UNIVERSIDADE DE BRASÍLIA
FACULDADE DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

CONTRATOS REST ROBUSTOS E LEVES: UMA ABORDAGEM EM DESIGN-BY-CONTRACT COM NEOIDL

LUCAS FERREIRA DE LIMA

ORIENTADOR: RODRIGO BONIFÁCIO DE ALMEIDA

DISSERTAÇÃO DE MESTRADO EM
ENGENHARIA ELÉTRICA

PUBLICAÇÃO: MTARH.DM - 017 A/99

BRASÍLIA/DF: JULHO - 2016.

UNIVERSIDADE DE BRASÍLIA
FACULDADE DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

CONTRATOS REST ROBUSTOS E LEVES: UMA ABORDAGEM EM DESIGN-BY-CONTRACT COM NEOIDL

LUCAS FERREIRA DE LIMA

DISSERTAÇÃO DE MESTRADO SUBMETIDA AO DEPARTAMENTO DE ENGENHARIA ELÉTRICA DA FACULDADE DE TECNOLOGIA DA UNIVERSIDADE DE BRASÍLIA, COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM ENGENHARIA ELÉTRICA.

APROVADA POR:

Prof. Rodrigo Bonifácio de Almeida, DSc. (CIC-UnB)
(Orientador)

Prof. Rafael Timóteo de Sousa Jr., DSc. (ENE-UnB)
(Examinador Interno)

Prof. Henrique Emanuel Mostaert Rebêlo, DSc. (UFPE)
(Examinador Externo)

BRASÍLIA/DF, 11 DE JULHO DE 2016.

FICHA CATALOGRÁFICA

LIMA, LUCAS FERREIRA DE

Contrato REST robustos e leves: uma abordagem em Design-by-Contract com NeoIDL. [Distrito Federal] 2016.

xvii, 98p., 297 mm (ENE/FT/UnB, Mestre, Engenharia Elétrica, 2016).

Dissertação de Mestrado - Universidade de Brasília.

Faculdade de Tecnologia.

Departamento de Engenharia Elétrica.

1. Computação orientada a serviços
2. *Design-by-Contract*
3. NeoIDL
4. REST

I. ENE/FT/UnB
II. Título (série)

REFERÊNCIA BIBLIOGRÁFICA

CONTRATOS REST ROBUSTOS E LEVES: UMA ABORDAGEM EM DESIGN-BY-CONTRACT COM NEOIDL

LIMA, L. F. (2016). Contratos REST Robustos e Leves: Uma Abordagem em Design-by-Contract com NeoIDL. Dissertação de Mestrado em Engenharia Elétrica, Publicação MTARH.DM - 17 A/99, Departamento de Engenharia Elétrica, Universidade de Brasília, Brasília, DF, 98p.

CESSÃO DE DIREITOS

NOME DO AUTOR: Lucas Ferreira de Lima.

TÍTULO DA DISSERTAÇÃO DE MESTRADO: Contratos REST Robustos e Leves: Uma Abordagem em Design-by-Contract com NeoIDL.

GRAU / ANO: Mestre / 2016

Lucas Ferreira de Lima
Cond. RK Conj. Antares Qd. L Cs. 48 Sobradinho
73.252-200 - Brasília - DF - Brasil.

DEDICATÓRIA

Aos meus pais
José Ferreira de Lima e
Terezinha Araújo de Lima †

AGRADECIMENTOS

Agradeço a Deus, pela fé e esperança para superar as adversidades.

A minha amada esposa Paula, pelo imenso companherismo, estímulo e por ter se desdobrado nos momentos de minha ausência, em especial nos cuidados com nossas jóias preciosas, nossos amados filhos João Filipe, Maria Luísa e Ana Júlia.

Aos meus pais, pela dedicação e carinho. Aos meus irmãos pelo exemplo determinante. Ao suporte indispensável de minha sogra e sogro, cunhados e sobrinhos.

Aos professores Rodrigo Bonifácio e Edna Canedo, pela orientação, paciência, incentivo e aprendizado.

Aos professores Rafael Timóteo, Ricardo Puttini e Valério Aymoré, pelas relevantes contribuições do conhecimento transmitido.

À equipe do TSE, pelo estímulo, colaboração, compreensão. Devo muito a vocês.

Aos professores da Universidade do Rio Grande no Norte, na pessoa do professor Uirá Kulesza.

Aos professores coordenadores André Noll, Ugo Dias e Kleber Melo, e à Adriana Reis e ao Igor pelo suporte no PPGEE.

Aos oficiais do Exército Brasileiro, na pessoa do Thiago Mael, pela enorme colaboração e crítica técnica. Nota de destaque para Leandro Loriato, pelas questões importantes debatidas.

À equipe da Secretaria de Orçamento Federal, na pessoa do Marcos César.

RESUMO

CONTRATOS REST ROBUSTOS E LEVES: UMA ABORDAGEM EM DESIGN-BY-CONTRACT COM NEOIDL

Autor: Lucas Ferreira de Lima
Orientador: Rodigo Bonifácio de Almeida
Programa de Pós-Graduação em Engenharia Elétrica
Brasília, julho de 2016

Contexto. A demanda por integração entre sistemas heterogêneos fez aumentar a adoção de soluções baseadas em computação orientada a serviços – SOC, sendo o uso de serviços Web a estratégia mais comum para implementar serviços, com a adoção crescente do estilo arquitetural REST. Por outro lado, REST ainda não dispõe de uma notação padrão para especificação de contratos e linguagens como Swagger, YAML e WADL cumprem com o único propósito de descrever serviços, porém apresentam uma significativa limitação: são voltadas para computadores, tendo escrita e leitura complexas para humanos, dificultando a abordagem *Contract-first*[1] estimulada em SOC. Tal limitação motivou a especificação da linguagem NeoIDL[2], concebida com o objetivo de ser mais expressiva para humanos, além de prover suporte a modularização e herança. *Problema.* Nenhuma dessas linguagens, incluindo a NeoIDL, dá suporte a contratos robustos, como os possíveis de serem descritos em linguagens ou extensões de linguagens com suporte a *Design-by-Contract* – DbC, exploradas tipicamente no paradigma de orientação a objetos. *Objetivos.* O objetivo geral deste trabalho é investigar o uso de construções de DbC no contexto de SOC, verificando a viabilidade e utilidade de sua adoção na especificação de contratos e implementação de serviços REST. *Resultados e Contribuições.* Essa dissertação contribui tecnicamente com uma extensão da NeoIDL para DbC, contemplando dois tipos de precondição e pós-condição: uma básica, que valida o valor de atributos e dados de saída; e outra baseada em serviços, em que composições de serviços são acionadas para validar se o serviço deve ser executado (ou se foi executado adequadamente, em caso de pós-condições). Sob a perspectiva de validação empírica, contribui-se com dois estudos. Um primeiro, verificou os requisitos de expressividade e reuso da NeoIDL, sendo realizado no domínio de Comando e Controle em parceria com o Exército Brasileiro. O segundo, teve maior interesse na análise da percepção de utilidade e facilidade de uso das construções DbC propostas para a NeoIDL, levando a respostas positivas em termos de facilidade de uso e aceitação.

[1] Descrito na Subseção 2.1.4.

[2] Além de ser uma linguagem (*Domain Specific Language*), a NeoIDL também possui um framework de geração de código para outras linguagens de propósito amplo.

ABSTRACT

LIGHTWEIGHT AND ROBUST REST CONTRACTS: AN APPROACH IN DESIGN-BY-CONTRACT WITH NEOIDL

Author: Lucas Ferreira de Lima
Orientation: Rodigo Bonifácio de Almeida
Program of Electrical Engineering Post-Graduation
Brasília, July 11, 2016

Context. The demand for integratin heterogeneus systems grows up the adoptions of solutions based on service oriented computing – SOC, in special with the increasing use of the REST architectural style. Nevertheless, there is no standard way to represent REST contracts. Swagger, YAML and WADL only provide mechanisms to describe services, which leads to a relevant limitation: they are made for computers and are hard for humans to write and read. This hinders the adoption of the Contract-First approach. This limitation motivated the creation of NeoIDL language, designed with the aim to be more expressive for humans, besides providing support to modularization and inheritance. *Problem* None of this specification languages, including NeoIDL, gives support to strong contracts as present in languages that supports Design-by-Contract, tipically found in the object oriented paradigm. *Objetives* The main objective of this work is to investigate the use of Design-by-Contract constructions in the SOC context, checking the viability and utility of its adoption at the REST contracts specification and service implementation. *Results and contributions* This master thesis contributes technically with the extention of NeoIDL towards supporting Design-by-Contract, adding to it two types of pre and post-conditions. The basic type checks the values of incoming and outgoing atributes. The service based type makes employes a kind of service composition by calling another service to check if the main service may be executed (or if it was correctly executed, in case of post-conditions). By the empirical validation perspective, this thesis contributes with two studies: the first, verifies the expressiveness and reusability requirements of NeoIDL, whitin the domain of Command and Control in colaboration with the Brazilian Army. The second study focused on the analysis of utility and easy of use perspectives of the Design-by-Contract constructions proposed. It gave us interesting answers in terms of acceptance and easy to use.

SUMÁRIO

LISTA DE TABELAS

LISTA DE FIGURAS

LISTA DE SÍMBOLOS, NOMENCLATURA E ABREVIAÇÕES

API	Application Programming Interface
BPM	Business Process Modeling
DbC	*Design-by-Contract*
DSL	Domain Specific Language
EAI	Interface Description Languages
Eiffel	Linguagem de programação orientada a objetos com suporte a DbC
GQM	Goals-Questions-Metrics
JML	Java Modeling Language
NeoCortex	Framework de serviços do Exército Brasileiro
NeoIDL	Linguagem e framework para especificação e geração de serviços REST
Twisted	Um framework orientada a eventos de rede escrito em Python
RAML	RESTful API Modeling Language
REST	Representational State Transfer
SOA	Software Oriented Architecture
SOC	Software Oriented Computing
Spec#	Extensão de linguagem C# para *Design-by-Contract*
Swagger	Open RESTful API Representation Language
TAM	Technology Acceptance Model
TI	Tecnologia da Informação
URI	Uniform Resource Identifier
WSDL	Web-services description language

1 INTRODUÇÃO

A computação orientada a serviços (*Service-oriented computing, SOC)* tem se mostrado uma solução de *design* de *software* que favorece o alinhamento às mudanças constantes e urgentes nas instituições [18]. Nessa abordagem, os recursos de software são empacotados como serviços, módulos bem definidos e auto-contidos, que provêem funcionalidades negociais e com independência de estado e contexto [44].

Os benefícios de SOC estão diretamente relacionados ao baixo acoplamento dos serviços que compõem a solução, de forma que as partes (nesse caso serviços) possam ser substituídas e evoluídas facilmente, ou ainda rearranjadas em novas composições. Contudo, para que isso seja possível, é necessário que os serviços possuam contratos bem escritos e independentes da implementação.

A relação entre quem provê e quem consome o serviço se dá por meio de um contrato. O contrato de serviço é o documento que descreve os propósitos e as funcionalidades do serviço, como ocorre a troca de mensagens, informações sobre as operações e condições para sua execução [22].

Nesse contexto, a qualidade da especificação do contrato é fundamental para o projeto de software baseado em SOC. Este trabalho de pesquisa aborda um aspecto importante para a melhoria da robustez de contratos de serviços: a construção de garantias mútuas por meio da especificação formal de contratos, agregando o conceito de *Design-by-Contract* [39].

1.1 PROBLEMA DE PESQUISA

As linguagens de especificação de contratos para SOC apresentam algumas limitações. Por exemplo, a linguagem WSDL (*Web-services description language*) [6] é considerada uma solução pouco expressiva pois se utiliza da notação XML e muitas marcações (*tags*) para descrever um contrato de *Web Service*. Essa característica desestimula a abordagem *Contract First*. Por essa razão, especificações WSDL são usualmente derivadas a partir de anotações em código fonte (*Code First*). Além disso, os conceitos

descritos em contratos na linguagem WSDL não são diretamente mapeados aos elementos que compõem as interfaces do estilo arquitetural REST[24] (*Representational State Transfer*).

Outras alternativas para REST, como Swagger[4] e RAML[3], usam linguagens de propósito geral (em particular JSON[2] e YAML[8]) adaptadas para especificação de contratos. Ainda que façam uso de contratos mais sucintos que WSDL, essas linguagens não se beneficiam da clareza típica das linguagens específicas para esse fim (como a IDL[1] CORBA[43]) e não oferecem mecanismos semânticos de extensibilidade (capacidade de extender uma especificação principal e agregar outros componentes) e modularidade (especificação do contrato em partes separadas e reusáveis).

Com o objetivo de mitigar esses problemas, a linguagem NeoIDL foi proposta para simplificar a especificação de serviços REST com mecanismos de modularização, suporte a anotações, herança em tipos de dados definidos pelo desenvolvedor, e uma sintaxe simples e concisa semelhante às IDLs presentes em *Apache Thrift*™[51] e *CORBA*™[43].

Por outro lado, a NeoIDL, da mesma forma que WSDL, Swagger e RAML não oferece construções para especificação de contratos formais com aspecto comportamental como os presentes em linguagens que suportam DbC (*Design by Contract*) [39], como Eiffel[38], JML[33] e Spec#[15]. Em outras palavras, a NeoIDL admite apenas contratos fracos (*weak contracts*), sem suporte a construções com pré e pós-condições.

1.2 OBJETIVO GERAL

O objetivo geral deste trabalho é investigar o uso de construções de *Design-by-Contract* no contexto de computação orientada a serviços, verificando a viabilidade e utilidade de sua adoção na especificação de contratos e implementação de serviços REST. Para atingir o objetivo geral, os seguintes objetivos específicos foram definidos, sendo diretamente mapeados nas principais contribuições do trabalho.

(OE1) Realizar análise empírica de expressividade e reuso da especificação de contratos em NeoIDL em comparação com *Swagger*, a partir de contratos reais do Exército Brasileiro (Seção 3.2);

[1] Interface Description Languages.

(OE2) Extender a sintaxe da NeoIDL para admitir construções de *Design-by-Contract*, com pré e pós-condições para operações de serviços REST (Seção 4.2);

(OE3) Incorporar à infraestrutura de *Plugins* da NeoIDL a capacidade de geração de código para o *framework Python Twisted* (Seção 4.3);

(OE4) Implementar regras de transformação que traduzem construções de DbC NeoIDL em código de validação para o framework *Python Twisted* (Seção 4.3);

(OE5) Coletar a percepção de desenvolvedores sobre a aceitação da especificação de contratos REST com *Design-by-Contract* na NeoIDL (Seção 4.4).

1.3 JUSTIFICATIVA E RELEVÂNCIA

A demanda por integração entre sistemas de várias origens e tecnologias diversas fez aumentar a adoção de soluções baseada em computação orientada a serviços. Isso se deve justamente à necessidade de tornar a interoperabilidade de soluções heterogênias o menos acopladas possível, de modo que mudanças nos requisitos de negócio ou na inclusão de novos serviços sejam atendidas com simplicidade, eficiência e rapidez.

O uso de *Web Service*[22] é a forma mais comum de se implementar os serviços. O desenvolvimento de *Web Service*, que eram inicialmente projetados sobre a abordagem SOAP, com o tráfego de mensagens codificadas em XML[5], tem gradativamente se intensificado no sentido da utilização de REST[24].

Um dos principais benefícios do uso de SOC está na possibilidade de reuso de seus componentes. Porém, reuso requer serviços bem construídos e precisos em relação a sua especificação [30]. A qualidade e precisão do contrato de serviço torna-se claramente um elemento fudamental para se auferir os benefícios da abordagem SOC.

Nesse contexto, REST não dispõe de um meio padrão para especificação de contratos. Linguangens como Swagger, YAML e WADL cumprem com o propósito de especificar contratos REST, porém padecem do mesmo problema: são voltados para computadores e de escrita e leitura complexa para humanos, o que prejudica a prática de *Contract-first*. A linguagem NeoIDL foi concebida com o objetivo de ser mais expressiva para humanos, além de outros propósitos, como extensibilidade e modularidade.

Todas essas linguagens tem, entretanto, uma outra limitação em comum: não dão

suporte a contratos robustos, com garantias. A estratégia para superar essa limitação foi de buscar no paradigma de orientação a objetos, que é uma das principais influências de orientação a serviços [22], o conceito de *Design-by-Contract*. Ambas as abordagens, orientação a serviços e a objetos, tem em comum a ênfase no reuso e comunicação entre componentes (serviços e classes).

A principal contribuição deste trabalho de pesquisa de mestrado está em incluir garantias na especificação de contratos REST, extendendo a linguagem NeoIDL para suportar construções de *Design-by-Contract*.

1.4 TRABALHOS RELACIONADOS

1.5 ESTRUTURA

Este trabalho está organizado em quatro capítulos. O capítulo 2 faz uma revisão teórica sobre o tema computação orientada a serviços (SOC), seus propósitos, princípios e terminologia. São tratadas estratégias recomendadas para se atingir os resultados esperados com SOC, com enfâse na prática de padronização do contrato, sobretudo quanto à abordagem *Contract-first* e ao uso *Web Services*. Ao final, é feita uma apresentação do conceito de *Design-by-Contract*, sua finalidade e implementações.

O capítulo 3 descreve a NeoIDL, uma linguagem e *framework* de geração de código para especificação e geração de serviços REST. É contextualizada a motivação para a criação da NeoIDL, sua estrutura sintática e componentes. Esse capítulo inclui um estudo empírico realizado no decorrer da pesquisa de mestrado sobre expressividade e reuso da NeoIDL em comparação a Swagger, em um cenário real do Exército Brasileiro.

As principais contribuições desta dissertação estão concentradas no capítulo 4. Esse capítulo se inicia com a proposta para incorporação de construções de pré e pós-condições à especificação de contratos e qual é o modelo de operação do serviço em execução. A extensão da linguagem é debatida em etapas, apresentando as condições mais simples, seguindo para as mais completas. Um estudo de caso com a geração de código para um serviços em *Python Twisted* com suporte a DbC é apresentado. A última seção demonstra os resultados de um estudo empírico subjetivo sobre a percepção de utilidade e de facilidade de uso da NeoIDL com DbC.

Por fim, o capítulo 5 apresenta as conclusões do trabalho de pesquisa e sugere a realização de trabalhos futuros.

2 REFERENCIAL TEÓRICO

2.1 COMPUTAÇÃO ORIENTADA A SERVIÇO

As empresas precisam estar preparadas para responder rápidamente e eficientemente a mudanças impostas por novas regulações, por aumento de competição ou ainda para usufruir de novas oportunidades. No contexto atual, em que as informações fluem de modo extremamente veloz, o tempo disperdiçado pelas organizações para se adaptar a um novo cenário tem um preço elevado, gerando expressiva perda de receita e, em determinados casos, podendo causar a falência.

No campo das instituições governamentais, a eficiência na condução das ações do Estado impõem que a estrutura de troca de informações entre os mais variados entes seja continuamente adaptável, mutuamente integrada. Pode-se tomar como exemplo a edição de nova lei que implique alteração no cálculo do tempo de serviço para aposentadoria. A nova fórmula deve se propagar para ser aplicada em várias instituições que compõem a máquina pública.

Nessas situações, os sistemas de informação das organizações devem possibilitar que a dinâmica de adaptação ocorra sem demora, sob pena de, em vez de serem ferramentais para apoiar continuamente os processos de negócio, se tornem entrave para a ágil incorporação dos novos processos. Por outro lado, a nova configuração deve se manter integra e funcional com o cenário de Tecnologia da Informação – TI – existente, normalmente complexo.

A eficiência na integração entre as soluções de TI é determinante para que se consiga alterar uma parte sem comprometer todo o ecossistema. A integração possibilita a combinação de eficiência e flexibilidade de recursos para otimizar a operação através e além dos limites de uma organização, proporcionando maior interoperabilidade [45].

A computação orientada a serviços – SOC – endereça essas necessidades em uma plataforma que aumenta a flexibilidade e melhora o alinhamento com o negócio, a fim de reagir rapidamente a mudanças nos requisitos de negócio. Para obter esses benefícios, contudo, os serviços devem cumprir com determinados quesitos, que incluem alta au-

tonomia ou baixo acoplamento [21]. Assim, o paradigma de SOC está voltado para o projeto de soluções preparadas para constantes mudanças, substituindo-se continuamente pequenas peças – os serviços – por outras atualizadas.

Portando, o objetivo da SOC é conceber um estilo de projeto, tecnologia e processos que permitam às empresas desenvolver, interconectar e manter suas aplicações e serviços corporativos com eficiência e baixo custo. Embora esses objetivos não sejam novos, SOC procura superar os esforços prévios como programação modular, reuso de código e técnicas de desenvolvimento orientadas a objetos [46].

As vertentes mais visionárias da computação orientada a serviços prevêem, em seu estado da arte, uma coordenação de serviços cooperantes por todo o mundo, onde os componentes possam ser conectados facilmente em uma rede de serviços pouquíssimo acoplados e, assim, criar processos de negócio dinâmicos e aplicações ágeis entre organizações e plataformas de computação [34].

2.1.1 Terminologia

Computação orientada a serviço é um termo *guarda-chuva* para descrever uma nova geração de computação distribuída. Desse modo, é um conceito que engloba vários pontos, como paradigmas e princípios de projeto, catálogo de padrões de projeto, padronização de linguagem, modelo arquitetural específico, e conceitos correlacionados, tecnologias e plataformas. A computação orientada a serviços é baseada em modelos anteriores de computação distribuída e os extendem com novas camadas de projeto, aspectos de governança, e uma grande gama de tecnologias de implementações especializadas, em grande parte baseadas em *Web Service* [22].

