UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

NADJIA JANDT FELLER

# Estendendo Rest-Unit: Geração Baseada em U2TP de *Drivers* e Dados de Teste para *RESTful Web Services*

Trabalho de Graduação.

Prof. Dr. Marcelo Soares Pimenta
Orientador

Porto Alegre, Julho de 2010.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos Alexandre Netto
Vice-Reitor: Prof. Rui Vicente Oppermann
Pró-Reitora de Graduação: Profa. Valquiria Link Bassani
Diretor do Instituto de Informática: Prof. Flávio Rech Wagner
Coordenador do CIC: Prof. João César Netto
Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

# AGRADECIMENTOS

Gostaria de agradecer aos meus pais, por todo seu amor, carinho e dedicação que sempre tiveram comigo, e pela compreensão com as crises nervosas e as semanas que passava “fechada” no quarto fazendo este trabalho.

Ao meu namorado, Daniel, companheiro de aventuras nestes anos de graduação, pelo apoio e carinho, fazendo com que os momentos de críticos na confecção deste e diversos outros trabalhos ficassem menos desesperadores. Também pela ajuda nos momentos em que o raciocínio travava e pela revisão do abstract deste trabalho.

Aos meus amigos, pelas muitas risadas, conselhos, teorias conspiratórias, noites de jantares e jogos, encontros no “osto”, almoços no RU e vários outros momentos. Nossos encontros e conversas me ajudaram na confecção deste trabalho, e às vezes “atrapalharam” um pouco (isto não é uma reclamação).

Ao meu orientador, Marcelo Pimenta, pelo apoio e orientação durante a confecção de todo este trabalho.

# SUMÁRIO

# LISTA DE ABREVIATURAS E SIGLAS

| | |
|---|---|
| API | Application Programming Interface |
| CORBA | Common Object Request Broker Architecture |
| DNS | Domain Name System |
| GUI | Graphical User Interface |
| HTML | Hypertext Markup Language |
| HTTP | Hypertext Transfer Protocol |
| IDL | Interface Description Language |
| JPEG | Joint Photographic Experts Group |
| JSON | JavaScript Object Notation |
| MIME | Multipurpose Internet Mail Extensions |
| MOF | Meta-Object Facility |
| NAT | Network Address Translation |
| OWL | Web Ontology Language |
| RPC | Remote Procedure Call |
| SOAP | Simple Object Access Protocol |
| UDDI | Universal Description, Discovery and Integration |
| UML | Unified Modeling Language |
| URI | Uniform Resource Identifier |
| URL | Uniform Resource Locator |
| U2TP | UML 2.0 Testing Profile |
| WS | Web Service |
| WSDL | Web Service Definition Language |
| XMI | XML Metadata Interchange |
| XML | Extensible Markup Language |
| YAML | YAML Ain’t Markup Language |

# LISTA DE FIGURAS

# LISTA DE TABELAS

# RESUMO

Os *RESTful Web services – Web services* baseados em REST (*REpresentational State Transfer*) – são uma solução que vem sendo amplamente utilizada para desenvolvimento de aplicações Web 2.0 e publicação de APIs na internet pela interface simples e de fácil entendimento, aliado ao suporte de *frameworks* de alta produtividade. Os *RESTful Web services*, assim como todos os sistemas de *software*, devem ser testados para que atinjam o nível de qualidade aceitável para que possam ser utilizados por outros sistemas com confiança. Essa disciplina de testes deve ser integrada ao desenvolvimento, ocorrendo desde o início do projeto e sendo aplicada ao longo de todo ciclo de vida.

No trabalho de diplomação de Filipe Borges (2009) foi proposta uma solução (REST-Unit) para gerar automaticamente os *drivers* de testes a partir de modelos especificados no padrão U2TP (UML 2.0 Test Profile), para validação do comportamento de *RESTful Web services*. Baseado em REST-Unit, foi desenvolvido neste trabalho REST-Unit+, cujo objetivo é estender a geração automatizada dos *drivers* de teste, propondo uma solução para relacioná-los com seus respectivos dados de teste, criando repositórios e partições de dados. Através disso, os testes gerados ficam mais completos e sua execução é facilitada, pois os tipos de dados aceitos nos testes já estão previamente especificados e documentados.

REST-Unit+ é uma solução para gerar os *drivers* e dados de teste a partir de um modelo U2TP. O modelo, quando exportado para XMI, permite que gere-se o código de teste. Um protótipo foi implementado para validação do algoritmo, e aplicado no decorrer de um exemplo que demonstra a aplicação completa desta solução. Este protótipo pode gerar a partir de um modelo o *driver* e os dados para testes de um Web Service RESTful.

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 em UML dos casos de teste e, principalmente, dos dados de teste (repositórios, partições e instâncias) utilizados para estes casos de teste, e a qualidade maior que se alcança trabalhando em um nível mais alto de abstração.

**Palavras-Chave:** U2TP, teste de software, dados de teste, geração de código de teste, RESTful Web Services, REST.

# Extending Rest-Unit: Drivers and Test Data Generation Based on U2TP for RESTful Web Services

## ABSTRACT

RESTful Web services are a solution that has been widely used for Web 2.0 development and API publication because of the simple and easy understanding interface, allied to high productivity framework support. RESTful Web services, as all software systems, must be tested to achieve an acceptable quality level so that can be used with trust by other systems. This test discipline should be integrated with development, since the beginning of the project and applied throughout the software development cycle.

At the graduation work of Filipe Borges (2009), a solution was proposed (REST-Unit) to automatically generate test drivers from models specified using U2TP (UML 2.0 Test Profile) pattern, to validate RESTful Web services behavior. In this project, based in REST-Unit, REST-Unit+ was developed, which goal is to extend test drivers automatic generation, proposing a solution to relate them with their test data, creating data pools and data partitions. Through this process, the generated tests become more complete and their execution is facilitated, because the accepted data types are already specified and documented.

REST-Unit+ is a solution for generating test drivers and test data from an U2TP model. The model, when exported to XMI, allows the test code to be generated. A prototype was implemented to validate the algorithm, and it was applied throughout an example that demonstrates the complete usage of this solution. This prototype can generate a RESTful Web service test driver and test data from a model.

The time spent on test case specification is compensated by the time saved with test code generation. Besides, it has as an advantage, the well documented UML test case model, and test data models (data pools, data partitions and instances) used in these test cases, and the bigger quality that is achieved working at a higher abstraction level.

**Keywords:** U2TP, software test, test data, test code generation, RESTful Web Services, REST.

# 1 INTRODUÇÃO

A *World Wide Web* é uma aplicação distribuída muito popular, e os *Web services* têm contribuído para que ela se torne uma plataforma distribuída cada vez mais poderosa. Neste contexto encontram-se os *RESTful Web Services*, que tentam atingir este objetivo de maneira simples, baseando-se no estilo arquitetural REST (REpresentional State Transfer) (Fielding, 2000). *RESTful Web Services* são uma solução que vem sendo amplamente utilizada para desenvolvimento de aplicações *Web* 2.0 e publicação de APIs na internet pela interface simples e de fácil entendimento, aliado ao suporte de *frameworks* de alta produtividade como Ruby on Rails (Ruby), Restlet (Java) e Django (Python).

Como qualquer aplicação, *Web services* também precisam ser testados, para assegurar seu funcionamento com qualidade aceitável. Essa disciplina de testes deve ser integrada ao desenvolvimento, ocorrendo desde o início do projeto e sendo aplicada ao longo de todo ciclo de vida. Porém, as ferramentas disponíveis exigem tempo do desenvolvedor, pois ele precisa escrever (e às vezes executar) cada teste manualmente e, em geral, individualmente.

No trabalho de Biasi (2006), é proposta uma abordagem para a geração automática de *drivers* de teste a partir de diagramas U2TP (*UML 2.0 Test Profile*) (OMG, 2005). Unindo estes conceitos, Borges (2009) propôs uma abordagem para a geração de *drivers* de teste para *RESTful Web Services*, com o objetivo de validar seu comportamento. Através disso é possível aumentar a produtividade do desenvolvimento de testes trabalhando em mais alto nível, diminuindo o tempo gasto com codificação de testes automatizados. Porém, nesta proposta ainda estão ausentes especificações referentes à geração de dados de teste.

## 1.1 Objetivo

Usando como ponto de partida o trabalho de Filipe Borges (2009), o objetivo deste trabalho é estender a geração automatizada dos *drivers* de teste, propondo uma solução para relacioná-los com seus respectivos dados de teste, criando repositórios e partições de dados. Através disso, os testes gerados ficam mais completos e sua execução é facilitada, pois os tipos de dados aceitos nos testes já estão previamente especificados e documentados.

### 1.1.1 Objetivos Específicos

Os objetivos específicos deste trabalho são:

- Avaliar a viabilidade da modelagem de dados de testes de *RESTful Web Services* utilizando o perfil da UML 2.0 para testes (U2TP);
- Propor uma solução de automatização de testes para esta modelagem;
- Implementar um exemplo de aplicação para avaliação e validação da solução apresentada.

### 1.1.2 Contribuições

Estão incluídas nas contribuições deste trabalho:

- Aumentar a produtividade do desenvolvimento de *RESTful Web Services*, diminuindo o esforço para desenvolvimento e execução dos testes;
- Através da geração de código de qualidade, contribuir para o fortalecimento das boas práticas para o estilo arquitetural REST;
- Diminuir o esforço para a implementação de dados de teste para *RESTful Web Services*, pois o código é gerado automaticamente;
- Contribuir para a especificação e documentação dos dados aceitos por *RESTful Web Services*;
- Contribuir para uma maior compreensão do uso do padrão U2TP de modelagem de testes.

## 1.2 Estrutura do Trabalho

Este trabalho está organizado da seguinte maneira: No capítulo 2 são apresentados os conceitos e características relacionados aos *RESTful Web Services* e ao estilo arquitetural REST; no capítulo 3 são apresentados os conceitos relativos a teste de software e dados de teste, incluindo os conceitos do U2TP, com suas características; no capítulo 4, trabalhos e ferramentas relacionados a este trabalho são citados; no capítulo 5, a solução REST-Unit+ é apresentada, juntamente com a solução de extensão para dados de teste proposta neste trabalho, um exemplo de aplicação e os resultados obtidos; e no capítulo 6 são descritas as conclusões, limitações e trabalhos futuros relativos a este trabalho.

# 2 RESTFUL WEB SERVICES E O ESTILO ARQUITETURAL REST

Como resultado do avanço da World Wide Web nas últimas décadas e da forma como esse avanço modificou a maneira como as empresas e os consumidores usam a Web, a indústria necessitou de um modelo de comunicação que permita que as aplicações de negócios possam se comunicar e trocar informações pela rede. A partir disso, surgiram os serviços Web, que permitem que aplicações interajam entre si, enviando e recebendo dados.

Como é apresentado por Richardson (2007) desde o uso de HTTP básico e XML até padrões como SOAP e WSDL os *Web services* cresceram muito em uso e complexidade, agregando uma diversidade de padrões. Para desenvolver um *Web service* baseado em SOAP é necessário conhecer XML, SOAP, WSDL, UDDI, WS-Policy, WS-Security, WS-Eventing, WS-Reliability, WS-Coordination, WS-Transaction, WS-Notification, WS-BaseNotification, WS-Topics, WS-Transfer, entre outros padrões.

Em busca de novamente utilizar a Web de forma simples, surgiu o modelo REST. Ele foi desenvolvido para por a base da Web de volta nos *Web services*, utilizando padrões básicos como HTTP, URI e XML (Richardson, 2007). Uma das capacidades trazidas pelos *RESTful Web services* é a possibilidade de facilmente desenvolver aplicações *Web* que possam ser utilizadas tanto por humanos quanto por máquinas.

O estilo arquitetural REST define um conjunto de princípios arquiteturais pelos quais podem ser projetados *Web services* que focam nos recursos de um sistema, incluindo como estados dos recursos são endereçados e transferidos sobre HTTP por uma ampla variedade de clientes escritos em diferentes linguagens. Se mensurado pelo número de *Web services* que o utilizam, REST emergiu nos últimos anos como um modelo de projeto de *Web services* predominante, pois é um estilo consideravelmente mais simples de ser utilizado.

Portanto, REST é considerado um avanço por ser mais simples, escalável e versátil do que soluções baseadas em RPC (Richardson, 2007). Para ilustrar isto, pode-se analisar os dados apresentados por He (2004): 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.

## 2.1 O Estilo Arquitetural REST

O estilo arquitetural REST surgiu a partir do trabalho de Roy Fielding (2000), baseado na análise de características de diversos tipos de arquiteturas de rede para a World Wide Web. Representional State Transfer destina-se a evocar uma imagem de como uma aplicação *Web* bem projetada se comporta: uma rede de páginas *Web* (uma máquina de estados virtual), onde o usuário progride com uma aplicação selecionando *links* (transições de estado), resultando na página seguinte (representando o próximo estado da aplicação) sendo transferida para o usuário e apresentada para seu uso.

Segundo Fielding (2000), uma arquitetura para a Web deve ser projetada no contexto de comunicar objetos de dados de grande granularidade em redes de alta latência e múltiplas fronteiras de confiança. Neste contexto é apresentado o estilo REST para sistemas hipermídia distribuídos, que fornece um conjunto de restrições em arquitetura que enfatizam a escalabilidade da interação entre componentes, generalidade de interfaces, implantação independente de componentes e componentes intermediários para reduzir a latência de interação, reforçar a segurança e encapsular sistemas legados. Estes objetivos são alcançados através das seguintes restrições de arquitetura para sistemas REST:

- **Cliente-servidor:** Através da separação de interesses, é possível melhorar a portabilidade da interface de usuário e melhorar a escalabilidade simplificando os componentes do servidor. Esta separação também permite que os componentes evoluam de forma independente, suportando múltiplos domínios organizacionais.
- **Comunicação sem estado (*stateless*):** cada requisição do cliente para o servidor deve conter toda a informação necessária para seu entendimento. No servidor não há nenhuma informação de estado armazenada, de forma que somente o cliente tenha esta informação. Isto induz as propriedades de visibilidade (apenas uma requisição é necessária para compreensão da comunicação), confiabilidade (torna mais fácil a recuperação de falhas parciais) e escalabilidade (servidor não consome seus recursos armazenando e gerenciando informações de estado). Uma desvantagem desta abordagem é o consumo da rede com dados repetitivos.
- ***Cache*:** para aumentar a eficiência do uso da rede (afetada pelos dados repetitivos), uma *cache* é utilizada pelo cliente. Isto exige que uma resposta a uma requisição seja marcada como "*cacheable"* ou "*noncacheable"*. Se uma resposta é "*cacheable"*, então o cliente tem direito a reusá-la posteriormente para requisições equivalentes. Como esta abordagem elimina algumas interações, ela melhora a eficiência, escalabilidade e o desempenho percebido pelo usuário. Uma desvantagem é a diminuição da confiabilidade, pois os dados armazenados em *cache* podem ser diferentes das respostas que seriam obtidas diretamente do servidor. Para amenizar isto, *caches* compartilhadas e *proxies* são utilizados.
- **Interface uniforme:** aplicando o princípio de generalidade às interfaces dos componentes, a arquitetura do sistema como um todo é simplificada e a visibilidade das interações é melhorada. Implementações são separadas do serviço que fornecem, o que encoraja a evolução independente. Uma

desvantagem desta abordagem é a degradação da eficiência, pois uma informação padronizada é utilizada, em detrimento de uma especializada para as necessidades da aplicação. Para atingir a interface uniforme, é necessário: identificação de recursos; manipulação de recursos através de representações; mensagens auto-descritivas; e hipermídia como motor do estado da aplicação.

- **Sistema em camadas:** permite que cada componente não veja além das camadas com quem se comunica. Isto promove independência de subsistema e diminui a complexidade do sistema. Camadas também podem ser utilizadas para encapsular serviços legados, simplificando componentes através do uso de um intermediário para a comunicação com o serviço legado. Intermediários também aumentam a escalabilidade do sistema, pois permitem balanceamento de carga entre múltiplas redes. A desvantagem dessa abordagem é o *overhead* e latência no processamento de dados, o que pode ser resolvido através de *caches* compartilhadas entre intermediários.
- **Código sob demanda (opcional):** permite que funcionalidades sejam estendidas através do *download* e execução de código, em forma de *scripts* ou *applets*. Isto diminui as exigências quanto ao que deve estar implementado no cliente, aumentando a extensibilidade, porém diminuindo a visibilidade, por isso é considerada opcional.

Um exemplo da aplicação destes elementos é apresentado na Figura 2.1 (Fielding, 2000). É possível ver representados os conectores clientes, à esquerda das conexões, e servidores, à direita das conexões. Estes conectores cliente e servidor podem ter anexados conectores de *cache*, representados como um "$" na figura. O conceito de camadas também pode ser visto. As informações entre as aplicações clientes (esquerda), e os servidores de dados, arquivos e aplicação (direita), necessitam passar por diversos níveis até chegarem ao seu destino.

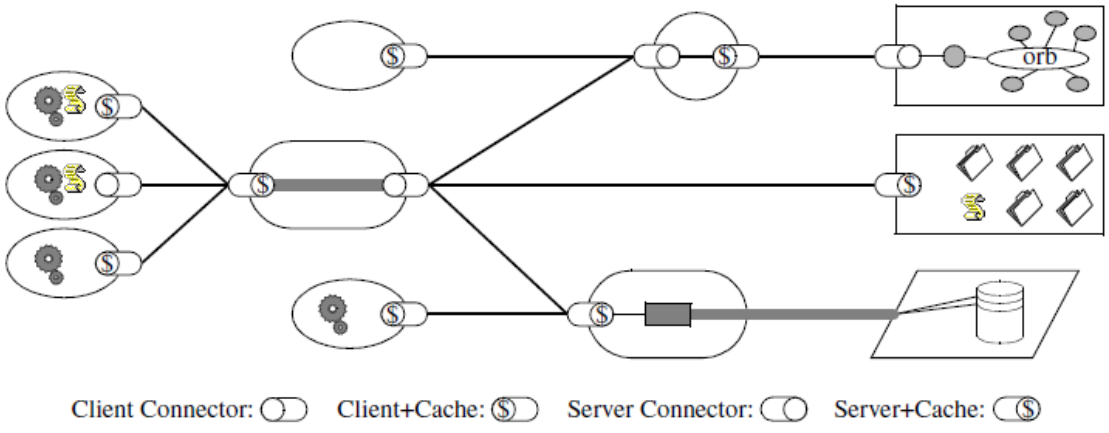



Figura 2.1: O Estilo Arquitetural REST.

### 2.1.1 Elementos Arquiteturais de REST

Os elementos arquiteturais de REST são elementos de dados (ex: recursos e suas representações), conectores (ex: cliente, servidor e *cache*) e componentes (ex: servidor

*Web* e navegador). REST ignora como é feita a implementação de componentes e os detalhes dos protocolos, dando ênfase ao papel dos elementos, suas interações e o significado da representação dos dados. Na Figura 2.2 (Fielding, 2000) pode-se ver com mais detalhes alguns elementos arquiteturais de um exemplo de sistema baseado em REST.

#### *2.1.1.1 Elementos de Dados*

O principal elemento de dados é o recurso, ele é o ponto chave da abstração da informação sendo tratada. Por exemplo, um recurso pode ser uma imagem ou documento, uma informação, uma coleção de outros recursos, etc. O recurso é definido por Fielding (2000) como: uma função de pertinência variável pelo tempo mapeia um conjunto de valores que são equivalentes; cada valor neste conjunto é um recurso ou um identificador de recurso.

Os identificadores de recurso são identificadores únicos que mapeiam para um recurso. Em sistemas Web, os identificadores de recurso são URIs e URLs. Representações são apresentações diferentes de um mesmo recurso, como HTML, XML, JPEG.

#### *2.1.1.2 Conectores*

Os conectores encapsulam as atividades de acessos e transferência de recursos. Eles apresentam uma interface de comunicação, aumentando a simplicidade e promovendo a separação de conceitos.

Os conectores primários são o cliente e o servidor. A principal diferença entre os dois conectores é que o cliente inicia as requisições e o servidor fica a espera de requisições dos clientes. Um terceiro tipo de conector é a *cache*, que fica localizada junto à interface do cliente ou servidor, e armazena o resultado de uma requisição para uso posterior.

Outros tipos de conectores são resolvedores de nomes (DNS) que resolvem parte ou todo um identificador de recursos e túneis que encapsulam uma requisição sobre outra, permitindo a travessia de *firewalls* e uso de NAT.

#### *2.1.1.3 Componentes*

Os componentes são a realização de uma interface de conector (um navegador realiza a interface de cliente, por exemplo). Os componentes mais comuns são: servidores *Web*, navegadores, aplicações, *proxies* e *gateways*.

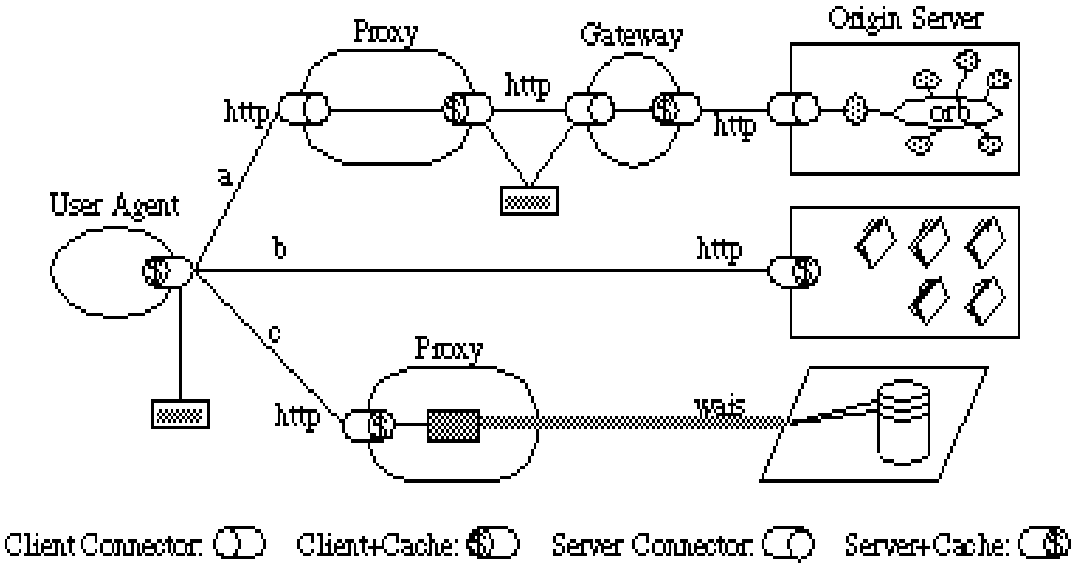



Figura 2.2: Visão arquitetural de um sistema baseado em REST.

### 2.1.2 REST e os padrões da Web

Na Web as páginas HTML, imagens, arquivos, entre outras coisas, são recursos e são identificados sem ambiguidade através de seus identificadores que são as URLs. Os clientes são os diversos *user-agents* e os servidores são os servidores onde estão armazenados os recursos.

Para manipular os recursos, os componentes da rede se comunicam através de uma interface padrão, HTTP, e trocam representações de recursos (arquivos XML, por exemplo, são recebidos e enviados). Segundo Fielding (2000), o protocolo HTTP, juntamente com as URIs criam a interface genérica necessária em uma arquitetura no estilo REST. Mensagens são enviadas em um sistema *RESTful* utilizando HTTP, que define um conjunto de operações bem definidas (POST, GET, PUT e DELETE) e que se aplicam a todos os recursos de informação. Além disso, a definição de HTTP ainda inclui códigos de estado (200 – OK, 404 – *not found*, etc.) e cabeçalhos (*Accept*, *User-Agent*, *Cache-Control*, etc.).

