# 3
# O Sistema: SetFramework

O sistema desenvolvido é composto de duas partes principais, um gerador de classes e um ambiente de programação que utiliza e interpreta a DSL desenvolvida. Também foram implementados três exemplos de uso pra demonstrar o poder das novas operações propostas e a utilidade da DSL criada aplicada interfaces com objetivo de fornecer semântica às ações dos usuários.

## 3.1
## DSL

PUC-Rio - Certificação Digital Nº 0410841/CA

O fator motivador para a implementação desta DSL foi a linguagem para manipulação de conjuntos SET-L. Esta linguagem permite que operações sobre conjuntos sejam realizadas, porém é uma linguagem restrita para alguns tipos primitivos de dados e com métodos específicos a eles. Não há a possibilidade de estender a linguagem e criar novos tipos de dados, classes e relações.

É importante definir o que é uma DSL. DSL é a sigla de *Domain-Specific Language* (DSL e DE METEUR, 2003). É uma linguagem de programação que oferece, através de notações apropriadas e abstrações, um grande poder de programação focado em, e normalmente restrito, um problema particular de um domínio específico.

DSLs são normalmente pequenas, oferecendo apenas um pacote restrito de notações e abstrações. Na literatura também são chamadas de micro-linguagens. Algumas vezes, entretanto, elas contêm o que é chamado de GPL (General-Purpose Language) como um sub-linguagem, oferecendo assim um poder expressivo no domínio específico.

DSLs são geralmente declarativas. Conseqüentemente, elas podem ser vistas como linguagens específicas, assim como linguagens de programação. Muitas DSLs são compatíveis com um compilador de DSL o qual gera aplicações a partir de programas DSL. Nesse caso, o compilador é referido como um gerador de aplicação na literatura e a DSL como uma linguagem específica da aplicação.

Relacionada com programação em um domínio específico está a programação feita pelo usuário final (EUP), a qual acontece quando usuários finais realizam tarefas simples de programação usando macro ou linguagem de script. Um exemplo típico é a programação usando a linguagem de macro do Excel.

Uma DSL pode ser escrita através da construção de um interpretador para interpretar a DSL ou então ser escrita estendendo uma linguagem base. Esse último método foi utilizado nessa dissertação.

Podemos encontrar diversas vantagens de se utilizar uma DSL (DSL), como:

- Programas que utilizam uma DSL são concisos, de certa forma auto-documentáveis e podem ser utilizados para diferentes propósitos.
- DSLs aumentam a produtividade, confiabilidade, manutenção e portabilidade.
- DSLs têm o conhecimento do domínio, e assim permitem a conservação e o reuso desse conhecimento.
- DSLs permitem a validação e otimização a nível do domínio.
- DSLs aumentam a testabilidade seguindo abordagens como por exemplo (SIRER, 1999).
- DSLs tornam mais fácil que uma pessoa com conhecimento do domínio utilize as suas diretivas para montar ou projetar um sistema para o mesmo.

A DSL criada nesta dissertação explora o fato de ser auto-documentável, pois torna-se fácil compreender as operações nela definidas. Aumenta a produtividade, confiabilidade, manutenção e portabilidade, pois por ser um código centralizado, muitos usuários programadores podem utilizar e também pode ser utilizado por diferentes tipos de aplicações. Torna-se mais fácil de manter por ser um repositório de métodos utlizado por diferentes sistemas.

Um outro ponto interessante é o fato de que a validação da DSL pode ser realizada na camada de domínio, ou seja, através do conhecimento do domínio da aplicação em questão é possível validar e verificar a corretude dos métodos.

PUC-Rio - Certificação Digital Nº 0410841/CA

De acordo com as operações idealizadas e o modelo definido, foi criada uma DSL, voltada para o domínio de conjuntos, com o objetivo principal de suprir as necessidades de operações que podem ser solicitadas por um sistema que está utilizando o modelo de informação aqui citado.

## 3.2
## Usuários do sistema

O sistema possui dois tipos distintos de usuários, o usuário programador e o usuário final. Ambos devem ter conhecimento do domínio da aplicação em questão e do problema que está sendo solucionado.

O usuário programador é quem define as classes da aplicação que estarão compondo o domínio e também é o responsável por definir a DSL. Opcionalmente, este usuário pode alterar as classes geradas.

O usuário final é o usuário que utiliza a aplicação final que foi construída através do uso do modelo e da DSL. As interações que ele realiza na interface são mapeadas em comandos e métodos do modelo e da DSL.

## 3.3
## Descrição do sistema

Primeiramente foi desenvolvido o módulo de gerador de classes (Figura 4). O usuário programador deve definir as classes em RDFS que deseja e que no futuro seus objetos irão formar os conjuntos, sendo representados, então, como elementos.

Após feita essa definição, esse módulo, que na realidade funciona em modo *console*, deve ser chamado especificando qual outro módulo estará utilizando as novas classes definidas. As classes são interpretadas a partir da definição feita no RDFS e geradas automaticamente para o usuário programador. Todos os membros e métodos relevantes são considerados nessa fase onde cada classe é gerada do modo mais completo possível para auxiliar o desenvolvedor a usufruir da DSL.

Então, o ambiente que recebe tais classes está pronto para que seja iniciado o desenvolvimento. O usuário deve definir os elementos que deseja estar utilizando como informação/dados do sistema de acordo com as suas classes definidas anteriormente. O usuário programador utilizará nessa etapa todas as

PUC-Rio - Certificação Digital Nº 0410841/CA

peculiaridades do sistema para desenvolver a solução final, que por sua vez terá suas informações mapeadas em elementos e conjuntos e utilizará as novas operações para conjuntos propostas.

Paralelamente, foi desenvolvida uma aplicação Web de exemplo para demonstrar as funcionalidades criadas.

Uma segunda aplicação Web foi implementada como um gerador de sistemas baseados em navegação facetada, ou seja, dado um repositório de objetos que formam as facetas (conjuntos) e itens (elementos), uma interface é gerada automaticamente para a navegação em forma de facetas organizadas hierarquicamente.

Finalmente, foi gerada uma aplicação decorrente da integração de dois módulos, a interface e o modelo. Esta aplicação tem como objetivo mostrar as possíveis funcionalidades oferecidas pelo modelo, o seu poder de resolver solicitações feitas pelo usuário na interface e sua capacidade de integração com outros sistemas para ditar a semântica de ações realizadas nas interfaces.

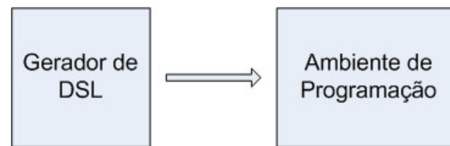


Figura 4: Modelos do Sistema

PUC-Rio - Certificação Digital Nº 0410841/CA

# 3.4 Especificação

## 3.4.1 Gerador de DSL

O usuário deve primeiramente definir o RDFS com as classes desejadas, e futuramente seus objetos serão interpretados como elementos de possíveis conjuntos.

Vejamos um exemplo do arquivo *classes.rdfs*. Nesse exemplo estamos definindo duas classes: *Vinho* e *Faceta*, onde cada vinho está associado a uma faceta.

```
<?xml version='1.0'?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
         xmlns:set="http://www.tecweb.inf.puc-rio.br/types/set#">
```

Quadro 1: Namespaces do RDFS de definição das classes

Primeiramente são definidos os *namespaces* que serão utilizados no schema. Neste caso foram definidos três, sendo os dois primeiros padrões já definidos com suas propriedades para RDF e para RDFS e o último um *namespace* específico definido para a aplicação.

Em seguida são definidas as classes.

```
    <rdf:Description rdf:about="http://www.tecweb.inf.puc-rio.br/ Class#Vinho">
        <rdf:type rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
    </rdf:Description>
    <rdf:Description rdf:about="http://www.tecweb.inf.puc-rio.br/Class#Faceta">
        <rdf:type rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
    </rdf:Description>
```

Quadro 2: Classes do RDFS de definição das classes

No atributo *descrição* definimos o *nome* da classe e no atributo type definimos o *tipo*, neste caso dizemos que é uma classe em si.

Posteriormente devemos definir as propriedades que irão formar a classe.

PUC-Rio - Certificação Digital Nº 0410841/CA

```
<rdf:Description rdf:about="http://www.tecweb.inf.puc-rio.br/Property#titulo">
    <rdf:type rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property"/>
    <set:varType rdf:resource="http://www.tecweb.inf.puc-rio.br/types/set#string"/>
    <rdfs:domain rdf:resource="http://www.tecweb.inf.puc-rio.br/Class#Vinho"/>
</rdf:Description>
```

Quadro 3: Propriedades do RDFS de definição das classes

Nesse exemplo (Quadro 3) temos que o nome da propriedade é *título* e também que é um nó do *tipo* propriedade. É necessário também definirmos o *tipo da variável* que nesse caso é uma string e o seu *domínio* para sabermos a qual classe essa propriedade pertence.

