REST-Unit: Geração baseada em U2TP de *Drivers* de Teste para RESTful Web Services

Filipe Borges [1], Marcelo Pimenta [1], Taisy Weber [1],

[1] Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brasil
{fqborges, mpimenta, taisy}@inf.ufrgs.br

Abstract. Web services are a powerful paradigm to develop and integrate web applications. In this arena, RESTful web services are gaining increasing popularity. They are based on well know web standards and also on REpresentational State Transfer, a software architecture for distributed system. However, REST isn't a formal standard and it can lead to cohesion and integration problems, forcing the need of costly testing tasks, impacting the project viability. To overcome this problem, we propose REST-Unit, a solution for automatic generation of test drivers. These drivers aim to validate the behavior of RESTful web services through models specified in the U2TP. Our solution shows many advantages comparing to traditional approaches to test RESTful web services. Furthermore, it contributes to step up U2TP standard, separates the roles of developers and testers, and strengthens the best practices of REST.

Keywords: web services, REST, U2TP, UML, test, code generation.

1. Introdução

Nos últimos anos *web services* surgiram para integrar sistemas, aplicações e processos. Dentre os de maior crescimento em uso – por ser um modelo simples e com fundamentação sobre padrões Web básicos como HTTP, URI e XML – estão os *RESTful web services – web services* baseados em REST (*REpresentational State Transfer*), doravante abreviados ***WSBRest***. Há mostras deste crescimento: em 2004 a Amazon já disponibilizava alternativamente versões SOAP e REST de sua API e o acesso à versão REST compreendia 85% do volume de acessos total [1]. Porém, REST não possui uma padronização rigorosa, o que traz a necessidade de uma disciplina de uso através da adoção de algumas boas práticas já aceitas e de teste constante durante todo o desenvolvimento.

No contexto do desenvolvimento de *WSBRest*, um dos principais objetivos do teste é buscar por violações às boas práticas adotadas no desenvolvimento da aplicação. Atividade de testes deve ser contínua e integrada ao desenvolvimento, ocorrendo desde o seu início e sendo mantida ao longo de todo o projeto, o que pode tornar-se muito custoso. Uma técnica amplamente utilizada para diminuir este custo é a

automação de testes, por reduzir drasticamente o tempo de execução dos mesmos e possibilitar a verificação automática dos resultados permitindo que sejam facilmente repetidos. Algumas vezes, porém, o custo de desenvolvimento de *drivers* de testes (código que rege a execução dos testes) pode não compensar as vantagens dos testes automatizados, o que nos leva a buscar soluções para o possível déficit de produtividade no desenvolvimento dos testes.

Este trabalho apresenta REST-Unit, uma solução para gerar automaticamente os *drivers* de testes para validação do comportamento de *WSBRest* através de modelos especificados no padrão U2TP (Perfil de Teste da UML 2.0). O objetivo desta solução é aumentar a produtividade do desenvolvimento de testes trabalhando em mais alto nível, diminuindo o tempo gasto com codificação de testes automatizados, de forma a integrar a atividade de teste em todo o ciclo de desenvolvimento a um custo aceitável. Assim, desde o início do desenvolvimento, os *WSBRest* podem ser testados e validados, ganhando-se em qualidade e garantindo que boas práticas estão sendo efetivamente aplicadas.

Este artigo é estruturado como segue. A seção 1 apresentou a motivação deste trabalho. A seção 2 resume conceitos de *WSBRest* e do estilo REST. A seção 3 apresenta REST-Unit, desde a especificação do teste até a geração do código. A seção 4 resume os trabalhos relacionados. A seção 5 conclui o presente trabalho.

2. RESTful web services: fundamentos do estilo arquitetural REST

Os *WSBRest* têm uma grande aceitação pelas suas características como simplicidade, escalabilidade e versatilidade herdadas do estilo arquitetural *Representational State Transfer* (REST), muitas vezes considerado como uma filosofia de *design*. Esta seção resume características principais de REST, fundamental para entender os *WSBRest*.

