Design-by-Contract meets the REST architectural style: A case study using NeoIDL

Lucas F. Lima[a,b], Rodrigo Bonifácio[c], Edna Dias Canedo[a,c]

[a] *Brasilia, Brazil, Campus Darci Ribeiro*
[b] *University of Brasilia - UnB*
[c] *University of Brasilia - UnB*

Abstract

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

Keywords: example, LaTeX, template

Email addresses: unb.lucaslima@gmail.com (Lucas F. Lima), author.two@mail.com (Rodrigo Bonifácio), author.three@mail.com (Edna Dias Canedo)

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 [?].

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.

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

A qualidade da especificação do contrato de serviço é fundamental para o projeto de software baseado em SOC. Este artigo 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*.

A seção XXX

2. Fundamentação

2.1. Computação orientada a serviço

A eficiência na integração entre as soluções de TI é um fator determinante para que se consiga alterar uma parte sem comprometer todo um 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 [?].

A computação orientada a serviços – SOC – endereça essas necessidades em uma plataforma que aumenta a flexibilidade, a fim de reagir rapidamente a mudanças nos requisitos de negócio. Para obter esses benefícios, contudo, os serviços devem cumprir com determinados quesitos, que incluem alta autonomia ou baixo acoplamento [?]. 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 está em 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 [?].

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 – os serviços – que podem ser consumidas tanto por usuários finais como por outros serviços [?].

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 [?]. 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.

2.2. Princípios SOA

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

O princípio do **Contrato padronizado** estabelece que serviços dentro de um mesmo inventário estejam 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. O padrão de projeto *Contract-first* é uma consequência direta deste princípio [**?**].

Os contratos de serviço devem impor aos seus consumidores requisitos de baixo acoplamento. Os contrato também devem ser desacoplados de seu ambiente. Essas relações são guiadas pelo princípio do **Baixo acomplamento**. A forma como o contrato é projetado e posicionado na arquitetura é que gerará o benefício do baixo acoplamento.

O projeto do serviço deve considerar como negativos os acoplamentos entre o contrato e a funcionalidade suportada pelo serviços, entre o contrato a a sua implementação subjacente, entre o contrato com a tecnologia e a lógica interna adotada para o serviço. Esses acoplamentos devem ser evitados.

Há, porém, um acoplamento desejado, que é o que gera dependência da lógica em relação ao contrato [**?**]. Ou seja, idealmente a implementação do serviço deve ser derivada do contrato, pondendo se ter inclusive a geração de código a partir do contrato.

O princípio da **Abstração** visa garantir a exposição apenas de informações necessárias e essenciais no contrato de serviço. Ou seja, o contrato deve possui tão somente as informações necessárias ao consumo e compreensão do serviço. A **Descoberta do serviço**, outro princípio SOA, prega que os contratos de serviços contenham informações (metadados) que possibilitem a descoberta e interpretação do serviço.

O aspecto do reuso é expresso em alguns princípios SOA. O princípio da **Reusabilidade** estabelece que os serviços que contenham ou expressem lógica agnóstica podem ser disponibilizaddos com recursos reusáveis. Nessa mesma linha, o princípio da **Composição** orienta para que serviços sejam participantes efetivos de composições, independentemente do tamanho ou complexidade da composição.

Por fim, os princípios da **Autonomia** – serviços têm controle sobre seu ambiente de execlução – e **Ausência de estado** – retringindo o controle de estado apenas a quanfo for inevitavelmente necessário – formam, junto com os demais, um conjunto de características que buscam fazer com que o projeto dos serviços alcance os objetivos da abordagem SOA.

2.3. Contract-First

A abordagem *Contract-first* preocupa-se sobretudo 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 [?]. A desvantagem está justamente na complexidade de escrita do contrato. Porém, várias ferramentas já foram e vem sendo desenvolvidas para facilitar essa tarefa.

A outra forma de se garantir que o contrato expressa o que serviço realiza é a abordagem *Code-first*, em que o contrato é normalmente gerado a partir de anotações no código fonte. 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 ainda a flexibilidade e extensibilidade, de modo que o reuso é prejudicado.