Essa definição é de um campo simples de uma classe, mas podemos fazer algo mais elaborado como, por exemplo, referenciar uma propriedade da classe à própria classe.

```
<rdf:Description rdf:about="http://www.tecweb.inf.puc-rio.br/Property#facetaPai">
    <rdf:type rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property"/>
    <set:varType rdf:resource="http://www.tecweb.inf.puc-rio.br/types/set#Faceta"/>
    <rdfs:domain rdf:resource="http://www.tecweb.inf.puc-rio.br/Class#Faceta"/>
</rdf:Description>
```

Quadro 4: Propriedades do RDFS de definição das classes

Aqui dizemos que o *título* da propriedade é facetaPai e que o *tipo de variável* é a própria classe Faceta a qual essa propriedade pertence, podemos ver isso pelo atributo *domínio*.

Podemos ainda definir uma operação inversa, como por exemplo, uma propriedade na classe *Faceta* que seja do tipo *Vinho*. Automaticamente o gerador de classes irá gerar um método associado para retornar todos os vinhos de uma determinada faceta.

Segue o exemplo:

PUC-Rio - Certificação Digital Nº 0410841/CA

```
    <rdf:Description                         rdf:about="http://www.tecweb.inf.puc-
rio.br/Property#inverted_classificaVinhos_facetas">
        <rdf:type          rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-
ns#Property"/>
        <set:varType                      rdf:resource="http://www.tecweb.inf.puc-
rio.br/types/set#Vinho"/>
        <rdfs:domain                      rdf:resource="http://www.tecweb.inf.puc-
rio.br/Class#Faceta"/>
    </rdf:Description>
```

Quadro 5: Propriedades do RDFS de definição das classes

Neste caso estamos definindo que o *tipo* dessa propriedade da classe *Faceta* é da classe *Vinho*. E através do nome, que deve conter no seu início a palavra *inverted*, o gerador sabe que terá que criar um método para retornar os objetos de acordo com a definição feita.

Tendo a especificação das classes definida, o próximo passo é executar o arquivo que irá gerar as classes em C# para posterior uso do programador. É necessário passar como parâmetro o caminho onde se encontra a solução final no Visual Studio, o nome do projeto onde as classes serão incluídas automaticamente depois de sua criação e o caminho onde se encontra o arquivo RDFS.

Após a execução especificada acima teremos as classes geradas e então seguimos com a definição dos métodos criados. Vamos tomar como exemplo a classe *Vinho*.

PUC-Rio - Certificação Digital Nº 0410841/CA

***Membros***

```
private string titulo;
private int ano;
private int numeroserie;
private string descricao;
private Faceta[] facetas;
```

Quadro 6: Membros da classe vinho

Foram definidos os cinco membros acima no RDFS de definição das classes e estes foram gerados pelo gerador de classes. O membro *facetas* é o único caso a parte pois é um vetor de objetos do tipo *Faceta*, sendo assim um *vinho* pode estar associado a várias *facetas*.

***Construtores***

```
public Faceta()
{
}

public Faceta(string titulo, int ano, int numeroserie, string descrição,
Faceta[] facetaPai)
{
     this.titulo = titulo;
     this.ano = ano;
     this.numeroserie = numeroserie;
     this.descricao = descricao;
     this.facetaPai = facetaPai;
}
```

Quadro 7: Construtores da classe

Sempre são criados dois tipos de construtores, o primeiro sem nenhum parâmetro e o segundo com todos os parâmetros possíveis, isto é, dando a possibilidade do usuário definir todos os membros do objeto no momento em que estiver sendo instanciado.

PUC-Rio - Certificação Digital Nº 0410841/CA

***Getters e Setters***

```
public string Titulo
{
      get {
            return titulo;
      }
      set {
            titulo = value;
      }
}
```

Quadro 8: Getter e Setter da classe

São definidos todos os *getters* e *setters* para cada tipo de membro da classe. Neste exemplo, para o *Titulo*. Temos uma exceção quando o campo é do tipo de uma outra classe, como por exemplo, o campo *facetas*. Neste caso mais um tipo de *getter* é definido e retorna um objeto do tipo *Set* que é o tipo de objeto padrão da aplicação para a representação de um conjunto. O tipo de objeto *Set* define um conjunto, dessa forma o novo *getter* gerado retorna um conjunto. Isso torna mais completa a classe, disponibilizando as *facetas* de um *vinho* em forma de um conjunto, já estando assim pronto para uma possível manipulação.

***Métodos***

Os três primeiros métodos gerados são definidos com três tipos diferentes de assinatura. Primeiro, é preciso definir o campo e o valor pelo qual se deseja fazer a busca no repositório; também com a possibilidade de usar também uma configuração para ordenação, sendo assim, por exemplo, pode-se escolher que um determinado conjunto (objeto retornado por estes métodos) resultante já esteja ordenado por um determinado membro da classe e por último deve-se definir uma expressão no formato ((titulo = ‘Exemplo’) OR (ISBN = 123456)) como parâmetro. Essa expressão pode ser formulada com quantos ORs e ANDs o usuário desejar, a única restrição é que a mesma deve estar parentetisada corretamente. Abaixo seguem os principais métodos gerados:

PUC-Rio - Certificação Digital Nº 0410841/CA

- `findFirstBy` – encontra o primeiro objeto de acordo os parâmetros passados. É feita uma busca no repositório por todos os objetos do tipo em questão e é retornado o primeiro da lista.

```
Vinho vinho = Vinho.findFirstBy("região", "Sul", "preço", "ASC");
```

Quadro 9: Exemplo do método findFirstBy

Neste caso o método retorna o primeiro elemento vinho que seja da região Sul ordenado por preço em ordem ascendente. Um exemplo de utilização dessa chamada seria um usuário final fazendo uma consulta pelo vinho da região Sul com menor preço. É possível também passar qual o tipo de ordenação que deseja ser feita na consulta, indicando por qual campo ela deve ser realizada. Desta forma, retornará o primeiro elemento que atenda a essa restrição também. E como foi mencionado acima é possível entrar como parâmetro uma expressão que será interpretada e funcionará como mais uma restrição para a busca.

- `findSetBy` – encontra um conjunto de objetos de acordo com um os parâmetros passados. É feita uma busca no repositório por todos os objetos do tipo em questão e retornado um conjunto com todos os objetos encontrados.

```
Set conjunto_de_vinhos = Vinho.findSetBy("faceta.titulo", "Cabernet");
```

Quadro 10: Exemplo do método findSetBy

Neste caso estamos buscando pelos vinhos que sejam do tipo Cabernet, que é identificado como uma faceta. Assim como o método especificado acima, é possível definir um campo para ordenação e o tipo e ainda uma expressão.

PUC-Rio - Certificação Digital Nº 0410841/CA

- findBy – funciona exatamente como o método acima, com a pequena diferença de que é retornado um vetor contendo os objetos do tipo em questão e não um conjunto.

```
Vinho[] array_de_vinhos = Vinho.findBy("faceta.titulo", "Cabernet");
```

Quadro 11: Exemplo do método findBy

O conceito de ordenação também está contemplado neste método.

Quando uma classe em questão tem associação com outra, como por exemplo, um objeto *Vinho* que tem associação com um objeto *Faceta*, os métodos abaixo são gerados. Um objeto vinho tem um atributo chamado Facetas que é representado por uma lista de facetas para o vinho.

- add<nome_da_classe> – relaciona um novo objeto ao objeto em questão. Isto torna mais completa a classe, permitindo que sejam feitas alterações em um objeto além de buscas e manipulações sobre os conjuntos.

```
Faceta novaFaceta = new Faceta("faceta teste");
vinho1.addFaceta(novaFaceta);
```

Quadro 12: Exemplo do método add

Neste caso estamos associando uma nova faceta para um objeto vinho. Este objeto já existia no repositório e em algum momento foi necessária esta associação com uma nova faceta.

- remove<nome_da_classe> – remove o relacionamento entre dois objetos.

```
vinho1.removeFaceta(novaFaceta);
```

Quadro 13: Exemplo do método remove

PUC-Rio - Certificação Digital Nº 0410841/CA

No exemplo acima estamos removendo a faceta que acabou de ser adicionada, desassociando assim o vinho desta faceta.

### 3.4.2
### Ambiente de programação

O ambiente de programação recebe automaticamente as classes geradas para que o usuário programador possa utilizá-las para o desenvolvimento da aplicação final.