REST busca alinhar o desenvolvimento de aplicações com a arquitetura da Web [2]. É descrito como uma rede de *websites* (um estado virtual), onde o usuário progride com uma aplicação selecionando as ligações (transições do estado), tendo como resultado a página seguinte (que representa o estado seguinte). O estilo REST aplica algumas restrições sobre os elementos da arquitetura, como i) Separação de conceitos, interface e responsabilidades entre cliente e servidor; ii) Sem estado: não é permitido estado de sessão entre cliente e servidor; toda requisição do cliente contém todos dados necessários para ser entendida pelo servidor; iii) *Cache*: o resultado de uma requisição individual pode ser armazenado para uso posterior; e iv) Interface uniforme: a interface de acesso entre os componentes deve ser uniforme. Mais detalhes sobre REST podem ser encontrados na literatura [1], [2], [3] e [4].

Os *WSBRest* são implementados usando os padrões que fizeram da Web o maior sistema distribuído existente. Eles utilizam-se da definição do padrão HTTP que contém métodos de acesso (GET, POST, PUT, DELETE, ...), códigos de status (200 (OK), 404 (Not Found), ...) e cabeçalhos HTTP (Accept, User-Agent, Cache-Control, entre outros). *WSBRest* são um padrão de fato, mas não são um padrão definido por algum organismo. Há, no entanto, um consenso sobre diversas práticas que se tornaram modelos por se apresentarem adequadas à filosofia que acompanha a arquitetura REST. Entre estas práticas incluem-se: o uso correto dos métodos HTTP

(também chamados de verbos) normalmente associados às operações CRUD; a semântica adequada do endereçamento dos recursos; e a manutenção de um padrão homogêneo, público e bem documentado, em toda aplicação, em todos recursos, em todas as situações. Mais detalhes sobre estas práticas e um conjunto de boas práticas adicionais podem ser encontrados na literatura [1], [2], [3] e [4].

Devido à falta de um rigor na padronização, muitas das características de cada aplicação ficam dependentes do bom senso dos analistas, projetistas e desenvolvedores. Para conseguir alcançar os níveis de qualidade aceitáveis pelo mercado, um projeto que use serviços *RESTful* deve possuir um bom planejamento e um grande investimento na disciplina de teste de software.

3. REST-Unit: de U2TP a geração de código de teste

Esta seção apresenta uma visão geral de REST-Unit, desde a modelagem de testes de *WSBRest* em U2TP até a geração de código para Test::Unit, o *framework* de teste unitário padrão adotado. Nossa meta é propor uma abordagem para testes que apresente os benefícios dos testes automatizados, porém sem o grande custo associado à implementação dos casos de teste. Os requisitos que nortearam esta abordagem são: utilizar modelagem dos casos de testes em uma linguagem alto nível; usar o perfil U2TP para especificação dos casos de testes; ter o código necessário para execução do teste gerado automaticamente; possibilitar execução automática dos testes; possibilitar a verificação automática dos resultados do teste.

A Fig. 1 apresenta os três passos que definem a idéia geral dessa proposta: especificação, exportação e geração. Na especificação, usando uma ferramenta de modelagem UML, são criados diagramas de classe e sequência que modelam os testes em U2TP. Na exportação, o modelo dos testes é exportado para o formato XMI. Na geração, o arquivo em formato XMI é usado como entrada para geração de *drivers* ou seja, do código que guia a execução dos testes.

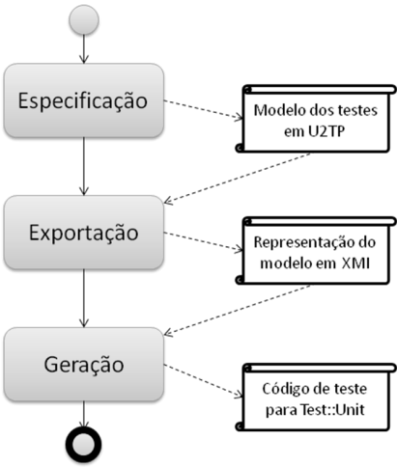

Fig. 1. REST-Unit: da especificação dos testes à geração de código.

3.1. Especificação: modelagem de testes em U2TP

Esta seção apresenta uma abordagem para especificar testes de *WSBRest* utilizando modelos do Perfil de Teste da UML 2.0 (UML 2.0 Test Profile ou abreviadamente U2TP). U2TP é um metamodelo MOF baseado na especificação da UML 2.0 para modelagem de teste caixa-preta e define uma linguagem para projetar, especificar, visualizar, analisar, construir e documentar artefatos de sistemas de teste [5]. Mais detalhes sobre U2TP podem ser encontrados na literatura técnica [5].

