UNIVERSIDADE DE BRASÍLIA
FACULDADE DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

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

LUCAS FERREIRA DE LIMA

ORIENTADOR: RODRIGO BONIFÁCIO DE ALMEIDA

DISSERTAÇÃO DE MESTRADO EM
ENGENHARIA ELÉTRICA

PUBLICAÇÃO: MTARH.DM - 017 A/99

BRASÍLIA/DF: JULHO - 2016.

UNIVERSIDADE DE BRASÍLIA
FACULDADE DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

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

LUCAS FERREIRA DE LIMA

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

APROVADA POR:

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

Prof. XXX
(Examinador Interno)

YYY
(Examinador Externo)

BRASÍLIA/DF, 01 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

A adoção do paradigma arquiteturação baseado em serviços... O presente trabalho foi desenvolvido, ..., tendo como objetivo geral ..... Como objetivos específicos, o trabalho procurou .....

A proposta partiu...

Durante a realização...

Os resultados sugerem...

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

LISTA DE TABELAS

LISTA DE FIGURAS

LISTA DE SÍMBOLOS, NOMENCLATURA E ABREVIAÇÕES

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 [5]. Nessa abordagem, os recursos de software são empacotados como serviços, os quais são módulos bem definidos e auto-contidos, provêem funcionalidades negociais e com estado e contexto independente [22].

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 definidos 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, condições sobre como as operações são realizadas e informações sobre as operações [8].

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 Desing-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*) [**?**] é considerada uma solução verbosa 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 (*Representational State Transfer*). Outras alternativas para REST, como Swagger e RAML[1], usam linguagens de propósito geral (em particular JSON e YAML) 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 IDLs CORBA) e não oferecem mecanismos semânticos de extensibilidade e modularidade.

Com o objetivo de mitigar esses problemas, a linguagem NeoIDLfoi 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 *Interface Description Languages* – IDLs – presentes em *Apache Thrift*™e CORBA™. Por outro lado, a NeoIDL, da mesma forma que WSDL, Swagger e RAML não oferece construções para especificação de contratos formais de comportamento como os presentes em linguagens que suportam DBC (*Design by Contract*) [18], como JML, Spec# e Eiffel. Em outras palavras, a NeoIDLadmite apenas contratos fracos (*weak contracts*), sem suporte a construções como pré e pós condições.

1.2 OBJETIVO GERAL

O objetivo geral de 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 implentação de serviços REST.

1.2.1 Objetivos específicos

1. Realizar análise empírica de expressividade e reuso da especificação de contratos em NeoIDLem comparação com *Swagger*, a partir de contratos reais do Exército Brasileiro.

2. Extender a sintaxe da NeoIDLpara admitir construções de Design by Contract, com pré e pós condições para operações de serviços REST.

3. Implementar um estudo de caso de geração de código em *Python Twisted* com suporte a Design by Contracta partir de contratos especificados em NeoIDL

[1]http://raml.org/spec.html

4. Coletar a percepção de desenvolvedores sobre a aceitação da especificação de contratos REST com Design by Contractna NeoIDL

1.2.2 JUSTIFICATIVA E RELEVÂNCIA

- aumento do uso de SOC - potencial do SOC está no baixo acomplamento dos serviços. - aumento do uso de REST - contratos fracos prejudicam a qualidade/aumentam os erros - baixa qualidade prejudica o reuso -

O uso do padrão REST para construção de *Web Service*é crescente no contexto do desenvolvimento de soluções basedas em serviço e não se dipõe de uma linguagem padrão para especificação de contratos fortes. A linguagem específica de domínio NeoIDL- foi desenvolvida para ser uma alternativa para especificação REST, caracterizada pela coesão e simplicidade em se compreender, mas que não dispunha de suporte a pré e pós condições.