O primeiro passo a ser dado nesta etapa é definir os elementos que estarão contidos no repositório. Para isso o usuário deve definir um arquivo RDF formado por todos os objetos do domínio em questão. Esta definição deve ser realizada de acordo com as classes especificadas.

O arquivo RDF começa com a definição dos *namespaces* que serão utilizados.

```
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:elem="http://www.tecweb.inf.puc-rio.br/vocabularies/elem#"
         xml:base="http://www.tecweb.inf.puc-rio.br/objects">
```

Quadro 14: Namespaces do RDF de definição dos elementos

O *namespace* **elem** é único personalizado para a aplicação, os outros dois são padrões já existentes com seus próprios vocabulários.

Segue-se então para a definição dos elementos em si. Neste exemplo será mostrada a definição de tipos de vinho, que aqui são chamadas de *facetas* e também dos *vinhos* que estão associados às *facetas*.

```
. . .

    <rdf:Description rdf:about="Elemento1">
        <elem:Class>Faceta</elem:Class>
        <elem:Titulo>Varietal</elem:Titulo>
        <elem:FacetaPai></elem:FacetaPai>
    </rdf:Description>

    <rdf:Description rdf:about="Elemento2">
        <elem:Class>Faceta</elem:Class>
        <elem:Titulo>Red Wine</elem:Titulo>
        <elem:FacetaPai rdf:resource="www.tecweb.inf.puc-
rio.br/objects#Elemento1"/>
    </rdf:Description>

    <rdf:Description rdf:about="Elemento3">
        <elem:Class>Faceta</elem:Class>
        <elem:Titulo>White Wine</elem:Titulo>
        <elem:FacetaPai rdf:resource="www.tecweb.inf.puc-
```

PUC-Rio - Certificação Digital Nº 0410841/CA

```
rio.br/objects#Elemento1"/>
    </rdf:Description>

    <rdf:Description rdf:about="Elemento4">
        <elem:Class>Faceta</elem:Class>
        <elem:Titulo>Merlot</elem:Titulo>
        <elem:FacetaPai rdf:resource="www.tecweb.inf.puc-
rio.br/objects#Elemento2"/>
    </rdf:Description>
. . .
```

Quadro 15: Definição dos elementos *facetas* no RDF

No trecho de código acima, todos os elementos são da classe *Faceta* e por sua vez são elementos que definem *facetas*. Nota-se também que o *Elemento1* (Varietal) não tem pai, sendo assim o primeiro elemento na hierarquia de *facetas*. O *Elemento2* (Red Wine) tem como pai o *Elemento1* assim como o *Elemento3* (White Wine). E por fim o *Elemento4* (Merlot) é filho do *Elemento2*.

Os campos a serem definidos neste momento são exatamente os campos que foram definidos para a classe em questão, *Faceta*.

Visualmente teríamos uma organização do tipo:

```
. . .
    Varietal (Elemento1)
    └ Red Wine (Elemento2)
      └ Merlot (Elemento4)
    └ White Wine (Elemento3)
. . .
```

Quadro 16: Exemplo de organização hierárquica das *facetas*

Pode-se também criar um outro bloco de organização, basta criar um elemento da classe *Faceta* sem possuir uma faceta pai e criar outros elementos abaixo dele na hierarquia, exatamente como foi feito no exemplo acima.

Criadas as *facetas*, devemos definir os *vinhos*, como por exemplo:

```
    <rdf:Description rdf:about="Elemento23">
        <elem:Class>Vinho</elem:Class>
        <elem:Titulo>V1</elem:Titulo>
        <elem:Ano>1999</elem:Ano>
        <elem:NumeroSerie>1</elem:NumeroSerie>
        <elem:Descricao>ipsum loren doren lores</elem:Descricao>
        <elem:Categoria rdf:resource="www.tecweb.inf.puc-
rio.br/objects#Elemento4"/>
        <elem:Categoria rdf:resource="www.tecweb.inf.puc-
rio.br/objects#Elemento15"/>
        <elem:Categoria rdf:resource="www.tecweb.inf.puc-
rio.br/objects#Elemento22"/>
```

PUC-Rio - Certificação Digital Nº 0410841/CA

```
    </rdf:Description>

    <rdf:Description rdf:about="Elemento24">
        <elem:Class>Vinho</elem:Class>
        <elem:Titulo>V2</elem:Titulo>
        <elem:Ano>2000</elem:Ano>
        <elem:NumeroSerie>2</elem:NumeroSerie>
        <elem:Descricao>lorem ipsum loris dorem</elem:Descricao>
        <elem:Categoria rdf:resource="www.tecweb.inf.puc-
rio.br/objects#Elemento3"/>
        <elem:Categoria rdf:resource="www.tecweb.inf.puc-
rio.br/objects#Elemento15"/>
    </rdf:Description>
```

Quadro 17: Exemplo de definição de *vinhos*

Como na definição de *facetas*, os campos a serem definidos são os mesmo que formam a classe *Vinho*. O *Elemento23*, que tem como título *V1*, está associado aos elementos 4, 15 e 22. O *Elemento4* foi definido anteriormente como sendo *Merlot*, ou seja, esse vinho está classificado como sendo do tipo *Merlot*. As outras duas associações poderiam ser a faixa de preço e a região do vinho respectivamente.

Todos os elementos que foram definidos neste arquivo RDF são carregados para o repositório interno do ambiente de programação para que o usuário possa eventualmente realizar consultas e manipular conjuntos presentes no mesmo.

PUC-Rio - Certificação Digital Nº 0410841/CA

O usuário pode, então, executar operações como:

```
[1]Set.Set conjunto1 = Vinho.findSetBy("titulo", "V15");

[2]Set.Set conjunto2 = Vinho.findSetBy("facetas.titulo", "Cabernet", "titulo",
"ASC");

[3]Set.Set conjunto3 = Vinho.findSetBy("((numeroserie = 2) OR (facetas.titulo =
'Burgundy'))");

[4] Set.Set conjuntoFacetas = sets.ExecuteQuery("SELECT FROM Faceta WHERE
((titulo = 'Varietal') OR (titulo = 'Region') OR (titulo = 'Price')) ORDER BY
titulo ASC");
```

Quadro 18: Exemplos de obtenção de conjuntos

No exemplo [1] o usuário define um conjunto chamado de *conjunto1* que é formado por todos os elementos *vinho* que tenham o campo *título* com o valor *V15*. Já em [2] é definido um *conjunto2* que é formado por todos os elementos *vinho* que estejam associados à *faceta* Cabernet e este conjunto terá seus elementos ordenados pelo campo *titulo* de maneira *ascendente*. Neste versão, só é possível realizar a ordenação por apenas um campo.

Nesta versão do sistema, a ordenação só pode ser realizada através de um único campo.

O *conjunto3* [3] está composto por todos os *vinhos* que têm o campo *número de série* com o valor 2 *ou* que estão associados à categoria Burgundy. Da mesma forma poderíamos utilizar o operador AND ao invés de OR.

O último exemplo, e talvez o mais poderoso, utiliza a definição de uma *query* com a sintaxe de SQL para obtenção do conjunto. O *conjuntoFacetas* [4] é definido dessa forma, onde são buscadas todas as *facetas*, pois a cláusula *FROM* está sendo feita em objetos do tipo *Faceta*, que tenham como *título* os valores *Varietal* ou *Region* ou *Price*.

Estes exemplos são apenas algumas das formas de se obter um conjunto a partir do repositório.

Tendo formados os conjuntos desejados é possível realizar operações entre eles.

PUC-Rio - Certificação Digital Nº 0410841/CA

```
[1]Set.Set conjuntoResposta = conjunto1.Uniao(conjunto2);

[2]Set.Set conjuntoResposta = sets.Intersecao(conjunto1, conjunto2);

[3]Set.Set conjuntoResposta = sets.Diferenca(conjunto1, conjunto2);

[4]Set.Set conjuntoFacetasDeUmVinho = vinho1.AllFacetas;

[5]Set.Set conjuntoVinhosDeUmaFaceta = faceta1.ClassificaVinhos;
```

Quadro 19: Exemplos de operações entre conjuntos

No exemplo [1] o *conjuntoResposta* está sendo formado pela *união* entre o *conjunto1* e o *conjunto2*. O mesmo ocorre para os exemplos [2] e [3], porém as operações nestes casos são *interseção* e *diferença* respectivamente.

O exemplo [4] mostra que tendo posse de um elemento *vinho* (*vinho1*) pode-se obter todas as facetas a que ele está associado. Inversamente, em [5] onde tendo posse de um elemento *faceta*, pode-se obter todos os *vinhos* associados.

O usuário programador também pode definir conjuntos através de um RDF criado previamente. Caso o usuário tenha a informação de conjuntos que serão necessários para sua aplicação, podem ser definidos e já estar disponíveis na aplicação.

