

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 EN-
GENHARIA ELÉTRICA.

APROVADA POR:

Prof. Rodrigo Bonifácio de Almeida, DSc. (ENE-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.

DEDICATÓRIA

Este trabalho é dedicado a ...
continuação

AGRADECIMENTOS

(Página opcional) Agradeço
continuação

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 – o que prejudica a abordagem *Contract-first*, prática estimulada em SOC. Tal limitação motivou a especificação da linguagem NeoIDL¹, 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*, exploradas tipicamente no paradigma de orientação a objetos. *Objetivos.* O objetivo geral deste trabalho é investigar o uso de construções de *Design-by-Contract* 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 pré-condição 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, esta dissertação contribui 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 como maior interesse a análise da percepção de utilidade e facilidade de uso das construções DbC propostas para a NeoIDL, levando a respostas positivas em termos de simplicidade e aceitação dos efeitos sobre código gerado a partir de especificações NeoIDL.

¹Além de ser uma linguagem (*Domain Specific Language*), a NeoIDL também possui um framework de geração de código para outras linguagens de propósito amplo.

ABSTRACT

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

..

..

..

..

SUMÁRIO

1	INTRODUÇÃO	1
1.1	PROBLEMA DE PESQUISA	1
1.2	OBJETIVO GERAL	2
1.3	JUSTIFICATIVA E RELEVÂNCIA	3
1.4	ESTRUTURA	4
2	REFERENCIAL TEÓRICO	5
2.1	COMPUTAÇÃO ORIENTADA A SERVIÇO	5
2.1.1	Terminologia	6
2.1.2	Objetivos, benefícios e características	8
2.1.3	Princípios SOA	10
2.1.4	Contract First	12
2.2	WEB SERVICES	13
2.2.1	SOAP (W3C)	13
2.2.2	REST (Fielding)	14
2.3	DESIGN BY CONTRACT	16
2.3.1	Implementações de DbC	18
3	NEOIDL: LINGUAGEM PARA ESPECIFICAÇÃO DE CONTRA-	
	TOS REST	21
3.1	APRESENTAÇÃO	21
3.1.1	Histórico e motivação	21
3.1.2	Linguagem	22
3.1.3	<i>Framework</i>	25
3.2	AVALIAÇÃO EMPÍRICA	28
3.2.1	Expressividade	28
3.2.2	Potencial de reuso	30
4	CONTRATOS REST COM DESIGN-BY-CONTRACT	31
4.1	PROPOSTA: SERVIÇOS COM DESIGN-BY-CONTRACT	31
4.1.1	Modelo de operação	32

4.1.2	Verificação das precondições	33
4.1.3	Verificação das pós-condições	33
4.2	EXTENSÃO DA NEOIDL PARA DESIGN-BY-CONTRACT	34
4.2.1	Precondição básica	35
4.2.2	Pós-condição básica	36
4.2.3	Precondição com chamada a serviço	37
4.2.4	Pós-condição com chamada a serviço	38
4.2.5	Sintaxe geral de pré e pós-condições	38
4.2.6	Fontes de dados para pré e pós-condições	42
4.3	ESTUDO DE CASO: PLUGIN TWISTED	44
4.3.1	Visão geral do Python <i>Twisted</i>	44
4.3.2	Arquitetura dos serviços <i>Twisted</i>	45
4.3.3	Geração de código	47
4.4	ESTUDO EMPÍRICO DA ANÁLISE SUBJETIVA	49
4.4.1	Método	51
4.4.2	GQM	51
4.4.3	Questionário	51
4.4.4	Análise dos Resultados	51
REFERÊNCIAS BIBLIOGRÁFICAS		53
APÊNDICES		57

LISTA DE TABELAS

3.1	Correlação da melhoria de expressividade com o tamanho da especificação em Swagger	30
-----	--	----

LISTA DE FIGURAS

2.1	Oito princípios da arquitetura orientada a serviços	10
2.2	Exemplo de pré e pós-condições em Eiffel	18
2.3	Exemplo de pré e pós-condições em JML	19
2.4	Exemplo de pré e pós-condições em Spec#	20
3.1	Tipos de dados definidos na NeoIDL	23
3.2	Sent message service specification in NeoIDL	24
3.3	Especificação de anotação na NeoIDL	25
3.4	Estrutura de um módulo NeoIDL	26
3.5	Gerador de código da NeoIDL	26
4.1	Digrama de atividades com verificação de pré e pós condições	32
4.2	Diagrama de atividades do processamento da precondition	33
4.3	Diagrama de atividades do processamento da pós-condição	34
4.4	Forma preliminar de precondition na NeoIDL	35
4.5	Exemplo de notação de precondition básica na NeoIDL	36
4.6	Exemplo de notação de pós-condição básica na NeoIDL	36
4.7	Exemplo de notação de precondition com chamada a serviço na NeoIDL	37
4.8	Exemplo de notação de pós-condição com chamada a serviço na NeoIDL	38
4.9	Exemplo de módulo NeoIDL com várias instruções de <i>Design-by-Contract</i>	42
4.10	Diagrama da fonte de dados para acionamento de pré e pós-condições	43
4.11	Arquitetura assíncrona do <i>Twisted</i>	45
4.12	Operação dos serviços na arquitetura <i>Twisted</i>	45
4.13	Arquitetura do serviço Twisted gerado pela NeoIDL	46
4.14	Modo de operação das pré e pós-condições no serviço Twisted	47
4.15	Plugin para geração de código Twisted com suporte a <i>Design-by-Contract</i>	48
4.16	Transformação de pós-condição NeoIDL (lado esquerdo) em código Python <i>Twisted</i> (lado direito)	48
4.17	Seção de código do filtro de serviços	49

LISTA DE SÍMBOLOS, NOMENCLATURA E ABREVIACÕES

SOC: Software Oriented Computing, modelo arquitetural baseado em serviços.