A modelagem dos testes é baseada na apresentada por Biasi e Becker [6] e é dividida em duas partes: modelagem estrutural e modelagem comportamental. Na modelagem estrutural é definido um diagrama de classes UML, que descreve a estrutura dos elementos que constituem os testes. Os elementos desta modelagem são ilustrados na Fig. 2.

Uma classe com o estereótipo *TestContext* modela o *test fixture,* ela deve ser única e conter pelo menos uma operação *TestCase*. Cada caso de teste é uma operação com estereótipo *TestCase* pertencente ao *TestContext*, esta não deve receber parâmetros e não deve ter valor de retorno e seu nome deve iniciar com o "test_". O *WSBRest* a ser testado é uma classe com o estereótipo *Sut,* e deve ter os atributos que o identificam – endereço de rede (atributo *server*), a porta (atributo *port*) e opcionalmente um caminho (atributo *path*) – com valores inicializados. A classe HttpStatusCode servirá como container dos códigos de status HTTP, cada um é modelado como um atributo com estereótipo *enum* e o valor padrão identificando o seu código. Opcionalmente poderá haver mais classes com o estereótipo *TestComponent*. Essas outras classes servirão para modelar dados que servem de entrada ou que são retorno das operações do *web service*, por exemplo dados XML que precisam ser passados em uma operação POST.

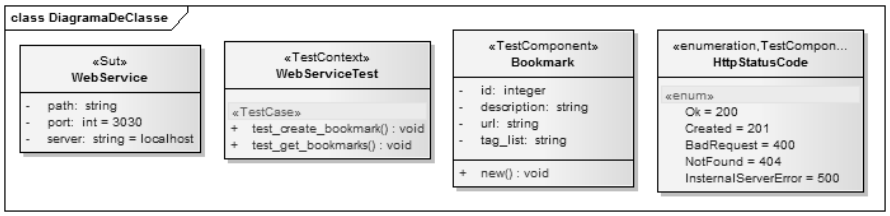

Fig. 2. Diagrama de classe: modelagem estrutural do teste.

Na modelagem comportamental define-se o como é testado. Para esta modelagem são usados diagramas UML de sequência. Um exemplo pode ser visto na Fig. 3. Esse diagrama tem as seguintes características: a primeira mensagem inicia a execução do caso de teste; a criação de um objeto é modelada como uma mensagem pontilhada partindo de TestContext; a penúltima mensagem do caso de teste é um acesso ao *WSBRest*, e quando a operação for POST ou PUT, esta deve ter como parâmetro um objeto TestComponent; a última mensagem dentro do caso de teste é o código de status retornado pelo acesso ao *WSBRest* e deve ser um dos valores da enumeração HttpStatusCode; a última mensagem do diagrama é o retorno da execução do caso de teste, que sempre é *pass*, e representa um *veredic pass*.

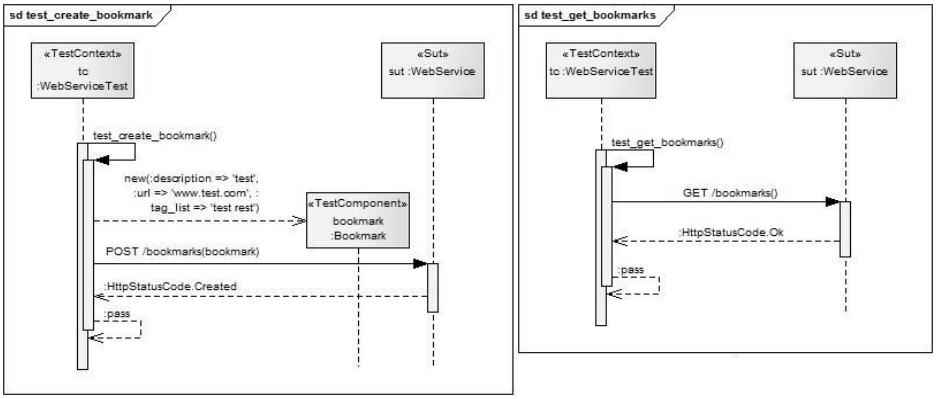

Fig. 3. TestCase modelados em diagrama de sequência definem o comportamento dos testes.

3.2. Exportação: representação do modelo U2TP em formato XMI

Para permitir a geração de código de teste a partir de modelos U2TP, este deve ser exportado para formato XMI, formato baseado em XML padronizado pela OMG. Este mapeamento é especificado em [7], e é suportado por ferramentas de modelagem UML. A Tabela 1 resume como elementos U2TP são representados em XMI.