Orientação a serviço é um paradígma de projeto cuja intenção é a criação de unidades lógicas moldadas individualmente para poderem ser utilizadas conjutamente e repetidamente, atendendo assim a objetivos e funções específicos associados com SOA e computação orientada a serviço.

A lógica concebida de acordo com orientação a serviço pode ser designada de **orientada a serviço**, e as unidades da lógica orientada a serviço são referenciadas como **serviços**. Como um paradigma de computação distribuída, a orientação a serviço pode ser comparada a orientação a objetos, de onde advém várias de suas

raízes, além da influência de *Interface Description Languages* – EAI, *Business Process Modeling* – BPM e *Web Service*[22].

A orientação a serviços é composta principalmente de oito princípios de projeto (os quais serão descritos na Subseção 2.1.3).

Arquitetura orientada a serviço - SOA representa um modelo arquitetural cujo objetivo é elevar a agilidade e a redução de custos e ao mesmo tempo reduzir o peso da Tecnologia da Informação (TI) para a organização. Isso é feito colocando o serviço como elemento central da representação lógica da solução [22].

Como uma arquitetura tecnológica, uma implementação SOA consiste da combinação de tecnologias, produtos, APIs, extensões da infraestrutura, etc. A implantação concreta de uma arquitetura orientada a serviço é única para cada organização, entretanto é caracterizada pela introdução de tecnologias e plataformas que suportam a criação, execução e evolução de soluções orientadas a serviços. O resultado é a formação de um ambiente projetado para produzir soluções alinhadas aos princípios de projeto de orientação a serviço.

Segundo Thomas Erl [22], o termo arquitetura orientada a serviço – SOA – vem sendo amplamente utilizado na mídia e nos produtos de divulgação de fabricantes e se tornado quase que sinônimo de computação orientada a serviço – SOC.

Serviço é a unidade da solução no qual foi aplicada a orientação a serviço. É a aplicação dos princípios de projeto de orientação a serviço que distigue uma unidade de lógica como um serviço comparada a outras unidades de serviços que podem existir isoladamente como um objeto ou componente [22].

Após a modelagem conceitual do serviço, os estágios de projeto e desenvolvimento produzem um serviço que é um programa de *software* independente com características específicas para suportar a realização dos objetivos associados a computação orientada a serviço.

Cada serviço possui um contexto funcional distinto e é composto de uma lista de capacidades relacionadas a esse contexto. Então um serviço pode ser considerado um conjunto de capacidades descritas em seu contrato.

Contrato de serviço é o conjunto de documentos que expressam as meta-informações do serviço, sendo a parte que descreve a sua interface técnica, a mais fundamental. Esses documentos compõem o contrato técnico do serviço, cuja essência é estabelecer uma API com as funcionalidades providas pelo serviço por meio de suas capacidades [22].

Os serviços implementados como *Web Service* SOAP normalmente são descritos em arquivos *Web Service Description Language* – WSDL, *XML schemas* and políticas (*WS-policy*). Já os serviços implementados como *Web Service* REST não possuem uma linguagem padrão para especificação de contratos. Já foram propostas algumas alternativas como WADL [26], Swagger [4], e NeoIDL [35].

O contrato de serviço também pode ser composto de documentos de leitura humana, como os que descrevem níveis de serviços (*SLA*), comportamentos e limitações. Muitas dessas características também podem ser descritas em linguagens formais (para processamento computacional).

No contexto de orientação a serviço, o projeto do contrato do serviço é de suma importância de tal forma que o princípio de projeto contrato de serviço padronizado dedica-se exclusivamente ao cuidado com contratos de serviços uniformes e de qualidade [22].

2.1.2 Objetivos, benefícios e características

De modo diferente de arquiteturas convencionais, ditas monolíticas, em que os sistemas são concebidos agregando continuamente funcionalidades a um mesmo pacote de *software*, a arquitetura orientada a serviço prega o projeto de pequenas aplicações distribuídas que podem ser consumidas tanto por usuários finais como por outros serviços [46].

A unidade lógica da arquitetura orientada a serviços é exatamente o serviço. Serviços são pequenos *softwares* que provêem funcionalidades específicas para serem reutilizadas em várias aplicações. Cada serviço é uma entidade isolada com dependências limitadas de outros recursos compartilhados [49]. Assim, é formada uma abstração entre os fornecedores e consumidores dos serviços, por meio de baixo acoplamento, e promovendo a flexibilidade de mudanças de implementação sem impacto aos consumidores.

A arquitetura SOC busca atingir um conjunto de objetivos e benefícios [20]:

(a) Ampliar a interoperabilidade intrínseca, de modo a se ter uma rápida resposta a mudanças de requisitos de negócio por meio da efetiva reconfiguração das composições de serviços;

(b) Ampliar a federação da solução, permitindo que os serviços possam ser evoluídos e governados individualmente, a partir da uniformização de contratos;

(c) Ampliar a diversificação de fornecedores, fazendo com que se possa evoluir a arquitetura em conjunto com o negócio, sem ficar restrito a características de derminados fornecedores;

(d) Ampliar o alinhamento entre a tecnologia e o negócio, especializando-se alguns serviços ao contexto do negócio e possibilitando sua evolução;

(e) Ampliar o retorno sobre investimento, pois muitos serviços podem ser rearranjados em novas composições sem que se tenha que se construir grandes soluções de custo elevado;

(f) Ampliar a agilidade, remontando as composições por reduzido esforço, beneficiando-se do reuso e interoperabilidade nativas dos serviços;

(g) Reduzir o custo de TI, como resultado de todos os benefícios acima citados.

Para possibilitar que esses benefícios sejam atingidos, quatro características são observadas em qualquer plataforma SOA. A primeira é o direcionamento efetivo ao negócio, levando-se em conta os objetivos estratégicos de negócio na concepção do projeto arquitetural. Se isso não ocorrer, é inevitável que o desalinhamento com os requisitos de negócio cheguem a níveis muito elevados bem rapidamente [20].

A segunda característica é a independência de fabricante. O projeto arquitetural que considera apenas um fabricante específico levará inadvertidamente a implantação dependente de características proprietárias. Essa dependência também reduzirá a agilidade na reação às mudanças e tornará a arquitetura inefetiva. A arquitetura orientada a serviço deve fazer uso de tecnologias providas pelos fornecedores, sem, no entanto, se tornar dependente dela, por meio de APIs e protocolos padrões de mercado.

Outra característica da aplicação da plataforma SOA é os serviços serem considerados recursos corporativos, ou seja, da empresa como um todo. Serviços desenvolvidos para atender um único objetivo perdem esta característica e se assemelham a soluções de propósito específico, tal como soluções monolíticas. O modelo arquitetural deve se guiar pela premissa de que os serviços serão compartilhados por várias áreas da empresa ou farão parte de soluções maiores, como serviços compatilhados.

A capacidade de composição é a quarta característica. Os serviços devem ser projetados não somente para serem reusados, mas também para possuir flexibilidade em

serem compostos em diferentes estruturas de variadas soluções. Confiabilidade, escalabilidade, troca de dados em tempo de execução com integridade são pontos chave para essa característica.

2.1.3 Princípios SOA

O paradigma de orientação a serviço é estruturado em oito princípios fundamentais [22], ilustrados na Figura 2.1. São eles que caraterizam a abordagem SOA e a sua aplicação faz com que um serviço se diferencie de um componente ou de um módulo. Os contratos de serviços permeiam a maior parte destes princípios.

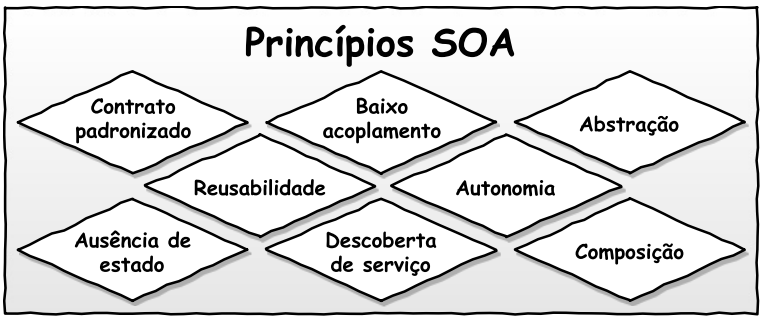

Figura 2.1: Oito princípios da arquitetura orientada a serviços [22]

Contrato padronizado - Serviços dentro de um mesmo inventário estão em conformidade com os mesmos padrões de contrato de serviço. Os contratos de serviços são elementos fundamentais na arquitetura orientada a serviço, pois é por meio deles que os serviços interagem uns com os outros e com potenciais consumidores. Este princípio tem como foco principal o contrato de serviço e seus requisitos. O padrão de projeto *Contract-first* é uma consequência direta deste princípio [22].

Baixo acomplamento - Os contratos de serviços impõem aos consumidores do serviço requisitos de baixo acoplamento e são, os próprios contratos, desacoplados do seu ambiente. Este princípio também possui forte relação com o contratos de serviço, pois a forma como o contrato é projetado e posicionado na arquitetura é que gerará o benefício do baixo acoplamento. O projeto deve garantir que o contrato possua tão somente as informações necessárias para possibilitar a compreensão e o consumo do serviço, bem como não possuir outras características que gerem acoplamento.

São considerados negativos, e que devem ser evitados, os acoplamentos

(a) do contrato com as funcionalidades que ele suporta, agregando ao contrato características específicas dos processos que o serviço atende;

(b) do contrato com a sua implementação, invertendo a estratégia de conceber primeiramente o contrato;

(c) do contrato com a sua lógica interna, expondo aos consumidores características que levem os consumidores a inadivertidamente aumentarem o acoplamento;

(d) do contrato com a tecnologia do serviço, causando impactos indesejáveis em caso de substituição de tecnologia.

Por outro lado, há um tipo de acoplamento positivo que é o que gera dependência da lógica em relação ao contrato [22]. Ou seja, idealmente a implementação do serviço deve ser derivada do contrato, pondendo se ter inclusive a geração de código a partir do contrato.

Abstração - Os contratos de serviços devem conter apenas informações essenciais e as informações sobre os serviços são limitadas àquelas publicadas em seus contratos. O contrato é a forma oficial a partir da qual o consumidor do serviço faz seu projeto e tudo o que está além do contrato deve ser desconhecido por ele. Por um lado este princípio busca a ocultação controlada de informações. Por outro, visa a simplificação de informações do contrato de modo a assegurar que apenas informações essenciais estão disponíveis.

Reusabilidade - Serviços contém e expressam lógica agnóstica e podem ser disponibilizados como recursos reutilizáveis. Este princípio contribui para se entender o serviço como um produto e seu contrato com uma API genérica para potenciais consumidores. Essa abordagem aplicada ao projeto dos serviços leva a desenhá-lo com lógicas não dependentes de processos de negócio específicos, de modo a torná-los reutilizáveis em vários processos.

Autonomia - Serviços exercem um elevado nível de controle sobre o seu ambiente em tempo de execução. O controle do ambiente não está ligado a dependência do serviço à sua plataforma em termos de projeto, mas sim ao aumento da confiabilidade sobre a execução e redução da dependência dos recursos sobre os quais não se tem controle. O que se busca é a previsibilidade sobre o comportamento do serviço.

Ausência de estado - Serviços reduzem o consumo de recursos restringindo a gestão de estado das informações apenas a quando for necessário. Este

princípio visa reduzir ou mesmo remover a sobrecarga gerada pelo gerenciamento do estado de cada operação, aumentando a escalabilidade da plataforma de arquitetura orientação a serviço como um todo. Na composição do serviço, o serviço deve armazenar apenas os dados necessários para completar o processamento, enquanto se aguarda o processamento do serviço acionado.

Descoberta de serviço - Serviços devem conter metadados por meio dos quais os serviços possam ser descobertos e interpretados. Tornar cada serviço de fácil descoberta e interpreção pelas equipes de projeto é o foco deste princípio. Os próprios contratos de serviço devem ser projetados para incorporar informações que auxiliem na sua descoberta.

Composição - Serviços são participantes efetivos de composição, independentemente do tamanho ou complexidade da composição. O princípio da composição faz com que os projetos de serviços sejam projetados para possibilitar que eles se tornem participantes de composições. Deve-se levar em conta, entretanto, os outros princípios no planejamento de uma nova composição, considerando a complexidade das composições a serem formadas.

2.1.4 Contract First

O princípio do baixo acoplamento tem por objetivo principal reduzir o acoplamento entre o cliente e o fornecedor do serviço. Há vários tipos de acoplamentos negativos, como citado na Subseção 2.1.3. Porém, um acoplamento é considerado positivo e desejável: da implementação a partir do contrato. Ou seja, a lógica do serviço deve corresponder ao que está especificado no contrato, permitindo assim que o serviço seja reutilizado exclusivamente com conhecimento do contrato.

Duas abordagens podem ser seguidas para se produzir esse efeito. A primeira é a geração do contrato a partir da lógica implementada, conhecida como *Code-first*. A outra propõe um sentido inverso, partindo-se do contrato para a geração do código, chamada *Contract-first*. A construção do serviço a partir da modelagem do contrato, defendida na abordagem *Contract-first*, é recomendada para a arquitetura orientada a serviço [22].

Embora muitas vezes preferível pelo desenvolvedor, a desvantagem do uso *Code-first* está no elevado impacto que alterações na implementação causam ao contrato, fazendo

com que os clientes dos serviços sejam também afetados. Reduz-se a flexibilidade e extensibilidade, de modo que o reuso é prejudicado. Ainda, eleva-se o risco de os serviços serem projetados para aplicações específicas e não voltados para reuso e composição [31].

A abordagem *Contract-first* preocupa-se principalmente com a clareza, completude e estabilidade do contrato para os clientes dos serviços. Toda a estrutura da informação é definida sem a preocupação sobre restrições ou características das implementações subjacentes. Do mesmo modo, as capacidades são definidas para atenderem a funcionalidades a que se destinam, porém com a preocupação em se promover estabilidade e reuso.

As principais vantagens do *Contract-first* estão no baixo acoplamento do contrato em relação a sua implementação, na possibilidade de reuso de esquemas de dados (XML ou JSON Schema), na simplificação do versionamento e na facilidade de manutenção [31]. A desvantagem está justamente na complexidade de escrita do contrato. Porém, várias ferramentas já foram e vem sendo desenvolvidas para facilitar essa tarefa.

2.2 WEB SERVICES

Web Service são aplicações modulares e autocontidas que podem ser publicadas, localizadas a acessadas pela *Web* [12]. A diferença entre o *Web Service* e a aplicação *Web* propriamente dita é que o primeiro se preocupa apenas com o dado gravado ou fornecido, deixando para o cliente a atribuição de apresentar a informação [49].

A necessidade das organizações de integrar suas soluções, seja entre os sistemas internos ou entre esses e sistemas de outras empresas [48], não é recente. Essa é uma das principais motivações do uso de *Web Service*, por possibilitar que soluções construídas com tecnologias distintas possam trocar informações por meio da *Web*. Nesse contexto, as arquiteturas orientadas a serviço fazem amplo uso de *Web Service* como meio para disponibilização de serviços.

Há dois tipos de *Web Service*: baseados em SOAP e baseados em REST. Os mais diversos tipos de aplicações podem ser concebidas utilizando *Web Services* SOAP ou REST, situação também aplicável a serviços. Originalmente os serviços utilizaram *Web Service* SOAP, trafegando as informações em uma mensagem codificada em um formato

de troca de dados (XML), por meio do protocolo SOAP (Seção 2.2.1). Entretanto, a adoção de *Web Service* REST (Seção 2.2.2) tem ganhado popularidade [42].

2.2.1 SOAP (W3C)

SOAP – *Simple Object Access Protocol* – é um protocolo padrão W3C que provê uma definição de como trocar informações estruturadas, por meio de XML, entre as partes de ambientes descentralizados ou distribuídos [6]. SOAP é um protocolo mais antigo que REST, e foi desenvolvido para troca de informações pela Internet se utilizando de protocolos como HTTP, SMTP, FTP, sendo o primeiro o mais comumente utilizado.

Por ser anterior, SOAP é o padrão de *Web Service* mais comumente utilizado pela indústria. Algumas pessoas chegam a tratar *Web Service* apenas como SOAP e WSDL [49]. SOAP atua como um envelope que transporta a mensagem XML, e possui vastos padrões para transformar e proteger a mensagem e a trasmissão.

2.2.1.1 Especificação de contratos

Os contratos em SOAP são especificados no padrão WSDL – *Web Services Description Language* – que define uma gramática XML para descrever os serviços como uma coleção de *endpoints* capazes de atuar na troca de mensagens. As mensagens e operações são descritas abstratamente na primeira seção do documento. Uma segunda seção, dita concreta, estabelece o protocolo de rede e o formato das mensagens.

Muitas organizações preferem utilizar SOAP por ele dispor de mais mecanismos de segurança e tratamento de erros [49]. Além disso, a tipagem de dados é mais forte em SOAP que em REST [42], uma vez que em SOAP se pode fazer uso de restrições e regras de validação providas pelo padrão *XML Schema* [7].

2.2.2 REST (Fielding)

O termo REST foi criado por Roy Fielding, em sua tese de doutorado [24], para descrever um modelo arquitetural distribuído de sistemas hipermedia. Um *Web Service* REST é baseado no conceito de recurso (que é qualquer coisa que possua uma *Uniform Resource Identifier* – URI) que pode ter zero ou mais representações [27].

O estilo arquitetural REST é cliente-servidor, em que o cliente envia uma requisição por um determinado recurso ao servidor e este retorna uma resposta. Tanto a requisição como a resposta ocorrem por meio da transferência de representações de recursos [42], que podem ser de vários formatos, como XML e JSON [49]. Toda troca de informações ocorre por meio do protocolo HTTP, com uma semântica específica para cada operação:

1. HTTP GET é usado para obter a representação de um recurso.
2. HTTP DELETE é usado para remover a representação de um recurso.
3. HTTP POST é usado para atualizar ou criar a representação de um recurso.
4. HTTP PUT é usado para criar a representação de um recurso.

As transações são independentes entre si e com as transações anteriores, pois o servidor não guarda qualquer informação de sessão do cliente. Todas as informações de estado são trafegadas nas próprias requisições, de modo que as respostas também são independentes. Essas características tornam os *Web Services* REST simples e leves [42].

O uso de REST tem se tornado popular por conta de sua flexibilidade e performance em comparação com SOAP, que precisa envelopar suas informações em um pacote XML [42], de armazenamento, transmissão e processamento onerosos.

2.2.2.1 Especificação de contratos

Ao contrário de SOAP, REST não dispõe de um padrão para especificação de contratos. Essa carência, que no início não era considerada um problema, foi se tornando uma necessidade cada vez mais evidente a medida em que se amplia o conjunto de *Web Services* implantados. Atualmente, existem algumas linguagens com o propósito de documentar o contrato REST.

A linguagem mais popular atualmente é *Swagger* cujo projeto se iniciou por volta de 2010 para atender a necessidade de um projeto específico, sendo posteriormente vendida para uma grande empresa. Em janeiro de 2016, *Swagger* foi doada para o *Open API Iniciative (OAI)* e denominada de *Open API Specification*. O propósito da

iniciativa é tornar *Swagger* padrão para especificação de APIs com independencia de fornecedor. Apoiam o projeto grandes empresas como Google®, Microsoft® e IBM®.

WADL (*Web Application Description Language*), uma especifição baseada em XML semelhante ao WSDL, foi projetada e proposta pela *Sun Microsystems*® e sua última versão submetida ao W3C em 2009. Outra linguagem proposta é a RAML[3] – abreviação de *RESTful API Modeling Language* – baseada em YAML e projetada pela MuleSoft®. Muitos projetos *open source* adotam RAML.

Todas estas linguagens possuem suporte tanto para *Code-first* como para *Contract-first* [54].

2.3 DESIGN BY CONTRACT

Design-by-Contract [39] - DbC - é um conceito oriundo da orientação a objetos, no qual consumidor e fornecedor firmam entre si garantias para o uso de métodos ou classes. De um lado o consumidor deve garantir que, antes da chamada a um método, algumas condições sejam por ele satisfeitas. Do outro lado o fornecedor deve garantir, se respeitadas suas exigências, o sucesso da execução.

O mecanismo que expressa essas condições são chamados de asserções (*assertions*, em inglês). As asserções que o consumidor deve respeitar para fazer uso da rotina são chamadas de **precondições**. As asserções que asseguram, de parte do fornecedor, as garantias ao consumidor, são denominadas **pós-condições**.

DbC tem o objetivo de aumentar a robustez do sistema e tem na linguagem Eiffel [38] um de seus precursores. Para os mantenedores do Eiffel, DbC é tão importante quanto classes, objetos, herança, etc. O uso de DbC na concepção de sistemas é uma abordagem sistemática que produz sistemas com mais corretude.

O conceito chave de *Design-by-Contract* é ver a relação entre a classe e seus clientes como uma relação formal, que expressa os direitos e as obrigações de cada parte [40]. Se, por um lado, o cliente tem a obrigação de respeitar as condições impostas pelo fornecedor para fazer uso do módulo, por outro, o fornecedor deve garantir que o retorno ocorra como esperado.

As precondições vinculam o cliente, no sentido de definir as condições que o habilitam para acionar o recurso. Corresponde a uma obrigação para o cliente e o benefício para o fornecedor [40] de que certos pressupostos serão sempre respeitados nas chamadas à rotina. As pós-condições vinculam o fornecedor, de modo a definir as condições para que o retorno ocorra. Corresponde a uma obrigação para o fornecedor e o benefício para o cliente de que certas propriedades serão respeitadas após a chamada à rotina.

De forma indireta, *Design-by-Contract* estimula um cuidado maior na análise das condições necessárias para, de forma consistente, se ter o funcionamento correto da relação de cooperação cliente-fornecedor. Essas condições são expressas em cada contrato, o qual especifica as obrigações a que cada parte está condicionada e, em contraponto, os benefícios garantidos.

