5
Ilustração de Uso

Como citado anteriormente, a interface gerada por este framework se encaixa apenas como uma *view* de um modelo e um *controller*. Este capítulo descreve os requisitos, esforços, ganhos e resultados da utilização do framework com um modelo e *controller* existentes. Neste caso os componentes de modelo e *controller* utilizados são os implementados por Leonardo Belmonte em sua dissertação de mestrado.

5.1
O Modelo

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

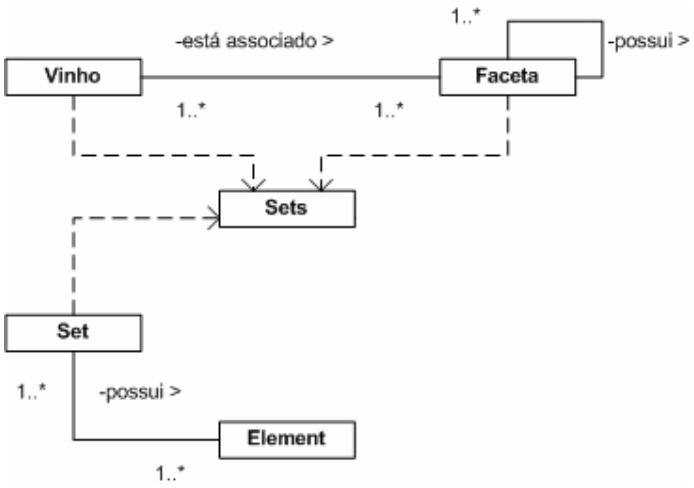

Figura 13 – Diagrama de Classes (adaptado de Belmonte, 2006)

Neste caso a classe *Vinho* representa os objetos e classe *Faceta* representa possíveis propriedades dos objetos. Um vinho está associado a uma ou mais facetas e uma faceta possui uma ou mais facetas pai.

A classe *Sets* é uma classe *singleton* que contém todas as informações do repositório de dados, onde estão armazenados conjuntos e elementos. *Singleto*, é um padrão de projeto de software (*Design Pattern*). Este padrão garante a existência de apenas uma instância de uma classe, mantendo um ponto global de acesso ao seu objeto.

Muitos projetos necessitam que algumas classes tenham apenas uma instância. Por exemplo, em uma aplicação que precisa de uma infraestrutura de log de dados, pode-se implementar uma classe no padrão *Singleton*. Desta forma existe apenas um objeto responsável pelo log em toda a aplicação que é acessível unicamente através da classe *Singleton*.

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*.

5.2 Mapeamento de Classes

O primeiro mapeamento concebido foi o mapeamento entre os elementos e conjuntos do modelo para os itens e grupos da interface. A própria nomenclatura deixa claro o tipo de mapeamento que foi feito. Os grupos do framework da classe *PGroup* foram mapeados para os conjuntos do modelo da classe *Set* e os itens do framework da classe *PCustomItem* foram mapeados para os elementos do modelo da classe *Element*. Neste caso o elemento utilizado é o do tipo *Vinho*. Assim, um vinho no modelo é representado na interface por um item, e um conjunto é representado por um grupo.

Os grupos, assim como os conjuntos no modelo, apenas representam o agrupamento de diversos vinhos. Seu mapeamento em relação a sua construção é simples. O mapeamento de métodos aplicáveis aos grupos será tratado nas próximas sessões.

Por conveniência, o tipo de item que deve ser adicionado na interface é uma arquivo texto de extensão *.set*. Este arquivo contém uma consulta que deve ser feita no repositório do modelo e este retorna um conjunto de vinhos.

5.2.1
Mapeando Grupos

A classe *PGroup* é a representação da classe *Set* na interface. O grupo exibido pode ser customizável quanto à sua renderização, contudo sua forma padrão é intuitiva o suficiente para idealizar um agrupamento de itens. O principal ponto que deve ser levado em conta durante a customização é a dos métodos aplicáveis aos grupos. Neste caso os métodos aplicáveis a grupos definidos pela DSL são: união, interseção e diferença. Desta forma, os métodos relativos a estas operações devem ser implementados.

```
public ArrayList GetClassFunctionsNames()
{
  ArrayList retorna = new ArrayList();
   retorna.Add("Arrange Cascade");
   retorna.Add("Arrange Side by Side");
   retorna.Add("Change Color");
   retorna.Add("-");
   retorna.Add("Union");
   retorna.Add("Intersection");
   retorna.Add("Difference");
   return retorna;
}
```