Tabela 1. Mapeamento de U2TP para XMI.

U2TP	*XMI*	*Observação*
TestContext	uml:Stereotype	Estereótipo TestContext
	uml:Class	Classe que representa o TestContext
	uml:Lifeline	TestContext no diagrama de sequência
Sut	uml:Stereotype	Estereótipo Sut
	uml:Class	Classe que representa o e Sut
	uml:Lifeline	Elemento Sut no diagrama de sequência
TestComponent	uml:Stereotype	Estereótipo TestComponent
	uml:Class	Classes TestComponent
	uml:InstanceSpecification e uml:Lifeline	TestComponents no diagrama de sequência
TestCase	uml:Stereotype	Estereótipo TestCase
	Operation em uml:Class	Métodos TestCase do TestComponent
	uml:Message	Caso de teste no diagrama de sequência

3.3. Geração: código Test::Unit a partir do XMI

Para que os *drivers* de teste possam ser gerados automaticamente há a necessidade de: i) identificar o mapeamento desejado entre o modelo U2TP e o código de teste e ii) desenvolver um algoritmo que, a partir do XMI, gere este código. O código será executado sobre Test::Unit, o *framework* padrão de testes unitários em linguagem Ruby. Este foi escolhido pois Ruby é cada vez mais popular e muito relacionada com aplicativos Web 2.0 e REST, graças ao *framework* Ruby on Rails. Foi escolhido também porque o *framework* Test::Unit é um *framework* da família xUnit, assim facilmente pode-se ter o código gerado para, por exemplo, Junit ou Nunit.

Mapeamento U2TP para Test::Unit

O seguinte exemplo ilustra o mapeamento de uma modelagem em U2TP (Fig. 2 e 3) para Test::Unit (Fig. 4). As linhas gerais do mapeamento estão descritas na tabela 2.

Tabela 2. Mapeamento dos elementos U2TP para código em Test::Unit

U2TP	*Test::Unit*	*Exemplo no código (Fig 4)*
TestContext	Classe derivada de Test::Unit::TestCase	Região 1
Sut	Atributos que identificam o *web service*	Região 2
TestCase	Método na subclasse de Test::Unit::TestCase	Regiões 3 e 4
TestComponent	Classe HttpStatusCode / Classe com implementaçao de to_xml	Regiões 5 e 6

```
require 'test/unit'
require 'net/http'

class HttpStatusCode                                                    5
  def self.ok() 200 end
  def self.created() 201 end
  def self.bad_request() 400 end
  def self.not_found() 404 end
  def self.internal_server_error() 500 end
end

class Bookmarks                                                         6
  attr_accessor :id
  attr_accessor :description
  attr_accessor :url
  attr_accessor :tag_list

  def initialize( params )
    @id = params[:id]
    @description = params[:description]
    @url = params[:url]
    @tag_list = params[:tag_list]
  end

  def to_xml
    xml = "<bookmark>"
    xml << "<id type='integer'>#{id}</id>" unless id.nil?
    xml << "<description>#{description}</description>" unless description.nil?
    xml << "<url>#{url}</url>" unless url.nil?
    xml << "<tag_list>#{tag_list}</tag_list>" unless tag_list.nil?
    xml << "</bookmark>"
  end
end

class TestWebService < Test::Unit::TestCase                             1
  Server = 'localhost'
  Port = 3000                                                           2
  Path = ''

  def test_get_bookmarks
    req = get '/bookmarks.xml'
    assert_equal HttpStatusCode.ok, req.code.to_i
  end                                                                   3

  def test_create_bookmark
    bookmark = Bookmarks.new (
      { :description => 'test', :url => 'test.org', :tag_list => 'test rest' }
    )
    req = post '/bookmarks.xml', bookmark.to_xml
    assert_equal HttpStatusCode.created, req.code.to_i
  end                                                                   4

  def get( path )
    Net::HTTP.start(Server, Port) { |http| http.get(Path + path) }
  end

  def post( path, payload )
    Net::HTTP.start(Server, Port) { |http| http.post(Path + path, payload) }
  end

  def put( path, payload )
    Net::HTTP.start(Server, Port) { |http| http.put(Path + path, payload) }
  end

  def delete( path )
    Net::HTTP.start(Server, Port) { |http| http.delete(Path + path) }
  end
end
```

Fig. 4. Código em Test::Unit para o teste modelado nas figuras 2 e 3. As regiões (*1, 2, 3, 4 e 5*) identificam os mapeamentos apresentados na tabela 2.