Nesse contexto, o contrato é um veículo de comunicação, por meio do qual os clientes tomam conhecimento das condições de uso, em especial das precondições. É fundamental que as precondições estejam acessíveis para todos os clientes para os quais as rotinas estão disponíveis, pois, sem que isso ocorra, o cliente corre o risco de acionar a rotina fora de suas garantias de funcionamento.

Segundo Bertrand Meyer [40], *Design-by-Contract* é um ferramental para análise, projeto, implementação e documentação, facilitando a construção de *softwares* cuja confiabilidade é embutida, no lugar de buscar essa característica por meio de depuração. Meyer utiliza uma expressão de Harlan D. Mills [41] para afirmar que *Design-by-Contract* permite construir programas corretos e saber que eles estão corretos.

Com o uso de *Design-by-Contract*, cada rotina é levada a realizar o trabalho para o qual foi projetada e fazer isso bem: com corretude, eficiência e genericamente suficiente para ser reusada. Por outro lado, especifica de forma clara o que a rotina não trata. Esse paradigma é coerente, pois para que a rotina realize seu trabalho bem, é esperado que se estabeleça bem as circunstâncias de execução.

Outra característica da aplicação de *Design-by-Contract* é que o recurso tem sua lógica concentrada em efetivamente cumprir com sua função principal, deixando para as precondições o encargo de validar as entradas de dados. Essa abordagem é o oposto à ideia de programação defensiva, pois vai de encontro à realização de checagens redundantes. Se os contratos são precisos e explícitos, não há necessidade de testes redundantes [39].

Todos esses aspectos são fundamentais para se possibilitar o reuso eficiente de componentes, que é o pilar da orientação a objetos e se aplica de forma análoga à orientação a serviços. Componentes reusáveis por várias aplicações devem ser robustos pois as consequências de falhas ou comportamentos incorretos são muito piores que as de aplicações que atendem a único propósito [39].

Há de se registrar ainda que, em orientação a objetos, existe outro tipo de asserção além das pré e pós-condições. Em vez de cuidar das propriedades de cada rotina individualmente, elas expressam condições globais para todas as instâncias de uma classe [40]. Essa categoria de asserção é denominada invariante. Uma vez que em orientação a serviço se preconiza a ausência de estado, o conceito de invariante não é explorado neste trabalho, sem prejuízo da realização de estudo específico sobre o tema.

2.3.1 Implementações de DbC

Eiffel - A linguagem Eiffel foi desenvolvida em meados dos anos 80 por Bertrand Meyer [38] com o objetivo de criar ferramentas que garantissem mais qualidade aos *softwares*. A ênfase do projeto de Eiffel foi promover reusabilidade, extensibilidade e compatibilidade. Características que só fazem sentido se os programas forem corretos e robustos.

Foi essa preocupação que incorporou à linguagem Eiffel o conceito de contratos. A partir desse estilo de projeto se criou a noção de *Design-by-Contract*, concretizada na linguagem por meio das precondições, pós-condições e invariantes [38]. Esta abordagem influenciou outras linguagem de programação orientadas a objeto.

A Figura 2.2 apresenta um exemplo de especificação de pré e pós-condição na linguagem Eiffel.

JML - *Java Modeling Language* é uma extensão da linguagem Java para suporte a especificação comportamental de interfaces, ou seja, controlar o comportamento de classes em tempo de execução. Para realizar essa função, JML possui amplo suporte a *Design-by-Contract*. As asserções (precondição, pós-condição e invariantes) são incluídas no código Java na forma de comentários (//@ ou /*@...@*/).

JML combina a praticidade de *Design-by-Contract* de linguagens como Eiffel com a expressividade e formalismo de linguagens de especificação orientadas a modelo [33].

```
1  class interface ACCOUNT create
2      make
3  feature
4      balance: INTEGER
5      ...
6      deposit (sum: INTEGER) is
7              -- Deposit sum into the account.
8          require
9              sum >= 0
10         ensure
11             balance = old balance + sum
12
13 end -- class ACCOUNT
```

Figura 2.2: Exemplo de pré e pós-condições em Eiffel

A Figura 2.3 apresenta um exemplo de especificação de pré e pós-condição na linguagem de extensão JML.

```
1  public class IntMathOps {
2
3    /*@ public normal_behavior
4      @ requires y >= 0;
5      @ ensures \result * \result <= y && y < (Math.abs(\result) + 1)
6      @ * (Math.abs(\result) + 1);
7      @*/
8
9    public static int intSqrt(int y)
10   {
11     return (int) Math.sqrt(y);
12   }
13 }
```

Figura 2.3: Exemplo de pré e pós-condições em JML

Spec# - é uma extensão da linguagem C#, à qual agrega o suporte para distinguir referência de objetos nulos de referência a objetos possivelmente não nulos, especificações de pré e pós-condições e um método para gerenciar exceções entre outros recursos [15].

A Figura 2.4 apresenta um exemplo de especificação de pré e pós-condição na linguagem Spec#.

```
1  class CircularList {
2
3    // Construct an empty circular list
4    public CircularList()
5      require true;
6      ensure Empty();
7
8    // Return my number of elements
9    public int Size()
10     require true;
11     ensure size = CountElements() && noChange;
12 }
```

Figura 2.4: Exemplo de pré e pós-condições em Spec#

Code Contract - é uma maneira mais moderna que Spec# para utilização de construções de DbC no *framework* .Net. *Code Contract* inclui classes para registrar as restrições (pré, pós-condições e invariantes) ao código C# e um analisador estático para análise em tempo de compilação e execução [1].

O uso de *Code Contract* também permite a geração de documentação da API com as informações do contrato. Estão disponíveis ainda ferramentas para geração de testes unitários para verificação das pré condições.

```
1  class SortedList<T> where T : IComparable
2  {
3    private List<T> list = new List<T>();
4    public void add(T item)
5    {
6      Contract.Requires<ArgumentNullException>(item != null, nameof(
           item));
7      Contract.Ensures(Contract.OldValue<int>(list.Count) + 1 == list
           .Count);
8      list.Add(item);
9      list.Sort();
10   }
11   public void remove(T item)
12   {
13     Contract.Requires<ArgumentNullException>(item != null, nameof(
           item));
14     Contract.Ensures(Contract.OldValue<int>(list.Count) <= list.
           Count);
15     list.Remove(item);
16   }
17   public T get(int i)
18   {
19     Contract.Ensures(Contract.OldValue<int>(list.Count) == list.
           Count);
20     return list[i];
21   }
22
23 }
```

Figura 2.5: Exemplo de pré e pós-condições em Code Contract

3 NEOIDL: LINGUAGEM PARA ESPECIFICAÇÃO DE CONTRATOS REST

3.1 APRESENTAÇÃO

A NeoIDL é uma linguagem específica de domínio (*Domain Specific Language - DSL*) elaborada com o objetivo de possibilitar, em um processo simples, a elaboração de contratos para serviços REST. Em seu projeto, foram considerados os requisitos de concisão, facilidade de compreensão humana, extensibilidade e suporte à herança simples dos tipos de dados definidos pelo usuário.

Além de ser uma linguagem, a NeoIDL é também um *framework* de geração de código que permite, a partir de contratos especificados na própria linguagem, a produção da implementação da estrutura do serviço. Os serviços podem ser construídos em várias linguagens e tecnologias, por meio de *plugins* da NeoIDL, conforme será apresentado neste capítulo.

As próximas subseções apresentam o histórico da NeoIDL, exemplificam sua sintaxe e descrevem sucintamente seu *framework* de geração de código.

3.1.1 Histórico e motivação

A NeoIDL surgiu no contexto de um acordo de colaboração entre a Universidade de Brasília e o Exército Brasileiro. O projeto do exército caracterizava-se pelos requisitos de modularidade – com a lógica distribuída inclusive geograficamente – e de execução em plataformas diversificadas. Diante dessa necessidade, o Exército desenvolveu um *framework* proprietário, voltado para arquitetura orientada a serviço e com suporte a implantação de serviços REST em várias linguagens, denominado NeoCortex.

A particularidade do NeoCortex de se utilizar serviços implementados em várias linguagens motivou o desenvolvimento de um programa gerador de serviços poliglotas – que produz código de várias linguagens de programação – a partir da descrição do contrato do serviço. A NeoIDL foi concebida para atender a essa finalidade. A primeira

decisão de projeto da NeoIDL foi a escolha da linguagem para se especificar o contrato de serviço.

Porém, as linguagens de programação disponíveis para especificação de contratos REST, como Swagger[4], WADL[26] e RAML[3], possuiam (e ainda possuem) limitações importantes para a abordagem desejada, qual seja elaborar primeiramente o contrato e, a partir dele, gerar a implementação do serviço. Todas essas linguagens utilizam notações de propósito geral (XML[5], JSON[2], YAML[8]), tornando os contratos extensos e de difícil compreensão humana. Além disso, elas não possuem mecanismos semânticos de extensibilidade e modularidade.

Partiu-se então para a criação uma nova linguagem, com sintaxe inspirada em linguagens mais claras e concisas – como *CORBA IDL*™[43] e *Apache Thrift*™[51] –, e que permitisse ainda a declaração de tipos de dados definidos pelo usuário e extensibilidade. Ambas, *CORBA* e *Apache Thrift*, possuem limitações nesses últimos aspectos. A sintaxe e as características da linguagem NeoIDL são brevemente discutidas na Subseção 3.1.2.

Em relação à geração de código, para se viabilizar a geração poliglota de código, a NeoIDL foi projetada para possuir uma arquitetura modular, de modo que novas linguagens ou características de implementação pudessem ser incorporadas por meio de *plugins* da NeoIDL. Assim, é possível desenvolver um novo *plugin* para geração de serviços em outras linguagens, por exemplo PHP, sem alterar qualquer outro componente do *framework*, conforme apresentado na Subseção 3.1.3.

A primeira versão da NeoIDL, ponto de onde partiu este trabalho de mestrado, dava suporte à geração de código em Java, Python e Swagger com as características necessárias para execução no NeoCortex. Nessa versão, foram desenvolvidos para o Exército Brasileiro nove serviços do domínio de Comando e Controle [9], os quais compreenderam aproximadamente cinquenta módulos[1] e geração de três mil linhas de código Python a partir da especificação dos contrato em NeoIDL. Alguns outros serviços foram ainda implementados em Java.

[1] O conjunto de documentos utilizados para descrever contratos na NeoIDL são denominados `módulos`.

3.1.2 Linguagem

A linguagem NeoIDL simplifica a especificação de contratos REST pois possui uma sintaxe concisa, própria de linguagens de especificação de interfaces (*interface description languages*). Ademais, a NeoIDL provê mecanismos de modularização e herança, de forma que os contratos possam ser separados em módulos, facilitando a herança e manutenção dos contratos.

Para demonstrar como os módulos são estruturados na NeoIDL, a seguir são apresentados alguns trechos de um serviço hipotético de envio de mensagens. Na primeira parte, um módulo faz uma definição de tipo de dado; em seguida, um segundo módulo, voltado para especificação do serviço em si, importa as definições do primeiro para então declarar as operações do serviço. Por fim, ao módulo de serviço é acrescentada uma anotação como forma de extender as características da operação.

```
1  module MessageData {
2      enum MessageType { Received, Sent };
3
4      entity Message {
5          string id;
6          string from;
7          string to;
8          string subject = 0;
9          string content;
10         MessageType type;
11     };
12 }
```

Figura 3.1: Tipos de dados definidos na NeoIDL

O trecho ilustrado na Figura 3.1 faz a definição de dois tipos de dados. *MessageType*, declarado no linha 2, é uma estrutura simples do tipo enumeração. No exemplo, *MessageType* pode ter os valores *Received* e *Send*. O outro tipo é *Message*, declarado entre as linhas 4 e 11, composto de seis atributos (*id, from, to, subject, content* e *type*). O atributo *type* de *Message* é do tipo *MessageType* recém declarado.

Na NeoIDL, é utilizada a abordagem *convenção sobre configuração*, de modo que todos os atributos declarados são obrigatórios, a menos que seja explicitamente declarado diferente. O atributo *subject* do tipo *Message* é um exemplo de atributo opcional (indicado pelo símbolo de igual seguido de zero).

Definido o tipo de dado, na linha proposta pela NeoIDL para suporte a herança e reuso, o módulo seguinte – conforme apresentado na Figura 3.2 – importa o conjunto de definições de *MessageData* e declara o serviço *sendbox* (linha 4). Esse serviço possui duas capacidades: *sendMessage* e *listMessages*. A capacidade *sendMessage* (linha 6) utiliza a operação `post` para submeter uma mensagem (tipo *Message*). A capacidade *listMessages* (linha 7) tem a finalidade de listar as mensagens com um determinado sequecial, por meio da operação `get`.

A instrução *path* (linha 5) completa a especificação do serviço, indicando o caminho (URI) onde as operações serão disponibilizadas. Esse atribuito é importante para se definir como as requisições serão roteadas entre os serviços.

```
1 module  Message {
2  import MessageData;
3
4  resource sentbox {
5   path = "/messages/sent";
6   @post void sendMessage(Message message);
7   @get [Message] listMessages(string seq);
8  };
9 };
```

Figura 3.2: Sent message service specification in NeoIDL

Seguindo a filosofia de *conversão sobre configuração*, a NeoIDL assume que os argumentos das operações dos tipos `POST` e `PUT` são enviadas no corpo da requisição. Nas operações dos tipos `GET` e `DELETE`, por outro lado, presume-se que os agumentos estão contidos no *path* da requisição ou ainda como *query string*.

A especificação dos contratos na NeoIDL pode ser enriquecida de forma simples com o uso de anotações, pois elas possibilitam extender a semântica de uma especificação sem que seja necessário alterar a sintaxe da linguagem NeoIDL. Essa versatilidade das anotações é bastante útil, pois a alteração da sintaxe da própria NeoIDL não é um esfoço trivial, por demandar a compatibilização de todos os *plugins* já construídos.

O módulo apresentado na Figura 3.3 contém, além das informações contidas no módulo da Figura 3.2, uma anotação denominada *Security Policy* (linhas 4 a 6) que é aplicada ao serviço *sentbox*. A declaração da anotação é feita ao final do módulo (linhas 14 a

18).

```
1  module  Message {
2   import MessageData;
3
4   @SecurityPolicy(method = "basic",
5                   algorithm="AES",
6                   role = "admin");
7
8   resource sentbox {
9    path = "/messages/sent";
10   @post void sendMessage(Message message);
11   @get [Message] listMessages(string seq);
12  };
13
14  annotation SecurityPolicy for resource {
15    string method;
16    string algorithm;
17    string role;
18  };
19
20 };
```

Figura 3.3: Especificação de anotação na NeoIDL

Além de serem aplicáveis a `resources`, as anotações também podem ser utilizadas em outros construtores da linguagem: `module`, `enum`, `entity`. Qualquer anotação na NeoIDL possui a mesma estrutura: um nome, um elemento alvo e uma lista de propriedades. Todas estas informações ficam disponíveis para utilização pelos *plugins*.

A estrutura de um módulo NeoIDL, como pôde ser verificado nos exemplos acima e ilustrada na Figura 3.4, é composta pelas seguintes seções: assinatura do módulo, seção de importação, enumerações e estruturas, serviços e anotações. O apêndice B apresenta a estrutura sintática e léxica da NeoIDL antes de terem sido incorporadas as construções relativas a *Design-by-Contract*.

Figura 3.4: Estrutura de um módulo NeoIDL

3.1.3 *Framework*

A parte da NeoIDL responsável pela geração de código de serviço para as várias linguagens é denominado *framework* NeoIDL. O *framework* NeoIDL possui suas partes: o núcleo e os *plugins*, conforme ilustrado na Figura 3.5. O núcleo é composto de módulos responsáveis (a) por fazer o *parse*[2] do contrato escrito em NeoIDL, (b) por processar a sintaxe da linguagem NeoIDL e (c) pelo gerenciamento dos *plugins*.

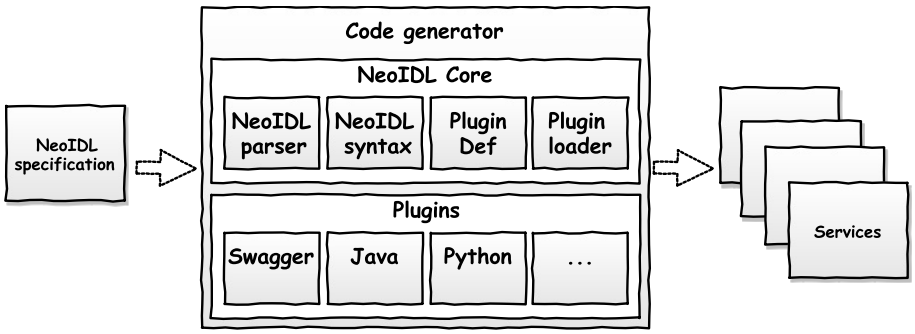

Figura 3.5: Gerador de código da NeoIDL

Fora do núcleo da NeoIDL ficam os *plugins*, responsáveis, cada um, por gerar código para as linguagens de destino. Assim, para se possibilitar a geração de código para uma nova linguagem ou segundo uma nova arquitetura, um novo *plugin* deve ser desenvolvido. A primeira versão da NeoIDL possuia *plugins* para Python, Java e Swagger.

As próximas subseções resumem o funcionamento de dois módulos onde estão contidas os principais trechos da lógica implementada na NeoIDL: O *PluginDef* e *PluginLoader*.

[2] O *parser* da NeoIDL foi construído utilizando `BNFConverter` [47] com a linguagem de programação funcional Haskell.

Mais detalhes podem ser obtidos na publicação *NeoIDL: A Domain Specific Language for Specifying REST Contracts Detailed Design and Extended Evaluation* [35].

3.1.3.1 Componente PluginDef

O módulo `PluginDef` estabelece as definições de regras de projeto (*Design Rules*) obrigatórias para o desenvolvimento de *plugins*, padronizando-os. De acordo com as regras de projeto, cada *plugin* precisa declarar uma instância do tipo *Plugin* e implementar uma função de transformação de acordo com a assinatura definida.

Além disso, cada instância de `Plugin` precisa ter o nome do `plugin` de forma que o componenente `PluginLoader` (Subseção 3.1.3.2) possa obter os dados necessários para o seu processamento. Resumidamente, a execução de um *plugin* consiste em aplicar sua função de transformação a um módulo NeoIDL e produzir uma lista de arquivos de código fonte.

3.1.3.2 Componente PluginLoader

O carregamento e validação dos *plugins* são competências do componente `PluginLoader`. Se todas as regras de definição do *plugin* tiverem sido atendidas, o *plugin* é carregado e estará pronto para ser acionado. Caso contrário, algumas exceções podem ocorrer, como, por exemplo, não haver definição de nenhum *plugin* no arquivo:

```
$./neoIDL
neoIDL: panic! (the ‘impossible’ happened)
 (GHC version 7.6.3 for x86_64-darwin):
   Not in scope: ‘Plugins.Python.plugin’
```

Ou ainda em razão de o *plugin* não ser uma instância do tipo `Plugin`:

```
$./neoIDL
neoIDL: panic! (the ‘impossible’ happened)
(GHC version 7.6.3 for x86_64-darwin):
 Couldn’t match expected type ‘Plugin’
  with actual type ‘[GHC.Types.Char]’
```

3.2 AVALIAÇÃO EMPÍRICA

A primeira versão da NeoIDL foi submetida a um estudo empírico de sua expressividade e reuso em um contexto real. As próximas subseções apresentam o resultado da análise comparativa da representação de 44 (quarenta e quatro) contratos escritos em Swagger v1.2 em relação à mesma especificação em NeoIDL.

Esse estudo é uma das contribuições deste trabalho de mestrado e foi publicado no periódico IJSeke [35].

3.2.1 Expressividade

A NeoIDL é uma DSL, conforme apresentado na Seção 3.1, e como tal se destina a atender a um propósito específico, nem mais, nem menos [29]. A NeoIDL foi projetada para permitir a especificação de contratos de serviços REST de forma mais expressiva e concisa, facilitando-se a escrita e leitura por humanos (mais detalhes na Subseção 3.1.1).

Programas escritos em DSLs costumam ser mais fáceis de escrever e, consequentemente, mais fáceis de se manter comparavelmente a programas escritos em linguagens de propósito geral [29]. Isso se deve justamente ao fato de a DSL tratar apenas um conjunto reduzido de situações e problemas, fazendo com que ela seja, muitas vezes, mais acessível ao público geral [52].

A expressividade é um dos principais critérios para se escolher uma linguagem. Entretanto a linguagem que não expressa todas as situações necessárias ao seu contexto de uso, por óbvio, não pode ser usada [37]. Nesse sentido, o primeiro teste a que a NeoIDL foi submetido constituiu-se na produção de contratos e serviços reais no início do projeto com o Exército Brasileiro (vide Subseção 3.1.1).

Assim, tendo a NeoIDL demonstrado sua capacidade de representar contratos REST reais, foi realizada uma segunda análise: quão expressiva seria a NeoIDL em comparação com outra linguagem com o mesmo objetivo. Foi escolhida Swagger [4], uma linguagem de especificação de contratos REST cujo uso tem crescido pela indústria. Em Swagger, os contratos são escritos em JSON[2] ou YAML[8], ambos com uma estrutura geral de chave-valor.

Sendo a facilidade de compreensão e manipulação por humanos um dos pilares de desenvolvimento da NeoIDL, foi adotada a estratégia de comparar a expressividade em termos de quantidade de linhas de código (SLOC - do inglês *Source Lines of Code*), uma vez que muitas linhas significam maior esforço para escrita, sobretudo na abordagem *Contract-first*.

Com esse propósito, foi obtido um conjunto de 44 contratos do Exército Brasileiro especificados em Swagger. A primeira etapa foi reescrever esses contratos em NeoIDL e então comparar a quantidade de linhas de código produzida com a quantidade de linhas de código dos contratos originais.

