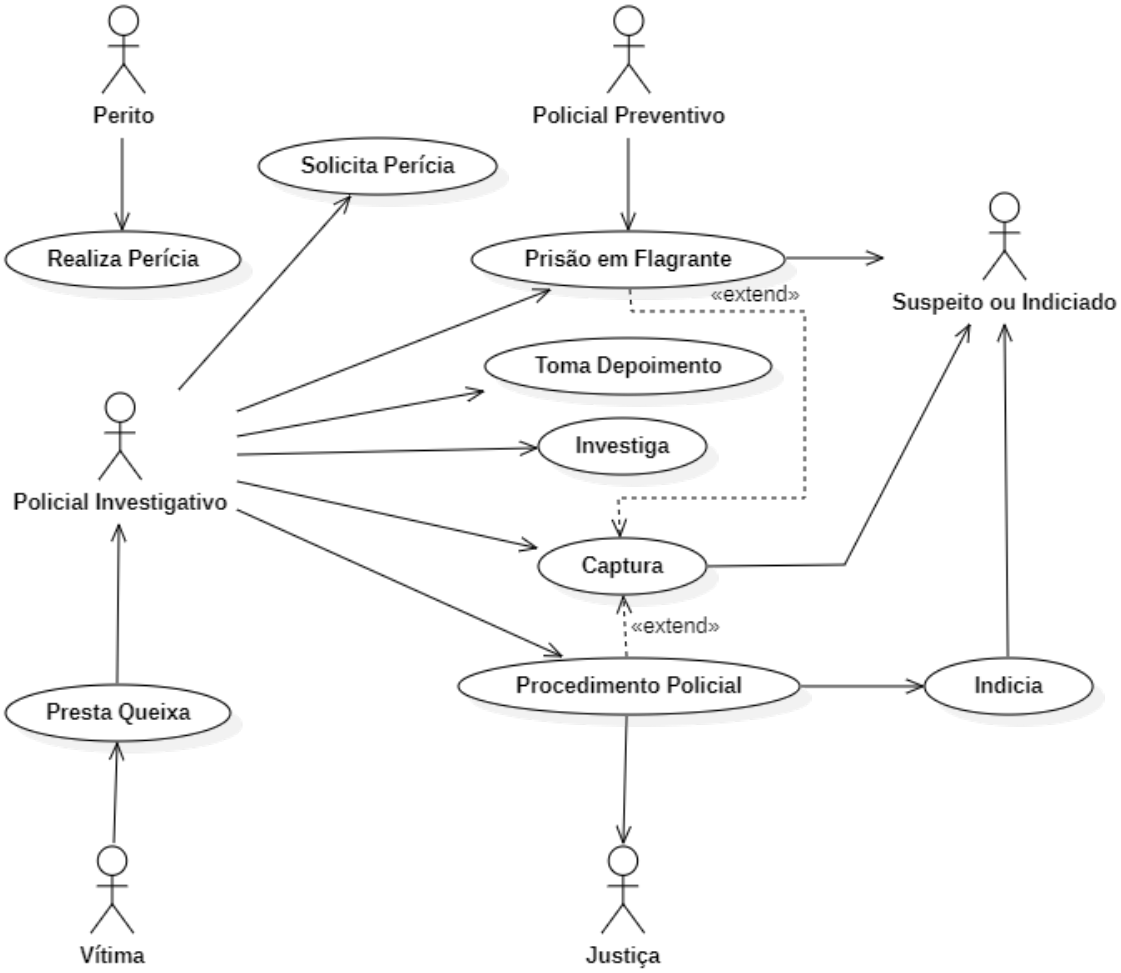

Figura 12 - Diagrama de casos do contexto da polícia investigativa [31]

Tabela 6 - Descrição dos casos de uso do contexto da polícia investigativa [31]

Nº	Caso de Uso	Ator	Descrição
1	Presta Queixa	Cidadão	O cidadão vai à delegacia e notifica um fato criminoso ao policial responsável pelo registro da ocorrência.
2	Investiga	Policial Investigativo	O delegado determina ao escrivão expedição de ordem de serviço para que o policial investigativo responsável pela investigação, faça o levantamento de testemunhas, da identidade do criminoso, quando

			estes ainda não são conhecidos, e das demais provas de esclarecimentos do crime.
3	Procedimento Policial	Policial Investigativo	O delegado instaura inquérito policial, ou, se a pena prevista para o crime praticado for até dois anos, formaliza o TCO. Concluídas todas as diligências, o delegado faz um relatório circunstanciado de todo o apurado nos autos do inquérito, representando ao Ministério Público pela prisão preventiva do infrator, nos casos previstos em Lei e faz remessa dos autos ao Poder Judiciário, acompanhado dos objetos e/ou materiais apreendidos.
4	Solicita Perícia	Policial Investigativo	Conforme o caso, o delegado solicita a realização de exames periciais, faz apreensão de materiais e/ou objetos relacionados com o crime.
5	Toma Depoimentos	Policial Investigativo	O delegado ordena ao escrivão que notifique as testemunhas, vítima(s) e/ou infrator(es) para que, no dia e hora marcados, prestem informações sobre o crime.
6	Captura	Policial Investigativo	Recebido o mandado de prisão da Justiça, o delegado determina aos seus agentes a captura do acusado e em seguida o encaminha ao presídio.
7	Realiza Perícia	Perito	O perito criminal faz o levantamento do local onde ocorreu o crime e a perícia de documentos. O médico legista realiza exames de corpo de

			delito. O toxicologista realiza exames laboratoriais em materiais ou objetos relacionados ao crime.
8	Prisão em Flagrante	Policial Investigativo/ Policial Investigativo	O indivíduo que é preso no ato ou logo após o cometimento do delito é conduzido à presença do delegado acompanhado das testemunhas do ato, para que ele seja autuado em flagrante, ou seja, realizado o TCO, quando a pena não for superior a dois anos. Se o infrator for menor de 18 anos, será encaminhado à delegacia especializada para formalizar o auto infracional.
9	Indicia	Policial Investigativo	O policial investigativo depois da realização de um procedimento, pode vir a inculpar alguém por um crime e indicá-lo por ele.

Funcionalidades de consulta em bancos de dados são muito importantes para o trabalho da polícia investigativa, desde buscas textuais simples até exemplos mais complexos, como buscas utilizando biometria, ou o álbum eletrônico de fotos onde é possível realizar uma consulta em um banco de dados de fotografias, tanto pelo nome quanto pelas características físicas do infrator. Outro exemplo também é a consulta integrada, que recebe como parâmetro de busca o nome de uma pessoa, e realiza uma pesquisa fonética no banco de dados, retornando todos os envolvimentos daquela pessoa, como mandados, documentos, prisões e procedimentos legais.

A busca fonética, considera pequenas variações na escrita, como equivalentes desde que sua fonética seja parecida, por exemplo, uma busca por uma pessoa com o nome “Gerson Pereira de Souza” também consideraria os resultados de “Jerson Pereira de Sousa”.

Outra funcionalidade bastante importante é a integração com bancos de dados de outros órgãos de segurança, como a polícia preventiva, a polícia técnico-cientifica, órgãos da justiça e do sistema prisional. A busca por informações nesses órgãos é essencial para o trabalho de investigação, e de identificação criminal.

Uma das responsabilidades da polícia civil é identificar o paradeiro de pessoas desaparecidas, por isso é interessante ter ferramentas que permitam a polícia divulgar dados das pessoas desaparecidas na internet e possam cadastrar as denúncias feitas pelos cidadãos em uma base única compartilhada por todos os órgãos de segurança.

2.5.3 Suporte à identificação civil e criminal

O próximo contexto apresentado pelo autor é o que se refere as atividades da polícia técnico-científica nos institutos de identificação, representados pelos casos de uso 9 e 15 da tabela 2.4. O ator central desse contexto é o perito, e suas principais atividades é a identificação, classificação, registro e controle das informações individuais do cidadão. As principais informação que são colhidas durante a identificação são as características do cidadão como nome, foto, filiação, data de nascimento e, sobretudo, impressões digitais.

Além da identificação civil, os institutos de identificação, também tem atividades relacionadas com a identificação criminal, que busca relacionar um cidadão já identificado com sua vida criminal.

A Figura 13 apresenta o diagrama de casos de uso com as interações realizadas neste contexto, e em seguida esses casos de uso são descritos na Tabela 7.

Tabela 7 - Descrição dos casos de uso do suporte à identificação civil e criminal [31]

Nº	Caso de Uso	Ator	Descrição
1	Solicita identificação Civil	Cidadão	O cidadão solicita ao instituto de identificação que realize sua identificação.
2	Solicita identificação Criminal	Policial Investigativo	A autoridade policial solicita a identificação criminal de algum cidadão indiciado por crime.
3	Solicita Nada Consta	Cidadão	O cidadão solicita que o instituto de identificação verifique seus antecedentes criminais.
4	Colhe dados textuais	Perito	O perito, tanto para realizar a identificação civil quanto para a criminal, colhe dados textuais sobre

			o cidadão como nome, sexo, filiação, data de nascimento etc.
5	Colhe dados biométricos	Perito	O perito colhe as impressões digitais do cidadão e/ou do criminoso.
6	Emite atestado de antecedentes	Perito	O atestado de antecedentes indica se cidadão tem ou não alguma passagem pelo cadastro criminal.
7	Emite carteira de Identidade	Perito	A carteira de identidade será emitida como dados que individualizam o cidadão.
8	Realiza classificação	Perito	As digitais são analisadas e classificadas segundo suas características principais.

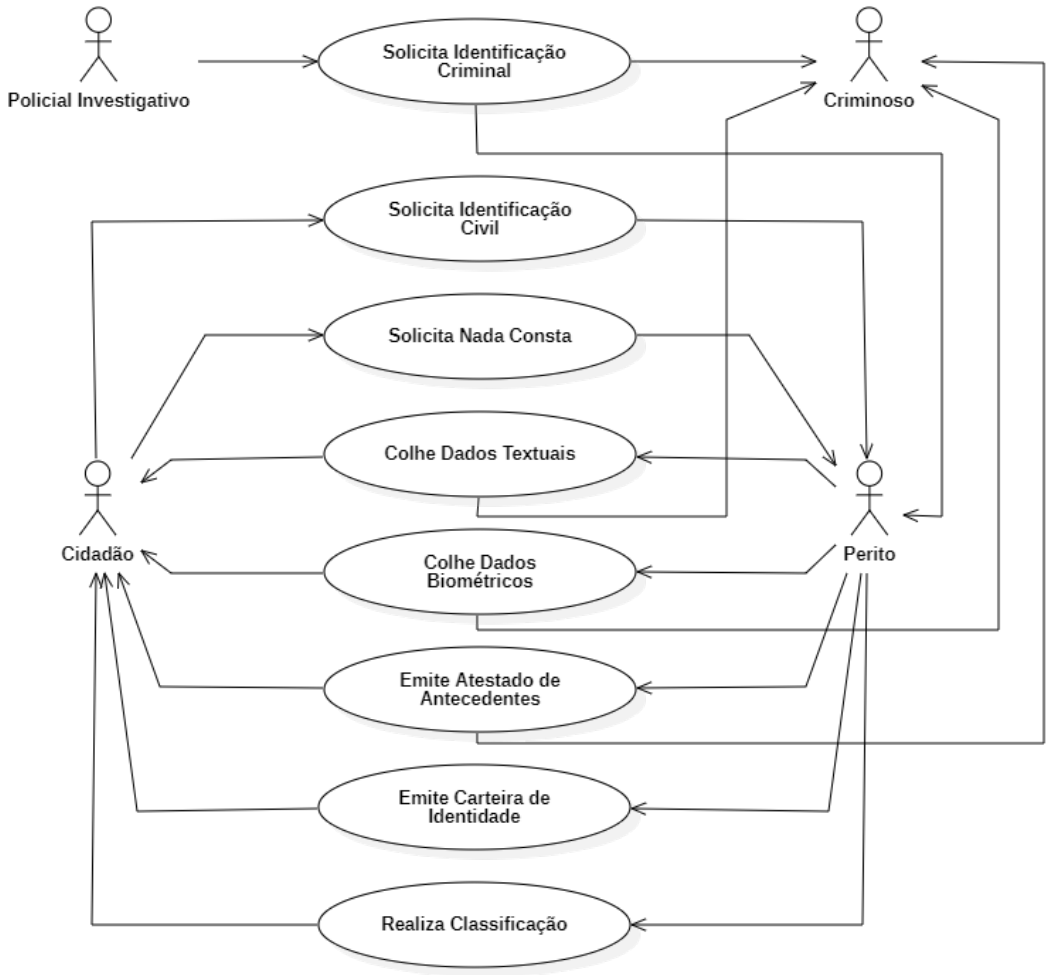

Figura 13- Diagrama de casos de uso do suporte à identificação civil e criminal [31]

As principais funcionalidades destacadas para esse contexto, são relacionadas ao registro e busca de imagens em bancos de dados, em especial as imagens provenientes da coleta de impressões digitais, coleta de assinaturas e fotografias do cidadão.

Outra atividade muito importante nos institutos de identificação é a busca nessas bases de dados, as mais comuns são as buscas 1:1 e as buscas 1:N. Nas buscas 1:1 compara-se a impressão digital de uma pessoa, com o seu registro na base de dados, a fim de comprovar sua identidade. Já as buscas 1:N a impressão digital da pessoa é comparada com parte ou toda a base de dados, o objetivo dessa busca é identificar a quem pertence uma determinada impressão digital.

Institutos de identificação também são responsáveis por expedir a carteira de identificação do cidadão, e outros documentos oficiais como por exemplo o atestado de bons antecedentes comprovando que cidadão não possui registros criminais.

2.5.4 Suporte à inteligência policial

O próximo contexto é um pouco menor que os anteriores e diz respeito às atividades da inteligência policial, principalmente as de coletar e tratar informações advindas de diversas fontes para subsidiar o trabalho dos órgãos de segurança.

O objetivo principal é construir um banco de dados contendo informes, informações e conhecimentos relativos a assuntos relevantes provenientes tanto de fontes abertas como de fontes fechadas. Essas informações serão categorizadas pelos analistas e agentes de inteligência para facilitar o compartilhamento delas com os diversos setores de inteligência dos órgãos de segurança pública.

A Figura 14 apresenta o diagrama de casos de uso com as atividades principais da policial de inteligência, e em seguida esses casos de uso são descritos na Tabela 8.