Para transições de estado da aplicação e para a informação da aplicação, é utilizada hipermídia, onde os estados são tipicamente representados por HTML ou XML. Como resultado disto, é possível navegar com um recurso REST a muitos outros, simplesmente seguindo ligações sem requerer o uso de registros ou outra infra-estrutura adicional.

## 2.2 RESTful Web Services

Um *RESTful Web Service* é um serviço *Web* implementado utilizando os princípios do estilo arquitetural REST. É uma coleção de recursos com três aspectos definidos: uma URI básica para o serviço *Web*; um tipo MIME de mídia para internet (MIME *types*) para os dados suportados (exemplos são JSON, XML e YAML); e um conjunto de operações suportadas utilizando métodos HTTP (POST, GET, PUT ou DELETE).

Como tem um ferramental leve e uma infra-estrutura simples, a utilização de *RESTful Web Services* exige mínimos esforço e custo, e poucas são as barreiras para a sua adoção (Pautasso, 2008).

Entre as organizações que disponibilizam APIs para REST ou que utilizam RESTful Web Services estão incluídas: Flickr (Flickr, 2010), Yahoo (Yahoo, 2010), Atom (Atom, 2010), Sun Cloud (Sun, 2010), Twilio (Twilio, 2010), Twitter (Twitter, 2010).

### 2.2.1 Boas Práticas REST

Não há um padrão oficial para *RESTful Web Services*, pois REST é um estilo arquitetural e não um protocolo, porém, como já visto, padrões como HTTP, XML, URI, MIME podem ser utilizados. Também não há um consenso sobre quais seriam as boas práticas aceitas para a construção de *RESTful Web Services* (Pautasso, 2008). Algumas recomendações informais incentivam o uso dos quatro verbos HTTP já citados, o uso de "boas" URIs e o uso de POX (*Plain Old XML*) para a formatação das mensagens. Outras recomendações focam no uso de apenas dois verbos HTTP (GET e POST).

A lista de algumas destas práticas é apresentada a seguir:

- Uso correto dos métodos HTTP: Os métodos HTTP, também chamados de verbos, são utilizados para manipulação dos recursos. São normalmente associados às operações CRUD (*create, retrieve, update e delete*):
    - **GET:** O método GET é associado com a operação *retrive* (recuperar). Deve ter como retorno uma representação do recurso no formato solicitado, sem efeitos colaterais;
    - **POST:** Esse método é associado com a operação *create* (criar). Um POST é normalmente executado sobre uma coleção de recursos para criar um recurso naquela coleção. Os dados fornecidos no *payload* são uma representação do recurso a ser criado;
    - **PUT:** Esse método é associado com a operação de *update* (atualizar). O PUT é executado sobre um recurso para atualizar seus dados. Os dados fornecidos no *payload* são uma representação do recurso com os dados atualizados;
    - **DELETE:** Associado ao *delete* (apagar), este método é executado sobre um recurso para destruí-lo;
- Uso correto dos códigos de *Status*: Uma operação sobre um recurso deveria sempre retornar um código HTTP adequado à semântica do resultado. Por exemplo, um GET sobre um recurso deve retornar 200 (Ok) quando o recurso for acessado corretamente ou 404 (*Not Found*) quando o recurso não existir.
- Uso dos cabeçalhos HTTP: É recomendado o uso de cabeçalhos HTTP para transmissão de metadados relacionados a uma requisição. Por exemplo, se uma requisição esperar uma representação em XML de um recurso como retorno, deve especificar um cabeçalho *Accept:Application/XML* na requisição.

- Semântica do endereço dos recursos: O endereçamento dos recursos deve ser semanticamente adequado. Como exemplo, a URI para acessar os dados de clientes de uma aplicação "XYZ" da organização "ABC" deve ser semelhante a "www.abc.org/xyz/clientes" ou no caso do acesso a um cliente específico com ID "42" deve ser semelhante a "www.abc.org/xyz/clientes/42".
- Manter um padrão homogêneo e bem documentado: O padrão de códigos de status HTTP, cabeçalhos HTTP, identificador dos recursos, etc., deve ser idêntico em toda uma aplicação, em todos os recursos, em todas as situações. Esse padrão deve ser bem documentado e disponível publicamente para todos os interessados no uso da aplicação. Recomenda-se utilizar algum padrão de publicação como WADL (Web *Application Description Language*) para disponibilizar a interface da aplicação publicamente.

Mais detalhes sobre estas práticas aqui apresentadas, e outras boas práticas em REST, podem ser encontrados nos trabalhos de Richardson (2007), He (2004) e Fielding (2000).

# 3 TESTE DE SOFTWARE

A construção de *software* de boa qualidade requer que as atividades de projeto e teste ocorram conjuntamente durante todo o desenvolvimento. Teste de *software* é o processo de executar o mesmo de uma maneira controlada com duas metas distintas: demonstrar ao desenvolvedor e ao cliente que o *software* atende aos requisitos; e descobrir falhas ou defeitos no *software* que apresenta comportamento incorreto, não desejável ou em não conformidade com sua especificação (Sommerville, 2007).

O *software* está entre os artefatos mais complexos e variáveis produzidos. Sua estrutura evolui e geralmente se degrada conforme ele cresce. Além disso, a não-linearidade dos sistemas de *software* e distribuição irregular de falhas dificulta a verificação. Verificações apropriadas do *software* dependem da disciplina de engenharia, do processo de construção, do produto final e dos requisitos de qualidade (Pezzè, 2008).

A atividade de teste durante o desenvolvimento tem por propósito garantir a qualidade e a confiabilidade do *software* e melhorá-lo encontrando defeitos, para que estes sejam removidos. Porém, a dificuldade intrínseca a atividade de teste, apoiada por diversos fatores como o seu custo, a escassez de bons profissionais e a dificuldade de implantação do processo de teste durante todo ciclo de vida, pode tornar o custo/benefício do teste de *software* impraticável quando não houver um bom planejamento desta atividade. O custo da verificação de *software* geralmente excede metade do custo geral do desenvolvimento e manutenção do mesmo (Pezzè, 2008).

A atividade de teste é muito importante para o processo de desenvolvimento, pois ao detectar os erros mais rapidamente diminui-se o custo de manutenção, o tempo gasto com retrabalho e aumenta a confiança no *software* sendo desenvolvido. No contexto do desenvolvimento de *RESTful Web Services*, um dos principais objetivos do teste de *software* é buscar por violações as boas práticas adotadas no desenvolvimento da aplicação.

## 3.1 Planejamento e Projeto de Testes

O processo de teste de *software* é composto por atividades que têm por objetivo executar um programa a fim de revelar suas falhas e avaliar sua qualidade. As principais atividades do processo de teste são: planejamento, projeto dos casos de teste, procedimento de teste, execução dos testes, avaliação dos resultados dos testes (Biasi, 2006).

O principal objetivo do planejamento de testes é definir informações sobre abrangência, abordagem, recursos e atividades de teste. Durante o desenvolvimento do plano de testes é necessário definir uma estratégia que contém:

- **Níveis de Teste:** O nível de teste é dependente da fase do processo de desenvolvimento de *software* em que o teste é aplicado. Os principais níveis são: Teste de Unidade (unidades de implementação), Teste de Integração (integração das unidades), Teste de Sistema (sistema de *software* funcional) e Teste de Aceitação (teste com relação às especificações do(s) usuário(s)).
- **Técnica de Teste:** A técnica de teste direciona a escolha de critérios para projetos de casos de teste. Os métodos de teste compreendem Teste Estrutural (caixa-branca) e Teste Funcional (caixa-preta). Os testes estruturais conhecem o código-fonte do programa e o testam internamente. Os testes funcionais testam o comportamento do sistema baseado nos requisitos, sem conhecimento da estrutura interna. Ao testar serviços *Web* se utiliza a técnica de caixa preta, pois o cliente desconhece a implementação do serviço sendo apenas dependente de sua interface.
- **Critérios:** O critério de teste serve para orientar o testador na geração dos casos de teste. Os critérios de teste são dependentes do método utilizado. No teste caixa-branca podem ser feitos testes de caminhos, de condições, *loops*, etc. e no teste caixa-preta os testes podem ser de limites, partições de equivalência, entre outros.
- **Tipo de Teste:** Os tipos de teste representam as características do *software* a serem testadas. Alguns exemplos são Teste de Funcionalidade, Teste de Interface e Teste de Segurança.

Com base no planejamento e nas especificações do sistema são projetados os casos de teste. Para a atividade de projeto desses casos de teste, é de grande ajuda o uso de uma linguagem de alto nível, clara e expressiva, a exemplo do perfil de teste da UML (U2TP). É conveniente citar que a responsabilidade pelo plano de testes é do papel de Gerente de Teste e a responsabilidade pelas especificações (projeto) dos casos de testes é do papel de Projetista de Testes.

## 3.2 Testes Unitários

O teste unitário tem como objetivo testar o menor artefato funcional de *software* (a unidade). É neste tipo de teste que são descobertos o maior número de defeitos. Ele é executado para melhorar a qualidade geral do *software* que passa pela equipe de teste de integração, sistema e depois para o cliente. O objetivo do processo de teste unitário é testar a funcionalidade da unidade contra a sua especificação, assim tentando encontrar falhas. A unidade pode ser definida como o menor componente funcional de *software* que pode ser testado: uma classe, método ou conjunto de métodos num ambiente orientado a objetos, um módulo ou uma função num ambiente procedural, etc. Num projeto REST a unidade constantemente é associada a um recurso ou pequeno grupo de recursos.

Normalmente os testes unitários são implementados e realizados pela mesma pessoa que implementou a unidade a ser testada: o programador. Devido a isso, um problema bem recorrente é que o projeto dos casos de teste de unidade acabam sendo feitos

também pelo programador, que apesar de ter o conhecimento da funcionalidade implementada e familiaridade com o código não tem a mesma visão e o mesmo conhecimento que um projetista de testes, além de adicionar um viés de imparcialidade, pelo conhecimento interno do *software*. Assim, uma especificação dos casos de teste de unidade pode acabar não sendo documentada ou planejada da maneira mais adequada.

Para preparar um teste unitário, as seguintes atividades podem ser executadas: planejar a abordagem geral para o teste unitário; especificar e projetar os casos de teste; definir os relacionamentos entre os testes; preparar o código auxiliar necessário para o teste de unidade (*Drivers* e *Stubs*) (Biasi, 2006).

O perfil de testes da UML (U2TP) pode ser utilizado como uma notação para especificação de testes unitários. Assim, os artefatos de teste ficam modelados e documentados, proporcionando uma notação padrão que pode ser usada independente do tipo de linguagem de programação na qual o sistema será desenvolvido.

### 3.2.1 Automação de Testes

A automação de testes pode oferecer um ganho ao diminuir o tempo gasto com os testes e com isso diminuir o custo total desta atividade. Esse ganho deve-se a capacidade de uma diminuição de testes manuais e redundantes maximizando a confiabilidade dos testes e diminuindo o custo de repetição.

Das atividades de preparação de um teste unitário, a criação de d*rivers* e *stubs* de teste pode ser automatizada, pois seu desenvolvimento consome recursos, tempo, custo e esforço. Um *driver* de teste pode ter as funções de: fazer chamadas a uma unidade sob teste; passar parâmetros para uma unidade sob teste; mostrar parâmetros; e mostrar resultados (parâmetros de saída). Já um *stub* de teste pode ter as seguintes funções: mostrar uma mensagem que foi chamada por uma unidade sob teste; mostrar parâmetros de entrada passados por uma unidade sob teste; e passar valores para uma unidade sob teste. Considerando um sistema orientado a objetos, d*rivers* e s*tubs* frequentemente oferecem meios de projetar e implementar classes especiais para executar as tarefas de teste requeridas (Biasi, 2006).

### 3.2.2 Frameworks de Teste Unitário

Diversas ferramentas estão disponíveis atualmente para dar suporte ao desenvolvimento de Testes Unitários. Elas têm contribuído para reduzir o tempo e esforço gasto na execução de testes, pois permitem a execução e verificação automatizada dos testes.

Os exemplos mais comuns são os *frameworks* da família XUnit, como JUnit para Java (JUnit, 2010), NUnit para .NET (NUnit, 2010) e Test::Unit para Ruby (Rubydoc, 2010). Estas ferramentas dão suporte a implementação e manutenção dos testes além de prover automação no nível de teste unitário.

## 3.3 Dados de Teste

Dados de teste são a definição (geralmente formal) de um conjunto de valores de entrada de teste que são usados durante a execução de um teste, e os resultados esperados mencionados para fins de comparação durante sua execução. Os dados de teste criam a condição que está sendo testada (como entrada ou como dados preexistentes) e são usados para comparar os resultados reais com os esperados.

A densidade dos dados de teste representa o volume ou a quantidade de dados usados no teste. A densidade é uma consideração importante, pois um volume muito pequeno de dados talvez não reflita as condições reais, ao passo que pode ser difícil gerenciar e manter um volume excessivo de dados. É possível aumentar a densidade dos dados de teste com a simples criação de mais registros. Ainda que normalmente essa seja uma boa solução, ela não considera as variações efetivas que se espera em dados reais (amplitude dos dados). Sem essas variações nos dados de teste, talvez não consigamos identificar defeitos. Portanto, os valores dos dados de teste devem refletir os valores dos dados encontrados no ambiente de implantação.

O escopo é a relevância dos dados de teste para o objetivo do teste, e está relacionado à densidade e à amplitude. Um grande volume de dados não significa que eles sejam apropriados. Da mesma forma que com a amplitude dos dados de teste, devemos garantir a relevância desses dados para o objetivo do teste, ou seja, é necessário haver dados de teste para suportar o objetivo do teste específico.

O teste é repetido dentro de iterações e entre elas. Para executar o teste de forma consistente, confiável e eficaz, os dados de teste devem ser retornados ao seu estado inicial antes da execução do teste. Isso deve ocorrer especialmente quando o teste for automatizado. Portanto, para garantir a integridade, confiança e eficácia do teste, é fundamental que os dados de teste não incluam influências externas, e seu estado deve ser conhecido no início, durante e no fim da execução do teste.

Quando gerenciados separadamente dos aspectos de procedimento do teste, dados de teste permitem a modificação das características exclusivas do teste de modo independente. É comum que vários elementos de dados de teste sejam especificados em um único contêiner de armazenamento e normalmente estejam agrupados pela finalidade ou pelo objetivo geral dos testes.

Enquanto testar todos os valores de entrada possíveis de um programa iria fornecer a visão mais completa sobre seu comportamento, o domínio de entrada é geralmente muito grande para testes exaustivos serem factíveis. Então, o procedimento usual é selecionar um subconjunto relativamente pequeno, que de alguma forma represente todo o domínio de entrada. Uma avaliação do comportamento do sistema com estes dados é então utilizada para prever o seu comportamento geral. Idealmente, os dados de teste seriam escolhidos de maneira que a execução do programa com este conjunto iria descobrir todos os erros, garantindo que qualquer sistema que produzir resultados corretos para os dados de teste irá também produzir resultados corretos para qualquer dado no domínio de entrada.

### 3.3.1 Repositório de Dados

Um repositório de dados é um conjunto dos dados de teste que são utilizados como valores de entrada em um sistema sob teste; é uma coleção de dados relacionados que supre valores de dados realistas para as variáveis em um teste.

Casos de teste são frequentemente executados repetidamente com diferentes valores de dados, para estimular o sistema de várias maneiras. Além disso, classes abstratas de equivalência são utilizadas para definir conjuntos de valores possíveis. Tipicamente estes valores são tirados de partições de dados ou listas de valores explícitos. Para isto, um repositório de dados provê maneiras para associar conjuntos de dados com

contextos de teste e casos de teste, contendo partições de dados (classes de equivalência) ou valores explícitos. (OMG, 2005).

### 3.3.2 Partições de Dados

Uma partição de dados é utilizada para definir uma classe de equivalência para um dado tipo (nomes de usuário válidos, por exemplo). Denotando o particionamento de dados explicitamente, é possível ter uma diferenciação dos dados mais visível (OMG, 2005).

O particionamento de equivalência é uma abordagem sistemática para seleção de dados de teste. Dados de entrada e saída caem em diferentes classes onde todos os membros de uma classe são relacionados. Cada uma destas classes é uma partição de equivalência onde o programa se comporta de uma maneira equivalente para cada membro da classe.

No particionamento de equivalência, o domínio de entrada (ou saída) do sistema é dividido em um número finito de partições (ou classes) de equivalência, onde se supõe que dados pertencentes a uma mesma partição revelem as mesmas falhas; e partições válidas e inválidas são consideradas. Nos testes, um ou mais dados representativos de cada partição devem ser selecionados, e cada partição deve ser considerada pelo menos uma vez.

As partições de equivalência são geralmente derivadas da especificação dos requisitos para atributos de entrada que influenciam o processamento do objeto de teste. Uma entrada tem alguns intervalos que são válidos e outros inválidos. Por exemplo, um parâmetro “mês” possui um intervalo válido nos números 1 a 12, que formam uma partição. Neste exemplo existem ainda outras duas partições para valores inválidos: a primeira seria dos valores menores ou iguais a zero; e a segunda dos valores maiores ou iguais a treze.

### 3.3.3 Seletores de Dados

Para facilitar a criação de estratégias de teste, elementos seletores de dados podem ser empregados. Esses são operações sobre os conjuntos de dados que retornam dados de entrada para os casos de teste, definindo como valores de dados ou classes de equivalência são selecionados de um repositório ou partição de dados.

## 3.4 O Perfil de Teste da UML (U2TP)

O perfil de teste da UML define uma linguagem para projetar, visualizar, especificar, analisar, construir e documentar os artefatos dos sistemas de teste. É uma linguagem de modelagem que pode ser utilizada com as principais tecnologias de componentes e objetos e aplicada para testar sistemas em vários domínios de aplicação (OMG, 2005). O perfil de teste da UML pode ser usado somente para a manipulação dos artefatos de teste ou de uma maneira integrada com UML para a manipulação conjunta de um sistema com seus respectivos artefatos de teste.

Este perfil de teste é baseado na especificação da UML 2.0, e foi criado seguindo dois princípios de projeto: integração com a UML; e reuso (dos conceitos da UML) e minimalismo (acrescentar apenas o necessário). Permite a especificação de testes para aspectos estruturais (estáticos) e comportamentais (dinâmicos) de modelos computacionais UML e é capaz de interoperar com tecnologias de teste existentes para testes caixa-preta.

U2TP estende a UML com conceitos específicos de teste que são agrupados em conceitos de arquitetura de teste, dados de teste, comportamento de teste, e tempo. Por ser um perfil da UML, ele se integra facilmente com esta, pois é baseado no seu metamodelo e reusa sua sintaxe.

A criação do U2TP foi baseada no metamodelo MOF (Meta-Object Facility), o padrão da OMG (Object Management Group) para a construção de metamodelos. É uma especificação que define uma linguagem abstrata para metamodelagem e um *framework* para especificação, construção e gerenciamento de metamodelos independentes de plataforma. Exemplos de sistemas que usam o MOF incluem ferramentas de modelagem e desenvolvimento, sistemas *data warehouse*, repositórios de metadados, entre outros (OMG, 2006).

U2TP pode ser usado isoladamente para a manipulação de artefatos de teste ou de uma maneira integrada com o resto da aplicação, manipulando simultaneamente artefatos de sistema e de teste. Havendo um modelo de projeto UML, é possível especificar testes para um sistema estendendo este modelo com os conceitos U2TP (Dai, 2004). Primeiramente, criando um novo pacote UML de teste, importando os elementos necessários do modelo de projeto do sistema, e após, especificando os elementos relativos aos conceitos da U2TP.

A arquitetura dirigida por modelos (MDA – *Model Driven Architecture*) da OMG busca padronizar o uso de linguagens de descrição, como UML. O desenvolvimento da versão 2 de UML, alinhado com a estratégia de MDA, teve como um de seus objetivos possibilitar a "execução" de UML, permitindo não só a geração de código, simulação e validação dos modelos, mas também a geração de testes. Esse objetivo permitiria uma melhor implementação de um processo de teste baseado em modelos.

### 3.4.1 Arquitetura de Teste

Arquitetura de teste é um conjunto de conceitos para especificar os aspectos estruturais de um contexto de teste cobrindo componentes de teste, o sistema sob teste, sua configuração, etc. A Figura 3.1 (OMG, 2005) descreve os seus principais elementos. São eles:

- **TestContext:** Um *TestContext* (contexto de teste) é uma classe que representa o agrupamento de vários casos de teste, ou seja, representa o conceito conhecido como suíte de teste. A notação para o elemento *TestContext* é uma classe com o estereótipo <<TestContext>>.
- **TestComponent:** Um *TestComponent* (componente de teste) é uma classe de um sistema em teste. *TestComponents* interagem com o SUT ou com outros *TestComponents* para realizar os casos de testes que são definidos dentro do *TestContext*. A notação para o elemento *TestComponent* é uma classe com o estereotipo <<TestComponent>>.

- **SUT:** O SUT (*System Under Test*, ou sistema sob teste) é o sistema, subsistema ou componente sendo testado. O SUT é exercitado através de suas operações públicas pelos *TestComponents*. Nenhuma informação a mais pode ser obtida do SUT, pois este é uma caixa-preta. A notação para o SUT é nomear o que se deseja testar com o estereotipo <<Sut>>.
- **Arbiter:** Uma propriedade de um caso de teste ou um *TestContext* para avaliar os resultados do teste e designar o *Veredict* de um caso de teste ou *TestContext*, respectivamente. Há um algoritmo padrão de arbitragem baseado em testes funcionais e de conformidade, que gera "*Pass*", "*Fail*", "*Inconc*" e "*Error*" como *Veredict*. Este algoritmo pode ser definido pelo usuário.
- **Scheduler:** Uma propriedade de um *TestContext* utilizado para controlar a execução de diferentes *TestComponents*. O *Scheduler* vai ter a informação sobre quais *TestComponents* existem em qualquer ponto no tempo, e vai colaborar com o *Arbiter* para informá-lo quando o *Veredict* final deve ser dado. Controla a criação e destruição de *TestComponents* e sabe quais fazem parte de cada caso de teste.

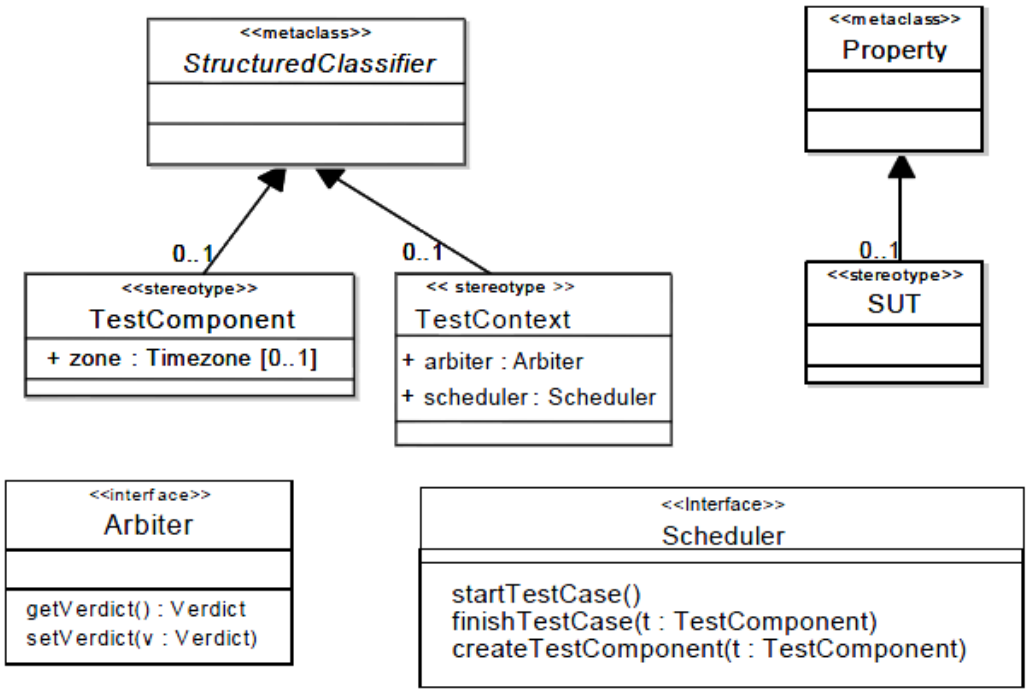



Figura 3.1: Arquitetura de Teste.

