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

TIAGO ROSA DA SILVA

PyRester: Uma abordagem baseada em modelos U2TP para geração de código de teste unitário para RESTful Web Services

Trabalho de Graduação.

Prof. Dr. Marcelo Pimenta
Orientador

Porto Alegre, novembro de 2011

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Tiago Rosa da Silva,

PyRester: Uma abordagem baseada em modelos U2TP para geração de código de teste unitário para RESTful Web Services /

Tiago Rosa da Silva. – Porto Alegre: Graduação em Ciência da Computação da UFRGS, 2011.

70 f.: il.

Trabalho de Conclusão (bacharelado) – Universidade Federal do Rio Grande do Sul. Curso de Ciência da Computação, Porto Alegre, BR–RS, 2011. Orientador: Marcelo Pimenta.

1. Teste baseado em modelos. 2. Teste dirigido por modelos. 3. Teste unitário. 4. Teste de software. 5. U2TP. 6. REST. 7. Python. I. Pimenta, Marcelo. II. Título.

AGRADECIMENTOS

Agradeço primeiramente aos meus pais pelo esforço e dedicação dispensados a mim não só durante a produção deste trabalho, mas em todos os momentos de minha vida. Agradeço a eles também pelos valores e princípios valiosos a mim transmitidos, que me ajudaram a chegar até aqui.

Agradeço também ao prof. Marcelo Pimenta pela orientação e apoio que me ofereceu não só neste trabalho mas ao longo de todo o curso de graduação, me ajudando a elaborar idéias e construir projetos que foram determinantes no meu crescimento como estudante e profissional.

Também agradeço aos meus demais amigos e familiares, em especial à minha namorada Kelly, pelo carinho e compreensão durante este projeto; e aos meus colegas de curso, em especial ao Alan, pela contribuição de idéias e pela ajuda na revisão deste trabalho.

Por fim, e acima de tudo, agradeço a Deus, a fonte de toda razão e lógica que, sendo a causa máxima da existência e a origem de toda a informação, ainda assim se revela pessoalmente a cada um de nós e nos oferece a possibilidade de compartilhar sua magnitude.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
CASE	Computer-Aided Software Engineering
HTML	Hypertext Markup Language
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
OMG	Object Management Group
REST	Representational State Transfer
SOAP	Simple Object Access Protocol
MIME	Multipurpose Internet Mail Extensions
OMG	Object Management Group
SUT	System Under Test
UML	Unified Modeling Language
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
U2TP	UML 2.0 Testing Profile
WS	Web Service
XMI	XML Metadata Interchange
XML	Extensible Markup Language

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Neste trabalho é proposto um método baseado em geração automática de código de testes a partir de modelos, que visa diminuir o esforço, o tempo e os custos necessários para a implementação de casos de teste unitário especificamente para *Web services* do tipo *RESTful* (ou seja, compatíveis com o estilo arquitetural REST). Isto possibilita a adoção de um processo de teste de unidade com mais facilidade e menos efeitos colaterais nesta classe de sistemas, aumentando a eficiência e produtividade de seu desenvolvimento e contribuindo para a implementação das boas práticas de *RESTful Web Services* através da escrita de código de maior qualidade e manutenibilidade.

Além das características principais da ferramenta que implementa este método, são apresentados conceitos relacionados ao perfil de testes da UML 2.0 (U2TP) e ao estilo arquitetural REST e sua utilização na construção de *RESTful Web Services*. Em seguida, é apresentada a proposta central deste trabalho, que diz respeito à geração automática de código de teste a partir de modelos. Um exemplo real de aplicação é então apresentado para a validação prática desta proposta.

Palavras-chave: Teste baseado em modelos, teste dirigido por modelos, teste unitário, teste de software, U2TP, REST, python.

PyRester: A U2TP model-based approach to generating unit testing code for RESTful Web Services

ABSTRACT

This paper proposes a method for automatic, model-based test code generation that aims to reducing time, cost and effort necessities in the process of developing unit testing code specifically for *RESTful Web Services* (i.e., services built according to the REST architectural style). This method enables an easier adoption of a unit testing process, reducing its undesired side effects, raising efficiency and productivity and helping the implementation of *RESTful Web Services* best practices through the production of better code, with higher quality and improved maintainability.

Beyond the main features of a software tool that implements this proposed method, concepts related to the UML Testing Profile (U2TP) and to the REST architectural style are presented. Following that, this paper's main proposal is described: a method for automatic, model-based test code generation. This proposal is then evaluated through its application to a real software system example.

Keywords: model-based testing, model-driven testing, software testing, unit testing, U2TP, REST, Python.

1 INTRODUÇÃO

Teste de software é uma atividade de investigação conduzida com o fim de obter informações acerca qualidade de um artefato de software. Com a evolução das disciplinas da Engenharia de Software e o advento de novas técnicas e processos, tais como o TDD (*Test-driven development*, ou desenvolvimento dirigido por testes), a atividade de teste tem ganhado visibilidade e relevância no ambiente profissional de TI, sendo hoje área de intensa pesquisa e forte desenvolvimento. A posição de *tester* (testador) está presente na maioria das equipes ágeis de desenvolvimento de software, e tem sua importância cada vez mais reconhecida e estabelecida no mercado de trabalho.

Uma das modalidades (ou *níveis*) de teste de software mais difundidas e praticadas é o **teste unitário** (ou **teste de unidade**). Esta atividade consiste na verificação da funcionalidade da menor parte testável de uma aplicação: a *unidade*. Testes unitários são tipicamente escritos pelos próprios desenvolvedores durante o processo de desenvolvimento, para garantir que cada função específica se comporta conforme o esperado. É neste nível de teste que o maior número de falhas é identificado e corrigido, possibilitando uma melhoria no nível geral de qualidade da base de código antes mesmo da aplicação de níveis mais abrangentes e interconectados de teste (como teste de **integração** ou de **sistema**). Além disso, o teste unitário facilita a refatoração do código, simplifica a integração, e também provê uma espécie de documentação funcional de um sistema, pois seus casos de teste descrevem o comportamento esperado das unidades do código e podem ser utilizados para entender seu funcionamento.

Entretanto, todos os benefícios obtidos através do teste de unidade tem um contraponto. Criar testes unitários abrangentes e relevantes é uma tarefa que exige esforço, demanda tempo e implica em custos para o processo de desenvolvimento. Segundo Sommerville (2010), o custo da implementação de testes pode chegar a 40% dos custos totais de um projeto de software. Em especial, o processo de teste de unidade, particularmente em metodologias de teste prévio como o TDD, requer a mobilização da equipe de desenvolvimento em torno da discussão de abstrações e da implementação dos casos de teste propriamente ditos, muitas vezes exigindo revisão de conceitos e mesmo uma boa dose de trabalho repetitivo ou tedioso na escrita do código de teste. Em muitos casos, a prática de teste unitário deixa de ser adotada por uma equipe justamente em função desta sobrecarga que acarreta no processo.

Uma das classes de aplicação que se beneficia largamente da utilização de teste de unidade são os *Web services*, ou serviços Web. Estes sistemas de software, que suportam a interoperação entre máquinas através da rede, possuem em sua estrutura a base da Web como a conhecemos hoje. Os principais usos de *Web services*, relacionados pri-

mariamente com integração de informações de sistemas corporativos e mesmo aplicações de inteligência de negócio, bem como seus aspectos inerentes de comunicação universal e largamente acessível, tornam os requisitos de qualidade para este tipo de sistema bastante relevantes. Por isso, as áreas de segurança, normalização e tratamento de erros são tópicos de intensa pesquisa e desenvolvimento nos dias de hoje. Neste contexto, os testes unitários de maneira específica são uma ferramenta poderosa para a obtenção de qualidade em *Web services*.

1.1 Objetivos

O presente trabalho tem por objetivo a apresentação de uma técnica que visa diminuir o esforço, o tempo e os custos necessários para a implementação de casos de teste unitário especificamente para *Web services* do tipo *RESTful* (ou seja, compatíveis com o estilo arquitetural REST), possibilitando a adoção do processo de teste de unidade com mais facilidade e menos efeitos colaterais nesta classe de sistemas, aumentando a eficiência e produtividade de seu desenvolvimento e contribuindo para a implementação das boas práticas de *RESTful Web Services* através da escrita de código de maior qualidade e manutenibilidade. Para isso, é proposto um método baseado em geração automática de código de testes a partir de modelos.

No trabalho de Biasi (2006) é apresentada uma proposta para a geração automatizada de *drivers* e *stubs* de teste para JUnit a partir de especificações U2TP (*UML 2.0 Testing Profile*, o perfil de testes da UML 2.0) (Biasi 2006). Com base nesta idéia, Borges (2009) apresenta uma proposta de geração de *drivers* de teste especificamente para *RESTful Web Services*. Esta proposta é estendida no trabalho de Feller (2010), que propõe um método para a geração de *drivers* e também de dados de teste para *RESTful Web Services*. As duas últimas propostas se baseiam na linguagem **Ruby** e seu *framework* de testes **Test::Unit**, e apresentam algumas restrições referentes a formato de representação de recursos e suportam apenas casos de teste bastante simples e nem sempre adequados para o teste de aplicações reais.

A proposta deste trabalho é também baseada na geração de código de teste a partir de modelos UML estendidos com aspectos de teste pelo U2TP, tendo agora como alvo a linguagem **Python**, através do *framework* de testes **PyUnit**. Nesta proposta, apresentada na forma de uma ferramenta denominada **PyRester**, conceitos existentes são estendidos e novos conceitos são introduzidos, incluindo a representação de recursos em múltiplos formatos (JSON ou XML) e o suporte a casos de teste complexos com múltiplas requisições HTTP e melhorada manipulação de resultados. Para a validação da proposta, é apresentado um exemplo de aplicação que demonstra a utilização do método de geração de código de testes em um sistema real, utilizado por usuários reais e que apresenta limitações e características próprias de uma aplicação independente.

1.2 Estrutura do trabalho

Este trabalho é organizado da seguinte forma: no capítulo 2 são apresentados os fundamentos e conceitos básicos para a compreensão da proposta, incluindo conceitos relacionados a teste de software e ao perfil de testes da UML 2.0 (U2TP), bem como conceitos relacionados ao estilo arquitetural REST e aos *RESTful Web Services*; no capítulo 3 são

elencados trabalhos relacionados à presente proposta, e uma comparação entre estes trabalhos em termos de suas características e funcionalidades é apresentada; no capítulo 4 é apresentada de forma conceitual a proposta central deste trabalho, que diz respeito à geração automática de código de teste a partir de modelos; o capítulo 5 descreve um exemplo de aplicação da proposta em um sistema real, de forma detalhada em etapas, e os resultados desta aplicação são considerados; por fim, no capítulo 6, são discutidas as conclusões da proposta, bem como suas limitações e possíveis melhorias futuras.

2 FUNDAMENTOS E CONCEITOS BÁSICOS - U2TP E RESTFUL WEB SERVICES

Este capítulo resume os fundamentos e conceitos básicos necessários para a compreensão do trabalho, incluindo aspectos gerais sobre teste de software e conceitos relacionados ao teste dirigido por modelos, uma apresentação básica do perfil UML de teste (U2TP), e ainda uma visão geral a respeito do estilo arquitetural REST e sua aplicação na construção de *RESTful Web Services*.

2.1 Teste de software

Software é um dos artefatos mais complexos e variáveis produzidos regularmente pelo homem. Por ser uma entidade conceitual e multifacetada, composta por elementos que incluem abstrações e representações de idéias e raciocínio humanos, e altamente diversificada em termos de forma e conteúdo, *software* possui uma grande complexidade conceitual associada ao seu funcionamento. Esta complexidade, amplificada pela inerente não-linearidade de sua construção, faz com que a *qualidade* dos sistemas - envolvendo qualidade estrutural e qualidade funcional - seja um aspecto fundamental no processo de desenvolvimento de *software*.

Desta forma, a *verificação* torna-se uma etapa fundamental para qualquer processo de desenvolvimento de *software* - e também uma atividade complexa e delicada. Requisitos de qualidade variam entre ambientes e domínios de aplicação, e sua estrutura evolui e até mesmo deteriora-se com o crescimento do sistema de *software* ao qual estão associados. O custo da verificação de *software* frequentemente ultrapassa a metade do custo geral de desenvolvimento e manutenção.

No entanto, ainda estamos longe de eliminar todas as inconsistências e produzir *software* livre de falhas. As abordagens modernas de desenvolvimento de *software*, tais como a orientação a objetos e a programação distribuída, trazem à tona novos desafios ao introduzirem conceitos e possibilidades diversificados que escapam ao alcance das técnicas tradicionais de obtenção de qualidade, instigando o desenvolvimento de novas técnicas e paradigmas de verificação que acompanhem estas mudanças. Assim, a variedade dos problemas e a riqueza de abordagens torna a escolha do conjunto de técnicas e processos para obtenção de qualidade dentro de um escopo de tempo e custo uma atividade desafiadora. Não existe uma solução universal para o problema da verificação da qualidade de um artefato se *software* - as soluções devem ser construídas de acordo com a complexidade, os requisitos, e o ambiente de desenvolvimento de cada caso (Pezze e Young 2007).

Teste de software é uma atividade essencial ao processo de desenvolvimento de software e, como tal, tem sido objeto de pesquisa e discussão intensas tanto no meio acadêmico

quanto no ambiente corporativo. Com o advento das correntes ágeis de desenvolvimento, novas técnicas e abordagens para teste de software tem sido propostas e avaliadas, o que tem ajudado a promover a importância e os benefícios do processo na obtenção da qualidade de software. Uma abordagem compreensiva a respeito do teste de software, incluindo descrições detalhadas e abrangentes sobre seus métodos, níveis, artefatos e processos, pode ser encontrada em Pezze e Young (2007).

2.2 Model-driven testing (Teste dirigido por modelos)

À medida que os sistemas de software tornam-se mais complexos, novos paradigmas se fazem necessários para sua construção. Um destes novos paradigmas é o **desenvolvimento dirigido por modelos** (*model-driven development*), que tem um impacto demonstrável na redução do *time to market* e no aumento da qualidade do produto. Este paradigma em particular prevê a introdução de modelos rigorosos durante o processo de desenvolvimento, permitindo a abstração e a automação. Para este objetivo, uma linguagem gráfica padronizada denominada **UML** (*Unified Modeling Language*, ou Linguagem de Modelagem Unificada) foi desenvolvida para a construção de sistemas de software. A UML permite que os requisitos de sistema e especificações de projeto sejam criadas e visualizadas de forma gráfica, o que auxilia a comunicação e a validação destes aspectos. Além disso, o uso da UML introduz a possibilidade do uso de técnicas automáticas de produção de software (Baker et al. 2010).

Entretanto, o desenvolvimento de sistemas de software de alta qualidade requer não somente processos sistemáticos de desenvolvimento, mas também processos sistemáticos de **teste**. É neste contexto que surge o **teste dirigido por modelos** (*model-driven testing*). Também conhecido como *model-based testing*, este processo consiste basicamente na derivação de parte ou do total dos casos de teste a partir de modelos que descrevem aspectos do sistema sob teste (SUT). Tais modelos fornecem ao processo informações a respeito da estrutura e do comportamento do SUT e permitem a geração de artefatos de teste funcional no mesmo nível de abstração dos modelos. Estes artefatos podem então ser utilizados para determinar a conformidade do SUT em relação a alguma propriedade desejada que é representada nos modelos, ou mesmo para identificar discrepâncias entre os modelos e o comportamento real do sistema (Baker et al. 2010) (Pezze e Young 2007).

2.3 U2TP - O perfil UML de testes

A abordagem de teste dirigido por modelos contemplada pelo presente trabalho propõe a derivação sistemática e automatizada de código de teste a partir de modelos UML. Para este objetivo, uma solução para a especificação de aspectos de teste necessários para a geração do código se faz necessária. A solução proposta surge na forma de uma extensão da UML que permite a definição destes aspectos - o **U2TP**.

O perfil UML de testes (U2TP - *UML 2.0 Testing Profile*) define uma linguagem para projeto, visualização, especificação, análise, construção e documentação de artefatos de sistemas de teste. O U2TP é uma linguagem de modelagem que pode ser utilizada com todas as principais tecnologias de objetos e componentes e aplicada para o teste de sistemas em diversos domínios de aplicação. U2TP pode ser utilizado por si só para o manuseio de artefatos de teste ou de maneira integrada com a UML para a manipulação de artefatos de sistema e de testes de forma conjunta (OMG 2005).

O U2TP estende a UML com conceitos específicos de teste, tais como componentes de teste, vereditos, padrões, etc. Estes aspectos são agrupados em conceitos para arquitetura de teste, comportamento de teste, dados de teste, e temporização de teste. Por ser um *perfil*, o U2TP integra-se de forma transparente à UML: U2TP é baseado no metamodelo da UML e reusa sua sintaxe (OMG 2005).

O U2TP é baseado na especificação da UML 2.0, e é definido pelo uso da abordagem de metamodelagem da UML, seguindo os princípios de **integração** com a UML (conformidade com os princípios da UML definidos na estrutura da versão 2.0) e **reuso e minimalidade** (preferência pelo uso direto de conceitos da UML quando possível, e pela extensão de conceitos existentes e eventualmente criação de novos conceitos coerentes e relevantes quando necessário) (OMG 2005).

A especificação oficial, completa e detalhada do U2TP pode ser encontrada no *website* do OMG (OMG 2005).

2.3.1 Arquitetura de teste

Arquitetura de teste é a definição de todos os conceitos estruturais necessários para a realização dos testes, em adição aos conceitos preexistentes da UML (Baker et al. 2010). Esta definição abrange tanto os componentes de teste quanto os aspectos do sistema sob teste (SUT).

Os principais elementos que compõem a arquitetura de teste do U2TP, exemplificados na Figura 2.1, são os seguintes (OMG 2005) (Baker et al. 2010):

- **Test Context** - Descreve o contexto dos testes, permitindo o agrupamento de casos de teste, a descrição da configuração de teste e a definição do controle dos testes, ou seja, da ordem de execução dos casos de teste.

- **Test Component** - Um componente de teste é uma classe de um sistema de teste. Os objetos do tipo componente de teste realizam o comportamento de um caso de teste ao se comunicarem com o SUT ou outros componentes através de uma série de interfaces.

- **SUT** - O *System Under Test* é o sistema, subsistema ou componente a ser testado. O SUT pode ser composto de vários objetos e é exercitado pelos *componentes de teste* através de suas operações e sinais com interface pública. O SUT é encarado pelos demais elementos de teste como uma *caixa-preta*.

- **Arbiter** - Uma propriedade de um *caso de teste* ou de um *contexto de teste* que serve para avaliar os resultados do teste e atribuir um *veredito* geral ao caso ou contexto de teste, respectivamente. Um algoritmo padrão de arbitragem é definido, baseado no teste funcional de conformidade, que gera *Pass* (sucesso), *Fail* (falha), *Inconc* (inconclusivo) e *Error* (erro) como vereditos, ordenados na forma *Pass* < *Inconc* < *Fail* < *Error*. Este algoritmo de arbitragem pode ser redefinido pelo usuário.