Vejamos o exemplo:

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:set="http://www.tecweb.inf.puc-rio.br/vocabularies/set#">

    <rdf:Description rdf:about="Conjunto1">
        <set:Title>Todos os vinhos do tipo Cabernet</set:Title>
        <set:Query>SELECT     FROM     Vinho     WHERE     (faceta.titulo     =
'Cabernet')</set:Query>
    </rdf:Description>

    <rdf:Description rdf:about="Conjunto2">
        <set:Title>Todos as facetas</set:Title>
        <set:Query>SELECT FROM Faceta</set:Query>
    </rdf:Description>

</rdf:RDF>
```

Quadro 20: Exemplos de definição de conjuntos

PUC-Rio - Certificação Digital Nº 0410841/CA

O *Conjunto1* representa todos os vinhos que são do tipo Cabernet. Para isto é necessário definir uma *query* que selecione os elementos desejados. O mesmo se aplica para o *Conjunto2* que abrange todos os elementos do tipo *faceta*.

Após essa etapa o usuário programador, no ambiente de programação, pode utilizar uma funcionalidade da classe *Sets* para carregar os conjuntos já definidos em RDF na memória e então realizar as operações sobre os mesmos.

```
Set.Set conjunto1 = sets.getSetByName("Conjunto1");
Set.Set conjuntoResposta = conjunto1.Uniao(conjunto2);
```

Quadro 21: Exemplos de utilização do método getSetByName

Deve-se passar como parâmetro o nome do conjunto e é retornado o objeto conjunto com seus respectivos elementos e que pode ser manipulado posteriormente.

## 3.5 Arquitetura de Implementação

Nesta seção será apresentada a arquitetura detalhada do sistema desenvolvido.

### 3.5.1 Diagrama de Arquitetura do Sistema

A arquitetura do sistema é representada pelo diagrama que segue (Figura 4). A camada de apresentação do sistema, no lado do cliente, possui uma interface web que é renderizada pelo browser. É nesta camada que será utilizado o modelo MVC que é explicado mais adiante. Esta camada realiza consultas via HTTP no servidor Web que se comunica com a camada de negócios realizando requisições.

A camada de negócios interage com a camada de dados, onde está o repositório central, enviando consultas necessárias para resolução de respostas a serem passadas para o servidor Web. A camada de dados responde a consulta solicitada e a camada de negócios envia a resposta calculada pela DSL para o servidor Web que então organiza as informações que serão retornadas ao usuário final via browser.

PUC-Rio - Certificação Digital Nº 0410841/CA

Figura 5: Arquitetura do Sistema

PUC-Rio - Certificação Digital Nº 0410841/CA

### 3.5.2
### Diagrama de Pacotes

Em muitos casos um único diagrama de classes pode ser grande demais para representar todo o sistema. Assim, é conveniente utilizarmos um elemento para organizar os subsistemas do modelo. Este elemento é chamado de diagrama de pacotes. Um pacote representa um grupo de classes (ou outros elementos) que se relaciona com outros pacotes através de uma relação de dependência. Um diagrama de pacotes pode ser utilizado em qualquer fase do processo de modelagem e visa organizar os modelos.

Abaixo temos o diagrama de pacotes para a aplicação desenvolvida. Primeiramente para o módulo gerador de classes e depois para o ambiente de programação utilizado pelo usuário programador.

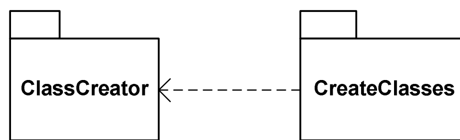


Figura 6: Diagrama de Componentes – módulo gerador

No diagrama acima temos dois pacotes que são utilizados no módulo gerador de classes de forma conjunta para que as classes possam ser criadas no final do processo.

O pacote *CreateClasses* contém a classe *Application* que é responsável por toda manipulação das classes, que compõe o pacote *ClassCreator*, para a criação das classes a partir do arquivo RDF de definição. Dizemos então que o primeiro pacote depende do segundo, pois utiliza as suas classes para realizar suas operações.

PUC-Rio - Certificação Digital Nº 0410841/CA

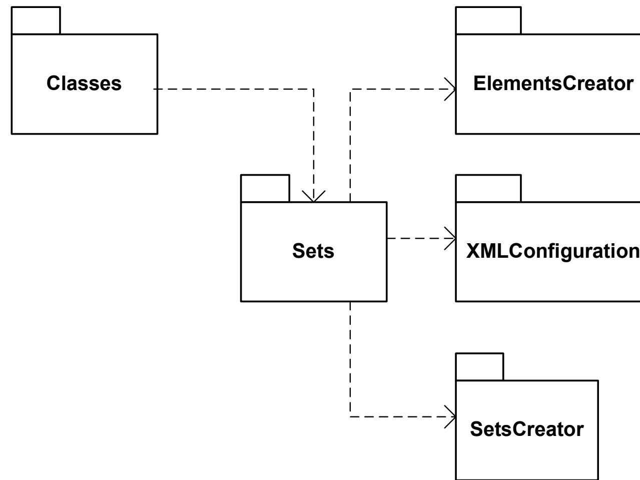


Figura 7: Diagrama de Componentes – ambiente de programação

Os componentes do diagrama acima compõem o ambiente de programação. Estes pacotes devem ser utilizados pelo usuário programador para que possa interagir nos conjuntos.

O pacote *Classes* contém as classes geradas pelo módulo gerador, que tem dependência com o pacote *Sets*, que é responsável pela manipulação das informações armazenadas no repositório de dados. Este componente por sua vez depende do *XMLConfiguration* e do *SetsCreator* que respectivamente desempenham os papeis de leitor das configurações necessárias para o funcionamento e leitura das informações definidas em RDF e criador de instâncias de objetos conjunto descritos pelo usuário, também em RDF.

O componente *ElementsCreator* também é utilizado pelo *Sets*. Este pacote é formado por três classes que são responsáveis pelo gerenciamento dos elementos. As classes exercem funções de leitura das informações em RDF, criação dos elementos no repositório de dados e representação de um objeto elemento.

PUC-Rio - Certificação Digital Nº 0410841/CA

### 3.5.3
### Definição das Classes

Seguem as definições das classes, onde podemos visualizar seus nomes, membros e métodos detalhados. Começaremos pelas classes do módulo gerador e posteriormente as classes do ambiente de programação.

***RDF***

| **RDF** |
| --- |
| -fileName : string = ""<br>-strString : string = "string"<br>-strInt : string = "int" |
| +RDF(in fileName : string)<br>+getClassDefinitions() : ArrayList<br>-getClasses(in store : Store) : Store<br>-getPropertiesByObject(in store : Store, in entityObject : Entity) : Store<br>-getPropertiesBySubject(in store : Store, in entitySubject : Entity) : Store<br>-transformText(in text : string) : string |

Figura 8: Classe RDF

PUC-Rio - Certificação Digital Nº 0410841/CA

A classe RDF é utilizada para a leitura do arquivo de definição das classes em RDF Schema. Possui um construtor que recebe o caminho físico do arquivo e métodos para manipulação do arquivo, como *getClasses*, *getPropertiesByObject* e *getPropertiesBySubject* que fazem consultas no repositório para obterem dados relevantes para geração de um objeto que representa uma instância da classe.

O método *getClassDefinitions* retorna uma lista com todas as instâncias da classe que foram definidas pelo usuário.

***ClassDefinition***

| **ClassDefinition** |
| --- |
| -className : string = ""<br>-attributeNames : ArrayList = new ArrayList()<br>-attributeTypes : ArrayList = new ArrayList() |
| +ClassDefinition()<br>+ClassDefinition(in className : string)<br>+ClassDefinition(in className : string, in attributeNames : ArrayList, in attributeTypes : ArrayList)<br>+ClassName() : string<br>+AttributeNames() : ArrayList<br>+AttributeTypes() : ArrayList |

Figura 9: Classe ClassDefinition

A classe acima representa uma classe definida no arquivo RDFS. Possui três membros que são utilizados para esta representação. São eles: *className*, que é o nome da classe, *attributeNames*, que são os nomes dos atributos da classe, e *attributeTypes*, que são os tipos dos atributos da classe.

***FileContentManagement***

| **FileContentManagement** |
|---|
| -projectGuid : string = "" |
| +FileContentManagement()<br>+IncludeNewClassInProjectFile(in solutionPath : string, in projectName : string, in arrClassNames : ArrayList)<br>+IncludeNewProjectInSolutionFile(in solutionPath : string, in projectName : string)<br>+UpdateNewProjectGUID(in solutionPath : string, in projectName : string)<br>-CreateProjectFile(in filePath : string, in projectName : string) |

Figura 10: Classe FileContentManagement

Esta classe realiza as manipulações nos arquivos da solução do usuário programador para que as classes geradas sejam incluídas corretamente em seu projeto.

***GenerateClass***

| **GenerateClass** |
|---|
| -strString : string = "string"<br>-strSystemString : string = "System.String"<br>-strInt : string = "int"<br>-strSystemInt32 : string = "System.Int32" |