DbC: Design by Contract, mecanismos de garantias com condições na chamadas a métodos, funções, serviços, etc.

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 [10]. Nessa abordagem, os recursos de software são empacotados como serviços, módulos bem definidos e auto-contidos, que provêm funcionalidades negociais e com independência de estado e contexto [32].

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 [13].

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*.

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*) [5] é considerada uma solução pouco expressiva que 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[15] (*Representational State Transfer*).

Outras alternativas para REST, como Swagger[3] e RAML[2], usam linguagens de propósito geral (em particular JSON[1] e YAML[6]) 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 IDL¹ CORBA[31]) e não oferecem mecanismos semânticos de extensibilidade e modularidade.

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*TM[38] e *CORBA*TM[31].

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*) [27], como Eiffel, JML e Spec#. 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);
- (OE2) Estender 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 possibilitar 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);

¹Interface Description Languages.

(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 heterogêneas 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*[13] 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[4], tem gradativamente se intensificado no sentido da utilização de REST[15].

Uma 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 [19]. A qualidade e precisão do contrato de serviço torna-se claramente um elemento fundamental para que auferir os benefícios da abordagem SOC.

Nesse contexto, REST não dispõe de um meio padrão para especificação de contratos. Linguagens 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 expressividade 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 [13], 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 ESTRUTURA

Este trabalho está organizado em quatro capítulos. . .

2 REFERENCIAL TEÓRICO

2.1 COMPUTAÇÃO ORIENTADA A SERVIÇO

As empresas precisam estar preparadas para responder rapidamente 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 desperdiçado 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 fundamental 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 íntegra e funcional com o cenário de 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 [33].

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, os serviços devem cumprir com determinados quesitos, que incluem alta autonomia ou baixo acoplamento [12]. 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 [34].

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 [22].

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 estendem com novas camadas de projeto, aspectos de governança, e uma grande gama de tecnologias de implementações especializadas, em grande parte baseadas em *Web Service* [13].

Orientação a serviço é um paradigma de projeto cuja intenção é a criação de unidades lógicas moldadas individualmente para podem ser utilizadas conjuntamente 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*[13].

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 TI para a organização. Isso é feito colocando o serviço como elemento central da representação lógica da solução [13].

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 [13], 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 [13].

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 [13].

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 [16], Swagger [3], e NeoIDL [23].

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 é dedicado exclusivamente para cuidar da contratos de serviços uniformes e com qualidade [13].

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 [34].

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 [37]. 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 [11]:

- (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 determinados

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 dos 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 [11].

A segunda característica é a independência de fabricante. O projeto arquitetural que considere 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 compartilhados.

A capacidade de composição é a quarta característica. Os serviços devem ser projetados não somente para serem reusados, mas também possuir flexibilidade para 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 [13], ilustrados na Figura 2.1. São eles que caracterizam a abordagem SOA e a sua aplicação fazem com que um serviço se diferencie de um componente ou de um módulo. Os contratos de serviços permeiam a maior parte destes princípios.

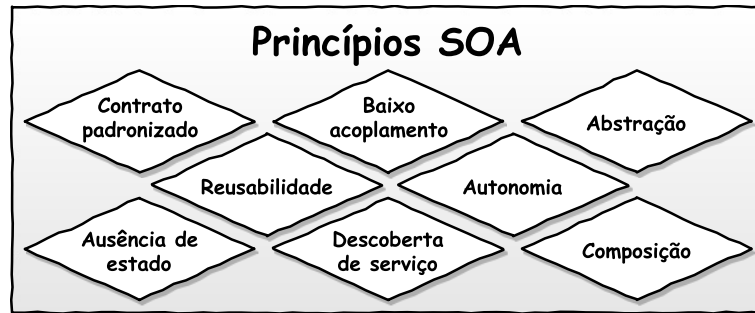


Figura 2.1: Oito princípios da arquitetura orientada a serviços

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 [13].

Baixo acoplamento - 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 inadvertidamente 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 [13]. Ou seja, idealmente a implementação do serviço deve ser derivada do contrato, podendo 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 interpretação 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 acopamentos 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.

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 abordagem *Contract-first* é recomendada para a arquitetura orientada a serviço [13].

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 [20].

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 [20]. 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 e acessadas pela *Web* [8]. 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 [37].

A necessidade das organizações de integrar suas soluções, seja entre os sistemas internos ou entre esses e sistemas de outras empresas [36], 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 [30].