- **Scheduler** - Uma propriedade de um *contexto de teste* usada para controlar a execução de diferentes componentes de teste. O *Scheduler* gerencia a existência, a criação e a destruição de componentes de teste e sabe quais componentes participam de quais casos de teste.

- **Utility Part** - Uma parte do sistema representando componentes variados que auxiliam os *componentes de teste* na realização do seu *comportamento de teste*.

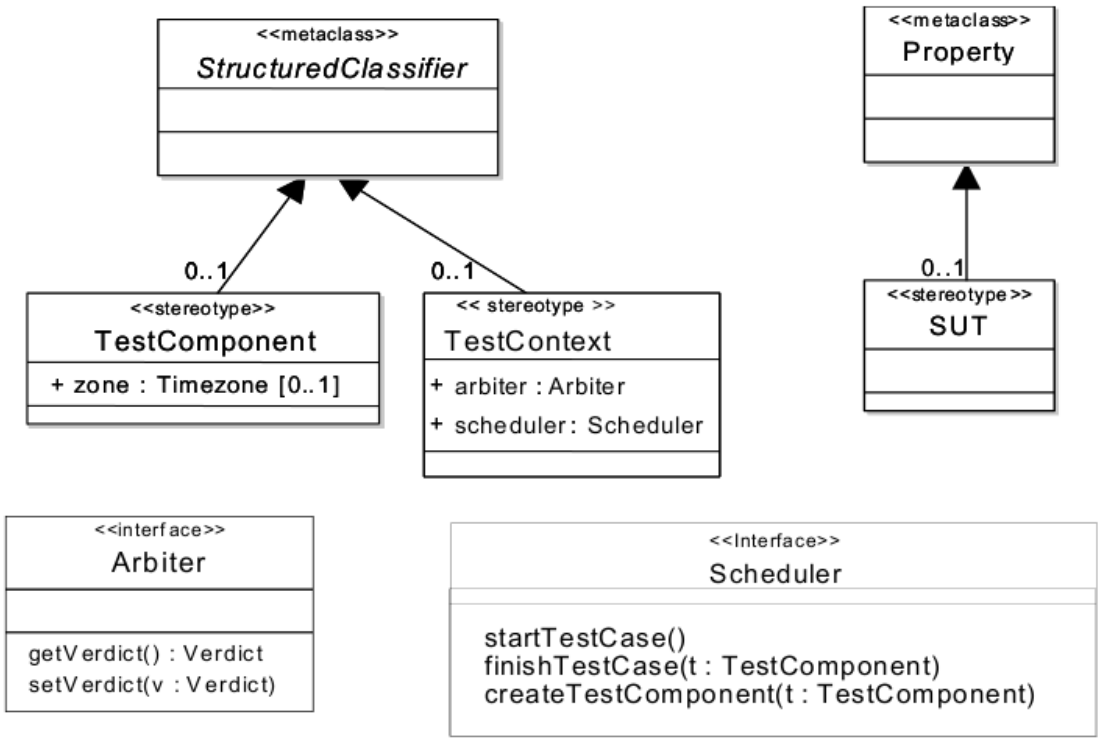

Figura 2.1: Elementos da arquitetura de teste do U2TP (OMG 2005).

2.3.2 Comportamento de teste

Comportamento de teste é o conjunto de conceitos que especificam o comportamento e o objetivo dos testes, especificando as ações e avaliações necessárias para que se possa abranger aquilo que deve ser testado (OMG 2005) (Baker et al. 2010).

Os principais elementos que compõem o conjunto de conceitos de comportamento de teste do U2TP são ilustrados nas Figuras 2.2 e 2.3 e descritos a seguir (OMG 2005) (Baker et al. 2010):

- **Test Control** - Um *controle de teste* é uma especificação da invocação de casos de teste dentro de um *contexto de teste*. Ou seja, é uma especificação técnica de como o SUT deve ser testado no dado contexto.

- **Test Case** - Um *caso de teste* é uma especificação de um caso através do qual o sistema deve ser testado, incluindo os objetos do teste, as entradas, os resultados e as condições. Ou seja, é uma especificação técnica completa de como o SUT deve ser testado no dado *objetivo de teste*.

  Um *caso de teste* é definido em termos de sequências, alternativas, laços, padrões de estímulo e observações vindos do SUT. Cada caso implementa um objetivo, e pode invocar outros casos de teste. Um *caso de teste* faz uso de um *árbitro* para avaliar os resultados de seu comportamento.

Casos de teste são propriedades de um *contexto de teste*, e podem ser entendidos como operações que especificam como um conjunto de *componentes de teste* cooperativos interagindo com um SUT realiza um *objetivo de teste*. Tanto o SUT quanto os diferentes *componentes de teste* fazem parte do *contexto de teste* ao qual o caso de teste pertence.

- **Test Invocation** - Um *caso de teste* pode ser invocado com parâmetros específicos e dentro de um contexto específico. A *invocação do teste* leva à execução do caso de teste.
- **Test Objective** - Um *objetivo de teste* é um elemento nomeado, associado a um *caso de teste*, que descreve *o que* deve ser testado.
- **Verdict** - Um *veredito* é uma avaliação da corretude do SUT. *Casos de teste* geram vereditos, que também podem ser utilizados para reportar falhas no sistema. Os valores predefinidos de veredito são os seguintes:
  - *Pass*: Indica que o *comportamento de teste* dá evidências para a corretude do SUT para um dado *caso de teste* específico;
  - *Fail*: Indica que o propósito do *caso de teste* foi violado;
  - *Inconclusive*: Usado em casos onde não se pode determinar entre *Pass* e *Fail*;
  - *Error*: Deve ser usado para indicar erros (ou exceções) no próprio processo de teste.

  Os *vereditos* de um *caso de teste* podem ser redefinidos pelo usuário e são calculados pelo *árbitro*.
- **Validation Action** - Uma ação para avaliar o estado da execução de um *caso de teste* através da avaliação das observações do SUT e/ou de parâmetros e características adicionais do SUT. Uma *ação de validação* é realizada por um *componente de teste* e define o veredito local deste componente.

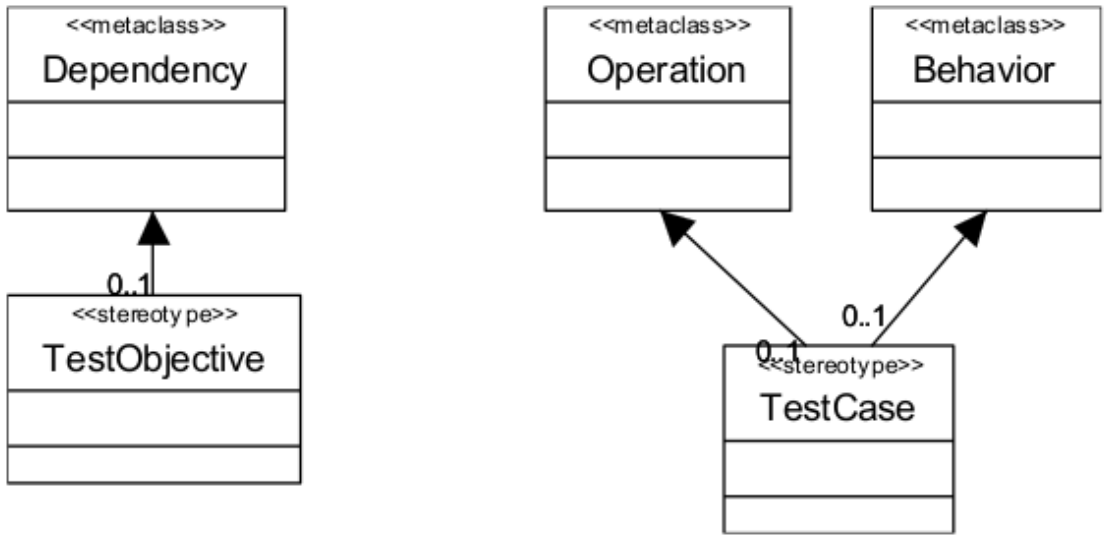

Figura 2.2: Elementos de comportamento de teste do U2TP - *Test Objective* e *Test Case* (OMG 2005).

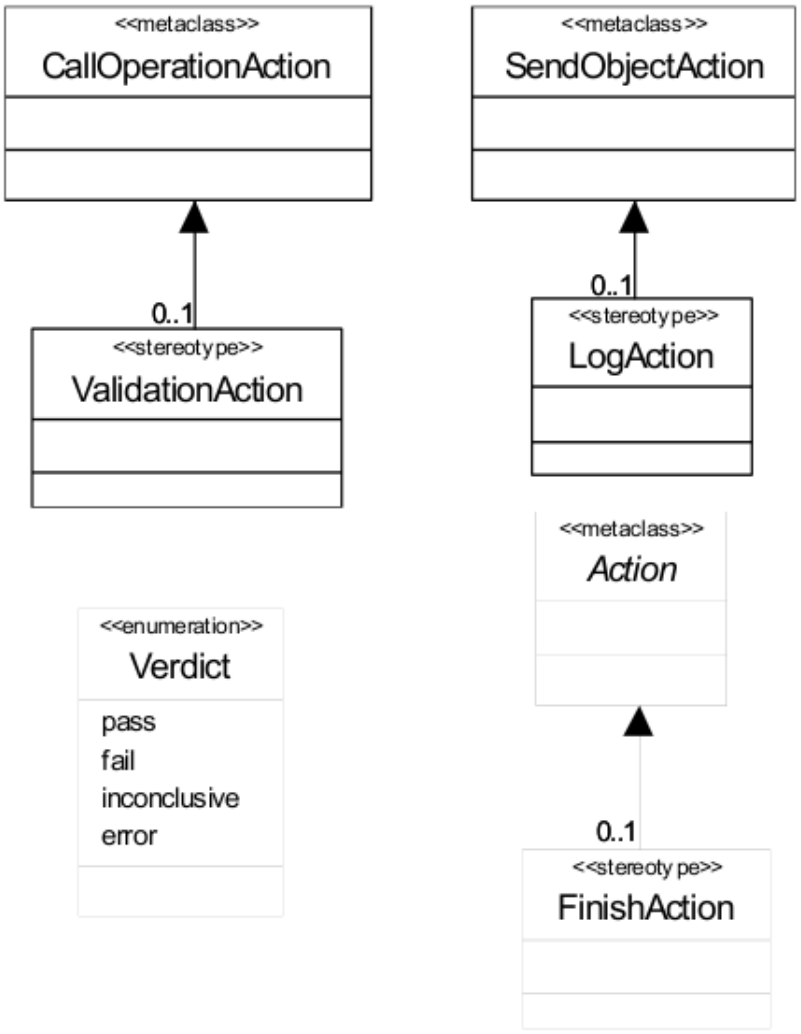

Figura 2.3: Elementos de comportamento de teste do U2TP - *Verdict* e *Validation Action* (OMG 2005).

2.3.3 Dados de teste

Dados de teste compreendem o conjunto de conceitos que especificam os dados utilizados no estímulo ao SUT, as observações do SUT e a coordenação entre *componentes de teste* (OMG 2005).

Os principais elementos que compõem o conjunto de conceitos de *dados de teste*, que podem ser visualizados na Figura 2.4, são os seguintes:

- **Data Pool** - Um *conjunto de dados* é uma coleção de *partições de dados* ou valores explícitos que são utilizados por um *contexto de teste*, ou por *componentes de teste*, durante a avaliação de *contextos de teste* e *casos de teste*. Ou seja, um *conjunto de dados* proporciona uma maneira de prover valores ou *partições de dados* para testes repetidos.

- **Data Partition** - Uma *partição de dados* é um valor lógico para um parâmetro usado em um estímulo ou em uma observação do SUT. Tipicamente, uma *partição de*

dados define uma classe de equivalência para um conjunto de valores (e.g., nomes de usuário válidos, endereços inválidos, etc.).

- **Data Selector** - Uma operação que define *como* valores de dados ou classes de equivalência são selecionados a partir de um *conjunto de dados* ou uma *partição de dados*.

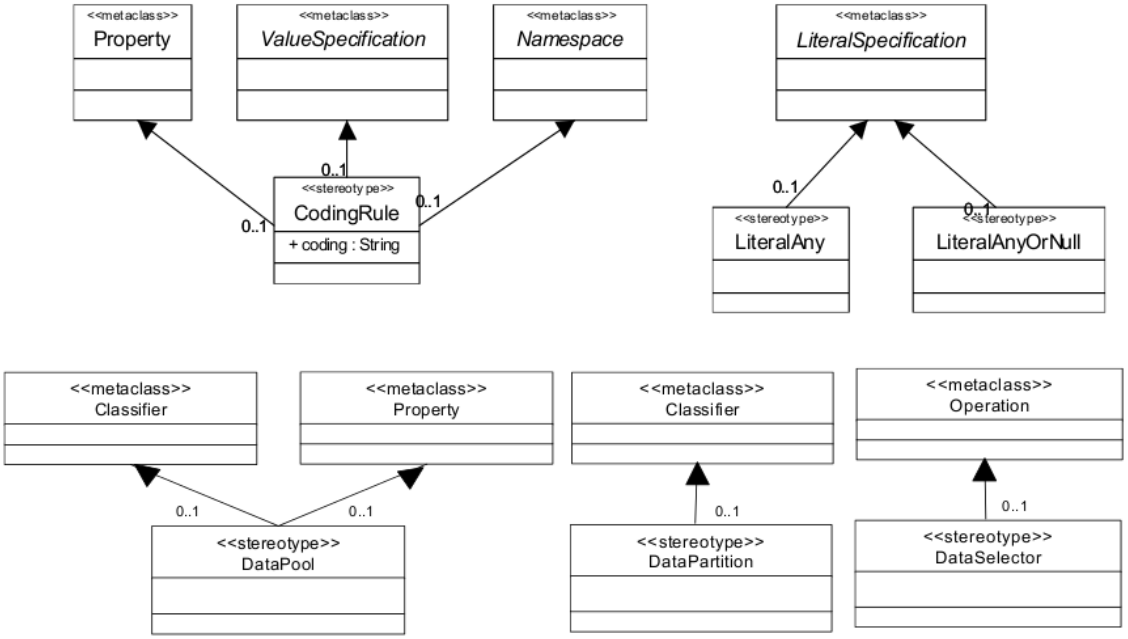

Figura 2.4: Elementos de dados de teste do U2TP (OMG 2005).

2.3.4 Temporização de teste

A *temporização de teste* do U2TP abrange conceitos relacionados à especificação de restrições de tempo, observações de tempo e/ou temporizadores dentro das especificações de *comportamento de teste*, a fim de que a execução dos testes seja temporalmente quantificada e/ou temporalmente observada (OMG 2005).

Os elementos de *temporização de teste* são ilustrados pela Figura 2.5 e descritos abaixo:

- **Timer** - *Temporizadores* são mecanismos que podem gerar um evento de *timeout* quando um valor de tempo especificado ocorre, tipicamente quando um intervalo de tempo preestabelecido termina, relativamente a um dado instante - normalmente o instante no qual o *temporizador* é iniciado. Os *temporizadores* pertencem aos *componentes de teste*, pois são definidos como propriedades destes elementos. Um *temporizador* pode ser parado, e seu estado atual de execução (e.g. ativo/inativo) pode ser checado.

- **Timezone** - *Zonas de tempo* são um mecanismo de agrupamento para *componentes de teste*. Cada componente pertence a uma certa *zona de tempo*, e componentes de uma mesma zona são sincronizados (i.e., tem o mesmo tempo).

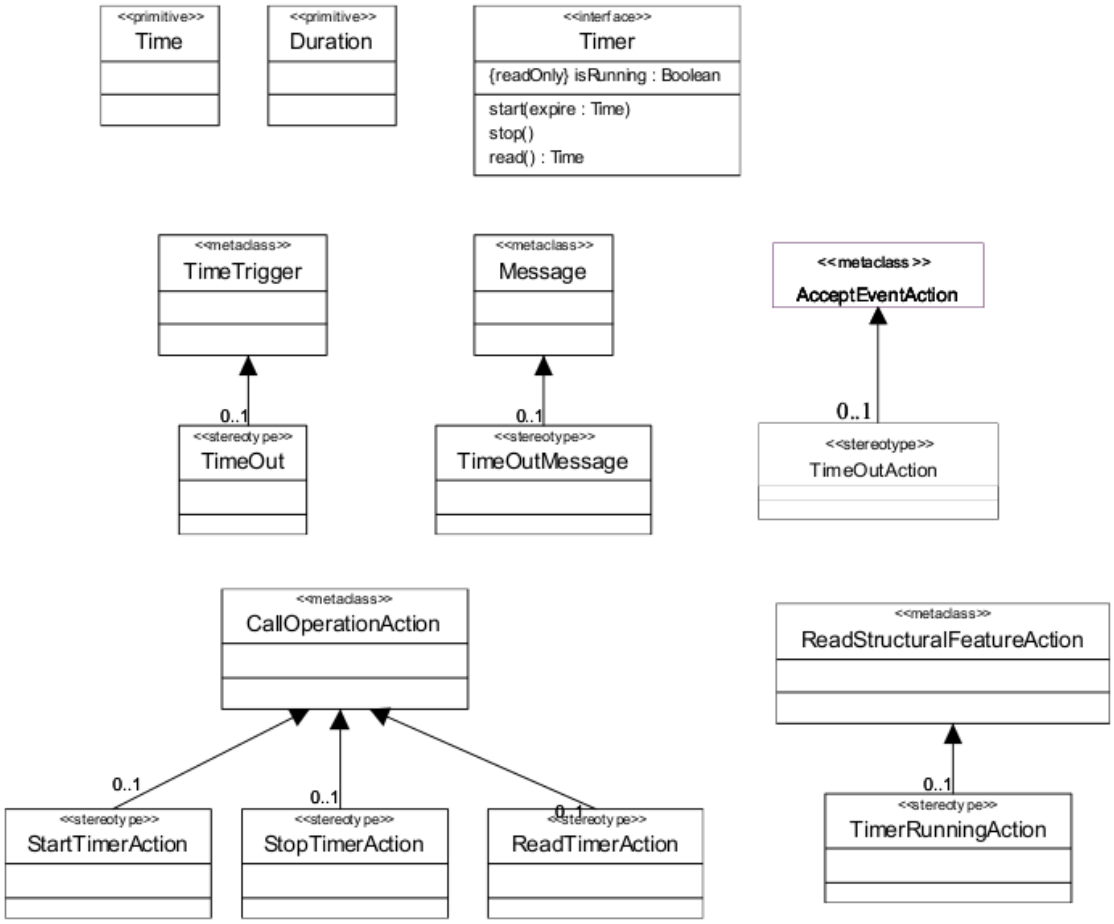

Figura 2.5: Elementos de temporização de teste do U2TP (OMG 2005).

2.4 RESTful Web Services

REST (REpresentational State Transfer) é um estilo arquitetural de software voltado para sistemas de hipermídia distribuídos, tais como a *World Wide Web*. Este estilo arquitetural, que foi proposto por Roy Fielding (2000) em sua dissertação de doutorado, tem sido largamente usado para guiar o *design* e o desenvolvimento de diversas aplicações da Web moderna. O estilo arquitetural REST, cuja estrutura pode ser visualizada na Figura 2.6, enfatiza a escalabilidade das interações entre componentes, a generalidade das interfaces, o desenvolvimento independente de componentes e a utilização de componentes intermediários para reduzir a latência das interações, enforçar a segurança e encapsular sistemas legados (Fielding 2000). As aplicações construídas nos moldes de REST são denominadas *RESTful*.