Para a transcrição dos contratos, em razão da NeoIDL não possuir recursos para transcrição bidirecional de Swagger para NeoIDL, foi desenvolvido e utilizado um *script* na linguagem *Perl*, o qual consta do Apêndice A desta dissertação.

Os quarenta e quatro contratos em Swagger contabilizaram 13.921 linhas de especificação. Os mesmos contratos especificados em NeoIDL somaram 5.140 linhas de especificação, correspondendo a uma redução média de 63%. Assim, para cada 10 linhas de especificação em Swagger são requeridas 4 linhas de especificação NeoIDL. Nessa análise foram consideradas linhas físicas de código, ignorando-se linhas em branco ou compostas apenas de delimitadores.

A proporção de redução não se deu de forma igual em todos os contratos. Por exemplo, o contrato de um determinado serviço[3] requereu 367 linhas na especificação Swagger e 112 linhas na especificação NeoIDL – redução da ordem de 69%. Em contraponto, outro serviço especificado em Swagger possuia 81 linhas e o correspondente em NeoIDL 42 linhas – redução de linhas de código pouco inferior a 50%.

Estatisticamente, o tamanho original dos contratos tem apenas uma pequena influência na expressividade avaliada. Dessa forma, não é possível assumir que contratos Swagger maiores terão um correspondente proporcionalmente menor em NeoIDL. Outros atributos como documentação mais descritiva, quantidade de entidades e o número de capacidades de cada serviço também não possuem correlação com redução de linha de código maior ou menor após o processo de tranformação para NeoIDL.

A Tabela 3.1 apresenta a correlação entre a melhoria na expressividade observada (me-

[3]Os nomes reais e conteúdos dos contratos foram omitidos em razão de acordo de confidencialidade.

dida como percentual de redução após a transformação da especificação Swagger em especificação NeoIDL) e algumas métricas relacionadas ao tamanho da especificação original em Swagger. Na Correlação Pearson [17], um *p-value* igual a 1 significa correlação positiva perfeita. Um *p-value* igual a -1 significa correlação negativa perfeita, enquanto *p-value* igual a zero indica que as medidas não possuem correlação linear.

Tabela 3.1: Correlação da melhoria de expressividade com o tamanho da especificação em Swagger

Métrica	Correlação Pearson's	*p-value*
LOC da especificação Swagger	0.19	0.20
Número de serviços	0.14	0.35
Numero de capacidades	0.14	0.34
Número de entidades	0.20	0.18

3.2.2 Potencial de reuso

De forma similar à NeoIDL, Swagger também possui recursos para reuso de estruturas, por meio de referências (marcação $ref). Entretanto, esse recurso praticamente não foi explorado no conjunto de contratos analisados, o que ocasinou a duplicação da declaração de estruturas entre os diferentes arquivos de especificação Swagger. Isso se deve, provavelmente, ao modo não intuitivo de se fazer referência em Swagger (baseado na referência a outro arquivo JSON) e à dificuldade de se identificar que uma determinada entidade foi declarada em outro contrato.

No conjunto dos 44 contratos Swagger analisados foram identificadas 42 entidades especificadas em pelo menos dois contratos. Uma entidade específica, de identificação da posição geográfica, muito utilizada no domínio de Comando e Controle, aparece declarada 12 vezes em contratos distintos. A Tabela 3.2 mostra a estatística de repetição da especificação de contratos.

Os resultados do estudo sobre a expressividade e potencial de reuso em Swagger comprovaram, no contexto dos contratos reais avaliados, que a especificação de contratos REST nessa linguagem carece de melhorias importantes. Verificou-se, ainda, que a NeoIDL é mais concisa e expressiva que Swagger para representar as mesmas informações de contratos REST. Sob a ótica do reuso, a abordagem proposta pela

Quantidade de entidades	Quantidade de especificações
420	1
26	2
5	3
7	4
2	5
2	6
1	10
1	12

Tabela 3.2: Estatística de entidades declaradas repetidamente entre contratos

NeoIDL, baseada em importações de especificações de entidades, tende a contribuir para a identificação de entidades já declaradas, facilitando que elas sejam reusadas.

Este capítulo apresentou como foi concebida e como estava estruturada a NeoIDL antes da incorporações de construções de *Design-by-Contract*. Em outras palavras, demonstrou-se o alicerce sobre o qual foram desenvolvidas a extensão da linguagem e o *framework* NeoIDL com suporte a contratos com garantias típicas de *Design-by-Contract*, contribuições que serão detalhadamente descritas no próximo capítulo.

4 CONTRATOS REST COM DESIGN-BY-CONTRACT

4.1 PROPOSTA: SERVIÇOS COM DESIGN-BY-CONTRACT

Os benefícios esperados pela adoção da arquitetura orientada a serviços somente serão auferidos com a concepção adequada de cada serviço. Por essa razão, é necessário planejar o projeto dos serviços criteriosamente antes de lançar mão do desenvolvimento, com preocupação especial em garantir um nível aceitável de estabilidade aos consumidores de cada serviço. Nessa etapa do projeto de desenho da solução, a especificação do contrato do serviço (Web API) exerce uma função fundamental.

Na sociedade civil, contratos são meios de se formalizar acordo entre partes a fim de definir os direitos e deveres de cada parte e buscar atingir o objetivo esperado dentro de determinadas regras. Cada parte espera que as outras cumpram com suas obrigações. Por outro lado, sabe-se que o descumprimeto das obrigações costuma implicar de penalizações até o desfazimento do contrato.

Contratos entre serviços Web seguem em uma linha análoga. O desenho das capacidades (operações) e dos dados das mensagens correspondem aos termos do contrato no sentido do que o consumidor deve esperar do serviço provedor. Porém identificou-se, após ampla pesquisa realizada sobre o tema, que as linguagens disponíves para especificação de contratos atingem apenas esse nível de garantias. No contexto de *Web Services* em REST, conforme descrito na Seção 2.2.2, há ainda a ausência de padrão para especificação contratos.

A proposta deste trabalho é extender os níveis de garantias, de modo a promover um patamar adicional com obrigações mútuas entre os serviços (consumidor e provedor). Isso se dá pela adoção do conceito de *Design-by-Contract* (Seção 2.3) em que a execução da capacidade do serviço garantirá a execução, desde que satisfeitas as condições prévias. As próximas subseções detalham o modo de operação dos serviços com as construções de *Design-by-Contract*.

4.1.1 Modelo de operação

As garantias para execução dos serviços são estabelecidas em duas etapas: pré e pós-condições. Nas precondições, o provedor do serviço estabelece os requisitos para que o serviço possa ser executado pelo cliente. A etapa de pós-condições tem o papel de validar se a mensagem de retorno do serviço possui os resultados esperados.

O diagrama apresentado na Figura 4.1 descreve como ocorre a operação das pré e pós-condições. O processo se inicia com a chamada à capacidade do serviço e a identificação da existência de uma precondição. Caso tenham sido estabelecidas precondições, essas são avaliadas. Caso alguma delas não tenham sido satisfeitas, o serviço principal não é processado e o provedor do serviço retorna o código de falha definido no contrato correspondente.

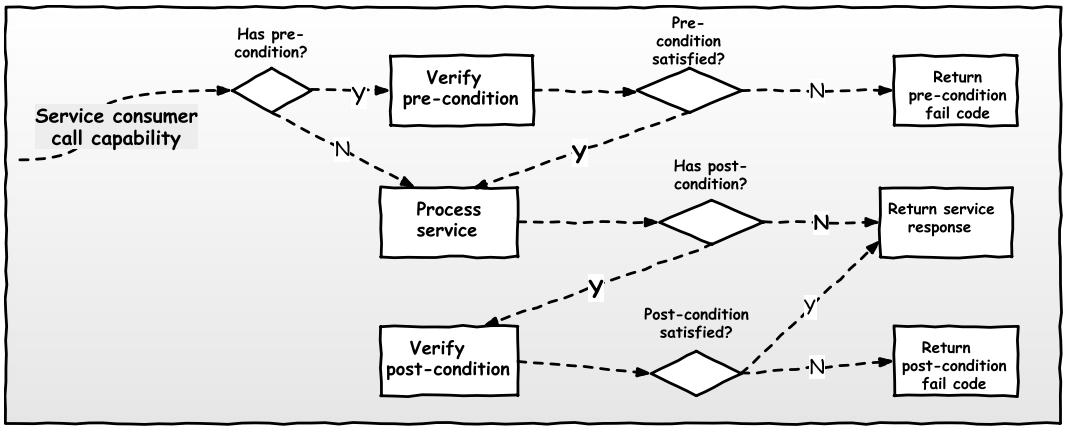

Figura 4.1: Digrama de atividades com verificação de pré e pós condições

Caso tenham sido definidas pós-condições, essas são acionadas após o processamento da capacidade, porém antes do retorno ao consumidor do serviço. Assim, conforme Figura 4.1, visando não entregar ao cliente uma mensagem ou situação incoerente, as pós-condições são validadas. Caso todas as pós-condições tenham sido satisfeitas, a mensagem de retorno é encaminhada ao cliente. Caso contrário, será retornado o código de falha definido para a pós-condição violada.

4.1.1.1 Observação sobre invariantes

Em *Design-by-Contract*, além dos conceitos de pré e pós-condições, há também a ideia de invariantes[40]. Quando aplicadas a uma classe na orientação a objetos, as invariantes estabelecem restrições sobre o estado armazenado nos objetos instanciados dessa classe. No contexto de orientação a serviços, tem-se por princípio a ausência de estados

dos serviços, descrito na Seção 2.1.3. Por essa razão, no estudo sobre a incorporação de *Design-by-Contract* em contratos de serviços, as invariantes não foram consideradas.

4.1.2 Verificação das precondições

As precondições podem ser do tipo baseado nos parâmetros da requisição ou do tipo baseado na chamada a outro serviço. Denomina-se, no contexto desta dissertação, de básica a precondição baseada apenas nos parâmetros da requisição (atributos da chamada ao serviço). Essa validação é direta, comparando os valores passados com os valores admitidos.

No caso das precondições baseadas em serviços, é realizada chamada a outro serviço para verificar se uma determinada condição é satisfeita. Este modo de funcionamento, que se assemelha a uma composição de serviço, é mais versátil, pois permite validações de condições complexas sem que a lógica associada seja conhecida pelo cliente. Assim, os contratos que estabelecem esse tipo de precondição se mantem simples.

A Figura 4.2 apresenta as etapas de verificação de cada precondição. Nota-se que a saída para as situações de desatendimento às precondições, independentemente do tipo, é o mesmo. O objetivo desta abordagem é simplificar o tratamento de exceção no consumidor.

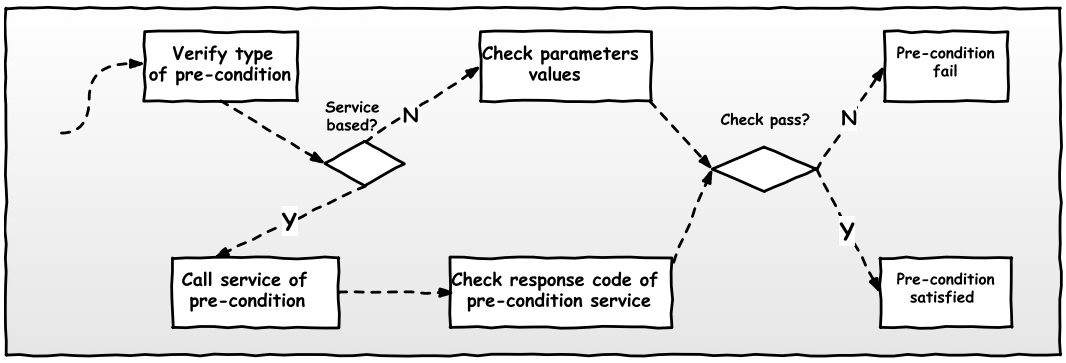

Figura 4.2: Diagrama de atividades do processamento da precondição

4.1.3 Verificação das pós-condições

A verificação das pós-condições acontece de modo muito similar a das precondições. Há também os dois tipos, baseado em valores e em chamadas a outros serviços. O diferencial está em que a validação dos valores passa a ocorrer a partir dos valores

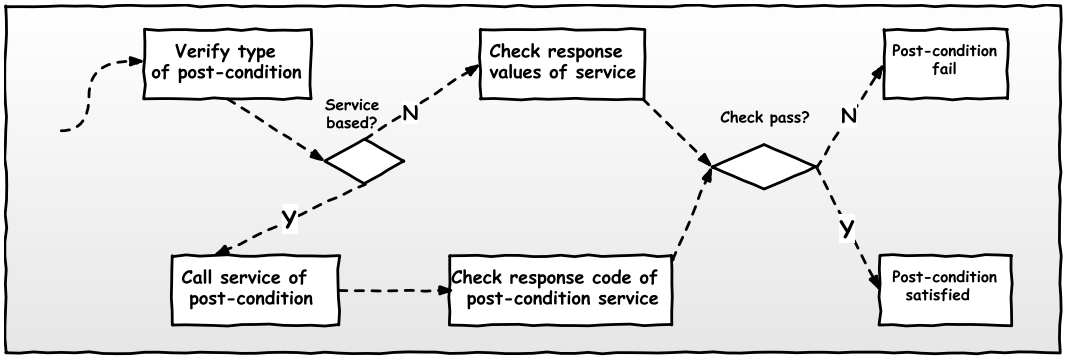

Figura 4.3: Diagrama de atividades do processamento da pós-condição

contidos na mensagem de retorno. A Figura 4.3 descreve as etapas necessárias para validação de cada precondição.

4.2 EXTENSÃO DA NEOIDL PARA DESIGN-BY-CONTRACT

A sintaxe escolhida para possibilitar a especificação de pré e pós condições na NeoIDL foi influenciada por três linguagens e extensões de linguagens de programação: Eiffel, JML e Spec# (exemplificadas na Subseção 2.3.1).

Em Eiffel, as asserções são expressões booleanas, de modo que uma pré e uma pós-condição podem ter resultado verdadeiro ou falso. As asserções também podem incluir chamadas a funções, extendendo a validações a lógicas mais sofisticadas [39]. Essas características, por serem simples e versáteis, foram consideradas adequadas e incorporadas à especificação de *Design-by-Contract* em contratos de serviços na NeoIDL.

A primeira sintaxe de *Design-by-Contract* na NeoIDL teve como base a sintaxe da JML, especialmente em como se associar as pré e pós-condições a cada serviço ou capacidade, assemelhando-se a comentários e iniciados pelo símbolo de arroba (@). A Figura 4.4 apresenta um exemplo de especificação de precondição seguindo a linha da JML.

Essa forma foi apresentada no Workshop de Teses e Dissertações do CBSoft em 2015 [36], ainda nos primeiros estágios do trabalho. Os revisores apontaram dificuldade de distinguir, na especificação, entre as pré e pós-condições e as capacidades, pois possuiam prefixos muito semelhantes (ver linhas 6 a 8). Essas críticas impulsionaram a busca por outra sintaxe mais adequada aos elementos textuais já existentes na NeoIDL.

```
1  module Catalogo {
2      (...)
3      service Catalogo {
4              path = "/catalogo";
5
6              /@require descricao != null
7              /@otherwise HTTP_Precondition_Failed
8              @post Catalogo incluirItem (string id, string descricao
                   , float valor);
9      }
10     (...)
11 }
```

Figura 4.4: Forma preliminar de precondição na NeoIDL

Spec# possui uma forma de especificação de asserções em que as pré e pós-condições são declaradas logo após a assinatura do método ou classe, apenas com o uso das palavras reservadas *require* e *ensure*, sem uso de símbolos. Essa abordagem foi aplicada à NeoIDL para versão final da sintaxe com suporte a *Design-by-Contract*.

As próximas subseções apresentam alguns exemplos de especificação de pre e pós-condições na NeoIDL e as mudanças introduzidas na sintaxe da linguagem. Inicialmente as condições de *Design-by-Contract* são demonstradas separadamente e, ao final, a Subseção 4.2.5 consolida o conjunto de novos elementos sintáticos e como eles são estruturados.

4.2.1 Precondição básica

Uma precondição básica é a que valida os valores recebidos na requisição, comparando-os com os valores estabelecidos na instrução *require* do contrato. Esse tipo de precondição assemelha-se a validação dos atributos recebidos por um método no paradigma de orientação a objetos.

A origem das informações, isto é, onde os valores que serão validados se encontram, depende da operação HTTP utilizada. A Subseção 4.2.6 descreve como esses dados são obtidos. Para realizar a comparação do valor recebido com o valor esperado, a NeoIDL admite seis operadores de comparação (Subseção 4.2.5.2).

A Figura 4.5 (linhas 7 e 8) apresenta um exemplo de precondição básica em uma forma simples, em que apenas um valor é testado (id) e, caso a condição não seja satisfeita, a instrução *otherwise* indica o valor a ser retornado (código HTTP Not Found).

```
1  module store {
2      (...)
3      resource order  {
4        path = "/order/{id}";
5
6        @get     Order getOrder (int id)
7              require (id > "0"),
8              otherwise "NotFound";
9      };
10 }
```

Figura 4.5: Exemplo de notação de precondição básica na NeoIDL

4.2.2 Pós-condição básica

Em termos sintáticos, as pós-condições básicas possuem uma forma muito semelhante às precondições básicas (4.2.1), diferindo-se exclusivamente pelo uso da instrução *ensure*. A Figura 4.6 (linhas 8 e 9) mostra um exemplo de pós-condição básica em que, após a execução da operação `GET`, se o valor do atributo *quantity* não for maior que zero, então o serviço não foi executado adequadamente e a exceção (*otherwise*) é retornada.

```
1  module store {
2      (...)
3
4      resource order  {
5        path = "/order/{id}";
6
7        @get     Order getOrder (int id)
8           ensure (quantity > "0"),
9           otherwise "NoContent";
10     };
11 }
```

Figura 4.6: Exemplo de notação de pós-condição básica na NeoIDL

4.2.3 Precondição com chamada a serviço

As precondições baseadas em serviços seguem uma sequência que envolve a chamada a outro serviço antes do processamento do serviço principal, em um tipo simples de composição de serviço. Essa abordagem permite que precondições complexas sejam validadas por serviços especializados, sem que a especificação do contrato de serviço se torne complexa. Essa proposta preserva ainda a ideia original de Eiffel[38], de que pré e pós-condições sejam expressões *booleanas*.

A primeira etapa do processo de execução da precondição de serviço consiste em fazer a chamada a um serviço (ver Figura 4.2) por meio de uma operação `GET`. Em seguida, o código de *status* retornado pelo serviço da precondição é comparado com o valor especificado na precondição do contrato.

Após o acionamento do serviço da precondição, o comportamento é o mesmo da precondição básica (ver 4.2.1). Caso a precondição seja satisfeita, é retornado o valor indicado pela instrução *otherwise*. As precondições de serviço na NeoIDL admitem os mesmos operadores de comparação que as precondições básicas.

A Figura 4.7 ilustra a especificação de uma precondição do tipo serviço (linhas 8 e 9). Assim, antes de executar a operação `POST` do serviço principal, o serviço *store.getOrder* é acionado. Caso esse serviço retorne o código correspondente a *HTTP Not Found*, a operação `POST` é executada. Caso contrário, o serviço principal retorna o valor correspondente a *HTTP Invalid Precondition*, em razão do estabelecido no *otherwise*.

```
1  module store {
2      (...)
3
4      resource order  {
5        path = "/order/{id}";
6
7        @post    int postOrder (Order order)
8            require (call store.getOrder(id) == "NotFound"),
9            otherwise "InvalidPrecondition"
10     };
11 }
```

Figura 4.7: Exemplo de notação de precondição com chamada a serviço na NeoIDL

4.2.4 Pós-condição com chamada a serviço

A pós-condição com chamada a serviço seguem a sequência de eventos indicada na Figura 4.3. No caso da pós-condição, a execução do serviço principal já ocorreu e a função do serviço na pós-condição é validar se a execução do serviço principal ocorreu com sucesso. Algumas pós-condições são naturais, como as que verificam se um objeto foi inserido após a operação de inclusão (método `POST`). Ou ainda, a que verifica se o objeto foi excluído após uma operação `DELETE`.

No exemplo da Figura 4.8, o serviço principal faz a exclusão de um objeto *Order*. A pós-condição (linha 8) verifica, após o processamento do `DELETE`, se o objeto foi efetivamente apagado por meio do serviço *store.getOrder*. Se o serviço da pós-condição retornar o valor *HTTP Not Found*, o objeto foi adequadamente excluído. Caso contrário, o serviço principal retornará o valor *HTTP Not Modified*, o qual foi estabelecido na instrução *otherwise*.

```
1  module store {
2      (...)
3
4      resource order  {
5        path = "/order/{id}";
6
7        @delete    int deleteOrder (int id)
8            ensure (call store.getOrder(id) == "NotFound"),
9            otherwise "NotModified";
10     };
11 }
```

Figura 4.8: Exemplo de notação de pós-condição com chamada a serviço na NeoIDL

4.2.5 Sintaxe geral de pré e pós-condições

Entre as subseções 4.2.1 e 4.2.4 foram apresentados separadametne exemplos simples de especificação de pré e pós-condições na NeoIDL de modo a facilitar a compreensão. Esta subseção demonstra a estruturação sintática das construções de *Design-by-Contract* agregadas NeoIDL por este trabalho.