1.2.3 ESTRUTURA

2 REFERENCIAL TEÓRICO

2.1 COMPUTAÇÃO ORIENTADA A SERVIÇO

As empresas precisam estar preparadas para responder rápida e eficientemente a mudanças impostas por novas regulações, por aumento de competição ou ainda para usufruir de novas oportunidades. No contexto atual, em que as informações fluem de modo extremamente veloz, o tempo disperdiçado pelas organizações para se adaptar a um novo cenário tem um preço elevado, gerando expressiva perda de receita e, em um 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 intituições 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 ferramental para apoiar continuamente os processos de negócio, se tornem entrave para a ágil incorporação dos novos processos. Por outro lado, a nova configuração deve se manter integra e funcional com o já complexo cenário de TI.

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 e a habilita para inteoperar facilmente [23].

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 [7]. Assim, o paradigma de SOC está voltado para o projeto de

soluções preparadas para constantes mudanças, substituindo-se pequenas peças – os serviços – por outras.

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 [24].

As vertentes mais visionárias – não ainda concretizada e utópica para muitos pesquisadores – da computação orientada a serviços prevêem 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 [15].

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 várias coisas, como paradigmas e princípios de projeto, catálogo de padrões de projeto, padronização de linguagem, modelo arquitetural específico, e conceitos correlacionados, tecnologias e plataformas. A computação orientada a serviços é baseada em modelos anteriores de computação distribuída e os extendem com novas camadas de projeto, aspectos de governança, e uma grande gama de tecnologias de implementações especializadas, em grande parte baseadas em *Web Service* [8].

Orientação a serviço é um paradígma de projeto cuja intenção é a criação de unidades lógicas moldadas individualmente para podem serem utilizadas conjutamente e repetidamente para se atender 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 EAI, BMP e *Web Service*[8].

A orientação a serviços é composta principalmente de oito princípios de projeto (descritos na seção 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 no como elemento central da representação lógica da solução [8].

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 [8], o termo arquitetura orientada a serviço – SOA – vem sendo amplamente utilizado na mídia e nos produtos de divugação de fabricantes que tem 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 da orientação dos princípios de projeto de orientação a serviço que distigue uma unidade de lógica como um serviço comporada a outras unidades de serviços que podem existir isoladamente como um objeto ou componente [8].

Após a modelagem conceitual do serviço, os estágios de projeto e desenvolvimento produzem um serviço que é 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 fundamental a que descreve a sua interface técnica. Eles compõem o contrato técnico do serviço, cuja essência é estabelecer uma API com as funcionalidades providas pelo serviço por meio de suas capacidades [8].

Os serviços implementados como *Web Service*SOAP normalmente são descritos em seu WSDL [1], *XML schemas* and políticas (*WS-policy*). Já os serviços im-

[1] *Web Service Description Language*

plementados como *Web Service* REST não possuem uma linguagem padrão para especificação de contratos. Já foram propostas algumas alternativas como WADL [10], Swagger [1], e NeoIDL [16].

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 se padronizar a criação dos contratos de serviços [8].

2.1.2 Objetivos, benefícios e características

De modo diferente de arquiteturas convecionais, 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 [24].

A unidade lógica da arquitetura orientada a serviços é exatamente os serviços. Serviços são pequenos que provêem funcionalidades específicas para serem reutilizadas em várias aplicações. Cada serviço é uma entidade isolada com dependências limitadas de outros recursos compartilhados [27]. 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 [6]:

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

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

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

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

(e) Ampliar o retorno sobre investimento, pois muitos serviços podem ser reajandos 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 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 muito rapidamente [6].

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 são 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 que os serviços serão compartilhados por várias áreas da empresa ou farão parte de soluções maiores, como serviços compatilhados.

A capacidade de composição é a quarta característica. Os serviços devem ser projetados não somente para serem reusados, mas também 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 estruturada em oito princípios fundamentais [8]. São eles que caraterizam 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:

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 firts* é uma consequência direta deste princípio [8].

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

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

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

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

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

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

Por outro lado, há um acoplamento positivo que o que gera dependência da lógica em relação ao contrato [8]. Ou seja, idealmente a implementação do serviço deve ser derivada do contrato, pondendo se ter inclusive a geração de código a partir do contrato.