Um dos principais usos de REST é a criação de *Web services* (conhecidos como *RESTful Web Services* ou *RESTful Web APIs*) simples, construídos através do uso de padrões populares e bem estabelecidos como HTML, URI e XML. Esta prática surgiu como uma alternativa às opções de arquitetura de *Web services* baseadas em padrões complexos, que ignoram ou reinventam elementos e características que já tornaram a *World Wide Web* uma tecnologia de sucesso. Deste modo, os *RESTful Web Services* são propostos como um modelo simplificado que busca padronizar a forma como a Web é utilizada, tanto por humanos como por máquinas, pois é estabelecido que as características que tornam um

website fácil de usar também são as características que tornam a API de um *Web service* fácil de ser utilizada pelo programador (Richardson e Ruby 2007).

2.4.1 O estilo arquitetural REST

O estilo arquitetural REST pode ser entendido como uma generalização dos princípios arquiteturais sobre os quais a própria Web é construída. Ou seja, em linhas gerais, a arquitetura REST descreve como sistemas de informação distribuídos, como a Web, são construídos e se comportam (Webber e Parastatidis 2010). Arquiteturas no estilo REST são baseadas nas interações entre duas entidades principais: *clientes* e *servidores*; clientes iniciam requisições a servidores, enquanto servidores processam requisições e retornam respostas apropriadas - ou seja, a base do estilo arquitetural REST é uma caracterização das macro-interações entre os principais componentes da Web. Estas interações são modeladas e apresentadas na forma de um *framework* de restrições.

As restrições que compõem a arquitetura de sistemas REST foram desenvolvidas de forma incremental, através da análise das restrições de outros estilos arquiteturais baseados em rede, as quais cumprem o papel de enfatizar os principais objetivos e prerrogativas do estilo arquitetural. Estas restrições, bem como seus objetivos, são propostas e descritas por Fielding em seu trabalho (Fielding 2000):

- **Cliente-Servidor** - Ao separar os aspectos de interface do usuário dos aspectos de armazenamento de dados, aumentamos a portabilidade da interface através de várias plataformas e aprimoramos a escalabilidade ao simplificar os componentes do servidor. Esta separação de interesses permite que os componentes evoluam independentemente, uma característica que é especialmente significativa para a Web.

- **Comunicação *stateless*** - A comunicação entre cliente e servidor deve ser desprovida de *estado*. Isto é, cada requisição enviada ao servidor pelo cliente deve conter toda a informação necessária para se fazer entendida, e não pode tirar vantagem de algum contexto armazenado no servidor. Deste modo, o estado da sessão é mantido inteiramente no lado do cliente. Esta restrição é responsável pelas propriedades de *visibilidade* (um sistema monitorador não precisa olhar para além dos dados de uma requisição para determinar a natureza daquela requisição), *confiabilidade* (torna mais fácil a recuperação de falhas parciais) e *escalabilidade* (por não precisar armazenar e tratar informação de estado entre as requisições, o componente servidor pode mais rapidamente liberar seus recursos, além de ter sua própria implementação simplificada). Como a maioria das decisões arquiteturais, a opção pela comunicação *stateless* tem suas desvantagens, sendo a principal delas o decréscimo na performance da rede pelo aumento da quantidade de dados repetidos enviados em uma série de requisições, já que estes dados não podem ser mantidos no servidor em um contexto compartilhado.

- ***Cache*** - A fim de melhorar a performance da rede, restrições de *cache* são adicionadas ao cliente. Estas restrições requerem que os dados contidos em uma resposta do servidor a uma requisição sejam marcados como *cacheable* ou *non-cacheable*. Se uma resposta é *cacheable*, então o cliente pode reusar os dados desta resposta em futuras requisições equivalentes. A grande vantagem da adição de restrições de *cache* ao cliente é possibilitar potencialmente a elmininação, parcial ou completa, de algumas interações, melhorando eficiência, escalabilidade e a

performance percebida pelo usuário ao reduzir a latência média de uma série de interações. Uma desvantagem natural é a possível diminuição da confiabilidade, visto que um conjunto de dados armazenados na *cache* do cliente pode, em um determinado momento do tempo, ser significativamente diferente dos dados que o cliente obteria diretamente do servidor por meio de uma requisição direta.

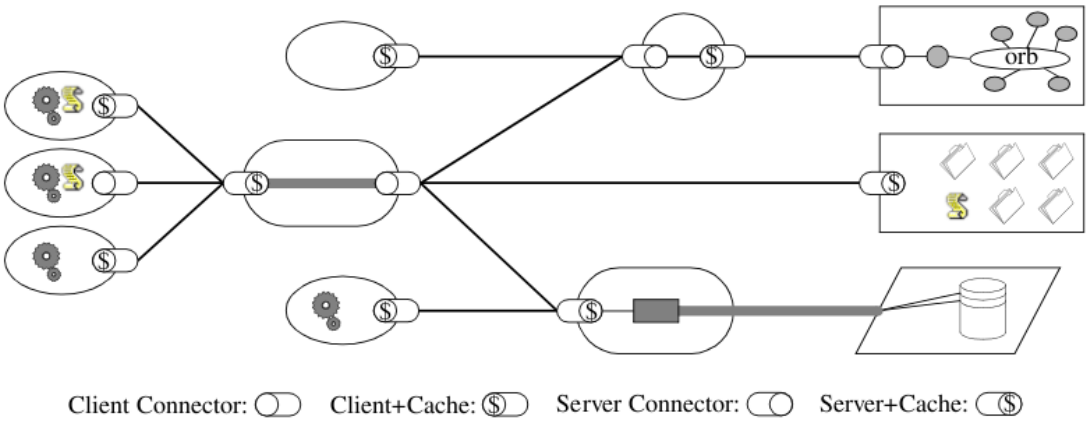

Figura 2.6: Visão estrutural do estilo arquitetural REST (Fielding 2000).

- **Interface uniforme** - A ênfase em uma interface uniforme entre componentes é a principal característica que difere REST dos demais estilos arquiteturais baseados em rede. Ao aplicar o princípio de Engenharia de Software da **generalidade** à interface dos componentes, a arquitetura do sistema em geral é simplificada e a visibilidade das interações é melhorada. Uma desvantagem, contudo, é que a uniformidade das interfaces diminui a eficiência, já que a informação é transferida em um formato padronizado ao invés de um formato específico que atende às necessidades de uma aplicação.

- **Sistema em camadas** - O estilo de sistema em camadas permite que uma arquitetura seja composta de camadas hierárquicas ao restringir o comportamento dos componentes de forma que cada componente não possa "enxergar"além do domínio da camada com a qual interage. Esta restrição limita a complexidade do sistema como um todo e promove a independência entre as camadas. As camadas também podem ser usadas para encapsular serviços legados e proteger novos serviços de clientes legados, simplificando componentes ao mover funcionalidade raramente usada para um intermediário compartilhado. Camadas intermediárias também podem ser usadas para melhorar a escalabilidade do sistema ao permitir o balanceamento de carga de serviços através de várias redes e processadores. A principal desvantagem da arquitetura em camadas é a adição de latência ao processamento dos dados, reduzindo a performance percebida pelo usuário, desvantagem essa que pode ser sanada pelo uso de *caches* compartilhadas entre os intermediários.

- **Código sob demanda** - REST permite que a funcionalidade do cliente seja estendida baixando e executando código na forma de *applets* ou scripts. Esta possibilidade simplifica a implementação dos clientes, que podem ter um menor número de funcionalidades pré-implementadas, e aumenta a extensibilidade do sistema, embora diminua sua visibilidade. Portanto, esta é uma restrição *opcional* na arquitetura REST.

REST é um estilo arquitetural híbrido, derivado da combinação de diversos outros estilos arquiteturais baseados em rede e restrições adicionais que definem uma interface uniforme de conexão. Como resultado, REST foi concebido como um estilo orientado a *recursos* para a abstração de elementos arquiteturais dentro de um sistema de hipermídia distribuído. REST ignora detalhes da implementação dos componentes e sintaxe de protocolo e concentra-se nos papéis desempenhados pelos componentes, nas restrições de sua interação com outros componentes e na sua interpretação de elementos de dados significativos. Deste modo, REST engloba as principais restrições em torno dos componentes, conectores e dados que definem a base da arquitetura da Web e, assim, definem também a essência de seu comportamento como uma aplicação baseada em rede (Fielding 2000).

Os elementos arquiteturais de REST são os seguintes (Fielding 2000):

- **Elementos de dados** - A principal abstração de dados em REST é o *recurso*. Um recurso é definido como um mapeamento conceitual para um conjunto de entidades. Usualmente, um recurso é algo que pode ser armazenado em um computador e representado como uma sequência de bits: um documento, uma linha em um banco de dados, ou o resultado da execução de um algoritmo. Contudo, um recurso também pode ser um objeto físico, como uma pessoa, ou mesmo um conceito abstrato. Ou seja, em REST um recurso é qualquer coisa suficientemente importante para ser referenciada por si própria (Richardson e Ruby 2007). Para nomear e referenciar um recurso, REST faz uso de *identificadores de recurso*. Para que as restrições da arquitetura se mantenham, é necessário que todo e qualquer recurso possua ao menos um identificador válido. Em implementações baseadas na Web, os identificadores são implementados pelas URIs e URLs que endereçam os recursos do sistema. Estes recursos podem ser apresentados em diferentes *representações*, que podem ser entendidas como formatos alternativos de encapsulamento dos dados. Por exemplo, um mesmo recurso (e.g., uma lista de nomes) pode ser recuperado como um documento XML, ou como uma página da web, ou mesmo como um arquivo de texto separado por vírgulas (Richardson e Ruby 2007).

- **Conectores** - *Conectores*, em REST, são elementos que encapsulam as atividades de acesso a recursos e transferência de representações de recurso. Os conectores apresentam uma interface abstrata para a comunicação de componentes, reforçando a simplicidade ao prover uma separação de interesses e esconder a implementação interna de recursos e mecanismos de comunicação. Os principais tipos de conectores são *cliente* e *servidor*; a diferença essencial entre estes é que, enquanto o *cliente* inicia a comunicação ao efetuar uma requisição, o *servidor* espera ativamente por conexões e responde a requisições para fornecer acesso aos seus serviços. Outro importante tipo de conector é a *cache*, que pode ser localizada na interface do *cliente* ou do *servidor* para salvar respostas marcadas como *cacheable* a fim de reusá-las posteriormente a fim de reduzir a latência das interações. Demais tipos de conectores incluem o *resolvedor* (de nomes de recursos) e o *túnel*, usado para transmitir a comunicação através de uma fronteira (por exemplo, um *firewall*).

- **Componentes** - Os *componentes* de REST são tipificados de acordo com seus papéis desempenhados na aplicação. Um exemplo comum de componente é o *agente de usuário*, que utiliza um conector *cliente* para iniciar uma requisição e torna-se também o receptor final da resposta esperada, papel que é comumente realizado

pelo *browser* ou navegador Web. Outros tipos de componentes incluem *servidores de origem*, *gateways* e *proxies*.

2.4.2 REST e Web services

Desde sua primeira definição formal, apresentada na dissertação de doutorado de Roy Fielding em 2000, REST vem sendo crescentemente adotado como uma alternativa mais simples para o desenvolvimento de Web services escaláveis, seguros e confiáveis. O crescimento na utilização e difusão de REST foi alavancado pelas diversas vantagens que REST apresenta sobre os principais padrões em uso no mercado, principalmente sobre o SOAP, à época utilizado por grandes *players* da indústria, tais como IBM e Microsoft - dentre elas, a simplicidade; a utilização de padrões conhecidos e bem estabelecidos (em detrimento da tentativa de introdução de novos e incipientes padrões); a maior escalabilidade; a melhor confiabilidade; e a maior segurança, visto que REST se baseia na infraestrutura de segurança já existente na Web (McMillan 2003).

Um *RESTful Web service* (representado na Figura 2.7) é um serviço Web simples, construído de acordo com os princípios arquiteturais de REST e utilizando padrões e protocolos estabelecidos e difundidos como HTTP, URI e XML e podendo ser definido como uma coleção de recursos com três aspectos definidos:

- Uma URI básica para o serviço (e.g., *http://servico.exemplo.com/recursos/*);
- Um tipo de mídia da Internet (MIME) para os dados suportados para o serviço (e.g., *XML*, *JSON*, *YAML*, etc.);
- Um conjunto de operações suportadas pelo Web service usando os quatro principais verbos que compõem os métodos HTTP (*GET*, *POST*, *PUT*, *DELETE*).

Não existe um padrão "oficial"para a criação de *RESTful Web services*. Diferentemente de SOAP, que é um *protocolo*, REST é um *estilo arquitetural*, ou seja, é baseado em um conjunto de restrições e elementos mas, ao invés de enforçar um modo ou procedimento específico para a criação dos serviços, apresenta um conjunto de boas práticas genéricas que são usadas para guiar o desenvolvimento. Alguns exemplos de boas práticas REST são descritos a seguir (Richardson e Ruby 2007):

- **Endereçabilidade** - Um serviço Web é *endereçável* se possui a capacidade de expor os aspectos interessantes de seu conjunto de dados através de *recursos*. Cada recurso deve ter seu próprio URI único, e um URI nunca deve representar mais de um recurso. O ideal é que cada URI identifique uma *representação* particular de um recurso.
- **Conectude** - O servidor deve poder guiar o cliente de um estado da aplicação para outro através do envio de *links* e formulários em suas representações. Ou seja, em um serviço *bem conectado*, o cliente pode caminhar pela aplicação seguindo *links* e preenchendo formulários.
- **Interface uniforme** - Todas as interações entre clientes e recursos devem ser mediadas através de alguns métodos HTTP básicos, que atuam de maneira análoga às operações representadas pelo acrônimo *CRUD*. Um recurso deve expor alguns ou todos estes métodos, e um método deve atuar da mesma maneira em qualquer recurso que o suporte. Os métodos HTTP básicos são:

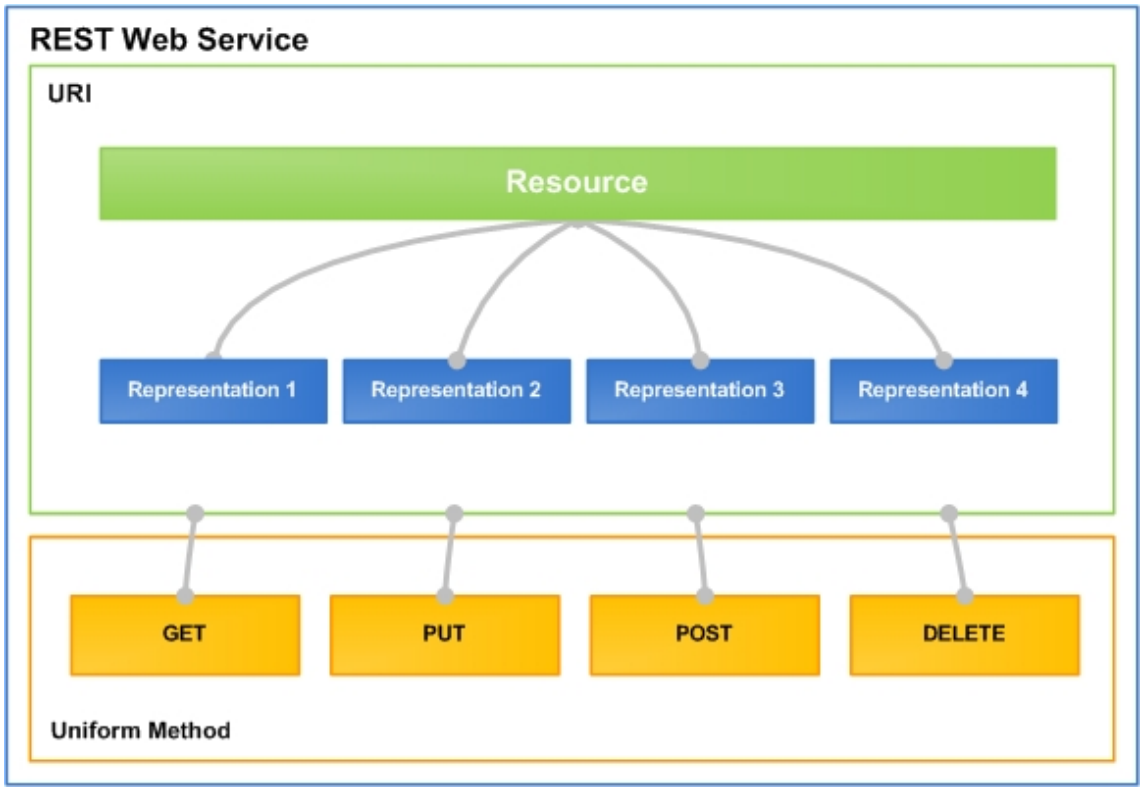

Figura 2.7: Diagrama de um RESTful Web Service (Apache Wink 2011).

  - *GET* - Uma requisição *GET* representa um pedido por informações a respeito de um recurso (análoga à operação *retrieve* do CRUD). A informação é entregue como um conjunto de cabeçalhos e uma representação. O cliente não deve enviar uma representação junto com uma requisição *GET*;
  - *POST* - Uma requisição *POST* é uma tentativa de criar um novo recurso a partir de um recurso existente (análoga à operação *create* do CRUD). A representação enviada junto com a requisição *POST* descreve o estado inicial do novo recurso;
  - *PUT* - Uma requisição *PUT* é uma asserção sobre o estado de um recurso (análoga à operação *update* do CRUD). Normalmente o cliente envia uma representação junto com a requisição, e o servidor tenta criar ou atualizar o recurso para que seu estado corresponda ao estado descrito na representação. Uma requisição *PUT* sem uma representação é simplesmente uma asserção de que um recurso deve existir através de um determinado URI;
  - *DELETE* - Uma requisição *DELETE* é uma asserção de que um recurso não deve mais existir (análoga à operação *delete* do CRUD). O cliente nunca deve enviar uma representação junto a uma requisição *delete*.

- **Responsividade através de códigos de status HTTP** - Todas as operações sobre recursos devem retornar um *código de status HTTP* para representar seu resultado. Na Tabela 2.1 podem ser visualizados alguns dos códigos de status mais utilizados e seus respectivos significados.

- **Uso adequado dos cabeçalhos HTTP** - Os cabeçalhos HTTP devem ser usados para identificar os metadados relacionados às requisições. Uma requisição que es-

Tabela 2.1: Códigos de status HTTP mais utilizados em REST

Código	Nome	Descrição
200	*OK*	Operação efetuada com sucesso.
201	*Created*	A requisição foi completada e um novo recurso foi criado.
301	*Moved Permanently*	Ação disparada mudou o URI de um recurso.
400	*Bad Request*	Houve um problema no lado do cliente.
404	*Not Found*	URI solicitado pelo cliente não identifica um recurso.
409	*Conflict*	Operação deixaria recurso em estado inconsistente.
410	*Gone*	Recurso solicitado não está mais disponível.
500	*Internal Server Error*	Houve um problema no lado do servidor.

pera por uma representação de recurso no formato JSON deve conter um cabeçalho do tipo *Accept:Application/JSON*.