Tabela 8 - Descrição dos casos de uso das atividades do policial de inteligência [31]

Nº	Caso de Uso	Ator	Descrição
1	Solicita Missão	Secretário	O secretário que é a instancia máxima da segurança, solicita da inteligência uma missão.
2	Solicita Informação	Analista	O analista coleta os dados existentes e, caso seja suficiente, elabora um

			relatório de inteligência/informação. Em caso de insuficiência de dados, ele encaminha um pedido de informação para os devidos órgãos, na pessoa de seus analistas de campo ou operações, para que eles incorporem novos dados a sua base de estudos.
3	Coleta de dados	Analista/Agente de Operações	O analista coleta os dados existentes em fontes abertas (fontes que estão disponíveis como meios de comunicação, livros etc.).
4	Busca de dados	Analista/Agente de Operações	O agente de inteligência (operações) busca dados nas fontes fechadas (fontes que não estão disponíveis, como ligações telefônicas, informantes etc.)
5	Produz conhecimento	Analista/Agente de Operações	O analista em posse dos dados coletados e dos relatórios recebidos, produz um relatório de inteligência/informação e repassa para os interessados.

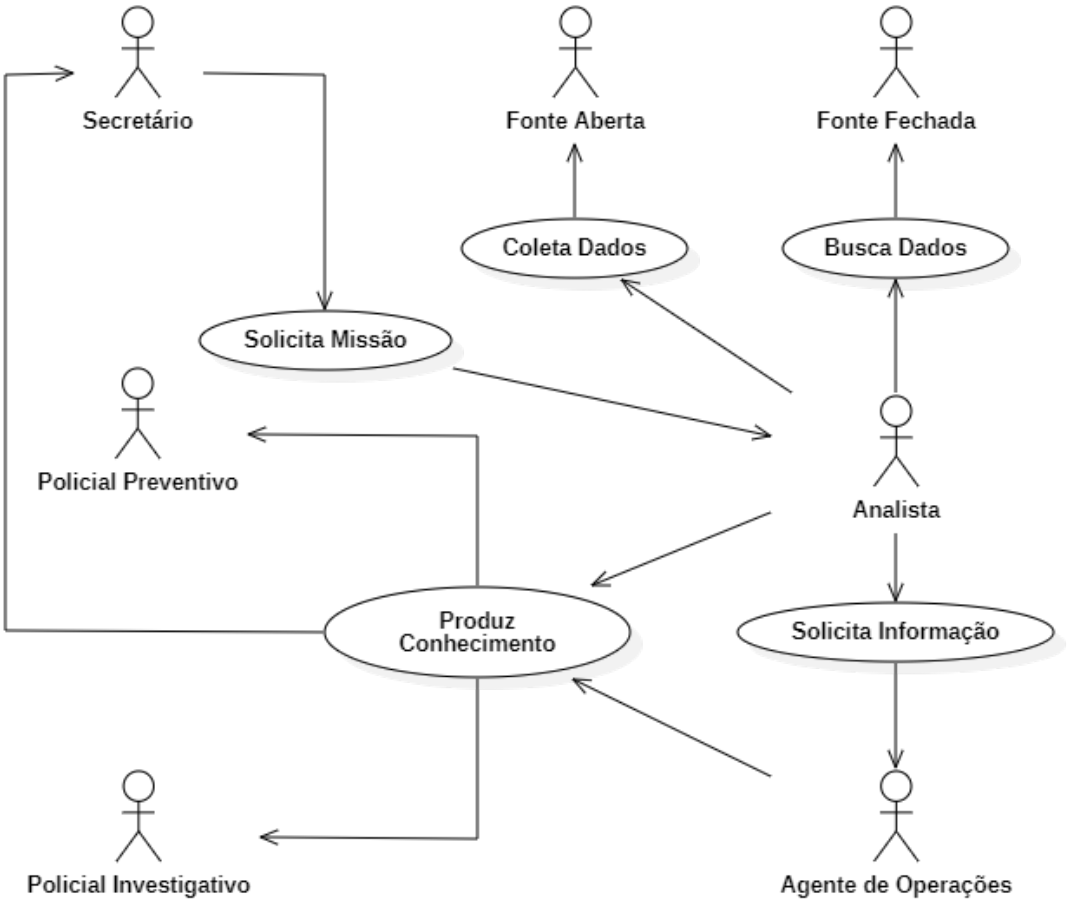

Figura 14 - Diagrama de casos de uso das atividades do policial de inteligência [31]

Funcionalidades de integração com diferentes fontes de informação são bastante importantes no contexto de inteligência, isso permite se consultar fontes abertas como o de companhias de água, de eletricidade, arquivos criminais, arquivos de lojistas, de hotéis e os vários outros tipos de informações disponibilizadas na internet de um modo geral e alimentar a base de dados interna.

Além disso é interessante que existam funcionalidades de classificação automática da informação registrada na base de dados, isso facilita a busca por documentos e dados para enriquecer a criação de dossiês relacionando aos temas requisitados a inteligência policial.

Também são responsabilidades da inteligência realizar a interceptação legal de comunicações, gravando e registrando as comunicações que forma monitoradas de um determinado alvo. E o suporte a tele denúncia, ou seja fornecer meios e o tratamento adequando para que o cidadão possa colaborar com o trabalho da polícia fornecendo informações úteis.

3 METODOLOGIA

Conforme apresentado na Seção 2.2.4 o desenvolvimento de um framework é um processo evolutivo, após o desenvolvimento da versão inicial do framework, e conforme novas instancias são criadas a partir dele, os problemas e experiências são capturadas, para que o framework possa ser reavaliado e novas funcionalidades ou correções podem ser incluídas nas versões futuras, neste trabalho apresentaremos o desenvolvimento inicial do framework proposto.

O processo geral de desenvolvimento de um framework, conforme mostra a Figura 15, possui três grandes etapas, primeiro inicia-se com a análise do domínio, é nesta etapa que são descobertos os requisitos funcionais do framework, também já é possível se ter uma noção parcial dos hot spots e dos frozen spots. Depois, na fase de design do framework, são especificados e modelados todos os pontos de flexibilidade e extensibilidade identificados, as abstrações, os *hot spots* e os *frozen spots*. Por fim, na etapa de instanciação aplicações são implementadas para fins de teste e validação [17].

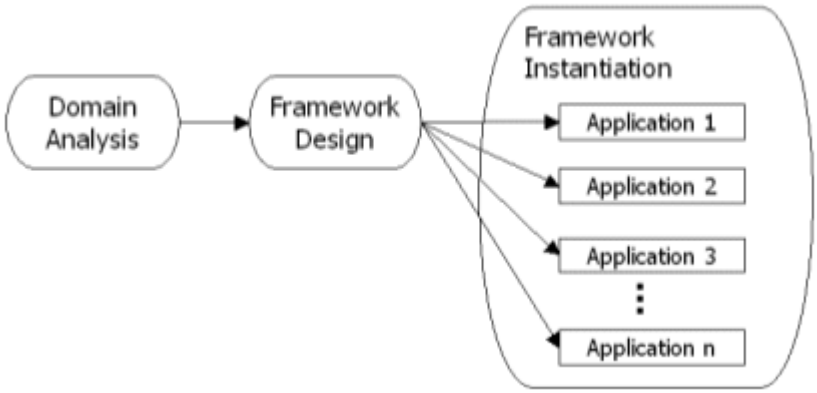

Figura 15 - Processo de desenvolvimento de um framework [17]

Existem várias metodologias para apoiar esse processo de desenvolvimento, algumas delas como *Feature-Oriented Domain Analysis* [32] e a *Feature-Oriented Reuse Method* [33] se baseiam na identificação e criação de diagramas funcionalidades para mapear funcionalidades do domínio e definir os pontos de flexibilidade, outras abordagens como a *Hot-Spot-Driven* [34] utiliza *hot-spot cards*, cujo objetivo é extrair dos especialistas do domínio toda a informação necessária para a identificação dos *hot spots*. Porém neste trabalho será utilizada a metodologia baseada em UML descrita por (Yang et al., 1999) [35], o principal motivo para

essa decisão foi o seu foco na utilização da UML durante as etapas, o que facilita a utilização das abstrações apresentadas na Seção 2.5.

3.1 Metodologia baseada em UML

A abordagem de (Yang et al., 1999) [35] consiste em quatro fases, análise, design, implementação e teste, cada fase possuí um conjunto específico de tarefas conforme mostra a Figura 16.

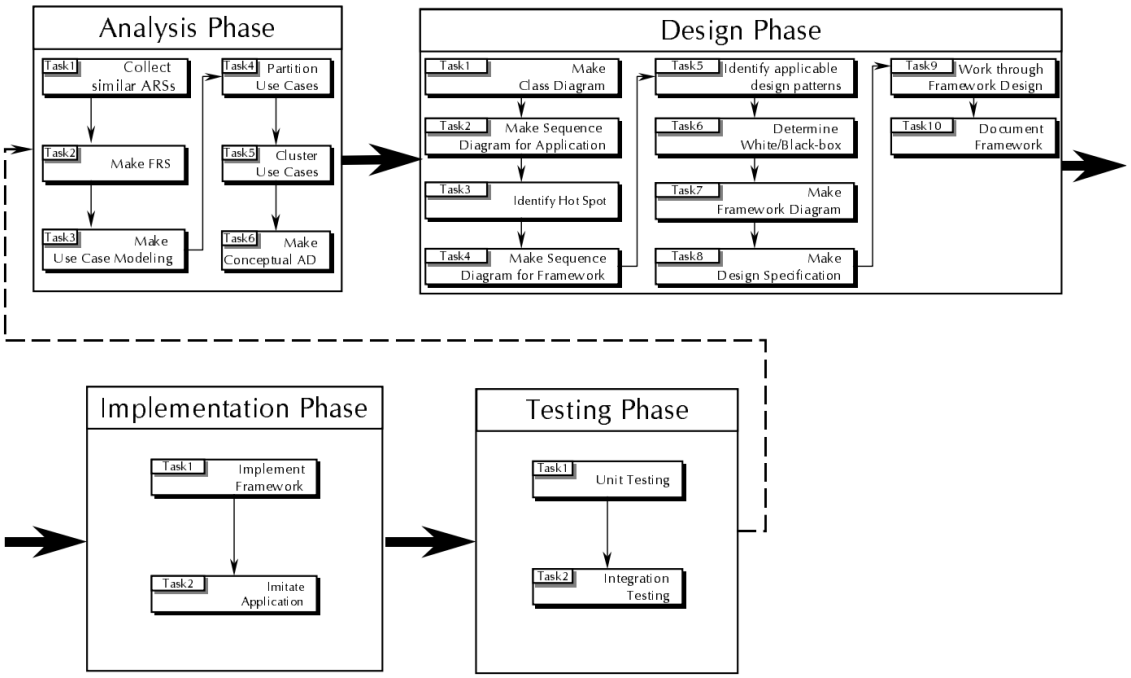

Figura 16- Processo de desenvolvimento de framework [35]

A fase de análise consiste em criar a *Framework Requirement Specification* (FRS) extraindo funcionalidades semelhantes de um conjunto de especificações de requisitos de aplicações do domínio, chamadas de *Application Requirement Specification* (ARS), com base nesses requisitos um modelo de casos de uso é construído, os frameworks envolvidos são identificados e suas interações mapeadas utilizando um diagrama de atividades.

Na fase de design as atividades são voltadas a identificar e modelar as classes e os *hot spots* dos frameworks identificados na análise, nesta fase também deve-se definir se esses frameworks serão *white-box*, *black-bo*x ou uma abordagem híbrida, além de documentá-los.

Por fim as fases de implementação e teste possuem as tarefas de implementar o framework em si, implementar instancias do framework e testá-las. As tarefas de cada etapa serão apresentadas nas próximas sessões.

3.1.1 Fase de Análise

A fase de análise consiste em seis tarefas, que são descritas a seguir e são representadas na Figura 17.

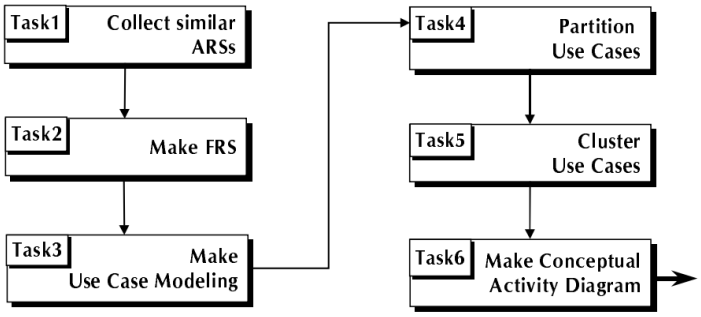

Figura 17- Tarefas da fase de análise [35]

- **Coletar *Application Requirement Specifications* (ARSs) similares**: Caso existam disponíveis especificações de requisitos de aplicações do domínio de problema, elas são reunidas, caso não estejam disponíveis ou sejam incompletas pode-se entrevistar usuários de aplicações para entender o fluxo de trabalho do domínio.
- **Criar a *Framework Requirement Specification* (FRS):** Os requisitos funcionais das ARSs selecionadas são classificados em, geral caso o requisito esteja presente em todas as ARSs ou específico caso seja um requisito particular de algumas ARS, com base nessa tarefa já é possível identificar possíveis hot spots e frozen-spots. A FRS é composta pelo nome do requisito, uma descrição e o fluxo de trabalho.
- **Criar o modelo de caso de uso:** Um modelo de casos de uso é criado com baseado nos requisitos funcionais levantados pelo FRS.
- **Particionar os casos de uso:** Os casos de uso são separados entre três camadas, a camada de *foundation* onde estão os casos de uso referentes a funcionalidades do sistema, a camada de *common business* que agrupa os casos de uso de descrevem funcionalidades que podem ser utilizadas em vários domínios e por fim, a camada de *core business* onde ficam as funcionalidades específicas do domínio de problema. A figura 18 ilustra essas camadas.