### 3.4.2 Comportamento de Teste

Comportamento de Teste é o conjunto de conceitos que especificam comportamentos de teste, seus objetivos e a avaliação de sistemas sob teste. Este grupo define os conceitos necessários para representar todos os elementos que fazem parte dos

aspectos dinâmicos dos procedimentos de teste. A Figura 3.2 (OMG, 2005) ilustra os principais elementos. São eles:

- **TestControl:** Um *TestControl* é uma especificação para a invocação de casos de teste em um *TestContext*. É uma especificação técnica de como o SUT deve ser testado no dado *TestContext*. Permite especificar a ordem de execução dos casos de teste.

- **TestCase:** Um *TestCase* é uma especificação de um caso para testar o sistema incluindo o quê testar, qual a entrada, o resultado e sob quais condições. Ele implementa um objetivo de teste. O *TestCase* utiliza um *Arbiter* para avaliar o resultado do seu comportamento de teste. É uma operação que especifica como um conjunto de *TestComponents* cooperativos interagindo com um SUT realizam um objetivo de teste. Um *TestCase* pertence a um *TestContext.* O tipo de retorno de um caso de teste deve ser um *Verdict*. Se o estereótipo de um caso de teste for aplicado em uma operação, a classe desta operação tem que ter estereótipo *TestContext* aplicado. Mas, se o estereótipo de um caso de teste for aplicado a um comportamento, o comportamento desse caso de teste tem que ter o estereótipo *TestContext* aplicado. O estereótipo não pode ser aplicado a ambos. A notação para um caso de teste é uma operação com o estereótipo <<TestCase>>.

- **TestObjective:** Um *TestObjective* (objetivo de teste) é um elemento que descreve o que deve ser testado, e está associado a um *TestCase*.

- ***Default*:** É um comportamento disparado por uma observação de teste que não é tratado pelo comportamento do caso de uso em si. *Defaults* são executados por *TestComponents*.

- **Veredict:** *Veredict* é a avaliação da corretude do SUT, produzida pelos casos de uso. *Veredicts* podem também ser utilizados para reportar falhas no sistema de teste. *Verdict* é um tipo de dados *enumeration* pré-definido que contém os valores "*pass*" (o sistema está correto para este caso de uso), "*fail*" (o propósito do teste foi violado), "*inconclusive"* (resultado inconclusivo) e "*error"* (erro no teste ou na sua execução). *Veredicts* podem ser definidos pelo usuário e são calculados pelo *Arbiter*.

- **ValidationAction:** Uma ação para avaliar o estado da execução de um *TestCase* através da avaliação das observações do SUT e/ou características ou parâmetros adicionais do SUT. É realizada por um *TestComponent* e define o *Veredict* local para aquele *TestComponent*.

- **LogAction:** Uma ação para registrar uma informação no *TestLog*.

- **TestLog:** Um *log* é uma interação resultante da execução de um caso de teste. Ele representa as mensagens trocadas entre os *TestComponents* e o SUT e/ou os estados dos *TestComponents* envolvidos. É associado com um *Veredict* representando a aderência de um SUT ao objetivo de teste do caso de teste associado.

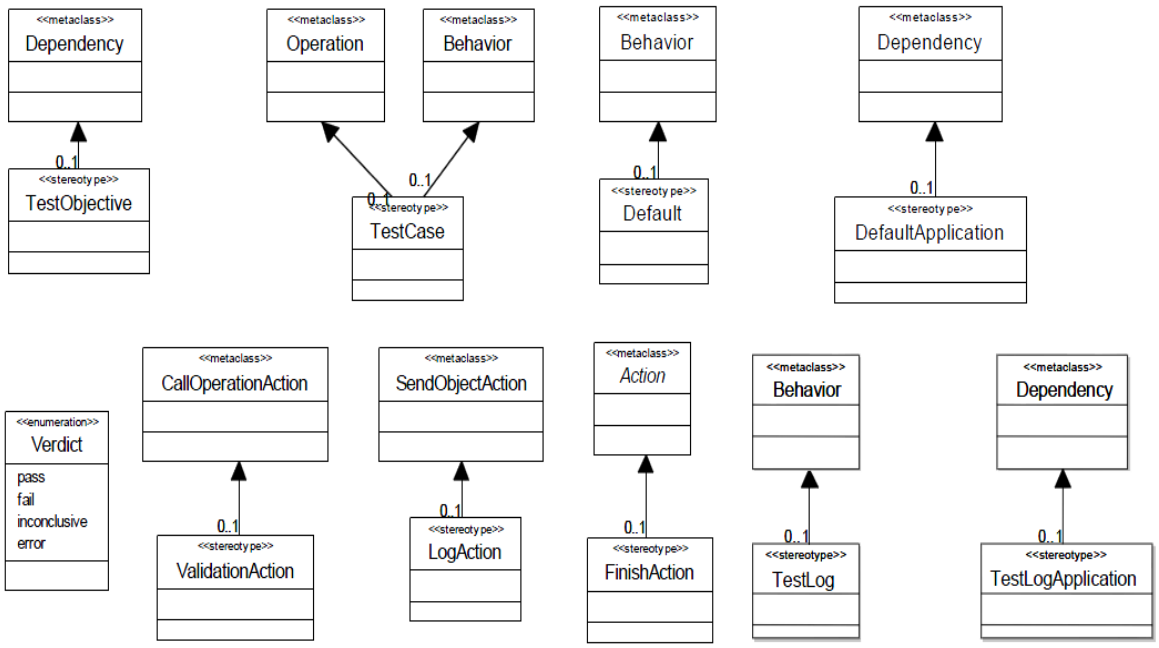



Figura 3.2: Comportamento de Teste.

### 3.4.3 Dados de Teste

O grupo de dados de teste define a sintaxe e semântica dos dados usados como entrada e saída dos procedimentos de teste. Dados explícitos e classes de equivalência formam conjuntos de dados; essas classes de equivalência especificam regras para formação de dados de entrada ou saída. Valores coringas (*wildcards*), em adição aos já presentes em UML padrão, são oferecidos. Os elementos seletores de dados (*data selectors*) são usados para facilitar a criação de estratégias de teste. A Figura 3.3 (OMG, 2005) ilustra os elementos principais do grupo de dados de teste. São eles:

- ***Wildcard*:** Permite que o usuário especifique explicitamente se o valor está presente ou não, e/ou se é de algum valor. *Wildcards* são símbolos especiais que representam valores ou intervalos de valores. Seus valores possíveis são: "*Any*" (qualquer valor), ou "*AnyOrNull*" (qualquer ou nenhum valor - omitido).

- **DataPool:** É uma coleção de *DataPartitions* ou valores explícitos que são utilizados em um *TestContext*, ou *TestComponent*, durante a avaliação dos contextos e casos de teste. Um *DataPool* contém um ou vários *DataPartition*, mas só pode estar associado a um *TestContext* ou um *TestComponent*. A notação para um *DataPool* é uma classe estereotipada com <<DataPool>>.

- **DataPartition:** Um valor lógico para um parâmetro utilizado em um estímulo ou em uma observação. Geralmente define uma classe de equivalência para um conjunto de valores ou tipo de dados. A notação para o elemento *DataPartition* é uma classe com o estereótipo <<DataPartition>> aplicado, e deve estar associado somente a um *DataPool* ou outro *DataPartition*.

- **DataSelector:** Uma operação que define como valores de dados ou classes de equivalência são selecionados de um *DataPool* ou *DataPartition*. A notação para o elemento *DataSelector* é uma operação estereotipada com <<DataSelector>>.
- **CodingRule:** As interfaces de um SUT utilizam certas codificações (CORBA, IDL ou XML, por exemplo) que têm que ser respeitadas pelos sistemas de teste. Então, *CodingRules* são parte de uma especificação de teste.

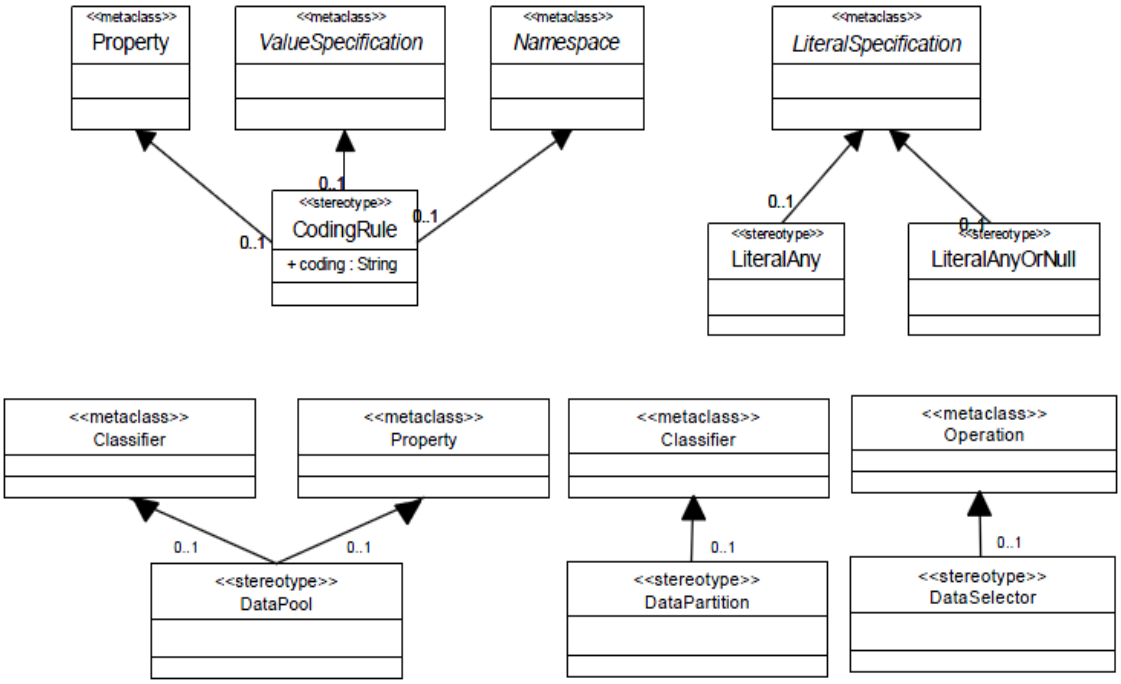



Figura 3.3: Dados de Teste.

### 3.4.4 Conceitos de Tempo

Conceitos de tempo são um conjunto que especifica restrições de tempo, observações de tempo e/ou *timers* nas especificações de comportamento de teste de maneira a ter uma execução de teste com tempo quantificado e/ou a observação da execução temporizada de casos de testes. A Figura 3.4 (OMG, 2005) ilustra estes conceitos. São eles:

- **Timer**: *Timers* são mecanismos que podem gerar um evento de *timeout* quando um valor de tempo especificado ocorrer. Pode ser quando um intervalo de tempo pré-especificado expira relativo a um instante dado. *Timers* pertencem a componentes de teste e são definidos como propriedades destes. Um *timer* pode ser parado. O tempo para expiração e o estado de um *timer* podem ser verificados.

- **Timezone**: É um mecanismo de agrupamento para *TestComponents*. Cada *TestComponent* pertence a no máximo um *Timezone*. *TestComponents* que pertencem ao mesmo *Timezone* têm o mesmo tempo, ou seja, estão sincronizados no tempo.

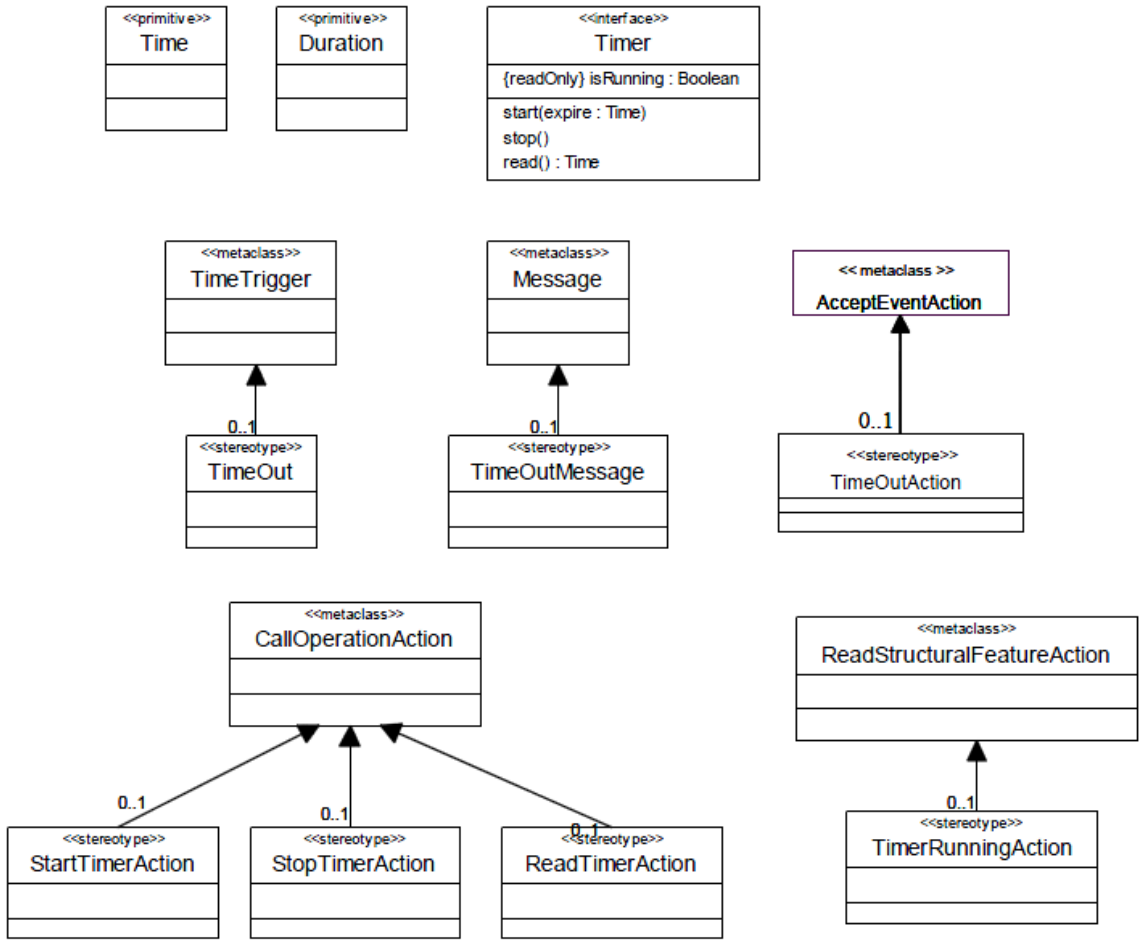



Figura 3.4: Conceitos de Tempo.

### 3.4.5 Metamodelo MOF para Teste

Esta sessão apresenta um metamodelo padrão para o Perfil de teste da UML 2.0, o qual é uma instância do MOF. Este metamodelo é limitado aos elementos da arquitetura e comportamento de teste e é apresentado na Figura 3.5 (OMG, 2005). A maioria dos elementos do Perfil de Teste apresentados anteriormente (Sessões 3.4.1 a 3.4.4) também estão presentes no metamodelo.

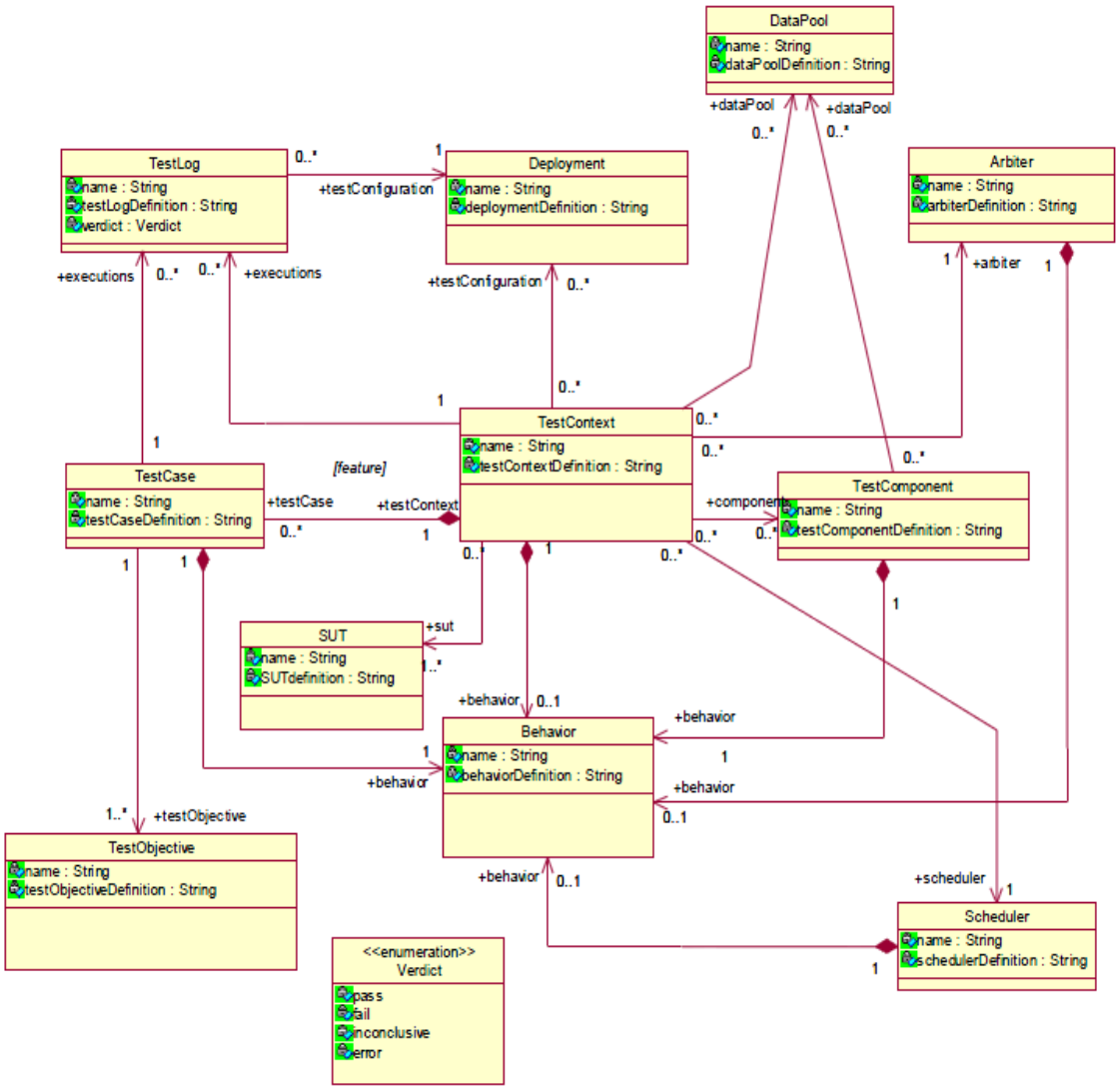



Figura 3.5: Arquitetura e Comportamento de Teste do Metamodelo baseado no MOF.

O elemento central dessa figura é o elemento *TestContext*, que tem associação com todos os demais elementos. Ele representa o passo inicial para especificação da parte de arquitetura de teste, e a partir dele, os demais elementos são instanciados e utilizados.

A Figura 3.6 (OMG, 2005) mostra os elementos principais do grupo de dados de teste. O elemento central desse grupo é o *DataPool*. O elemento *DataPartition* é usado obrigatoriamente sempre que o elemento *DataPool* for usado, pela associação de agregação entre esses dois elementos. O elemento *DataSelector*, deve estar associado a no máximo um *DataPool* e um *DataPartition*.

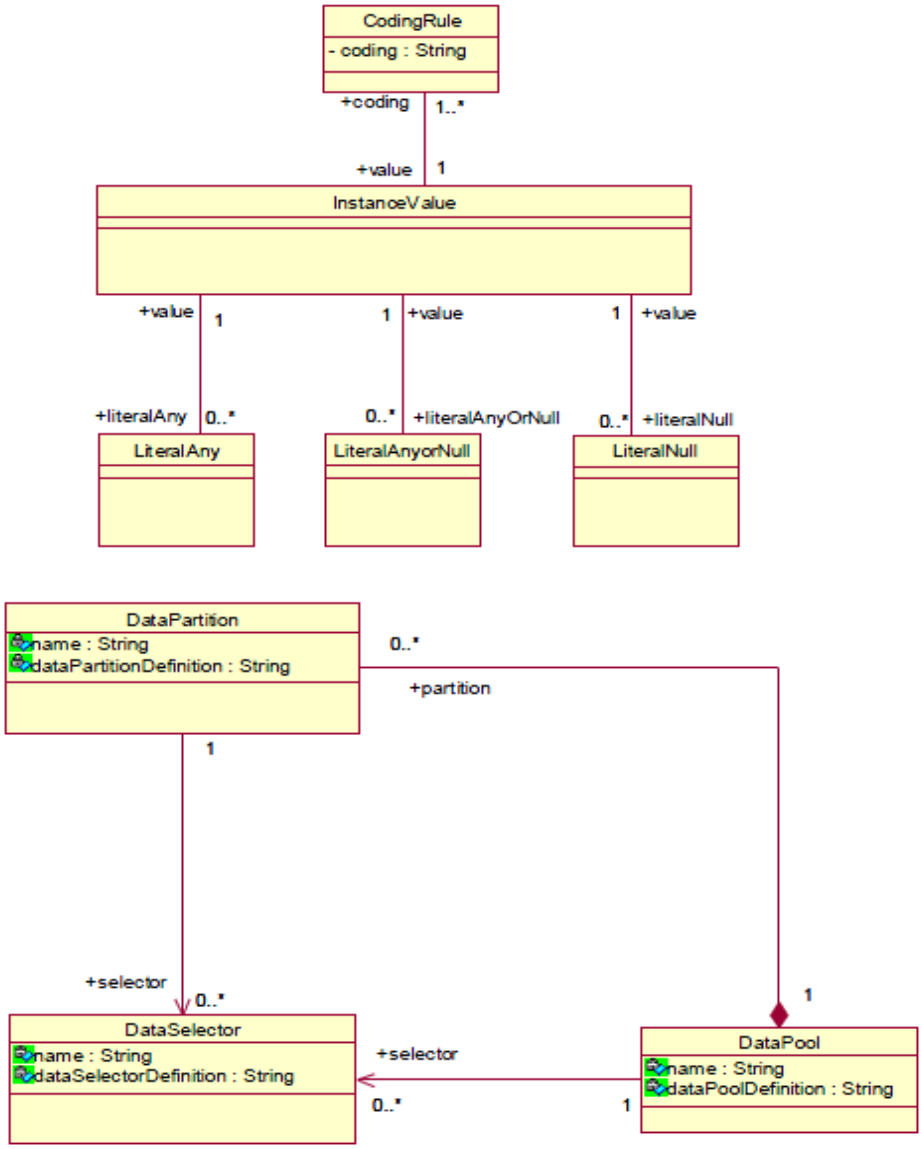



Figura 3.6: Grupo Dados de teste do Metamodelo baseado no MOF.