3 TRABALHOS RELACIONADOS

Nesta seção são mencionadas algumas ferramentas de propósito relacionado à proposta deste trabalho. As características principais destas ferramentas são apresentadas e em seguida comparadas entre si e em relação às características principais da proposta.

3.1 cURL

cURL (cURL 2011) é uma ferramenta de linha de comando para transferência de dados com sintaxe URL através de diversos protocolos (incluindo HTTP e HTTPS). cURL suporta autenticação, *proxies*, *cookies*, tunelamento e diversas outras utilidades.

```
tiago@clapton $ curl http://127.0.0.1:8000/ForCA/api/courses/232.json
{"code": "GET74813", "name": "Gettable Course", "short_name": "BLA", "resume": n
ull, "grade": null, "id": 232}
```

Figura 3.1: Exemplo de uso de cURL - requisição GET de recurso representado em JSON

cURL é uma ferramenta bastante difundida e estabelecida, que vem sendo desenvolvida desde 1997 e aplicada nos mais diversos contextos, inclusive no teste de *RESTful Web Services*. Com cURL, é possível realizar as requisições HTTP necessárias para o teste destas aplicações de maneira simples e compatível com grande parte dos mecanismos de teste disponíveis no mercado. Um exemplo de uso do cURL pode ser visualizado na Figura 3.1.

3.2 REST Console

REST Console (REST Console 2011) é um cliente HTTP e visualizador de requisições destinado a auxiliar a construção, desenvolvimento e teste de APIs *RESTful*.

Esta ferramenta é apresentada como uma extensão para o navegador *Google Chrome*, e suas principais características são a facilidade de uso e a diversidade de opções para a criação das requisições. Outras características importantes são: possibilidade de customização dos cabeçalhos HTTP; suporte a autenticação *OAuth*; interface customizável e modular; possibilidade de fazer *upload* de arquivos via interface.

REST Console (ilustrado na Figura 3.2) se relaciona a este trabalho por ser uma ferramenta também destinada ao teste de *RESTFul Web Services*, que atualmente é bastante utilizada pelos usuários do navegador supracitado para o desenvolvimento de aplicações deste tipo.

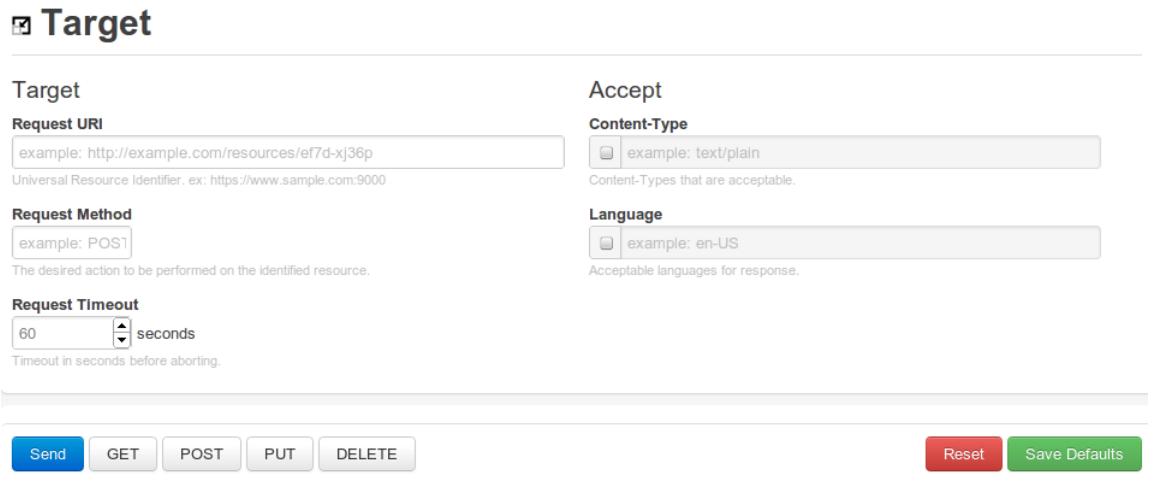

Figura 3.2: Captura de tela do REST Console

3.3 Python Rest Client

Python REST Client (Python REST Client 2011) é uma *library* que implementa um cliente REST com base nas bibliotecas *httplib2* e *urllib2* da linguagem Python.

```
from restful_lib import Connection

# Should also work with https protocols
base_url = "http://ora.ouls.ox.ac.uk:8080/fedora"
conn = Connection(base_url, username="fedoraAdmin", password="blahblah")

# A DELETE request on http://example.org/items/11232344
conn = Connection("http://example.org", username="XXX", password="XXX")
conn.request_delete('/items/11232344')

# A GET request to /search, with param q = "Test", header "Accept: text/json"
conn.request_get("/search", args={'q':'Test'}, headers={'Accept':'text/json'})
```

Figura 3.3: Exemplo de código utilizando Python REST Client

Esta biblioteca apresenta uma solução a nível de código para o desenvolvimento de testes unitários para *RESTFul Web Services*, através da utilização da classe *connection* que é disponibilizada e implementa funcionalidades de execução de requisições HTTP parametrizáveis ao estilo REST, cujas respostas podem ser analisadas para fins de validação. Um exemplo de código pode ser visto na Figura 3.3.

Python REST Client se relaciona a este trabalho por ser uma solução na linguagem Python para auxiliar a construção de testes para APIs *RESTful*.

3.4 REST-Unit

REST-Unit (Borges 2009) é uma proposta de solução para a geração automática de código de teste unitário para *RESTful Web Services* a partir de modelos U2TP. Neste trabalho, o autor propõe um mecanismo para a geração de código para o *framework* de teste unitário *Test::Unit*, considerado o *framework* padrão para testes unitários na linguagem Ruby (Borges et al. 2009).

Tabela 3.1: Comparação das características dos trabalhos relacionados ao **PyRester**

Característica/Ferramenta	cURL	REST Console	Python REST Client	REST-Unit	REST-Unit+
Suporte a requisições GET	**sim**	**sim**	**sim**	**sim**	**sim**
Suporte a requisições POST	**sim**	**sim**	**sim**	**sim**	**sim**
Suporte a requisições PUT	**sim**	**sim**	**sim**	não	não
Suporte a requisições DELETE	**sim**	**sim**	**sim**	não	não
Gerenciamento de dados de teste	não	não	não	não	**sim**
Interface programática	**sim**	não	**sim**	**sim**	**sim**
Geração de código de teste	não	não	não	**sim**	**sim**
Adequação a *frameworks* de teste	não	não	não	**sim**	**sim**
Suporte a *payload* XML	**sim**	**sim**	**sim**	**sim**	**sim**
Suporte a *payload* JSON	**sim**	**sim**	**sim**	não	não
Casos de teste com múltiplas *requests*	não	não	**sim**	não	não

As etapas do processo de geração de código de acordo com a proposta de **REST-Unit** são: modelagem dos aspectos de teste da aplicação-alvo em UML, seguindo as definições do U2TP; exportação dos modelos U2TP especificados para o formato de representação XMI; e geração automática dos *drivers* de teste a partir da representação XMI, através de um *parser* que identifica os elementos da modelagem e os classifica de acordo com seu estereótipo e outras características.

REST-Unit se relaciona a **PyRester** por apresentar uma solução para a geração automática de código de testes unitários a partir dos modelos da aplicação, utilizando o perfil UML de testes (U2TP).

3.5 Rest-Unit+

REST-Unit+ (Feller 2010) é uma proposta de extensão de **REST-Unit**, que visa incluir aspectos não contemplados na proposta original, tais como o tratamento explícito dos dados e partições de dados de teste. Com esta extensão, os testes gerados são mais abrangentes e completos e sua execução é facilitada.

REST-Unit+ se relaciona ao presente trabalho também por apresentar uma solução para a geração automática de código de testes unitários a partir de modelos, incluindo definições referentes a dados e partições de dados de teste.

4 PYRESTER - GERAÇÃO DE CÓDIGO DE TESTE UNITÁRIO A PARTIR DE MODELOS U2TP

A seguir, é apresentada de forma conceitual a proposta central deste trabalho, referente à metodologia do processo de geração automática de código de teste a partir de modelos.

4.1 Visão geral

As disciplinas de engenharia combinam atividades de *design* e construção com atividades de checagem e validação de produtos e resultados, a fim de que defeitos possam ser identificados e removidos. Analogamente, na Engenharia de Software, a construção de artefatos de software de alta qualidade requer que as atividades de *design* e verificação sejam realizadas de maneira complementar e conjunta durante o processo de desenvolvimento (Pezze e Young 2007).

Sabe-se que artefatos de software, de qualquer natureza, devem ser testados para que seja garantido seu bom funcionamento, de acordo com seus objetivos, no ambiente para o qual foram desenvolvidos. Esta necessidade é, atualmente, um consenso entre profissionais e acadêmicos da área. Enquanto é necessário que o processo de teste de software seja **eficaz** na identificação de defeitos, o que é seu objetivo principal, também espera-se que seja **eficiente** nesta tarefa, realizando-a da maneira mais rápida e menos custosa possível (Fewster e Graham 1999).

Uma das mais conhecidas e difundidas abordagens para o aumento da eficiência no desenvolvimento de software é a geração automática de código. Diversas ferramentas CASE, IDEs e ferramentas de modelagem UML implementam esta técnica e geram código em diversas linguagens a partir de artefatos de mais alto nível de abstração. A geração de código permite que a produtividade do processo de desenvolvimento seja aumentada, ao mesmo tempo em que diminui a incidência de falhas de código, pois possibilita que tarefas de codificação repetitivas, extensas e desgastantes sejam realizadas de maneira rápida e precisa, poupando tempo e custo de desenvolvimento e diminuindo a sobrecarga cognitiva que diversos elementos secundários e não-essenciais introduzem na leitura e escrita do código.

Uma das tarefas de codificação mais repetitivas e onerosas é a criação de código de teste. Portanto, dentro do âmbito de teste de software, a geração automática de código é uma prática bastante adotada e conveniente. Várias abordagens de teste de software inclusive propõem a geração automática de código como parte essencial de seu processo. No entanto, principalmente nos sistemas atuais, que se caracterizam pela utilização de diversas tecnologias e camadas de funcionalidade, nem sempre é fácil encontrar mecanismos viáveis para a geração de código de teste a partir de artefatos de nível mais alto de

abstração. Com isso em vista, este trabalho se propõe a apresentar uma solução para a geração automática de código de teste com base na modelagem de uma aplicação desenvolvida na forma de um *RESTful Web Service*.

4.1.1 A proposta

Tendo em vista as dificuldades estabelecidas pelo processo de teste de software, apresentamos aqui uma proposta, na forma de uma ferramenta, que tem por objetivo minimizar as dificuldades deste processo dentro de uma classe ou subconjunto particular de aplicação - nesta abordagem, aplicações desenvolvidas na forma de *RESTful Web Services*. Esta proposta é implementada por meio do **PyRester**, uma aplicação que aborda a geração de código de artefatos de teste unitário para o framework *PyUnit*, da linguagem Python, a partir de modelos construídos nos moldes do U2TP.

4.2 Modelagem dos aspectos de teste em U2TP

O primeiro passo no processo de geração de código de testes unitários através do **PyRester** é a modelagem dos aspectos de teste da aplicação alvo (doravante denominada *SUT*). Estes aspectos são modelados nos padrões do perfil de testes da UML (U2TP), através do mapeamento dos artefatos que compõem o mecanismo de teste da aplicação em elementos UML que representam sua estrutura e seu comportamento.

4.2.1 Modelagem estrutural

A modelagem estrutural compreende a representação dos artefatos que descrevem as definições que serão utilizadas no processo de teste. Nesta etapa são modelados a aplicação alvo, o conjunto de dados e partições de teste, o contexto de testes abrangendo a listagem dos casos de teste a serem realizados, e ainda algumas definições auxiliares na comunicação com o serviço Web que será testado. Os elementos da modelagem estrutural são representados em um diagrama de classes UML, e sua classificação é dada pela aplicação dos estereótipos previstos no U2TP:

- **SUT** - O sistema a ser testado é modelado através de uma classe com o estereótipo *«SUT»*. Os atributos desta classe servem para identificar o acesso ao serviço Web, informando ao gerador de código *server* (servidor onde está localizada a aplicação), *path* (caminho para a aplicação no servidor) e *port* (porta de acesso à aplicação no servidor). Na Figura 4.1 é ilustrada a modelagem do SUT.

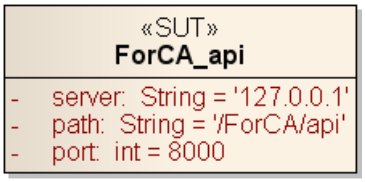

Figura 4.1: Exemplo de modelagem estrutural de SUT

- **TestComponent** - Em classes com o estereótipo *«TestComponent»* são representadas definições auxiliares para a geração dos testes e comunicação com o serviço Web a ser testado. Nesta abordagem, é necessária a enumeração dos códigos de status HTTP que podem ser retornados pela aplicação durante o processo de teste.

Estes códigos devem estar listados em uma classe denominada *HttpStatusCode*, onde cada código de status é um atributo do tipo *int* com estereótipo *«enum»*, cujo nome é a descrição do status e cujo valor é o próprio código do status. Na Figura 4.2 é ilustrada a modelagem do TestComponent.

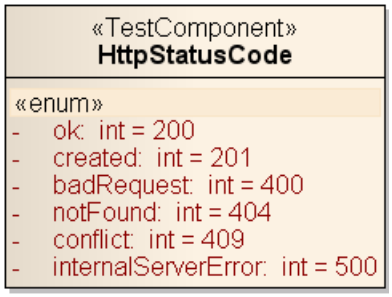

Figura 4.2: Exemplo de modelagem estrutural de TestComponent

- **TestContext** - Uma (e somente uma) classe com o estereótipo *«TestContext»* deve existir no modelo, contendo uma ou mais operações com estereótipo *«TestCase»*, que representam as chamadas dos métodos de teste que serão gerados. Na Figura 4.3 é ilustrada a modelagem do TestContext.

- **TestCase** - Os casos de teste que serão gerados são representados como operações da classe de estereótipo *«TestContext»*. Cada uma destas operações deve ser estereotipada como *«TestCase»* e retornar um valor nulo (*void*).

  O nome de cada um dos métodos é importante: todos os casos de teste devem começar com *test_<req>*, onde *<req>* define a operação a ser testada. Os valores possíveis para *<req>* são os seguintes:

  - *get_all* - Testa uma operação GET sem parâmetros ou argumentos, que recupera todos os recursos da coleção.
  - *get* - Testa uma operação GET sobre um recurso específico. Deve fornecer como parâmetro um objeto representando um recurso, que deverá ser primeiramente criado no SUT (operação POST) e então recuperado (operação GET).
  - *post* - Testa uma operação POST, que cria um novo recurso. Deve fornecer como parâmetro um obejto representando um recurso.
  - *put* - Testa uma operação PUT, que atualiza um recurso existente. Deve fornecer como parâmetro um objeto representando um recurso, que deverá ser primeiramente criado no SUT (operação POST) e então atualizado (operação PUT).
  - *delete* - Testa uma operação DELETE, que remove um recurso existente. Deve fornecer como parâmetro um objeto representando um recurso, que deverá ser primeiramente criado no SUT (operação POST) e então removido (operação DELETE).

- **DataPool** - Uma classe estereotipada como *«DataPool»*, associada a um *TestComponent* ou um *TestContext*, pode conter um conjunto de valores ou partições de dados relacionados a casos de teste. Cada *DataPool* pode ser referenciado por apenas

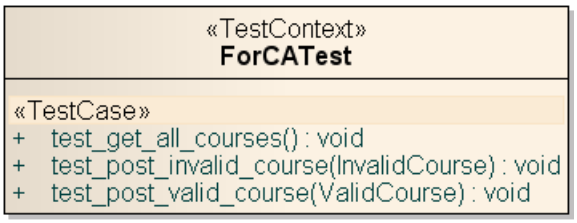

Figura 4.3: Exemplo de modelagem estrutural de TestContext e operações TestCase

um *TestComponent* ou um *TestContext*, e cada *TestComponent* ou *TestContext* pode ter zero, um ou vários *DataPool* associados. Ao *DataPool* podem ser associados elementos do tipo *DataPartition*, designando as partições de dados especificadas.

- **DataPartition** - Uma partição de dados de teste é representada por uma classe estereotipada como *«DataPartition»*. Um *DataPartition* deve estar associado a um *DataPool* e contém um conjunto de dados de teste que será utilizado na geração dos casos de teste. Um *DataPartition* pode estar associado a um único *DataPool* ou outro *DataPartition*, mas um *DataPool* pode estar associado a zero, um ou vários *DataPartition*.

  Os dados de teste da partição de dados são representados através de instâncias das classes *DataPartition*, também marcadas com este estereótipo, modeladas na forma de elementos do tipo *InstanceSpecification*, presentes na UML 2.0. Estas instâncias já possuem os valores dos atributos que serão usados como parâmetros na execução dos casos de teste especificados na classe *TestContext*.

  Na Figura 4.4 é ilustrada a modelagem das classes DataPool e DataPartition.

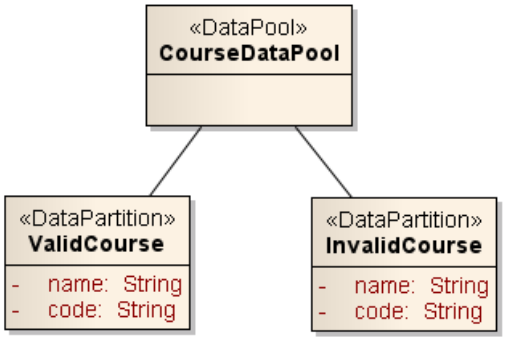

Figura 4.4: Exemplo de modelagem estrutural de DataPool e DataPartition, com operações DataSelector implícitas

- **DataSelector** - Seletores de dados, modelados como operações estereotipadas como *«DataSelector»*, são elementos que definem diferentes estratégias de seleção para os conjuntos de dados definidos em classes do tipo *DataPool* e *DataPartition*. Estas duas classes podem ter zero, uma ou várias operações do tipo *DataSelector* em sua definição. Nesta abordagem, os métodos *get* e *set* gerados para a manipulação dos atributos especificados nas classes *DataPartition* cumprem o papel de seletores de dados, apesar de estarem definidos de forma implícita.

4.2.2 Modelagem comportamental

Na modelagem comportamental são especificados aspectos referentes ao procedimento de testes, a fim de definir os objetivos e o comportamento de cada caso de teste. Para esta etapa da modelagem são utilizados diagramas de sequência UML, também classificados de acordo com os estereótipos preestabelecidos. Nesta abordagem, o elemento principal da modelagem comportamental é o *TestCase*, representado na modelagem estrutural como um método da classe *«TestContext»*, cujo comportamento é modelado nesta etapa. A modelagem comportamental de um caso de teste é ilustrada na Figura 4.5.