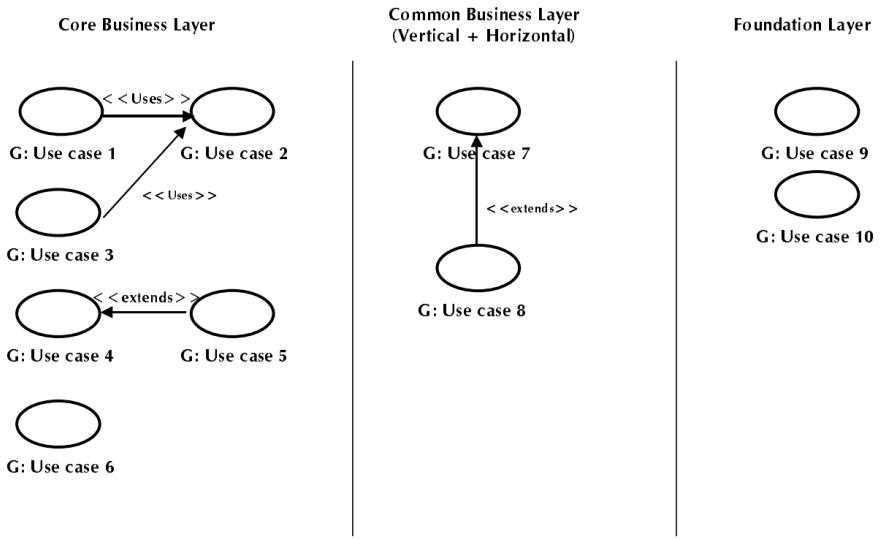

Figura 18- Particionamento de Casos de Uso [35]

- **Agrupar casos de uso:** Para cada camada, os casos de uso relacionados são agrupados de acordo com o seguinte critério: Casos de uso que tem funcionalidade independentes não são agrupados em nenhum grupo, os casos de uso que tem relacionamento "*extends*" ou "*uses*" ficam no mesmo grupo, cada grupo criado será um framework.
- **Criar um Diagrama de Atividades conceitual:** Cada caso de uso de um framework é mapeado em uma atividade, e o fluxo descrito pelo caso de uso define a sequência entre as atividades. Assim é possível descrever os fluxos de trabalho entre os casos de uso e entre os frameworks identificados, a Figura 19 mostra um exemplo.

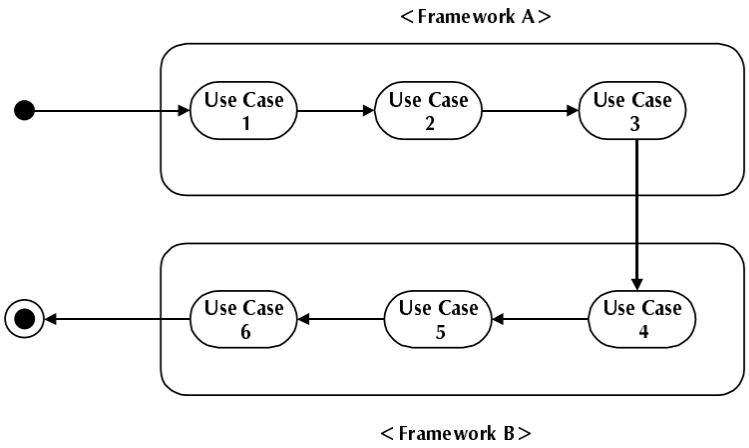

Figura 19- Diagrama de atividades conceitual [35]

3.1.2 Fase de Design

A seguir serão apresentadas as dez tarefas da fase de design, e sua sequência é ilustrada na Figura 20.

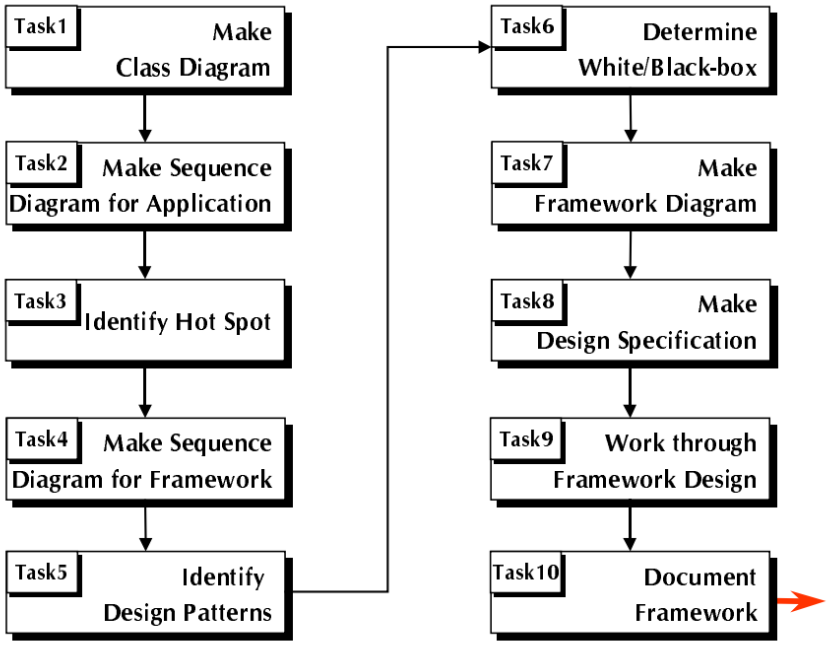

Figura 20 - Tarefas da fase de Design [35]

- **Criar diagramas de classes:** Classes, relacionamentos, atributos e operações candidatas são identificadas a partir da modelagem de diagramas de classes das ARS.
- **Criar diagramas de sequência para as ARS:** Diagramas de sequência são criados para as ARS com a finalidade de identificar possíveis operações para os diagramas de classes construídos na tarefa anterior.
- **Identificação de Hot Spots:** As classes modeladas para as ARS são analisadas em busca de atributos e operações comuns, que podem ser modelados em uma classe abstrata representando um hot spot do framework.
- **Criar um diagrama de sequência para o Framework:** Utilizando as classes abstratas encontradas na tarefa anterior como base, um diagrama de sequência é criado para descrever o fluxo das mensagens entres as classes do framework.

- **Identificar padrões de projeto aplicáveis:** O objetivo de se modelar o diagrama de sequência do framework no passo anterior é identificar a utilização de possíveis padrões de projeto.
- **Definir a abordagem White-Box ou Black-Box:** Para os hot spots identificados deve se definir se serão utilizadas abordagens baseadas em composição (*Black-Box*) ou baseadas em herança/especialização (*White-Box*).
- **Criar o Diagrama do Framework:** É criado um diagrama de classes contendo as classes, interfaces e os relacionamentos entre elas para cada framework identificado na fase de análise, também são modelados os relacionamentos entre os frameworks.
- **Criar a especificação do design:** As classes e interfaces do framework são especificadas, pode se utilizar especificações no formato de pseudocódigo.
- **Revisar o design do framework:** As tarefas dessa fase são revisadas até se atingir a consistência desejada entre os diagramas.
- **Documentação do framework:** Uma documentação formal e o manual de uso para desenvolvedores deve ser criado.

3.1.3 Fase de Implementação e Fase de Teste

As últimas duas fases são menores e suas tarefas são relativas à implementação e teste do framework. A fase de Implementação possui duas tarefas, a primeira é a implementação do framework em si e a segunda é a implementação de instancias do framework.

A fase de teste, possui também duas tarefas, uma para se executar os testes unitário das classes das instâncias criadas e outra para se executar os testes de integração entre os frameworks que elas utilizam. A Figura 21 e a Figura 22 exibem essas tarefas.

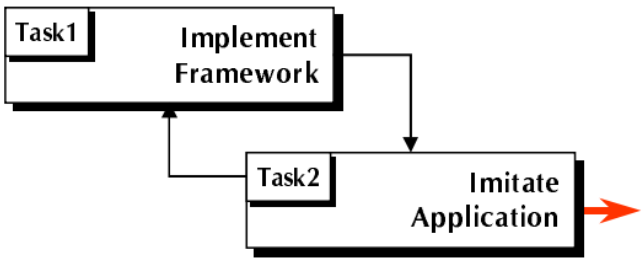

Figura 21- Tarefas da fase de implementação [35]

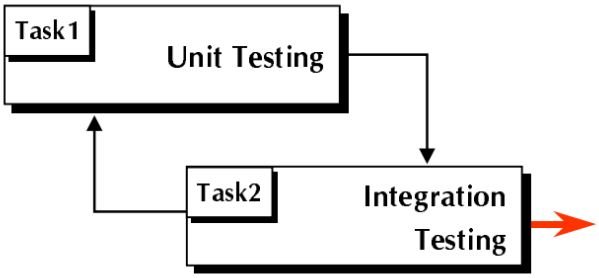

Figura 22- Tarefas da fase de teste [35]

4 REST in Safety (RiS)

Conforme comentado anteriormente, o objetivo deste trabalho é apresentar o desenvolvimento inicial de um framework que auxilie na criação de API's *RESTful* na área de segurança pública. O framework foi batizado de ***REST in Safety***, se valendo do trocadilho com a palavra REST, no sentido de o estilo arquitetural REST sendo aplicado no domínio de segurança pública, quanto no sentido de "Descanse em segurança" que é a tradução literal do inglês para a frase "*REST in Safety*", para facilitar a apresentação do projeto, o nome do framework será abreviado para **RiS**.

O **RiS** é um framework de aplicação orientado a objetos, que utilizará uma arquitetura do estilo REST para criar serviços na internet, que poderão ser utilizados em sistemas orientados a serviços. Nesta seção será apresentado os principais artefatos, as decisões de projeto e de implementação da versão inicial do framework **RiS**.

4.1 Fase de Análise

Conforme abordado anteriormente na seção 3.1.1 a fase de análise se preocupa em identificar as funcionalidades e os frameworks envolvidos no fluxo de trabalho. Por isso o artefato mais importante dessa etapa é o diagrama de casos de uso do framework, além de representar as funcionalidades disponíveis, é a partir dele que os casos de uso serão particionados nas camadas de *foundation*, *common business* e *core business*. E são esses os casos de uso que serão mapeados no diagrama de atividades, com a finalidade de entender a interação entre esses frameworks.

Na Figura 23 está representado o diagrama de casos de uso do **RiS**, ele foi construído de acordo com o ponto de vista do ator **Serviço** que representa os serviços expostos pela API. O

seu principal caso de uso é o do tratamento da requisição, quando um serviço recebe uma requisição ele precisa autenticá-la, identificar o recurso que está sendo requisitado, executar o processamento correto para aquele recurso e devolver uma representação como resposta. Todos os outros casos de uso se iniciam a partir do tratamento da requisição, pois eles representam as funcionalidades disponibilizadas pelo framework para o processamento dos recursos.

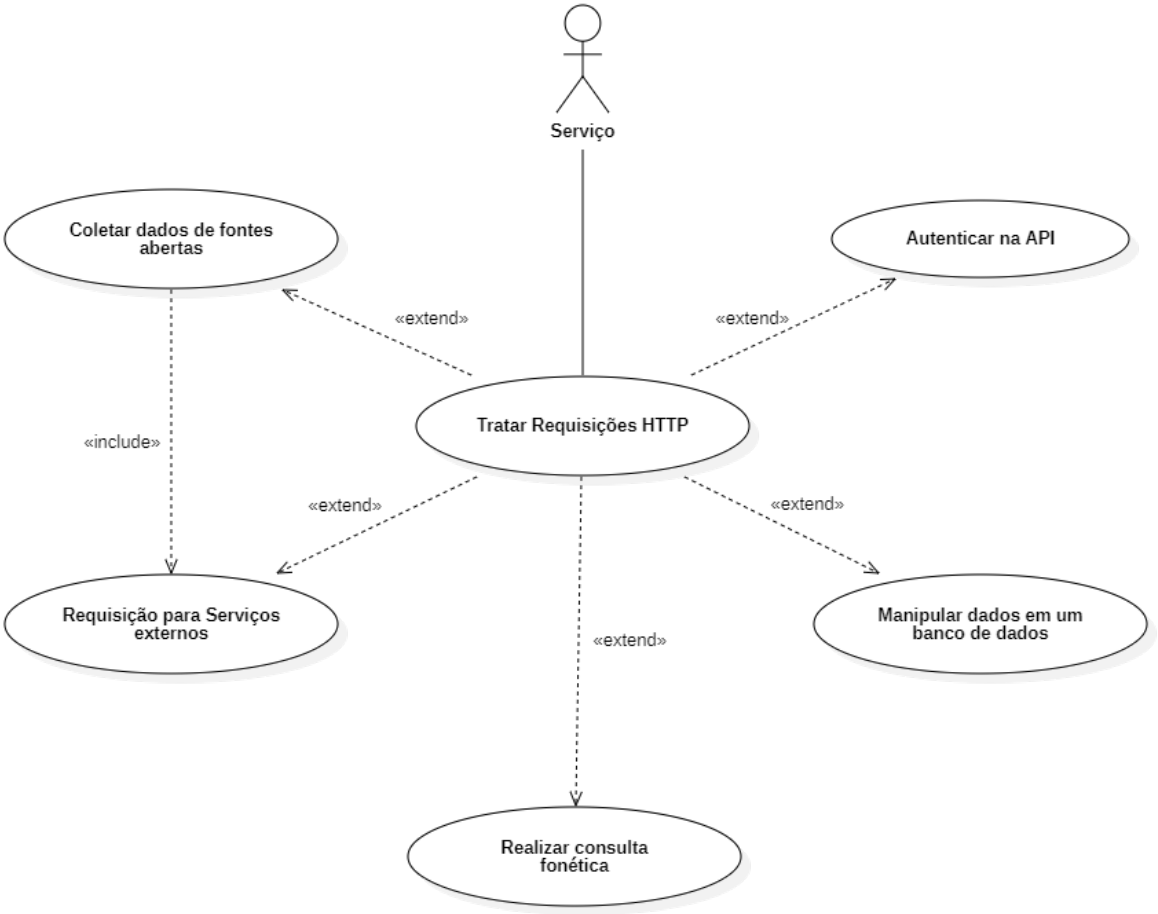

Figura 23 - Diagrama de casos de uso do RiS

Esses casos de uso foram extraídos dos casos de uso e descrições das funcionalidades apresentados na seção 2.5, algumas funcionalidades genéricas foram incluídas, pois elas são parte de muitos dos casos de uso descritos por (FURTADO, 2002), como por exemplo o acesso a banco de dados e a integração com sistemas externos. Outras são mais específicas e utilizam essas outras funcionalidades, como por exemplo a coleta de dados em fontes abertas. A descrição completa dos casos de uso do RiS é apresentada na Tabela 9.

Um ponto que precisa ser destacado é sobre a escolha da integração com os serviços Infosimples [36] e com a BrasilAPI [37], embora o governo brasileiro disponibilize um catálogo

de APIs Governamentais [38], uma grande parte delas só pode ser acessada por órgãos e organizações com autorização prévia, além de que algumas delas são pagas e exigem além da autorização a contração do acesso com o Serviço Federal de Processamento de Dados (SERPRO).

O Infosimples é um serviço pago que utiliza uma automação em sites governamentais e outras fontes públicas para coletar dados, os dados coletados são fornecidos através uma camada de *web services* e uma API *RESTful* [38], por isso que ele foi escolhido como alternativa para se consultar dados abertos. Já o Brasil API é um agregador de serviços públicos, como consultas nos dados do IBGE, dados sobre um CNPJ, sobre um CEP, a cidade a qual se refere um DDD entre outros.