### 3.4.6 U2TP e o Teste Unitário

Os elementos apresentados nas seções anteriores são usados em todos os níveis de teste. Em nível de unidade, os elementos apropriados são apresentados na Figura 3.7 (Biasi, 2006).

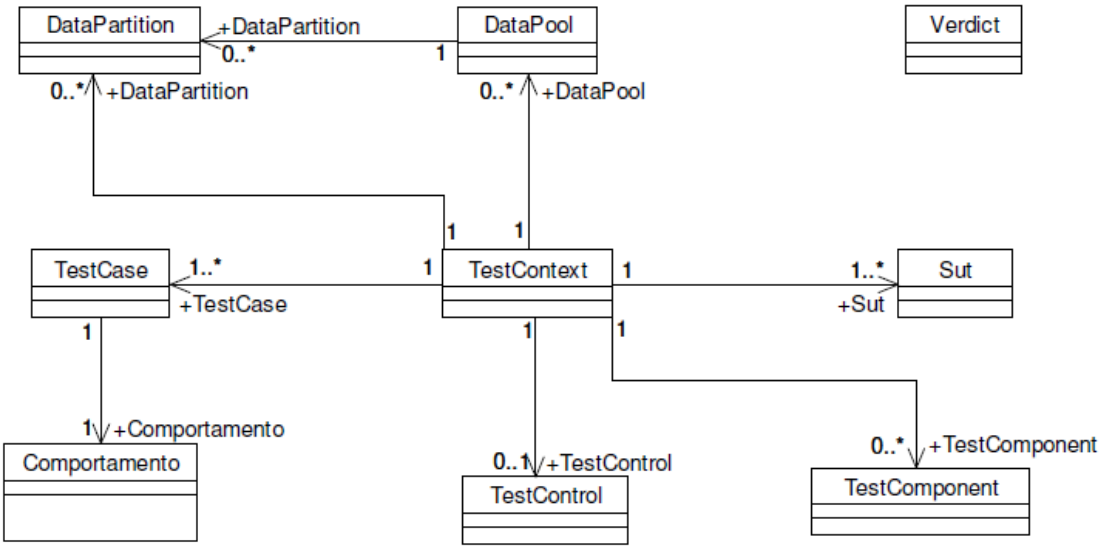



Figura 3.7: Elementos do U2TP para teste em nível de unidade.

A Figura 3.7 ilustra o metamodelo para representação de teste em nível de unidade para os elementos pertencentes aos grupos de Arquitetura de Teste (*Sut*, *TestContext*, *TestComponent*, *TestCase*, *TestControl*), Comportamento de Teste (Comportamento e *Verdict*) e Dados de teste (*DataPool* e *DataPartition*). Como podemos observar pela figura, a cardinalidade de algumas associações mudaram com relação ao metamodelo MOF, como é o exemplo da associação de *TestContext* com *Sut*, onde agora, um *TestContext* deve obrigatoriamente ter um ou mais SUT associados. Assim, como este metamodelo é focado somente para teste de unidade, algumas restrições foram acrescentadas as associações entre os elementos.

- SUT: Esse elemento é especificado para representar a unidade sob teste. Sempre haverá no mínimo um SUT.
- TestContext: Esse elemento é especificado para representar uma suíte de testes para a unidade a ser testada. Assim, um *TestContext* tem que conter no mínimo um SUT e um *TestCase*.
- TestCase: Esse elemento é especificado para representar os casos de teste em nível de unidade. Sempre haverá no mínimo um caso de teste para testar o SUT.
- TestComponent: Esse elemento é especificado como sinônimo de um "*stub*" em teste unitário, ou seja, é usado para simular serviços explicitamente requisitados pelo SUT. Pode ser usado sempre que um ou mais casos de teste necessitar de serviços que não estejam disponíveis no projeto de teste.
- TestControl: Esse elemento deve ser especificado para indicar a ordem de execução dos casos de teste sempre que for necessário.
- Comportamento: Esse elemento é especificado para representar o comportamento de um caso de teste. Precisa ser especificado para cada caso de teste.

- Verdict: Esse elemento é especificado para definir o veredito dos casos de teste.
- DataPool: Esse elemento pode ser usado para representar um conjunto de valores concretos usados por um ou mais casos de teste para testar um determinado SUT.
- DataPartition: Esse elemento pode ser usado para particionar os valores de *DataPool* quando for necessário.

# 4 TRABALHOS RELACIONADOS

Esta sessão resume os trabalhos relacionados a RESTUnit+, que compreendem: REST-Unit, trabalhos sobre teste de RESTful Web Services e trabalhos sobre geração de dados de teste. Esta abordagem foi escolhida, pois, além do trabalho de Borges (2009), não foi encontrado nenhum trabalho semelhante na geração de casos de teste para Web Services RESTful, utilizando U2TP.

## 4.1 Teste de Web Services RESTful

Diversas ferramentas para auxílio ao teste de Web Services RESTful podem ser encontradas na literatura e na Web. Elas podem ser de teste manual ou automatizado, e estão descritas a seguir.

Nenhuma das abordagens 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 dos dados de teste deve ser escrita pelo programador. Na abordagem de teste manual, podem ser utilizadas ferramentas para auxiliar na criação dos dados necessários aos testes.

### 4.1.1 Teste Manual

Com essa abordagem é adicionado um sobrecusto de execução dos testes, pois é necessária a execução manual dos casos de teste. Dessa forma a execução é a fase dos testes que tem maior custo e demanda maior tempo. Além disso, é necessária a análise humana dos dados de entrada e dos resultados para verificar o sucesso do teste, o que adiciona mais um ponto sujeito a erros.

Esta abordagem só se mostra vantajosa em situações onde o sistema implementado é pequeno e tem-se a disponibilidade integral de um testador, para rápida resposta às alterações no sistema. Entretanto teste manual é ainda muito utilizado em diversos projetos para teste de *Web services* durante o desenvolvimento, principalmente nos pequenos projetos.

Neste método, todos os parâmetros do teste devem ser fornecidos pelo testador a cada teste executado e a saída deve ser conferida pelo testador para verificar se está de acordo com o esperado.

**RESTClient:**

RESTClient (RESTClient, 2010) é uma aplicação Java para testar Web Services RESTful. Pode ser utilizada para testar diferentes comunicações HTTP. A interface da aplicação é uma GUI Swing como apresentado na Figura 4.1. O teste é feito acessando um recurso através de uma URL inserida num campo texto. As opções do acesso ao *Web service* estão disponíveis em abas. Todos os métodos HTTP estão disponíveis em uma aba. Ao efetuar o teste, a resposta é exibida logo após o acesso em uma aba abaixo, apresentando o código de *status* e abas para os cabeçalhos, dados retornados pela resposta e uma aba para resultado de teste no caso de uso de *scripts*, o que não é utilizado nesta abordagem.

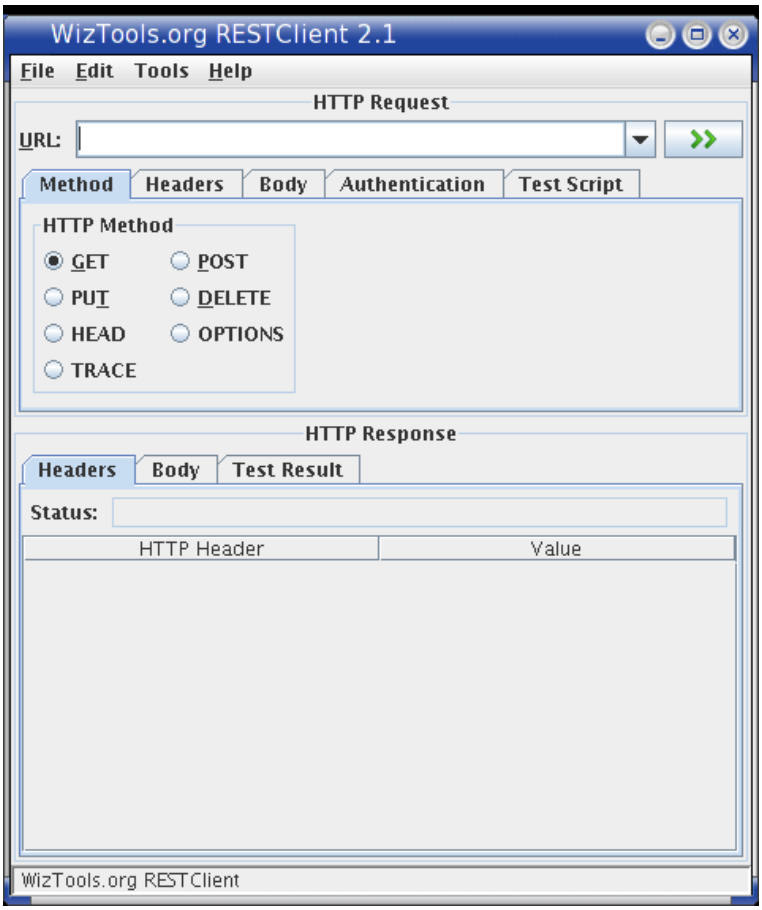



Figura 4.1: RESTClient.

**cURL:**

cURL (Curl, 2010) é uma ferramenta de linha de comando para transferência de dados com sintaxe URL, suportando diversos protocolos, entre eles o HTTP. cURL é uma ferramenta de código fonte aberto e tem versões com base em diversos sistemas operacionais.

Para efetuar um teste é feita uma requisição HTTP ao servidor, no identificador (URL) do recurso que se deseja testar passando como parâmetro outras opções como o

método HTTP utilizado e os dados a serem enviados, caso existam. Um exemplo do uso do cURL é apresentado na Figura 4.2 (Borges, 2009).

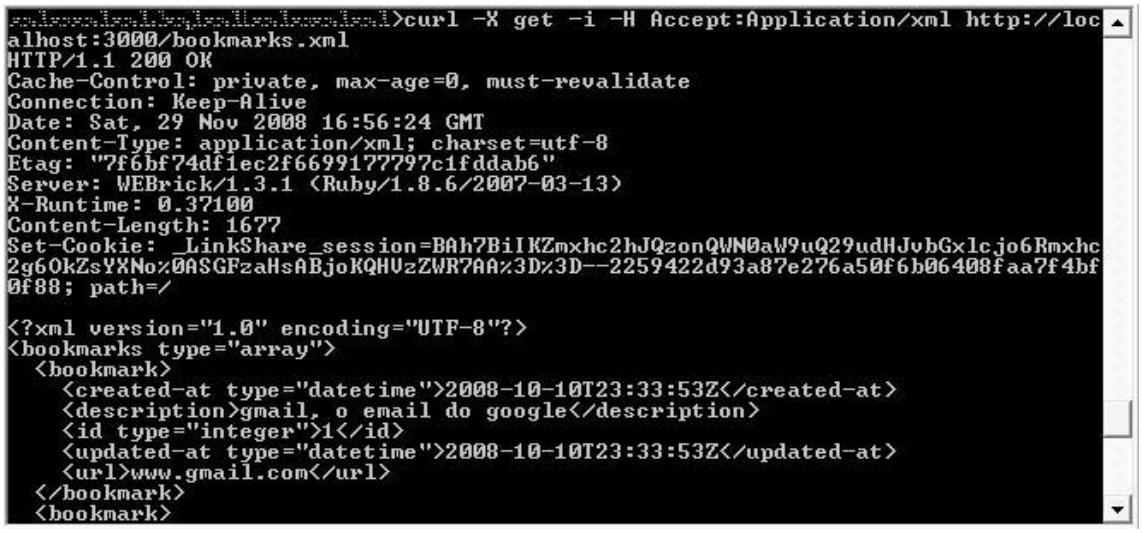



Figura 4.2: Acesso a RESTful *Web service* com cURL em terminal Windows.

**NetBeans:**

A IDE NetBeans (Sun, 2010) oferece suporte ao desenvolvimento e teste de *Web services* RESTful. O teste pode ser iniciado a partir de menus e é gerada uma página *Web* para execução dos testes no navegador. A partir dessa página pode-se testar o serviço definindo o recurso que será testado (ex: "/customers"), o método que será utilizado (GET, POST, PUT, DELETE), o MIME *type* esperado como formato de resposta. O retorno da invocação do serviço é exibido na mesma tela, tanto o código HTTP retornado quando o *payload*. Além disso, há uma aba no mesmo grupo da exibição do resultado que exibe os cabeçalhos HTTP que acompanham o retorno. A Figura 4.3 (Borges, 2009) apresenta uma página gerada pelo NetBeans para execução dos testes em um navegador.

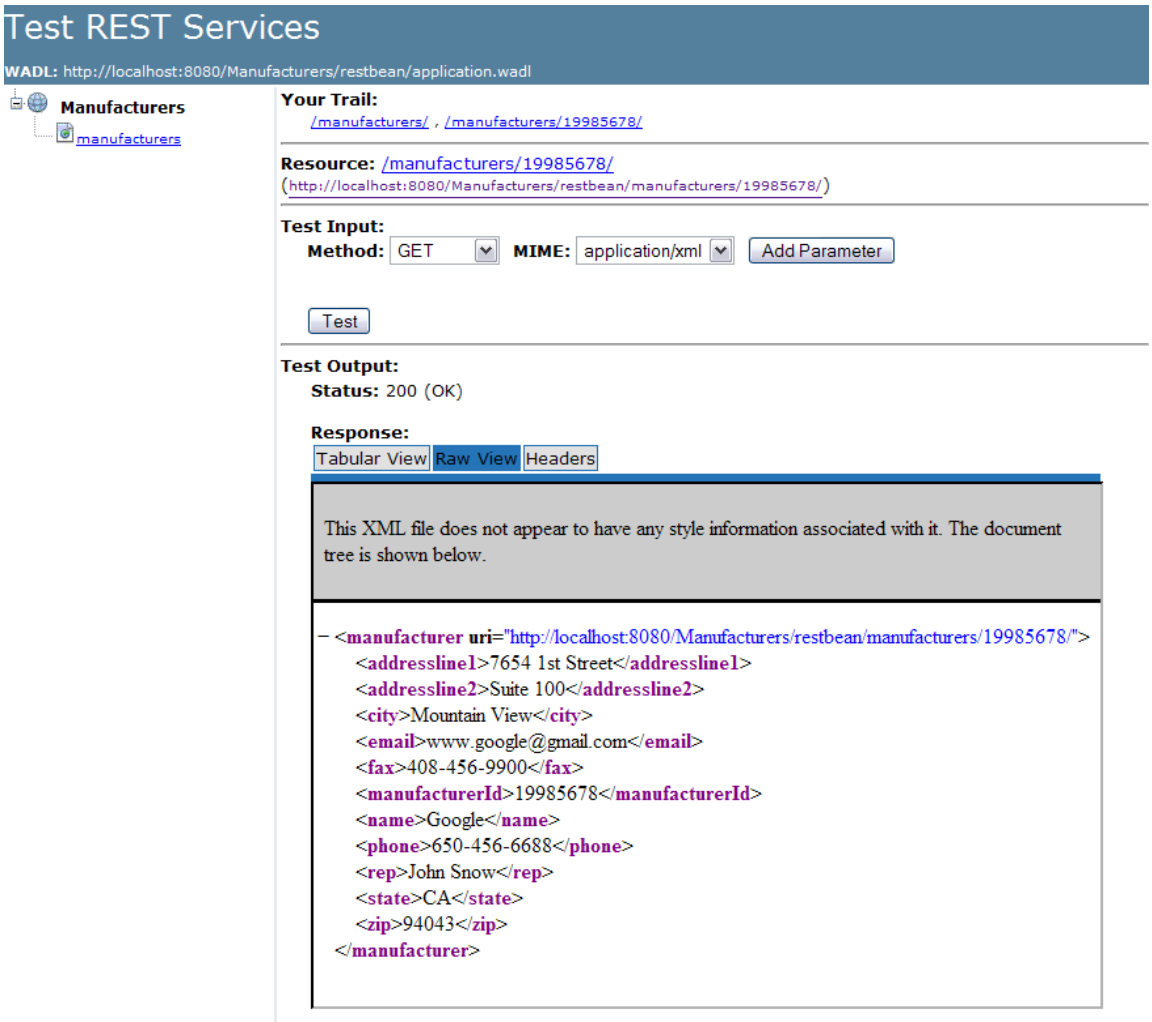



Figura 4.3: Página para teste de RESTful *Web service* com NetBeans.

**Eclipse HTTP Client:**

Eclipse HTTP Client (HTTP4e) (YWebb, 2010) é um *plugin* para o ambiente Eclipse para realizar chamadas HTTP e RESTful. Ele simplifica o trabalho para testar *Web services*, REST, JSON e HTTP. Possui editores visuais para cabeçalhos, parâmetros e corpo HTTP, e pode gerar códigos para diversas linguagens (entre elas: Java, PHP, C#, Python, Ruby). A Figura 4.4 (YWebb, 2010) mostra a GUI do Eclipse HTTP Client para uma chamada REST.

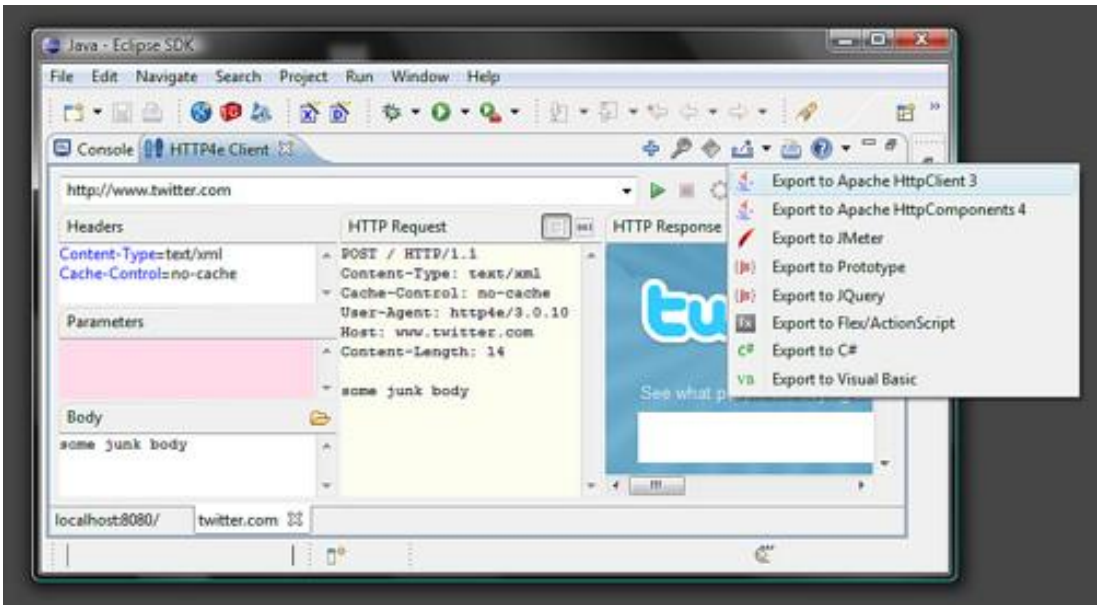



Figura 4.4: Eclipse HTTP Client.

### 4.1.2 Teste Automatizado

Na abordagem de teste automatizado, a execução dos casos de testes é feita automaticamente por um programa ou *script*. Por este motivo, ela tem características que a tornam mais adequada a integração com o ciclo de desenvolvimento dos *Web services*, como a rápida execução dos casos de teste, a facilidade de repetição dos testes e verificação automática dos resultados de testes. Por essa abordagem permitir a execução de testes em lote, é a opção no caso de projetos de maior porte.

Por outro lado, essa abordagem apresenta um *overhead* associado à implementação dos casos de teste. Para permitir a execução automática dos testes, o comportamento destes deve ser definido em algum formato padronizado que possa ser interpretado em um *software*, isto é, os testes precisam ser programados de alguma forma. Muitas vezes esse *overhead* é muito elevado e pode não compensar os seus benefícios, o que nos leva a buscar soluções para o aumento de produtividade nessa atividade dos testes para que possamos utilizar esses benefícios num custo aceitável. Além disso, quem implementa o comportamento destes testes é normalmente um programador, e este pode não ter a bagagem de conhecimento adequada para escrever casos de teste eficientes.

Ferramentas que suportam esta abordagem foram analisadas no desenvolvimento deste trabalho, estas incluem Test::Unit, o *framework* de teste unitário padrão em Ruby (Rubydoc, 2010), e outros *frameworks* de teste unitário da família XUnit, como NUnit e JUnit. Porém, nenhuma ferramenta que suporte nativamente testes unitários de *Web services* RESTful foi encontrada. O suporte para acesso ao *Web service* nos testes unitários é adicionado pelo uso de alguma biblioteca específica para isto, como por exemplo a biblioteca Net::HTTP de Ruby.

**Test::Unit:**

Test::Unit é o *framework* de teste unitário em Ruby, auxilia no projeto, depuração e avaliação do código de aplicações por permitir facilmente escrever e manter testes para elas (Rubydoc, 2010).

Este *framework* é um porte da família de bibliotecas de teste unitário XUnit, e por isto tem uma semelhança muito grande com JUnit e NUnit. Os testes são especificados em linguagem Ruby sobre este *framework*. A classe que contém os métodos de teste estende a classe Test::Unit::TestCase, e os testes são os métodos desta classe cujo nome inicia com "`test`". O Test::Unit não tem suporte direto a requisições HTTP, porém para isto é utilizado o *framework* padrão de acesso ao protocolo HTTP do Ruby, Net::HTTP. Um exemplo de teste com Test::Unit é mostrado na Figura 4.5.

```
1  require 'test/unit'
2  require 'net/http'
3
4  class TestWebService < Test::Unit::TestCase
5
6      def test_get_bookmarks
7          req = Net::HTTP.start("localhost", 3000) { |http|
8              http.get('/bookmarks.xml')
9          }
10         assert_equal 200, req.code.to_i
11     end
12 end
```

Figura 4.5: Exemplo de teste de RESTful *Web service* com Test::Unit.

A execução dos testes é completamente automática, bastando apenas ser iniciada pelo testador. A execução do teste pode ser iniciada em uma interface gráfica ou por linha de comando executando a classe que estende Test::Unit::TestCase. Em todas as interfaces, o resultado dos testes é avaliado automaticamente e apresentado ao fim da execução, contendo um pequeno resumo estatístico dos resultados dos casos de teste e a indicação dos casos de teste que não executaram corretamente. Um exemplo de execução pode ser visto na Figura 4.6.


Figura 4.6: Execução de um teste Test::Unit em linha de comando.

## 4.2 Geração de Dados de Teste

É possível encontrar na literatura muitas maneiras diferentes de geração de dados para teste de *software*, desde geração de dados randômicos para bancos de dados, geração de dados para testes estruturais utilizando metaheurísticas, até geração de dados a partir de diagramas UML. Nesta sessão serão apresentados trabalhos que se relacionam com este, gerando dados de teste a partir de diagramas UML, U2TP, ou para testes de *Web services*.

É proposto por Nayak et al. (2010) uma abordagem para sintetizar dados de teste a partir de informações contidas em elementos de modelos como diagramas de classe, sequência e restrições OCL (*Object Constraint Language*). O diagrama de sequência é enriquecido com informações de atributos e restrições derivados do diagrama de classe e restrições OCL e mapeia isto para um grafo estruturado composto. As especificações de teste são geradas então a partir deste grafo. Para cada especificação de teste, um sistema de resolução de restrições auxilia na geração de dados de teste.