2.4. Web Services

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

SOAP – *Simple Object Access Protocol* – é um protocolo padrão W3C para troca de mensagens em sistemas distribuídos, normalmente sobre o protocolo HTTP e acessível pela Internet. As mensagens em SOAP são codificadas XML e seus contratos 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.

O estilo arquitetural REST foi projetado por Roy Fielding, em sua tese de doutorado [?], para descrever um modelo distribuído de sistemas hipermedia sobre o protocolo HTTP, com uma semântica específica para cada operação. Tanto a requisição como a resposta ocorrem por meio da transferência de representações de recursos [?], que podem ser de vários formatos, como XML e JSON [?].

O uso de REST tem se tornado popular por conta de sua flexibilidade e performance em comparação com SOAP, o qual precisa envelopar suas informações em um pacote XML [?], de armazenamento, transmissão e processamento onerosos. 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, sendo *Swagger* a mais popular.

2.5. Design-by-Contract

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

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

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 [**?**]. 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. De forma indireta, *Design-by-Contract* estimula um cuidado maior na análise das condições necessárias para um funcionamento correto do recurso.

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 [**?**].

Eiffel, JML e Spec# são exemplos de linguagem e extensões de linguagens que dão suporte a *Design-by-Contract*. A linguagem **Eiffel** foi desenvolvida em meados dos anos 80 por Bertrand Meyer [**?**] com o objetivo de criar ferramentas que garantissem mais qualidade aos *softwares*. A ênfase do projeto de Eiffel foi promover reusabilidade, extensibilidade e compatibilidade.

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

3. NeoIDL: DSL para constratos REST

A NeoIDL é uma linguagem específica de domínio (*Domain Specific Language - DSL*) elaborada com o objetivo de possibilitar, em um processo simples, a elaboração de contratos para serviços REST. Em seu projeto, foram considerados os requisitos de concisão, facilidade

de compreensão humana, extensibilidade e suporte à herança simples dos tipos de dados definidos pelo usuário.

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

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

Partiu-se então para a criação uma nova linguagem, com sintaxe inspirada em linguagens mais claras e concisas – como *CORBA IDL*™[?] e *Apache Thrift*™[?] –, e que permitisse ainda a declaração de tipos de dados definidos pelo usuário e extensibilidade. Ambas, *CORBA* e *Apache Thrift*, possuem contudo limitações nesses últimos aspectos.

A NeoIDL suporta a geração de código poliglota, por meio da implementação de plugins NeoIDL para cada linguagem alvo. Atualmente, há plugins para Java, Python NeoCortex[1], e Python Twisted. Detalhes sobre o framework de geração de código da NeoIDL foram publicados em [?].

O artigo [?] também descreve uma estudo empírico sobre a expressividade e potencial de reuso da NeoIDL. A base desse estudo foi a análise de contratos reais escritos em Swagger em comparação com os equivalentes em NeoIDL. Em resumo, nesse cenário concreto, identificou-se serem necessárias 4 linhas de especificação em NeoIDL para cada conjunto de 10 linhas em Swagger.

4. Contratos REST robustos

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.

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, há ainda a ausência de padrão para especificação contratos.

[1]Framework proprietário do Exército Brasileiro

A proposta deste trabalho é extender os níveis de garantias, de modo a promover um patamar adicional com obrigações mútuas entre os serviços (consumidor e provedor). Isso se dá pela adoção do conceito de *Design-by-Contract* 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. Modelo de operação

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

O diagrama apresentado na Figura 1 descreve como ocorre a operação das pré e pós-condições. O processo se inicia com a chamada à capacidade do serviço e a identificação da existência de uma precondição. Caso tenham sido estabelecidas precondições, essas são avaliadas. Caso alguma delas não tenham sido satisfeitas, o serviço principal não é processado e o provedor do serviço retorna o código de falha definido no contrato correspondente.