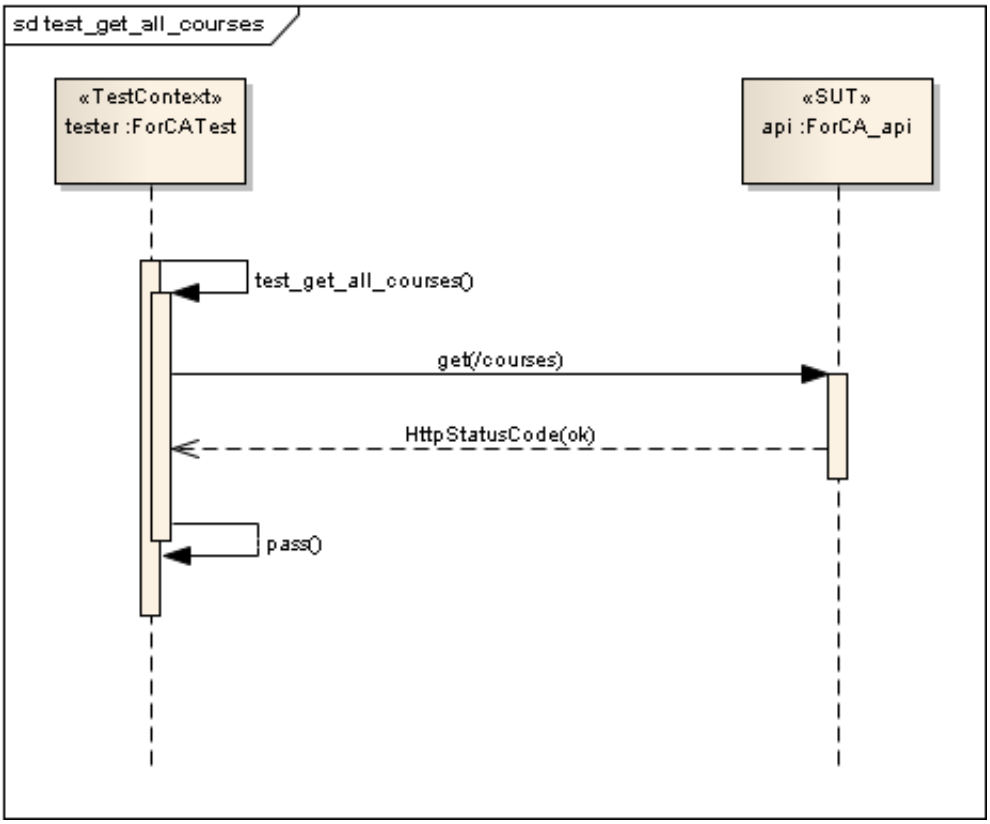

Figura 4.5: Exemplo de modelagem de caso de teste através de diagrama de sequência UML

Algumas restrições de modelagem devem ser obedecidas na modelagem comportamental para possibilitar a correta geração do código de testes (Borges 2009):

- O diagrama de sequência deve ter o mesmo nome do caso de teste que modela;
- O caso de teste deve iniciar com uma auto-mensagem por parte do *TestContext*, chamando seu próprio método correspondente ao teste atual;
- As mensagens do *TestContext* para o *SUT* devem ser operações HTTP definidas no protocolo da aplicação. Cada uma das operações deve conter um parâmetro, que indica o *namespace* da coleção de recursos sobre a qual a operação deve atuar e, quando necessário, referencia um objeto representando um recurso que deverá ser serializado e enviado para o *SUT*. No parâmetro, o caminho deve ser enviado contendo a barra inicial (/) e o objeto, caso exista, deve ser indicado entre parênteses após o caminho. No caso de um argumento dinâmico para identificar o recurso no

próprio caminho, este deve ser indicado entre menor (<) e maior (>) após o caminho e antes do parâmetro. Exemplo: *put(/courses/<id>(puttableCourse))*.

- As respostas do *SUT* para o *TestContext* podem ser de apenas dois tipos:
  - Uma resposta *ResourceId(id_name)* retorna para a classe testadora o identificador do recurso manipulado pela última operação solicitada pelo *TestContext*. Para o correto funcionamento desta funcionalidade, o SUT deve implementar corretamente o estilo REST de resposta a requisições, enviando no conteúdo da resposta o identificador (*id* ou URL) do recurso manipulado pela requisição. O parâmetro da mensagem de resposta *ResourceId(id_name)* indica o nome da variável onde será armazenada a identificação do recurso no *TestContext*.
  - Uma resposta *HttpStatusCode(status)* indica para a classe testadora o *status* HTTP esperado para a resposta da última requisição enviada. O parâmetro desta resposta deve ser um dos atributos da classe *TestComponent* definida na modelagem estrutural. Esta resposta deve ocorrer apenas uma vez no caso de teste, e deve ser a última mensagem enviada do *SUT* para o *TestContext*.
- O diagrama de sequência deve terminar com uma auto-mensagem do *TestContext*, contendo a operação *pass()*.

4.3 Geração do código de testes

A principal funcionalidade da proposta do **PyRester** é a geração automática de código de testes unitários, especificamente para o *framework* de testes **PyUnit**, da linguagem **Python**. A idéia central da solução apresentada é a proposta de um método para gerar código diretamente a partir de representações dos modelos UML e U2TP da aplicação alvo, de forma que o código gerado compreenda e implemente os mecanismos de teste descritos em tais modelos, a fim de que sejam gerados *drivers* e *dados* de teste unitário prontos para serem utilizados.

Python é uma linguagem de programação de alto nível de propósito geral, cuja filosofia de *design* enfatiza a legibilidade do código. Python suporta múltiplos paradigmas de programação (programação orientada a objetos; programação imperativa; programação funcional; programação procedural; programação reflexiva) e apresenta diversos mecanismos (tais como tipagem dinâmica e gerenciamento automático de memória) que a caracterizam como uma linguagem *dinâmica*. A linguagem tem sido utilizada de variadas formas em diversos contextos, sendo um dos mais comuns o desenvolvimento de aplicações Web através de *frameworks* de aplicação Web tais como *Django*, *Zope*, *Pylons*, *web2py* e *TurboGears* (Python 2011).

O uso da linguagem Python neste trabalho se deve a diversos fatores. Além da prévia familiaridade com a linguagem, do fato de a aplicação alvo da demonstração do método ser construída com o uso da mesma e das características de clareza, legibilidade, simplicidade e extensibilidade, outros fatores que motivaram a utilização são a facilidade de uso e abrangência de recursos de sua biblioteca padrão, e também a sua grande portabilidade, visto que atualmente existem implementações de Python para praticamente todos os ambientes operacionais conhecidos, e muitos dos sistemas baseados em Linux, por exemplo, trazem Python como um componente padrão pré-instalado.

O *framework* de testes **PyUnit** foi adotado como padrão para a geração do código de teste por ser o módulo padrão de teste unitário da linguagem Python e, assim, estar presente em praticamente todas as implementações e versões desta. Além disso, também contribuiu o fato de que PyUnit é diretamente derivado de **JUnit** e, portanto, faz parte da família de *frameworks* de teste *XUnit*, o que torna fácil tanto a compreensão do código gerado quanto a adaptação da proposta para os *frameworks* correspondentes de diversas outras linguagens que também são utilizadas para o fim da criação de *RESTful Web services* (PyUnit 2011).

Na Figura 4.6 são ilustradas as etapas do processo de geração de código do **PyRester**.

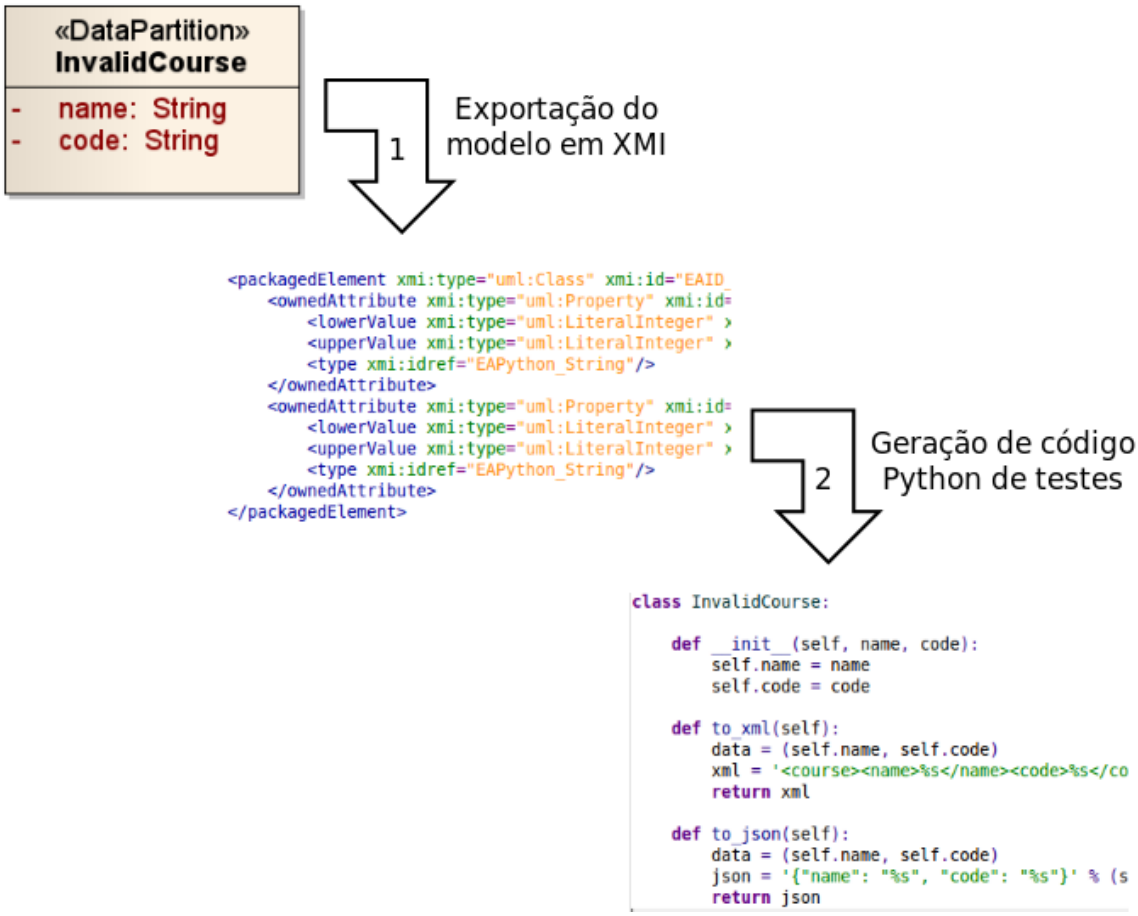

Figura 4.6: Ilustração das etapas do processo de geração de código do PyRester

4.3.1 Exportação dos modelos em XMI

Após a etapa de modelagem dos aspectos de teste em UML e U2TP, o primeiro passo para o processo de geração de código do PyRester é a exportação dos elementos desta modelagem para o formato XMI (XML Metadata Interchange). XMI é um padrão do OMG para o intercâmbio de informações de metadados via XML, que pode ser utilizado para quaisquer metadados que podem ser representados em MOF (Meta-Object Facility). Um exemplo de representação XMI pode ser visualizado na Figura 4.7.

O intercâmbio de diagramas UML ainda é um problema ativo no âmbito das linguagens de modelagem. Com o advento da UML 2.x foram estabelecidas duas principais linhas de solução para este problema: o padrão XMI, já bastante utilizado para este fim,

foi atualizado para prover compatibilidade com as estruturas do novo padrão de modelagem; ao mesmo tempo, a OMG iniciou a definição de um formato de intercâmbio específico para a UML, denominado *UML Diagram Interchange*. Ambos os padrões foram analisados durante a construção da proposta deste trabalho.

O XMI, em sua versão 2.1, foi escolhido como padrão de exportação dos modelos para o PyRester por ser suportado por um conjunto maior de ferramentas de modelagem UML, e por ter seu uso mais difundido e bem estabelecido, em comparação com o UML Diagram Interchange, que é suportado por poucas ferramentas e tem sua definição ainda em construção, apresentando ambiguidades e deficiências de documentação. Quando da escrita deste trabalho, o XMI 2.1 representa a versão mais atual deste padrão de intercâmbio de dados.

Várias ferramentas de modelagem UML foram avaliadas para a aplicação da proposta do PyRester. Dentre os critérios foram incluídos a familiaridade do autor com a ferramenta, a facilidade de uso, o suporte às estruturas da UML 2.0 necessárias para a modelagem dos aspectos de teste em U2TP, a compatibilidade com o padrão de intercâmbio XMI 2.1 e também a expressividade da ferramenta em termos de uso e difusão. Após a análise destes critérios, a ferramenta **Enterprise Architect**, da *Sparx Systems*, foi escolhida, em sua versão 8.0.

```
<?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:/
    <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_D9B4FB45_6013_4ae8_9BD4_162358
            <packagedElement xmi:type="uml:Collaboration" xmi:id="EAID_CB000000_B45_6013_4a
            <packagedElement xmi:type="uml:Class" xmi:id="EAID_EA4264FA_E3CE_46d0_854F_6611
            <packagedElement xmi:type="uml:Association" xmi:id="EAID_8A1933FA_C3F5_42cf_99F
            <packagedElement xmi:type="uml:Association" xmi:id="EAID_CBDA219B_DD60_47e5_A5B
            <packagedElement xmi:type="uml:Association" xmi:id="EAID_038E42A5_B26C_4870_8CF
            <packagedElement xmi:type="uml:Class" xmi:id="EAID_EB58DB04_1D8B_47b7_B4E3_5CF0
            <packagedElement xmi:type="uml:Class" xmi:id="EAID_6426BAA2_4FFA_4eff_ADDE_9F8F
            <packagedElement xmi:type="uml:Class" xmi:id="EAID_A2006B31_7D06_4289_AFE9_F8C5
            <packagedElement xmi:type="uml:Class" xmi:id="EAID_046C3D05_B85A_408b_B90A_C6DF
            <packagedElement xmi:type="uml:Class" xmi:id="EAID_92B8F459_6B6F_400e_8764_18CA
            <packagedElement xmi:type="uml:InstanceSpecification" xmi:id="EAID_8B00CE13_941
            <packagedElement xmi:type="uml:InstanceSpecification" xmi:id="EAID_751A7A05_D7A
        </packagedElement>
        <thecustomprofile:DataPool base_Class="EAID_EA4264FA_E3CE_46d0_854F_66113862AF5B"/>
        <thecustomprofile:TestContext base_Class="EAID_EB58DB04_1D8B_47b7_B4E3_5CF028A72D6C
        <thecustomprofile:TestCase base_Operation="EAID_800D0EDA_C172_4167_8440_66684BF4EAB
        <thecustomprofile:TestCase base_Operation="EAID_37776DE4_00C0_4347_B4ED_45EF10087A5
        <thecustomprofile:TestCase base_Operation="EAID_B709652F_5744_4ef6_9124_7AA75F3A0B4
        <thecustomprofile:SUT base_Class="EAID_6426BAA2_4FFA_4eff_ADDE_9F8F2D07D316"/>
        <thecustomprofile:TestComponent base_Class="EAID_A2006B31_7D06_4289_AFE9_F8C58CCEC1
        <thecustomprofile:enum base_Attribute="EAID_BA320ECB_1AAC_4734_9078_A649815B3359"/>
        <thecustomprofile:enum base_Attribute="EAID_5B24C7CB_D923_4eff_A52C_A086825B6E6C"/>
        <thecustomprofile:enum base_Attribute="EAID_877401FE_8EA2_4164_B2B4_BE87C3BCD3E1"/>
```

Figura 4.7: Exemplo de representação XMI de modelagem de aspectos de teste

4.3.2 Mapeamento do XMI para código Python

A partir da representação XMI gerada no passo anterior, o **PyRester** faz uso de um *parser* para realizar a interpretação desta representação e, a partir desta, gerar código Python para a execução dos testes modelados na primeira etapa. Este parser, desenvolvido também na linguagem Python, é o componente principal do PyRester. O código gerado pelo parser utiliza definições do *framework* de testes unitários **PyUnit**, considerado a ferramenta padrão para testes em Python e largamente utilizado por sua facilidade de uso

e conformidade com outros *frameworks* da família XUnit.

O *parser* do **PyRester** é um artefato de código baseado em recuperação e manipulação de dados, e abarca naturalmente a complexidade e multiplicidade de restrições e condições desta classe de aplicação. Por este motivo, este código possui chaveamentos e definições bastante granulares a fim de manipular corretamente a representação de entrada dos dados a fim de interpretá-la corretamente. Em linhas gerais, este *parser* tem seu funcionamento descrito pelas seguintes etapas:

- As estruturas da representação XMI são capturadas em uma estrutura de árvore que será posteriormente percorrida.

- Os estereótipos de todos os elementos são identificados e mapeados em uma estrutura para referência.

- Os diferentes tipos de elementos são identificados e separados em estruturas específicas.

- As classes da representação dos elementos da modelagem estrutural são percorridas e testadas de acordo com seu estereótipo e demais características de identificação. De acordo com o estereótipo, diferentes elementos (atributos, operações, etc.) são recuperados da representação destas classes e tem seus valores e parâmetros identificados. Estes elementos são, de acordo com o estereótipo da classe em análise, formatados em conformidade com a representação esperada na linguagem alvo. Estas representações formatadas são escritas no arquivo de saída respeitando as regras de identação e formatação.

- No caso das representações de elementos da modelagem comportamental, as mensagens que compõem os aspectos de comunicação são identificadas e têm seus atributos, parâmetros e argumentos recuperados e classificados. Os aspectos destas representações são então mapeados de acordo com o tipo de caso de teste, dentro dos possíveis tipos delineados na proposta. Estas representações são então formatadas em conformidade com a representação esperada na linguagem alvo e escritas no arquivo de saída, também respeitando regras de identação e formatação.

- Por fim, código de controle (*imports*, definições de métodos auxiliares, etc.) é escrito no arquivo de saída para que este possa ser prontamente executado, realizando a execução dos casos de teste especificados, bem como a avaliação de seus resultados.

O código gerado pelo parser do PyRester a partir da representação XMI dos aspectos de teste, ao ser executado, efetua os testes especificados no modelo, atendendo às restrições e parâmetros presentes nos diversos elementos da modelagem de testes. Ao executar o código gerado, as requisições HTTP expressas nos casos de teste representados pelos diagramas de sequência são enviadas ao *SUT*, de acordo com a dinâmica especificada, e seus resultados são obtidos e verificados. No caso das requisições do tipo POST ou PUT, o corpo da requisição é montado gerando-se representações JSON dos respectivos objetos. O arquivo contendo os testes pode ser facilmente integrado a uma suíte de testes existente.

A Tabela 4.1 apresenta o mapeamento entre os elementos da representação XMI e os trechos do código de testes gerado.