Tabela 9 - Descrição dos casos de uso do RiS

Nº	Caso de Uso	Ator	Descrição
1	Tratar requisições HTTP	Serviço	Ao receber uma requisição o serviço precisa tratá-la, de maneira que ela seja autenticada, isto é, o *token* de acesso fornecido na requisição precisa ser de um usuário válido e com acesso a API. Caso seja uma requisição válida, a requisição é repassada para o recurso solicitado na URL, e após o recurso ser processado, uma representação é devolvida como resposta no formato JSON.
2	Autenticar na API	Serviço	O processo de autenticação recebe as credenciais de um usuário e as valida utilizando arquivos criptografados, caso a validação seja bem-sucedida um *token* de acesso é gerado e atribuído a esse usuário.
3	Manipular dados em uma tabela do BD	Serviço	O RiS fornecerá suporte a utilização de bancos de dados, com funcionalidades básicas como gerenciar a conexão com o banco de dados, executar *queries* SQL, e com funcionalidades específicas como mapear tabelas em recursos da API, evitando que código repetitivo seja escrito ao se criar serviços que

			apenas acessam ou modificam uma tabela do banco de dados.
4	Requisição para serviços externos	Serviço	O RiS fornecerá funcionalidades para facilitar o consumo de *web services SOAP* e outras APIs *RESTful*.
5	Coletar dados de fontes abertas	Serviço	O RiS terá funcionalidades pré-implementadas para acessar dados externos fornecidos pela API RESTful BrasilAPI e pelo serviço pago Infosimples.
6	Realizar consulta fonética	Serviço	A funcionalidade de consulta fonética, permite buscar um dado em um conjunto de dados sem se ter certeza se o dado de entrada está escrito na forma correta.

Com os casos de uso definidos, podemos dividi-los nas camadas de *foundation*, *common business* e *core business*, essa divisão é apresentada na Figura 24.

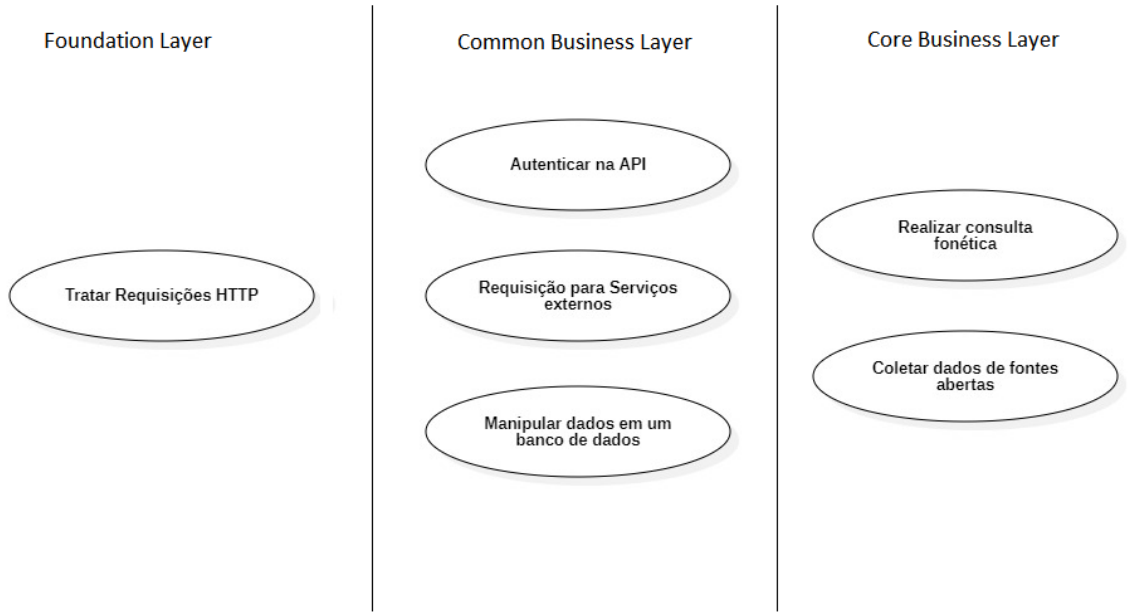

Figura 24 - Particionamento dos casos de uso do RiS

Conforme mostra a Figura 24 nenhum caso de uso com relacionamentos do tipo *include* ou *extend* foram adicionados na mesma camada, por isso não foi encontrado novos frameworks nessa etapa.

A partir desses casos de uso também foi construído o diagrama de atividades para demonstrar o fluxo de trabalho entre eles, o diagrama é exibido na Figura 25. Embora seja bastante simples, ele nos permite entender qual o fluxo e a relação entre as funcionalidades disponibilizadas pelo RiS.

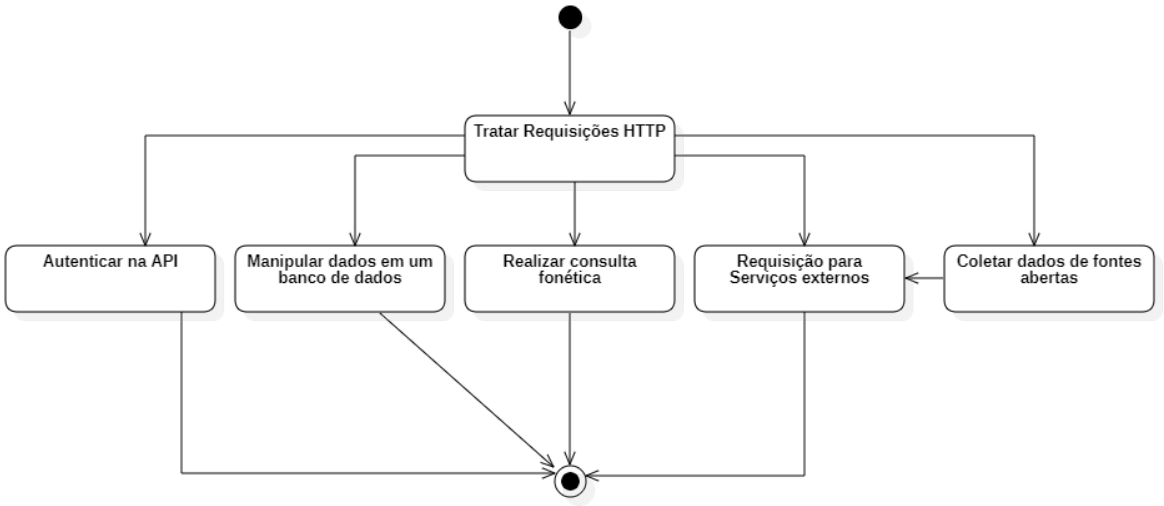

Figura 25 - Diagrama de Atividades do RiS

Estes são os principais artefatos produzidos durante a etapa de análise do RiS, esses artefatos ajudam a delimitar o escopo do framework e apresentar conceitualmente as funcionalidades e interações entre elas. Baseado nessas funcionalidades e nesse escopo a próxima seção abordará o design do framework.

4.2 Fase de Design

Esta etapa tem como objetivos identificar e projetar as classes, operações, *hot-spots* do framework e identificar também oportunidades de aplicação de padrões de projeto nesta estrutura. Os principais artefatos dessa fase são o diagrama de classes e os diagramas de sequência do RiS, eles serão fundamentais não só para a etapa de implementação, mas para entender como os usuários do framework irão estender e implementar as funcionalidades disponíveis.

Na figura 26 é apresentado o diagrama de sequência geral do RiS, nele é possível ver o fluxo geral de como uma requisição é tratada pelos componentes do framework, o fluxo se inicia com uma requisição HTTP da aplicação cliente que será tratada pela classe controladora

do RiS, que é a ***RequestProcessor***, incialmente ela irá identificar o recurso requisitado e irá validar o token de acesso com auxílio da classe ***RequestValidator***, com o recurso identificado e a requisição validada, a ***RequestProcessor*** carrega o recurso através de uma classe que estende a classe abstrata **Resource.**

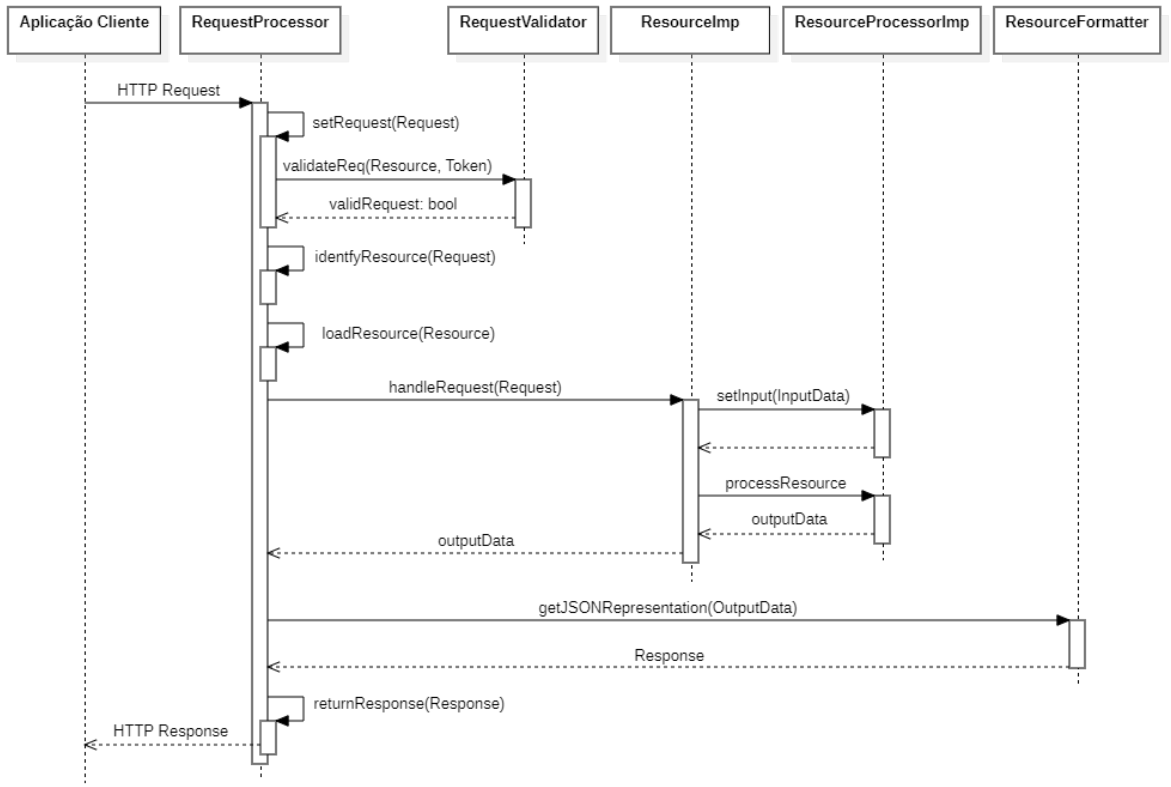

Figura 26 - Diagrama de sequência do RiS

Essa classe filha da classe *Resource*, no diagrama chamada de *ResourceImp* deve ser implementada pela aplicação para representar um recurso exposto na API, a implementação deve sobrescrever os métodos *handlePOST*, *handleGET*, *handlePUT* e *handleDELETE*, cada método será responsável por tratar uma operação HTTP suportada pelo recurso, a implementação precisa apenas sobrescrever os métodos correspondentes às operações que se deseja suportar. Observe que no diagrama exibido na Figura 26, optou-se por utilizar um método genérico *handleRequest* em vez e repetir as chamadas para cada método, pois o fluxo a partir deles é o mesmo.

Outro ponto importante do fluxo é a classe abstrata *ResourceProcessor*, que é a abstração utilizada para encapsular a lógica executada ao se manipular recursos, como por exemplo o acesso a banco de dados, a autenticação na API, a busca fonética etc. essas funcionalidades

serão implementadas através de classes filhas da *ResourceProcessor*, que no diagrama são representadas pela classe *ResourceProcessorImp*.

A classe *ResourceProcessor* fornece o método *setInput* que recebe um objeto do tipo *InputData* e o adiciona a um atributo interno, e o método abstrato *processResource* que deve ser sobrescrito para implementar a lógica da funcionalidade desejada, esse método retorna um objeto de do tipo *OutputData.*

Ao final do fluxo um objeto do tipo *OutputData* é retornado, ou caso alguma exceção ocorra em algum ponto, ela deve ser lançada até a classe *RequestProcessor* que com auxílio da classe *ResourceFormatter* que irá gerar uma representação no formato JSON do objeto *OutputData* obtido ou do erro de uma exceção lançada. Essa representação é então retornada como resposta da requisição recebida.

O design das classes *Resource* e *ResourceProcessor* são baseados no padrão de projeto *Template Method* [15], a maior vantagem nessa abordagem é que podemos definir uma interface uniforme para objetos semelhantes e assim manter um baixo acoplamento entre recursos e funcionalidades, além de facilitar o design de novos recursos e novas funcionalidades, como exemplo a Figura 27 exibe o diagrama de sequência da funcionalidade de autenticação na API.

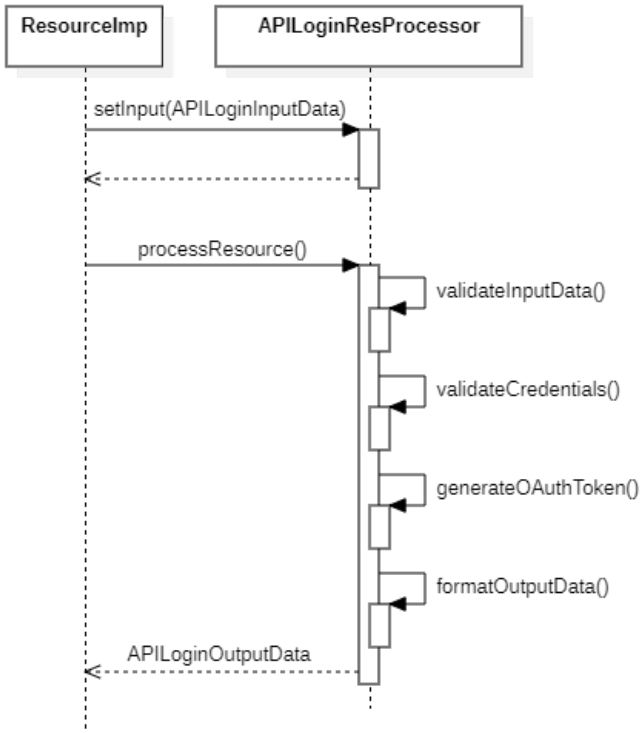

Figura 27- Diagrama de Sequência do login na API

A Figura 28, exibe o diagrama de sequência da funcionalidade de acesso a sistemas externos, essa funcionalidade é interessante para exemplificar a situação em que implementações de *ResourceProcessor* são combinadas para executar alguma outra lógica. Observe que nesta situação a classe *ExternalSystemResProcessor*, se comporta como um *Mediator* [15].