Abstração - Os constratos 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 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 de outro serviço.

Descoberta de serviço - Serviços devem conter metadados por meio dos quais os serviços possam ser descobertos e interpretados. Tornar cada

serviço de fácil descoberta e interpreção pelas equipes de projeto é o foco deste princípio. Os próprios contratos de serviço devem ser projetados para incorporar informações que auxiliem na sua descoberta.

Composição - Serviços são participantes efetivos de composição, independentemente do tamanho ou complexidade da composição. O princípio da composição faz com que os projetos de serviços sejam projetados para possibilitar que eles se tornem participantes de composições. Deve-se levar em conta, entretanto, os outros princípios no planejamento de uma nova composição, considerando a complexidade de composições 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 acima. 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 propõem 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 [8].

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 seja 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 [13].

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 funcionalidade a que se destina, porém com a preocupação em se promover estabilidade e reuso.

As principais vantagens do *Contract-first* no baixo acoplamento do contrato em relação a sua implementação, possibilidade de reuso de esquemas de dados (XML ou JSON Schema), simplificação do versionamento e facilidade de manutenção [13]. A desvantagem está justamente na complexidade de escrita to contrato. Porém várias ferramentas já foram desenvolvidas para facilitar essa tarefa.

2.2 Web Services

Web Service são aplicações modulares e autocontidas que podem ser publicadas, localizadas a acessadas pela *Web* [3]. 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 [27].

A necessidade das organizações de integrar suas soluções, seja entre os sistemas internos ou entre esses e sistemas de outras empresas [25], não é recente. Essa é uma das principais motivações do uso de *Web Service*, por possibilitar que soluções contruídas 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 tipo de aplicações podem ser concebidas utilizando SOAP ou REST *Web Service*, 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 2.2.1) tem ganhado popularidade [21].

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 ambientes descentralizado ou distribuído [2]. 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 mais antigo, SOAP é o *Web Service* mais comumente utilizado pela indústria.

Algumas pessoas chegam a tratar *Web Service* apenas como SOAP e WSDL [27]. SOAP atua como um envelope que transporta a mensagem XML, e possui vastos padrões para transformar e proteger a mensagem e a trasmissão.

2.2.1.1 Especificação de contratos

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

Muitas organizações preferem utilizar SOAP por ele dispor de mais mecanismos de segurança e tratamento de erros [27]. Além disso a tipagem de dados é mais forte em SOAP que em REST [21].

2.2.2 REST (Fielding)

O termo REST foi criado por Roy Fielding, em sua tese de doutorado [9], 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 URI) que pode ter zero ou mais representações [11].

O estilo arquitetural REST é cliente-servidor, em que o cliente enviar uma requisitação 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 [21], que podem ser vários formatos, como XML e JSON [27]. Toda troca de informações ocorre por meio do protocolo HTTP, com uma semântica específica para cara operação:

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

As transações são independentes entre si e com as transações anteriores, pois o servidor não guarda qualquer informação de sessão do cliente. Todas as informações de estado são trafegadas nas proprias requisições, de modo que as respostas também são independentes. Essas características tornam as *Web Service* REST simples e leves [21].

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 [21].

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 Service* implantados. Atualmente existem algumas linguagens com o propósito de documentar o contrato REST.

A linguagem mais popular é *Swagger* cujo projeto se iniciou por volta de 2010 para atender a necessidade de um projeto específico, sendo posteriormente vedida para uma grande empresa. Em janeiro de 2016, *Swagger* foi doada para o *Open API Iniciative (OAI)* e denominada de *Open API Specification*. O própito da iniciativa é tornar *Swagger* padrão para especificação de APIs com independencia de fornecedor. Apoiam o projeto grandes empresas como Google, Microsoft e IBM.

WADL (*Web Application Description Language*), uma especição baseada em XML semelhante ao WSDL, foi proposta projetada pela *Sun Microsystems* e sua última versão submetida em 2009. Outra linguagem proposta é a RAML – 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* [28].