| +GenerateClass()<br>+GenerateClassFile(in classDefinition : ClassDefinition, in projectPath : string)<br>-firstLetterToUpperCase(in word : string) : string |

Figura 11: Classe GenerateClass

A classe GenerateClass é responsável pela geração de uma classe. O método *GenerateClassFile* recebe como parâmetro uma instância que representa uma classe, que é do tipo *ClassDefinition*, e então gera o arquivo da classe.

PUC-Rio - Certificação Digital Nº 0410841/CA

***Application***

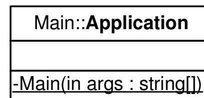


Figura 12: Classe Application

A classe Application é compilada como um executável que recebe três parâmetros e utiliza a classe *GenerateClass* para gerar os arquivos das classes. Os parâmetros são: *solutionPath* que representa o caminho físico onde se encontra a solução na qual as classes serão inseridas, *projectName* que indica o nome do pacote que irá englobar as classes geradas e *rdfFilePath* que é o caminho físico do arquivo RDFS de definição das classes.

Abaixo as classes do ambiente de programação:

***Element***

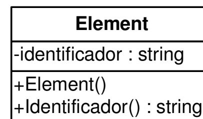


Figura 13: Classe Element

A classe Element representa os elementos que compõem os conjuntos. Todas as classes que são utilizadas na aplicação final devem herdar desta classe, pois é ela que define o atributo identificador, que deve estar presente nos objetos para que tenham uma identificação única e que possam ser posteriormente mapeados para qualquer tipo de interface.

PUC-Rio - Certificação Digital Nº 0410841/CA

***Set***

| Set |
|---|
| -identificador : string<br>-titulo : string<br>-elementos : ArrayList |
| +Identificador() : string<br>+Titulo() : string<br>+Elementos() : ArrayList<br>+Set()<br>+Set(in titulo : string, in elementos : ArrayList)<br>+sort(in comparer : IComparer) : Set<br>+Uniao(in sets : params Set[]) : Set<br>-UniaoAux(in C2 : Set) : Set<br>+Intersecao(in sets : params Set[]) : Set<br>-IntersecaoAux(in C2 : Set) : Set<br>+Diferenca(in sets : params Set[]) : Set<br>-DiferencaAux(in C2 : Set) : Set<br>-PertenceElemento(in elemento : object, in elementos : ArrayList) : bool<br>+MapSet(in f : FunctionMapSet) : Set<br>+Map(in f : FunctionMap) : object<br>+ReturnRelations() : ArrayList<br>+addElement(in element : object)<br>+removeElement(in element : object)<br>+getFirstElement() : object<br>+getElementByPosition(in position : int) : object<br>+getElementByIdentifier(in identifier : string) : object |

Figura 14: Classe Set

A classe Set representa os conjuntos da aplicação. Cada conjunto tem um identificador único, um título e uma lista de elementos, que podem ser de qualquer tipo.

Esta classe possui dois construtores e métodos para manipulação dos conjuntos. O método *sort* recebe um objeto comparador que serve como base para a ordenação dos elementos de um conjunto. Os métodos *União*, *Interseção* e *Diferença* são as operações básicas da teoria de conjuntos, assim como o método *PercenteElemento*, que indica se um elemento pertence a um conjunto. Há também métodos mais avançados: *MapSet e Map*. Ambos são métodos que esperam uma função como parâmetro. A primeira aplica esta função para todos os membros do conjunto, retornando um novo conjunto, e a segunda aplica para todos os elementos como um todo, retornando um objeto escalar.

Também podemos através desta classe saber quais relações que os objetos de um conjunto mantêm com outros conjuntos através do método *ReturnRelations*. E por fim, temos os métodos *addElement* e *RemoveElement* que são utilizados para adicionar um elemento e remover um elemento respectivamente.

E ainda temos três métodos para retornar apenas um elemento de um dado conjunto. O método *getFirstElement* retorna o primeiro elemento de um conjunto,

PUC-Rio - Certificação Digital Nº 0410841/CA

o *getElementByPosition* retorna o elemento que está em uma dada posição e o *getElementByIdentifier* retorna o elemento que possui um dado identificador.

### *Sets*

| Sets |
| --- |
| -instance : Sets = null<br>-db : ObjectContainer<br>-config : Configuration<br>-DATABASE_FILE : string = @"D:\tese\Project\data.db"<br>-isClosed : bool = false |
| -Sets()<br>+Open(in delete : bool) : Sets<br>+AddElement(in objeto : object)<br>+UpdateElement(in objeto : object)<br>+RemoveElement(in objeto : object)<br>+ExecuteQuery(in strDefinition : string) : Set<br>-ReturnMainToken(in strWhereClauses : string) : string<br>-ReturnConditionsList(in strWhereClauses : string) : ArrayList<br>-AddAndConstraint(in query : Query, in strWhereClauses : string) : Constraint<br>-AddOrConstraint(in query : Query, in strWhereClauses : string) : Constraint<br>-ReturnFilterField(in strClause : string) : string<br>-ReturnFilterValue(in strClause : string) : string<br>-AddConstraint(in query : Query, in strClause : string) : Constraint<br>-OperatorType(in strClause : string) : string<br>-AddOperator(in constr : Constraint, in strOperator : string)<br>+getSetByName(in setName : string) : Set<br>+getSimilarElements(in similarField : string, in fieldValue : string) : Set<br>+Uniao(in sets : params Set[]) : Set<br>-UniaoAux(in C1 : Set, in C2 : Set) : Set<br>+Intersecao(in sets : params Set[]) : Set<br>-IntersecaoAux(in C1 : Set, in C2 : Set) : Set<br>+Diferenca(in sets : params Set[]) : Set<br>-DiferencaAux(in C1 : Set, in C2 : Set) : Set<br>-PertenceElemento(in elemento : object, in elementos : ArrayList) : bool<br>+Close()<br>+DeleteDataBase()<br>+ReturnRelationSet(in obj : object, in relationName : string) : Set<br>+MapIn(in sourceSet : Set, in targetSet : Set, in relationName : string, in firstOperation : string, in secondOperation : string) : Set<br>+getSetByIdentifier(in identifier : string) : Set<br>+getElementByIdentifier(in identifier : string) : Element |

Figura 15: Classe Sets

A classe Sets é uma classe *singleton*, que é responsável pelo armazenamento e manipulação do repositório de elementos. É também utilizada para interpretar alguns comandos específicos aos conjuntos.

Encontramos métodos para adicionar, atualizar e remover elementos no repositório, *AddElement*, *UpdateElement* e *RemoveElement* respectivamente.

É nesta classe que temos o método *ExecuteQuery*, que é responsável por executar uma consulta passada como parâmetro. Esta consulta pode ser construída dinamicamente, pois a linguagem para defini-la é similar à linguagem do SQL, onde a *query* pode ser formada por composição de operações.

Também é possível carregar um conjunto já definido previamente pelo usuário através do método *getSetByName*, onde o nome do conjunto é passado como parâmetro e é retornado um objeto do tipo conjunto que poderá ser utilizado posteriormente para manipulação de seus elementos.

PUC-Rio - Certificação Digital Nº 0410841/CA

O método *getSimilarElements* espera como parâmetros um campo e um valor, e a partir destes dados faz uma busca e retorna todos os elementos similares.

Podemos utilizar as funções de *União, Interseção e Diferença* que foram mencionadas acima. E como esta classe manipula o repositório de elementos, possui métodos para abrir e fechar o mesmo, *Open* e *Close* respectivamente.

Temos o método *ReturnRelationSet* que recebe como parâmetro um objeto e uma relação, e retorna o conjunto correspondente a esta relação e o método *MapIn* que mapeia um tipo de conjunto em outro. Recebe um conjunto de origem, um conjunto destino, a relação que se deseja obter os elementos do conjunto de origem e as operações a serem realizadas entre estes elementos obtidos e a operação a ser executada entre os dois conjuntos do mesmo tipo no final da manipulação.

Também há um método, *getSetByIdentifier*, que dado um identificador é retornado o conjunto correspondente e o método *getElementByIdentifier* que dado um identificador é retornado o elemento correspondente.