Tabela 4.1: Exemplo de mapeamento de modelos para código Python

Elemento	Código Python

TestComponent	`class HttpStatusCode:`
	`  ok = 200`
	`  created = 201`
	`  badRequest = 400`
	`  notFound = 404`
	`  conflict = 409`
	`  internalServerError = 500`

DataPartition (classes) e *DataSelector*	`class ValidCourse:`
	`  def __init__(self, **kwargs):`
	`  self.args = kwargs`
	`  self.name = kwargs['name']`
	`  self.code = kwargs['code']`
	
	`  def to_json(self):`
	`  return json.dumps(self.args)`
	
	`class InvalidCourse:`
	`  def __init__(self, **kwargs):`
	`  self.args = kwargs`
	`  self.name = kwargs['name']`
	`  self.code = kwargs['code']`
	
	`  def to_json(self):`
	`  return json.dumps(self.args)`

TestContext (classe)	`class ForCATest(unittest.TestCase):`

SUT	`  server = '127.0.0.1'`
	`  path = '/ForCA/api'`
	`  port = 8000`

DataPartition (objetos)	`  invalid = InvalidCourse(code="INV123456", name="Invalid 101")` `  valid = ValidCourse(code="VAL01234", name="Intro to Valid")`

TestContext (métodos)	`  def test_get_all_courses(self):`
	`  uri = "http://" + self.server + ":" + str(self.port) + self.path + "/courses"`
	`  response, content = http.request(uri, method="GET")`
	`  self.assertEqual(HttpStatusCode.ok, response.status)`
	
	`  def test_post_valid_course(self):`
	`  uri = "http://" + self.server + ":" + str(self.port) + self.path + "/courses"`
	`  body = self.valid.to_json()`
	`  headers = {"Content-type": "application/json"}`
	`  response, content = http.request(uri, method="POST", headers=headers, body=body)`
	`  self.assertEqual(HttpStatusCode.created, response.status)`
	
	`  def test_post_invalid_course(self):`
	`  uri = "http://" + self.server + ":" + str(self.port) + self.path + "/courses"`
	`  body = self.invalid.to_json()`
	`  headers = {"Content-type": "application/json"}`
	`  response, content = http.request(uri, method="POST", headers=headers, body=body)`
	`  self.assertEqual(HttpStatusCode.badRequest, response.status)`

4.3.3 Geração dos drivers e dados de teste para PyUnit

A execução do *parser* sobre o XMI gerado a partir da modelagem UML resulta em um arquivo (denominado, por padrão, *testcode.py*) que contém código de teste unitário *PyUnit*, composto por métodos (*drivers*) e dados de teste. Este arquivo, ao ser executado, efetuará a execução dos testes e a análise de seus resultados, retornando um veredito sobre a validade do SUT nos casos em questão. Um exemplo de arquivo de saída pode ser visualizado na Figura 4.8.

```
import json
import unittest
import httplib2

http = httplib2.Http()

class ValidCourse:
    def __init__(self, **kwargs):
        self.args = kwargs
        self.name = kwargs['name']
        self.code = kwargs['code']

    def to_json(self):
        return json.dumps(self.args)

class InvalidCourse:
    def __init__(self, **kwargs):
        self.args = kwargs
        self.name = kwargs['name']
        self.code = kwargs['code']

    def to_json(self):
        return json.dumps(self.args)

class HttpStatusCode:
    ok = 200
    created = 201
    badRequest = 400
    notFound = 404
    conflict = 409
    internalServerError = 500

class ForCATest(unittest.TestCase):
    server = '127.0.0.1'
    path = '/ForCA/api'

    invalid = InvalidCourse(code="INV123456", name="Invalid 101")
    valid = ValidCourse(code="VAL01234", name="Intro to Valid")

    def test_get_all_courses(self):
        uri = "http://" + self.server + ":" + str(self.port) + self.path + "/courses"
        response, content = http.request(uri, method="GET")
        self.assertEqual(HttpStatusCode.ok, response.status)

    def test_post_valid_course(self):
        uri = "http://" + self.server + ":" + str(self.port) + self.path + "/courses"
        body = self.valid.to_json()
        headers = {"Content-type": "application/json"}
        response, content = http.request(uri, method="POST", headers=headers, body=body)
        self.assertEqual(HttpStatusCode.created, response.status)

    def test_post_invalid_course(self):
        uri = "http://" + self.server + ":" + str(self.port) + self.path + "/courses"
        body = self.invalid.to_json()
        headers = {"Content-type": "application/json"}
        response, content = http.request(uri, method="POST", headers=headers, body=body)
        self.assertEqual(HttpStatusCode.badRequest, response.status)


if __name__ == '__main__':
    unittest.main()
```

Figura 4.8: Exemplo de código de teste unitário gerado pelo PyRester

5 EXEMPLO DE APLICAÇÃO DA PROPOSTA

Neste capítulo é demonstrado um exemplo prático de aplicação da proposta deste trabalho em um sistema real. Cada etapa do processo é descrita em termos de seu funcionamento e seus resultados, que são analisados em seguida.

5.1 Descrição do SUT (System Under Test)

ForCA (Fórum Colaborativo de Avaliação) é um sistema de avaliação docente colaborativa idealizado, desenvolvido e mantido por alunos do Instituto de Informática da Universidade Federal do Rio Grande do Sul. No ForCA, alunos cadastrados podem postar avaliações sobre suas próprias experiências acadêmicas, indicando um professor, uma disciplina e um semestre e informando seu parecer sobre este conjunto. Professores também podem se cadastrar a fim de responder às avaliações que alunos enviaram a seu respeito. Em matéria de dados, o ForCA armazena listagens de professores ativos, disciplinas do currículo, avaliações postadas, entre outros. Uma captura de tela do sistema **ForCA** pode ser visualizada na Figura 5.1.

Este sistema foi escolhido para fazer parte da demonstração da proposta deste trabalho por diversos motivos: familiaridade do autor com a aplicação-alvo; arquitetura da aplicação favorável à criação de interfaces no estilo REST; boa estruturação do esquema de modelos da aplicação, resultando em maior facilidade de mapeamento dos métodos HTTP às operações correspondentes do CRUD; possibilidade da aplicação da proposta em um sistema real e em atividade, proporcionando uma oportunidade verossímil de verificação dos resultados, bem como a criação de benefícios reais e tangíveis para a cobertura de testes do sistema, o que caracteriza a possibilidade de oferecer uma melhoria efetiva à aplicação.

Para o escopo deste trabalho, um subconjunto dos dados do sistema ForCA foi escolhido para ser exposto na forma de uma interface REST tornando, assim, o próprio sistema um *RESTful Web service*, neste aspecto. A interface desenvolvida permite a manipulação dos dados de disciplinas (**courses**) que integram os currículos dos cursos abrangidos pelo Instituto de Informática. Ou seja, os *recursos* do serviço web desenvolvido são de um único tipo: *courses*. As propriedades dos *courses* são vistas em seu modelo de dados, descrito na Tabela 5.1.

5.1.1 Protocolo da aplicação

O protocolo da aplicação exemplo consiste em cinco operações básicas, que compreendem os quatro métodos HTTP básicos e permitem a realização de um controle completo sobre o conjunto de dados de disciplinas, possibilitando que todas as operações do

Tabela 5.1: Modelo de dados dos recursos do tipo *courses*

Propriedade	Descrição	Caráter
name	Título completo da disciplina	**Obrigatório**
code	Código da disciplina	**Obrigatório**
resume	Súmula da disciplina	Opcional

Tabela 5.2: Protocolo da aplicação de exemplo (ForCA)

Método	URI	Descrição	Status possíveis
GET	ForCA/api/courses/	Recuperação de todos os *courses*.	**200**, 500
GET	ForCA/api/courses/**id**	Recuperação de *course* específico.	**200**, 400, 404, 500
POST	ForCA/api/courses/	Criação de novo *course*.	**201**, 400, 409, 500
PUT	ForCA/api/courses/**id**	Atualização de *course* específico.	**200**, 400, 404, 409, 500
DELETE	ForCA/api/courses/**id**	Deleção de *course* específico.	**200**, 400, 404, 500

CRUD sejam efetuadas sobre estes dados através da interface REST exposta. A descrição das cinco operações básicas da interface, incluindo os parâmetros e URIs de acesso e também os códigos de status HTTP possíveis e esperados para cada uma delas pode ser vista na Tabela 5.2.

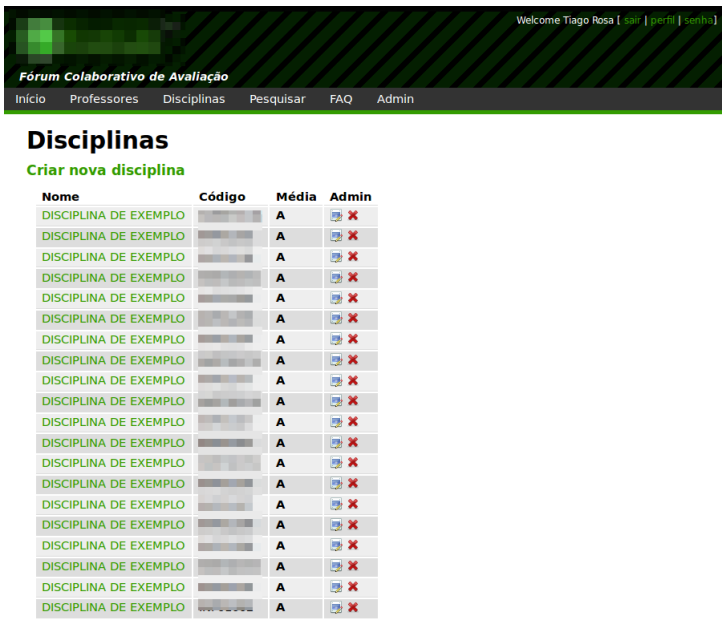

Figura 5.1: Captura de tela da seção de Disciplinas do sistema ForCA

5.2 Etapa 1 - Modelagem dos aspectos de teste em U2TP

A primeira etapa do processo é a modelagem dos aspectos de teste da aplicação utilizando o padrão U2TP. Os diversos aspectos são modelados em diferentes elementos,

cada qual responsável pela definição de um determinado conjunto de especificidades. Estes elementos representam os parâmetros da aplicação alvo, os dados e partições de dados a serem usados para os testes e a própria dinâmica dos casos de teste, incluindo a especificação das requisições e definição das respostas esperadas durante o fluxo de execução.

Neste exemplo, serão modelados elementos de teste suficientes para a geração de seis casos de teste, contemplando todo o protocolo da aplicação descrito acima. A seguir, cada um dos elementos resultantes da modelagem da aplicação *ForCA* é descrito e detalhado:

- **SUT** - Uma classe chamada *ForCA_api* (ilustrada na Figura 5.2), com estereótipo *«SUT»*, é definida no diagrama de classes, contendo alguns parâmetros para acesso da aplicação a partir do *driver* de testes. O atributo **server** contém a identificação do servidor que roda a aplicação; o atributo **path** contém o caminho da aplicação no servidor; e o atributo **port** indica a porta pela qual a aplicação pode ser acessada no servidor.

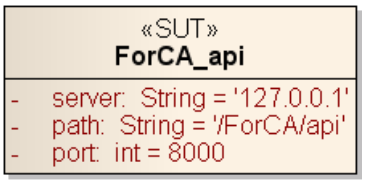

Figura 5.2: Modelagem dos parâmetros de acesso à aplicação

- **TestComponent** - A classe *HttpStatusCode* (ilustrada na Figura 5.3), com estereótipo *«TestComponent»*, possui uma enumeração de atributos (marcados com o estereótipo *«enum»*) que corresponde à listagem de códigos de status HTTP que a aplicação pode retornar para chamadas a funções de seu protocolo.

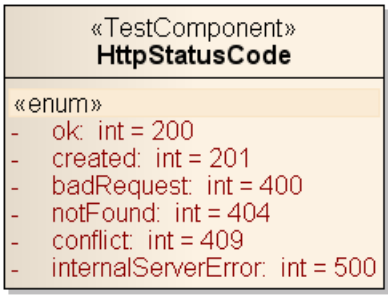

Figura 5.3: Modelagem dos códigos de status HTTP esperados para a aplicação

- **DataPool e DataPartition** - No diagrama também são definidas classes que modelam os elementos de dados a serem usados nos testes: a classe *CourseDataPool*, com estereótipo *«DataPool»*, associa as partições de dados definidas nas classes *ValidCourse* e *InvalidCourse*, com estereótipo *«DataPartition»*, que representam taxonomias específicas de dados que entrarão para o fluxo de testes. A modelagem destas classes pode ser visualizada na Figura 5.4.

  Cinco objetos (quatro instâncias da classe de partição de dados *ValidCourse* e uma instância da classe de partição de dados *InvalidCourse*) são especificados na modelagem para serem usados como objetos de dados passados como parâmetros no

corpo da requisição de cada um dos seis casos de teste contemplados por este exemplo. Cada um destes objetos é instanciado e inicializado com valores adequados para a validação esperada no SUT - o objeto *invalid*, por exemplo, possui no atributo *code* um *string* com um número de caracteres maior do que a aplicação-alvo permite para o código de uma disciplina e, portanto, caracteriza uma disciplina inválida. A modelagem destes objetos pode ser visualizada na Figura 5.5.

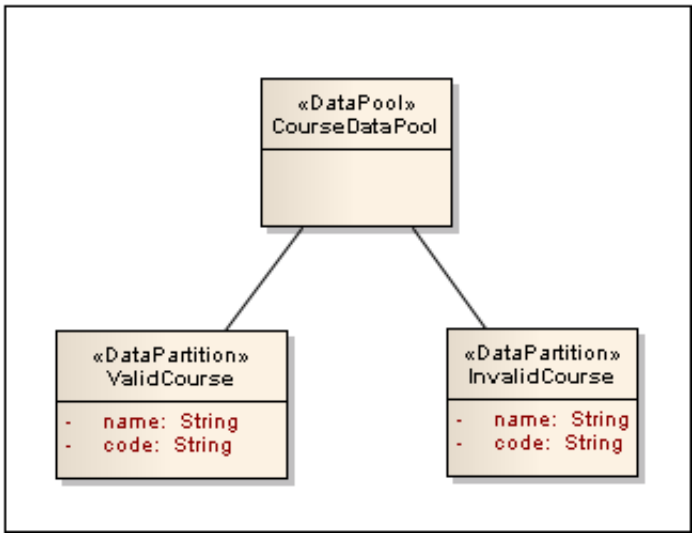

Figura 5.4: Modelagem das classes de partição de dados a serem usadas para o teste da aplicação

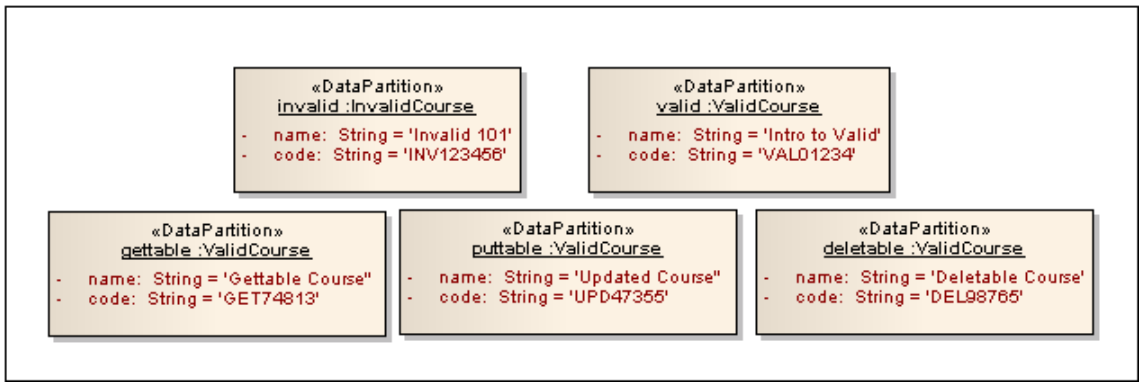

Figura 5.5: Modelagem dos objetos de dados a serem usados para o teste da aplicação

- **TestContext e TestCase** - A classe *ForCATest* (ilustrada na Figura 5.6), com estereótipo *«TestContext»*, define o contexto de teste da aplicação. Seus métodos, marcados com o estereótipo *«TestCase»*, são as definições dos casos de teste a serem realizados na aplicação. Neste exemplo, foram definidos seis casos de teste, cinco dos quais recebem um parâmetro que será serializado e enviado no corpo da requisição.

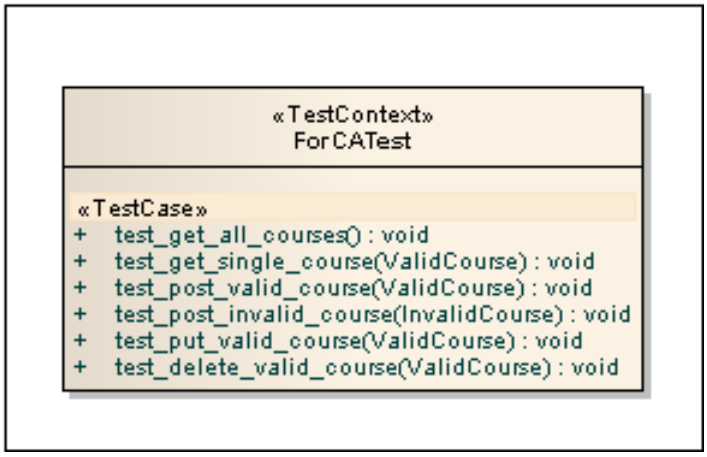

Figura 5.6: Modelagem da definição dos casos de teste propostos para a aplicação

Já na modelagem comportamental, os casos de teste são modelados como diagramas de sequência. Duas *lifelines* compõem cada um dos testes: a *lifeline* **tester**, com estereótipo *«TestContext»*, representa o *driver* de testes, que envia requisições ao SUT e aguarda suas respostas. Já a *lifeline* **api**, com estereótipo *«SUT»*, representa a aplicação alvo, que recebe requisições do *driver* de teste e responde com códigos HTTP correspondentes ao status de cada uma delas.

- **test_get_all_courses** - Neste caso de teste, são buscadas todas as *disciplinas* presentes na base de dados, acessadas através de uma requisição GET sem parâmetros. A resposta esperada deve conter em seu corpo a representação JSON do conjunto completo de *disciplinas* cadastradas na aplicação, e seu status deve ser 200 (*ok*). A modelagem deste caso de teste pode ser vista na Figura 5.7.