4.2.5.1 Listas de pré e pós-condições

Um módulo NeoIDL possui uma seção para declaração dos serviços (ver Figura 3.4). Cada serviço declarado na NeoIDL pode ter um ou mais capacidades, as quais correspondem às operações HTTP utilizadas na arquitetura REST. Sintaticamente, um serviço possui uma lista de capacidades[1]:

```
{Serviço [Capacidade]}
```

A sintaxe da NeoIDL foi extendida para admitir a vinculação de pré e pós-condições às capacidades. Essas contruções de *Design-by-Contract* são opcionais, ou seja, uma capacidade pode não ter nenhuma pré ou pós-condição. Por outro lado, pode-se incluir nas capacidades mais de uma precondição e mais de uma pós-condição ou ainda qualquer combinação delas, simultâneamente.

```
{Capacidade [Condição DbC]}
```

É possível ainda, caso uma precondição ou pós-condição se aplique a todas as capacidades de um serviço, é possível declará-la para o serviço como um todo, logo antes da declaração das capacidades:

```
{Serviço [Condição DbC] [Capacidade]}
```

Assim, generalizando, a NeoIDL passou a suportar a seguinte sintaxe:

```
{Serviço [Condição DbC] [Capacidade [Condição DbC] ]}
```

[1] Os símbolos "[" e "]" identificam uma lista.

4.2.5.2 Tipos de construções de *Design-by-Contract*

Conforme exemplificado entre as subseções 4.2.1 e 4.2.4, as construções de *Design-by-Contract* possuem as seguintes estruturas:

Condição básica: **TipoCondição** {`[Argumento Comparação Valor]`}
Condição serviço: **TipoCondição** {`Serviço (Parâmetro) Comparação ValorSrv`}
Exceção: **Otherwise** {`Valor de Retorno `}

Onde:

TipoCondição indica se é uma precondição (*require*) ou uma pós-condição (*ensure*);

Argumento indica o nome do atributo que será testado;

Comparação indica a operação de comparação que será utilizada. A NeoIDL admite seis operadores de comparação: igualdade (==), diferença (<>), maior (>), maior ou igual (>=), menor (<) e menor ou igual (<=). Eles podem ser aplicados a qualquer tipo de pré ou pós-condição.

Mais de um atributo pode ser testado em uma pré ou pós-condição. Na expressão acima, essa característica é simbolizada com a indicação de uma lista.

Valor corresponde ao valor que será comparado com o **Argumento**. As pré e pós-condições básicas admitem algumas combinações com os operadores *not*, *and* e *or*, formando expressões booleanas e, assim, permitindo estabelecer regras mais abrangentes.

Por exemplo, a precondição apresentada na Figura 4.5 poderia ser escrita como ***require*** *(****not*** *id <= 0)*. Os operadores *and* e *or* são infixos (ex.: ***require*** *((id > 0* ***or*** *id <= -1000))*). O uso do operador *and* produz o mesmo efeito de se declarar duas precondições.

Serviço indica o nome do serviço que será acionado. As informações para indicação concreta da localização do serviço, ou seja, a URL de acionamento, não são indicados nesse atributo de pre e pós-condição. Essas informações dependem do local de implantação do serviço e não convém que estejam declaradas no contrato, sob pena de que mudanças no ambiente de implantação do serviço gerem impacto ao contrato. Entretanto, caso se queira fazer essa vinculação, pode ser criada uma anotação NeoIDL, com essa finalidade, para o serviço.

Parâmetro tem a função de indicar os valores que serão passados para a chamada ao serviço da pré ou pós-condição.

ValorSrv corresponde a um valor que será comparado com o *Status Code HTTP* retornado pelo serviço. Na NeoIDL, esses valores são convencionados como o nome do corresponde ao código HTTP, com palavras justapostas (Ex.: "NotFound"tem o valor do código 404, de HTTP *Not Found*)

Valor de Retorno é utilizado na cláusula *otherwise* para indicar o valor que o serviço principal deve retornar caso uma precondiçaõ ou pós-condição não seja atendida.

4.2.5.3 Um exemplo completo de módulo com *Design-by-Contract*

A Figura 4.9 apresenta um exemplo de módulo NeoIDL de um serviço completo, utilizando alguns dos recursos apresentados na Subseção 4.2.5.2. Esse serviço possui três operações, cada uma com duas condições de *Design-by-Contract*.

A operação `GET` possui uma precondição em que o valor do atributo *id* deve ser maior que zero (linha 8), caso contrário a operação deve retornar o valor correspondente a "NotFound". Uma pós-condição assegura que, se o serviço for executado adequadamente, o valor de *quantity* será maior que zero (linha 9). Se for violada a pós-condição, a operação `GET` retorna o valor "NoContent".

A operação `POST` possui duas precondições (linhas 13 e 14). A primeira, básica, valida o valor do atributo *quantity*. A segunda, aciona o serviço *store.getOrder* com o parâmetro *id*. Nessa operação, foi definido um valor de exceção único para as duas precondições. A NeoIDL associa essa instrução de *otherwise* geral às instruções anteriores que não possuem condição e exceção específica (caso da linha 8).

Na operação `DELETE` foi utilizado o operador lógico *and* na precondição (linha 18). A pós-condição faz a chamada ao serviço *store.getOrder*. Tanto a pré como a pós-condição possuem o mesmo valor de exceção (linha 20).

```
module store {
    (...)

    resource order  {
      path = "/order/{id}";

      @get     Order getOrder (int id)
          require (id > "0") otherwise "NotFound",
          ensure (quantity > "0"),
          otherwise "NoContent";

      @post    int postOrder (Order order)
          require (quantity > "0"),
          require (call store.getOrder(id) == "NotFound"),
          otherwise "InvalidPrecondition";

      @delete    int deleteOrder (int id)
          require ((id <> "" and id > 0)),
          ensure (call store.getOrder(id) == "NotFound"),
          otherwise "NotModified";

    };
}
```

Figura 4.9: Exemplo de módulo NeoIDL com várias instruções de *Design-by-Contract*

4.2.6 Fontes de dados para pré e pós-condições

Os serviços REST na NeoIDL seguem uma convenção em relação ao conteúdo presente na requisição e na resposta para cada tipo de operação *HTTP*, conforme descrito na Subseção 3.1.2. Por exemplo, o método `GET` submete os argumentos pelo *path* ou como *query string* da requisição e não em seu corpo. Esses aspectos foram considerados para se definir a fonte das informações para os argumentos e parâmetros utilizados nas pré e pós-condições.

A Figura 4.10 resume a origem para cada operação. Há diferenças também entre os tipos básico e baseado em serviços das construções de *Design-by-Contract*. As operações `GET` e `DELETE` não possuem argumentos encaminhados no corpo da requisição. Para essas operações, os argumentos são retirados do *path* ou *query string* para valida-

ção das precondições, tanto no tipo básico como no tipo baseado em composição. Na figura, essa origem é identificada como *Request arguments*.

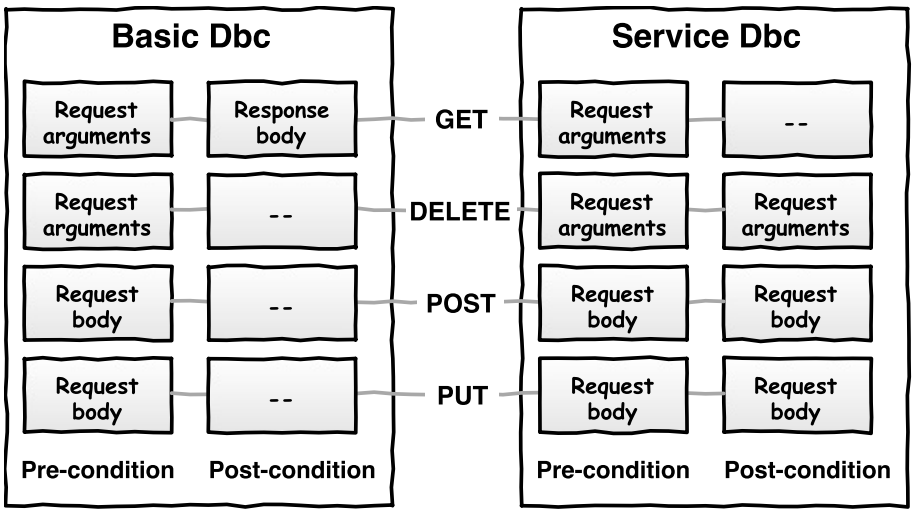

Figura 4.10: Diagrama da fonte de dados para acionamento de pré e pós-condições

Por outro lado, as operações `POST` e `PUT` submetem os dados a serem inseridos ou alterados pelo corpo da requisição (*Request body*), local de onde as precondições básicas e baseadas em serviços devem extrair os parâmetros de validação. A NeoIDL utiliza a notação JSON[2] no corpo das requisições. A representação dos argumentos utiliza o padrão separado por pontos para indicar elementos mais internos (ex. "Pessoa.Nome").

No caso das pós-condições, a origem das informações diverge entre o tipo básicos e o tipo por serviços. A única operação que admite pós-condição básica é a operação `GET`, em que os argumentos de validação são extraídos do corpo da resposta (*Response body*). As demais operações não retornam dados no corpo da requisição, pois estão voltadas para alteração de informações e não para consultas. Além disso, não há utilidade em se validar dados de requisição em uma pós-condição.

Já as pós-condições baseadas em serviços não suportam a operação `GET`. Embora seja tecnicamente possível acionar um serviço com base nos argumentos da requisição, como não se tem modificação nos dados por essa operação, a validação dessas informações pode se dar por meio de serviço deve ser feita nas precondições. Ademais, as pós-condições básicas da operação `GET` cumprem com o objetivo que validar as informações de saída.

A pós-condição da operação `DELETE` pode acionar serviços com base nos argumentos da requisição (*Request arguments*). Um caso típico é de acionar o serviço de consulta para

verificar se o dado foi efetivamente excluído. As operações `POST` e `PUT` podem acionam serviços em pós-condições utilizando as informações do corpo da requisição (*Request body*), uma vez que essas operações não possuem dados no corpo da resposta.

4.3 ESTUDO DE CASO: PLUGIN TWISTED

A incorporação de regras de *Design-by-Contract* aos contratos para serviços REST escritos em NeoIDL elevam a um novo patamar os níveis de garantias com a estabilidade comportamental dos serviços. Nesse sentido, a preocupação em garantir ao cliente que o serviço proverá as informações de que ele necessita aumenta, reforçando os benefícios observados com a prática *Contract-first*. Ou seja, o desenho do serviço considera ainda mais a perspectiva do consumidor do serviço.

A Seção 4.2 tratou do elemento *Contrato*, sobre como ele pode ser escrito em NeoIDL com suporte a construções de *Design-by-Contract*. O contrato, porém, é apenas uma especificação, no sentido de descrever regras e não de torná-las executáveis em si. Todavia, a NeoIDL é, além de uma linguagem formal, um *framework* de geração de código poliglota (Subseção 3.1.1) por meio de *plugins*.

Para se comprovar a viabilidade de se conceber serviços com suporte a pré e pós-condições, foi desenvolvido, no decorrer da pesquisa de mestrado, um *plugin* da NeoIDL que cumpre com tal finalidade. As próximas subseções detalham como é estruturada a arquitetura do serviço gerado e seu comportamento em relação a *Design-by-Contract*. Ao final, alguns aspectos sobre a implementação do próprio *plugin* são descritos.

4.3.1 Visão geral do Python *Twisted*

Adotou-se o *framework Python Twisted*[23] como tecnologia para construção dos serviços com pré e pós-condições. A escolha se deu em razão de *Twisted* possuir uma arquitetura voltada para processamento de requisições de vários tipos de protocolos de rede sobre uma infraestrutura simples e autônoma.

O *Twisted* é baseado em eventos e adota a estratégia de tratamentos de requisições de forma assíncrona, em detrimento ao uso de *threads*. O núcleo do *Twisted* é o *loop*

do objeto *reactor* responsável por aguardar e direcionar o processamento dos eventos (Figura 4.11). Requisições HTTP, como as das chamadas a serviços REST, são tratadas como eventos.

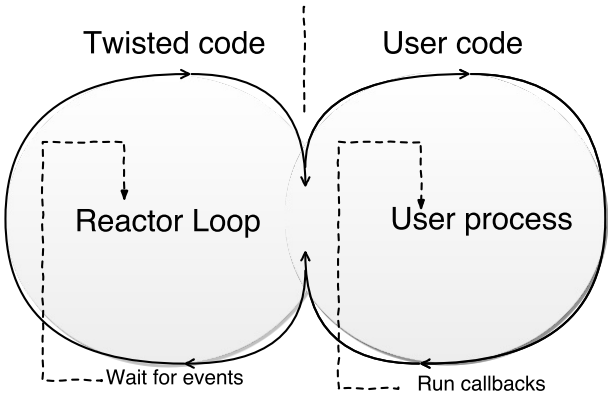

Figura 4.11: Arquitetura assíncrona do *Twisted*

O *reactor* entrega os eventos para serem tratados por processamentos especializados, indicados no lado direito da Figura 4.11. Caso o processamento de um evento seja lento, deve-se disparar um processamento assíncrono, e registrar no *reactor* uma chamada para quando o processamento assíncrono se encerrar, o qual será tratado como um outro evento. Esse controle é feito por um objeto denominado *Deferred*, que contém uma lista de *Callbacks* [23].

4.3.2 Arquitetura dos serviços *Twisted*

Os serviços *Twisted* gerados pela NeoIDL são estruturados em uma arquitetura que extende a arquitetura base do *Twisted Reactor*, incorporando serviços ao processamento das requisições, como ilustrado na Figura 4.12. Os serviços são autônomos entre si, e processam as requisições de acordo com o roteamento realizado pelo servidor.

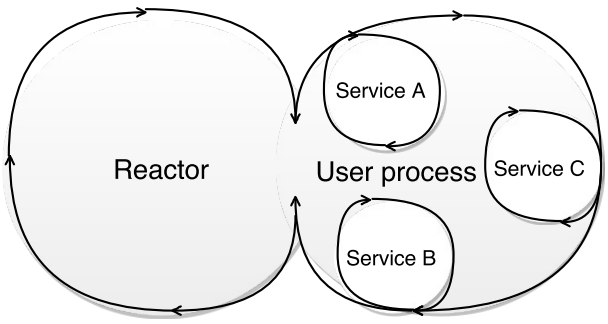

Figura 4.12: Operação dos serviços na arquitetura *Twisted*

Em termos de classes, a NeoIDL gera um conjunto de tres módulos base, que constituem o pacote do *Twisted server*, representados na parte superior da Figura 4.13. Essas classes são fixas, ou seja, não dependem da quantidade nem do conteúdo dos serviços declarados no módulo NeoIDL.

A classe *Server* é o núcleo do *Twisted server*. Ela é responsável por fazer executar o servidor HTTP, receber as requisições, identificar as operações da requisição (`GET`, `POST`, `PUT` ou `DELETE`), e direcionar a requisição para o serviço específico. A identificação do serviço é feita por meio de um arquivo de rota (routes.json), o qual possui uma tradução entre os *paths* das requisições e os serviços responsáveis por cada uma delas.

A classe *Utils* contém um conjunto de funções utilitárias, como a que realiza o *parse* da requisição para extrair os argumentos repassados na URI ou *query string*. Ela também define o objeto que trafega a resposta dos serviços entre os serviço e o servidor.

Essas duas classes são suficientes para o servidor quando não se utiliza *Design-by-Contract* nos serviços. O pacote *DbcConditions* consolida o conjunto de classes responsáveis por processar as pré e pós-condições. A principal função de *DbcConditions* é realizar a comparação entre o valor real e o valor esperado, efetivando toda a lógica corresponde às construções de *Design-by-Contract* descritas na Subseção 4.2.5.2.

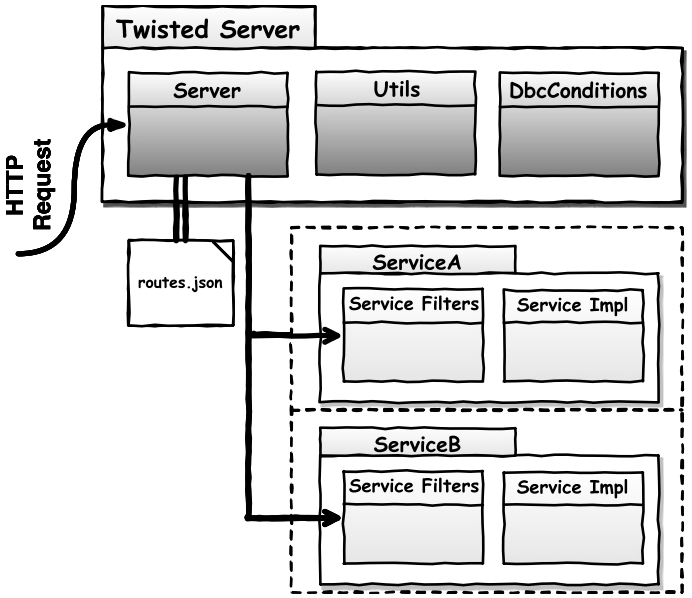

Figura 4.13: Arquitetura do serviço Twisted gerado pela NeoIDL

Em *DbcConditions* estão as funções que carregam a lista de pré e pós-condições para cada serviço. A chamada para as pré e pós-condições baseadas em serviços também é

construída nesse pacote. Esse pacote é fundamental para o funcionamento das construções de *Design-by-Contract* e consta transcrita no Apêndice C.

Para cada serviço declarado no módulo NeoIDL são geradas duas classes: os filtros do serviço e o serviço em si. A classe *ServiceFilter* recebe a requisição; carrega e processa as precondições; aciona o serviço e; ao final, carrega e processa as pós-condições. Esse modo de operação dos filtros é ilustrado na Figura 4.14. A classe do serviço, por fim, contém apenas a estrutura para implementação da lógica interna do serviço.

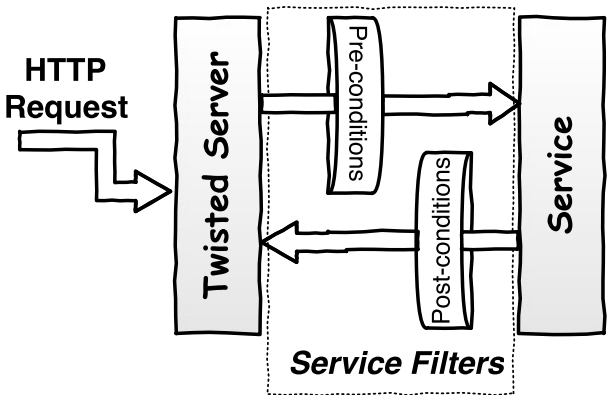

Figura 4.14: Modo de operação das pré e pós-condições no serviço Twisted

4.3.3 Geração de código

O plugin *Twisted* é um conjunto de seis módulos Haskell. Para cada tipo de arquivo gerado, há um módulo no *plugin*, conforme Figura 4.15. Os módulos *PPService*, *PPUtils* e P*PDbcConditions* apenas imprimem um código Python fixo, sem qualquer interferência do módulo NeoIDL processado. *PPRoute* é um pequeno módulo (42 linhas) que processa as informações contidas nas instruções *path* dos serviços declarados no módulo NeoIDL e mapeia a correspondência entre as URIs e os serviços.

O módulo *PPService* gera uma classe com um método para cada operação do serviço. Na versão atual do *plugin*, os métodos são implementados com uma lógica de gravação de objetos em um banco em memória, apenas para simplificar o teste dos código gerado. Essa implementação não é relevante, uma vez que a lógica especializada do serviço real a substituirá.

A parte correspondente ao acionamento das pré e pós-condições é produzido pelo módulo *PPServiceFilter*. O código produzido por este módulo é composto de duas

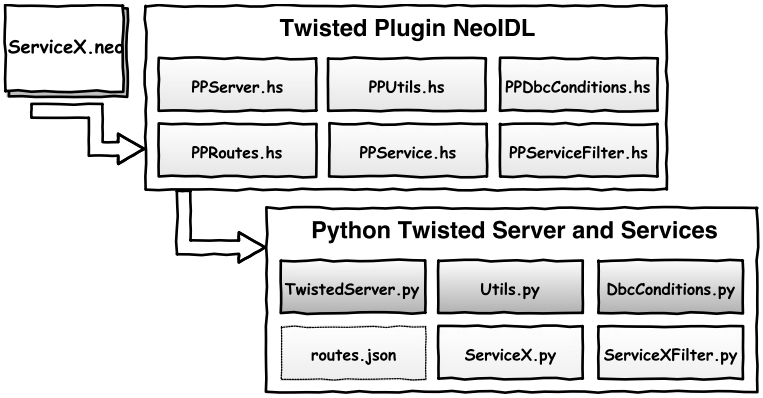

Figura 4.15: Plugin para geração de código Twisted com suporte a *Design-by-Contract*

seções: (i) a declaração das condições de *Design-by-Contract* e (ii) o carregamento dessas condições. A primeira seção, de declaração das condições *Design-by-Contract* contém a lista de pré e pós-condições do serviço. A Figura 4.16 apresenta um exemplo de condição especificada em NeoIDL transformada em código *Python Twisted*.

O lado esquerda dessa figura contém a especificação de uma pós-condição (***ensure***, na linha 2) associada à operação `GET` (linha 1). Constitui-se de uma pós-condição básica, que testa se o valor de resposta *quantity* é maior que zero (linhas 3 a 5). Caso a pós-condição não seja satisfeita, o serviço retorna o código *HTTP No Content*.

```
1  @get    Order getOrder (int id)
2      ensure (
3          quantity
4          >
5          "0"
6      ),
7      otherwise "NoContent";
```

```
1  self.list.append(
2    DbcCheckBasic(
3      'quantity',
4      '>',
5      '0',
6      204,
7      ValuesSource.responseBody,
8      DbcConditionType.PostCondition,
9      OperationType.GET
10   )
11 )
```

Figura 4.16: Transformação de pós-condição NeoIDL (lado esquerdo) em código Python *Twisted* (lado direito)

Do lado direito da Figura 4.16 está o código *Python Twisted* correspondente. O motor de transformação identifica que a condição especificada é uma pós-condição básica (classe *DbcCheckBasic* na linha 2 e tipo na linha 8) associada uma operação `GET` (linha 9). Conforme convenção adotada sobre a fonte de informações (Subseção 4.2.6), a pós-condição é carregada com a indicação de que os argumentos devem ser lidos do corpo da resposta do serviço (linha 7).