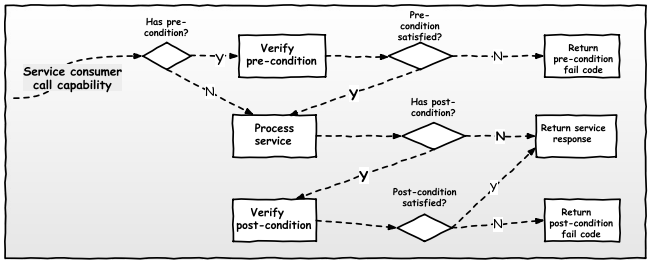

Figura 1: Digrama de atividades com verificação de pré e pós condições

Caso tenham sido definidas pós-condições, essas são acionadas após o processamento da capacidade, porém antes do retorno ao consumidor do serviço. Assim, conforme Figura 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. Observação sobre invariantes

Em *Design-by-Contract*, além dos conceitos de pré e pós-condições, há também a ideia de invariantes[**?**]. Quando aplicadas a uma classe em 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. Por essa razão, no estudo sobre a incorporação de *Design-by-Contract* em contratos de serviços, as invariantes não foram consideradas.

4.2. 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 deste artigo, de básica a precondição baseada apenas nos parâmetros da requisição. 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 mantém simples.

A Figura 2 apresenta as etapas de verificação de cada precondição. Nota-se que a saída para as situações de desatendimento às precondições, independentemente do tipo, é o mesma. Essa abordagem simplifica o tratamento de exceção no consumidor.

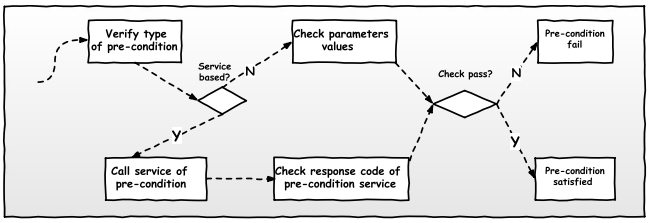

Figura 2: Diagrama de atividades do processamento da precondição

4.3. Verificação das pós-condições

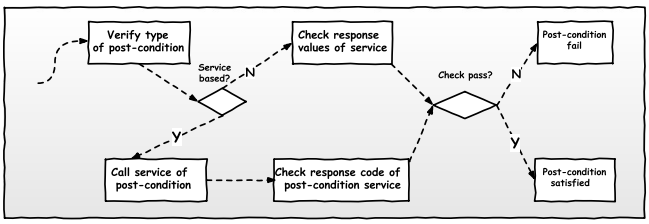

Figura 3: Diagrama de atividades do processamento da pós-condição

A verificação das pós-condições acontece de modo muito similar a das precondições. Há também os dois tipos, baseada 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 descreve as etapas necessárias para validação de cada precondição.

4.4. Inclusão de condições de DbC na especificação NeoIDL

Um módulo NeoIDL possui uma seção para declaração dos serviços. Cada serviço declarado na NeoIDL pode ter uma ou mais capacidades, as quais correspondem às operações HTTP utilizadas na arquitetura REST. Sintaticamente, um serviço possui uma lista de capacidades[2]:

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

```
{Serviço [Capacidade]}
```

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

```
{Capacidade [Condição DbC]}
```