Outra abordagem bastante utilizada em telecomunicações e comunicação de dados é modelar os testes utilizando U2TP e gerar a versão executável com mapeamentos baseados no TTCN-3 (*Testing and Test Control Notation Version 3.0*) (ETSI, 2010). TTCN-3 é uma especificação de teste e linguagem de implementação que pode ser utilizada para definir procedimentos para testes caixa-preta de sistemas distribuídos. Os constituintes de uma especificação de teste TTCN-3 são derivados das especificações nativas de U2TP, como definições de tipo e dados de teste e a definição do comportamento de teste e casos de teste. Um componente nativo é equipado com versões executáveis específicas dos testes TTCN-3 juntamente com um ambiente para execução dos testes. Um exemplo é o trabalho de Zander et al. (2005), que gera testes executáveis automaticamente a partir de U2TP, e um mapeamento para TTCN-3 é proposto, utilizando regras de transformação. A saída gerada é um código teste em Java completo e compilado. Os conceitos U2TP para dados de teste (*DataPool*, *DataPartition* e *DataSelector*) também são mapeados para os respectivos elementos de modelo do TTCN-3.

Bai et al. (2008) propõe uma abordagem para testes de *Web services* baseado em ontologias. Um modelo de ontologia de teste (compatível com U2TP) é definido para especificar conceitos, relacionamentos e semântica. Baseado na linguagem de ontologias OWL, discute técnicas para geração de subdomínios para partição dos dados de teste de entrada. Repositórios de dados são estabelecidos para cada parâmetro do serviço. Partições de dados são derivadas por propriedades de classe e análise de relacionamentos.

## 4.3 REST-Unit

REST-Unit é uma solução para geração automática de *drivers* de teste. Estes *drivers* têm como objetivo validar o comportamento de *RESTful Web services*, através de modelos especificados em U2TP. Esta solução apresenta vantagens comparada com as abordagens tradicionais de teste de *RESTful Web services*, além de contribuir para reforçar o padrão U2TP e fortalecer as boas práticas para o estilo REST. REST-Unit foi criado para preencher algumas lacunas da geração automática de *drivers* de teste para

*Web services*. O código é gerado com o objetivo de ser executado sobre Test::Unit, o *framework* padrão de testes unitários em linguagem Ruby.

Três passos definem a idéia geral da geração de *drivers* de teste: especificação (em diagramas que modelam os testes em U2TP), exportação (para o formato XMI) e geração (de código que guia a execução dos testes). A especificação é feita em U2TP, utilizando a modelagem proposta por Biasi (2006). Após a especificação do teste através dos diagramas UML de classe e sequência, o modelo é exportado para o formato XMI. A geração automática do código dos *drivers* de teste é feita através de um algoritmo que implementa o mapeamento U2TP para código, utilizando para isso o arquivo XMI gerado anteriormente. Neste algoritmo, diversas classes dividem a responsabilidade de geração de código, que buscam o elemento de interesse e seus atributos no arquivo XMI. Cada elemento é identificado por seu estereótipo e outras características específicas.

Porém, esta solução não trata explicitamente os dados de teste, como especificado no U2TP, utilizando conceitos de *DataPool* e *DataPartition*. Ela simula os dados utilizando componentes de teste. A solução REST-Unit+ apresentada neste trabalho visa preencher esta lacuna.

# 5 REST-UNIT+: GERAÇÃO DE DADOS DE TESTE A PARTIR DE U2TP

Os Web Services RESTful vêm sendo muito bem aceitos, principalmente pela simplicidade e clareza da interface publicada. Porém, como qualquer outra aplicação, devem também ser testados o mais cedo possível dentro do ciclo de seu desenvolvimento. Uma técnica largamente utilizada para isto é automação de testes, pois reduz o tempo de execução dos mesmos e possibilita a verificação automática dos resultados, permitindo que sejam facilmente repetidos diversas vezes durante o desenvolvimento. Algumas vezes, porém, o custo de desenvolvimento de *drivers* de testes pode não compensar as vantagens adquiridas pelo uso de testes automatizados, o que nos leva a buscar soluções para o possível *déficit* de produtividade no desenvolvimento dos testes.

Técnicas de geração de código e uso de alto nível de abstração já são utilizados para aumentar a produtividade do desenvolvimento de software. O grande número de ferramentas para modelagem UML que também geram código para diversas linguagens são uma demonstração disso. A geração de código é uma técnica já bem difundida no desenvolvimento de software visando aumentar a produtividade e diminuir a quantidade de erros de codificação, possibilitando que tarefas que são repetitivas e custosas em termo de tempo de desenvolvimento sejam realizadas muito mais rapidamente e com baixíssimo risco de erro humano. O uso do alto nível de abstração permite que seja mais fácil criar, entender e manipular os artefatos de software, ao ocultar detalhes que não são interessantes para o nível em que se trabalha.

Em teste de software, a geração de código de teste é ainda mais difundida, para fazer com que quem esteja testando não consuma sua produtividade em tarefas simples e repetitivas, quando novo código ou testes mais complexos poderiam estar sendo feitos. Porém é difícil encontrar meios de utilizar alto nível de abstração para especificar os testes, especialmente os de unidade, e, mais ainda, de especificar os dados que serão utilizados nos testes.

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. Para permitir isso, utilizaremos técnicas como geração de código e uso de alto nível de abstração para desenvolver uma solução para geração automática de *drivers* e dados de teste para *RESTful Web services* a partir de modelos num padrão de modelagem de testes de mais alto nível baseado em UML, o padrão U2TP.

No presente trabalho, o código é gerado com o objetivo de ser executado sobre Test::Unit, o *framework* padrão de testes unitários em linguagem Ruby, pelos motivos: a linguagem Ruby estar crescendo em popularidade e estar muito relacionada com

aplicativos Web 2.0 e REST; Ruby é uma linguagem expressiva, o que torna o código gerado mais claro para demonstração e avaliação; o *framework* Test::Unit é um *framework* da família XUnit, assim facilmente pode-se ter o código gerado para JUnit, NUnit ou outro *framework* da mesma família.

## 5.1 Visão geral da proposta

Os requisitos que nortearam esta abordagem, em conformidade com Borges (2009), 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 Figura 5.1 apresenta os passos que definem a idéia geral dessa proposta, em conformidade com o trabalho de Borges (2009): especificação, exportação e geração.

1. Na especificação, usando uma ferramenta de modelagem UML, são criados diagramas de classe e sequência que modelam os testes em U2TP.
2. Na exportação, o modelo dos testes é exportado para o formato XMI.
3. 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, e dos dados necessários para a execução dos testes.

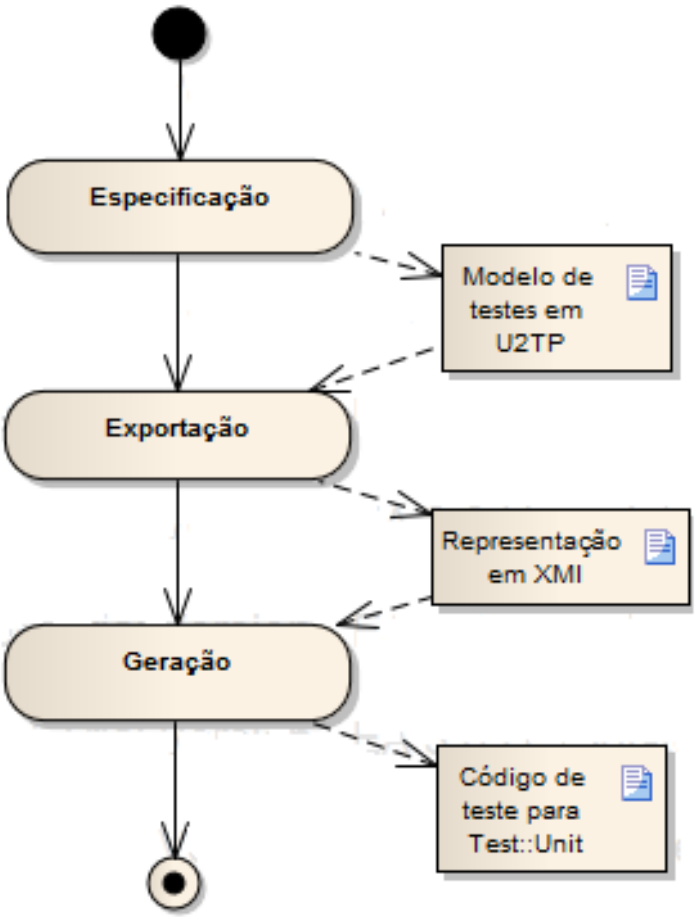



Figura 5.1: Passos da solução REST-Unit+, da especificação à geração dos casos de teste.

## 5.2 Especificação: Modelagem de Testes em U2TP

Esta sessão apresenta uma abordagem para especificar testes de RESTful *Web services* utilizando modelos do Perfil de Teste da UML 2.0. A modelagem dos testes em U2TP é baseada na modelagem apresentada por Biasi (2006), Borges (2009) e nos exemplos apresentados pela OMG em sua especificação do perfil de testes da UML 2.0 (OMG, 2005). Esta modelagem é independente da modelagem UML do *Web service* a ser testado. Isto ocorre porque os modelos U2TP são utilizados para a especificação de testes caixa preta, e não há necessidade de termos conhecimentos da estrutura interna dos Web Services RESTful que são testados. Tais modelos têm como objetivo primário a geração de código e dados de teste e como objetivo secundário documentar a interface do serviço, o padrão do uso dos verbos e códigos HTTP que foram citados anteriormente.

A modelagem dos testes foi dividida em duas partes: modelagem estrutural, representada utilizando diagramas de classe UML, e modelagem comportamental, que utiliza diagramas UML de sequência.

### 5.2.1 Modelagem Estrutural

Na modelagem estrutural é definida a estrutura dos elementos que constituem os testes, as classes e as operações pertencentes a estas classes. Para esta modelagem é usado um diagrama de classes UML.

#### *5.2.1.1 <<TestContext>>*

O contexto de teste é modelado como uma classe com o estereótipo *TestContext*, como ilustrado na Figura 5.2. Esta classe deve existir e ser única no modelo e deve conter pelo menos uma operação com o estereótipo *TestCase*.

#### *5.2.1.2 <<TestCase>>*

Cada caso de teste é modelado como uma operação com estereótipo *TestCase* pertencentes à classe estereotipada como *TestContext*. Essa operação não deve ter valor de retorno, e seu nome deve iniciar com "`test_`".

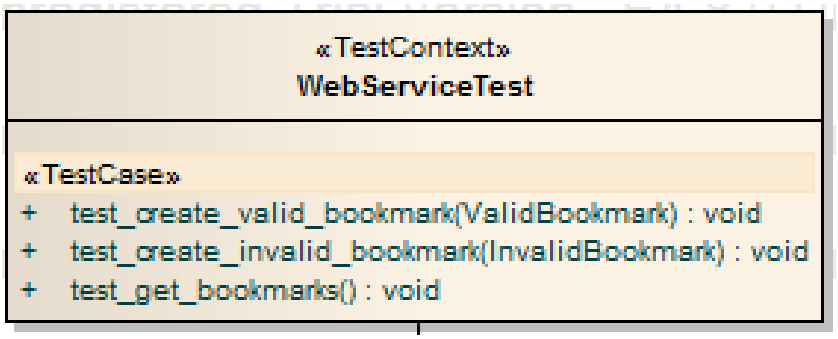



Figura 5.2: Exemplo de representação de TestContext e TestCase na modelagem de testes.

#### *5.2.1.3 <<Sut>>*

O *Web service* a ser testado é modelado como uma classe com o estereótipo *Sut* (Figura 5.3). Esta classe deve ter os atributos que identificam o serviço, com valores inicializados, para permitir a geração automática do código. Estes atributos são o endereço do servidor de rede (atributo "server" do tipo string), a porta em que a aplicação funciona (atributo "port" do tipo inteiro) e opcionalmente um caminho (atributo "path" do tipo string).

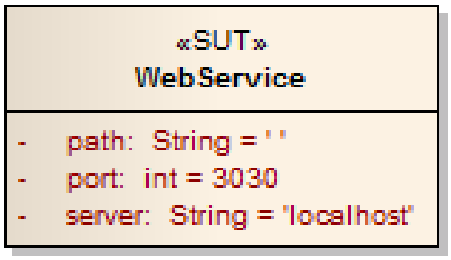



Figura 5.3: Exemplo de representação de um SUT na modelagem de testes.

#### *5.2.1.4 << TestComponent>>*

Este elemento será representado na modelagem como uma classe estereotipada como *TestComponent*. Opcionalmente poderá haver mais classes com este estereótipo. Essas classes servirão para modelar elementos de apoio ao teste do *Web service*.

Mais especificamente para RESTful *Web services*, uma classe HttpStatusCode com estereótipo *TestComponent* deve estar presente no modelo. Ela servirá como contêiner dos códigos de status HTTP que podem ser retornados como resultado de um acesso ao serviço *Web* durante a execução de um teste. Cada *status* é modelado como um atributo com estereótipo *enum* e o valor padrão identificando o seu código de status HTTP. Os nomes para os atributos podem ser dados arbitrariamente, mas por motivos de clareza na modelagem devem estar de acordo com a semântica dos códigos de *status* e devem levar em conta também que o código de teste poderá conter estes nomes. Um exemplo de uma classe HttpStatusCode, com códigos de status HTTP nomeados adequadamente, é apresentado na Figura 5.4.

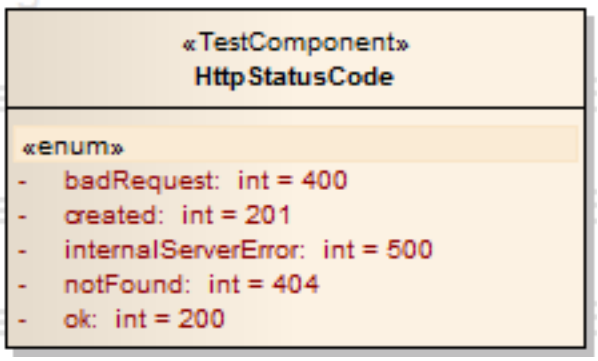



Figura 5.4: Exemplo de representação de um *TestComponent* na modelagem de testes.

#### *5.2.1.5 <<DataPool>>*

O elemento *DataPool* é representado por uma classe, com estereótipo "DataPool", associada a um *TestContext* ou um *TestComponent*, e suas operações para o acesso ao repositório de dados. Ela especifica um contêiner para valores explícitos ou partições de dados que são utilizados por casos de teste. Pode ser apenas referenciado por um *TestContext* ou um *TestComponent*, mas nunca ambos. Zero ou mais *DataPools* podem ser associados a *TestComponents* ou *TestContexts*. A Figura 5.5 mostra um exemplo de representação de *DataPool* e sua relação com as outras classes do modelo.

#### *5.2.1.6 <<DataPartition>>*

Partições de dados são representadas por classes com o estereótipo “DataPartition” que devem estar associadas a uma classe *DataPool*, e as operações necessárias para acessar os dados da partição de dados. Uma *DataPartition* especifica um contêiner para um conjunto de valores. Uma partição de dados pode estar associada apenas a um *DataPool* ou outra *DataPartition*. Zero ou mais *DataPartitions* podem ser definidas para um *DataPool*. Na Figura 5.5, pode ser visto um exemplo de representação de *DataPartition*.

Para representar os dados que serão utilizados nos casos de teste, instâncias da classe estereotipada como *DataPartition* também são representadas no modelo. Desta maneira, o projetista de testes já pode, no momento da especificação do teste, e de maneira mais simples e em alto nível de abstração, definir os parâmetros que serão passados para cada caso de teste, pois estes também serão gerados automaticamente pelo gerador de código.

Cada caso de teste definido na classe estereotipada como *TextContext* deve ter como parâmetro uma instância de uma *DataPartition*, cobrindo assim todas as partições de dados representadas no modelo.

#### *5.2.1.7 <<DataSelector>>*

*DataSelectors* são operações sobre valores ou conjuntos de valores, e podem ser associados a um *DataPool* ou *DataPartition*. São representados por operações estereotipadas com “*DataSelector*”, que acessam dados ou são pertencentes a classes com estereótipo “*DataPool*” ou “*DataPartition*”. Zero ou mais seletores de dados podem ser definidos para *DataPools* ou *DataPartitions*. *DataSelectors* permitem a definição de diferentes estratégias de seleção de dados.

Métodos do tipo “*get*” e “*set*” podem ser considerados como seletores de dados, porém nesta solução eles serão gerados automaticamente a partir dos atributos da classe, sem a necessidade de serem explicitados no modelo. Isto faz com que o modelo tenha mais fácil compreensão e fique menos poluído visualmente. Na Figura 5.5, os *DataSelector* são os métodos “*get*” e “*set*” para os atributos das classes estereotipadas como *DataPartition*, que ficam implícitos.

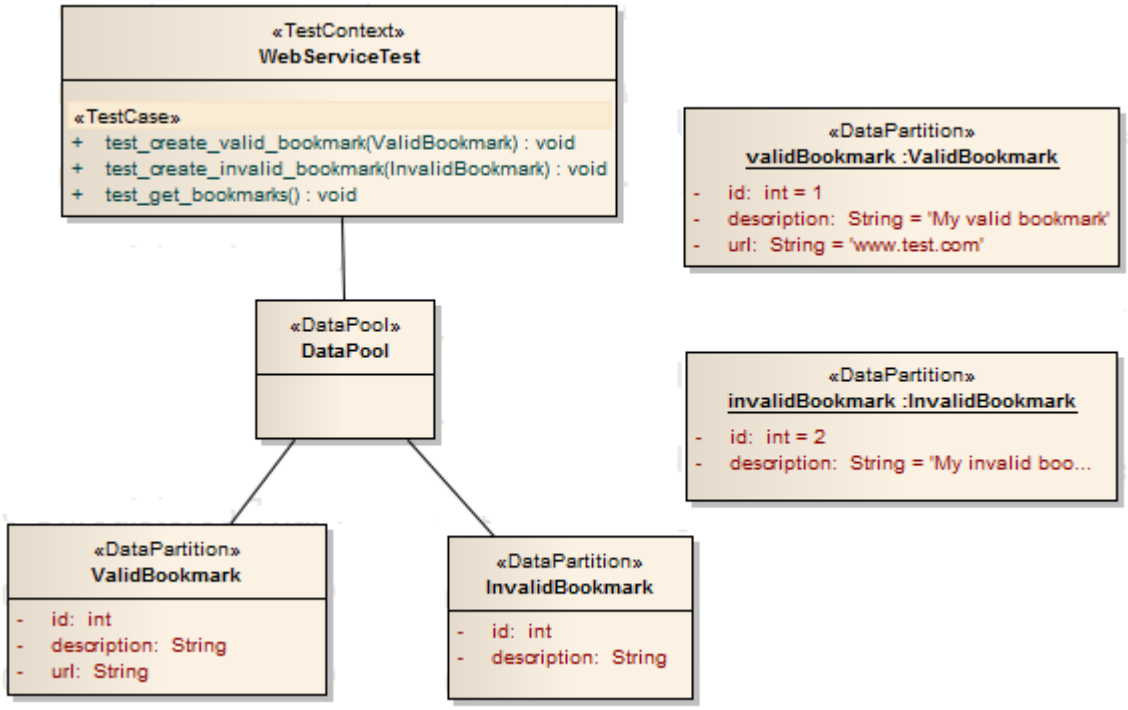



Figura 5.5: Exemplo de representação elementos de dados no modelo de teste.

### 5.2.2 Modelagem Comportamental

A modelagem comportamental define o que é testado e como é testado. Para esta modelagem são usados diagramas UML de sequência. Os elementos chave desta modelagem são os elementos do grupo de comportamento de teste.

#### *5.2.2.1 <<TestCase>>*

Os *TestCases*, e os passos de sua execução, são modelados em um diagrama de sequência. Um exemplo de diagrama de sequência que modela um caso de teste, “test_create_valid_bookmark”, é apresentado na Figura 5.6:

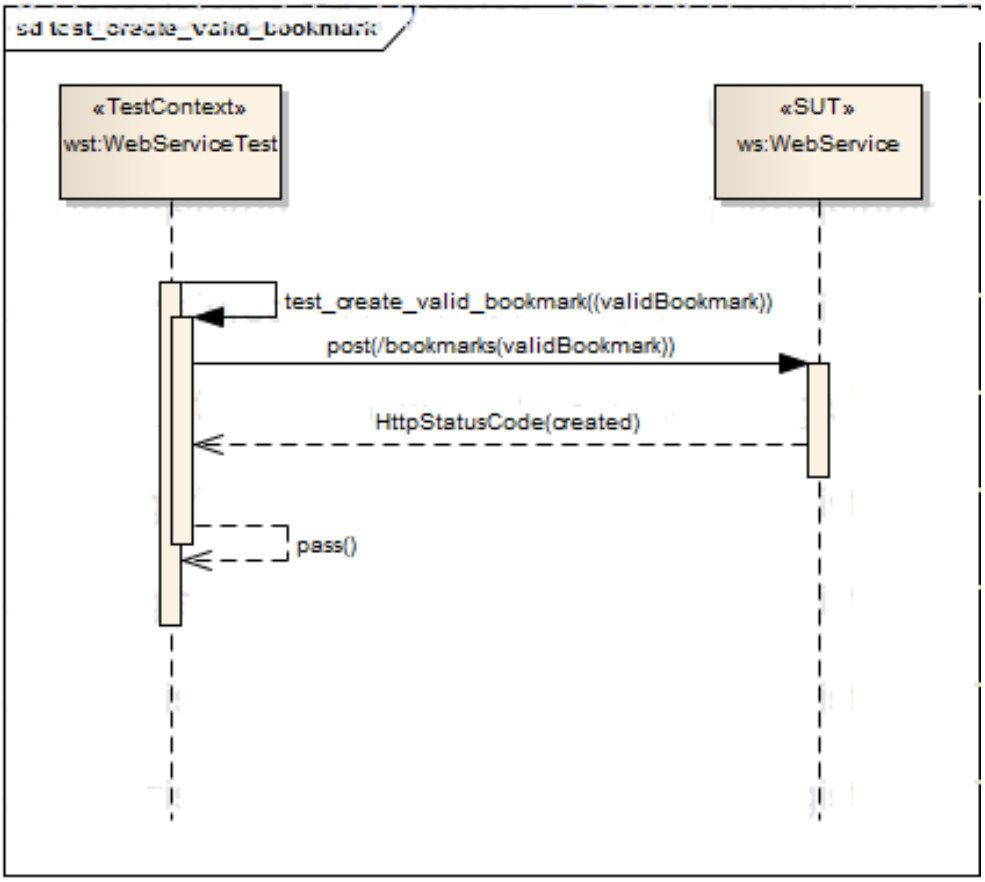



Figura 5.6: Exemplo de modelo de caso de teste em diagrama de sequência.

Para permitir a geração de código dos casos de testes, as seguintes restrições devem ser seguidas na modelagem (Borges, 2009):

- A primeira mensagem do caso de teste dever ser uma auto mensagem por parte do *TestContext* iniciando a execução do caso de teste.
- O retorno do caso de teste deve ser o *veredict "pass"*.
- O nome do diagrama de sequência deve ser o mesmo do caso de teste.
- A penúltima mensagem dentro do caso de teste é sempre o acesso ao *Web service*, e deve ser uma mensagem enviada do *TestContext* par o *Sut*.
- O acesso ao *Web service* deve ser um acesso a um recurso REST, isto é, uma operação HTTP dentre GET, POST, PUT, DELETE executada em um identificador de um recurso, "post /bookmarks(validBookmark)" na figura acima.
- Quando a operação do item anterior for POST ou PUT devem ter como parâmetro um objeto *DataPartition*.
- A última mensagem dentro do caso de teste é sempre o valor esperado de retorno do acesso ao *Web service*, que deve ser um dos valores da enumeração HttpStatusCode.