Os parâmetros da pós-condição possuem uma correspondência direta, em que as linhas

3 a 5 do lado esquerdo correspondem também às linhas 3 a 5 do lado direito. O valor de exceção da especificação NeoIDL é traduzido no código HTTP correspondente (linha 6). Ambos os códigos apresentados na Figura 4.16 são, na realidade, escritos em uma única linha. O uso de múltiplas linhas foi adotado aqui apenas para efeitos didáticos.

A segunda seção da classe *ServiceFilter* é genérica para qualquer serviço. A Figura 4.17 contém um trecho dessa seção. Cada método se inicia com o carregamento das precondições (linha 10), seguindo pelo acionamento do serviço principal (linha 12). Por fim, as pós-condições são carregadas (linha 14).

```
1  class Resource(object):
2
3      @classmethod
4      def get(cls, result, agent, request, args):
5
6          d = defer.Deferred()
7          dbcCondList = DbcConditionsList()
8
9          # loads pre conditions filters
10         dbcCondList.addFilterCondition(d, OperationType.GET,
               DbcConditionType.PreCondition, agent, request, args)
11         # call Get operation
12         d.addCallback(ServiceXSrv.do_Get,       request, args)
13         # loads post conditions filters
14         dbcCondList.addFilterCondition(d, OperationType.GET,
               DbcConditionType.PostCondition, agent, request, args)
15         # errorBack to return otherwise value if some filter fails
16         d.addErrback (HandleOtherwise.handle, request)
17
18         d.callback(utils.serviceResponse())
19         return d
```

Figura 4.17: Seção de código do filtro de serviços

4.4 ESTUDO EMPÍRICO POR MEIO DE ANÁLISE SUBJETIVA

A construção de *softwares* confiáveis é um dos grandes objetivos da engenharia de *softwares*. O conceito de confiança, porém, não é uma característica geral, mas uma noção relativa: um *software* é correto em relação a uma determinada especificação [13].

Nesse cenário, a qualidade da especificação é fundamental. Em particular, no que tange a relação de confiança entre elementos de *software*, a qualidade do contrato é essencial.

Design-by-Contract surgiu para atender a essa necessidade e tem se mostrado como uma abordagem efetiva no contexto de orientação a objetos desde suas primeiras experimentações [30] ainda na década de 90. A proposta é particularmente útil quando se lida com sistemas distribuídos [13].

O objetivo dessa seção é investigar empiricamente o uso de construções de *Design-by-Contract* no contexto de computação orientada a serviços, um subconjunto da vertente de sistemas distribuídos, verificando a viabilidade e utilidade de sua adoção na especificação de contratos e implementação de serviços REST. Em outras palavras: submeter a uma avaliação subjetiva o objetivo geral desse trabalho de pesquisa.

4.4.1 Planejamento do estudo empírico

Há diversas formas e técnicas para se conduzir um estudo empírico, cada uma com suas vantagens e desvantagens [50]. Nas técnicas diretas, o pesquisador atua de forma explícita e perceptível pela equipe do projeto de *software*, interagindo com eles em maior ou menos grau. As técnicas indiretas são caracterizadas pelo acesso indireto do pesquisador aos participantes, por meio da captura de informações do ambiente de trabalho deles. Há ainda a abordagem independente, em que se analisa apenas o produtos de trabalho, como documentação e código fonte.

As técnicas diretas são divididas em inquisitivas e de observação. *Brainstorming*, grupo focais, entrevistas, questionários e modelagem conceitual estão entre as técnicas inquisitivas. Essas são, muitas vezes, a única forma de obter o engajamento dos envolvidos na realização das atividades. Entretanto, elas tem o risco de serem excessivamente subjetivas e de medição de resultados pouco precisa.

A abordagem observatória está ligada a técnicas como seções de *think-aloud*[2], sombreamento[3] e participante-observador (em que o pesquisar participa realizando atividades chave para o projeto). As técnicas observatórias permitem um estudo em tempo real

[2]Momentos em que os participantes são convidados a expressar seus pensamentos em voz alta ('pensar alto'), a partir de questões levantadas pelo investigador

[3]Consiste em acompanhar o participante na realização de todas as suas atividades

dos fenômenos. Porém, elas levam a uma fase de análise difícil, pois geram muitas informações e requerem um elevado conhecimento para se interpretar corretamente os dados.

No decorrer do trabalho de pesquisa, foram cogitadas algumas estratégias para condução da investigação sobre adoção de *Design-by-Contract* em serviços REST. Entre as alternativas, estava a realização de experimento com alunos de graduação, apresentando especificações textuais com características de *Design-by-Contract* e solicitando a eles a elaboração de especificação correspondente em NeoIDL.

Outra possibilidade de estudo seria, a partir da análise do código fonte de *softwares* de código aberto que utilizam serviços REST escritos em *Swagger*, investigar a existência de comentários ou documentação que caracterizassem situações típicas de pré e pós-condições.

Essas ações foram modeladas conforme a técnica GQM[16] e, após avaliação dos benefícios e riscos, elas foram descartadas por demandarem tempo excessivo em suas etapas de preparação e execução. Dada essa conclusão, adotou-se uma terceira alternativa: aplicação de questionário, que, embora exija cuidado e reflexão sobre a modelagem das perguntas e respostas, é mais rápido de se preparar e permite a participação de pessoas em tempos e locais distintos.

O modelo GQM – *Goals-Questions-Metrics* – é uma abordagem estruturada e documentada na qual se estebelece uma meta para medição de processos e produtos de *software*. A Tabela 4.2 apresenta como as metas são descritas em um modelo GQM, com a finalidade de caracterizá-las objetivamente em termos de objeto, finalidade, foco, ponto de vista e ambiente da análise.

GQM	
Objeto	O processo ou estudo.
Finalidade	Motivação por trás da meta (por que).
Foco	A qualidade do objeto em estudo (o quê).
Ponto de vista	Perspectivas da meta (de quem).
Ambiente	Contexto da aplicação da medição.

Tabela 4.1: Estruturação de metas em GQM

A meta de investigar a utilidade do uso de construções de *Design-by-Contract* na

NeoIDL estruturada conforme o método GQM é apresentada na Tabela 4.2.

GQM	
Analisar	contratos especificados em NeoIDL
Com o propósito de	avaliá-los
Com relação a	facilidade de uso e utilidade das construções de *Design-by-Contract*
Do ponto de vista de	arquitetos e desenvolvedores experientes
No contexto de	um serviço real.

Tabela 4.2: Meta de investigar o uso de Dbc na NeoIDL estruturado conforme método GQM

Em termos de processo, a técnica GQM é organizada em seis etapas: (i) desenvolver o conjunto de metas de medição associadas; (ii) gerar perguntas, baseadas em modelos, que definam os objetivos tão bem quanto possível e de modo quantificável; (iii) especificar as respostas necessárias para essas questões; (iv) desenvolver os mecanismos para coleta de dados; (v) coletar, validar, e analisar a necessidade de ação corretiva; (vi) analisar os dados para avaliar a conformidades com as metas correspondentes.

O modelo escolhido para subsidiar a elaboração das perguntas foi o *Technology Acceptance Model* – TAM, proposto por Davis em 1989 [19] e que possui uma sólida base teórica e ampla utilização. O principal objetivo do modelo TAM é identificar os motivos que levam à aceitação ou rejeição de uma tecnologia. O modelo está estruturado em dois conceitos: percepção da utilidade (*perceived usefullness* - PU) e percepção sobre a facilidade de uso (*perceived ease of use* - PEU) [28].

Percepção de utilidade é medido com o grau em que uma pessoa acredita que utilizar uma determinada tecnologia vai melhorar sua eficiência em suas atividades. A percepção de facilidade está relacionada ao grau em que uma pessoa acredita que determinada tecnologia será livre de esforço. Adicionamente, alguns estudos incluem um outro aspecto, que é a predisposição ao uso (*self-predicted future usage*), que está relacionado à indicação da preferência de uma determinada tecnologia em detrimento de outras [32].

Os questionários baseados em TAM apresentam afirmações indicando que uma determinada tecnologia é fácil de se usar e útil. As respostas normalmente são escalas que vão desde a discordância total até a concordância total quanto a afirmação apresentada. Essa escala é denominada *Likert Scale* [11]. As respostas são agrupadas para se avaliar

se, no conjunto, as respostas mais favoráveis indicam uma tendência de aceitação. O processo de elaboração do questionário é apresentado na Subseção 4.4.2.

A distribuição do questionário foi feita por formulário eletrônico publicado na Internet, utilizando a ferramenta *Google Forms*. Essa ferramenta simplifica o processo de escrita do questionário e também a consolidação das informações. O *link* para acesso ao questionário foi encaminhado a grupos de arquitetos e desenvolvedores cujo nível técnico elevado era de conhecimento dos pesquisadores. O questionário ficou disponível para respostas por um período de três semanas, até que atingisse um volume de respostas satisfatório.

Os resultados obtidos foram consolidados em uma planilha eletrônica e manipulados por meio do ferramental estatístico da linguagem R. A análise dos dados obtidos é apresentada na Subseção 4.4.3.

4.4.2 Modelagem do questionário

Uma vez que a NeoIDL tem como principal objetivo a modelagem de contratos de serviços REST, a avaliação da aceitabilidade da NeoIDL relativamente a outra linguagem com o mesmo propósito pode indicar se o projeto da linguagem está alinhado às expectativas dos desenvolvedores e arquitetos. Entretanto, é necessário avaliar especificamente os efeitos da inclusão de construções de *Design-by-Contract* na NeoIDL, já que não há esse suporte nas outras linguagens para especificação de contratos REST (Subseção 4.1).

O questionário foi organizado em três seções: a primeira seção têm cinco perguntas para traçar o perfil do respondente, principalmente em relação a sua experiência técnica; a segunda seção, com três perguntas, faz uma avaliação sobre especificação formal de contratos e a sintaxe da NeoIDL e de Swagger [4]; Oito questões formam a última seção, a qual dá enfoque ao principal ponto, que é a avaliação das utilidade do uso de *Design-by-Contract* na NeoIDL.

4.4.2.1 Primeira seção do questionário

A Tabela 4.3 apresenta a lista de perguntas da primeira seção. A pergunta sobre o local de prestação de serviços tem a finalidade de identificar e, na fase de análise, eliminar respostas oriundas de locais para os quais o questionário não foi submetido, pois este não exigia autenticação. A segunda e terceira questões visam mapear se o respondente possui experiência e conhecimento suficiente para responder adequadamente o questionário. A questões quatro e cinco pretendem indicar o nível de especialização em *Web Services* REST e na linguagem Swagger.

Perfil técnico-profissional

no.	**Questão**	**Respostas**
1	Para qual órgão ou empresa você presta serviços atualmente?	Questão aberta
2	A quanto tempo você trabalha com desenvolvimento Web?	Experiência, em anos.
3	A quanto tempo você desenvolve com uso de APIs Web (Web Service)?	Experiência, em anos.
4	Qual o seu nível de experiência com especificação de API REST?	Lista A (Tabela 4.4)
5	Qual o seu nível de experiência com especificação de contratos com Swagger?	Lista B (Tabela 4.4)

Tabela 4.3: Questões da primeira seção do questionário que traçam o perfil técnico-profissional

As questões 2 e 3 possuiam uma escala de respostas em que a primeira era não ter experiência, as intermediárias agrupadas a cada dois anos, e a última indicava experiência superior a dez anos. As respostas às questões 4 e 5 estão listadas na Tabela 4.4.

Lista A	Nunca desenvolvi com a tecnologia
	Já desenvolvi serviços algumas APIs REST, mas não conheço a especificação
	Tenho experiência superior a dois anos em desenvolvimento REST, mas não conheço a especificação
	Tenho experiência superior a dois anos em desenvolvimento REST e conheço a especificação
Lista B	Não sei do que se trata ou apenas ouvi a respeito
	Já tive contato com especificações em Swagger
	Já especifiquei APIs REST em Swagger
	Tenho experiência superior a um ano em Swagger

Tabela 4.4: Lista de respostas para as questões 4 e 5

4.4.2.2 Segunda seção do questionário

A partir da segunda seção do questionário, o método TAM foi utilizado para nortear a elaboração das perguntas. O questionário TAM foi adaptado ao cenário de avaliação da aceitação de uma DSL, pois, durante a pesquisa, não foram identificados estudos que utilizam TAM para fazer estudo idêntico. Entretanto, a aplicação de TAM abrange um campo vasto e as questões pode ser adaptadas ao objeto de estudo [14], desde que os aspectos base do método sejam mantidos.

A base da segunda seção do questionário é um conjunto de especificações de serviços. Inicialmente, é apresentada em texto descritivo, a especificação de um serviço que deve recuperar uma lista de informações, desde que atendida uma condição. Em seguida, o mesmo serviço é especificado em Swagger. Na sequência, a especificação é feita em NeoIDL. Ao final, um trecho do código Java que implementa o serviço real é apresentado.

Como a NeoIDL é uma DSL muito pouco conhecida, foram acrescentadas questões sobre a sua expressividade, pois essa característica é considerada o principal critério para a escolha de uma linguagem [37]. O quão fácil é idenficar os elementos da linguagem e compreender o que eles representam é fundamental para se decidir usar uma DSL. Além disso, espera-se de uma DSL, por não ter o compromisso de atender a propósitos gerais, que ela seja mais acessível [52].

Comparação de especificações

no.	**Afirmação**	**Aspecto TAM**
6	A especificação do contrato formalmente, seja em Swagger ou NeoIDL, em relação a descrição textual, aumentará meu nível de acerto na implementação.	Efetividade
7	Identificar e compreender as operações e atributos na especificação Swagger é simples para mim.	Clareza e compreensão
8	Identificar e compreender as operações e atributos na especificação NeoIDL é simples para mim.	Clareza e compreensão

Tabela 4.5: Questões da segunda seção do questionário com comparação de especificações

As questões da segunda seção são apresentadas na Tabela 4.5. Na questão 6, é feita uma afirmação comparando-se a especificação textual com especificação formal, com o objetivo de avaliar a efetividade em se adotar uma linguagem formal. As questões 7 e 8 avaliam a clareza e compreensão das especificações feitas em Swagger e NeoIDL. O objetivo dessas duas questões é fazer uma comparação entre as duas linguagens nesses aspectos. As respostas utilizaram a escala Likert [11] com cinco níveis.

4.4.2.3 Terceira seção do questionário

A terceira seção é a principal, pois enfoca na utilização de construções de *Design-by-Contract* em contratos REST. Essa seção possui uma parte introdutória onde é apresentada uma conceituação de *Design-by-Contract*. Em seguida, é apresentada a especificação de serviço da seção anterior em NeoIDL acrescentando a ela as construções de *Design-by-Contract*. Após essa parte introdutória, são feitas quatro questões (9 a 12) sobre a simplicidade e utilidade do novo contrato em NeoIDL.

Na sequência, é apresentado um trecho curto sobre geração automática de código a partir de especificações formais. O propósito dessa parte é tratar também da utilidade do recurso de geração de código provido pela NeoIDL. Duas perguntas são feitas sobre esse ponto (13 e 14). Por fim, são feitas duas afirmações (15 e 16) sobre a predisposição ao uso futuro da NeoIDL.

Construções de *Design-by-Contract* na NeoIDL

no.	**Afirmação**	**Aspecto TAM**
9	Conhecer previamente e explicitamente as precondições será útil para mim.	Util
10	Aprender a identificar as precondições na NeoIDL parece ser simples pra mim.	Facilidade de aprender
11	Parece ser fácil para mim declarar uma precondição na NeoIDL.	Clareza e compreensão
12	Me lembrar da sintaxe da precondição na NeoIDL é fácil.	Fácil de lembrar
13	É claro e compreensível para mim o efeito da precondição sobre o código gerado.	Controlável
14	A geração do código de pré e pós-condições aumentará minha produtividade na implementação do serviço.	Eficiência
15	Assumindo ter a disposição a NeoIDL no meu trabalho, para especificação de contratos e geração de código, eu presumo que a utilizarei regularmente no futuro.	Adoção
16	Nesse mesmo contexto, eu vou preferir utilizar contratos escritos em NeoIDL do que descritos de outra forma.	Preferência

Tabela 4.6: Questões da terceira seção do questionário sobre *Design-by-Contract* na NeoIDL

4.4.3 Análise dos Resultados

Esta subseção apresenta e discute os resultados do questionário aplicado. As respostas são avaliadas em grupos, iniciando pela avaliação do perfil de respondentes. Em seguida, são avaliadas as questões 6 a 8, sobre a especificação formal de contratos e a expressividade de Swagger e NeoIDL. As respostas às questões 9 a 12, sobre a aplicabilidade de *Design-by-Contract* na especificação de contratos REST, são debatidos na sequência. A penúltimo grupo trata da geração de código com *Design-by-Contract*, com as questões 13 e 14. Por fim, são discutidas as duas últimas questões, sobre predisposição ao uso.

4.4.3.1 Perfil dos respondentes

A primeira questão tinha o objetivo de apenas identificar respostas submetidas por pessoas fora do público alvo. Cinco instituições das quais se conhecia o nível técnico e com as quais se tinha contato com profissionais concentraram a maior parte das respostas. As demais respostas são empresas relacionas a essas instituições, por indicação de

técnicos experientes do primeiro grupo. Sob este aspecto, nenhuma das respostas foi considerada anormal e, portanto, todas foram validadas, totalizando 26 respondentes.

A segunda questão levantou o tempo de experiência dos respondentes com desenvolvimento de sistemas Web. O resultado foi sumarizado na Tabela 4.7. Como o público alvo, definido na meta GQM (Tabela 4.2), é de desenvolvedores experientes, os respondentes que apontaram experiência inferior a três anos estão fora do perfil esperado e suas respostas foram descartadas (linhas em cinza). Assim, o questionário teve um conjunto de 21 respostas válidas.

Experiência com desenvolvimento Web	Q2
Não tenho experiência	2
Entre 1 e 2 anos	3
Entre 3 e 4 anos	1
Entre 5 e 6 anos	3
Entre 7 e 8 anos	5
Entre 9 e 10 anos	1
Há mais de 10 anos	11

Tabela 4.7: Respostas sobre a experiência com desenvolvimento Web

A terceira questão identificou o conhecimento dos respondentes com desenvolvimento de *Web Services*. Conforme apresentado na Tabela 4.8, as respostas estão polarizadas, em que 40% (quarenta por cento) tem nenhuma ou pouca experiência (menos de dois anos). Por outro lado, 23% (vinte e três por cento) dos respondentes possui experiência muito alta (mais de 10 anos). Caso o volume de respostas fosse muito grande (mais de 200 pessoas), poderia ser feita uma análise comparativa da NeoIDL de acordo com a experiência de *Web Service*.

Experiência com *Web Service*	Q3
Não tenho experiência	3
Entre 1 e 2 anos	5
Entre 3 e 4 anos	6
Entre 5 e 6 anos	2
Entre 7 e 8 anos	0
Entre 9 e 10 anos	0
Há mais de 10 anos	5

Tabela 4.8: Respostas sobre experiência com *Web Services*

A questão quatro identificou o conhecimento e experiência do respondente com o modelo arquitetural REST. Apenas dois (menos de 10%) respondentes não tem experiência alguma com desenvolvimento de *Web Services* REST, como pode ser verificado na

Tabela 4.9. Outros 42% (quarenta de dois por cento) dos respondentes, porém, tem experiência e conhecimento sobre os padrões arquiteturais que formam o modelo REST.

Experiência com REST	Q4
Nunca desenvolvi com a tecnologia	2
Já desenvolvi serviços algumas APIs REST, mas não conheço a especificação	7
Tenho experiência superior a dois anos em desenvolvimento REST, mas não conheço a especificação	3
Tenho experiência superior a dois anos em desenvolvimento REST e conheço a especificação	9

Tabela 4.9: Respostas sobre conhecimento e experiência com REST

A última questão sobre o perfil do respondente tratou do conhecimento e experiência com a especificação de contratos em Swagger. O conjunto de respostas confirmou a tendência pelo mercado pela uso de Swagger. Mais de 70% (setenta por cento) dos respondentes já tiveram algum contato com especificação de contratos em Swagger (Tabela 4.10). Esse cenário indica que o conjunto de profissionais é qualificado e possui capacidade crítica para responder às demais questões.

Experiência com Swagger	Q5
Não sei do que se trata ou apenas ouvi a respeito	6
Já tive contato com especificações em Swagger	8
Já especifiquei APIs REST em Swagger	5
Tenho experiência superior a um ano em Swagger	2

Tabela 4.10: Conhecimento e experiência dos respondentes com Swagger

4.4.3.2 Análise da especificação de contratos formais e expressividade de Swagger e NeoIDL

A partir da questão 6, as respostas foram dadas na escala *Likert* sobre as afirmações colocadas. Os dados foram processados com utilização do software estatístico R, com o objetivo de apresentar os resultados por meio de gráficos que facilitam a interpretação de tendência. A cor vermelha é utilizada para indicar discordância sobre as afirmações, sendo que o vermelho mais escuro corresponde a discordância total. Da mesma forma, a cor azul é utilizada para indicar concordância.

Na questão 6 foi apresentada uma afirmação sobre a efetividade de se especificar contratos formalmente, em detrimento de especificação textual descritiva. Conforme apresentado na primeira barra da Figura 4.18, praticamente todos os respondentes concor-

daram com a afirmação e nenhum discordou. Esse resultado reforça a utilidade de se utilizar contratos formais para especificação de serviços.