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 partes em um ambiente descentralizado ou distribuído [5]. 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 [37]. SOAP atua como um envelope que transporta a mensagem XML, e possui vastos padrões para transformar e proteger a mensagem e a transmissão.

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 [37]. Além disso a tipagem de dados é mais forte em SOAP que em REST [30].

2.2.2 REST (Fielding)

O termo REST foi criado por Roy Fielding, em sua tese de doutorado [15], 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 [17].

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 [30], que podem ser de vários formatos, como XML e JSON [37]. 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 tráfegadas nas próprias requisições, de modo que as respostas também são independentes. Essas características tornam os *Web Services* REST simples e leves [30].

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 [30], 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 Initiative (OAI)* e denominada de *Open API Specification*. O propósito da iniciativa é tornar *Swagger* padrão para especificação de APIs com independência de fornecedor. Apoiam o projeto grandes empresas como Google®, Microsoft® e IBM®.

WADL (*Web Application Description Language*), uma especificação 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[2] – 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* [41].

2.3 DESIGN BY CONTRACT

Design-by-Contract [27] - 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 fornecem, 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 [26] 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 correteza.

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 [28]. 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 [28] 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 disponíveis 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 [28], *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 [29] 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 [27].

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 [27].

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 [28]. 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.

2.3.1 Implementações de DbC

Eiffel - A linguagem Eiffel foi desenvolvida em meados dos anos 80 por Bertrand Meyer [26] 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 [26]. Esta abordagem influenciou outras linguagem de programação orientadas a objeto.

A figura 2.2 apresenta um exemplo de especificação de pré e póscondição na linguagem Eiffel.

```
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

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 [21].

A figura 2.3 apresenta um exemplo de especificação de pré e póscondição 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 isqrt(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, um método para gerenciar exceções entre outros recursos [9].

A figura 2.4 apresenta um exemplo de especificação de pré e póscondição na linguagem de extensão JML.

```
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#

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 *pluggins* da NeoIDL.

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 vários linguagens, denominado NeoCortex.

A particularidade do NeoCortex de se utilizar serviços implementados em vários linguagens motivou o desenvolvimento um programa gerador de serviços políglotas – 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 linguagem para se especificar o contrato de serviço.

Porém, as linguagens de programação disponíveis para especificação de contratos REST, como Swagger[3], WADL[16] e RAML[2], possuíam (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[4], JSON[1], YAML[6]), 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 *emphCORBA IDL*TM[31] e *Apache Thrift*TM[38] –, 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 [7], os quais compreenderam aproximadamente cinquenta módulos¹ e geração de três mil linhas de código Python a partir da especificação dos contratos em NeoIDL. Alguns outros serviços foram ainda implementados em Java.

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,

¹O conjunto de documentos utilizados para descrever contratos na NeoIDL são denominados *módulos*.

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 estender 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 na 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 (`<Type> <Ident> = 0;`).

Definido o tipo de dado, o módulo seguinte (figura 3.2) , na linha proposta pela NeoIDL para suporte a herança e reuso, 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 atributo 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 do tipos **POST** e **PUT** são enviadas no corpo da requisição. Nas operações dos tipos **GET** e **DELETE**, por outro lado, presume-se que os argumentos 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 estender 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 esforço 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 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 ?? apresenta a estrutura sintática e léxica da NeoIDL antes de terem sido incorporadas as construções relativas a *Design-by-Contract*.

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

Especificação NeoIDL

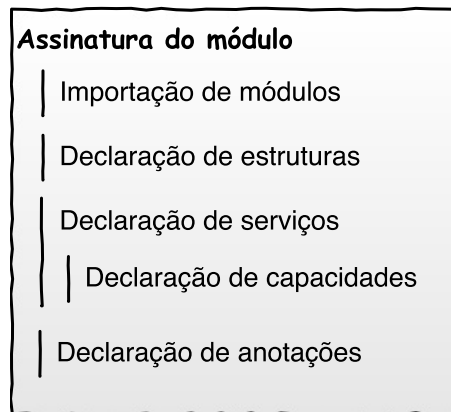


Figura 3.4: Estrutura de um módulo NeoIDL

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*.

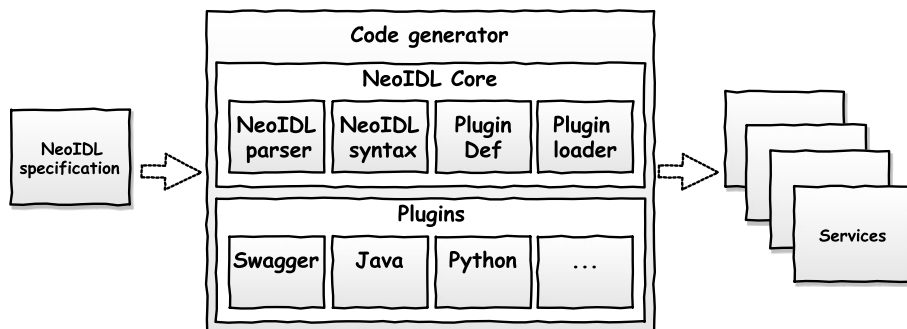


Figura 3.5: Gerador de código da NeoIDL

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 possuía *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*. Mais detalhes podem ser obtidos na publicação *NeoIDL: A Domain Specific Language for Specifying REST Contracts Detailed Design and Extended Evaluation* [23].