PUC-Rio - Certificação Digital Nº 0410841/CA

As classes que seguem dificilmente serão utilizadas direatamente pelo usuário programador. São utilizadas para popular o repositório de elementos a partir do RDF definido pelo usuário.

***ElementsCreator***

| ElementsCreator::**ElementsCreator** |
|---|
| -strString : string = "string"<br>-strSystemString : string = "System.String"<br>-strInt : string = "int"<br>-strSystemInt32 : string = "System.Int32" |
| +ElementsCreator()<br>+createElementObjects(in arrElements : ArrayList) : ArrayList |

Figura 16: Classe ElementsCreator

Esta classe é utilizada para criar os elementos. Recebe como parâmetro uma lista com objetos que definem os elementos e então mapeia e através de reflexão cria os elementos tipados que serão incluídos no repositório. Essa operação é feita através do método *createElementObjects*.

***ElementsRDF***

| ElementsCreator::**ElementsRDF** |
|---|
| -fileName : string = "" |
| +ElementsRDF(in fileName : string)<br>+getElements() : ArrayList<br>-getElementsName(in store : Store) : ArrayList<br>-getElements(in store : Store, in arrElementsName : ArrayList) : ArrayList<br>-transformText(in text : string) : string |

Figura 17: Classe ElementsRDF

A classe ElementsRDF é utilizada para ler o RDF e interpretá-lo de maneira que possa ser gerada uma lista de objetos que definem os elementos que são passados para a classe definida anteriormente.

PUC-Rio - Certificação Digital Nº 0410841/CA

### *Element*

| ElementsCreator::**Element** |
|---|
| -elementName : string = ""<br>-elementClass : string = ""<br>-elementAttributes : ArrayList = new ArrayList()<br>-elementValues : ArrayList = new ArrayList() |
| +Element()<br>+Element(in elementName : string, in elementClass : string, in elementAttributes : ArrayList, in elementValues : ArrayList)<br>+ElementName() : string<br>+ElementClass() : string<br>+ElementAttributes() : ArrayList<br>+ElementValues() : ArrayList |

Figura 18: Classe Element

Esta classe é responsável por representar um elemento. Contém os atributos necessários: *ElementName*, *ElementClass*, *ElementAttributes* e *ElementValues*, que respectivamente, atribuem um nome para o elemento, a classe ao qual este pertence, os seus atributos e os valores correspondentes.

### *XMLConfiguration*

| XMLConfiguration::**XMLConfiguration** |
|---|
| +reader : XmlTextReader |
| +XMLConfiguration()<br>+getElementsRDFPath() : string<br>+getSetsRDFPath() : string |

Figura 19: Classe XMLConfiguration

Esta classe é responsável por ler as configurações de caminho de diretórios feita em um XML. Outras classes utilizam estes caminhos para identificar de onde serão lidos os caminhos dos arquivos RDF de definição.

### *SetsRDF*

| SetsCreator::**SetsRDF** |
|---|
| -fileName : string = "" |
| +SetsRDF(in fileName : string)<br>+getSets(in sets : Sets) : ArrayList<br>-getSetsName(in store : Store) : ArrayList<br>-getSets(in store : Store, in arrSetsNames : ArrayList, in sets : Sets) : ArrayList<br>-transformText(in text : string) : string |

Figura 20: Classe SetsRDF

A classe SetsRDF é responsável pela geração de objetos do tipo conjunto. Esta classe lê o RDF e retorna uma lista de objetos conjunto que foram definidos.

PUC-Rio - Certificação Digital Nº 0410841/CA

As duas classes que seguem foram geradas pelo módulo de geração automático de classes. Ambas foram definidas em RDF previamente. Os métodos presentes nestas classes foram definidos na seção 3.2.1.

| Classes::**Vinho** |
| --- |
| -titulo : string<br>-ano : int<br>-numeroserie : int<br>-descricao : string<br>-facetas : Faceta[] |
| +Vinho()<br>+Vinho(in titulo : string, in ano : int, in numeroserie : int, in descricao : string, in facetas : Faceta[])<br>+Titulo() : string<br>+Ano() : int<br>+Numeroserie() : int<br>+Descricao() : string<br>+AllFacetas() : Set<br>+Facetas() : Faceta[]<br>+findFirstBy(in field : string, in text : string) : Vinho<br>+findFirstBy(in field : string, in text : string, in orderByField : string, in order : string) : Vinho<br>+findSetBy(in field : string, in text : string) : Set<br>+findSetBy(in field : string, in text : string, in orderByField : string, in order : string) : Set<br>+findBy(in field : string, in text : string) : Vinho[]<br>+findBy(in field : string, in text : string, in orderByField : string, in order : string) : Vinho[]<br>+findFirstBy(in field : string, in number : int) : Vinho<br>+findFirstBy(in field : string, in number : int, in orderByField : string, in order : string) : Vinho<br>+findSetBy(in field : string, in number : int) : Set<br>+findSetBy(in field : string, in number : int, in orderByField : string, in order : string) : Set<br>+findBy(in field : string, in number : int) : Vinho[]<br>+findBy(in field : string, in number : int, in orderByField : string, in order : string) : Vinho[]<br>+addFacetas(in facetas : Faceta)<br>+removeFacetas(in facetas : Faceta)<br>+refresh()<br>+findFirstBy(in expression : string) : Vinho<br>+findSetBy(in expression : string) : Set<br>+findBy(in expression : string) : Vinho[] |

Figura 21: Classe Vinho

| Classes::**Faceta** |
| --- |
| -titulo : string<br>-facetaPai : Faceta[] |
| +Faceta()<br>+Faceta(in titulo : string, in facetaPai : Faceta[])<br>+Titulo() : string<br>+AllFacetaPai() : Set<br>+FacetaPai() : Faceta[]<br>+ClassificaVinhos() : Set<br>+findFirstBy(in field : string, in text : string) : Faceta<br>+findFirstBy(in field : string, in text : string, in orderByField : string, in order : string) : Faceta<br>+findSetBy(in field : string, in text : string) : Set<br>+findSetBy(in field : string, in text : string, in orderByField : string, in order : string) : Set<br>+findBy(in field : string, in text : string) : Faceta[]<br>+findBy(in field : string, in text : string, in orderByField : string, in order : string) : Faceta[]<br>+addFacetaPai(in facetaPai : Faceta)<br>+removeFacetaPai(in facetaPai : Faceta)<br>+refresh()<br>+findFirstBy(in expression : string) : Faceta<br>+findSetBy(in expression : string) : Set<br>+findBy(in expression : string) : Faceta[] |

Figura 22: Classe Faceta

PUC-Rio - Certificação Digital Nº 0410841/CA

PUC-Rio - Certificação Digital Nº 0410841/CA

### 3.5.4 Diagrama de Classes

Um diagrama de classes é uma representação da estrutura e relações das classes que servem de modelo para objetos.

É uma modelagem muito útil para o sistema que define todas as classes que o sistema possui e é a base para a construção do diagrama de seqüência.

O diagrama de classe descreve as classes que formam a estrutura do sistema e suas relações. As relações entre as classes podem ser associações, agregações ou heranças. As classes possuem além de um nome, os atributos e as operações que desempenham para o sistema. Uma relação indica um tipo de dependência entre as classes. Essa dependência pode ser forte, como no caso da herança ou da agregação, ou mais fraca como no caso da associação, mas indica que as classes relacionadas cooperam de alguma forma para cumprir um objetivo para o sistema. Nesta seção podemos visualizar todas as classes que podem ser usadas pelo usuário programador.

Abaixo o diagrama de classes para a aplicação desenvolvida.

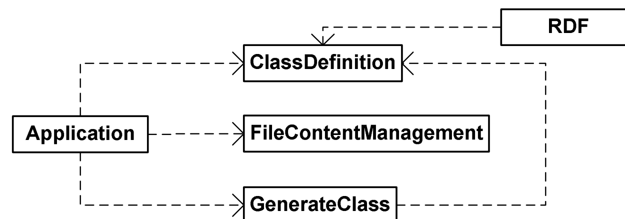


Figura 23: Diagrama de classes do módulo gerador de classes

Este módulo é responsável pela geração das classes a serem utilizadas pelo usuário programador. A classe *Application* é responsável por gerar os arquivos para as classes definidas a partir de um RDF pelo usuário. Está diretamente associada às classes *ClassDefinition*, *FileContentManagement* e *GenerateClass.*

A classe *ClassDefinition* tem como função representar uma classe definida. Ela contém o nome da classe, os nomes de seus atributos e os valores correspondentes para cada atributo. Esta classe faz o mapeamento direto das informações contidas no RDF para uma classe de definição representada em C#. Manipulações são realizadas para que a classe possa ser gerada corretamente para o usuário programador.