Gerador de código para Test::Unit

O objetivo dessa sessão é apresentar em linhas gerais um algoritmo que a partir do documento XMI gere *drivers* de teste em Test::Unit levando em conta a especificação do mapeamento entre U2TP e XMI e o mapeamento entre U2TP e Test::Unit.

Para permitir a geração do código é necessário identificar no documento XMI os elementos chave que serão os pontos iniciais da geração. Esses elementos são: HttpStatusCode, os diversos TestComponent e o TestContext. No algoritmo aqui apresentado, diversas classes dividem as responsabilidades da geração do código.

A classe HttpStatusCode é responsável pelo elemento HttpStatusCode do modelo. Sua inicialização busca o elemento HttpStatusCode e seus atributos no XMI. A classe TestComponent é responsável pelos elementos TestComponent do modelo. Na inicialização cada objeto recebe o nome do TestComponent e o conteúdo do documento XMI, busca o respectivo elemento TestComponent e identifica os seus atributos. Cada TestComponent é identificado por ser uma classe, pelo nome e pelo estereótipo «TestComponent». A classe TestContext é responsável pelo TestContext do modelo, identificado como sendo a única classe UML estereotipada «TestContext». A inicialização (Fig. 5) recebe o conteúdo do documento XMI e busca o elemento TestContext, os nomes dos TestCases, inicializa cada um e adiciona no *array* `@test_cases` e, por fim, busca os atributos do elemento SUT e atribui aos campos `@port`, `@server` e `@path`.

A classe TestCase é responsável pelos elementos casos de teste (TestCase) do modelo. A inicialização cada objeto recebe o nome do TestCase e o conteúdo do documento XMI, busca as interações que representa o TestCase e lê os passos necessários para a execução do teste, inicializando e armazenando no *array* `@passos`. Cada um desses passos pode ser uma instanciação, uma requisição ou uma asserção.

Os passos do caso de teste são tratados pelas classes Instanciacao, Requisicao e Assercao. A classe Instanciacao é responsável por um passo de instanciação. Um objeto da classe Instanciacao é inicializado com o nome da classe a ser criada, o nome do objeto criado e os parâmetros passados para o construtor. A classe Requisicao é responsável por um passo de requisição. Um objeto da classe Requisicao é inicializado com o identificador do recurso relativo ao endereço raiz do *WSBRest*, um método HTTP e opcionalmente o nome de um objeto parâmetro. A classe Assercao é responsável por um passo de asserção. Um objeto da classe Assercao é inicializado com o código HTTP esperado como resultado.

Uma última classe, a classe Gerador, tem como responsabilidade iniciar o *parse* do documento XMI, delegando as responsabilidades para as classes correspondentes e gerar o código também delegando as responsabilidades para as classes correspondentes. Para isso a classe Gerador tem um método construtor, `new`, e um método `gerar_codigo`. O método construtor é o que dispara a inicialização de todos os outros objetos das outras classes envolvidas. O método construtor tem os seguintes passos: inicialização do HttpStatusCode; descoberta e inicialização dos TestComponents; inicialização do TestContext. O método `gerar_codigo` da mesma forma que o construtor, delega a responsabilidade para geração de código para cada classe especializada. Um esqueleto inicial é gerado e, dentro dele, adicionado o código gerado pelos objetos HttpStatusCode, TestComponents e TestContext. O TextContext também delega a responsabilidade da geração de código de cada caso de

teste para os objetos TestCase, que por sua vez delegam a geração de código de cada passo para os respectivos objetos. Este comportamento é ilustrado no diagrama na Figura 6.