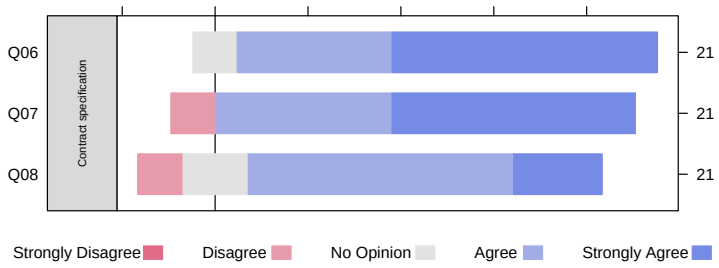

Figura 4.18: Gráfico com o resultado das questões 6 a 8

A questão 7 consultou a percepção dos respondentes sobre a facilidade de identificar as operações e atributos em um contrato especificado em Swagger. A questão 8 contém a mesma afirmação sobre um contrato especificado em NeoIDL. O resultado de ambas questões estão na Figura 4.18 e indicam que tanto os contratos em Swagger como em NeoIDL são claros e fáceis de se compreender. Vale ressaltar que a maior parte dos respondentes já possuia contato anterior com Swagger (Tabela 4.10), o que reforça o resultado positivo para a NeoIDL.

4.4.3.3 Análise sobre aceitabilidade de construções de *Design-by-Contract*

Entre as questões 9 e 12 foram feitas afirmações sobre a facilidade de se identificar os elementos e a utilidade de se especificar pré e pós-condições em contratos de serviços REST com a NeoIDL. Os resultados são apresentados na Figura 4.19.

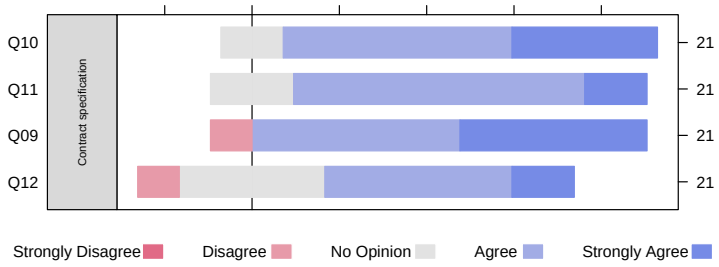

Figura 4.19: Gráfico com o resultado das questões 9 a 12

A questão 9 tem a afirmação de que o conhecimento explícito de precondição terá utilidade para o desenvolvedor responsável por implementar o serviço. Quase todos os respondentes concordaram com a afirmação, em maior e menor grau. Um pequeno

conjunto discordou, indicando que outros elementos podem ser necessários para esse conjunto identificar a utilidade. Pode haver influência ainda da experiência do respondente com implementação de serviços.

A afirmação da questão 10 é sobre a facilidade de se identificar pré e pós-condições em contrato na NeoIDL. Na questão 11, o enfoque é sobre a facilidade de se declarar pré e pós-condições na NeoIDL. Nos dois casos, nenhum dos respondentes manifestou discordância, sendo que a concordância total foi mais intensa para a afirmação da questão 10. Apenas uma pequena parcela informou que nem concorda nem discorda da afirmação.

A questão 12 afirma que é fácil se lembrar da sintaxe de uma precondição na NeoIDL. Essa questão está relacionada a como se pode associar mentalmente as construções sintáticas ao efeito que elas geram. Muito embora se esperasse um resultado mais dividido, por ser uma construção influenciada por linguagens diversas, os resultados indicam que a sintaxe é fácil de ser lembrada.

4.4.3.4 Análise da geração de código para construções de *Design-by-Contract*

Duas questões foram inseridas para tratar da utilidade da geração de código com a NeoIDL para contratos com suporte a *Design-by-Contract*. A questão 13 apontava ser claro o efeito da precondição sobre o código gerado. A maior parte dos respondentes indicou que a geração do código para a precondição é util em relação a produção de efeitos controláveis. A Figura 4.20 apresenta os resultados.

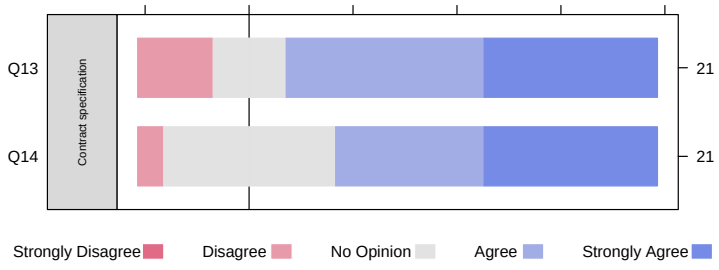

Figura 4.20: Gráfico com o resultado das questões 13 e 14

A questão 14 afirmou que a geração de código sobre precondições ampliará a produtividade de implementação do serviço. O resultado apontou que uma parte expressiva

dos respondentes não concordou nem discordou. Para esses, a especificação de pré-condições não incluencia a produtividade. Por outro lado, tem-se que a maior parte concordou com a contribuição positiva da geração de código das precondições para a produtividade. Uma parcela muito pequena discordou, ou seja, eles acreditam que a geração de código para precondição prejudicará o desempenho.

4.4.3.5 Análise sobre a predisposição ao uso

As duas últimas questões tratam de quão inclinado ao uso da NeoIDL em seu ambiente de trabalho estaria o respondente. Esta análise é relativa [32], ou seja, depende de quais os benefícios trazidos pela NeoIDL em relação aos benefícios trazidos por soluções alternativas, como especificação textual de contratos ou ainda por meio de Swagger. Os resultados para essas questões são apresentadas na Figura 4.21.

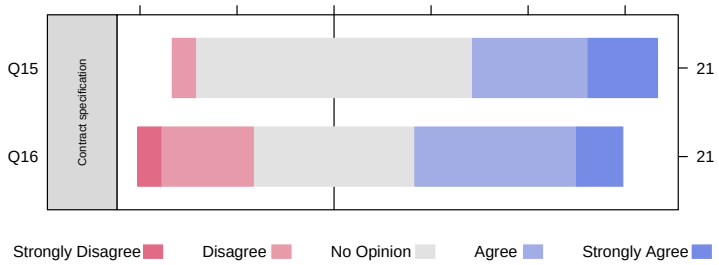

Figura 4.21: Gráfico com o resultado das questões 15 e 16

A questão 15 apresenta uma afirmação sobre a predisposição ao uso regular da NeoIDL no futuro. Pelos resultados da pesquisa, verifica-se que a maior parcela dos respondentes nem concorda nem discorda. Esse resultado é relevante, pois, conforme debatido anteriormente (Tabela 4.10), boa parte dos respondentes já tem contato com Swagger, que consiste em uma alternativa direta à NeoIDL. Se, por um lado os respondentes não concordam prontamente em utilizar NeoIDL, por outro não descartam.

É necessário investigar que fatores influenciariam positivamente a predisposição do uso da NeoIDL e verificar se são fatores internos às empresas em que trabalham atualmente os respondentes ou quais pontos podem ser melhorados na NeoIDL a fim de ampliar o nível de aceitação. Em relação às demais respostas, nota-se uma tendência mais forte de concordância sobre o uso que regular, que a de discordância, o que reforça o potencial de adesão ao uso da NeoIDL.

A última questão abordou diretamente se haveria preferência de uso pela NeoIDL em detrimento a outras linguagens. O resultado é o mais equilibrado do questionário, havendo uma pequena tendência para a preferência pela NeoIDL. O resultado da questão 16 levanta ainda outra hipótese sobre a quantidade de respondentes em dúvida na questão 15: fatores individuais, como não lidar diretamente com especificação de contratos, podem explicar haver um grande grupo que não sabe informar se utilizaria regularmente a NeoIDL.

Assim, essas questões abrem outras perspectivas de estudo, com um enfoque sobre os fatores institucionais e o que mais poderia ser desenvolvido na NeoIDL para atender às necessidades das equipes de desenvolvimento.

4.4.4 Ameaças

Questionários são instrumentos práticos para se realizar estudos empíricos que visam obter a opiniões subjetivas. Uma de suas principais vantagens está em permitir a participação de várias pessoas, ampliando a abrangência temporal e geográfica. Há também mais domínio e facilidade de análise sobre os dados coletados.

Entretanto, algumas ameaças podem influenciar a conclusão dos resultados. Para o questionário aplicado neste trabalho de pesquisa, podemos destacar alguns pontos que podem influenciar ou limitar a qualidade dos dados, bem como direcionar para a realização de outros estudos empíricos sobre o tema.

Um primeiro ponto a observar é a quantidade de respondentes. Embora seja um número superior a de alguns outros estudos avaliados [10] [28] [32], uma maior quantidade de participante ampliaria a qualidade dos dados, pois eliminaria influência de fatores muito específicos a um conjunto de respondentes, e possibilitaria análise de cenários, utilizando visões de dados. O nível técnicos dos respondentes e a distribuição entre mais de quatro entidades, contudo, reduzem os impactos dessa ameaça.

Outro ponto de melhoria são ações que visem proporcionar um conhecimento mínimo sobre a NeoIDL. Todas as respostas foram dadas apenas pelas informações apresentadas no próprio questionário e os resultados poderiam ser diferentes se houvesse um treinamento prévio sobre a NeoIDL. Sob esse aspecto, tem-se que a compreensão sobre dos elementos da sintaxe básica na NeoIDL (questão 6) teve resultado inferior que a

compreensão dos elementos de *Design-by-Contract* (questão 8). Esse resultado é inesperado, pois pressupõe-se que identificar os elementos de *Design-by-Contract* só seja possível após compreender a sintaxe básica.

Ainda, como o questionário foi aplicado apenas uma vez, não foram incluídos pontos de melhoria a partir do primeiro conjunto de respostas para uma segunda avaliação. Conforme debatido na Subseção 4.4.3, algumas questões poderiam ser inseridas em uma nova aplicação do questionário, de modo produzir informações que permitissem conclusões mais completas.

Em estudos futuros, essas melhorias podem ser aplicadas. Em que pese esses pontos de atenção, o estudo realizado por meio do questionário atigiu seus resultados e produziu dados de grande relevância para o objeto em pesquisa.

5 CONCLUSÕES E TRABALHOS FUTUROS

5.1 CONCLUSÕES

A necessidade de estratégias de integração de sistemas e soluções adaptáveis às constantes necessidade des mudanças tem levado às empresas a, cada vez mais, adotarem o modelo de computação orientada a serviços – SOC [45] [22]. A qualidade da especificação do serviço por meio de seu contrato é um dos fatores determinantes para o sucesso do uso de SOC.

A NeoIDL foi criada para ser uma alternativa às linguagens de especificação de *Web Services*REST, uma vez que estas possuem uma sintaxe pouco expressiva para humanos, além de não disporem de mecanismos com suporte a extensibilidade e modularização. O estudo empírico da comparação entre especificações Swagger e NeoIDL demonstrou a capacidade expressiva e potencial de reuso da NeoIDL. Entretanto, nem a NeoIDL, nem as demais linguagens possibilitavam especificar contratos robustos, como os existentes em linguagens com suporte a *Design-by-Contract*.

Esse trabalho apresentou uma extensão da NeoIDL para possibilitar a especificação de contratos REST com suporte a pré e pós-condições e, a partir do contrato, permitir a geração de código de serviços REST com essas garantias, seguindo a abordagem *Contract-first*. A proposta se baseou no paradgima de orientação a objetos, uma das principais influências da orientação a serviços.

Essa proposta foi submetida ao *feedback* de profissionais experientes e também ao *Workshop* de teses de dissertações do WTDSoft 2015. Os elementos sintáticos de linguagens com suporte a *Design-by-Contract* como Eiffel, JML e Spec# foram avaliados e proporcionaram a criação de novas construções sintáticas para a NeoIDL mantendo a harmonia com a linguagem pre-existente, sem que se perdesse o potencial para criação de regras de validação flexíveis e abrangentes.

A NeoIDL passou a permitir a validação dos parâmetros de entrada e dados de saída de uma requisição (por meio de pré e pós-condições básicas) assim como permitir também fazer requisição a outros serviços para realizar validações mais complexas, por meio de

pequenas composições de serviços. A construção de um *Plugin Twisted* demonstrou ser viável produzir código que realize, em tempo de execução, a validação das regras estabelecidas no contrato, sem que o desenvolvedor tenha que se preocupar com elas e direcione seu esforço para a implementação das regras de negócio.

A avaliação subjetiva, realizada por meio de questionário baseado nas técnicas GQM e TAM, apresentou resultados satisfatórios à hipótese de pesquisa, em que grande parte dos respondentes indicou que a NeoIDL com suporte a *Design-by-Contract* é uma ferramenta com potencial de adoção, demonstrando ser uma linguagem e um *framework* úteis e fáceis de serem utilizados sob a perspectiva de quem escreve contratos e implementa serviços.

Ficou demonstrado, no contexto de avaliação desse trabalho, que os conceitos de *Design-by-Contract*, quais sejam, expressar direitos e obrigações entre os clientes e fornecedores de recursos, proporcionando qualidade na análise, projeto, implementação e comunicação, são também aplicáveis no paradigma de orientação a serviços. Ademais, que serviços baseados em *Web Services*REST podem ser simples e leves sob a ótica da especificação mas também serem robustos sobre a ótica da estabilidade e comportamento.

5.2 TRABALHOS FUTUROS

O estudo sobre especificação de contratos para serviços REST é um campo de pesquisa aberto, sobretudo pelo fato de não haver um formato oficial, estabelecido pelo W3C ou outra entidade de padrões internacionais como o IEEE. Este trabalho explorou o aspecto do uso de construções de *Design-by-Contract* em serviços REST que, embora tenha produzido resultados promissores, ainda não fecha a questão por completo.

Ainda sobre a proposta apresentada nessa dissertação, a análise dos resultados (Seção 4.4.3) levanta a possibilidade de exploração de algumas respostas, em especial dos fatores que levaram uma parte importante dos respondentes a informar que tem dúvida (nem discordam nem concordam) com o adoção da NeoIDL em suas atividades. Um novo questionário poderia ser elaborado sobre esse ponto.

A realização de um experimento que avalie o uso por completo da abordagem de pré e pós-condições desde a especificação dos requisitos, a elaboração dos contratos, até

os testes dos serviços implementados em um cenário hipotético poderia trazer mais insumos para a melhoria da NeoIDL. Ainda mais relevante seria a experimentação da NeoIDL com *Design-by-Contract* em um cenário de serviços reais.

Sob a perspectiva de implementação, podem ser elaborados *plugins* para outras linguagens de programação além de *Python Twisted* com suporte a *Design-by-Contract*, de modo a ampliar o potencial de utilização do *framework*. O projeto da NeoIDL também pode incorporar mecanismos que facilitem a especificação de contratos, como recursos ambientes de desenvolvimento (IDEs) e *code complete*. Ainda um portal de onde pudessem ser gerados os códigos sem a necessidade de instalação local do *framework*.

Um outro ponto é exploração do potencial da NeoIDL na disponibilização de recursos de *hypermedia* [53], fornecendo mecanismos em que o cliente possa interagir com o servidor para identificar os serviços que deseja seguir. A tranformação bidirecional entre o código gerado e a especificação do contrato, útil em cenários de manutenção, também pode ser explorada.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] Code contracts. https://msdn.microsoft.com/en-us/library/dd264808(v=vs.110).aspx. Code Contract MSDN site.

[2] Javascript object notation. http://www.json.org. JSON official site.

[3] Raml - the simplest way to design apis. http://raml.org. The Official RAML web site.

[4] Swagger - the world's most popular framework for apis. http://swagger.io/. Swagger project official site.

[5] World wide web consortium (w3c) - extensible markup language (xml). https://www.w3.org/XML. XML W3C official site.

[6] World wide web consortium (w3c) - web services description language. https://www.w3.org/TR/wsdl. WSDL W3C official site.

[7] Xml schema part 2: Datatypes second edition. https://www.w3.org/TR/xmlschema-2. WSDL XML Schema official site.

[8] Yaml ain't markup language. http://yaml.org. The Official YAML web site.

[9] Alberts, D. S. e Hayes, R. E. *Understanding Command and Control*. DoD Command and Control Research Program, 1st edition, 2006.

[10] Albuquerque, D., Cafeo, B., Garcia, A., Barbosa, S., Abrahao, S., e Ribeiro, A. Quantifying usability of domain-specific languages: An empirical study on software maintenance. *Journal of Systems and Software*, 101:245–259, 2015.

[11] Allen, I. E. e Seaman, C. A. Likert scales and data analyses. *Quality Progress*, 40(7):64, 2007.

[12] Alonso, G., Casati, F., Kuno, H., e Machiraju, V. *Web services*. Springer, 2004.

[13] Arnout, K. e Simon, R. The. net contract wizard: Adding design by contract to languages other than eiffel. In *Technology of Object-Oriented Languages and Systems, 2001. TOOLS 39. 39th International Conference and Exhibition on*, pages 14–23. IEEE, 2001.

[14] Babar, M. A., Winkler, D., e Biffl, S. Evaluating the usefulness and ease of use of a groupware tool for the software architecture evaluation process. In *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on*, pages 430–439. IEEE, 2007.

[15] Barnett, M., Leino, K. R. M., e Schulte, W. The spec# programming system: An overview. In *Construction and analysis of safe, secure, and interoperable smart devices*, pages 49–69. Springer, 2004.

[16] Basili, V. R. Software modeling and measurement: the goal/question/metric paradigm. 1992.

[17] Benesty, J., Chen, J., Huang, Y., e Cohen, I. Pearson correlation coefficient. In *Noise reduction in speech processing*, pages 1–4. Springer, 2009.

[18] Chen, H.-M. Towards service engineering: service orientation and business-it alignment. In *Hawaii International Conference on System Sciences, Proceedings of the 41st Annual*, pages 114–114. IEEE, 2008.

[19] Davis, F. D. Perceived usefulness, perceived ease of use, and user acceptance of information technology. *MIS quarterly*, pages 319–340, 1989.

[20] Erl, T. *SOA design patterns*. Pearson Education, 2008.

[21] Erl, T. *Soa: principles of service design*, volume 1. Prentice Hall Upper Saddle River, 2008.

[22] Erl, T., Karmarkar, A., Walmsley, P., Haas, H., Yalcinalp, L. U., Liu, K., Orchard, D., Tost, A., e Pasley, J. *Web service contract design and versioning for SOA*. Prentice Hall, 2009.

[23] Fettig, A. *Twisted network programming essentials*, volume 2. "O'Reilly Media, Inc.", 2013.

[24] Fielding, R. T. *Architectural styles and the design of network-based software architectures*. Tese de Doutorado, University of California, Irvine, 2000.

[25] Forsberg, M. e Ranta, A. Bnf converter. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, Haskell '04, pages 94–95, New York, NY, USA, 2004. ACM.

[26] Hadley, M. J. Web application description language (wadl). 2006.

[27] He, H. What is service-oriented architecture. *Publicação eletrônica em 30/09/2003*, 2003.

[28] Hernandes, E., Zamboni, A., Di Thommazo, A., e Fabbri, S. Avaliação da ferramenta start utilizando o modelo tam e o paradigma gqm. In *Proceedings of 7th Experimental Software Engineering Latin American Workshop (ESELAW 2010)*, page 30, 2010.

[29] Hudak, P. Modular domain specific languages and tools. In *Software Reuse, 1998. Proceedings. Fifth International Conference on*, pages 134–142. IEEE, 1998.

[30] Jazequel, J.-M. e Meyer, B. Design by contract: The lessons of ariane. *Computer*, 30(1):129–130, 1997.

[31] Karthikeyan, T. e Geetha, J. Contract first design: The best approach to design of web services. *(IJCSIT) International Journal of Computer Science and Information Technologies*, 5:338–339, 2014.

[32] Laitenberger, O. e Dreyer, H. M. Evaluating the usefulness and the ease of use of a web-based inspection data collection tool. In *Software Metrics Symposium, 1998. Metrics 1998. Proceedings. Fifth International*, pages 122–132. IEEE, 1998.

[33] Leavens, G. T. e Cheon, Y. Design by contract with jml, 2006.

[34] Leymann, F. Combining web services and the grid: Towards adaptive enterprise applications. In *CAiSE Workshops (2)*, pages 9–21, 2005.

[35] Lima, L., Bonifácio, R., Canedo, E., Castro, T. M.de , Fernandes, R., Palmeira, A., e Kulesza, U. Neoidl: A domain specific language for specifying rest contracts detailed design and extended evaluation. *International Journal of Software Engineering and Knowledge Engineering*, 25(09n10):1653–1675, 2015.

[36] Lima, L. F. Contratos rest robustos e leves: uma abordagem em design-by-contract com neoidl. *WTDSoft 2015*, page 7, 2015.

[37] Mackinlay, J. e Genesereth, M. R. Expressiveness and language choice. *Data & Knowledge Engineering*, 1(1):17–29, 1985.

[38] Meyer, B. Eiffel: A language and environment for software engineering. *Journal of Systems and Software*, 8(3):199–246, 1988.

[39] Meyer, B. Applying'design by contract'. *Computer*, 25(10):40–51, 1992.

[40] Meyer, B. *Object-oriented software construction*, volume 2. Prentice hall New York, 1997.

[41] Mills, H. D. The new math of computer programming. *Communications of the ACM*, 18(1):43–48, 1975.

[42] Mumbaikar, S., Padiya, P., e others,. Web services based on soap and rest principles. *International Journal of Scientific and Research Publications*, 3(5), 2013.

[43] (OMG), O. M. G. Interface definition language 3.5. Technical report, Object Management Group, 2014. http://www.omg.org/spec/IDL35/3.5/PDF/.

[44] Papazoglou, M. P., Traverso, P., Dustdar, S., e Leymann, F. Service-oriented computing: State of the art and research challenges. *Computer*, (11):38–45, 2007.