A classe *FileContentManagement* é responsável por toda a parte de interação com os arquivos que são gerados e também pela inclusão das classes geradas na solução que o usuário programador irá utilizá-las.

A classe *RDF* lê as informações contidas no arquivo RDF de definição das classes e realizar as manipulações necessárias para então gerar as classes de mapeamento do tipo *ClassDefinition* que serão depois utilizadas pela classe *GenerateClass*.

A classe *GenerateClass* tem como papel gerar as classes. Ela recebe todas as definições das classes em RDF mapeadas para a *ClassDefinition* e monta todos os construtores, membros e métodos necessários para formar a classe.

PUC-Rio - Certificação Digital Nº 0410841/CA

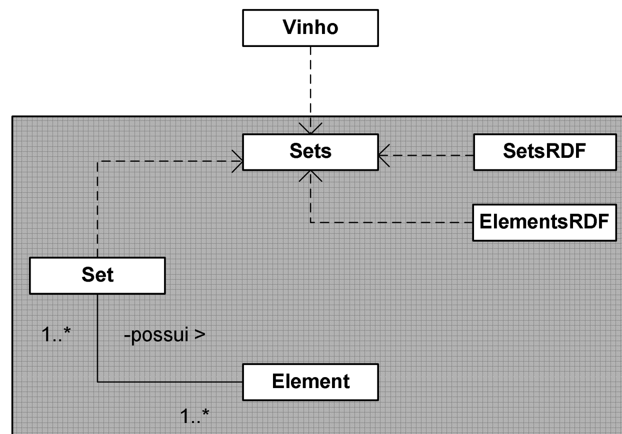


Figura 24: Diagrama de classes do ambiente de programação

O diagrama de classes ilustra o ambiente de programação que é utilizado pelo usuário programador e as classes necessárias para poder realizar futuras interações com conjuntos e elementos.

As classes que estão dentro do retângulo cinza, são as classes do ambiente de programação, a classe *Vinho* é uma classe que faz parte da DSL. A classe *Vinho* é apenas um exemplo de classe que foi gerada pelo módulo gerador de DSLs.

A classe *Sets* é uma classe singleton que contém todas as informações do repositório de dados, onde estão armazenados conjuntos e elementos. A classe *Set* representa um conjunto propriamente dito e possui um ou mais elementos, que por sua vez pode estar associado a um ou mais conjuntos. A classe *Set* contém todas

as operações implementadas para manipulações de conjuntos e está diretamente associada à classe *Sets*.

A classe *SetsRDF* é responsável por identificar os conjuntos definidos pelo usuário via RDF e criá-los no repositório de dados. Esta classe também está associada à classe *Sets*.

As classes *ElementsRDF* exerce o papel de leitura dos elementos definidos no RDF e criação dos elementos no repositório de dados.

### 3.5.5
### Diagrama de Seqüência

Diagrama de seqüência é um tipo de diagrama usado em UML (Unified Modeling Language), representa a seqüência de processos (mais especificamente, de mensagens passadas entre objetos) em um programa de computador.

Um diagrama de seqüência descreve a maneira como os grupos de objetos colaboram em algum comportamento ao longo do tempo. Ele registra o comportamento de um único caso de uso. Ele exibe os objetos e as mensagens passadas entre esses objetos no caso de uso. Um design pode ter uma grande quantidade de métodos em classes diferentes, o que torna difícil determinar a seqüência global do comportamento. Esse diagrama é simples e lógico, a fim de tornar óbvios a seqüência e o fluxo de controle.

Abaixo segue o diagrama de seqüência:

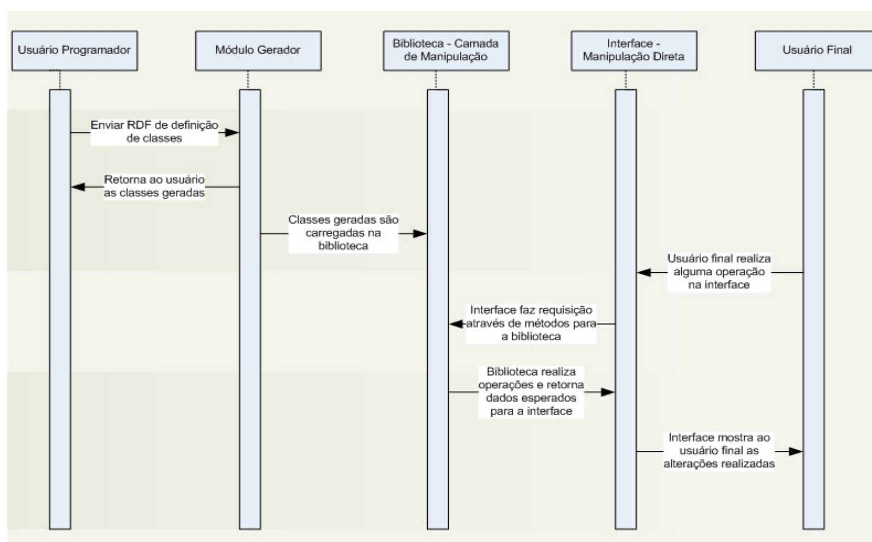


Figura 25: Diagrama de Seqüência

PUC-Rio - Certificação Digital Nº 0410841/CA

Primeiramente o usuário programador de acordo com as necessidades de uma interface define as classes que serão utilizadas. Esta definição é feita através de um RDF Schema e é enviado para o Módulo Gerador, que por sua vez envia como resposta ao usuário as classes geradas automaticamente e tais classes são carregadas na biblioteca responsável pela manipulação dos conjuntos. Estas são as interações e o fluxo de informações que ocorrem entre o usuário programador e o sistema.

Há também as interações entre o usuário final e a interface que ele manipula. Teríamos então o usuário final realizando alguma operação na interface, que por sua vez manda um pedido de execução de algum método para a biblioteca. Esta responde então com o conteúdo resultante da operação para a interface que por fim mostra as alterações para o usuário.

PUC-Rio - Certificação Digital Nº 0410841/CA

## 3.6 Modelo de Implementação

O modelo de implementação ideal para a utilização da biblioteca criada é o MVC (*Model View Controller*) (MVC). O MVC é uma arquitetura de software que separa o modelo de dados da aplicação, a interface do usuário, e o controle de lógica em três componentes distintos. Desta forma, alterações feitas em um componente causa impactos mínimos nos outros.

MVC muitas vezes é confundido com um design pattern para software. Entretanto, o MVC se assemelha mais a uma arquitetura de aplicação do que a um típico design pattern. Então, o termo *architectural pattern* pode ser aplicado.

Para a construção de uma aplicação utilizando uma arquitetura MVC devemos definir três módulos:

***Modelo***

Representação da informação para um domínio específico no qual a aplicação é executada. Modelo é apenas um outro nome para a camada de domínio. Domínio lógico acrescenta significado aos dados.

***View***

Renderiza o modelo em uma forma adequada para a interação, tipicamente um elemento da interface do usuário. MVC é frequentemente utilizado em aplicações Web, onde o módulo *View* é a página HTML e o código que obtém os dados para a página.

***Controller***

Representa eventos, geralmente ações do usuário, e realiza mudanças no componente *Model* e às vezes no *View*.

Muitas aplicações utilizam repositórios de dados como, por exemplo, bancos de dados. MVC não menciona essa camada de dados, pois considera que ela está encapsulada pelo componente *Model*.

Um fluxo de controle geralmente funciona da seguinte forma:

1. O usuário interage com a interface de alguma forma, pressionando um botão, por exemplo.
2. Um controlador trata o evento vindo da interface.
3. O componente *controller* acessa o componente *model*, possivelmente atualizando-o de uma maneira compatível com a ação do usuário.
4. O componente *view* utiliza o componente *model* para gerar uma interface para o usuário apropriada. O *view* pega os dados do *model*. O *model* não tem conhecimento direto do *view*.
5. A interface do usuário espera por novas interações do usuário, a qual começa o ciclo novamente.

O MVC introduz o objeto *controller* entre o *view* (GUI) e o *model* (objeto) para fazer a comunicação desses dois objetos. A implementação atual do objeto *controller* pode variar, mas a idéia de que o objeto transforma eventos em mudanças nos dados e na execução dos métodos deve ser considerada como a essência deste *pattern*.

Uma implementação típica para uma aplicação Web irá utilizar o componente *view* para desenhar o HTML em resposta a uma ação do usuário. Entretanto, se um XML é requerido como resposta, apenas o componente *view*

PUC-Rio - Certificação Digital Nº 0410841/CA

necessita alterações (para um componente que manipule XML), enquanto os outros dois componentes se mantêm inalterados.

O diagrama abaixo mostra a relação existente entre os componentes. A linha sólida indica associação direta e a pontilhada indica associação indireta. E as direções das setas indicam o fluxo de informações.

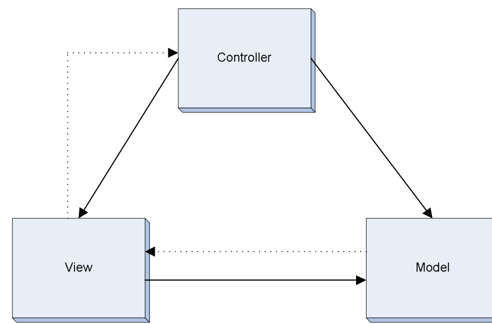


Figura 26: Modelo MVC

PUC-Rio - Certificação Digital Nº 0410841/CA

Embora seja muito utilizado, o MVC tem vantagens e desvantagens que devem ser consideradas antes de projetar uma solução.

Se for construído corretamente, os modelos podem compartilhar um certo grau de estabilidade, onde interface de usuário pode sofrer mudanças dramáticas (tipicamente por problemas de usabilidade). Separando a *view* do *model* faz o *model* ficar mais robusto.

Muitos métodos para execução de projetos realizam primeiramente o design da interface do usuário e finalizam a interface. O efeito colateral disso é que freqüentemente o domínio do problema não é entendido por completo pelos programadores antes do final do processo de implementação. Assim, quando os desenvolvedores estão aptos a criar uma boa interface, são impossibilitados de alterá-las. Modelos MVC permitem que o código seja mais flexível no processo de desenvolvimento, permitindo que sejam realizadas alterações no momento exato que forem necessárias.