É possível ainda, caso uma precondição ou pós-condição se aplique a todas as capacidades de um serviço, 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] ]}
```

4.5. Sintaxe geral de pré e pós-condições

As construções de *Design-by-Contract* possuem as seguintes estruturas:

Condição básica: **TipoCondição** `{[Argumento Comparação Valor]}`

Condição serviço: **TipoCondição** `{Serviço (Parâmetro) Comparação ValorSrv}`

Exceção:
Otherwise `{Valor de Retorno }`

Onde:

- **TipoCondição** indica se é uma precondição (*require*) ou uma pós-condição (*ensure*);
- **Argumento** indica o nome do atributo que será testado;
- **Comparação** indica a operação de comparação que será utilizada. A NeoIDL admite seis operadores de comparação: igualdade (==), diferença (<>), maior (>), maior ou igual (>=), menor (<) e menor ou igual (<=). Eles podem ser aplicados a qualquer tipo de pré ou pós-condição.

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

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

  Por exemplo, a precondição apresentada na Figura 4 poderia ser escrita como ***require*** *(**not** id <= 0)*. Os operadores *and* e *or* são infixos (ex.: ***require*** *((id > 0 **or** id <= -1000))*). O uso do operador *and* produz o mesmo efeito de se declarar duas precondições.

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

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

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

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

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#

4.6. Um exemplo de módulo com Design-by-Contract

A Figura 4 apresenta um exemplo de módulo NeoIDL de um serviço completo com três operações, cada uma com duas condições de *Design-by-Contract*.

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

A operação `POST` possui duas precondições (linhas 13 e 14). A primeira, básica, valida o valor do atributo *quantity*. A segunda, aciona o serviço *store.getOrder* com o parâmetro *id*. Nessa operação, foi definido um valor de exceção único para as duas precondições. A

NeoIDL associa essa instrução de *otherwise* geral às instruções anteriores que não possuem condição e exceção específica (caso da linha 8).

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

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

4.7. 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*. Por exemplo, o método `GET` submete os argumentos pelo *path* ou como *query string* da requisição e não em seu corpo. A Figura 5 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, de modo que 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*.

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[**?**] 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

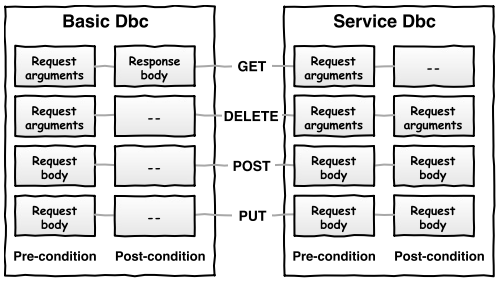

Figura 5: Diagrama da fonte de dados para acionamento de pré e pós-condições

operações não retornam dados no corpo da requisição, pois estão voltadas para alteração e não consulta de informações. Além disso, não há utilidade em se validar dados de requisição em uma pós-condição.

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

A pós-condição da operação `DELETE` pode acionar serviços com base nos argumentos da requisição (*Request arguments*). Um caso típico é de acionar o serviço de consulta para verificar se o dado foi efetivamente excluído. As operações `POST` e `PUT` podem acionam serviços em pós-condições utilizando as informações do corpo da requisição (*Request body*), uma vez que essas operações não possuem dados no corpo da resposta.

4.8. 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.

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 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.

Adotou-se o *framework Python Twisted*[**?**] 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.

4.9. Arquitetura dos serviços Twisted

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