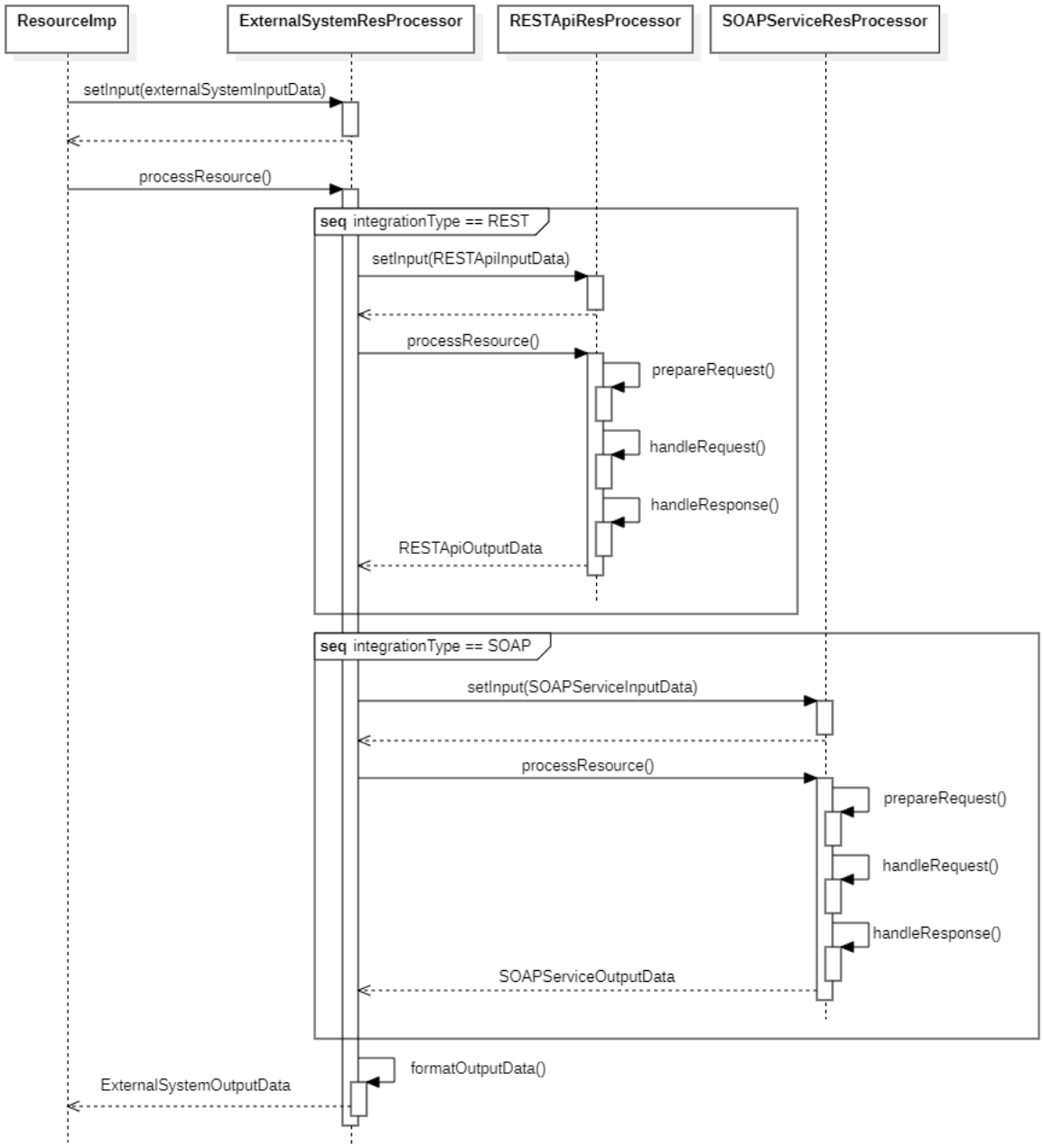

Figura 28 - Diagrama de sequência do acesso a sistemas externos

O design da classe *ExternalSystemResProcessor* pode ser reaproveitada no design de outras funcionalidades, sem a necessidade conhecer os detalhes de sua implementação, como por exemplo, quais outros *ResourceProcessor* ela utiliza em sua lógica, ou quais outros métodos ele possui além do *processResource*. Um exemplo disso é apresentado na Figura 29, que exibe a funcionalidade para coletar dados de fontes abertas.

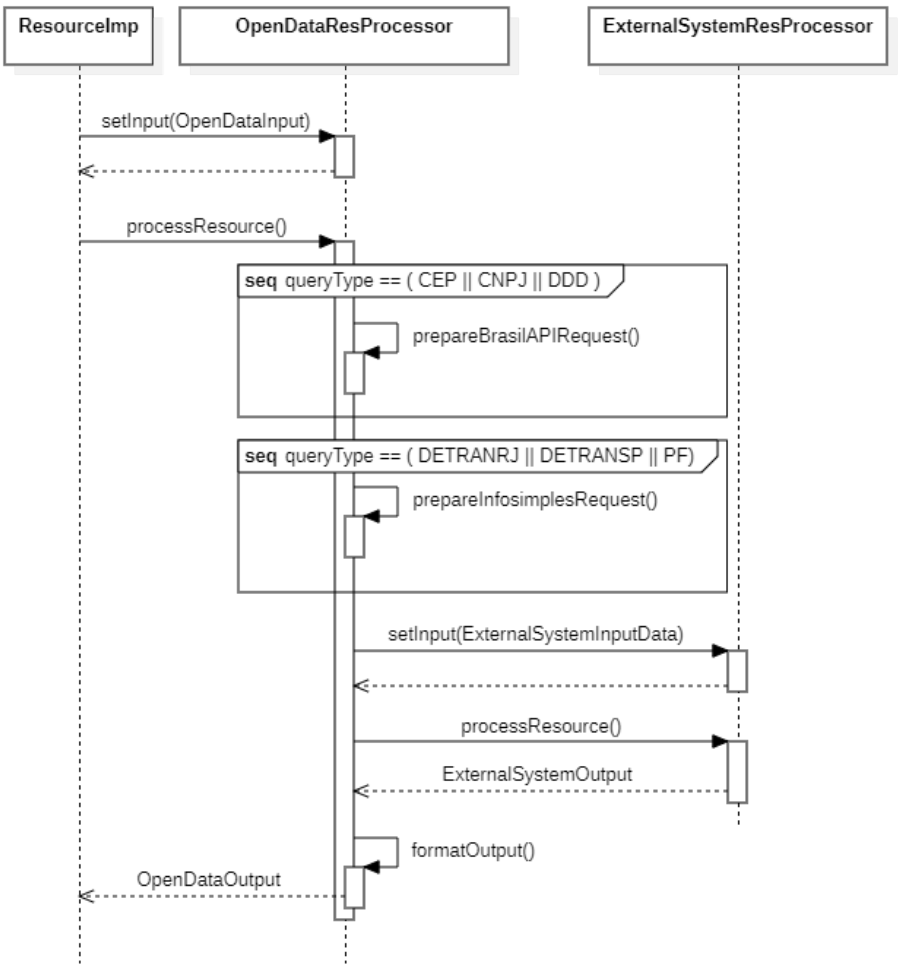

Figura 29 - Diagrama de Sequência da coleta de dados em fontes abertas

As demais funcionalidades de realizar acesso a bancos de dados e consulta fonética, tem seus diagramas de sequências apresentados nas Figuras 30 e 31 respectivamente. Um ponto a se destacar na Figura 30 é a utilização do padrão de projeto *Strategy* [15] para gerenciar as conexões do banco de dados através de subclasses. Essa abordagem permite a instância do framework especificar facilmente quais conexões irá utilizar através de subclasses da classe abstrata *DBConnector*, o objeto da classe que implementa a estratégia da conexão é informado em um atributo do objeto *DatabaseInputData*.

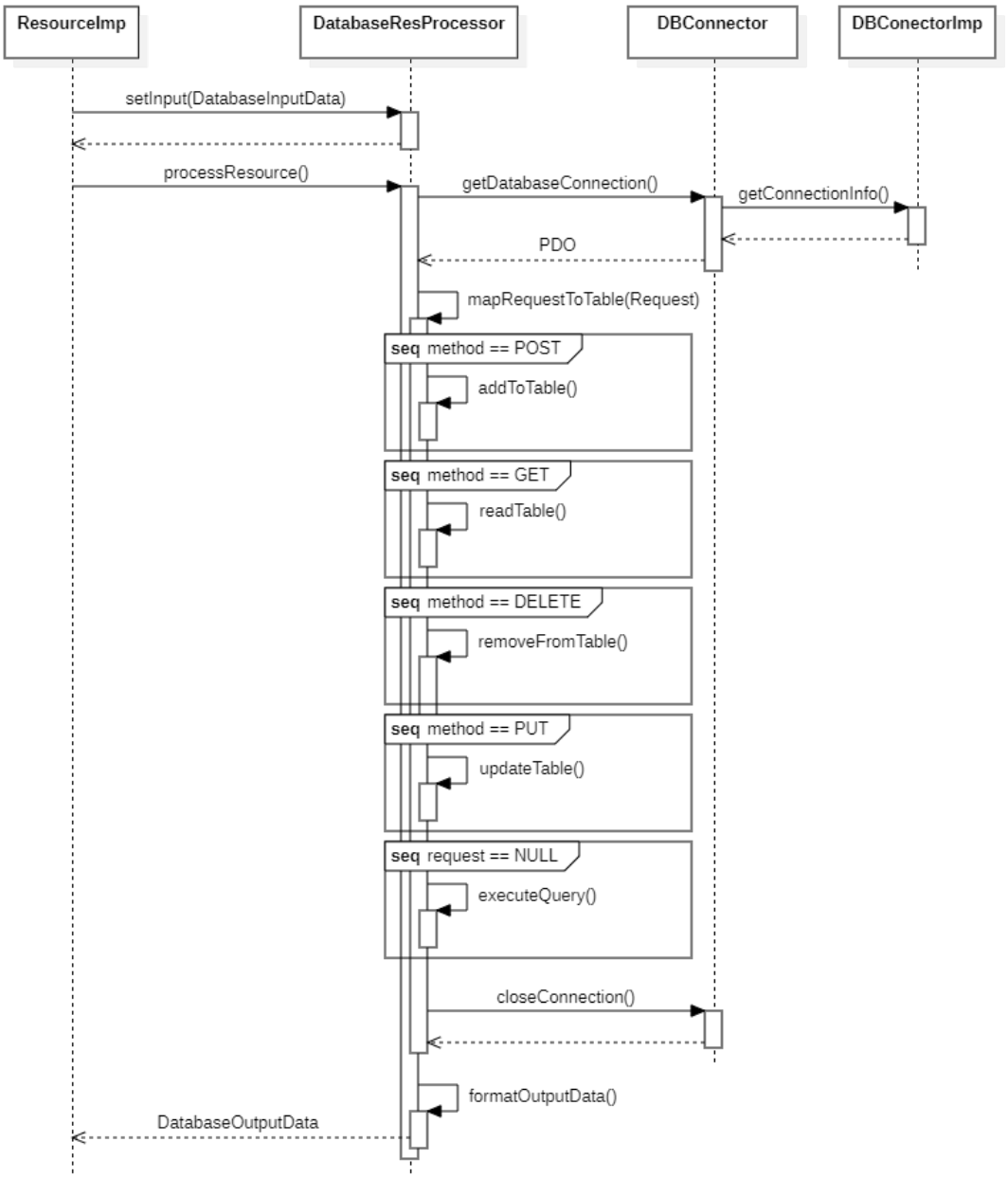

Figura 30 - Diagrama de sequência do acesso ao banco de dados

Os principais *hot-spots* do RiS são as classes *Resource* e *ResourceProcesor* conforme comentado anteriormente, a aplicação deve implementar subclasses dessas classes abstratas, deve-se implementar ao menos uma subclasse de *Resource* para existir uma instância do RiS. Como essa personalização é feita através herança, esses host-spots são do tipo *white-box*. Os outros *hot-spots white-box* são a classe que implementa a estratégia de conexão com algum banco de dados e as interfaces *InputData* e *OutputData* que são estendidas para padronizar os tipos de dados entre os objetos *ResourceProcessor*.

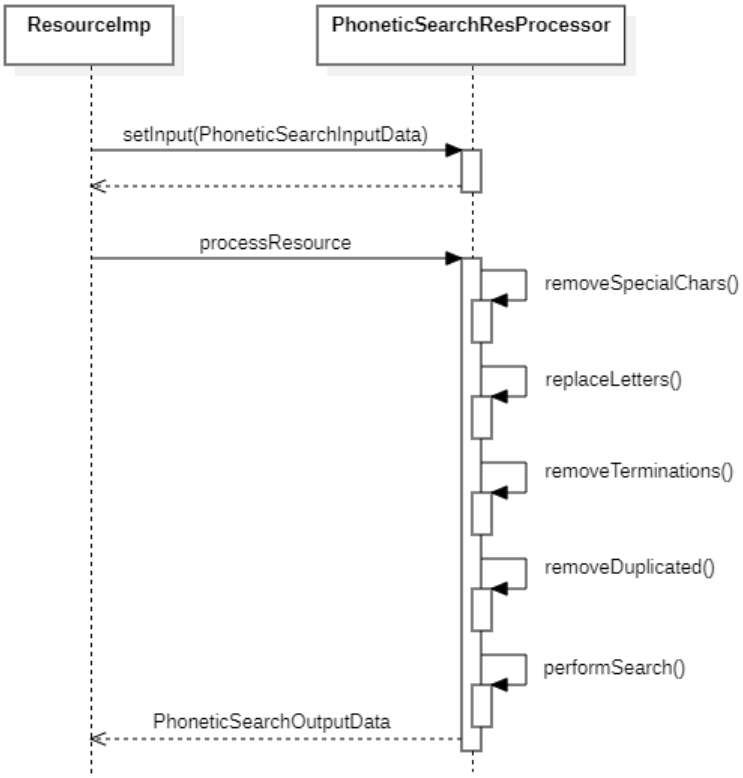

Figura 31 - Diagrama de sequência da busca fonética

Já os *hot-spots* do tipo *black-box* são as subclasses de *ResourceProcessor* que foram descritas nos diagramas de sequência apresentados nessa seção, essas subclasses são fornecidas com o RiS e permitem que o usuário do framework as combine para gerar a representação desejada de um recurso que esteja implementando, ou ainda podem ser combinadas dentro da implementação de um novo *ResourceProcessor,* como foi feito na classe *ExternalSystemResProcessor* apresentada no diagrama da Figura 28, essa classe combina outros dois *ResourceProcessors*, para realizar seu processamento.