Nesta dissertação a biblioteca desenvolvida se encaixa como *controller* e *model*, pois ao mesmo tempo define o modelo de dados que é utilizado e realiza as operações necessárias para fazer a elo entre a interface e o modelo.

Como componente *view* vamos considerar a dissertação do Ricardo R. Kawase, que é um framework para geração de interfaces de manipulação direta.

Este framework criará uma interface que por sua vez irá utilizar a linguagem voltada para o domínio do problema, e o usuário poderá executar operações sobre seus elementos de interface sem ter que compreender tais operações literalmente. O framework também exerce o papel de *controller*, pois há operações que se executadas realizam alterações no *view*. O objetivo final é conseguirmos acoplar os componentes e criar uma estrutura MVC flexível o suficiente para que o usuário possa definir a semânticas das operações a serem realizadas em interações na interface.

O usuário pode, por exemplo, realizar uma interação de *drag-and-drop* de um conjunto de pessoas sobre um conjunto de fotos e definir qual operação será executada pelo componente *controller* sobre os dados representados pelo *model*. E a manipulação direta executada pelo usuário será responsável por uma alteração no *view*. Há também outras manipulações disponíveis ao usuário, como: aplicar um filtro sobre um grupo de imagens, e então as imagens que não atendem as restrições definidas para o filtro desaparecem (para esse filtro também pode ser definida a semântica a ser aplicada).

É justamente neste momento que encontramos a maior dificuldade para unir os três componentes e obter uma única solução baseada na arquitetura MVC. O *controller* que é responsável pela comunicação entre o meu projeto e o do Ricardo teve que ser ajustado para que o *model* pudesse enviar e receber informações do *view* e vice-versa. Teve que ser criado um mapeamento entre os tipos de dados que são utilizados em cada componente, um mapeamento da interface para o domínio e outro do domínio para a interface.

Este mapeamento funciona da seguinte forma: a interface informa para a DSL que operações devem ser executadas sobre quais conjuntos e/ou elementos. A DSL por sua vez executa os comandos recebidos e retorna uma resposta de como a operação deve ser refletida na interface. Estamos realizando um mapeamento da interface para o domínio da aplicação e depois o caminho inverso, do domínio para a interface.

Abaixo podemos ver um exemplo da classe que foi implementada para fazer o mapeamento entre o *view* e o *model*, seu código completo encontra-se no apêndice I desta dissertação.

PUC-Rio - Certificação Digital Nº 0410841/CA

### *Map*

| Map |
|---|
| -staticID : string = ""<br>-staticGroupDescription : string = "" |
| +Map()<br>-LoadElements(in sets : Sets)<br>-LoadSetAsGroup(in conjunto : Set) : PGroup<br>-convertSetToGroup(in conjunto : Set) : PGroup<br>+AddGroup(in group : PGroup, in cloneGroup : PGroup)<br>+AddItem(in item : PItem, in group : PGroup)<br>+RemoveItem(in item : PItem)<br>+RemoveGroup(in group : PGroup)<br>+LoadGroup(in queryString : string)<br>+doItemAction(in action : string, in item : PItem, in sourceGroup : PGroup, in targetGroup : PGroup) : bool<br>+doGroupAction(in action : string, in group1 : PGroup, in group2 : PGroup, in lastGroup : bool) : PGroup<br>+doGroupActionMenu(in action : string, in groups : ArrayList)<br>+getFacetas(in item : PItem) : ArrayList<br>+createGroupFromFacet(in facetName : string)<br>+ApplicationClose()<br>+ApplicationOpen() |

Figura 27: Classe Map

Esta classe é composta de métodos estáticos, não sendo necessário assim instanciá-la para utilização. Vejamos método por método para identificarmos quando é realmente necessária a existência de um mapeamento e a melhor maneira de realizá-lo para que as informações possam ser trocadas entre a interface e o modelo.

Neste estudo de caso que foi desenvolvido, estamos utilizando uma interface composta por grupos e por itens. Estes objetos são mapeados em conjuntos e elementos respectivamente no modelo de informação. Assim, todas as operações que ocorrem na interface são repassadas para o modelo que as executa e então as retorna para a interface, e que finalmente se modifica para contemplar as alterações ou operações realizadas pela interação do usuário.

O método *MapElements* é apenas utilizado para a criação de conjuntos para demonstração, não necessita ser criado. O método *getGroup* é utilizado para retornar um grupo da interface. O método *LoadElements* carrega os objetos definidos pelo usuário em RDF no repositório do modelo, e permite que o usuário final adicione novos conjuntos, mapeados em grupos, a partir desses elementos. Este métodos são chamados quando a aplicação está iniciando, e então é necessário que os elementos sejam carregados no repositórios.

O método *LoadSetAsGroup* recebe um conjunto como entrada e retorna um grupo para que este seja então visualizado na interface pelo usuário final. Da mesma forma funciona o método *convertSetToGroup* que retorna um grupo a

PUC-Rio - Certificação Digital Nº 0410841/CA

partir de um conjunto mas não o disponibiliza para o usuário na interface. Estes métodos são usados quando é necessário converter um conjunto para um grupo.

Os métodos *AddItem* e *AddGroup* devem ser implementados para que recebam um item ou um grupo da interface e os adicione no repositório que representa o modelo de informação. O mesmo acontece nos métodos *RemoveItem* e *RemoveGroup* que recebem um item ou um grupo da interface e os deleta do modelo. Estes métodos são utilizados quando se deseja manipular itens ou grupos da interface, e estes devem estar tanto na interface quanto no modelo para que as operações possam ser executadas corretamente.

O método *doItemAction* é responsável por executar as mesmas operações no modelo que são realizadas na interface, como por exemplo copiar e colar. O método *doGroupAction* é executado para executar as operações nos grupos da interface, estas operações são realizadas nos conjuntos que os representam no modelo. Como por exemplo, união, interseção e diferença. Estes métodos são executados quando o usuário está interagindo com a interface de manipulação direta.

A partir de um item da interface é possível também obter outras informações. Neste caso existe um método *getFacetas* que retorna todas as facetas que um elemento está relacionado. Como complemento deste método, há um outro método *createGroupFromFacet* que cria um grupo da interface a partir de uma faceta de um item/elemento. Estes métodos são utilizados quando se deseja obter informações sobre um elemento do modelo e a partir deste dado gerar outros conjuntos e consequentemente grupos da interface.

Também existem os método *ApplicationOpen* e *ApplicationClose* que são utilizados para executar operações no início ou no término da aplicação.

Utilizando a arquitetura MVC podemos mapear as operações realizadas no *view* em operações a serem executadas no *model*. A naturalidade da abordagem será dependente da naturalidade da interface para representar o domínio onde se encontra a aplicação.

Com o MVC, cada componente estará praticamente independente do outro. Esta arquitetura permite também o mapeamento das operações realizadas no *view* em operações no *model*. Se for preciso realizar uma alteração na biblioteca de métodos para manipulação dos conjuntos, não será preciso nenhuma alteração no

PUC-Rio - Certificação Digital Nº 0410841/CA

componente de interface. Da mesma forma se algum elemento de interface tiver que ser adicionado, nada afetará os outros componentes.

PUC-Rio - Certificação Digital Nº 0410841/CA