²O *parser* da NeoIDL foi construído utilizando *BNFConverter* [35] com a linguagem de programação funcional Haskell.

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 componente `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 em relação à mesma especificação em NeoIDL.

Esse estudo é uma das contribuições deste trabalho de mestrado e foi publicado no periódico IJSeke [23].

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 [18]. 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 [18]. Isso se deve justamente ao fato de a DSL tratar apenas um conjunto reduzido de situações e problemas, fazendo com ela seja, muitas vezes, mais acessível ao público geral [39].

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 [25]. 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 [3], uma linguagem de especificação de contratos REST cujo uso tem crescido pela indústria. Em Swagger, os contratos são escritos em JSON[1] ou YAML[6], 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 este 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 desta dissertação (Apêndice ??).

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 serviço³ requereu 367 linhas na especificação Swagger e 112 linhas na especificação NeoIDL – redução da ordem de 69%. Em contraponto, outro serviço especificado em Swagger possuía 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 transformação para NeoIDL.

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

³Os nomes reais 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, um *p-value* igual a 1 significa correlação positiva perfeita. *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	<i>p-value</i>
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 ocasionou na 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 40 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.

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 descumprimento 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 disponíveis 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 estender os níveis de garantias, de modo a promover um patamar adicional com obrigações mutuas entre os serviços (consumidor e provedor). Isso se dá para adoção do conceito de *Design-by-Contract* (debatido na 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 da 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 retornar o código de falha definido no contrato correspondente.

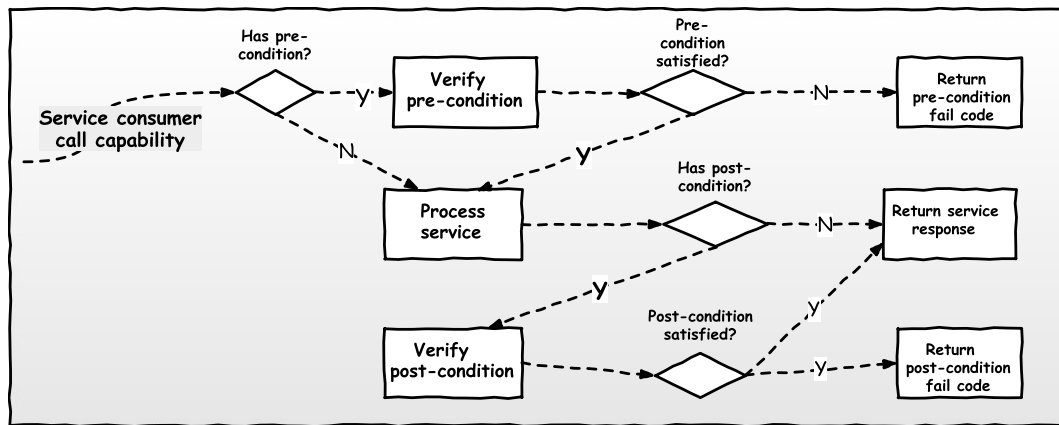


Figura 4.1: Diagrama de atividades com verificação de pré e pós condições

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[28]. 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 detalha 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 tratameto de exceção no consumidor.