A versão final do diagrama de classes da primeira versão do RiS, é apresentado na Figura 32, nele são apresentadas todas as classes e seus respectivos relacionamentos, na Figura 33 é possível identificar as camadas a qual cada classe pertence. Perceba que todos *hot-spots* da camada *foundation* são do tipo *white-box*, já a camada *common business* é mista, pois possui vários *ResourceProcessors* e a estratégia de acesso a bancos de dados, a camada *core business* é composta apenas por *hot-spots* do tipo *black-box*.

Esses foram os principais artefatos e decisões de projeto durante a fase de design, a próxima seção discutirá as decisões de implementação e as instâncias que serão criadas para validação do framework.

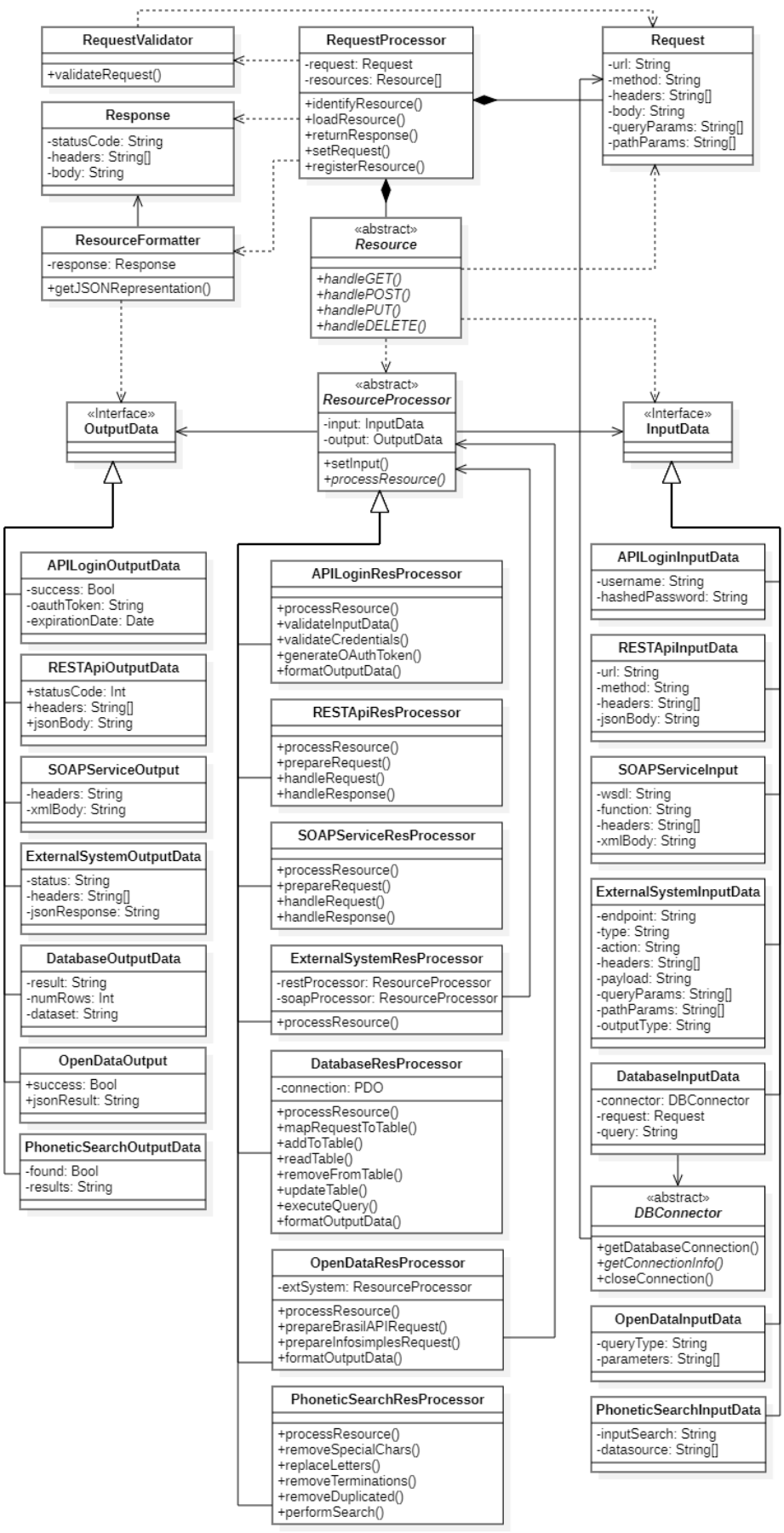

Figura 32- Diagrama de classes do RiS

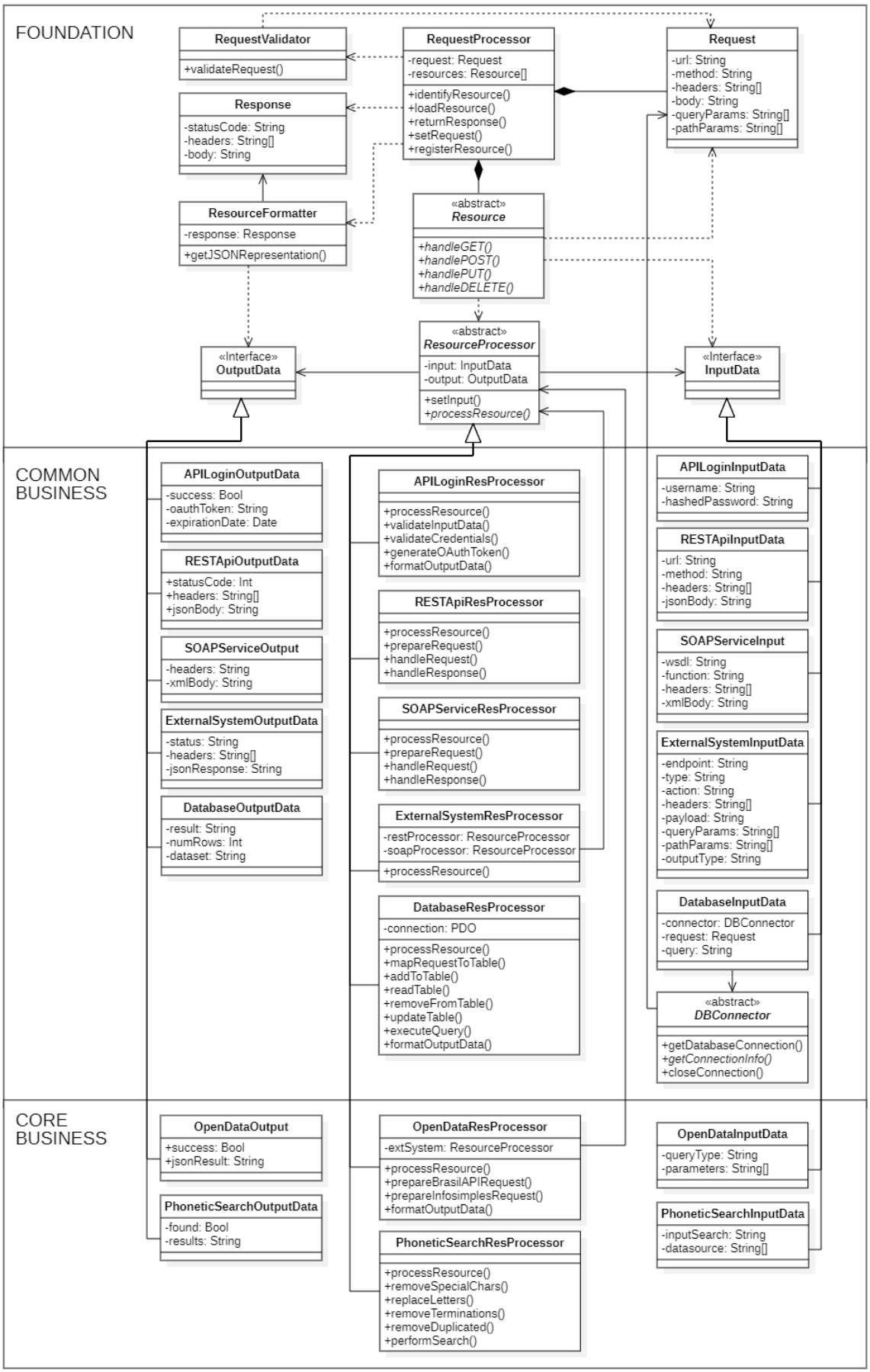

Figura 33- Diagrama de classes do RiS com camadas

4.3 Fase de Implementação

Nesta seção, serão discutidas as principais decisões sobre a implementação do RiS, a estrutura de diretórios, arquivos configuração e o design das aplicações que serão construídas para validação do framework.

A principal decisão de implementação foi em relação a linguagem, o PHP foi escolhido por diversas razões, algumas delas são a simplicidade da linguagem e dos recursos necessários para executar o interpretador da linguagem em um servidor web, a flexibilidade fornecida pela linguagem ao se trabalhar com tipos, os recursos nativos oferecidos como por exemplo, acesso facilitado e uniforme a bancos de dados através do *PHP Data Object* (PDO), as interfaces para manipulação de requisições HTTP e consumir *web services* SOAP, a facilidade de trabalhar com JSON, XML e arquivos texto, entre outras.

Um ponto importante do funcionamento do framework relacionado a estrutura de implementação, é que o tratamento de requisições se baseia no mecanismo de reescrita de URL disponibilizado pela maioria dos servidores web, esse mecanismo permite o servidor web interceptar a URL de uma requisição e reescrevê-la redirecionando a requisição.

No caso do RiS, essa reescrita é utilizada para redirecionar todas as requisições para o script principal da aplicação, um arquivo chamado *index.php*, durante a reescrita a URL original é enviada para esse script como um parâmetro, por padrão o RiS apenas fornece um arquivo *.htaccess* para habilitar o módulo de reescrita de URL no servidor web apache, e um arquivo *server.php* que pode ser usado para a mesma função com o servidor que acompanha o interpretador da linguagem PHP (ao se referir a esse servidor, será utilizado o nome *cli-server)*.

Na estrutura fornecida pelo RiS, existem também dois arquivos de configuração específicos para a aplicação, o mais importante é o *routes.php*, esse arquivo centraliza os registros de rotas da aplicação, os recursos implementados são acessados através de um identificador, que é a URL da requisição, ao registrar uma rota, a aplicação está vinculando qual é a URL que determinado *Resource* irá responder, além de definir os parâmetros da URL (parâmetros de path) que serão aceitos pelo *Resource*, a Figura 34 mostra um exemplo do registro de rotas de uma aplicação do RiS.

O outro arquivo de configuração disponível para a aplicação é o *config.json*, ele possui uma seção de metadados para informações como nome da API, versão, desenvolvedor, URL da documentação, e outra seção de parâmetros de configuração, para configurar comportamentos como habilitar o modo de depuração, tamanho máximo de upload, tamanho máximo do corpo

de uma requisição POST, tempo de *timeout* da requisição e quantidade máxima de memória disponível. A Figura 35 exibe um exemplo de *config.json* de uma aplicação do RiS.

routes.php

```php
<?php
namespace app;

use app\resources\LoginResource;
use app\resources\SinespSearchResource;
use ris\foundation\RequestProcessor;


// https://api.example.com/login
RequestProcessor::registerResource("login", LoginResource::class);

// https://api.example.com/busca-sinesp-veiculos/{placa}
RequestProcessor::registerResource("busca-sinesp-veiculos/{placa}", SinespSearchResource::class);

// https://api.example.com/busca-sinesp-mandados/{cpf}
RequestProcessor::registerResource("busca-sinesp-mandados/{cpf}", SinespSearchResource::class);

// https://api.example.com/busca-sinesp-procurados/{cpf}
RequestProcessor::registerResource("busca-sinesp-procurados/{cpf}", SinespSearchResource::class);

// https://api.example.com/busca-sinesp-desparecidos?nome={nome}
RequestProcessor::registerResource("busca-sinesp-desparecidos?nome={nome}", SinespSearchResource::class);


?>
```

Figura 34 - exemplo de um arquivo de rotas do RiS

Todos os parâmetros do arquivo de config.json são opcionais, caso a seção de metadados esteja vazia, a API será nomeada "*MyAPI*" e sua versão será definida como "1.0", e caso a seção de parâmetros de configuração esteja vazia, os parâmetros não serão definidos e o interpretador do PHP irá operar com seus valores configurados.

routes.php | config.json

```json
{
    "metadata": {
        "apiName": "BuscaSINESP",
        "version": "1.0",
        "dev": "Paulo C. Rocha",
        "docs": "https://api.example.com/docs/"
    },
    "config": {
        "debug": true,
        "maxUploadSize": "15M",
        "postMaxSize": "8M",
        "memoryLimit": "250M",
        "maxExecutionTime": "60"
    }
}
```

Figura 35 - JSON de configuração do RiS

Um esclarecimento se faz necessário em relação ao porquê este arquivo não ser um script *php* como os demais arquivos de configuração, e os motivos para essa escolha é que o formato JSON é leve, de fácil manipulação e cumpre bem a função de parametrização, mas o mais importante é que um dos objetivos de arquivos de configuração é parametrizar variáveis específicas de ambientes, como por exemplo manter a depuração ativa em ambientes de desenvolvimento e homologação, e desativada em ambientes de produção, e o formato JSON é muito amigável com ferramentas de configuração remota e entrega contínua.

A estrutura de diretórios do RiS foi projetada para facilitar a organização do código e facilitar a identificação e carregamento de classes em tempo de execução, no primeiro nível que será chamado de diretório raiz, estão os arquivos *index.php, server.php* e o *.htaccess* que foram comentados anteriormente, além destes arquivos estão dois diretórios, um deles chamado de *app* que contém todo o código da aplicação e o diretório *ris*, que contém o código do framework.

Essa separação tem o objetivo de isolar o código da aplicação e o código do framework, e com isso facilitar atualizações de forma independente, a Figura 36 exibe a estrutura com todos os diretórios do RiS e os cinco arquivos de configuração *index.php, server.php, config.json, .htaccess* e *routes.php*.