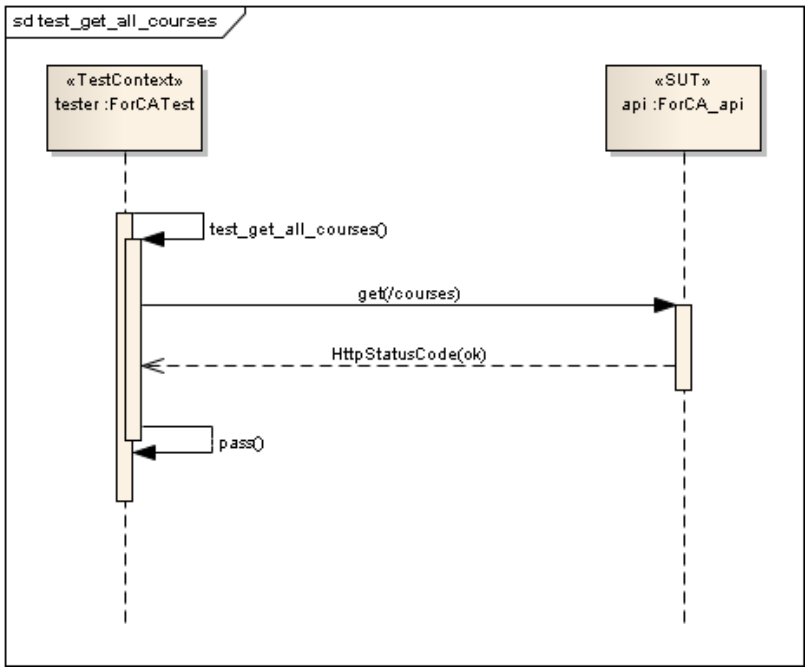

Figura 5.7: Modelagem do caso de teste *test_get_all_courses*

- **test_post_valid_course** - Neste caso de teste, é efetuada a criação de uma nova *disciplina* no sistema, através de uma requisição POST, cujo parâmetro é uma instância da classe *ValidCourse*, que representa uma *disciplina* válida, com atributos bem-formados. A resposta esperada deve apresentar o status 203 (*created*). A modelagem deste caso de teste pode ser vista na Figura 5.8.

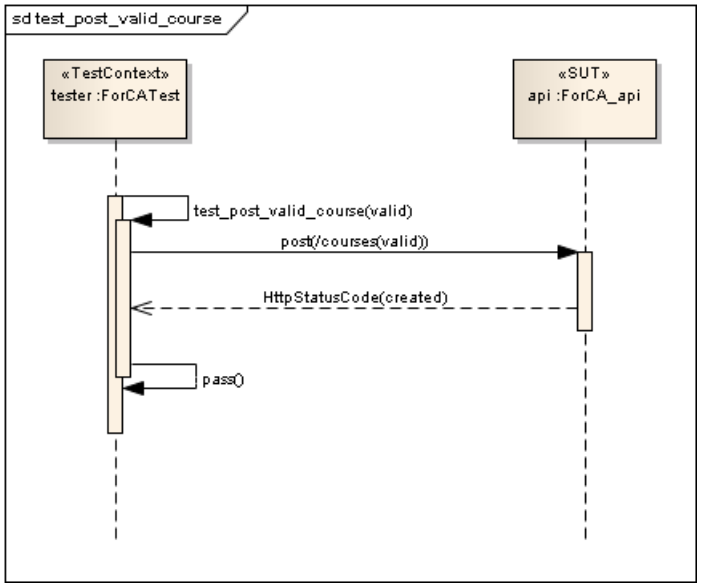

Figura 5.8: Modelagem do caso de teste *test_post_valid_course*

- **test_post_invalid_course** - Neste caso de teste, é efetuada a criação de uma nova *disciplina* no sistema, através de uma requisição POST, cujo parâmetro é uma instância da classe *InvalidCourse*, que representa uma *disciplina* inválida, com atributos malformados (o atributo *code* tem mais caracteres do que o permitido pela validação do sistema). A resposta esperada deve apresentar o status 400 (*badRequest*). A modelagem deste caso de teste pode ser vista na Figura 5.9.

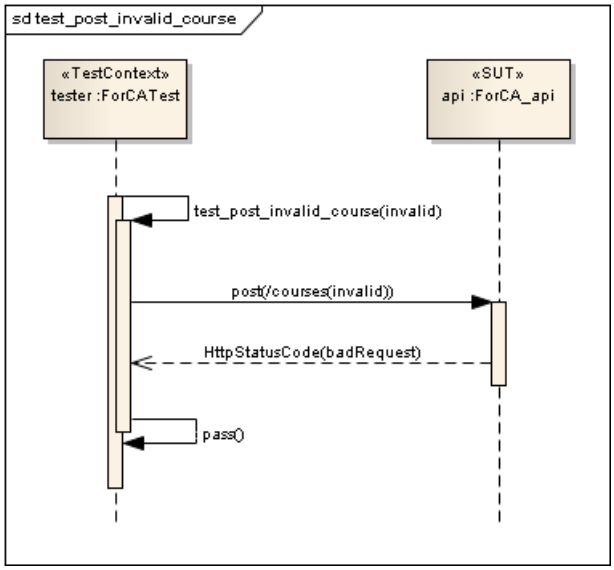

Figura 5.9: Modelagem do caso de teste *test_post_invalid_course*

O correto funcionamento dos casos de teste a seguir depende da corretude de uma característica do SUT: a resposta a uma requisição com parâmetro (ou seja, com a serialização de um objeto em seu corpo) deve conter a identificação do recurso que representa o objeto - seja em forma de um *id*, ou da própria URL do recurso na aplicação.

- **test_get_single_course** - Neste caso de teste, duas operações distintas são realizadas para verificar o correto funcionamento da operação GET atuando sobre um recurso específico. Primeiro, uma nova disciplina é criada no sistema através de uma requisição POST, contendo em seu corpo a serialização de um objeto da classe *ValidCourse* (uma disciplina válida). Em seguida, é realizada a busca pelo recurso criado, através de uma requisição GET, contendo a identificação do recurso em questão (argumento *id*). Caso a recuperação do recurso obtenha sucesso, a resposta esperada deve apresentar o status 200 (*ok*). A modelagem deste caso de teste pode ser vista na Figura 5.10.

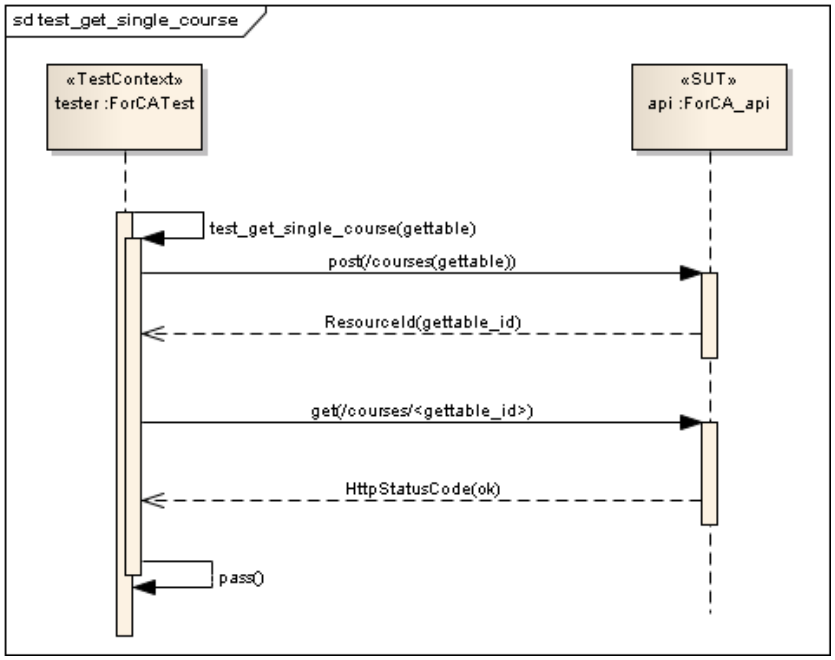

Figura 5.10: Modelagem do caso de teste *test_get_single_course*

- **test_put_valid_course** - Neste caso de teste, duas operações distintas são realizadas para verificar o correto funcionamento da operação PUT. Primeiro, uma nova disciplina é criada no sistema através de uma requisição POST, contendo em seu corpo a serialização de um objeto da classe *ValidCourse* (uma disciplina válida). Em seguida, é realizada a atualização do recurso criado, através de uma requisição PUT, contendo como argumento a identificação do recurso a ser atualizado (argumento *id*) e em seu corpo a serialização de outro objeto da classe *ValidCourse* (outra disciplina válida). Caso a atualização do recurso obtenha sucesso, a resposta esperada deve apresentar o status 200 (*ok*). A modelagem deste caso de teste pode ser vista na Figura 5.11.

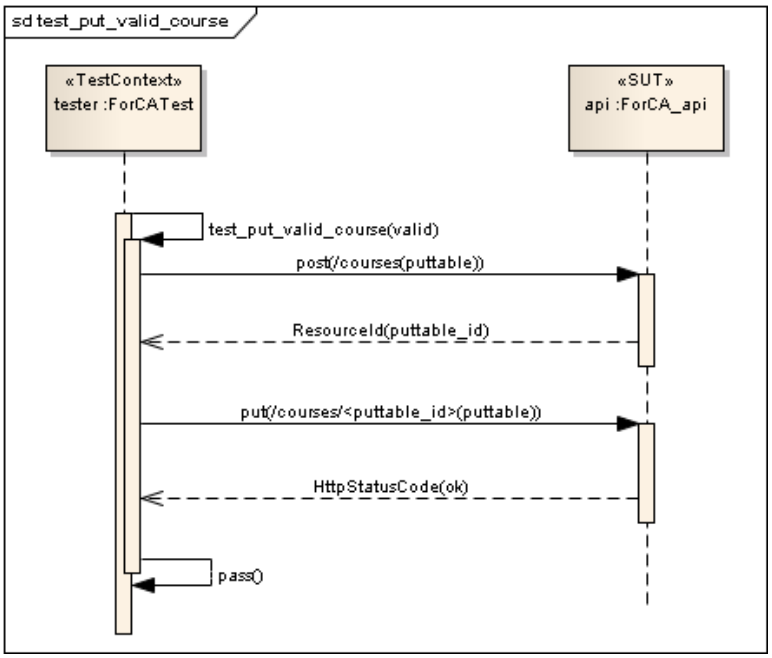

Figura 5.11: Modelagem do caso de teste *test_put_valid_course*

- **test_delete_valid_course** - Neste caso de teste, duas operações distintas são realizadas para verificar o correto funcionamento da operação DELETE. Primeiro, uma nova disciplina é criada no sistema através de uma requisição POST, contendo em seu corpo a serialização de um objeto da classe *ValidCourse* (uma disciplina válida). Em seguida, é realizada a deleção do recurso criado, através de uma requisição DELETE, contendo a identificação do recurso em questão (argumento *id*). Caso a deleção do recurso obtenha sucesso, a resposta esperada deve apresentar o status 200 (*ok*). A modelagem deste caso de teste pode ser vista na Figura 5.12.

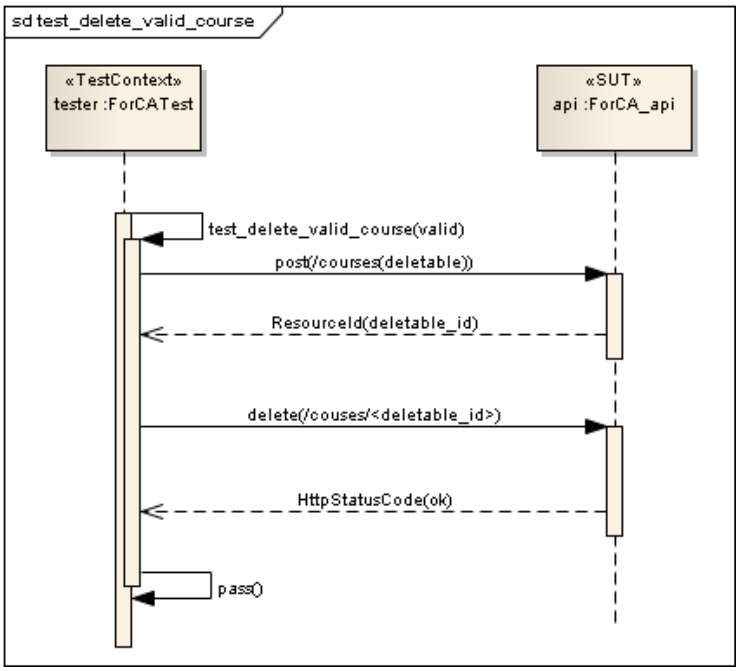

Figura 5.12: Modelagem do caso de teste *test_delete_valid_course*

Na Figura 5.13 pode ser visualizada a modelagem completa dos aspectos estruturais de teste do **ForCA**.

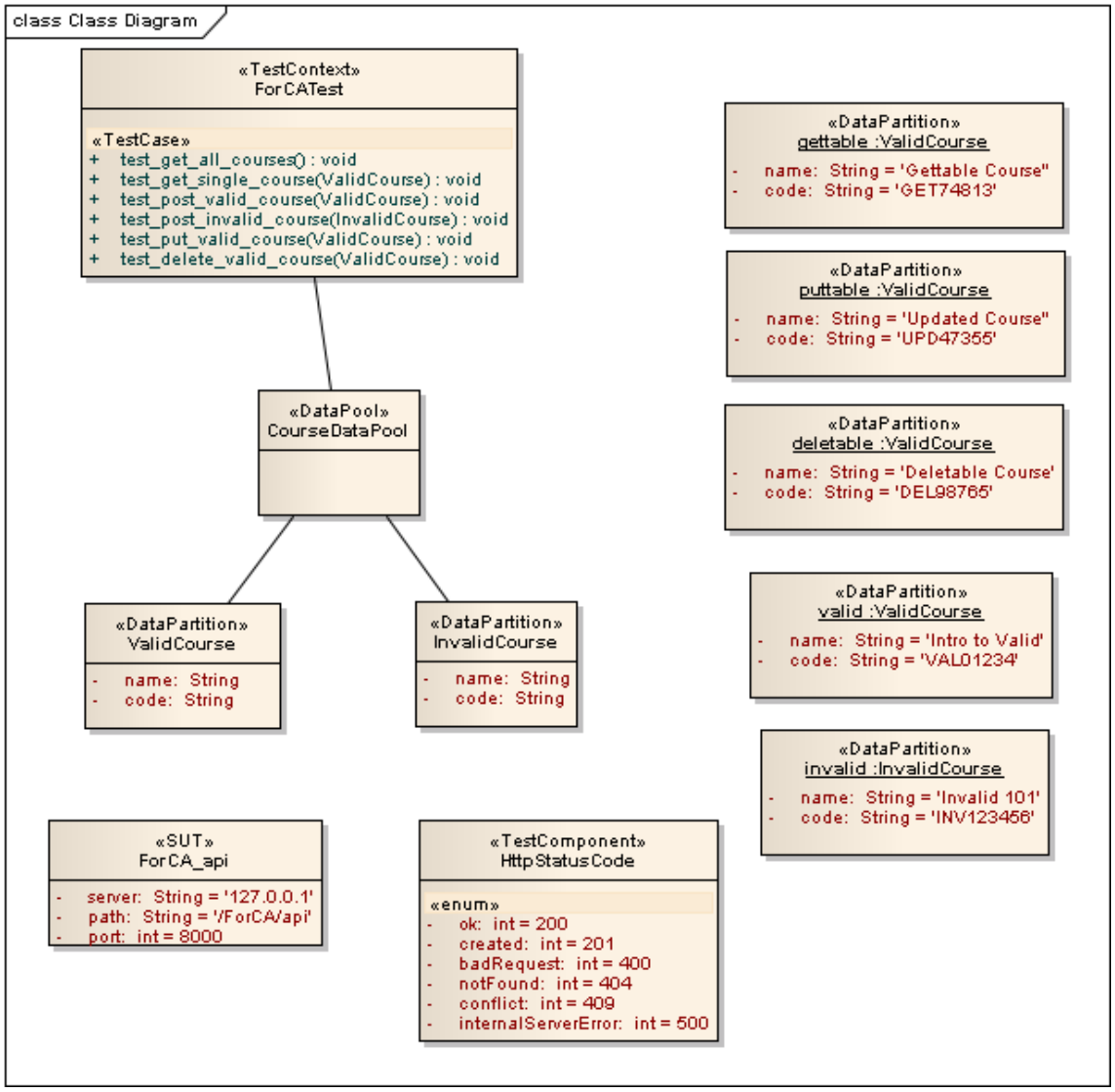

Figura 5.13: Diagrama de classes da modelagem dos aspectos estruturais de teste do ForCA

5.3 Etapa 2 - Exportação dos modelos em XMI

A segunda etapa do processo é a exportação dos modelos gerados para o formato XMI. Esta operação é suportada por grande parte das ferramentas de modelagem existentes no mercado, e pode ser feita de forma a não incluir na representação exportada informações referentes a *layout* e organização visual dos diagramas, sem comprometer a eficácia da geração de código de teste.