2.3 Design by Contract

[**?**]

Design by Contract [18] - 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 satisfeitas (denominadas de pré-condições). Do outro lado o fornecedor deve garantir, se respeitadas as pré-condições, as propriedades relacionadas ao sucesso da execução (pós-condições).

DbC tem o objetivo de aumentar a robustez do sistema e tem na linguagem Eiffel [17] 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 mais corretos.

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 oblicações de cada parte [19]. 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 pré-condições 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 [19] 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 a análise das condições de consistência necessárias para o funcionamento correto da relação de cooperação entre clinte-fornecedor. Essas condições são expressas em cada contrato, o qual especifica as obrigações a que cada parte está condicionada.

Segundo Bertrand Meyer [19], 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 [20] para afirmar que Design by Contractpermite construir programas corretos e saber que 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.

– Assetion violation rule (2) A precondition violation is the manifestation of a bug in the client. A postcondition violation is the manifestation of a bug in the supplier

– Usar na motivação – The problem gets aggravated by the fact that modern software applications are expected to make use of these reusable modules as much as possible, in an effort to reduce both the development costs and the production time. Unfortunately, this situation opens the door for the propagation of security vulnerabilities among several applications as a result of the incorrect enforcement of security properties in reusable software modules and threatens the security and safety of applications as a whole. [26]

– Usar na motivação There is a simple lesson here: Reuse without a precise specification mechanism is a disastrous risk. Effective reuse requires design by contract. Without a precise specification attached to each reusable component— precondition, postcondition, invariant— no one can trust a supposedly reusable component. Without a specification, it is probably safer to redo than to reuse. [12]

The cornerstone of object-oriented technology is reuse. For reusable components, which may be used in thousands of different applications, the potential consequences of incorrect behavior are even more serious than for applicationspecific developments.

The mechanisms for expressing such conditions are called assertions. Some assertions. called preconditions and postconditions. apply to individual routines. Others, the class invariants, constrain all the routines of a given class and will be discussed later.The construct Old may appear only in a routine postcondition.

This principle is the exact opposite of the idea of defensive programming, since it directs programmers to avoid redundant tests. If the contract is precise and explicit, there is no need for redundant checks.

Client programmers do not expect miracles. As long as the conditions on the use of a routine make sense, and the routine's documentation states these conditions (the con-

tract) explicitly, the programmers will be able to use the routine properly by observing their part of the deal.

Eiffel assertions are Boolean expressions, with a few extensions such as the old notation. Since the whole power of Boolean expressions is available, they may include function calls. Because the full power of the language is available to write these functions, the conditions they express can be quite sophisticated. [18]

– Reasonable Precondition principle Every routine precondition (in a "demanding" design approach) must satisfy the following requirements: • The precondition appears in the official documentation distributed to authors of client modules. • It is possible to justify the need for the precondition in terms of the specification only.

This clearly unacceptable situation is akin, in human contracts, to a deal in which the supplier would impose some conditions not stated explicitly in the contract, and hence could reject a client's request as incorrect without giving the client any way to determine in advance whether it is correct.

– Precondition Availability rule Every feature appearing in the precondition of a routine must be available to every client to which the routine is available. There is no such rule for postconditions.

Preconditions and postconditions describe the properties of individual routines. There is also a need for expressing global properties of the instances of a class, which must be preserved by all routines. Such properties will make up the class invariant, capturing the deeper semantic properties and integrity constraints characterizing a class.

- Invariants There isalso a need for expressing global properties of the instances of a class, which must be preserved by all routines. Invariants have a clear interpretation in the contract metaphor. Human contracts often contain references to general clauses or regulations that apply to all contracts within a certain category; [19]

2.3.1 Implementações de DbC

- Eifel

Much of the emphasis in the design of Eiffel has been on promoting such quality factors as reusability, extendibility, and compatibility. But these qualities are meaningless unless programs are also correct and robust. In fact, as techniques for the production of truly reusable. Eiffel includes language constmcts that promote a systematic approach to software construction. The regular use of these constructs, and the general attitude they imply towards program const ction, have proved extremely beneficial as to the correctness and robustness of software built with Eiffel.