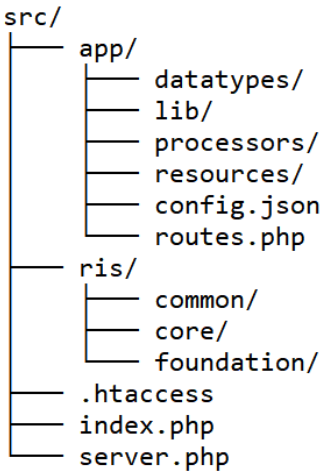

Figura 36 - Estrutura de diretórios do RiS

Dentro do diretório *ris* o código do framework está estruturado em três outros diretórios, são eles *foundation*, *common* e *core*, as classes de cada camada ficam dentro de seu respectivo diretório, a escolha por essa estrutura tem um único objetivo, que é permitir o versionamento e atualizações por camada do framework.

No diretório *app*, existe uma estrutura pré-definida de quatro diretórios, que são *resources, processors, datatypes* e *lib*. Os diretórios *resources* e *processors*, como o nome sugere são reservados para as implementações da classe *Resource* e as implementações da classe *ResourceProcessor* respectivamente, o diretório datatypes é reservado para as implementações das interfaces *InputData* e *OutputData* o diretório *lib* opcional é reservado para bibliotecas e outras classes utilizadas pela aplicação, é também dentro do diretório *app* que ficam os arquivos *routes.php* e *config.json*

Já nesse caso a estrutura de diretórios fornecida pelo RiS tem o principal objetivo de facilitar identificação de classes e melhorar a performance da aplicação, embora não seja uma estrutura obrigatória, é fortemente sugerido que a aplicação se estruture dessa forma. O RiS tentará identificar classes da aplicação no diretório app, independente de qual subdiretório ela esteja, mas a busca se inicia pelos diretórios pré-definidos. Por esse mesmo motivo a estrutura dentro dos diretórios *resources, processors, datatypes* e *lib* é livre para a aplicação estruturar da maneira que desejar.

4.3.1 API de cadastro de cidadãos e ocorrências

Já que discutimos como será implementado o RiS, apresentaremos agora o design de duas aplicações simples para validar a versão inicial do framework. A primeira aplicação construída sob o RiS pretende explorar as funcionalidades de banco de dados e busca fonética, os casos de uso dessa API são apresentados na Figura 37.

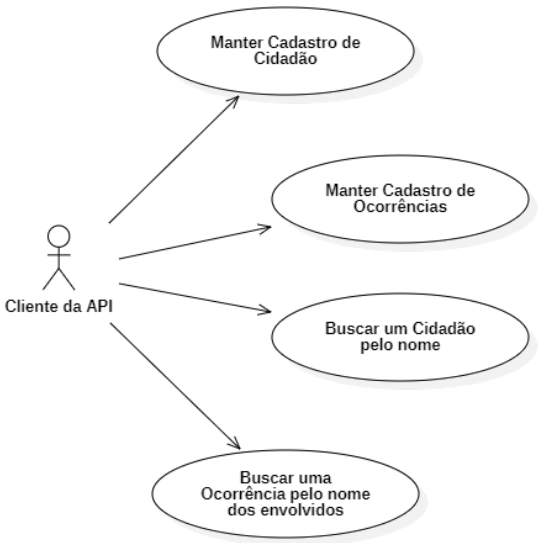

Figura 37 - Casos de uso da API de cadastro de cidadãos e ocorrências

As funcionalidades dessa API estão relacionadas a manter registros em bancos de dados e em como buscá-los mesmo se ter certeza da grafia correta de um nome, o diagrama de classes dessa aplicação é exibido na Figura 38, nele é possível ver além das classes que serão implementadas nessa aplicação a separação entre as classes do RiS e as classes da API.

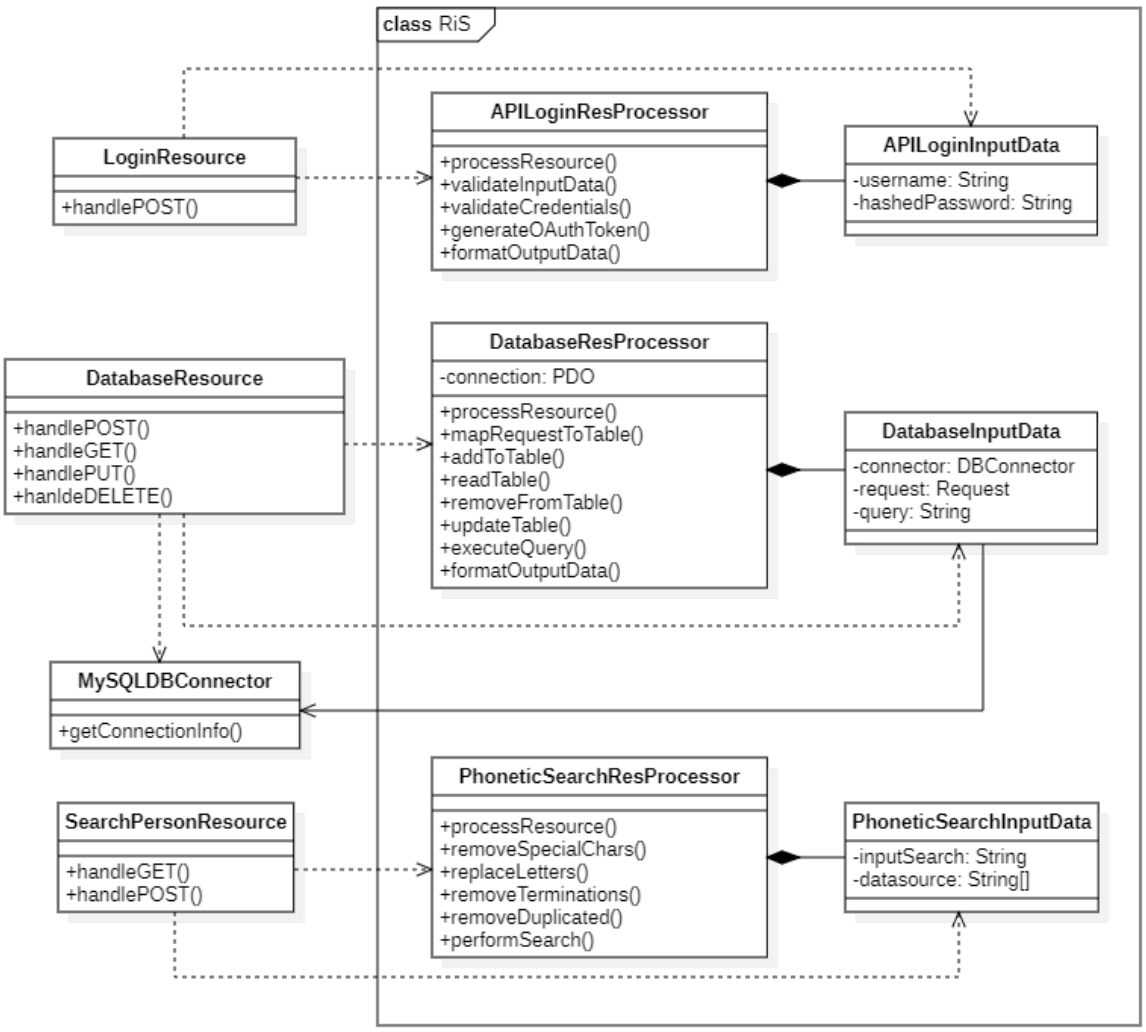

Figura 38 - Diagrama de classes da API de cadastro de cidadãos e ocorrências

Para essa API foram necessárias apenas implementações de *Resources*, os *ResourceProcessors* usados nessa aplicação foram todos fornecidos pelo RiS, a API implementada disponibilizará cinco serviços, são eles:

- O Serviço de Login que recebe requisições POST na URL: “/login”

- O Serviço para cadastros dos cidadãos no banco de dados, que recebe requisições GET, POST, PUT e DELETE na URL: "*/cidadaos/<cidadaoID>"*

- O Serviço para cadastro das ocorrências no banco de dados, e recebe requisições GET, POST, PUT e DELETE na URL: "*/ocorrencias/<ocorrenciaID>*"
- O Serviço para buscar cidadãos pelo nome que recebe requisições GET e POST na URL: *"/busca-cidadaos*"
- O Serviço para buscar cidadãos pelo nome dos envolvidos que recebe requisições GET e POST na URL: "*/busca-ocorrencias*"

4.3.2 API de busca no SINESP

A segunda aplicação desenvolvida com o RiS explora as funcionalidades de acesso à sistemas externos para coletar dados, é uma API bem simples que visa consultar dados no banco de dados do SINESP, os casos de uso de API são exibidos na Figura 39.

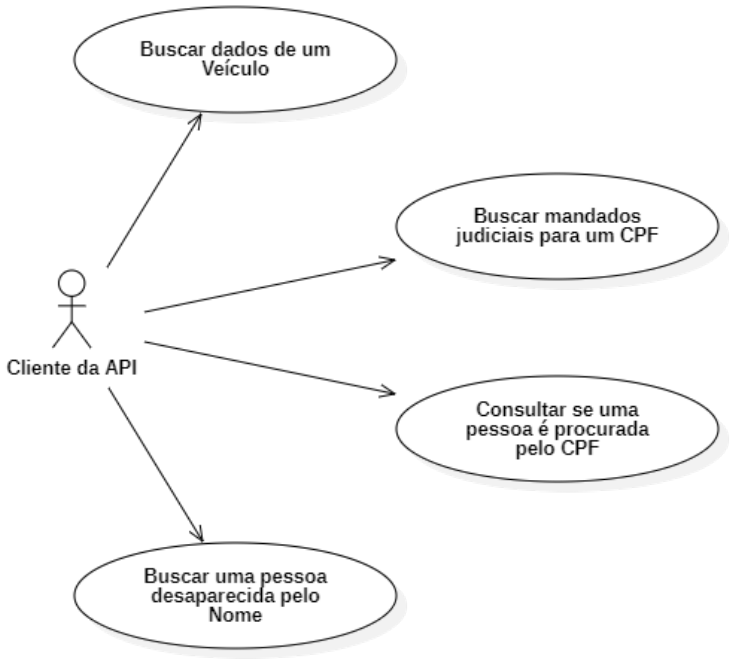

Figura 39 - Casos de uso da API de busca no SINESP

Essas consultas foram baseadas nas consultas disponíveis pelo aplicativo SINESP Cidadão [40], e o acesso a esses dados é feito pela API do Infosimples, órgãos de segurança pública com

acesso ao SINESP podem ter os mesmos dados acessando diretamente os serviços do SINESP com suas credenciais.

As classes que serão implementadas para essa aplicação são apresentadas na Figura 40, também se optou por exibir a separação das classes do framework e da aplicação para facilitar o entendimento. Outro ponto que vale a pena ser destacado é que ao contrário da aplicação anterior nesse caso houve a implementação de um *ResourceProcessor* que utiliza o *OpenDataResProcessor* para acessar os dados do Infosimples.

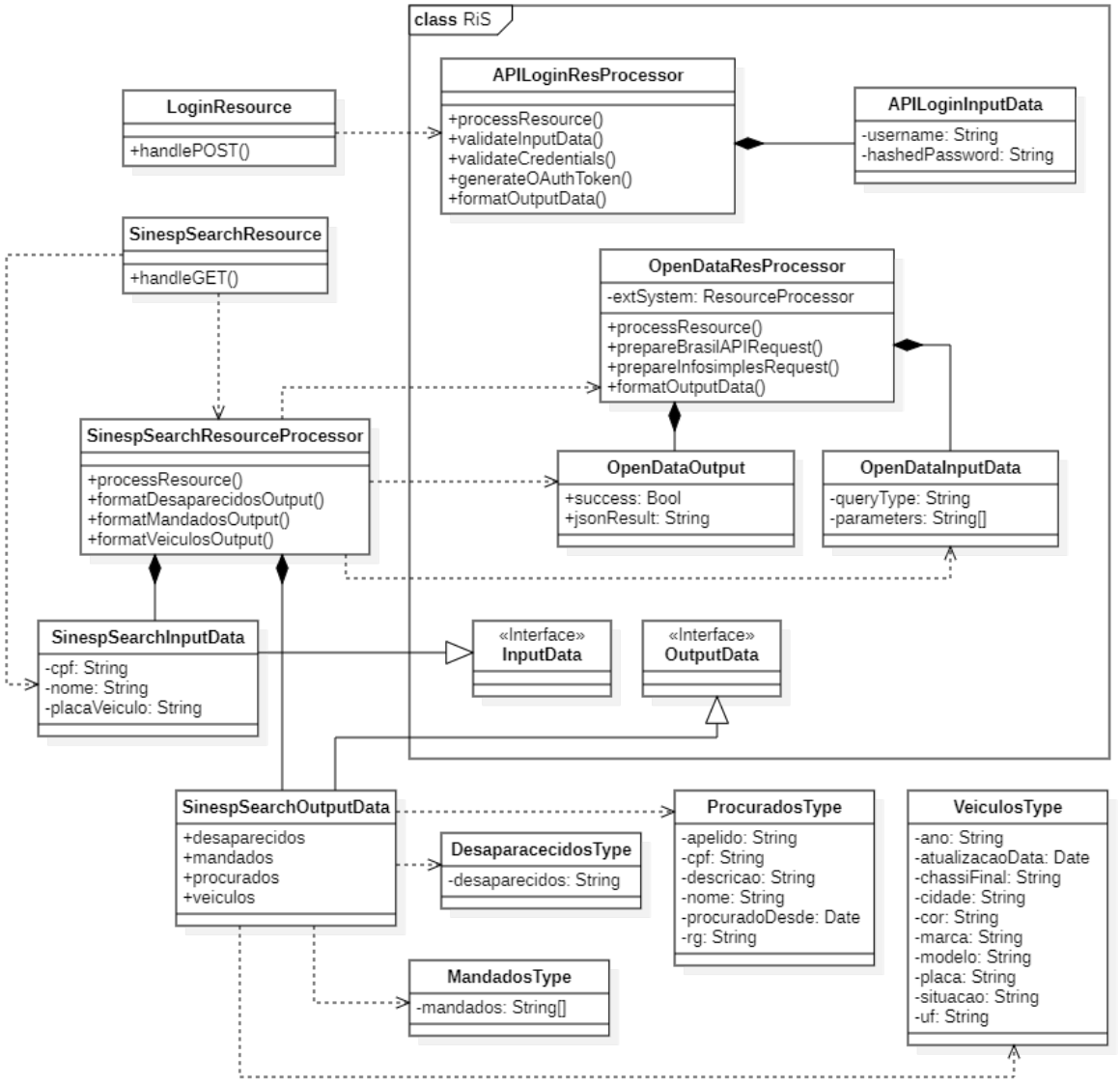

Figura 40 - Diagrama de classes da API de busca no SINESP

Embora tenha sido implementado apenas um *Resource* além do *LoginResource*, essa API fornece cinco URLs para acessar seus serviços, todas essas rotas são direcionadas para o *SinespSearchResource*, os serviços e URLs disponíveis são:

- O Serviço de Login que recebe requisições POST na URL: "*/login*"
- O Serviço de buscar veículos no SINESP pela placa que apenas recebe requisições GET na URL: "*/busca-sinesp-veiculos/<placa>*"
- O Serviço de buscar mandados judiciais no SINESP pelo CPF da pessoa que apenas recebe requisições GET na URL: "*/busca-sinesp-mandados/<cpf>*"
- O Serviço de buscar procurados pela justiça no SINESP pelo CPF da pessoa que apenas recebe requisições GET na URL: "*/busca-sinesp-procurados/<cpf>*"
- O Serviço de buscar desaparecidos no SINESP pelo nome que apenas recebe requisições GET na URL: "*/busca-sinesp-desparecidos*"

5 CONCLUSÃO E TRABALHOS FUTUROS

No início do trabalho foi definido como objetivo projetar e implementar um framework com uma arquitetura reutilizável, e fornecer meios para encapsular funcionalidades utilizadas nos sistemas que suportam o domínio da segurança pública para construir serviços.