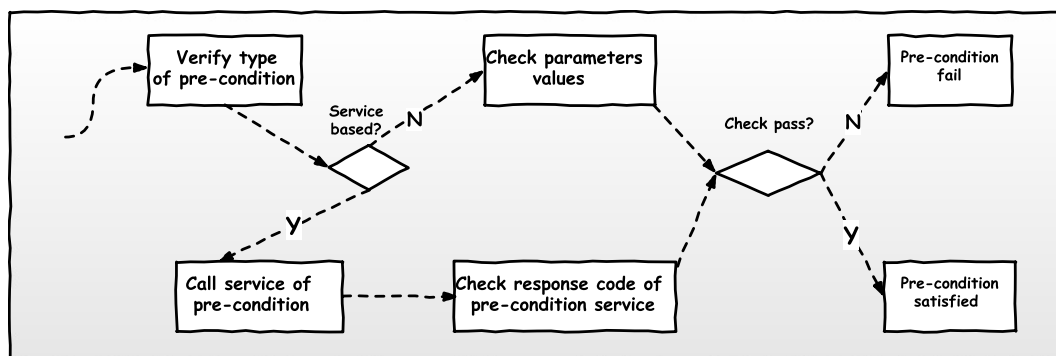


Figura 4.2: Diagrama de atividades do processamento da precondição

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

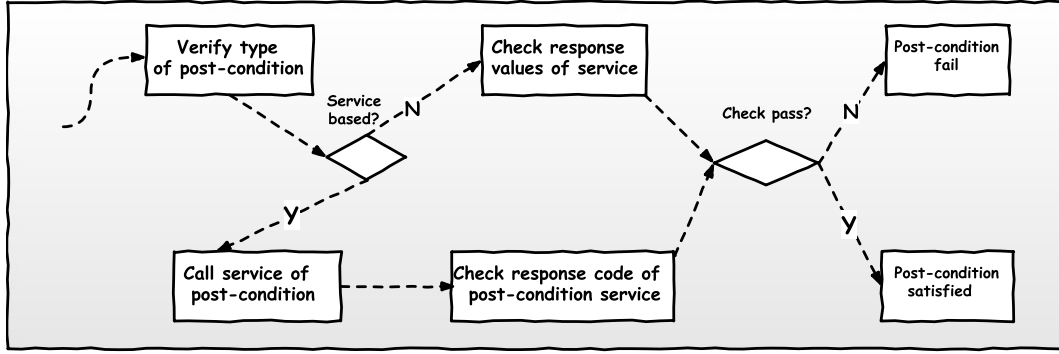


Figura 4.3: Diagrama de atividades do processamento da pós-condição

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

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 [27]. 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 poscondições 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 pré-condição seguindo a linha da JML.

Essa forma foi apresentada no Workshop de Teses e Dissertações do CBSOft em 2015 [24], ainda nos primeiros estágios do trabalho. Os revisores apontaram dificuldade de distinguir, na especificação, entre as pré e poscondições e as capacidades, pois possuíam 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
9             , float valor);
10    }
11    (...)
12 }

```

Figura 4.4: Forma preliminar de precondição na NeoIDL

Spec# possui uma forma de especificação de asserções em que as pré e poscondições 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 poscondições 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 pré-condição 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 pré-condição básica na NeoIDL

4.2.2 Pós-condição básica

Em termos sintáticos, as pós-condições básicas possuem uma forma muito semelhante às pré-condições 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 seja complexa. Essa proposta preserva ainda a ideia original de Eiffel[26], 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 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 separadamente exemplos simples de especificação de pré e 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 mais capacidades, as quais correspondem às operações HTTP utilizadas na arquitetura REST. Sintaticamente, um serviço possui uma lista de capacidades¹:

```
{Serviço [Capacidade]}
```

A sintaxe da NeoIDL foi estendida para admitir a vinculação de pré e pós-condições às capacidades. Essas construções 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 pré-condição e mais de uma pós-condição ou ainda qualquer combinação delas, simultaneamente.

```
{Capacidade [Condição DbC]}
```

É possível ainda, caso uma pré-condição 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] ]}
```

¹Os 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 pré-condição (*require*) ou uma pós-condição (*ensure*);

Argumento indica o nome do atributo que será testado;

Comparação indica o operação de comparação que a ser 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 poscondição.

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 pré-condição 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 pré-condições.

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 pré 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 pré-condição ou pós-condição não seja atendida.

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

A Figura 4.9 apresenta um exemplo 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 pré-condição 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 pré-condições (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 pré-condições. 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 pré-condição (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).

```

1 module store {
2     (...)
3
4     resource order {
5         path = "/order/{id}";
6
7         @get    Order getOrder (int id)
8             require (id > "0") otherwise "NotFound",
9             ensure (quantity > "0"),
10            otherwise "NoContent";
11
12        @post    int postOrder (Order order)
13            require (quantity > "0"),
14            require (call store.getOrder(id) == "NotFound"),
15            otherwise "InvalidPrecondition";
16
17        @delete   int deleteOrder (int id)
18            require ((id <> "" and id > 0)),
19            ensure (call store.getOrder(id) == "NotFound"),
20            otherwise "NotModified";
21
22    };
23 }

```

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 validação das

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

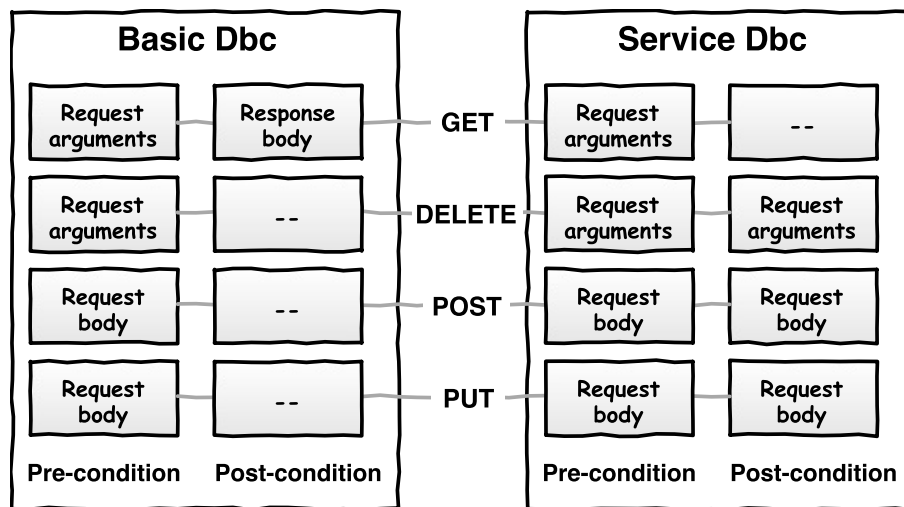


Figura 4.10: Diagrama da fonte de dados para acionamento de pré e pós-condições

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[1] 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 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*[14] 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). Uma requisição HTTP, como as das chamadas a serviços REST, são tratados como eventos.