The precondition and postcondition of a routine may be viewed as an explicit contract between the class implementer and the authors of client classes. The precondition binds the clients: a call that does not satisfy it is not valid, and the class may do what it pleases with it. The postcondition binds the class: If the precondition is satisfied, the client is entitled to expect that the routine will terminate in a state that satisfies the postcondition. An approach to software construction based on this notion of contract is developed [17]

- JML

The Java Modeling Language (JML) [5], is a behavioral interface specification language (BISL) for Java, with a rich support for DBC contracts. Using JML, the behavior of Java modules, e.g., what a Java class or interface is expected to do at runtime, can be specified using pre, post conditions, and class invariants, which are commonly expressed in the form of assertions, and are added to Java source code as comments of the form //@ or /*@...@*/.

JML stands for "Java Modeling Language". It is a formal behavioral interface specification language for Java. As such it allows one to specify both the syn- tactic interface of Java code and its behavior. The syntactic interface of Java code consists of names, visibility and other modifiers, and type checking in- formation. JML combines the practicality of DBC language like Eiffel [12] with the expressiveness and formal- ity of model-oriented specification languages. [14]

- Spec#

The Spec language is a superset of C#, an object-oriented language targeted for the .NET Platform. C# features single inheritance whose classes can implement multiple interfaces, object references, dynamically dispatched methods, and exceptions, to

mention the features most relevant to this paper. Spec adds to C# type support for distinguishing non-null object references from possibly-null object references, method speci- fications like pre- and postconditions, a discipline for managing exceptions, and support for constraining the data fields of objects. In this section, we explain these features and rationalize their design [4]

3 A LINGUAGEM PARA ESPECIFICAÇÃO DE CONTRATOS NEOIDL

3.1 NeoIDL

3.1.1 Objetivos

3.1.2 Histórico

3.1.3 Arquitetura do framework

3.1.4 Avaliações

3.1.4.1 Expressividade

Nesta subseção falar da avaliação feita nos contratos do exército.

3.1.4.2 Potencial de reuso

Complementar a avaliação de expressividade com o potencial de reuso.

3.2 Proposta: Serviços com Desing-by-Contract

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

Na sociedade civil, contratos são meios de se formalizar acordo entre partes a fim de definir os direitos e deveres de cada parte e buscar atingir o objetivo esperado dentro de determinadas regras. Cada parte espera que as outras cumpram com suas

obrigações. Por outro lado, sabe-se que o descumprimeto das obrigações costuma implicar de penalizações até o desfazimento do contrato.

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

A proposta deste trabalho é extender 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. O detalhamento do processo é exposto nas seções que se seguem.

3.2.1 Modelo de operação

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

O diagrama da Figura 3.1 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 pré-condição. Caso tenham sido estabelecidas pré-condições, 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.

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 3.1, 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.