Em termos de classes, a NeoIDL gera um conjunto de três módulos base, que constituem o pacote do *Twisted server*, representados na parte superior da Figura 6. Essas classes são fixas, ou seja, não dependem da quantidade nem do conteúdo dos serviços declarados no módulo NeoIDL.

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

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

Essas duas classes são suficientes para o servidor quando não se utiliza *Design-by-Contract* nos serviços. O pacote *DbcConditions* consolida o conjunto de classes responsáveis por processar as pré e pós-condições. A principal função de *DbcConditions* é realizar a comparação entre o valor real e o valor esperado, efetivando toda a lógica corresponde às construções de *Design-by-Contract* descritas na Subseção 4.5.

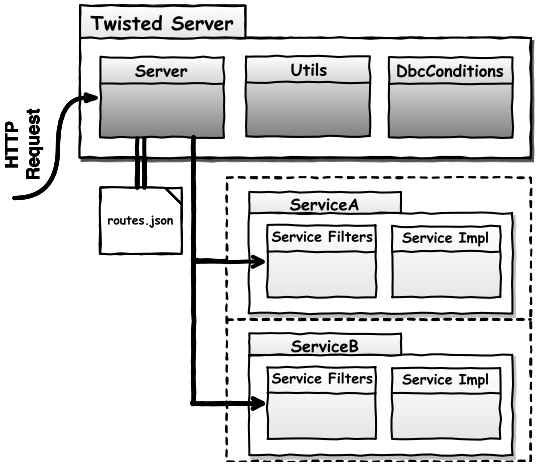

Figura 6: Arquitetura do serviço Twisted gerado pela NeoIDL

Em *DbcConditions* estão as funções que carregam a lista de pré e pós-condições para cada serviço. A chamada para as pré e pós-condições baseadas em serviços também é construída nesse pacote. Esse pacote é fundamental para o funcionamento das construções de *Design-by-Contract*.

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 7. A classe do serviço, por fim, contém apenas a estrutura para implementação da lógica interna do serviço.

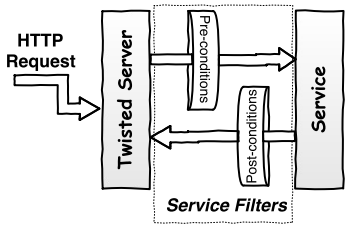

Figura 7: Modo de operação das pré e pós-condições no serviço Twisted

4.10. 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 8. Os módulos *PPService*, *PPUtils* e P*PDbcConditions* apenas imprimem um código Python fixo, sem qualquer interferência do módulo NeoIDL processado. *PPRoute* é um pequeno módulo (42 linhas) que processa as informações contidas nas instruções *path* dos serviços declarados no módulo NeoIDL e mapeia a correspondência entre as URIs e os serviços.

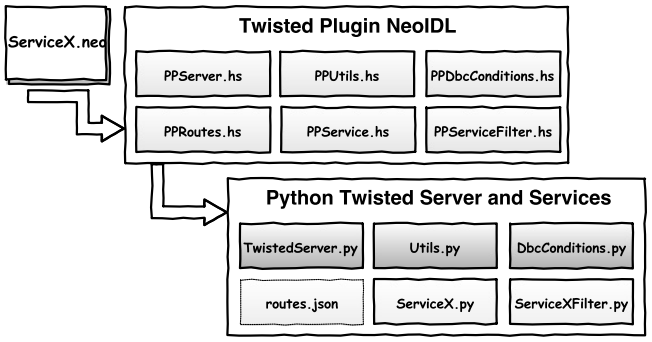

Figura 8: Plugin para geração de código Twisted com suporte a *Design-by-Contract*

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 seções: (i) a declaração das condições de *Design-by-Contract* e (ii) o carregamento dessas condições. A primeira seção, de declaração das condições *Design-by-Contract* contém a lista de pré e

pós-condições do serviço. A Figura 9 apresenta um exemplo de condição especificada em NeoIDL transformada em código *Python Twisted*.

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

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

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

Figura 9: Transformação de pós-condição NeoIDL (lado esquerdo) em código Python *Twisted* (lado direito)

Do lado direito da Figura 9 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 4.7), a pós-condição é carregada com a indicação de que os argumentos devem ser lidos do corpo da resposta do serviço (linha 7).

A segunda seção da classe *ServiceFilter* é genérica para qualquer serviço. A Figura **??** 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).

4.11. ESTUDO EMPÍRICO SUBJETIVO

A meta de investigar a utilidade do uso de construções de *Design-by-Contract* na NeoIDL estruturada conforme o método GQM é apresentada na Tabela 1.

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

Tabela 1: Meta de investigar o uso de Dbc na NeoIDL estruturado conforme método GQM

O modelo escolhido para subsidiar a elaboração das perguntas foi o *Technology Acceptance Model* – TAM, proposto por Davis em 1989 [**?**] e que possui uma sólida base teórica

e ampla utilização. O principal objetivo do modelo TAM é identificar os motivos que levam à aceitação ou rejeição de uma tecnologia. O modelo está estruturado em dois conceitos: percepção da utilidade (*perceived usefullness* - PU) e percepção sobre a facilidade de uso (*perceived ease of use* - PEU) [**?**].

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