[45] Papazoglou, M. P., Traverso, P., Dustdar, S., e Leymann, F. Service-oriented computing: a research roadmap. *International Journal of Cooperative Information Systems*, 17(02):223–255, 2008.

[46] Papazoglou, M. P. e Van Den Heuvel, W.-J. Service oriented architectures: approaches, technologies and research issues. *The VLDB journal*, 16(3):389–415, 2007.

[47] Ranta, A. *Implementing Programming Languages. An Introduction to Compilers and Interpreters*. Texts in computing. College Publications, 2012.

[48] Rao, J. e Su, X. A survey of automated web service composition methods. In *Semantic Web Services and Web Process Composition*, pages 43–54. Springer, 2004.

[49] Serrano, N., Hernantes, J., e Gallardo, G. Service-oriented architecture and legacy systems. *Software, IEEE*, 31(5):15–19, 2014.

[50] Shull, F., Singer, J., e Sjøberg, D. I. *Guide to advanced empirical software engineering*, volume 93. Springer, 2008.

[51] Slee, M., Agarwal, A., e Kwiatkowski, M. Thrift: Scalable cross-language services implementation. Technical report, Facebook, 2012. http://thrift.apache.org/static/files/thrift-20070401.pdf.

[52] Taha, W. Domain-specific languages. In *International Conference on Computer Engineering Systems (ICCES)*. Springer, 2008.

[53] Webber, J., Parastatidis, S., e Robinson, I. *REST in practice: Hypermedia and systems architecture*. "O'Reilly Media, Inc.", 2010.

[54] Wideberg, R. Restful services in an enterprise environment - a comparative case study of specification formats and hateoas. Dissertação de Mestrado, Royal Institute of Technology, Stockholm, Sweden, 2015.

APÊNDICES

A TRANFORMAÇÃO SWAGGER PARA NEOIDL

```
1  #include
2  using namespace std;
3  int main()
4  {
5
6  #!/usr/bin/perl
7
8  use JSON;
9
10 use Data::Dumper;
11
12 $arquivo = $ARGV[0];
13 my $model_name = ( split '/', $arquivo )[ -2 ];
14
15 {
16   local $/ = undef;
17   open (FILE, $arquivo) or die "Couldn't open file: $!";
18   binmode FILE;
19   $json_file = <FILE>;
20   close FILE;
21 }
22
23 $text = decode_json($json_file);
24 #$dumped = Dumper($text);
25
26 # cabecalho
27 print "module ".lc($model_name)." {\n";
28 #TODO: Path do modulo nao disponivel no swagger
29 print "\n\tpath = \"xxx\";\n\n";
30
31 for $models (keys $text->{'models'}){
32         print "\tentity ".$models." {\n";
33
34         if (exists $text->{'models'}->{$models}->{'properties'}){
35                 $lista_models = $text->{'models'}->{$models}->{'properties'};
36         } else {
37                 $lista_models = $text->{'models'}->{$models};
38         }
39         for $nome_propriedade (keys $lista_models){
40
41                 if (exists $lista_models->{$nome_propriedade}){
42                         $atributos_propriedade = $lista_models->{$nome_propriedade};
43                 } else {
44                         $atributos_propriedade = $lista_models->{'data'}[0]->{
                            $nome_propriedade};
45                 }
46
47                 #tratamento para o caso de o campo ser array
48                 if (exists $atributos_propriedade->{'type'}){
```

```
49                          if ($atributos_propriedade->{'type'} eq "array"){
50                                  if (exists $atributos_propriedade->{'items'}->{'$ref'
                                        }){
51                                      print "\t\t[".$atributos_propriedade->{'items'
                                            }->{'$ref'}."] : ".
52                                      $nome_propriedade.";";
53                                  } else {
54                                      print "\t\t[".$atributos_propriedade->{'items'
                                            }->{'type'}."] : ".
55                                      $nome_propriedade.";";
56                                  }
57                          } else
58                          {
59                                  print "\t\t".$atributos_propriedade->{'type'}." : ".
60                                        $nome_propriedade.";";
61                          }
62                  } else #nao possui atributo de tipo definido
63                  {
64                          print "\t\t".$atributos_propriedade." : ".$nome_propriedade.";
                                ";
65                  }
66                  #verificar se o dado eh obrigatorio
67                  $comentario = "";
68                  if ($atributos_propriedade->{'required'} != "false") {
69                          $comentario = "Required.";
70                  }
71
72                  if (exists $atributos_propriedade->{'allowableValues'}){
73                          $comentario .= " Valores possiveis: ";
74                          foreach $valores (@{$atributos_propriedade->{'allowableValues'
                                }->{'values'}}){
75                              $comentario .= $valores." ";
76                          }
77                  }
78
79                  #TODO: Descricao dos campos.
80                  if (length($comentario) > 0){
81                          print "   /*".$comentario."*/";
82                  };
83                  print "\n";
84          }
85          print "\t};\n\n";
86  }
87  #=cut
88
89  #imprime a lista de servicos
90  foreach $service (@{$text->{'apis'}}){
91          #Obs: Nome do servico nao disponivel no swagger
92          print "\tservice <NomeServico> {\n";
93          print "\t\tpath = \"".$service->{'path'}."\";\n";
94
95          foreach $capacidades (@{$service->{'operations'}}){
96                  $capacidade_pp = "\t\t\@".lc($capacidades->{'method'}).
97                                   " ".$capacidades->{'type'}.
98                                   #Obs: Nome da capacidade n o dispon vel
99                                   " "."<NomeCapacidade>".
```

```
100                                     " (";
101                     $capacidade_doc_pp = "\n\t\t/**\n";
102                     $capacidade_doc_pp .= "\t\t* \@ summary: ".$capacidades->{'summary'}."
                            \n";
103                     $capacidade_doc_pp .= "\t\t* \@ params: \n";
104                     $parametro_e_tipo = "";
105
106                     foreach $parametros (@{$capacidades->{'parameters'}}){
107                             $parametro_e_tipo .= $parametros->{'type'}." ".$parametros->{'
                                    name'}.", ";
108                             $capacidade_doc_pp .= "\t\t\*\t".$parametros->{'paramType'}."
                                    ".$parametros->{'name'};
109                             if ($parametros->{'required'}) {
110                                     $capacidade_doc_pp .= "(*)";
111                             }
112                             $capacidade_doc_pp .= ": ".$parametros->{'description'}."\n";
113                             if (exists $parametros->{'enum'}){
114                                     $capacidade_doc_pp .= "\t\t\*\t\t\\--> Valores
                                            possiveis: ";
115                                     foreach $enum (@{$parametros->{'enum'}}){
116                                             $capacidade_doc_pp .= $enum." ";
117                                     }
118                                     $capacidade_doc_pp .= "\n";
119                             }
120                     }
121                     $capacidade_pp .= substr($parametro_e_tipo, 0, length (
                            $parametro_e_tipo) - 2);
122                     $capacidade_pp .= ");\n";
123
124                     if (exists $capacidades->{'responseMessages'}) {
125                             $capacidade_doc_pp .= "\t\t* \@ responseCodes: \n";
126                             foreach $respostas (@{$capacidades->{'responseMessages'}}){
127                                     $capacidade_doc_pp .= "\t\t*\t".$respostas->{'code'}.
128                                                     ": ".$respostas->{'message'};
129                                     if (exists $respostas->{'responseModel'}){
130                                             $capacidade_doc_pp .= " --Response Model: ".
                                                    $respostas->{'responseModel'};
131                                     }
132                                     $capacidade_doc_pp .= "\n";
133                             }
134                     }
135
136                     $capacidade_doc_pp .= "\t\t*/\n";
137
138                     print $capacidade_doc_pp;
139                     print $capacidade_pp;
140             }
141             print "\t}\n\n";
142     }
143     #=cut
144
145     print "\n";
146
147      }
```

B ESTRUTURA DA LINGUAGEM NEOIDL

Estas informações foram geradas automaticamente pelo BNF-Converter [25] *parser generator*) a partir da gramática da NeoIDL.

B.1 ESTRUTURA LÉXICA DA NEOIDL

1. Identificadores

   Identificadores ⟨*Ident*⟩ são literais (*strings*) não delimitados que começam com uma letra seguida por letras, números e os caracteres _ ’, exceto palavras reservadas.

2. Literais

   Literais de texto são cadeias de caracteres ⟨*String*⟩ com a forma ‘‘x‘‘, onde x é qualquer sequencia de caracter, exceto ‘‘, a menos que precedido por \.

   Literais numéricos ⟨*Int*⟩ são sequências não vazias de números.

   Literais de ponto flutuantes ⟨*Double*⟩ tem a estrutura definida pela seguinte expressão regular: ⟨*digit*⟩+‘.’⟨*digit*⟩+(‘*e*’‘-’?⟨*digit*⟩+)?, ou seja, duas sequências de números separadas por um ponto, opcionalmente precedida de um símbolo de negativo.

3. Palavras reservadas e símbolos

   O conjunto de palavras reservadas são os terminais da gramática da linguagem NeoIDL. As palavras reservadas não compostas de letras são chamados símbolos, que são tratados de forma diferente dos identificadores. A sintaxe analisador léxico segue regras típicas de linguagens como Haskell, C e Java, incluindo correspondência mais longa e convenções para o espaço.

   As palavras reservadas utilizadas na NeoIDL são as seguintes:

`annotation`	`call`	`entity`
`enum`	`extends`	`float`
`for`	`import`	`int`
`module`	`path`	`resource`
`string`		

Os símbolos utilizados na NeoIDL são os seguintes:

{	}	;
=	.	@
(	)	0
==	<>	>
>=	<	<=
[	]	@get
@post	@put	@delete
/@require	/@ensure	/@invariant
/@otherwise	/**	*/
*	@desc	@param
@consume	,	

B.2 ESTRUTURA SINTÁTICA DA NEOIDL

Não-terminais são delimitados entre ⟨ e ⟩. O símbolo ::= (produto), | (união) e ϵ (regra vazia) advêm da notação BNF. Todos os demais símbolos são terminais.

⟨*Modulo*⟩ ::= `module` ⟨*Ident*⟩ {
⟨*ListImport*⟩
⟨*MPath*⟩
⟨*ListEnum*⟩
⟨*ListEntity*⟩
⟨*ListResource*⟩
⟨*ListDecAnnotation*⟩
}

⟨*Import*⟩ ::= `import` ⟨*NImport*⟩ ;

⟨*MPath*⟩ ::= ϵ
| `path` = ⟨*String*⟩ `;`

⟨*NImport*⟩ ::= ⟨*Ident*⟩
| ⟨*Ident*⟩ `.` ⟨*NImport*⟩

⟨*Entity*⟩ ::= ⟨*ListDefAnnotation*⟩ `entity` ⟨*Ident*⟩ `{` ⟨*ListProperty*⟩ `}` `;`
| ⟨*ListDefAnnotation*⟩ `entity` ⟨*Ident*⟩ `extends` ⟨*Ident*⟩ `{` ⟨*ListProperty*⟩ `}` `;`

⟨*Enum*⟩ ::= `enum` ⟨*Ident*⟩ `{` ⟨*ListValue*⟩ `}` `;`

⟨*DecAnnotation*⟩ ::= `annotation` ⟨*Ident*⟩ `for` ⟨*AnnotationType*⟩ `{` ⟨*ListProperty*⟩ `}` `;`

⟨*DefAnnotation*⟩ ::= `@` ⟨*Ident*⟩ `(` ⟨*ListAssignment*⟩ `)` `;`

⟨*Parameter*⟩ ::= ⟨*Type*⟩ ⟨*Ident*⟩ ⟨*Modifier*⟩

⟨*Assignment*⟩ ::= ⟨*Ident*⟩ = ⟨*Value*⟩

⟨*Modifier*⟩ ::= ϵ
| = `0`

⟨*AnnotationType*⟩ ::= `resource` | `enum` | `entity` | `module`

⟨*Resource*⟩ ::= ⟨*ListDefAnnotation*⟩ `resource` ⟨*Ident*⟩ `{` `path` = ⟨*String*⟩ `;` ⟨*ListCapacity*⟩ `}` `;`

⟨*Capacity*⟩ ::= ⟨*NeoDoc*⟩ ⟨*ListDefNAnnotation*⟩ ⟨*Method*⟩ ⟨*Type*⟩ ⟨*Ident*⟩ `(` ⟨*ListParameter*⟩ `)` `;`

⟨*Method*⟩ ::= `@get` | `@post` | `@put` | `@delete`

C CLASSES DO PACOTE DBCCONDITIONS

```
1  from abc import ABCMeta, abstractmethod
2  from twisted.web.http_headers import Headers
3  from twisted.internet import defer
4  import utils
5
6  import json, re
7
8  # Class to handle the exceptions of dbc conditions
9  class DbcException(Exception):
10     def __init__(self, srvResp):
11         self.serviceResponse = srvResp
12
13 class HandleOtherwise():
14     @classmethod
15     def handle (cls, result, request):
16         request.setResponseCode(result.value.serviceResponse.code)
17         return result.value.serviceResponse
18
19 # Class to define constants used in DbcCondition*
20 class ValuesSource:
21     argument = 'argument'
22     requestBody = 'requestBody'
23     responseBody = 'responseBody'
24
25 class DbcConditionType:
26     PreCondition = 'Pre condition'
27     PostCondition = 'Post condition'
28
29 class OperationType:
30     GET = 'GET'
31     POST = 'POST'
32     UPDATE = 'UPDATE'
33     DELETE = 'DELETE'
34
35 # Abstract class of dbcCheck. This class is implemented by DbcCheckBasic and
       DbcCheckService
36 class DbcCheck(object):
37     __metaclass__ = ABCMeta
38
39     #This method is used to validate the DbCCondition (generic)
40     @abstractmethod
41     def checkCondition(self, result, agent, request, args): pass
42
43     #This method is used in filter of types of dbcCondition (require and ensure) and
           Http method (GET, POST, DELETE, UPDATE)
44     def checkOperationAndType(self, operation, conditionType):
45         if (self.operationType == operation and self.conditionType == conditionType):
46             return True
47         return False
48
```

```
49      #This method compares the value retrived from request/response (argVar) and the
            value expected for dbccondition (self.value)
50      def checkValue (self, argVar):
51          if self.testType == '==': return     argVar  == self.value
52          if self.testType == '<>': return     argVar  <> self.value
53          if self.testType == '>' : return int(argVar) >  int(self.value)
54          if self.testType == '>=': return int(argVar) >= int(self.value)
55          if self.testType == '<' : return int(argVar) <  int(self.value)
56          if self.testType == '<=': return int(argVar) <= int(self.value)
57          return False
58
59      # This util method find a value on a json structure using a string dotted argument
            ('field1.field2')
60      # It supports up to five levels
61      def jsonValueOfStringId (self, jsonBody, stringId):
62          index = stringId.split(".")
63          if len(index) == 1: return str(json.loads(jsonBody)[index[0]])
64          if len(index) == 2: return str(json.loads(jsonBody)[index[0]][index[1]])
65          if len(index) == 3: return str(json.loads(jsonBody)[index[0]][index[1]][index
                [2]])
66          if len(index) == 4: return str(json.loads(jsonBody)[index[0]][index[1]][index
                [2]][index[3]])
67          if len(index) == 5: return str(json.loads(jsonBody)[index[0]][index[1]][index
                [2]][index[3]][index[4]])
68
69
70  class DbcCheckBasic (DbcCheck):
71      """Class to check a basic precondition, i. e., direct check arguments (from
            request or response) values"""
72
73      def __init__(self, arg,  testType, value, excptVal, checkType, conditionType,
            operationType):
74          """Constructor of a basic dbc condition check.
75
76          Keyword arguments:
77          arg -- The name of argument to be compared. (Example: 'fieldId', 'field.id')
78          testType -- Type of comparation. (Example: '==', '<>', '>=')
79          value -- The value to be tested with argument value
80          checkType -- The source of values. It may be request URI, request Body and
                response Body (See ValuesSource())
81          excptVal -- If the dbc condition fail, this value is returned to client (HTTP
                status value)
82          conditionType -- Indicate if is pre or post condition (See DbcConditionType())
83          operationType -- Indicate the type of HTTP operation (See OperationType())
84          """
85          self.arg = arg
86          self.testType = testType
87          self.value = value
88          self.checkType = checkType
89          self.excptVal = excptVal
90          self.conditionType = conditionType
91          self.operationType = operationType
92
93
94      def checkCondition(self, result, agent, request, args):
95          """ Method to check if value expected by dbc condition occurrs """
```

```
96
97          if self.checkType == ValuesSource.argument:
98              """ Case 1: Value of argument is in request URI """
99              if not self.checkValue(args[self.arg]):
100                 request.setResponseCode(self.excptVal)
101                 resp = utils.serviceResponse(self.excptVal, self.conditionType + '
                        failure ('+self.arg+' '+self.testType+' '+self.value+'; Value of
                        attribute '+self.arg+' is '+args[self.arg]+')')
102                 raise DbcException(resp)
103
104
105         elif self.checkType == ValuesSource.requestBody:
106             """ Case 2: Value of argument is in the request body (json format) """
107             requestBody = args['requestContent'];
108
109             if requestBody <> '[]':
110                 valueOfArg = self.jsonValueOfStringId(requestBody, self.arg)
111
112                 if not self.checkValue(valueOfArg):
113                     request.setResponseCode(self.excptVal)
114                     resp = utils.serviceResponse(self.excptVal, self.conditionType + '
                            failure ('+self.arg+' '+self.testType+' '+self.value+'; Value
                            of attribute '+self.arg+' is '+valueOfArg+')')
115                     raise DbcException(resp)
116
117
118         elif self.checkType == ValuesSource.responseBody:
119             """ Case 3: Value of argument is in the response body (json format) """
120             if result.body <> '[]':
121                 valueOfArg = self.jsonValueOfStringId(result.body, self.arg)
122
123                 if not self.checkValue(valueOfArg):
124                     request.setResponseCode(self.excptVal)
125                     resp = utils.serviceResponse(self.excptVal, self.conditionType + '
                            failure ('+self.arg+' '+self.testType+' '+self.value+'; Value
                            of attribute '+self.arg+' is '+valueOfArg+')')
126                     raise DbcException(resp)
127
128         return result
129
130
131
132
133
134 class DbcCheckService (DbcCheck):
135     """Class to check a basic precondition, i. e., direct check arguments (from
            request or response) values"""
136
137     def __init__(self, url, testType, value, excpt, checkType, conditionType,
            operationType):
138         """Constructor dbc condition check based on a service.
139
140         Keyword arguments:
141         url -- The URL of the service. Variable are enclosed by '{' and '}'. (Example:
                'http://localhost/srv1/{id}')
142         testType -- Type of comparation. (Example: '==', '<>', '>=')
```

```
143         value -- The value to be tested with argument value
144         checkType -- The source of values. It may be request URI, request Body and
                response Body (See ValuesSource())
145         excptVal -- If the dbc condition fail, this value is returned to client (HTTP
                status value)
146         conditionType -- Indicate if is pre or post condition (See DbcConditionType())
147         operationType -- Indicate the type of HTTP operation (See OperationType())
148         """
149         self.url = url
150         self.testType = testType
151         self.value = value
152         self.checkType = checkType
153         self.excpt = excpt
154         self.conditionType = conditionType
155         self.operationType = operationType
156 
157     def checkCondition(self, result, agent, request, args):
158         """ Method to check if value expected by dbc condition occurrs """
159 
160 
161         def cbResponse (response, request, getUrl):
162             """ Callback function of request. On Twisted, request are asynchronous """
163             if not self.checkValue(str(response.code)):
164                 request.setResponseCode(self.excpt)
165                 resp = utils.serviceResponse(self.excpt, self.conditionType + '
                        failure (GET '+getUrl+' '+self.testType+' '+self.value+'; Request
                        returns '+str(response.code)+')')
166                 raise DbcException(resp)
167             else:
168                 return result
169 
170         # Arguments to be replaced are enclosed between '{' and '}'
171         splitUrl = re.split('\\{|\\}', self.url)
172 
173         getUrl = ''
174         if self.checkType == ValuesSource.argument:
175             """ Case 1: Value of argument is in request URI """
176             # The odd elements in the splited url are elements to be replaced with its
                    efective values
177             for i in range(0, len(splitUrl)):
178                 if i % 2 == 1:
179                     getUrl += args[splitUrl[i]]
180                 else:
181                     getUrl += splitUrl[i]
182 
183         elif self.checkType == ValuesSource.requestBody:
184             """ Case 2: Value of argument is in the request body (json format) """
185             requestBody = args['requestContent'];
186 
187             if requestBody <> '[]':
188 
189                 # The odd elements in the splited url are elements to be replaced with
                        its efective values
190                 for i in range(0, len(splitUrl)):
191                     if i % 2 == 1:
192                         getUrl += self.jsonValueOfStringId(requestBody, splitUrl[i])
```

```
193                     else:
194                         getUrl += splitUrl[i]
195
196
197         d = agent.request('GET', getUrl, Headers({}), None)
198
199         d.addCallback(cbResponse, request, getUrl)
200         return d
201
202
203
204 class DbcConditionList(object):
205     """ Class that process a list of DbcConditions. Used on service operations filters
            """
206     __metaclass__ = ABCMeta
207
208     @abstractmethod
209     def __init__(self): pass
210
211     def getListByType(self, operation, dbcType):
212         """ Retrives from the list the conditions of a specific type and for a
                specific operation"""
213         rtn = []
214         for dbc in self.list:
215             if dbc.checkOperationAndType(operation, dbcType):
216                 rtn.append(dbc)
217         return rtn
218
219     def addFilterCondition(self, d, operation, dbcType, agent, request, args):
220         """ Add to a callback chain dbc conditions of a specific type and for a
                specific operation """
221         dbc = self.getListByType(operation, dbcType)
222         for f in dbc:
223             d.addCallback(f.checkCondition, agent, request, args)
```