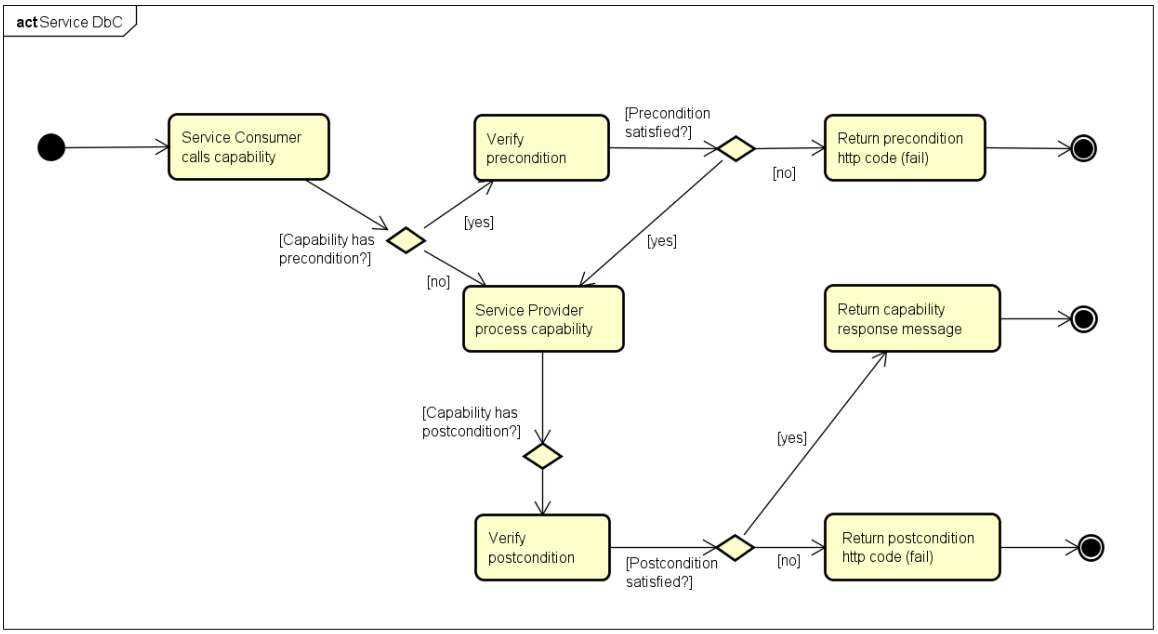

Figura 3.1: Digrama de atividades com verificação de pré e pós condições

3.2.1.1 Verificação das pré-condições

As pré-condições podem ser do tipo baseado nos parâmetros da requisição ou do tipo baseado na chamada a outro serviço. Denominamos, para o contexto desta dissertação, de básica a pré-condição baseada apenas nos parâmetros da requisição (atributos da chamada ao serviço). Nessa validação é direta, comparando os valores passados com os valores admitidos.

No caso das pré-condições 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 pré-condição se mantem simples.

A Figura 3.2 detalha as etapas de verificação de cada pré-condição. Nota-se que a saída para as situações de desatendimento às pré-condições, independentemente do tipo, é o mesmo. O objetivo desta abordagem é simplificar o tratametno de exceção no consumidor.