As respostas normalmente são escalas que vão desde a discordância total até a concordância total quanto a afirmação apresentada. Essa escala é denominada *Likert Scale* [**?**]. As respostas são agrupadas para se avaliar se, no conjunto, as respostas mais favoráveis indicam uma tendência de aceitação. O processo de elaboração do questionário é apresentado na Subseção **??**.

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

4.12. Questionário

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

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

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

A base da segunda seção do questionário é um conjunto de especificações de serviços. Inicialmente, é apresentada em texto descritivo, a especificação de um serviço que deve

recuperar uma lista de informações, desde que atendida uma condição. Em seguida, o mesmo serviço é especificado em Swagger. Na sequência, a especificação é feita em NeoIDL. Ao final, um trecho do código Java que implementa o serviço real é apresentado.

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

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

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

4.13. Análise dos Resultados

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

4.13.1. Perfil dos respondentes

A primeira questão tinha o objetivo de apenas identificar respostas submetidas por pessoas fora do público alvo. Cinco instituições das quais se conhecia o nível técnico e com as quais se tinha contato com profissionais concentraram a maior parte das respostas. As demais respostas são empresas relacionas a essas instituições, por indicação de técnicos experientes do primeiro grupo. Sob este aspecto, nenhuma das respostas foi considerada anormal e, portanto, todas foram validadas, totalizando 26 respondentes.

A segunda questão levantou o tempo de experiência dos respondentes com desenvolvimento de sistemas Web. O resultado foi sumarizado na Tabela 2. Como o público alvo, definido na meta GQM (Tabela 1), é de desenvolvedores experientes, os respondentes que apontaram experiência inferior a três anos estão fora do perfil esperado e suas respostas foram descartadas (linhas em cinza). Assim, o questionário teve um conjunto de 21 respostas válidas.

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

Tabela 2: Respostas sobre a experiência com desenvolvimento Web

A terceira questão identificou o conhecimento dos respondentes com desenvolvimento de *Web Services*. Conforme apresentado na Tabela 3, as respostas estão polarizadas, em que 40% (quarenta por cento) tem nenhuma ou pouca experiência (menos de dois anos). Por outro lado, 23% (vinte e três por cento) dos respondentes possui experiência muito alta (mais de 10 anos). Caso o volume de respostas fosse muito grande (mais de 200 pessoas), poderia ser feita uma análise comparativa da NeoIDL de acordo com a experiência de *Web Service*.

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

Tabela 3: Respostas sobre experiência com *Web Services*

A questão quatro identificou o conhecimento e experiência do respondente com o modelo arquitetural REST. Apenas dois (menos de 10%) respondentes não tem experiência alguma com desenvolvimento de *Web Services* REST, como pode ser verificado na Tabela 4. Outros 42% (quarenta de dois por cento) dos respondentes, porém, tem experiência e conhecimento sobre os padrões arquiteturais que formam o modelo REST.

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

Tabela 4: Respostas sobre conhecimento e experiência com REST

A última questão sobre o perfil do respondente tratou do conhecimento e experiência com a especificação de contratos em Swagger. O conjunto de respostas confirmou a tendência pelo mercado pela uso de Swagger. Mais de 70% (setenta por cento) dos respondentes já tiveram algum contato com especificação de contratos em Swagger (Tabela 5). Esse cenário

indica que o conjunto de profissionais é qualificado e possui capacidade crítica para responder às demais questões.