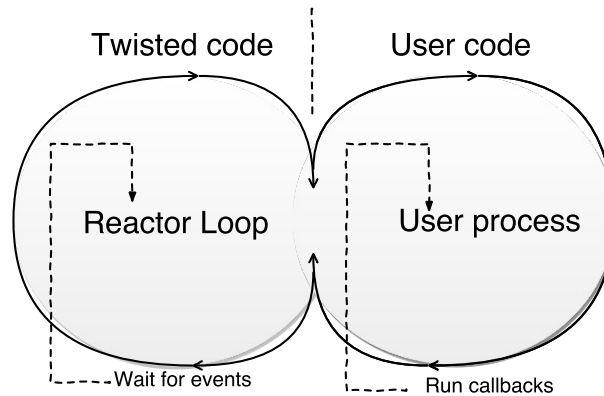


Figura 4.11: Arquitetura assíncrona do *Twisted*

O *reactor* entrega os eventos para serem tratados para 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* [14].

4.3.2 Arquitetura dos serviços *Twisted*

Os serviços *Twisted* gerados pela NeoIDL são estruturados em uma arquitetura que estende 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.

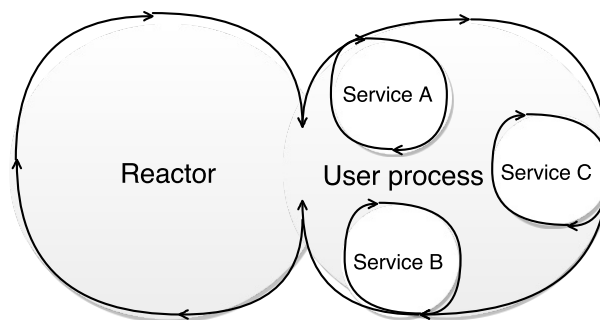


Figura 4.12: Operação dos serviços na arquitetura *Twisted*

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 subir 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.

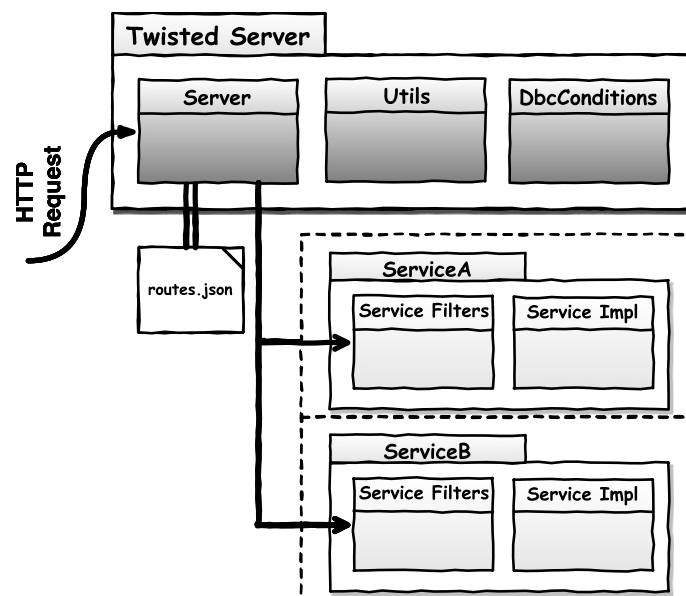


Figura 4.13: Arquitetura do serviço Twisted gerado pela NeoIDL

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. Essa pacote é fundamental para o funcionamento das construções de *Design-by-Contract* e consta transcrita no Apêndice ??.

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.

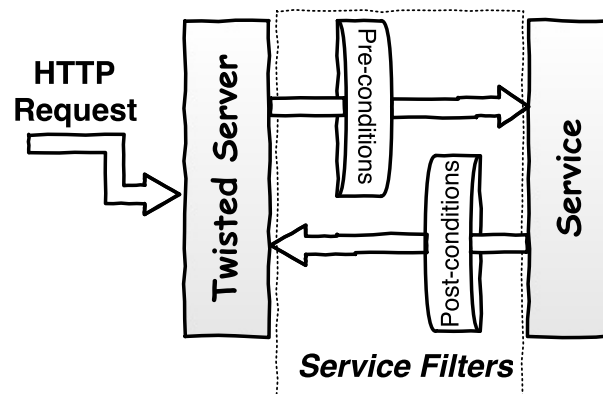


Figura 4.14: Modo de operação das pré e pós-condições no serviço Twisted

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 *PPDbcConditions* 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

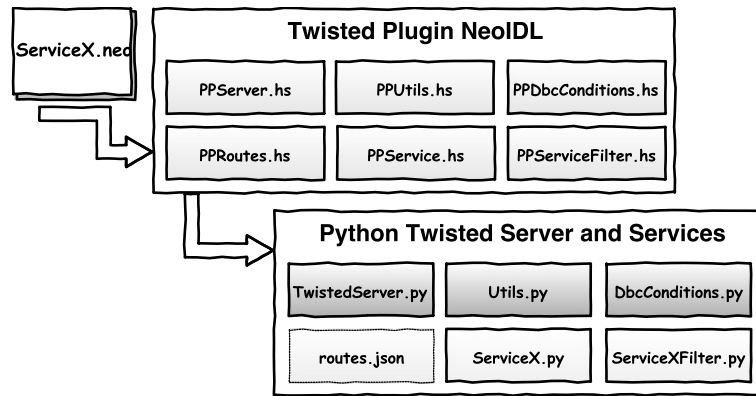


Figura 4.15: Plugin para geração de código Twisted com suporte a *Design-by-Contract*

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 pos-condições do serviço. A Figura 4.16 apresenta um exemplo de condição especificada em NeoIDL transformada em código *Python Twisted*.