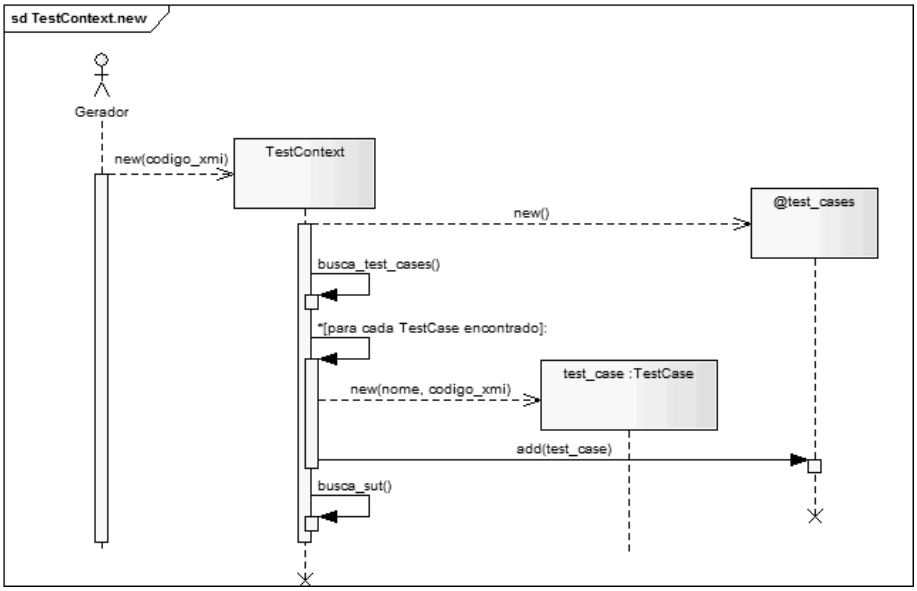

Fig. 5. Inicialização de um objeto TestContext.

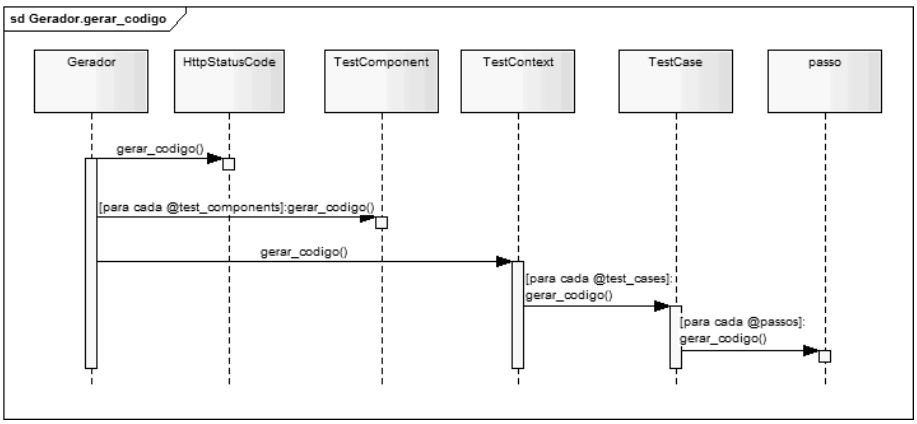

Fig. 6. Comportamento da geração de código utilizando a classe Gerador.

3.4. Resultados

REST-Unit é uma solução para gerar os *drivers* de teste a partir de um modelo U2TP. O modelo, quando exportado para XMI, permite que, como visto na seção 3.3, gere-se

o código de teste para Test::Unit. Pretende-se que a qualidade deste código seja pelo menos tão boa quanto um código equivalente escrito manualmente. O tempo despendido na especificação dos casos de teste é compensado pelo tempo economizado com a geração do código de testes. Além disso, tem-se como vantagem o modelo bem documentado dos casos de teste, e a qualidade maior que se alcança trabalhando em um nível mais alto de abstração. Um protótipo foi implementado para validação do algoritmo, e aplicado sobre o exemplo apresentado neste artigo. A partir do modelo de teste (fig. 2 e fig. 3), o protótipo gerou o código de teste que pode ser visto na fig. 4.

4. Trabalhos relacionados

Esta seção resume os trabalhos relacionados a REST-Unit: abordagens cuja meta final é obviamente o teste, mas cuja forma de proceder e o esforço envolvido é bem diferente.

Teste manual: Com essa abordagem é adicionado um grande *overhead* de execução dos testes, pois é necessária a alocação de profissionais para execução manual dos casos de teste. A execução é a fase dos testes que apresenta maior custo. É necessário confiar no testador para execução dos testes e conferência dos resultados. Entretanto, teste manual é ainda muito aplicado para teste de web services durante o desenvolvimento, principalmente em projetos pequenos. Exemplos de ferramentas que suportam teste manual são: NetBeans [8], que a partir da versão 6.0 possui suporte para desenvolvimento e teste de *WSBRest*; cURL [9], ferramenta para transferência de dados com endereçamento URL; RESTClient [10], uma aplicação Java Swing desenvolvida para testar *WSBRest*.