#### *5.2.2.2 Veredict*

O valor de retorno de todos os casos de teste é sempre um *veredict "pass"*, pois é esperado que o acesso ao *Web service* seja executado sem erros ou falhas, isto é, seja sempre um acesso válido ao *Web service* que retorna um código de status HTTP. Este código HTTP, por sua vez, deve estar em conformidade com a resposta esperada com relação aos dados de teste enviados/recebidos do *Web service*.

## 5.3 Exportação: representação do modelo U2TP em formato XMI

Uma opção para facilitar a geração de código de teste a partir de modelos U2TP é a capacidade de ter estes modelos representados em um formato de mais fácil tratamento computacional. Um formato que tem essa característica é o XMI (XML Metadata Interchange), formado baseado em XML e padronizado pela OMG. Apesar de existir uma especificação de mapeamento da U2TP para XMI definida pela OMG (OMG, 2005), esta especificação não é suportada por muitas ferramentas de modelagem UML, e não foi utilizada na realização deste trabalho. O mapeamento aqui utilizado é o especificado pela OMG para mapeamento MOF/XMI (OMG, 2007).

Diversas ferramentas *open source* e proprietárias disponíveis para modelagem UML e exportação para XMI foram avaliadas. Exemplos incluem Rational Rose (IBM, 2010) e Jude (ChangeVision, 2010). Destas foi escolhida a ferramenta Enterprise Architect versão 8.0 para uso neste trabalho, pois ela atende necessidades de modelagem UML 2.0 e exportação para XMI 2.0, possui versão disponível para *download*, está participando de uma tentativa de padronização do formato XMI (OMG, 2010) e finalmente por familiaridade com a ferramenta. Esta é uma ferramenta comercial de autoria da Sparx Systems (Sparx, 2010).

Os elementos U2TP do modelo podem ser representados por mais de um elemento em representação XML, no documento XMI. Esses elementos no XMI representam uma visão do elemento U2TP que é representado. A Tabela 5.1 demonstra como os elementos da U2TP são representados no documento XMI.

Tabela 5.1: Mapeamento de U2TP para XMI

| *U2TP* | *XMI* | *Observação* |
|---|---|---|
| TestContext | thecustomprofile:TestContext em uml:Class | Estereótipo TestContext |
| | uml:Class | Classe que representa o TestContext |
| | uml:Lifeline | Elemento TestContext no diagrama de sequência |
| Sut | thecustomprofile:Sut em uml:Class | Estereótipo Sut |
| | uml:Class | Classe que representa o e Sut |

| | | |
|---|---|---|
| | uml:Lifeline | Elemento Sut no diagrama de sequência |
| TestComponent | thecustomprofile:TestComponent em uml:Class | Estereótipo TestComponent |
| | uml:Class | Classes TestComponent |
| | uml:Lifeline | Instâncias dos TestComponent no diagrama de sequência |
| TestCase | thecustomprofile:TestCase em uml:Class | Estereótipo TestCase |
| | Operation em uml:Class | Métodos TestCase do TestComponent |
| | uml:Interaction | Conjunto de interações do teste |
| | uml:Message | Mensagem que representa o caso de teste no diagrama de sequência |
| DataPool | thecustomprofile:DataPool em uml:Class | Estereótipo DataPool |
| | uml:Class | Classes DataPool |
| DataPartition | thecustomprofile:DataPartition em uml:Class | Estereótipo DataPartition |
| | uml:Class | Classes DataPartition |
| | uml:InstanceSpecification | Instâncias das classes DataPartition |
| DataSelector | thecustomprofile:DataSelector em uml:Class | Estereótipo DataSelector |
| | Operation em uml:Class | Métodos DataSelector do DataPool ou DataPartition |
| | uml:Interaction | Conjunto de interações de dados |
| | uml:Message | Mensagem que representa o seletor de dados no diagrama de sequência |

## 5.4 Geração: código Test::Unit a partir do XMI

Esta sessão apresenta a geração de código a partir de modelagem U2TP. Para facilitar a compreensão, um exemplo será utilizado, em que será gerado código de teste a partir da modelagem U2TP, fazendo um paralelo com o caso geral.

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 JUnit, NUnit ou outro *framework* da mesma família.

### 5.4.1 Mapeamento U2TP para Test::Unit

O seguinte exemplo ilustra o mapeamento de uma modelagem U2TP. A Figura 5.7 mostra o diagrama de classes, com as estruturas do teste.

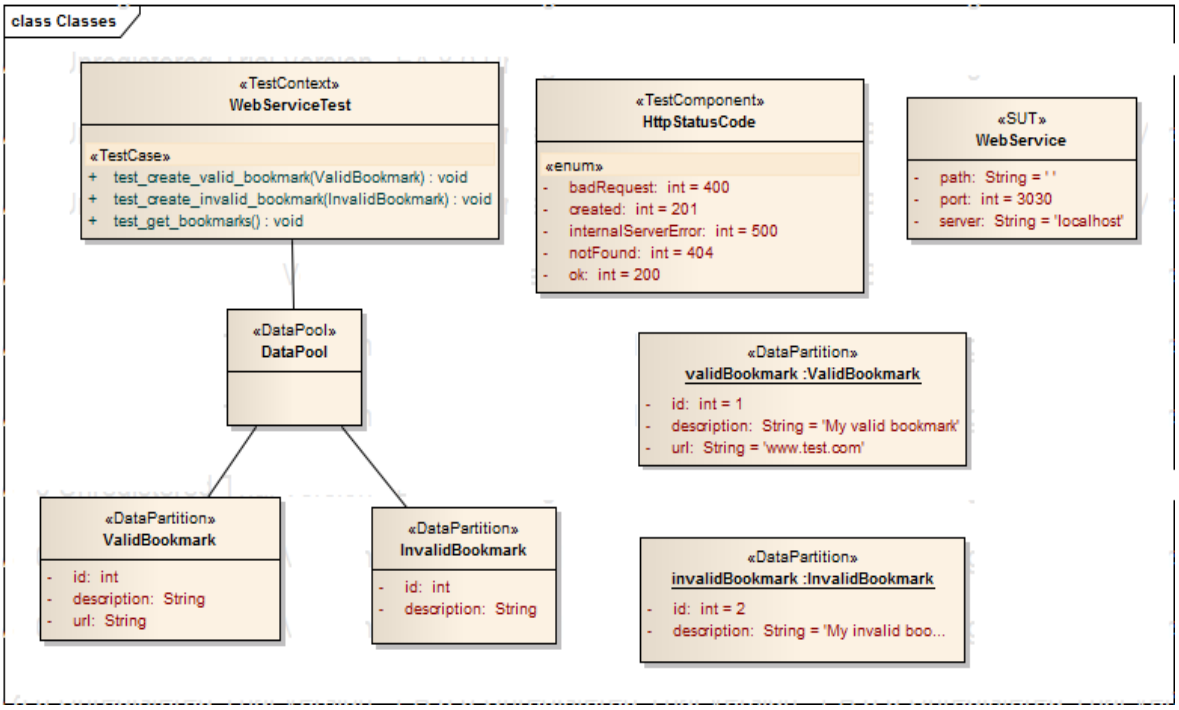



Figura 5.7: Diagrama de classe: modelagem estrutural do teste.

A Figura 5.8 mostra um caso de teste utilizando um elemento de uma partição de dados.

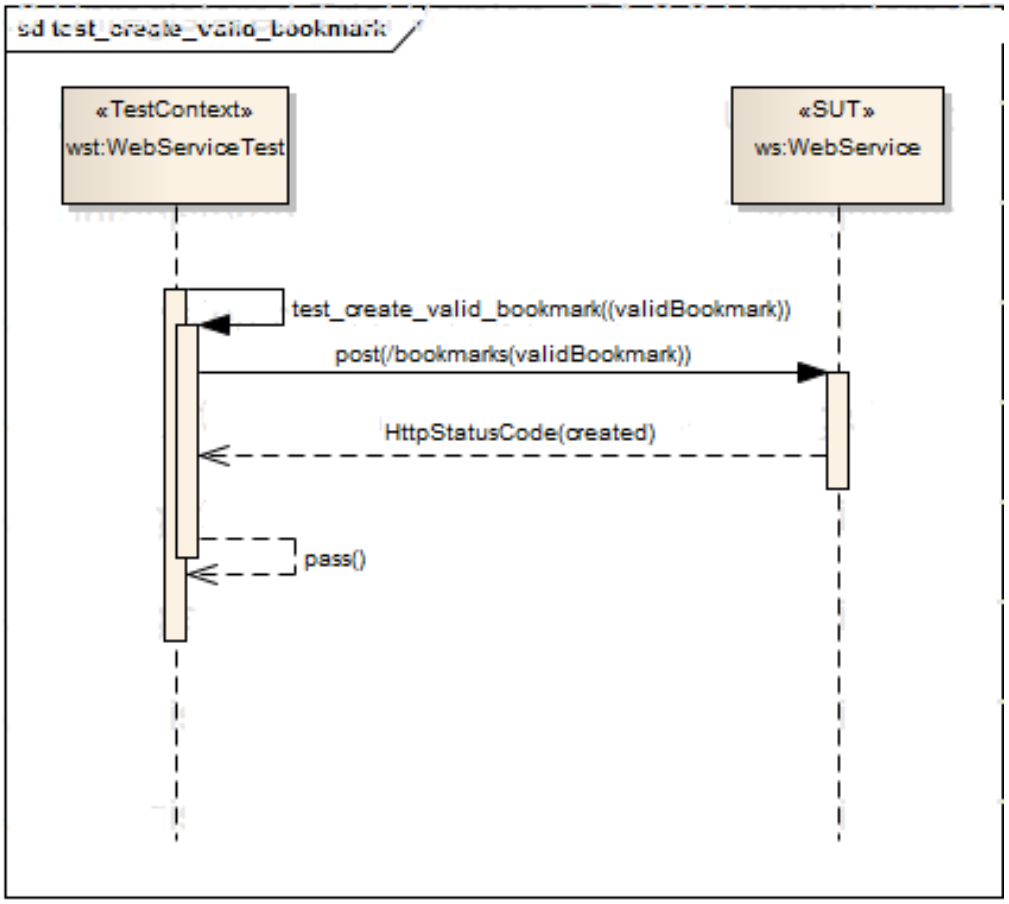



Figura 5.8: Caso de teste “test_create_valid_bookmark”.

A Figura 5.9 apresenta outro caso de teste utilizando outra partição de dados.

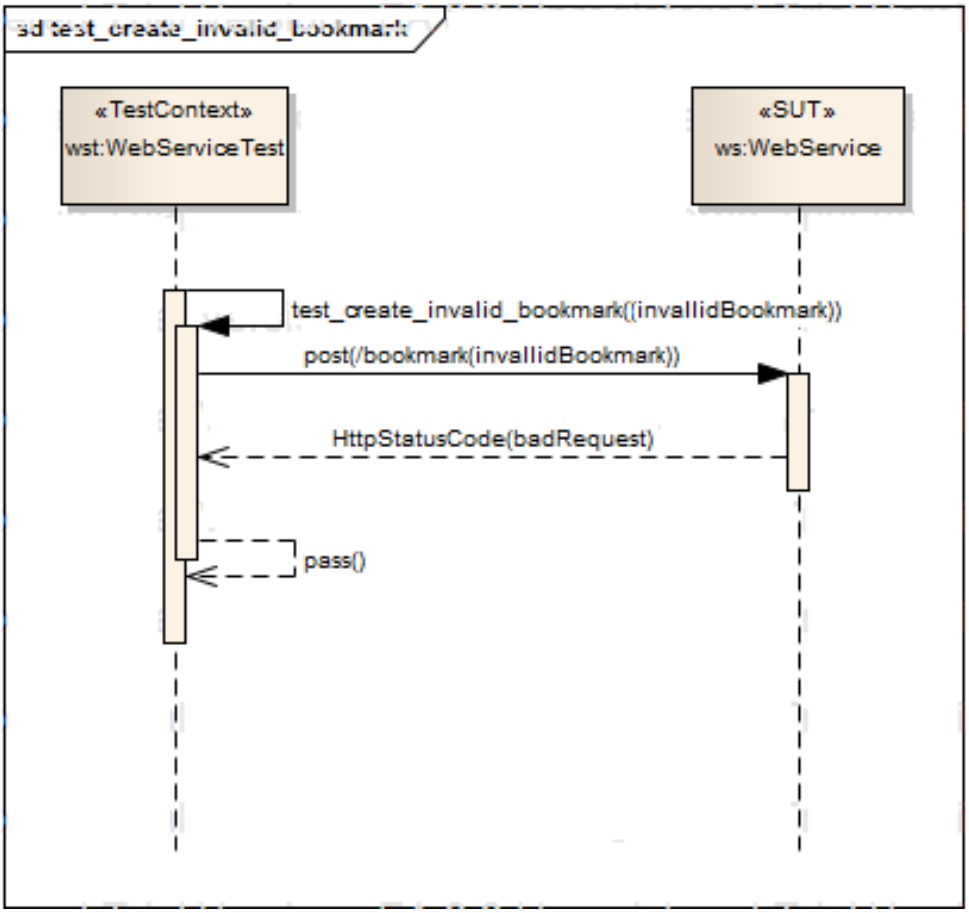



Figura 5.9: Caso de teste “test_create_invalid_bookmark”.

E a Figura 5.10 mostra um caso de teste para recuperação de todos os itens de um recurso.

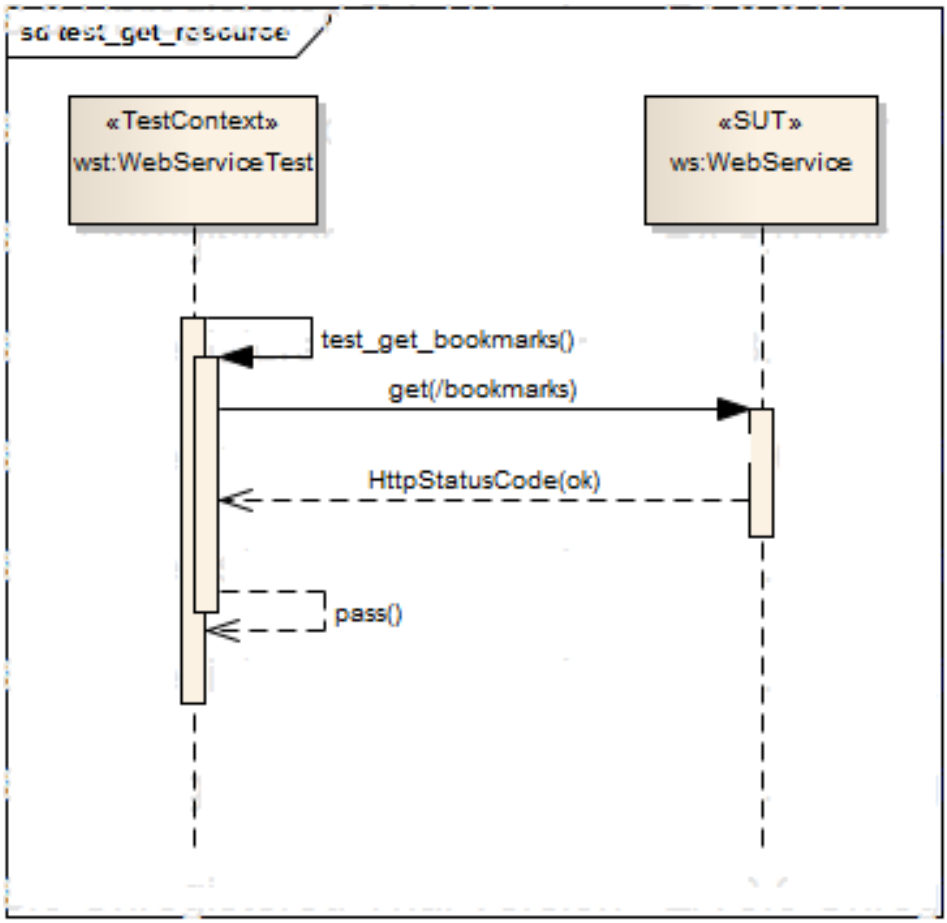
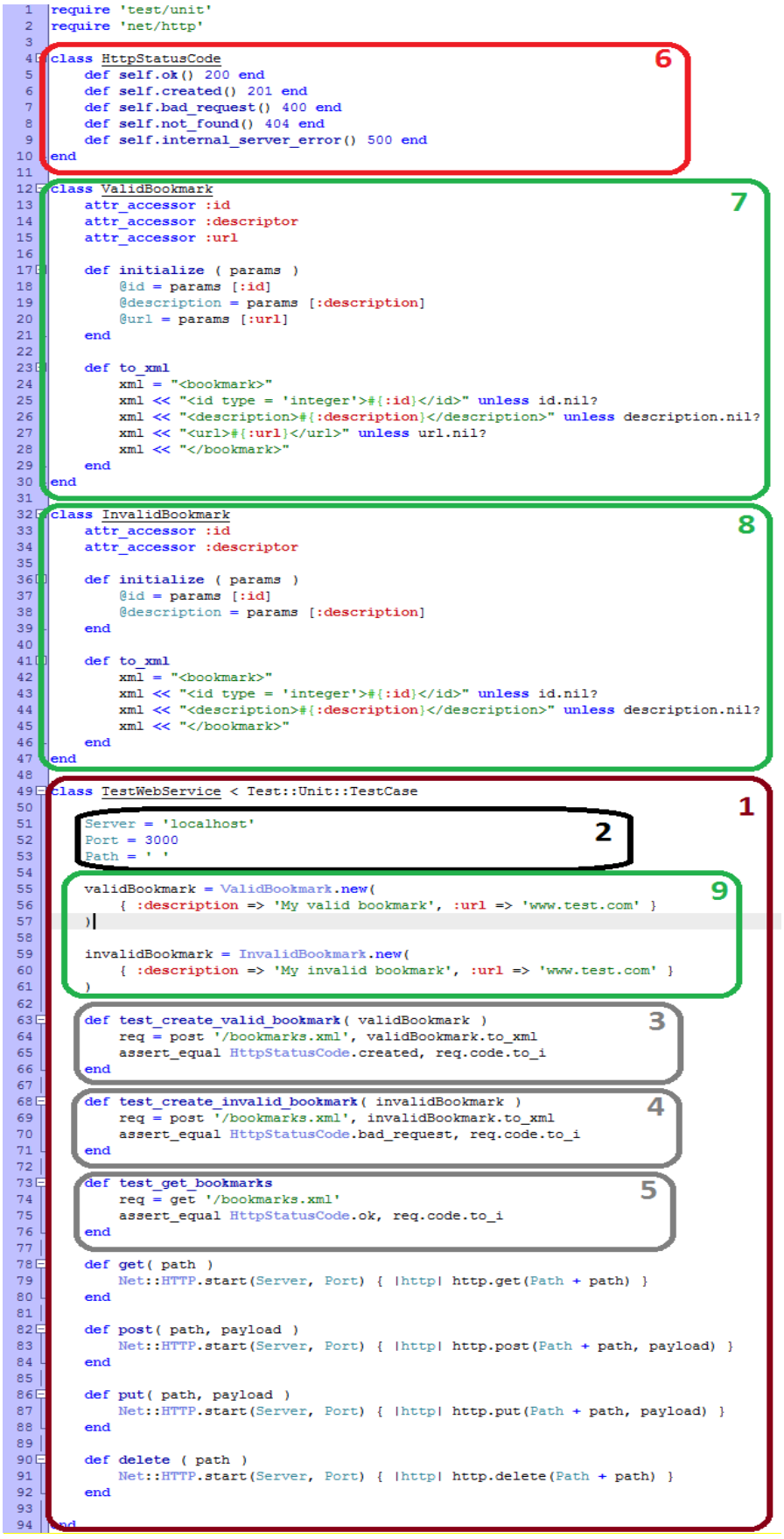



Figura 5.10: Caso de teste “test_get_bookmarks”.

A modelagem U2TP apresentada nas figuras anteriores é então mapeada para código Test::Unit, como o exemplificado na Figura 5.11.




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

class HttpStatusCode
    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 ValidBookmark
    attr_accessor :id
    attr_accessor :descriptor
    attr_accessor :url

    def initialize ( params )
        @id = params [:id]
        @description = params [:description]
        @url = params [:url]
    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 << "</bookmark>"
    end
end

class InvalidBookmark
    attr_accessor :id
    attr_accessor :descriptor

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

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

class TestWebService < Test::Unit::TestCase

    Server = 'localhost'
    Port = 3000
    Path = ' '

    validBookmark = ValidBookmark.new(
        { :description => 'My valid bookmark', :url => 'www.test.com' }
    )

    invalidBookmark = InvalidBookmark.new(
        { :description => 'My invalid bookmark', :url => 'www.test.com' }
    )

    def test_create_valid_bookmark( validBookmark )
        req = post '/bookmarks.xml', validBookmark.to_xml
        assert_equal HttpStatusCode.created, req.code.to_i
    end

    def test_create_invalid_bookmark( invalidBookmark )
        req = post '/bookmarks.xml', invalidBookmark.to_xml
        assert_equal HttpStatusCode.bad_request, req.code.to_i
    end