O lado esquerdo 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*.

<pre> 1 @get Order getOrder (int id) 2 ensure (3 quantity 4 > 5 "0" 6), 7 otherwise "NoContent"; </pre>	<pre> 1 self.list.append(2 DbcCheckBasic(3 'quantity', 4 '>', 5 '0', 6 204, 7 ValuesSource.responseBody, 8 DbcConditionType.PostCondition, 9 OperationType.GET 10) 11) </pre>
--	---

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,
11                                         DbcConditionType.PreCondition, agent, request, args)
12         # call Get operation
13         d.addCallback(ServiceXSrv.do_Get, request, args)
14         # loads post conditions filters
15         dbcCondList.addFilterCondition(d, OperationType.GET,
16                                         DbcConditionType.PostCondition, agent, request, args)
17         # errorBack to return otherwise value if some filter fails
18         d.addErrback(HandleOtherwise.handle, request)
19
20         d.callback(utils.serviceResponse())
21         return d

```

Figura 4.17: Seção de código do filtro de serviços

4.4 ESTUDO EMPÍRICO DA ANÁLISE SUBJETIVA

...

...

A language's expressiveness is the major criterion for choosing a language to state a given set of facts: a language that cannot express the facts should not be used. However, additional criteria are needed to choose among languages that are sufficiently expressive for a set of facts. Two of these criteria are how easy it is to state the facts in the language and how easy is to perceive the facts once they are stated.

expressividade de uma linguagem é o principal critério para a escolha de uma linguagem para indicar um determinado conjunto de fatos: uma linguagem que não podem expressar os fatos não devem ser usados. No entanto, os critérios adicionais são necessários para escolher entre os idiomas que são suficientemente expressivo para um conjunto de fatos. Dois desses critérios são quão fácil é expor os fatos na linguagem e como é fácil de perceber os fatos, uma vez que são demonstrados.

[25]

Instead of aiming to be the best for solving any kind of computing problem, DSLs aim to be particularly good for solving a specific class of problems, and in doing so they are often much more accessible to the general public than traditional programming languages. [39]

They offer substantial gains in expressiveness and ease of use compared with GPLs in their domain of application'. [2] describes the typical costs of a DSL, noting that a small extra initial investment in a DSL implementation typically leads to long term savings, in comparison to alternative routes. [40]

A domain specific language (DSL) is a programming language tailored for a particular application domain. Characteristic of an effective DSL is the ability to develop complete application programs for a domain quickly and effectively. A DSL is not (necessarily) "general purpose." Rather, it should capture precisely the semantics of an application domain, no more and no less.

There are lots of advantages to using DSLs, starting with the fact that programs are generally easier to write, reason about, and modify compared to equivalent programs written in general purpose languages. Indeed, these are the same advantages gained from using any high-level programming language. [18]

4.4.1 Método

4.4.2 GQM

4.4.3 Questionário

4.4.4 Análise dos Resultados

* Questionário montado para avaliar a utilidade de DbC com NeoIDL

* Inicialmente motivado pelo estudo do Alessandro Garcia

* GQM e Avaliação TAM

* Montagem do questionário

1. Perfil técnico-profissional do respondente 1.1 Para qual órgão ou empresa você presta serviços atualmente? 1.2 A quanto tempo você trabalha com desenvolvimento Web 1.3 A quanto tempo você desenvolve com uso de APIs Web (Web Service) 1.4 Qual o seu nível de experiência com especificação de API REST 1.5 Qual o seu nível de experiência com especificação de contratos com Swagger

3 Questões sobre especificação e implementação de APIs Web 3.1 A especificação do contrato formalmente, seja em Swagger ou NeoIDL, em relação a descrição textual, aumentará meu nível de acerto na implementação (efetividade). 3.2 Identificar e compreender as operações e atributos na especificação Swagger é simples para mim. 3.3 Identificar e compreender as operações e atributos na especificação NeoIDL é simples para mim.

4. DbC 4.1 Conhecer previamente e explicitamente as precondições será útil para mim. (Useful) 4.2 Aprender a identificar as precondições na NeoIDL parece ser simples pra mim (Easy to learn) 4.3 Parece ser fácil para mim declarar uma precondição na NeoIDL (Clear and understandable) 4.4 Me lembrar da sintaxe da precondição na NeoIDL é fácil (Remember)

5. Geração de código 5.1 É claro e compreensível para mim o efeito da precondição sobre o código gerado (Controllable) 5.2 A geração do código de pré e pós-condições aumentará minha produtividade na implementação do serviço (Job performance) 5.3 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. 5.4 Nesse mesmo contexto, eu vou preferir utilizar contratos escritos em NeoIDL do que descritos de outra forma

- * Distribuição do questionário