Teste automatizado: Neste caso, a execução dos casos de testes é conduzida automaticamente por uma ferramenta. Porém existe um custo adicional de implementação, pois os testes precisam ser programados de alguma forma. Nesta abordagem o maior custo incide sobre a fase de implementação dos casos de teste. Ferramentas que suportam esta abordagem são por exemplo Test::Unit, o *framework* de teste unitário padrão em Ruby, e outros *frameworks* de teste unitário da família xUnit, como NUnit e JUnit.

Nenhuma abordagem exige o uso de uma modelagem de alto nível dos casos de teste, bem como o uso do padrão U2TP para modelagem dos casos de teste. Na abordagem de teste automatizado pode-se fazer uso de ferramentas para auxiliar na criação do código, mas grande parte do código dos *drivers* de teste deve ser escrito pelo programador. Na abordagem de teste manual, pode-se utilizar ferramentas para auxiliar na criação dos dados necessários aos testes. Aliás, nenhuma abordagem que gere automaticamente os *drivers* de teste para *web services* foi encontrada durante a pesquisa para realização deste trabalho. REST-Unit foi criado para preencher justamente estas lacunas.

5. Conclusões

Neste artigo introduzimos REST-Unit, uma abordagem para geração automática de *drivers* de teste a partir de modelos especificados em U2TP visando a validação de comportamento de *WSBRest.* Todas as fases (a saber, especificação, exportação e geração do código de teste) de REST-Unit foram explicadas e exemplificadas. Como vimos, esta solução contribui para separar os papéis de quem projeta testes, quem os implementa e executa. Contribui também para fortalecer as boas práticas para o estilo REST e diminuir o esforço na implementação do código dos testes unitários para testar *WSBRest* através de sua geração automática.

Uma das limitações do trabalho decorre da dificuldade em modelar em U2TP conceitos relacionados aos cabeçalhos HTTP, por não haver um mapeamento intuitivo dessas entidades com os conceitos do U2TP. Não foram utilizados também todos os conceitos existentes de U2TP para teste unitário: DataPool e DataPartition por exemplo não foram usados.

Outras limitações são relacionadas ao uso de XML. Não foi definido um modelo para tratamento dos dados XML retornados pelo web service pela dificuldade de seu mapeamento no modelo sem que prejudicasse a clareza. Assim não foi possível reutilizar estes dados para validação, ou como entrada para novos acesso a esse serviço.

Como perspectivas de trabalhos futuros, incluem-se: a) especificação de conceitos para permitir tratamento de cabeçalhos HTTP; b) especificação de conceitos que permitam o tratamento dos dados retornados pelo *web service*, de forma a efetuar validações mais complexas destes dados e utilizá-los como entradas para outros acesso ao *web service;* c) adotar modelagem que permita o uso de outros conceitos do U2TP, como DataPool e DataPartition; d) permitir tratamento de mais tipos de dados e aumentar o domínio de *RESTful web services* que podem ser testados.

Referências

1. He, H: Implementing REST Web Services: Best Practices and Guidelines. O'Reilly (2004), http://www.xml.com/pub/a/2004/08/11/rest.html
2. Fielding, R: Architectural Styles and the Design of Network-based Software Architectures. Universidade da Califórnia, Irvine (2000), http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm
3. Richardson, L., Ruby, S.: RESTful Web Services. O'Reilly, Beijing (2007)
4. Ruby on Rails, http://www.rubyonrails.org
5. UML 2.0 Testing Profile Specification. Technical Report PTC/04-04-02. OMG (2004), http://www.omg.org/docs/ptc/04-04-02.pdf
6. Biasi, L., Becker, K.: Geração Automatizada de Drivers e Stubs de Teste para JUnit a partir de Especificações U2TP. PUCRS, Porto Alegre (2006)
7. MOF 2.0/XMI Mapping Specification. OMG (2005), http://www.omg.org/cgi-bin/doc?formal/2005-09-01
8. NetBeans home, http://www.netbeans.org
9. cURL e libcurl, http://curl.haxx.se/
10. RESTclient, http://code.google.com/p/rest-client/