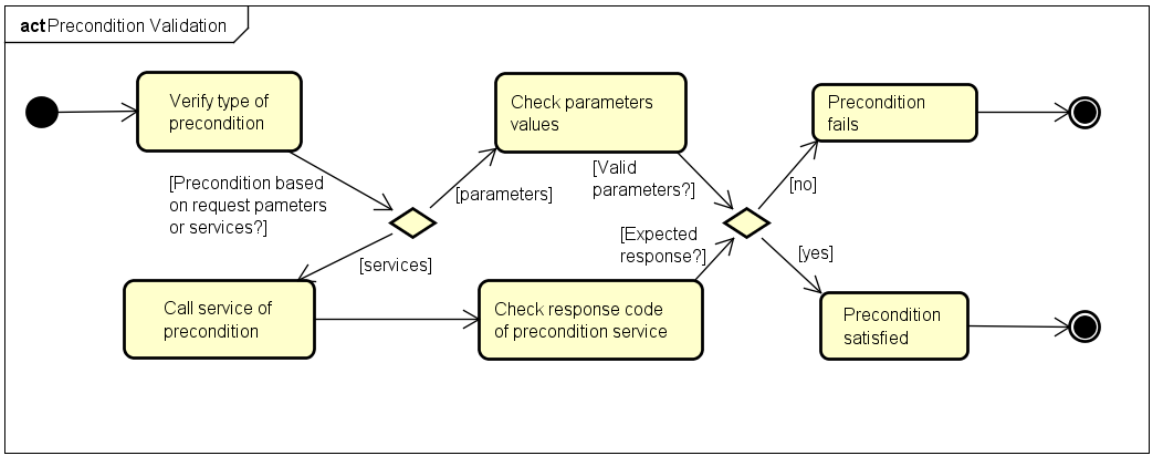

Figura 3.2: Diagrama de atividades do processamento da pré-condição

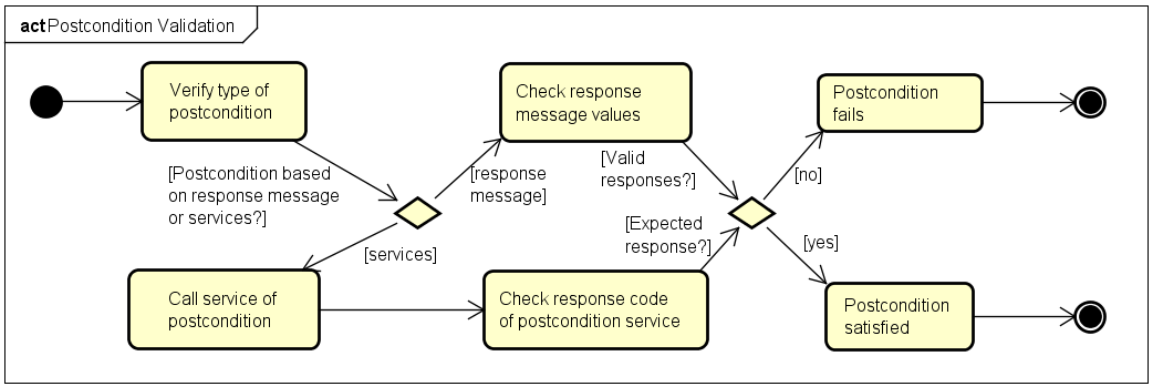

Figura 3.3: Diagrama de atividades do processamento da pós-condição

3.2.1.2 Verificação das pós-condições

A verificação das pós-condições acontece de modo muito similar a das pré-condições. 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 contidos na mensagem de retorno. A Figura 3.3 descreve as etapas necessárias para validação de cada pré-condição.

[language=NeoIDL,firstnumber=1]DBCsimple.neo

Figura 3.4: Exemplo da notação DBC básica na NeoIDL

3.3 Implementação de Design-by-Contract na NeoIDL

3.3.1 Pré-condição básica

3.3.2 Pós-condição básica

...

3.3.3 Pré-condição com chamada a serviço

...

3.3.4 Pós-condição com chamada a serviço

...

3.4 Estudo de caso

...

3.4.1 Pluggin Twisted

4 AVALIAÇÃO SUBJETIVA

4.1 Utilidade da NeoIDL com DbC

4.1.1 Método

4.1.2 GQM

4.1.3 Questionário

4.1.4 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 pré-condições será útil para mim. (Useful) 4.2 Aprender a identificar as pré-condições na NeoIDL parece ser simples pra mim (Easy to learn) 4.3 Parece ser fácil para mim declarar uma pré-condição na NeoIDL

(Clear and understandable) 4.4 Me lembrar da sintaxe da pré-condição na NeoIDL é fácil (Remember)

5. Geração de código 5.1 É claro e compreensível para mim o efeito da pré-condição 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

* Avaliação dos resultados

* 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

5 CONCLUSÕES E TRABALHOS RELACIONADOS

5.1 CONCLUSÕES GERAIS

A boa definição dos contratos ...

5.2 TRABALHOS RELACIONADOS E PESQUISAS FUTURAS

...

- Trabalhos relacionados com hipermedia

REFERÊNCIAS BIBLIOGRÁFICAS

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

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

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

[4] 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.

[5] 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.

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

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

[8] 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.

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

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

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

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

[13] 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.

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

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

[16] 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.

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

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

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

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

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

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] Rubio-Medrano, C. E., Ahn, G.-J., e Sohr, K. Verifying access control properties with design by contract: Framework and lessons learned. In *Computer Software and Applications Conference (COMPSAC), 2013 IEEE 37th Annual*, pages 21–26. IEEE, 2013.

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

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

APÊNDICES

A CONTRATO NEOIDL COM DBC

...