- * Ameaças - não foi fornecido nenhum material sobre a NeoIDL, apresentando somente o uma descrição de serviço - O questionário foi aplicado uma única vez, sem melhorias a partir do primeiro conjunto de respostas

- * Questionários futuros

A principal questão de pesquisa a ser avaliada com o uso do questionário é a utilidade em se agregar ao design das especificações de serviços REST as garantias de pré e pós-condições. Em segundo momento, pressupondo a utilidade, avaliar se a NeoIDL cumpre satisfatoriamente com este propósito, agregando à sintaxe da linguagem a possibilidade de se expressar pré e pós-condições.

- * Separar os respondentes em faixas de experiência. Verificar se as respostas dos menos experientes precisam ser descartadas pela pouca capacidade crítica. Separar a análise entre os respondentes que conhecem e os que não conhecem Swagger.

- * Perspectivas de comparação a) Experiência com desenvolvimento com uso de REST b) Experiência com Swagger c) Utilidade da especificação formal de contratos d) Percepção da NeoIDL sem DbC

REFERÊNCIAS BIBLIOGRÁFICAS

- [1] Javascript object notation. <http://www.json.org>. JSON official site.
- [2] Raml - the simplest way to design apis. <http://raml.org>. The Official RAML web site.
- [3] Swagger - the world's most popular framework for apis. <http://swagger.io/>. Swagger project official site.
- [4] World wide web consortium (w3c) - extensible markup language (xml). <https://www.w3.org/XML>. XML W3C official site.
- [5] World wide web consortium (w3c) - web services description language. <https://www.w3.org/TR/wsdl>. WSDL W3C official site.
- [6] Yaml ain't markup language. <http://yaml.org>. The Official YAML web site.
- [7] Alberts, D. S. e Hayes, R. E. *Understanding Command and Control*. DoD Command and Control Research Program, 1st edition, 2006.
- [8] Alonso, G., Casati, F., Kuno, H., e Machiraju, V. *Web services*. Springer, 2004.
- [9] 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.
- [10] 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.
- [11] Erl, T. *SOA design patterns*. Pearson Education, 2008.
- [12] Erl, T. *Soa: principles of service design*, volume 1. Prentice Hall Upper Saddle River, 2008.

- [13] 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.
- [14] Fettig, A. *Twisted network programming essentials*, volume 2. "O'Reilly Media, Inc.", 2013.
- [15] Fielding, R. T. *Architectural styles and the design of network-based software architectures*. Tese de Doutorado, University of California, Irvine, 2000.
- [16] Hadley, M. J. Web application description language (wadl). 2006.
- [17] He, H. What is service-oriented architecture. *Publicação eletrônica em 30/09/2003*, 2003.
- [18] Hudak, P. Modular domain specific languages and tools. In *Software Reuse, 1998. Proceedings. Fifth International Conference on*, pages 134–142. IEEE, 1998.
- [19] Jazequel, J.-M. e Meyer, B. Design by contract: The lessons of ariane. *Computer*, 30(1):129–130, 1997.
- [20] 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.
- [21] Leavens, G. T. e Cheon, Y. Design by contract with jml, 2006.
- [22] Leymann, F. Combining web services and the grid: Towards adaptive enterprise applications. In *CAiSE Workshops (2)*, pages 9–21, 2005.
- [23] 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.
- [24] Lima, L. F. Contratos rest robustos e leves: uma abordagem em design-by-contract com neoidl. *WTDSOft 2015*, page 7, 2015.
- [25] Mackinlay, J. e Genesereth, M. R. Expressiveness and language choice. *Data & Knowledge Engineering*, 1(1):17–29, 1985.
- [26] Meyer, B. Eiffel: A language and environment for software engineering. *Journal of Systems and Software*, 8(3):199–246, 1988.

- [27] Meyer, B. Applying 'design by contract'. *Computer*, 25(10):40–51, 1992.
- [28] Meyer, B. *Object-oriented software construction*, volume 2. Prentice hall New York, 1997.
- [29] Mills, H. D. The new math of computer programming. *Communications of the ACM*, 18(1):43–48, 1975.
- [30] Mumbaikar, S., Padiya, P., e others,. Web services based on soap and rest principles. *International Journal of Scientific and Research Publications*, 3(5), 2013.
- [31] (OMG), O. M. G. Interface definition language 3.5. Technical report, Object Management Group, 2014. <http://www.omg.org/spec/IDL35/3.5/PDF/>.
- [32] 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.
- [33] 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.
- [34] 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.
- [35] Ranta, A. *Implementing Programming Languages. An Introduction to Compilers and Interpreters*. Texts in computing. College Publications, 2012.
- [36] 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.
- [37] Serrano, N., Hernantes, J., e Gallardo, G. Service-oriented architecture and legacy systems. *Software, IEEE*, 31(5):15–19, 2014.
- [38] 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>.
- [39] Taha, W. Domain-specific languages. In *International Conference on Computer Engineering Systems (ICCES)*. Springer, 2008.
- [40] Tratt, L. Evolving a dsl implementation. In *Generative and Transformational Techniques in Software Engineering II*, pages 425–441. Springer, 2008.

- [41] 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