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

Tabela 5: Conhecimento e experiência dos respondentes com Swagger

4.13.2. Análise da especificação de contratos formais e expressividade de Swagger e NeoIDL

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

Na questão 6 foi apresentada uma afirmação sobre a efetividade de se especificar contratos formalmente, em detrimento de especificação textual descritiva. Conforme apresentado na primeira barra da Figura 10, praticamente todos os respondentes concordaram com a afirmação e nenhum discordou. Esse resultado reforça a utilidade de se utilizar contratos formais para especificação de serviços.

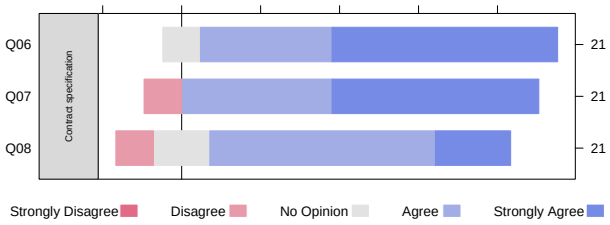

Figura 10: Gráfico com o resultado das questões 6 a 8

A questão 7 consultou a percepção dos respondentes sobre a facilidade de identificar as operações e atributos em um contrato especificado em Swagger. A questão 8 contém a mesma afirmação sobre um contrato especificado em NeoIDL. O resultado de ambas questões estão na Figura 10 e indicam que tanto os contratos em Swagger como em NeoIDL são claros e fáceis de se compreender. Vale ressaltar que a maior parte dos respondentes já possuia contato anterior com Swagger (Tabela 5), o que reforça o resultado positivo para a NeoIDL.

4.13.3. Análise sobre aceitabilidade de construções de Design-by-Contract

Entre as questões 9 e 12 foram feitas afirmações sobre a facilidade de se identificar os elementos e a utilidade de se especificar pré e pós-condições em contratos de serviços REST com a NeoIDL. Os resultados são apresentados na Figura 11.

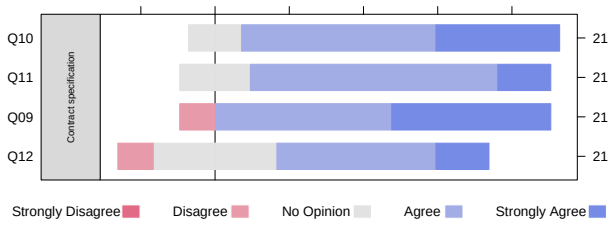

Figura 11: Gráfico com o resultado das questões 9 a 12

A questão 9 tem a afirmação de que o conhecimento explícito de precondição terá utilidade para o desenvolvedor responsável por implementar o serviço. Quase todos os respondentes concordaram com a afirmação, em maior e menor grau. Um pequeno conjunto discordou, indicando que outros elementos podem ser necessários para esse conjunto identificar a utilidade. Pode haver influência ainda da experiência do respondente com implementação de serviços.

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

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

4.13.4. Análise da geração de código para construções de Design-by-Contract

Duas questões foram inseridas para tratar da utilidade da geração de código com a NeoIDL para contratos com suporte a *Design-by-Contract*. A questão 13 apontava ser claro o efeito da precondição sobre o código gerado. A maior parte dos respondentes indicou que a geração do código para a precondição é util em relação a produção de efeitos controláveis. A Figura 12 apresenta os resultados.

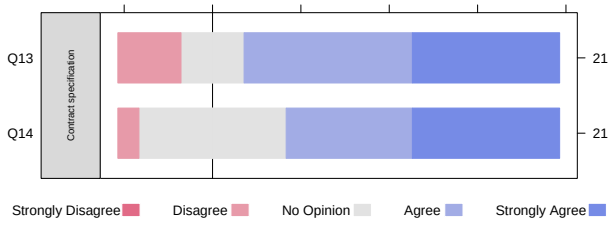

Figura 12: Gráfico com o resultado das questões 13 e 14

A questão 14 afirmou que a geração de código sobre precondições ampliará a produti-

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

4.13.5. Análise sobre a predisposição ao uso

As duas últimas questões tratam de quão inclinado ao uso da NeoIDL em seu ambiente de trabalho estaria o respondente. Esta análise é relativa [?], ou seja, depende de quais os benefícios trazidos pela NeoIDL em relação aos benefícios trazidos por soluções alternativas, como especificação textual de contratos ou ainda por meio de Swagger. Os resultados para essas questões são apresentadas na Figura 13.

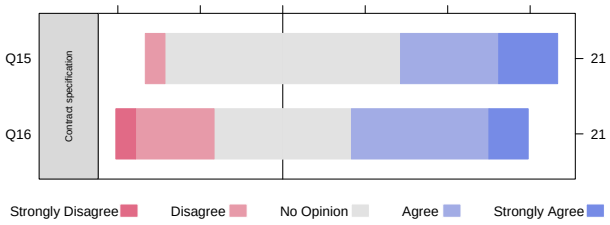

Figura 13: Gráfico com o resultado das questões 15 e 16

A questão 15 apresenta uma afirmação sobre a predisposição ao uso regular da NeoIDL no futuro. Pelos resultados da pesquisa, verifica-se que a maior parcela dos respondentes nem concorda nem discorda. Esse resultado é relevante, pois, conforme debatido anteriormente (Tabela 5), boa parte dos respondentes já tem contato com Swagger, que consiste em uma alternativa direta à NeoIDL. Se, por um lado os respondentes não concordam prontamente em utilizar NeoIDL, por outro não descartam.