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

    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
```

Figura 5.11: Código em Test::Unit do teste modelado nas figuras 5.7 a 5.10.

A Tabela 5.2 sumariza o mapeamento correspondente entre os elementos U2TP e o código em Test::Unit, destacando a(s) região(ões) do código que os representam.

Tabela 5.2: Mapeamento dos elementos U2TP para código Test::Unit

| *U2TP* | *Test::Unit* | *Exemplo no código (*Figura 5.11*)* |
|---|---|---|
| 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, 4 e 5 |
| TestComponent | Classe HttpStatusCode | Região 6 |
| DataPool | Classe DataPool | Nenhuma. É implementada somente quando possui atributos ou métodos. |
| DataPartition | Classes ValidBookmark e InvalidBookmark / Instanciação dos objetos validBookmark e invalidBookmark | Regiões 7, 8 e 9 |
| Data Selector | Métodos das classes DataPool e DataPartition | Regiões 7 e 8 (métodos “`attr_accessor`”, “`initialize`”, “`to_xml`”) |

#### *5.4.1.1 TestContext*

O elemento *TestContext* é mapeado para uma subclasse de Test::Unit::TestCase. Esta classe tem função de agrupar os casos de teste (*TestCase*) e conter a implementação dos métodos auxiliares para execução de operações GET/POST/PUT/DELETE. O nome deste elemento no modelo pode ser usado como nome da classe no código e também como nome do arquivo .rb. Na Figura 5.11 este elemento está codificado nas linhas 49 a 96 (região 1) como a classe `TestWebService`.

#### *5.4.1.2 Sut*

O elemento *Sut* não tem mapeamento direto no código, já que ele representa o *Web service* a ser testado. Porém, os valores dos atributos deste elemento são utilizados no *TestContext* para endereçar o serviço a ser testado. Estes atributos aparecem no código

nas linhas 51 a 53 (região 2) da Figura 5.11 como constantes dentro da classe `WebServiceTest`.

#### *5.4.1.3 TestComponent*

Os elementos *TestComponent* são codificados como uma classe. A classe `HttpStatusCode` representada no diagrama de classe (Figura 5.7) mapeia para uma classe em Ruby (linhas 4 a 10). Para melhor identificação está destacado na região 6 da Figura 5.11. Cada um dos atributos desta classe foi modelado como um *accessor* e teve seu nome transformado para estilo padrão Ruby de nomenclatura.

#### *5.4.1.4 DataPool*

O elemento *DataPool* é mapeado no código como uma classe com seus atributos e métodos que foram representados no modelo. No caso deste exemplo, o elemento não aparece no código, pois no modelo não havia atributos ou métodos pertencentes a ele.

#### *5.4.1.5 DataPartition*

O elemento *DataPartition* é mapeado no código como uma classe com seus atributos e métodos que foram representados no modelo (regiões 7 e 8; linhas 12 a 30 e 32 a 47, respectivamente). Cada atributo da classe no modelo é mapeado para um atributo no código. O construtor da classe (método `initialize)` recebe um *hash* de pares nome/valor para inicialização dos valores dos atributos. O ponto chave desta classe é o método `to_xml`, que deve gerar, a partir dos valores inicializados nos atributos, um código XML que representa os dados a serem enviados para o servidor como *payload* das operações de POST e PUT.

Os objetos da classe *DataPartition*, que representam os dados que serão utilizados pelos casos de teste são representados no código pela instanciação de objetos da classe *DataPartition*, na classe *TestContext* (região 9; linhas 55 a 57 e linhas 59 a 61).

#### *5.4.1.6 DataSelector*

O elemento *DataSelector* é representado pelos métodos gerados para as classes *DataPool* e *DataPartition*. Neste exemplo temos os métodos: `attr_accessor`, que define os métodos tipo "*get*" e "*set*" para os atributos das classes (linhas 13 a 15 e 33 a 34); `initialize`, que é o construtor das classes (linhas 17 a 21 e 36 a 39); e `to_xml`, que converte os atributos para o formato XML aceito pelo *Web service* (linhas 23 a 29 e 41 a 46).

#### *5.4.1.7 TestCase*

Os elementos *TestCase* são métodos dentro da classe que representa o elemento *TestContext*, e esses métodos têm o mesmo nome do *TestCase*. No código do exemplo (Figura 5.11) eles aparecem como os métodos `test_create_valid_bookmark`,

`test_create_invalid_bookmark` e `test_get_bookmarks`, destacados nas regiões 3 (linhas 63 a 66), 4 (linhas 68 a 71) e 5 (linhas 73 a 76). Esses métodos são mapeados da seguinte forma:

- As mensagens que modelam uma operação HTTP (GET, POST, PUT, DELETE) são mapeadas para um acesso ao *Web service*. Esse acesso deve passar dados obtidos de um elemento *DataPartition* através do método `to_xml` no caso dos verbos POST e PUT. O valor de retorno das operações de acesso ao *Web service* é guardado na variável local `req`. Essas mensagens sempre aparecem com a penúltima mensagem dentro do corpo do caso de teste no diagrama de sequência.
- O código de *status* HTTP que retorna do acesso ao *Web service* é comparado com o valor esperado, modelado como a última mensagem dentro do corpo do caso de teste no diagrama de sequência.

#### *5.4.1.8 Veredict*

O *veredict*, valor de retorno do caso de teste no modelo, não tem mapeamento direto no código. Este valor, sempre *pass* nesta solução, é o resultado esperado da execução do caso de teste e significa sucesso na execução do teste.

### 5.4.2 Gerador de Código Para Test::Unit

Nas sessões anteriores foi apresentado o mapeamento de conceitos entre U2TP e Test::Unit e entre U2TP e XMI. A geração do XMI a partir de um modelo U2TP é feita pela ferramenta de modelagem UML, sendo necessária a existência de um algoritmo para mapeamento entre o documento XMI e código Test::Unit, para que seja possível, a partir de uma modelagem U2TP em uma ferramenta UML, que todo o processo de geração de código possa ser feito automaticamente. O objetivo dessa sessão é apresentar um algoritmo que a partir do documento XMI gere *drivers* e dados de teste em Test::Unit levando em conta a especificação do mapeamento entre U2TP e XMI e o mapeamento entre U2TP e Test::Unit apresentados nas sessões anteriores.

Neste trabalho, para a geração de código, foi utilizada XSLT (eXtensible Stylesheet Language for Transformation), uma linguagem para transformação de documentos XML em outros documentos XML (W3C, 1999). Como XMI é um tipo de documento XML, é possível aplicar as regras de transformação XSLT para gerar outro documento em texto puro, neste caso o código Test::Unit.

#### *5.4.2.1 O Processo de Transformação da XSLT*

XSLT foi feita para ser usada como parte de XSL (eXtensible Stylesheet Language), uma linguagem de estilos para XML, e inclui vocabulário XML para especificar formatação. XSL especifica o estilo de um documento XML utilizando XSLT para descrever como este é transformado em outro documento XML que usa vocabulário de formatação. O documento original não é alterado, um novo documento é criado com base no conteúdo do existente. O novo documento pode ser gerado em sintaxe XML ou outro formato, como HTML ou texto.

O processo XSLT (Tittel, 2003) consiste de uma transformação de árvore, que considera os documentos XML e XSL como entrada e gera uma árvore de resultados baseada nas instruções contidas no arquivo XSL. Isto pode incluir informações de filtragem (nodos a incluir ou não) ou a reordenação de dados XML em um *layout* mais apresentável antes de aplicar os atributos de estilo finais, no momento da exibição.

O conjunto completo de passos envolvidos incluem (Tittel, 2003):

1. Um analisador XML interpreta o documento XML e forma a árvore.
2. A árvore é entregue para um processador XSLT.
3. O processador XSLT compara os nodos na árvore com as instruções contidas na folha de estilo referida (XSL).
4. Quando o processador encontra um "casamento", produz a saída de um fragmento da árvore (árvore de resultados).
5. A árvore é enviada para um agente de usuário, em um formato como HTML ou texto.

XSL utiliza *templates* que são associados com um elemento XML. Um *template* XSL usa o atributo "*match*" ou "*pattern*" para indicar qual é o *template* necessário e então, cria a transformação a partir dali. São oferecidos diversos recursos para a definição da folha de estilos: laços, condicionais, expressões, tipos de dados, variáveis locais e globais, operadores e eixos da XPath, entre outros.

#### *5.4.2.2 A Geração de Código*

Para a transformação do documento XMI em código Test::Unit, foram definidas regras de transformação, conforme as *tags* XML encontradas no documento gerado pela ferramenta de modelagem UML. Estas regras seguem um algoritmo, que pode ser descrito da seguinte maneira:

1. Para cada classe: verifica se possui métodos ou atributos (que decide se a classe será gerada ou não) e se o estereótipo é um dos de interesse (*Sut*, *TestContext*, *TestComponent*, *DataPool* ou *DataPartition*). Se for o caso, escreve a declaração da classe, com os atributos (se houver), com seus valores padrão, e os métodos (se houver). Para que um método seja gerado, é necessário que esteja presente no modelo um diagrama de sequência que o descreva.
2. Se a classe for *TestComponent*: escreve em código os atributos que foram definidos no modelo. No caso de *HttpStatusCode*, os atributos serão definidos como itens de uma enumeração, pois receberam o estereótipo *enum*.
3. Se a classe for *DataPool*: escreve em código os atributos que foram definidos no modelo, com seus tipos e seus métodos *get* e *set* (`attr_accessor`). Gera o método construtor para os atributos (`initialize`) e o método para geração do XML a partir dos atributos (`to_xml`).
4. Se a classe for *DataPartition*: escreve em código os atributos que foram representados no modelo, com seus tipos e seus métodos *get* e *set* (`attr_accessor`). Gera o método construtor para os atributos (`initialize`) e o método para geração do XML a partir dos atributos (`to_xml`).

5. Se a classe for *TestContext*: escreve em código os atributos do SUT definidos no modelo, a instanciação dos objetos *DataPartition* representados no modelo; define métodos para os que estiverem estereótipo *TestCase*, e métodos para definir as operações HTTP: GET, POST, PUT e DELETE.
6. Ainda na classe *TestContext*, para cada *TestCase*: a primeira mensagem ("`test_`") é utilizada para criar o cabeçalho do método; a mensagem de acesso ao *Web service* é gerada com o caminho e o *payload* (se aplicável); após a mensagem de retorno do código de *status* HTTP, uma asserção é feita, comparando o código recebido com o esperado; a mensagem "`pass`" é ignorada, sendo substituída pelo resultado da asserção.

Um trecho da folha de estilos XSL com as regras para transformação XSLT pode ser visto na Figura 5.12.

```
<?xml version="1.0" ?>
- <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform" xmlns:uml="http://schema.omg.org/spec/UML/2.1"
  xmlns:xmi="http://schema.omg.org/spec/XMI/2.1" xmlns:thecustomprofile="http://www.sparxsystems.com/profiles/thecustomprofile/1.0">
  <xsl:output method="text" omit-xml-declaration="yes" />
  <!-- Document Root -->
- <xsl:template match="/xmi:XMI/uml:Model">
    <xsl:text>require 'test/unit' require 'net/http'</xsl:text>
    <!-- Classes -->
  - <xsl:for-each select="packagedElement/packagedElement[@xmi:type='uml:Class']">
      <!-- testa se tem atributos ou métodos -->
    - <xsl:if test="count(ownedAttribute) + count(ownedOperation) != 0">
        <!-- descobre o estereótipo de cada classe -->
      - <xsl:choose>
          <!-- DataPartition -->
        - <xsl:when test="ancestor::uml:Model/thecustomprofile:DataPartition/@base_Class = @xmi:id">
            <!-- declaração da classe -->
            <xsl:text>class</xsl:text>
            <xsl:value-of select="@name" />
            <xsl:text />
            <!-- declaração dos atributos -->
          - <xsl:for-each select="ownedAttribute">
              <xsl:text>attr_accessor :</xsl:text>
              <xsl:value-of select="@name" />
```

Figura 5.12: Exemplo de folha de estilo XSL para processamento XSLT.

## 5.5 Exemplo de Aplicação

Nesta sessão será apresentado um exemplo ilustrando a aplicação de REST-Unit+. Para isto, é apresentada uma aplicação desenvolvida em *framework* Ruby on Rails que expõe uma API REST, o modelo de alguns casos de teste para esta aplicação a exportação para XMI e a geração e execução do código dos testes. Para geração do código dos testes foi desenvolvida uma folha de estilo XSL com o algoritmo representado na sessão 5.4.2.2 para a geração de código Teste::Unit em linguagem Ruby.

### 5.5.1 Exemplo de RESTful Web Service

Uma aplicação foi desenvolvida em Ruby on Rails por motivos didáticos, sendo um exemplo de Web Service RESTful. Essa aplicação é um repositório de links. Não há gerência de usuários, permissões ou segurança.

#### *5.5.1.1 Protocolo da Aplicação*

Essa aplicação tem uma arquitetura REST, organizada em apenas um tipo de recursos: *bookmarks*. Os *bookmarks* têm as propriedades: id (obrigatório), descrição e URL (obrigatório), além de propriedades de informação de acesso e criação.

Os *bookmarks* são identificados dentro da aplicação por dois formatos de URIs:

- Todos os *bookmarks*: `/bookmarks`
- Um *bookmark* específico com identificador **id**: `/bookmarks/`**`id`**

São oferecidas as operações sobre os recursos apresentadas na Tabela 5.3. Essas operações seguem os padrões de boas práticas aceitos para a semântica dos métodos HTTP apresentados na sessão 2.2.1, Boas Práticas REST:

Tabela 5.3: Protocolo da aplicação exemplo.

| **Recurso** | **URL** | **Métodos** | **Operação** | **Códigos Status** |
|---|---|---|---|---|
| Todos bookmarks | /bookmarks | GET | Busca | 200 |
| | | POST | Criação | 201, 400 |
| Um Bookmark | /bookmarks/**id** | GET | Busca | 200, 400, 404 |
| | | PUT | Atualização | 200, 400, 404 |
| | | DELETE | Deleção | 200, 400, 404 |

Além de uma interface Web, a API da aplicação pode ser acessada em forma de um *RESTful Web service*, o formato de dados utilizado no *Web service* é XML. Pode-se, inclusive, acessar o *Web service* pelo próprio navegador (Figura 5.13). Para utilização do formato de dados XML, basta indicar o formato de dados desejado pela extensão da URL. Por exemplo, a coleção de *bookmarks* em XML é identificada como: `/bookmarks.xml`.

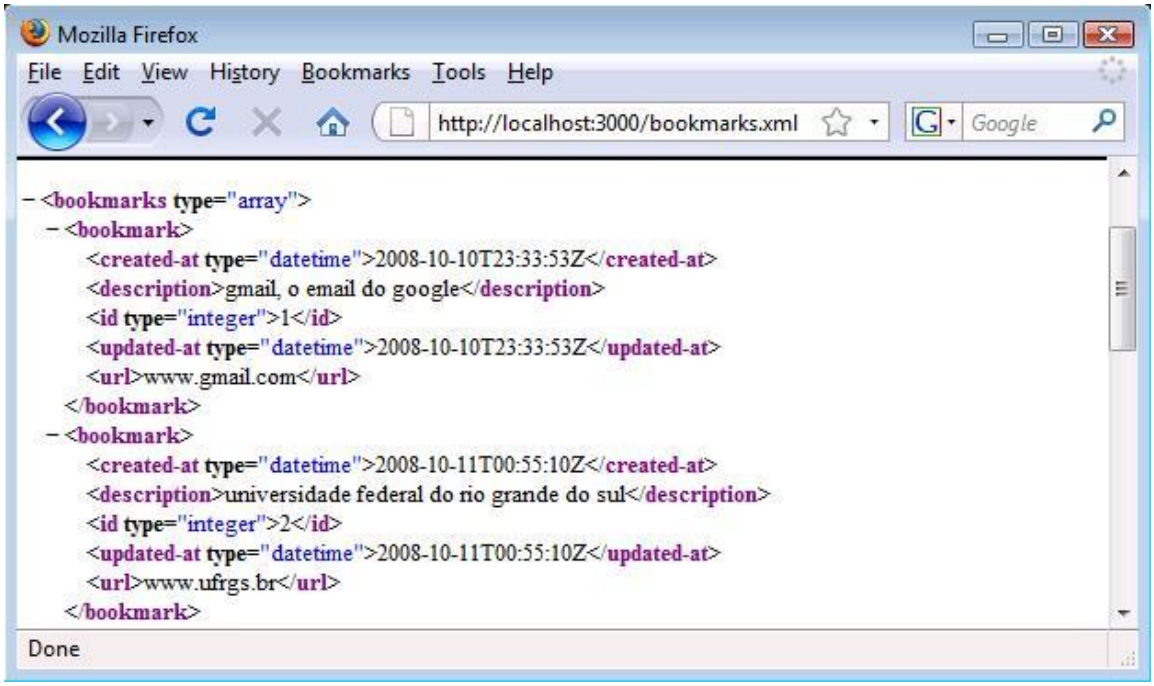



Figura 5.13: Busca da lista de bookmarks em formato XML.

### 5.5.2 Especificação dos Testes

Os testes para a aplicação apresentada acima foram modelados em U2TP utilizando a ferramenta Enterprise Archtect 8.0. Para este exemplo, são modelados três casos de teste.

1. *Criação de um novo bookmark com dados válidos (test_create_valid_bookmark)*

Este caso de teste é identificado por uma de criação de um novo *bookmark* no recurso /`bookmarks`. Este caso envia todos os dados do novo recurso (id, descrição e url) e espera que a criação seja feita sem erros, recebendo como código de *status* HTTP o valor 201 (Created).

2. *Criação de um novo bookmark com dados inválidos (test_create_invalid_bookmark)*

Este caso de teste é identificado por uma de criação de um novo *bookmark* no recurso /`bookmarks`, porém sem o atributo "url". Este caso envia os dados do novo recurso (id e descrição) e espera que a criação não seja feita, recebendo como código de *status* HTTP o valor 400 (Bad Request).

3. *Acesso a coleção de bookmarks*

Este caso de teste é identificado por uma operação de busca ao recurso `/bookmarks`. Esse caso de teste efetua a busca e espera um acesso sem erros, recebendo como código de *status* HTTP 200 (Ok).

#### *5.5.2.1 Modelo U2TP dos Testes*

Utilizando a ferramenta de modelagem UML, foram criados os diagramas que modelam os casos de teste. Esses diagramas são divididos em um diagrama de classe que contém os elementos que participam dos testes e um diagrama de seqüência para modelar o comportamento de cada caso de teste. O modelo U2TP dos testes pode ser visto na Figura 5.14, na Figura 5.15, na Figura 5.16 e na Figura 5.17. A Figura 5.14 apresenta a modelagem estrutural do teste, utilizando o diagrama de classes.

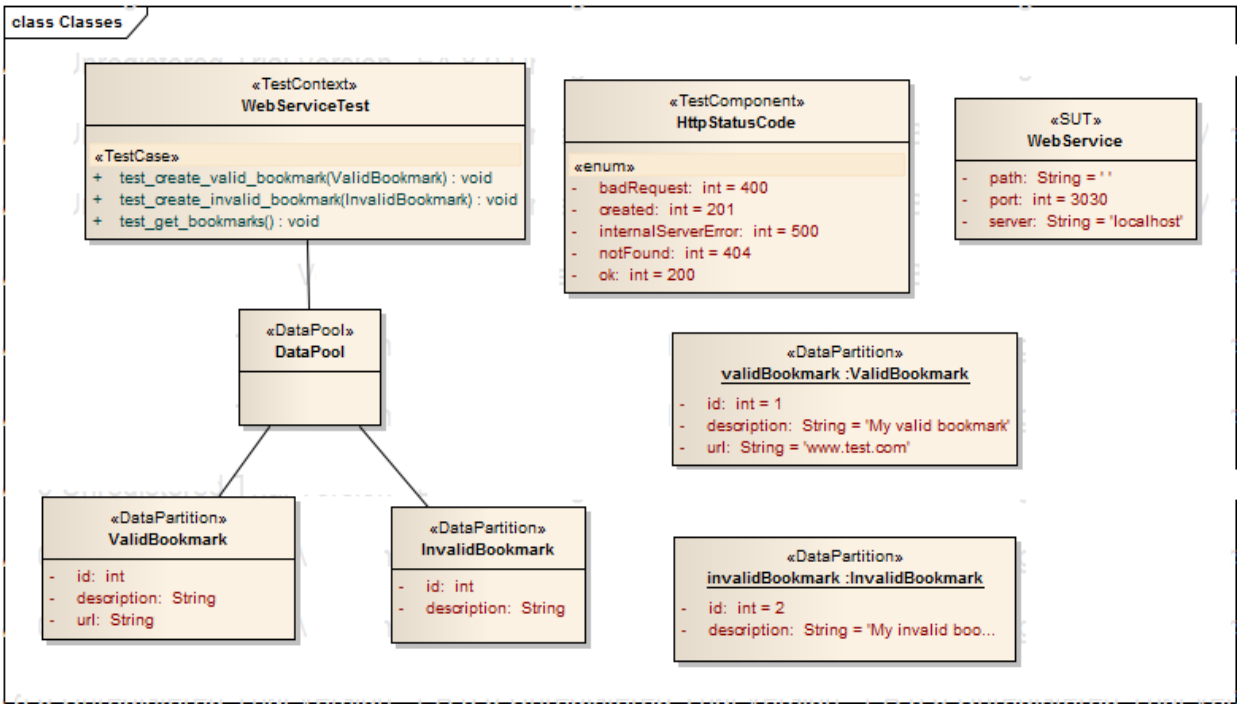



Figura 5.14: Modelo dos casos de teste em U2TP para a aplicação exemplo.

A Figura 5.15 apresenta a modelagem em diagrama de sequência do caso de teste "test_create_valid_bookmark", que testa a criação de um item de uma das partições de dados (*ValidBookmark*).

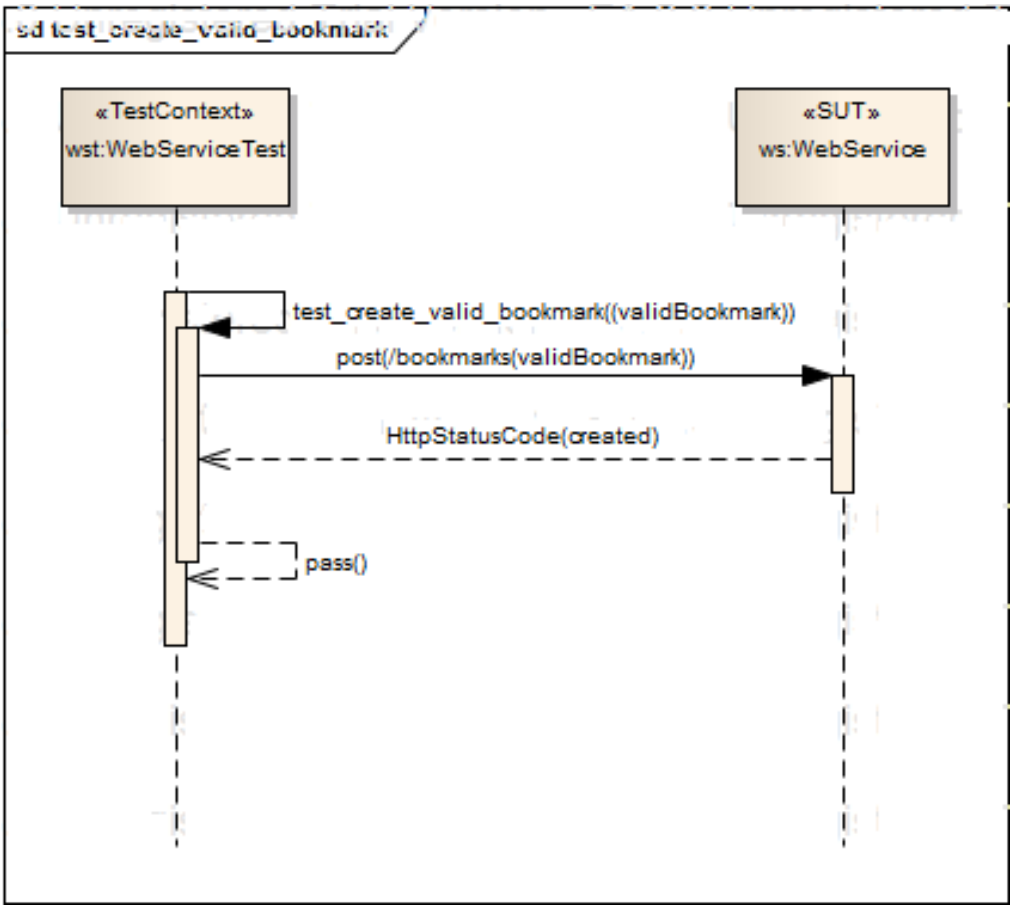



Figura 5.15: Modelo dos casos de teste em U2TP para a aplicação exemplo (caso de teste "test_create_valid_bookmark").

A Figura 5.16 apresenta a modelagem em diagrama de sequência do caso de teste “test_create_invalid_bookmark”, que testa a criação de um item da outra partição de dados (*InvalidBookmark*).

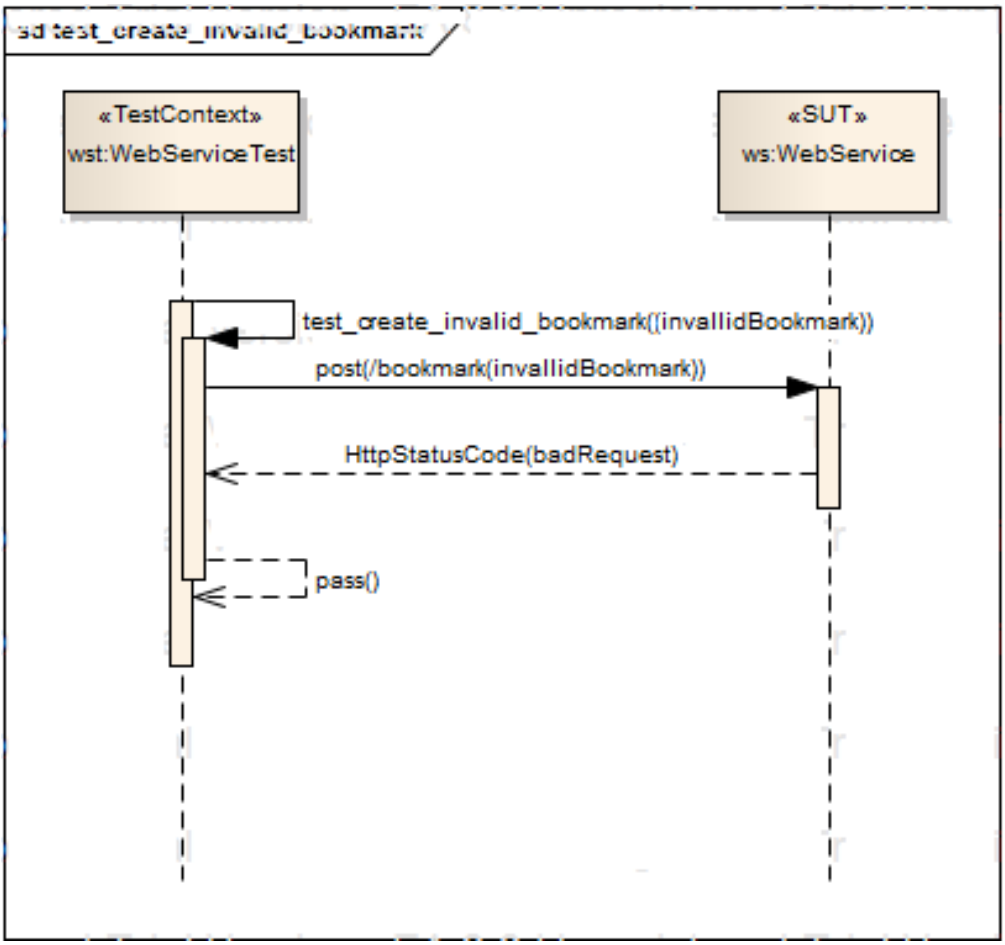



Figura 5.16: Modelo dos casos de teste em U2TP para a aplicação exemplo (caso de teste “test_create_invalid_bookmark”).

A Figura 5.17 apresenta a modelagem em diagrama de sequência do caso de teste "test_get_bookmarks", que testa a recuperação de todos os recursos *bookmarks*.

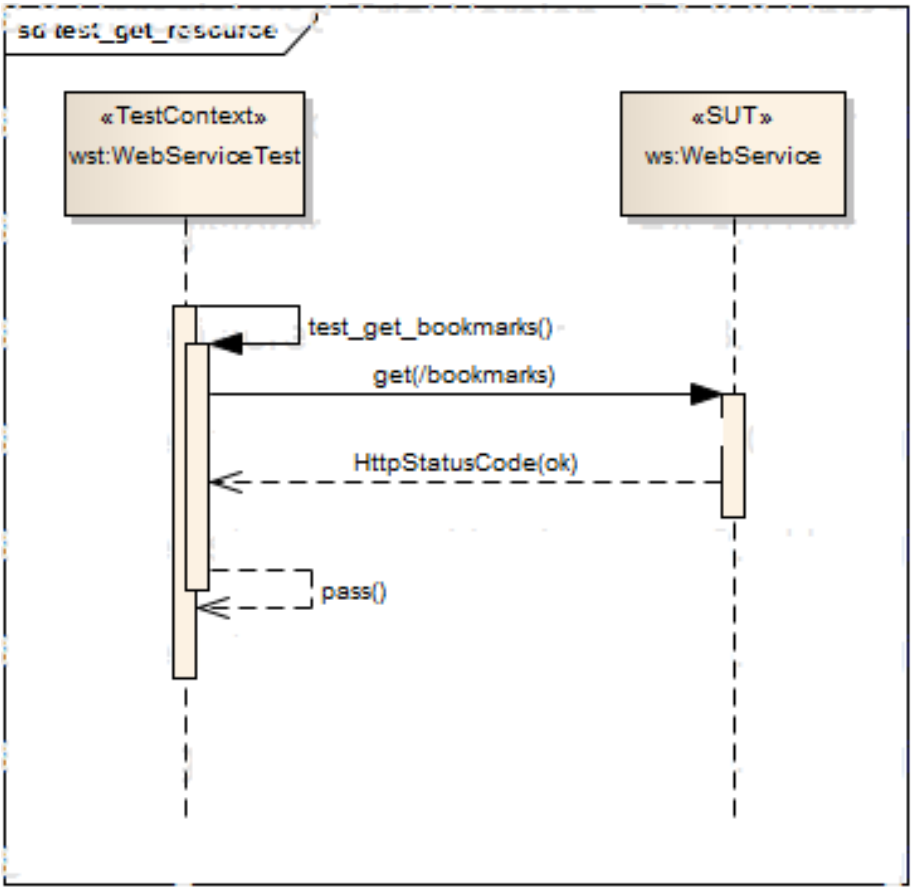



Figura 5.17: Modelo dos casos de teste em U2TP para a aplicação exemplo (caso de teste "test_get_bookmarks").

### 5.5.3 Exportação para XMI

A ferramenta Enterprise Architect, como já foi apresentado, possui a funcionalidade e exportação dos diagramas UML para arquivo no formato XMI. Utilizando esta funcionalidade, o modelo dos casos de teste foi exportado para um arquivo XMI para servir de entrada para o protótipo de transformações XSLT para Test::Unit. Um trecho da representação em XMI dos casos de testes é ilustrada na Figura 5.18, e teve trechos colapsados por motivos de clareza.