Neste exemplo, o XMI gerado tem uma estrutura semelhante ao representado na Figura 5.14:

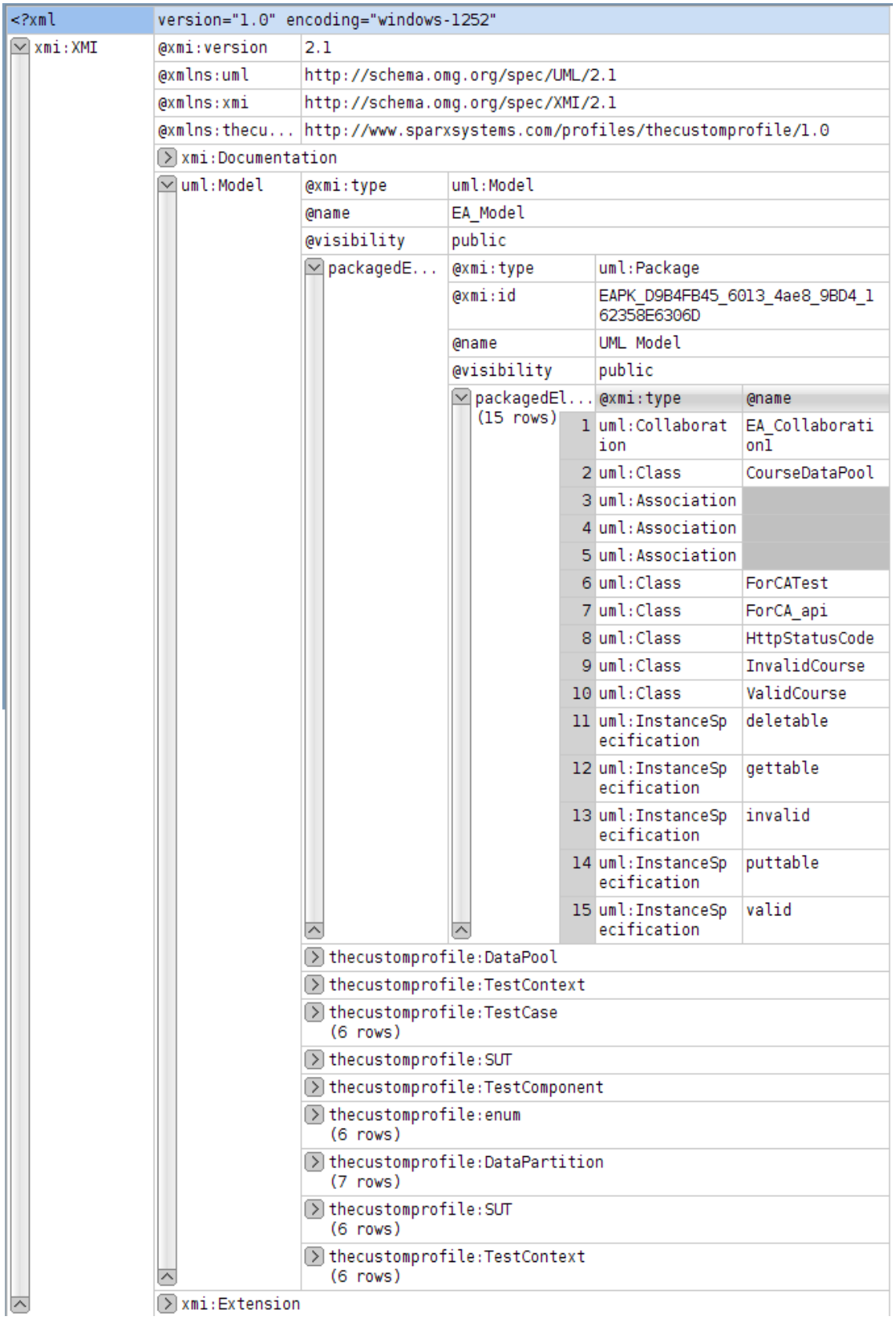

Figura 5.14: Estrutura da representação XMI da modelagem dos aspectos de teste do ForCA

5.4 Etapa 3 - Geração dos drivers e dados de teste para PyUnit

A terceira e principal etapa do processo é a geração do código de testes a partir da representação XMI gerada na etapa 2. É nesta etapa que entra em ação o *parser* de XMI do **PyRester** que, varrendo o arquivo XMI de entrada, identifica os elementos, seus componentes e estereótipos, organizando logicamente estas construções e criando código Python executável que captura a estrutura e o comportamento dos testes especificados na modelagem U2TP. Na Figura 5.15 pode ser visualizado um trecho do código do *parser* do **PyRester**.

```
#Iterates through the classes, verifying stereotype to build the code structure
for cls in classes:

    #TestComponent class
    if stereomap[cls] == 'TestComponent':
        if len(classes[cls]) < 1:
            raise Exception, "TestComponent class is empty"
        gen.write('class %s:' % classes[cls].attrib['name'])
        gen.indent()

        #Gets and defines enum attributes
        for attr in classes[cls]:
            if attr.tag == 'ownedAttribute':
                attr_id = attr.attrib[keyfy('id', 'xmi')]
                #Checks if attribute has 'enum' stereotype
                if attr_id not in stereomap:
                    continue
                if stereomap[attr_id] != 'enum':
                    continue
                attr_name = attr.attrib['name']
                attr_value = None
                for val in attr:
                    if val.tag == 'defaultValue':
                        attr_value = val.attrib['value']

            #Writes the attributes and default values
            if attr_name and attr_value:
                gen.write('%s = %s' % (attr_name, attr_value))

        gen.dedent()
        gen.write('')
```

Figura 5.15: Trecho do código do *parser* de XMI do **PyRester**

Para a geração do código de teste, o *parser* do **PyRester** pode ser executado em linha de comando como uma aplicação Python padrão, recebendo como argumento o nome do arquivo XMI de entrada a ser analisado:

```
python pyrester.py forca_model.xmi
```

Pela própria natureza da aplicação e da linguagem, o **PyRester** é multiplataforma, podendo ser executado nos mais diversos ambientes e sistemas operacionais que suportem Python. Em cada um destes ambientes, o funcionamento do **PyRester** se dá de forma análoga à execução de qualquer script Python, não requerendo parâmetros ou configurações adicionais em qualquer plataforma. O código de testes gerado pela aplicação do *parser* sobre a modelagem de testes proposta para o sistema **ForCA** é ilustrada na Figura 5.16 e na Figura 5.17.

```
import json
import unittest
import httplib2

http = httplib2.Http()

class ValidCourse:
    def __init__(self, **kwargs):
        self.args = kwargs
        self.name = kwargs['name']
        self.code = kwargs['code']

    def to_json(self):
        return json.dumps(self.args)

class InvalidCourse:
    def __init__(self, **kwargs):
        self.args = kwargs
        self.name = kwargs['name']
        self.code = kwargs['code']

    def to_json(self):
        return json.dumps(self.args)

class HttpStatusCode:
    ok = 200
    created = 201
    badRequest = 400
    notFound = 404
    conflict = 409
    internalServerError = 500

class ForCATest(unittest.TestCase):
    server = '127.0.0.1'
    path = '/ForCA/api'
    port = 8000

    deletable = ValidCourse(code="DEL98765", name="Deletable Course")
    gettable = ValidCourse(code="GET74813", name="Gettable Course")
    invalid = InvalidCourse(code="INV123456", name="Invalid 101")
    puttable = ValidCourse(code="UPD47355", name="Updated Course")
    valid = ValidCourse(code="VAL01234", name="Intro to Valid")

    def test_get_all_courses(self):
        uri = "http://" + self.server + ":" + str(self.port) + self.path + "/courses"
        response, content = http.request(uri, method="GET")
        self.assertEqual(HttpStatusCode.ok, response.status)

    def test_get_single_course(self):
        uri = "http://" + self.server + ":" + str(self.port) + self.path + "/courses"
        body = self.gettable.to_json()
        headers = {"Content-type": "application/json"}
        response, content = http.request(uri, method="POST", headers=headers, body=body)
        try:
            if content.isdigit():
                gettable_id = content
            else:
                gettable_id = content.split("/")[-1]
        except:
            raise Exception, "GET (single) test POST request response content is empty"

        uri = "http://" + self.server + ":" + str(self.port) + self.path + "/courses/" + gettable_id
        response, content = http.request(uri, method="GET")
        self.assertEqual(HttpStatusCode.ok, response.status)
```

Figura 5.16: Código de teste gerado a partir da modelagem dos aspectos de teste do ForCA (página 1 de 2)

```
    def test_post_valid_course(self):
        uri = "http://" + self.server + ":" + str(self.port) + self.path + "/courses"
        body = self.valid.to_json()
        headers = {"Content-type": "application/json"}
        response, content = http.request(uri, method="POST", headers=headers, body=body)
        self.assertEqual(HttpStatusCode.created, response.status)

    def test_post_invalid_course(self):
        uri = "http://" + self.server + ":" + str(self.port) + self.path + "/courses"
        body = self.invalid.to_json()
        headers = {"Content-type": "application/json"}
        response, content = http.request(uri, method="POST", headers=headers, body=body)
        self.assertEqual(HttpStatusCode.badRequest, response.status)
```

```
    def test_post_invalid_course(self):
        uri = "http://" + self.server + ":" + str(self.port) + self.path + "/courses"
        body = self.invalid.to_json()
        headers = {"Content-type": "application/json"}
        response, content = http.request(uri, method="POST", headers=headers, body=body)
        self.assertEqual(HttpStatusCode.badRequest, response.status)

    def test_put_valid_course(self):
        uri = "http://" + self.server + ":" + str(self.port) + self.path + "/courses"
        body = self.puttable.to_json()
        headers = {"Content-type": "application/json"}
        response, content = http.request(uri, method="POST", headers=headers, body=body)
        try:
            if content.isdigit():
                puttable_id = content
            else:
                puttable_id = content.split("/")[-1]
        except:
            raise Exception, "PUT test POST request response content is empty"

        uri = "http://" + self.server + ":" + str(self.port) + self.path + "/courses/" + puttable_id
        body = self.puttable.to_json()
        headers = {"Content-type": "application/json"}
        response, content = http.request(uri, method="PUT", headers=headers, body=body)
        self.assertEqual(HttpStatusCode.ok, response.status)

    def test_delete_valid_course(self):
        uri = "http://" + self.server + ":" + str(self.port) + self.path + "/courses"
        body = self.deletable.to_json()
        headers = {"Content-type": "application/json"}
        response, content = http.request(uri, method="POST", headers=headers, body=body)
        try:
            if content.isdigit():
                deletable_id = content
            else:
                deletable_id = content.split("/")[-1]
        except:
            raise Exception, "DELETE test POST request response content is empty"

        uri = "http://" + self.server + ":" + str(self.port) + self.path + "/courses/" + deletable_id
        response, content = http.request(uri, method="DELETE")
        self.assertEqual(HttpStatusCode.ok, response.status)
```

Figura 5.17: Código de teste gerado a partir da modelagem dos aspectos de teste do ForCA (página 2 de 2)

5.5 Etapa 4 - Execução dos testes

A etapa final do processo consiste na execução dos testes propriamente ditos. O arquivo gerado na etapa 3, contendo o código de teste unitário para o ForCA, pode ser executado normalmente conforme os padrões do *framework* PyUnit. No exemplo abaixo, o arquivo de testes foi executado com o parâmetro *-v*, que aumenta a verbosidade do *output* na tela. Na Figura 5.16 pode ser visualizada a saída de terminal da execução do **PyRester**.

```
tiago@clapton~/Desktop/TCC/pyrester$ python testcode.py -v
test_delete_valid_course (__main__.ForCATest) ... ok
test_get_all_courses (__main__.ForCATest) ... ok
test_get_single_course (__main__.ForCATest) ... ok
test_post_invalid_course (__main__.ForCATest) ... ok
test_post_valid_course (__main__.ForCATest) ... ok
test_put_valid_course (__main__.ForCATest) ... ok

----------------------------------------------------------------------
Ran 6 tests in 4.244s

OK
```

Figura 5.18: Saída da execução do arquivo de testes gerado pelo **PyRester**

6 CONCLUSÕES

Aqui são apresentadas e discutidas as conclusões alcançadas após este trabalho, incluindo a análise dos resultados obtidos e a avaliação da proposta em termos de suas limitações e suas possibilidades de melhoria futura.

6.1 Análise dos resultados obtidos

Neste trabalho foi desenvolvida uma proposta para a geração de código de teste a partir de modelos. Esta proposta foi validada em termos práticos através da implementação de um gerador de código de teste, construído com o uso de tecnologias altamente relevantes nos dias atuais como Python, UML 2.0, U2TP, XML e JSON. A validade e a funcionalidade desta proposta foi avaliada através de sua aplicação a um sistema de software real, construído fora do escopo deste trabalho. Os resultados desta aplicação alcançaram todos os objetivos propostos por este trabalho, efetivamente facilitando a geração de código de teste unitário funcional e relevante, de forma automática, com a capacidade de avaliar objetivamente a funcionalidade do sistema sob teste.

Em uma visão mais abrangente, a eficácia da proposta deste trabalho contribui de forma determinante para a validação de todo um modelo teórico que abrange desde técnicas de teste dirigido por modelos até os conceitos de arquitetura e comportamento de teste propostos pelo U2TP, passando pelos aspectos mais básicos e importantes do estilo arquitetural REST e, consequentemente, das aplicações do tipo *RESTful Web Services*. O sucesso da aplicação da proposta desenvolvida neste trabalho é também um resultado positivo para a avaliação prática destes conceitos, especialmente no que diz respeito ao seu uso combinado para aplicações práticas de teste automatizado.

Portanto, os resultados obtidos através deste trabalho apresentam contribuições importantes para a diminuição do tempo, custo e esforço necessários para a implementação de código de teste unitário para *RESTful Web Services*, facilitando a adoção deste processo na etapa de desenvolvimento desta classe de sistemas. Desta forma, estes resultados contribuem também de forma direta para a eficiência e a produtividade do desenvolvimento de serviços Web de maior qualidade. Isto, por sua vez, influi diretamente na melhoria da utilização das boas práticas REST através do desenvolvimento de aplicações de maior confiabilidade e manutenibilidade, contribuindo não apenas para o estabelecimento de um estilo arquitetural mas também para o avanço dos padrões de qualidade relacionados aos sistemas desta classe.

Em relação aos trabalhos relacionados, além de ampliar o escopo de aplicação do método para diferentes linguagens e tecnologias, o **PyRester** estende a proposta do método ao introduzir novas possibilidades e abranger características ausentes em outras propostas de forma a apresentar a união destas funcionalidades em uma ferramenta única. Em

Tabela 6.1: Comparação do *PyRester* com os trabalhos relacionados

Característica/Ferramenta	cURL	REST Console	Python REST Client	REST-Unit	REST-Unit+	PyRester
Suporte a requisições GET	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**
Suporte a requisições POST	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**
Suporte a requisições PUT	**sim**	**sim**	**sim**	não	não	**sim**
Suporte a requisições DELETE	**sim**	**sim**	**sim**	não	não	**sim**
Gerenciamento de dados de teste	não	não	não	não	**sim**	**sim**
Interface programática	**sim**	não	**sim**	**sim**	**sim**	**sim**
Geração de código de teste	não	não	não	**sim**	**sim**	**sim**
Adequação a *frameworks* de teste	não	não	não	**sim**	**sim**	**sim**
Suporte a *payload* XML	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**
Suporte a *payload* JSON	**sim**	**sim**	**sim**	não	não	**sim**
Casos de teste com múltiplas *requests*	não	não	**sim**	não	não	**sim**

especial, em relação às propostas de Borges (2009) e Feller (2010), são introduzidas duas características importantes: a generalização do formato do *payload* das requisições (possibilitando o chaveamento entre os formatos XML e JSON) e, principalmente, o suporte a casos de teste complexos, com múltiplas requisições, que capturam de forma mais abrangente o comportamento de testes de unidade mais cmpletos, úteis e relevantes. Estas adições propostas tornam o método de geração de código de testes muito mais funcional e prontamente aplicável a aplicações reais. Além disso, neste trabalho é utilizado um exemplo real de aplicação, buscando a validação do método de forma mais prática e verossímil. Na Tabela 6.1, o **PyRester** é comparado com as demais soluções relacionadas em termos de suas características e funcionalidades.

6.2 Limitações da proposta apresentada

Pela própria natureza da proposta, que busca avaliar a aplicação do método de geração automática de código de teste com base em modelos U2TP especificamente para as aplicações do tipo *RESTful Web Services*, existem algumas limitações neste trabalho que devem ser consideradas:

- O processo de geração de código é bastante sensível à forma; mudanças no formato das representações XMI dos modelos, ainda que pequenas, podem requerer alterações importantes, especialmente no *parser* de XMI do **PyRester**. Estas mudanças podem ser provocadas por atualizações no padrão ou, mais frequentemente, pelas diferenças sintáticas de geração de XMI apresentadas pelas diversas ferramentas de modelagem causadas pelas próprias ambiguidades da definição deste padrão.
- Os aspectos de temporização de teste do U2TP não são explorados de forma importante, por causa da complexidade de seu mapeamento e pelo pouco benefício que

apresentam ao escopo deste trabalho em específico.

- A representação interna dos dados dos *Web services* deve necessariamente ser JSON ou XML. Não são considerados pela proposta outros formatos de representação. Além disso, singularidades ou inconsistências na representação mesmo nestes formatos podem atrapalhar o processo de geração de código de teste.

- A proposta não abrange validações mais complexas dos dados retornados pelo SUT, pois seu mapeamento para geração automática requer a inclusão de muitas especificidades e casos secundários que aumentariam a complexidade do código e diminuiriam sua clareza e objetividade.

6.3 Melhorias futuras

A proposta do **PyRester**, por suas próprias características de método, apresenta a possibilidade de diversas melhorias futuras e também de novas aplicações com funcionamento semelhante:

- O processo pode ser estendido para abranger outras classes de aplicação e também outros objetivos de teste, podendo fazer parte de uma ferramenta parametrizável.

- Os aspectos temporais do U2TP podem ser melhor explorados para a validação de sistemas mais complexos ou para a abrangência de conceitos de teste de performance.

- Os dados retornados pelo SUT podem ser tratados em termos de validações mais complexas, estendendo o escopo da proposta para diferentes objetivos e mecanismos de teste.

- O *PyRester* pode ser adaptado para um serviço *Web* que permita a geração de código de teste com base em *input* do usuário, oferecendo facilidades adicionais como representação gráfica dos modelos fornecidos como entrada e intercâmbio entre formatos de representação.

- A proposta pode se tornar especialmente conveniente e intuitiva ao ser integrada a uma ferramenta de modelagem. O **PyRester** pode ser empacotado como um módulo ou *plugin* destinado a ferramentas de modelagem, tornando o desenvolvimento de testes para as aplicações modeladas muito mais visual e interativo.

REFERÊNCIAS

[Apache Wink 2011]APACHE Wink. 2011. Acesso em novembro de 2011. Disponível em: <https://cwiki.apache.org/WINK/1-introduction-to-apache-wink.html>.

[Baker et al. 2010]BAKER, P. et al. *Model-Driven Testing: Using the UML Testing Profile*. Berlin: Springer, 2010. ISBN 3642091598.

[Biasi 2006]BIASI, L. B. *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 2009]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.

[Borges et al. 2009]BORGES, F. Q.; PIMENTA, M. S.; WEBER, T. S. Rest-unit: geração baseada em u2tp de drivers de teste para restful web services. In: UFPEL. *Conferencia Latinoamericana de Informatica*. Pelotas, RS, 2009. p. 11–21.

[cURL 2011]CURL. 2011. Acesso em outubro de 2011. Disponível em: <http://curl.haxx.se/>.

[Feller 2010]FELLER, N. J. *Estendendo Rest-Unit: Geração Baseada em U2TP de Drivers e Dados de Teste para RESTful Web Services*. 2010. 79 p. Projeto de Diplomação (Bacharelado em Ciência da Computação) - Instituto de Informática, UFRGS, Porto Alegre.

[Fewster e Graham 1999]FEWSTER, M.; GRAHAM, D. *Software Test Automation*. [S.l.]: Addison-Wesley Professional, 1999. ISBN 0201331403.

[Fielding 2000]FIELDING, R. T. *Architectural Styles and the Design of Network-based Software Architectures*. Tese (Doutorado) — University of California, Irvine, 2000. 180 p.

[McMillan 2003]MCMILLAN, R. *A RESTful approach to Web services*. fev. 2003. Acesso em agosto de 2011. Disponível em: <http://www.networkworld.com/ee/2003/eerest.html>.

[OMG 2005]OMG. *UML Testing Profile Version 1.0*. 2005. 113 p. Acesso em outubro de 2011. Disponível em: <http://www.omg.org/cgi-bin/doc?formal/05-07-07>.

[Pezze e Young 2007]PEZZE, M.; YOUNG, M. *Software Testing and Analysis: Process, Principles and Techniques*. [S.l.]: Wiley, 2007. ISBN 0471455938.

[Python 2011]PYTHON. 2011. Acesso em novembro de 2011. Disponível em: <http://python.org/>.

[Python REST Client 2011]PYTHON REST Client. 2011. Acesso em outubro de 2011. Disponível em: <http://code.google.com/p/python-rest-client/>.

[PyUnit 2011]PYUNIT. 2011. Acesso em novembro de 2011. Disponível em: <http://pyunit.sourceforge.net/>.

[REST Console 2011]REST Console. 2011. Acesso em outubro de 2011. Disponível em: <http://www.restconsole.com>.

[Richardson e Ruby 2007]RICHARDSON, L.; RUBY, S. *Restful Web Services*. Beijing: O'Reilly Media, 2007. ISBN 0596529260.

[Webber e Parastatidis 2010]WEBBER, J.; PARASTATIDIS, S. *REST in Practice: Hypermedia and Systems Architecture*. [S.l.]: O'Reilly Media, 2010. ISBN 0596805829.