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

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

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

4.14. Ameaças

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

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

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

Outro ponto de melhoria são ações que visem proporcionar um conhecimento mínimo sobre a NeoIDL. Todas as respostas foram dadas apenas pelas informações apresentadas no próprio questionário e os resultados poderiam ser diferentes se houvesse um treinamento prévio sobre a NeoIDL. Sob esse aspecto, tem-se que a compreensão sobre dos elementos da sintaxe básica na NeoIDL (questão 6) teve resultado inferior que a compreensão dos elementos de *Design-by-Contract* (questão 8). Esse resultado é inesperado, pois pressupõe-se que identificar os elementos de *Design-by-Contract* só seja possível após compreender a sintaxe básica.

Ainda, como o questionário foi aplicado apenas uma vez, não foram incluídos pontos de melhoria a partir do primeiro conjunto de respostas para uma segunda avaliação. Conforme debatido na Subseção 4.13, algumas questões poderiam ser inseridas em uma nova aplicação do questionário, de modo produzir informações que permitissem conclusões mais completas.

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

5. Conclusão

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

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

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

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

A NeoIDL passou a permitir a validação dos parâmetros de entrada e saída de uma requisição (por meio de pré e pós-condições básicas) assim como permitir também fazer requisição a outros serviços para realizar validações mais complexas, por meio de pequenas composições de serviços. A construção de um *Plugin Twisted* demonstrou ser viável produzir código que realize, em tempo de execução, a validação das regras estabelecidas no contrato, sem que o desenvolvedor tenha que se preocupar com elas e direcione seu esforço para a implementação das regras de negócio.

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

5.1. TRABALHOS FUTUROS

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

Ainda sobre a proposta apresentada nessa dissertação, a análise dos resultados (Seção 4.13) levanta a possibilidade de exploração de algumas respostas, em especial dos fatores que levaram uma parte importante dos respondentes a informar que tem dúvida (nem discordam nem concordam) com o adoção da NeoIDL em suas atividades. Um novo questionário poderia ser elaborado sobre esse ponto.

A realização de um experimento que avalie o uso por completo da abordagem de pré e pós-condições desde a especificação dos requisitos, a elaboração dos contratos, até os testes dos serviços implementados em um cenário hipotético poderia trazer mais insumos para a melhoria da NeoIDL. Ainda mais relevante seria a experimentação da NeoIDL com *Design-by-Contract* em um cenário de serviços reais.

Sob a perspectiva de implementação, podem ser elaborados *plugins* para outras linguagens de programação além de *Python Twisted* com suporte a *Design-by-Contract*, de modo

a ampliar o potencial de utilização do *framework*. O projeto da NeoIDL também pode incorporar mecanismos que facilitem a especificação de contratos, como recursos ambientes de desenvolvimento (IDEs) e *code complete*. Ainda um portal de onde podessem ser gerados os códigos sem a necessidade de instalação local do *framework*.

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

Apêndice A. Section in Appendix

Sample text. Sample text. Sample text. Sample text. Sample text. Sample text. Sample text. Sample text. Sample text. Sample text. Sample text. Sample text. Sample text.

Referências