```
<?xml version="1.0" encoding="windows-1252" ?>
- <xmi:XMI xmi:version="2.1" xmlns:uml="http://schema.omg.org/spec/UML/2.1" xmlns:xmi="http://schema.omg.org/spec/XMI/2.1"
    xmlns:thecustomprofile="http://www.sparxsystems.com/profiles/thecustomprofile/1.0">
    <xmi:Documentation exporter="Enterprise Architect" exporterVersion="6.5" />
  - <uml:Model xmi:type="uml:Model" name="EA_Model" visibility="public">
    - <packagedElement xmi:type="uml:Package" xmi:id="EAPK_06175BD4_7CF4_45a3_84B1_66578C6DD528" name="Exemplo 1" visibility="public">
      + <packagedElement xmi:type="uml:Collaboration" xmi:id="EAID_CB000000_BD4_7CF4_45a3_84B1_66578C6DD52" name="EA_Collaboration1" visibility="public">
        <packagedElement xmi:type="uml:Class" xmi:id="EAID_3CF3F11B_AD39_498b_B843_82B32CA49DA9" name="DataPool" visibility="public" />
      + <packagedElement xmi:type="uml:Association" xmi:id="EAID_4E19C5B9_8B96_4746_A51C_FDB9CA58FBD7" visibility="public">
      + <packagedElement xmi:type="uml:Association" xmi:id="EAID_B234EBCD_BFA8_4ede_9572_637DCA10CC21" visibility="public">
      + <packagedElement xmi:type="uml:Association" xmi:id="EAID_9D9D6115_F148_419f_9AA4_C67CF4F7768D" visibility="public">
      - <packagedElement xmi:type="uml:Class" xmi:id="EAID_7900BD19_5713_4a9b_A721_D1480AC64163" name="HttpStatusCode" visibility="public">
        + <ownedAttribute xmi:type="uml:Property" xmi:id="EAID_4E9C091D_8506_487a_8186_B88B363D7F44" name="badRequest" visibility="private"
            isStatic="false" isReadOnly="false" isDerived="false" isOrdered="false" isUnique="true" isDerivedUnion="false">
        + <ownedAttribute xmi:type="uml:Property" xmi:id="EAID_02BB61A9_7AB8_4893_89D6_7D0EEF7AF1BD" name="created" visibility="private" isStatic="false"
            isReadOnly="false" isDerived="false" isOrdered="false" isUnique="true" isDerivedUnion="false">
        + <ownedAttribute xmi:type="uml:Property" xmi:id="EAID_A799FABF_7FE9_4d8a_8850_DC1D7BAE8073" name="internalServerError" visibility="private"
            isStatic="false" isReadOnly="false" isDerived="false" isOrdered="false" isUnique="true" isDerivedUnion="false">
        + <ownedAttribute xmi:type="uml:Property" xmi:id="EAID_6A66FCBE_D9F7_4bf5_A398_95F04591E657" name="notFound" visibility="private" isStatic="false"
            isReadOnly="false" isDerived="false" isOrdered="false" isUnique="true" isDerivedUnion="false">
        + <ownedAttribute xmi:type="uml:Property" xmi:id="EAID_D82ACDAC_5F9E_41a5_BA45_E2466CCBCBCF" name="ok" visibility="private" isStatic="false"
            isReadOnly="false" isDerived="false" isOrdered="false" isUnique="true" isDerivedUnion="false">
        </packagedElement>
```

Figura 5.18: Detalhe do documento XMI (com trechos colapsados).

### 5.5.4 Geração do Código dos Testes

Para possibilitar a geração dos *drivers* e dados dos testes a partir do documento XMI, foi desenvolvida uma folha de estilo XSL que, juntamente com o XMI, são entradas para um processador XSLT que gera código Test::Unit para linguagem Ruby. O algoritmo que a folha de estilo segue é o apresentado na sessão 5.4.2.2. Um trecho da folha de estilo XSL pode ser visto na Figura 5.19.

```
<?xml version="1.0" ?>
- <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform" xmlns:uml="http://schema.omg.org/spec/UML/2.1"
    xmlns:xmi="http://schema.omg.org/spec/XMI/2.1" xmlns:thecustomprofile="http://www.sparxsystems.com/profiles/thecustomprofile/1.0">
    <xsl:output method="text" omit-xml-declaration="yes" />
    <!--  Document Root  -->
  - <xsl:template match="/xmi:XMI/uml:Model">
      <xsl:text>require 'test/unit' require 'net/http'</xsl:text>
      <!--  Classes  -->
    - <xsl:for-each select="packagedElement/packagedElement[@xmi:type='uml:Class']">
        <!--  testa se tem atributos ou métodos  -->
      - <xsl:if test="count(ownedAttribute) + count(ownedOperation) != 0">
          <!--  descobre o estereótipo de cada classe  -->
        - <xsl:choose>
            <!--  DataPartition  -->
          - <xsl:when test="ancestor::uml:Model/thecustomprofile:DataPartition/@base_Class = @xmi:id">
              <!--  declaração da classe  -->
              <xsl:text>class</xsl:text>
              <xsl:value-of select="@name" />
              <xsl:text />
              <!--  declaração dos atributos  -->
            - <xsl:for-each select="ownedAttribute">
                <xsl:text>attr_accessor :</xsl:text>
                <xsl:value-of select="@name" />
```

Figura 5.19: Trecho da folha de estilo XSL utilizada para geração de código Ruby.

O processador XSLT utilizado foi Saxon 9.1.0.5 (Saxonica, 2010), pois é de fácil instalação e uso em linha de comando, além de também vir integrado em ferramentas de edição XML, como o Stylus Studio 2010 (Progress, 2010), utilizada neste trabalho para a edição da folha de estilo XSL.

É possível gerar o código de teste utilizando Saxon em linha de comando. A plataforma escolhida para execução do processador foi Java (.NET é a outra disponível), portanto, o comando para transformação do XMI para código Ruby seria:

```
java  -jar diretório_do_saxon/saxon9he.jar
-s:arquivo_xmi_fonte -xsl:folha_de_estilo_xsl
-o:nome_arquivo_saída.rb
```

É possível acrescentar outras opções de transformação, mas estas se mostraram suficientes.

O código de teste gerado para o modelo U2TP (Figura 5.14, Figura 5.15, Figura 5.16 e Figura 5.17) pode ser visto na Figura 5.20.

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

class HttpStatusCode
    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 ValidBookmark
    attr_accessor :id
    attr_accessor :descriptor
    attr_accessor :url

    def initialize ( params )
        @id = params [:id]
        @description = params [:description]
        @url = params [:url]
    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 << "</bookmark>"
    end
end

class InvalidBookmark
    attr_accessor :id
    attr_accessor :descriptor

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

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

class TestWebService < Test::Unit::TestCase

    Server = 'localhost'
    Port = 3000
    Path = ' '

    validBookmark = ValidBookmark.new(
        { :description => 'My valid bookmark', :url => 'www.test.com' }
    )

    invalidBookmark = InvalidBookmark.new(
        { :description => 'My invalid bookmark', :url => 'www.test.com' }
    )

    def test_create_valid_bookmark( validBookmark )
        req = post '/bookmarks.xml', validBookmark.to_xml
        assert_equal HttpStatusCode.created, req.code.to_i
    end

    def test_create_invalid_bookmark( invalidBookmark )
        req = post '/bookmarks.xml', invalidBookmark.to_xml
        assert_equal HttpStatusCode.bad_request, req.code.to_i
    end

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

    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
```

Figura 5.20: Resultado da geração de código de *drivers* e dados de teste pela folha de estilo XSL.

### 5.5.5 Execução dos Testes

Os testes são executados por linha de comando, executando o arquivo com o interpretador Ruby (Figura 5.21). A interface padrão ao se executar os testes dessa forma é a interface por linha de comando do Test::Unit.


Figura 5.21: Execução dos testes.

### 5.5.6 Avaliação dos Resultados

Foi apresentado acima um exemplo ilustrando a aplicação e a validade da solução proposta neste trabalho para gerar os testes de uma aplicação desenvolvida com objetivos didáticos. Neste exemplo foi demonstrado que REST-Unit+ é capaz de cumprir com o que se propõe: os *drivers* e dados de teste foram gerados automaticamente a partir do modelo dos testes em U2TP, exportado para formato XMI.

# 6 CONCLUSÃO

Neste trabalho foi apresentada REST-Unit+, uma extensão da solução REST-Unit (Borges, 2009), uma abordagem para geração automática de dados de teste a partir de modelos especificados em U2TP, visando a validação de comportamento de RESTful Web Services. Este capítulo apresenta as considerações finais deste trabalho, é apresentado um resumo dos resultados e contribuições, limitações do trabalho e propostas de trabalhos futuros.

## 6.1 Resultados

REST-Unit+ é uma solução para gerar os *drivers* e dados de teste a partir de um modelo U2TP. O modelo, quando exportado para XMI, permite que, como visto na sessão 5.4, gere-se o código de teste para Test::Unit. Todas as fases (especificação, exportação e geração do código de teste) de REST-Unit+ foram explicadas e exemplificadas.

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, principalmente, dos dados de teste (repositórios, partições, seletores e instâncias) utilizados para estes 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 no decorrer de um exemplo que demonstra a aplicação completa desta solução. Este protótipo pode gerar a partir de um modelo (Figura 5.14, Figura 5.15, Figura 5.16 e Figura 5.17) o *driver* e os dados para testes de um Web Service RESTful (Figura 5.20).

## 6.2 Contribuições

Como pôde ser visto no decorrer do trabalho, este satisfaz aos objetivos que foram propostos. As principais contribuições do trabalho são, portanto:

- Aumenta a produtividade do desenvolvimento de *RESTful Web Services*, diminuindo o esforço para desenvolvimento e execução dos testes;
- Através da geração de código de qualidade, contribui para o fortalecimento das boas práticas para o estilo arquitetural REST;

- Diminui o esforço para a implementação de *drivers*, *stubs* e dados de teste para *RESTful Web Services*, pois o código é gerado automaticamente;
- Contribui para a especificação e documentação dos dados aceitos por *RESTful Web Services*;
- Contribui para uma maior compreensão do uso do padrão U2TP de modelagem de testes.

## 6.3 Limitações do Trabalho

A proposta apresentada pelo presente trabalho tem foco em testar as boas práticas que emergem como consenso entre os arquitetos que utilizam do estilo arquitetural REST. Porém, houve uma dificuldade em modelar em U2TP conceitos relacionados aos cabeçalhos HTTP, por não haver um mapeamento intuitivo dessas entidades com os conceitos do U2TP.

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.

As principais limitações deste trabalho são resumidas a seguir:

- Não foi definido um modelo para os conceitos relacionados aos cabeçalhos HTTP pela dificuldade de mapeamento do conceito.
- Não foi definido um modelo para tratamento dos dados XML retornados pelo *Web service* pela dificuldade de mapeamento para UML.
- O XMI utilizado neste trabalho é dependente da ferramenta Enterprise Architect, pois a indefinições do padrão XMI causam diferença sensível entre os produtos existentes.
- Os dados dos *Web services* tratados são sempre XML, não foram considerados *Web services* que tratam dados JSON e YAML.

## 6.4 Trabalhos Futuros

São propostos os seguintes trabalhos futuros, com bases nas limitações e possíveis melhorias deste trabalho:

- Especificação de conceitos para permitir tratamento de cabeçalhos HTTP.
- 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*.
- Desenvolver uma modelagem que permita o uso de outros conceitos do U2TP, como conceitos de tempo, muitas vezes utilizados em acessos a *Web services*.

- Estender o presente trabalho para permitir tratamento de diversos tipos de dados e aumentar o domínio de *RESTful Web services* que podem ser testados.

# REFERÊNCIAS


ATOM, Atom Publishing Protocol. Disponível em: <http://tools.ietf.org/html/rfc5023>. Acesso em: Maio 2010.

BAI, X. et al. **Ontology-Based Test Modeling and Partition Testing of Web Services.** In *Proceedings of the 2008 IEEE international Conference on Web Services* (September 23 - 26, 2008). pp. 465-472. ICWS. IEEE Computer Society, Washington, DC.

BIASI, L. D. **Geração Automatizada de *Drivers* e *Stubs* de Teste para JUnit a partir de Especificações U2TP.** 2006. 153 p. Dissertação (Mestrado em Ciência da Computação) – Faculdade de Informática, PUCRS, Porto Alegre.

BORGES, F. Q. **REST-Unit: Geração baseada em U2TP de *Drivers* de Teste para *RESTful Web Services.*** 2009. 54 p. Projeto de Diplomação (Bacharelado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

CHANGEVISION, Change Vision Inc. JUDE Design & Modeling Tool. Disponível em: < http://jude.change-vision.com/jude-web/index.html>. Acesso em: Junho 2010.

CURL, Curl and libcurl. Disponível em: <http://curl.haxx.se/>. Acesso em: Maio 2010.

DAI Z. R. et al. **From Design to Test with UML Applied to a Roaming Algorithm for Bluetooth Devices.** TestCom 2004, LNCS 2978, pp. 33–49, 2004.

ETSI, ETSI World Class Standards. TTCN-3: Testing and Test Control Notation. Disponível em: <http://www.ttcn-3.org/>. Acesso em: Junho 2010.

FIELDING, R. T. **Architectural Styles and the Design of Network-based Software Architectures.** 2000. 180 p. Dissertação (Doutorado em Ciência da Computação e Informação) – University of California, Irvine.

FLICKR, Flickr. Disponível em: <http://www.flickr.com/services/api>. Acesso em: Maio 2010.

HE, H. **Implementing REST Web Services: Best Practices and Guidelines.** O'Reilly, 2004. Disponível em: <http://www.xml.com/pub/a/2004/08/11/rest.html>. Acesso em: Abril 2010.

IBM, International Business Machines Corp. IBM Rational Rose. Disponível em: < http://www-01.ibm.com/software/awdtools/developer/rose/>. Acesso em: Junho 2010.

JUNIT, JUnit. Disponível em: <www.junit.org>. Acesso em: Maio 2010.

NAYAK, A., SAMANTA D. **Automatic Test Data Synthesis using UML Sequence Diagrams.** 2009. 30 p. Journal of Object Technology 09, No. 2, March-April 2010.

ETH Zurich. Disponível em: <http://www.jot.fm/issues/issue_2010_03/article2.pdf>. Acesso em: Junho 2010.

NUNIT, NUnit. Disponível em: <www.nunit.org>. Acesso em: Maio 2010.

OMG. **UML Testing Profile Version 1.0.** Technical Report PTC/05-07-07, OMG, 2005. 113 p. Disponível em: <http://www.omg.org/cgi-bin/doc?formal/05-07-07>. Acesso em: Março 2010.

OMG. **Meta Object Facility (MOF) Specification**. Technical Report PTC/06-06-01, OMG, 2006. 90p. Disponível em: <http://www.omg.org/cgi-bin/doc?formal/2006-01-01>. Acesso em: Maio 2010.

OMG. **MOF 2.0/XMI Mapping, Version 2.1.1.** Technical Report Formal/07-12-01, OMG, 2007. 120 p. Disponível em: < http://www.omg.org/spec/XMI/2.1.1/PDF/index.htm>. Acesso em: Maio 2010.

OMG. Model Interchange Wiki. Disponível em: <http://www.omgwiki.org/model-interchange/doku.php>. Acesso em: Maio 2010.

PAUTASSO, C.; ZIMMERMANN, O.; LEYMANN, F. **RESTful Web Services vs. "Big" Web Services: Making the Right Architectural Decision.** WWW 2008, April 21–25, 2008, Beijing, China.

PEZZÈ, M. **Software testing and analysis: process, principles, and techniques.** 2008. 487 p. Hoboken, N.J.: John Wiley & Sons.

PROGRESS, Progress Software Corporation. Stylus Studio 2010 – XML Editor, XML Data Integration, XML Tools, Web Service and XQuery. Disponível em: <http://www.stylusstudio.com/>. Acesso em: Junho 2010.

RESTCLIENT, RESTClient. Disponível em: <http://code.google.com/p/rest-client/>. Acesso em: Maio 2010.

RICHARDSON, L.; RUBY S. RESTful Web Services. 1.ed. Beijing: O'Reilly, 2007. 446p.

RUBYDOC, Ruby-Doc.org: Documenting the Ruby Language**.** Disponível em: <http://www.ruby-doc.org>. Acesso em: Maio 2010.

SAXONICA, Saxonica Limited. Saxon, The XSLT and XQuery Processor. Disponível em: <http://saxon.sourceforge.net/>. Acesso em: Junho 2010.

SOMMERVILLE, I. **Engenharia de software**. 8. ed. São Paulo: Pearson Prentice Hall, c2007. xiv, 552 p. : il.

SPARX, Sparx Systems Pty Ltd. Enterprise Architect UML modeling tool. Disponível em: <http://www.sparxsystems.com/>. Acesso em: Junho 2010.

SUN, Sun Cloud API. Disponível em: <http://kenai.com/projects/suncloudapis/pages/Home>. Acesso em: Maio 2010.

SUN, Sun. NetBeans IDE. Disponível em: <http://netbeans.org/>. Acesso em: Maio 2010.

TITTEL, E. **Teoria e Problemas de XML.** Porto Alegre: Bookman. 207 p. Coleção Shaum, 2003.

TWILIO, Twilio Cloud Communications. Disponível em: <http://www.twilio.com/docs/api/2008-08-01/rest/>. Acesso em: Maio 2010.

TWITTER, Twitter API Wiki. Disponível em: <http://apiwiki.twitter.com/REST-API-Documentation>. Acesso em: Maio 2010.

W3C, W3C Recommendation. **XSL Transformations (XSLT) Version 1.0.** 1999. Disponível em: <http://www.w3.org/TR/xslt>. Acesso em: Junho 2010.

YAHOO, Yahoo. Disponível em: <http://developer.yahoo.com/java/howto-reqRestJava.html>. Acesso em: Maio 2010.

YWEBB, YWebb Consulting Ltd. HTTPe, REST HTTP Client plugin for Eclipse. Disponível em: <http://www.yWebb.com/eclipse-restful-http-client-plugin-http4e/>. Acesso em: Maio 2010.

ZANDER J. et al. **From U2TP Models to Executable Tests with TTCN-3 - An Approach to Model Driven Testing**. 2005. 14p. Lecture Notes in Computer Science 3502. pp. 289-303. Springer Berlin / Heidelberg.