Embora nesta primeira versão do framework, poucas funcionalidades específicas do domínio da segurança pública foram implementadas, a arquitetura projetada se mostrou flexível e simples de utilizar, as abstrações de *Resource* e *ResourceProcessor* permitem a aplicação delegar ao framework a arquitetura do serviço e o processamento da requisição e focar somente na lógica e no recurso exposto pela API.

Além da inversão de controle obtida por essas abstrações, elas também permitiram desacoplar a definição do serviço de sua lógica, e é sob esse mecanismo que as espera-se encapsular o conhecimento de domínio que foi capturado e disponibilizá-lo para reutilização em novos recursos.

Outra vantagem observada na estrutura projetada é que ela é expansível, as funcionalidades do domínio que não foram implementadas nessa primeira versão podem ser implementadas em outras apenas criando *ResourcesProcessors*, e seus *InputData* e *OutputData*. Vale ressaltar que essas atualizações podem ainda ser feitas por camadas, uma atualização na camada de core

business não tem impacto nenhum na camada de *foundation* e nem na camada common business.

Considerando que o desenvolvimento de um framework é um processo contínuo e iterativo, e os resultados obtidos nos projetos da arquitetura e dos mecanismos para encapsular e reutilizar conhecimento de domínio no framework RiS foram bastante satisfatórios, então o objetivo do trabalho foi atingido, essa versão inicial já possui os recursos para expandir e evoluir o RiS a partir de seus blocos fundamentais *Resource, ResourceProcessor, InputData* e *OutputData.*

5.1 Trabalhos Futuros

Existem duas propostas de trabalhos futuros para se seguir, durante a implementação do RiS e das instancias de validação, muitos pontos de melhoria foram percebidos e anotados, além de que várias funcionalidades foram deixadas de lados nesta primeira versão, então um caminho óbvio é continuar a evolução do RiS.

A obra de (FURTADO, 2002) é muito rica em detalhes sobre funcionalidades, fluxo de trabalho, implementações de sistemas do domínio da segurança pública, e ainda existem muitas funcionalidades para se extrair de lá e evoluir a camada de *core business* do RiS.

Alguns exemplos de melhorias são por exemplo o mapeamento de recursos em tabelas do banco de dados, talvez em uma revisão futura ele possa ser delegado ao framework e configurado pela aplicação através de uma classe *Strategy* ou um arquivo de configuração. O *ResourceProcessor* de login do RiS é muito simples, não é prático mantê-lo, para testar a validação de tokens de acesso no *RequestProcessor* ele foi suficiente, mas para APIs em produção é quase obrigatório escrever um novo *ResourceProcessor* para o login.

Na arquitetura atual do RiS o processamento de requisições é um frozen-spot do framework, uma sugestão para próximas revisões é estudar onde é possível flexibilizar esse fluxo, isso seria especialmente útil para que aplicação tenha controle em pontos específicos do processamento, como por exemplo permitir a aplicação interceptar os parâmetros antes de serem enviados a algum Resource, isso é fundamental na implementação de validações para proteção de dados (*Data Fencing*).

Outra proposta de trabalho futuro é reaproveitar a arquitetura ou o design do RiS em outros domínios de problema, ou seja reutilizar a camadas de *foundation* e *common business* projetada neste trabalho em um outro framework para construir APIs *RESTful* de algum outro domínio.

REFERÊNCIAS BIBLIOGRÁFICAS

1. CERQUEIRA, Daniel. Gastos com segurança pública. In: FÓRUM BRASILEIRO DE SEGURANÇA PÚBLICA. *Anuário Brasileiro de Segurança Pública*. 2019. Disponível em: <https://forumseguranca.org.br/wp-content/uploads/2019/10/Anuario-2019-FINAL_21.10.19.pdf>. Acesso em: 5 de junho de 2021
2. MIRANDA, Zil. Política de ciência, tecnologia e inovação para segurança pública. *Revista Brasileira de Segurança Pública*, v. 6, p. 434-453, ago./set. 2012. Disponível em: <https://www.forumseguranca.org.br/revista/index.php/rbsp/article/view/129/126>. Acesso em: 5 de junho de 2021
3. FIGUEIREDO, Isabel. A Gestão de Informações e o Papel da Senasp. *Boletim de Análise Político-Institucional*, nº 11, jan./jun. 2017. Disponível em: <http://repositorio.ipea.gov.br/handle/11058/8073>. Acesso em: 6 de junho de 2021
4. OLIVEIRA, Gerda G. R.; DUFLOTH, Simone C.; HORTA, Cláudia J. G.. Informações sobre criminalidade no Brasil sob a ótica dos pressupostos dos dados abertos governamentais e da Lei de Acesso à Informação. *Revista Brasileira de Segurança Pública* , v. 2, p. 48-64, 2014. ago./set. 2014. Disponível em <https://www.revista.forumseguranca.org.br/index.php/rbsp/article/view/387/179>. Acesso em 6 de junho de 2021
5. BRASIL. Lei nº 12.527, de 18 de novembro de 2011. Regula o acesso a informações previsto no inciso XXXIII do art. 5º , no inciso II do § 3º do art. 37 e no § 2º do art. 216 da Constituição Federal; altera a Lei nº 8.112, de 11 de dezembro de 1990; revoga a Lei nº 11.111, de 5 de maio de 2005, e dispositivos da Lei nº 8.159, de 8 de janeiro de 1991; e dá outras providências.
6. SOMMERVILLE, Ian. *Engenharia de Software*. Trad. Luiz C. Queiroz. 10ª ed. São Paulo: Pearson Education do Brasil Ltda, 2019. 756 p. ISBN 978-85-430-2497-4
7. SCHMIDT, Douglas C; GOKHALE, Aniruddha; NATARAJAN, Balachandran. Leveraging Application Frameworks. *ACM Queue*, v. 2, n. 5, p. 66-75, jul. 2004. ISSN: 1542-7730. DOI: 10.1145/1016998. Disponível em: <http://doi.acm.org/10.1145/1016998.1017005>. Acesso em: 13 de junho de 2021.
8. FRAKES, William; PRIETO-DIAZ, Rubén; GOGIA, B. K.; DARE: Domain analysis and reuse environment - Phase I Final Report. *DARPA Order No. 5916 - Contract # DAAH01-92-C-R040*, U. S. Army Missile Command (1992). Disponível

<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.464.290>. Acesso em 13 de junho de 2021

9. EZRAN, Michel; MORISIO, Maurizio; TULLY, Collin. *Practical Software Reuse*, 1ª ed. Springer, 2002. 222 p. ISBN 978-1-4471-0141-3
10. ALMEIDA, Eduardo; ALVARO, Alexandre; GARCIA, Vinicius; MASCENA, Jorge; BURÉGIO, Vanilson; NASCIMENTO, Leandro; LUCRÉDIO, Daniel; MEIRA, Silvio. *C.R.U.I.S.E: Component Reuse in Software Engineering*. Gráfica Dom Bosco, 2007. 210 p.
11. KRUEGER, Charles. Software reuse. *ACM Computing Serveys*, V. 24, N. 02 p.131-183. Jun. 1992. Disponível em <https://doi.org/10.1145/130844.13085>. Acesso em: 18 de junho de 2021
12. FERREIRA, Hiran N. M.; NAVES, Thiago F. *Reuso de software: suas vantagens, técnicas e práticas*. IX Enacomp, 2011. Disponível em <https://www.enacomp.com.br/2011/anais/trabalhos-aprovados/pdf/enacomp2011_submission_43.pdf>. Acesso em: 18 de junho de 2021
13. JOHNSON, Ralph E.; FOOTE, Brian. *Designing reusable classes*. Journal of Object-Oriented Programming vol. 1 - num. 2, p. 22–35, jun./jul. 1988. Disponível em <http://www.laputan.org/drc/drc.html>. Acesso em: 19 de junho de 2021
14. FAYAD, Mohamed E., SCHMIDT, Douglas C. *Object-oriented Application frameworks*. Communications of the ACM, vol. 40, num. 10 p. 32-38, 1997. Disponível em <https://dl.acm.org/doi/10.1145/262793.262798>. Acesso em: 9 de julho de 2021
15. GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John; BOOCH, Grady. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1ª ed., Addison-Wesley Professional, nov. 1994. 540 p. ISBN: 978-0-201-63361-0.
16. MATTSSON, Michael. *Object-oriented Frameworks - A survey of methodological issues*, Tese de Licenciatura, Department of Computer Science, Lund University, Suécia, 1996.
17. MARKIEWICZ, Marcus E.; LUCENA Carlos J. P.; *Object oriented framework development*, XRDS: Crossroads, The ACM Magazine for Students, vol. 7, num. 4, p. 3–9, jun. 2001. Disponível em <https://doi.org/10.1145/372765.372771>. Acesso em: 11 de julho de 2021
18. TALIGENT INC., Building Object-Oriented Frameworks, A Taligent White Paper, 1994. Disponível em <http://lhcb.web.cern.ch/computing/Components/postscript/buildingoo.pdf>. Acesso em: 11 de julho de 2021

19. BUSCHMANN, Frank et al., *Pattern-Oriented Software Architecture A System of Patterns*, 1ª ed. John Wiley & Sons, Sussex, 1996. 467 p. ISBN 0-471-95889-7
20. BOSCH, Jan; MOLIN, Peter; MATTSSON, Michael; BENGTSSON, PerOlof. *Object-oriented framework-based software development: problems and experiences*. ACM Computing Surveys, vol. 32, março 2000. Disponível em <https://doi.org/10.1145/351936.351939>. Acesso em: 24 de junho de 2021
21. ENDREI, Mark et al., *Patterns: service-oriented architecture and Web Services*. IBM, 2004. Disponível em <https://www.redbooks.ibm.com/abstracts/sg246303.html>. Acesso em: 25 de julho de 2021
22. ERL, Thomas. SOA: Princípios do design de serviço. Trad. Edson F, Carlos S. São Paulo: Pearson Prentice Hall, 2008. 320 p. ISBN 978-85-7608-189-3
23. W3C, *Web Services Architecture*. W3C Working Group Note 11 February 2004. Disponível em <https://www.w3.org/TR/ws-arch/>. Acesso em: 4 de julho de 2021
24. ABRAMS, Charles; SCHULTE, Roy W. *Service-Oriented Architecture Overview and Guide to SOA Research*. Technical report G00154463, Gartner Research, jan. 2008
25. FIELDING, Roy T. *Architectural styles and the design of network-based software architectures*. Dissertação de Ph.D.. University of California, Irvine, 2000. Disponível em <https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>. Acesso em: 1 de agosto de 2021
26. RICHARDSON, Leonard; RUBY, Sam. *Restful web services*. 1ª ed, O'Reilly, 419 p. Maio de 2007. ISBN 978-0-596-52926-0
27. ERL, Thomas et al., *SOA with REST: Principles, Patterns & Constraints for Building Enterprise Solutions with REST*, 1ª ed., Pearson, 2012. 577 p. ISBN 978-0-13-701251-0
28. FIELDING, Roy T., *REST APIs must be hypertext-driven*, Untangled musings of Roy T. Fielding, 2008. Disponível em <http://roy.gbiv.com/untangled/2008/rest-apismust-be-hypertext-driven>. Acesso em: 8 de agosto de 2021
29. IETF, *Uniform Resource Identifiers (URI): Generic Syntax*, RFC-2396, agosto 1998. Disponível em <https://www.ietf.org/rfc/rfc2396.txt>. Acesso em: 8 de agosto de 2021
30. PAUTASSO, Cesare; ZIMMERMAN, Olaf; LEYMANN, Frank. *Restful web services vs. "big"' web services: making the right architectural decision*. In: Proceedings of the 17th international conference on World Wide Web (WWW '08). ACM, New York, 2008. Disponível em <https://doi.org/10.1145/1367497.1367606>. Acesso em: 6 de agosto de 2021.

31. FURTADO, Vasco. Tecnologia da informação na segurança pública. 1ª ed, Rio de Janeiro, Garamond, 264 p., 2002. ISBN 85-8643587-2
32. KANG, Kyo C. et al., *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Software Engineering Institute, Carnegie Mellon University, 1999. Disponível em <https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=11231>. Acesso em 21 de agosto de 2021.
33. KANG, Kyo C. et al., *FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures*. In: Annals of Software Engineering, vol. 5, p. 143, 1998. Disponível em <https://doi.org/10.1023/A:1018980625587>. Acesso em 21 de agosto de 2021.
34. PREE, Wolfgang. *Hot-spot-driven framework development*. In: Summer School on Reusable Architectures in Object-Oriented software Development. ACM. p. 123-127. 1995. Disponível em <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.76.4910>. Acesso em 22 de agosto de 2021.
35. YANG, Young J. et al., *A UML-based object-oriented framework development methodology*. In: Proceedings Asia Pacific Software Engineering Conference, p. 211-218. 1998. Disponível em <https://doi.org/10.1109/APSEC.1998.733722>. Acesso em 22 de agosto de 2021.
36. Infosimples. Disponível em <https://infosimples.com>. Acesso em 1 de setembro de 2021.
37. BrasilAPI. Disponível em <https://infosimples.com>. Acesso em 1 de setembro de 2021.
38. Catálogo de APIs Governamentais, Ministério da Economia. Disponível em <https://www.gov.br/conecta/catalogo>. Acesso em 1 de setembro de 2021.
39. PHP: Hypertext Preprocessor. Disponível em <https://www.php.net> Acesso em 10 de setembro de 2021.
40. SINESP Cidadão, Governo do Brasil. Disponível em <https://play.google.com/store/apps/details?id=br.gov.sinesp.cidadao.android> Acesso em 15 de setembro de 2021.