Quadro 15 – Configurando nomes das funções.

```
public ArrayList GetClassFunctions()
{
  ArrayList retorna = new ArrayList();
  retorna.Add(new System.EventHandler(ArrangeCascade));
  retorna.Add(new System.EventHandler(ArrangeSideBySide));
  retorna.Add(new System.EventHandler(ChangeGroupColor));
  retorna.Add(new System.EventHandler(GenericFunctionMenu));
  retorna.Add(new System.EventHandler(GenericFunctionMenu));
  retorna.Add(new System.EventHandler(GenericFunctionMenu));
  retorna.Add(new System.EventHandler(GenericFunctionMenu));
  return retorna;
}
```

Quadro 16 – Configurando chamadas das funções.

```
protected void GenericFunctionMenu(object sender, System.EventArgs e)
{
  string optionText = (sender as MenuItem).Text;
  if(optionText=="Union")
  {
    if(this == MainForm.mainForm.currentGroup)
      Map.doGroupActionMenu("union", MainForm.mainForm.selectedGroups);
  }
  else if(optionText=="Intersection")
  {
    if(this == MainForm.mainForm.currentGroup)
      Map.doGroupActionMenu("intersection", MainForm.mainForm.selectedGroups);
  }
  else if(optionText=="Difference")
  {
    if(this == MainForm.mainForm.currentGroup)
      Map.doGroupActionMenu("difference", MainForm.mainForm.selectedGroups);
  }
}
```

Quadro 17 – Configurando chamadas das funções.

Os métodos aplicáveis que são utilizados em interações de manipulação direta na interface também devem ser configurados. Estas chamadas decorrentes de manipulações diretas são configuradas no método do *doAction()*.

```
…
else if (actionName == "union")
{
  if(targetGroup!=null)
    resultGroup = this.Union(RootLayer, oldParent, targetGroup, true, lastGroup);
  else
    resultGroup = null;
}...
```

Quadro 18 – Configurando chamadas de manipulação direta.

O método *doAction()* recebe diversos parâmetros, dentre eles o *actionName* o qual indica a interação efetuada. O nome desta interação é o mesmo que foi escolhido no arquivo de configuração. Para este trecho do código, cabe agora ao método

Union() da classe *PGroup()* chamar e retornar os resultados de um método de união na classe *Map*.

5.2.2
Mapeando Items

A classe *PCustomItem* foi montada para que sua renderização exiba as informações do vinho de uma compactada e outra integral. A forma compacta exibe apenas uma etiqueta com o nome do vinho.

Figura 14 – Modelo de exibição compacta do item vinho.

Já a exibição integral mostra uma pequena ficha do vinho com mais informações: nome do vinho, descrição, atributos e imagens associadas.

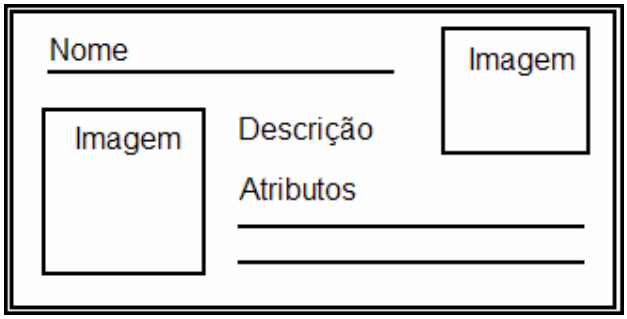

Figura 15 – Modelo de exibição integral do item vinho.

Para o sucesso deste mapeamento, a classe customizável *PCustomItem* também recebeu alguns novos atributos relativos aos atributos da classe *Vinho*. Estes atributos (*região*, *tipo* do vinho, *ano*, *descrição*) foram adicionados no método *GetCustomInfo()* já citado anteriormente. Da mesma forma estes foram adicionados aos filtros possíveis no método *GetPossibleFilters()*. Finalmente, implementando os testes de comparação no método *TestFilters()* os vinhos podem ser filtrados pelos seus atributos através da lente da interface.

```
public void GetCustomInfo()
{
  customInfoTag.Add("title");
```

```
    customInfoValue.Add("");
    customInfoReadOnly.Add(true);
    customInfoTag.Add("type");
    customInfoValue.Add("");
    customInfoReadOnly.Add(true);
    customInfoTag.Add("region");
    customInfoValue.Add("");
    customInfoReadOnly.Add(true);
    customInfoTag.Add("year");
    customInfoValue.Add("");
    customInfoReadOnly.Add(true);
  }
```

Quadro 19 – Adicionando atributos à classe customizada.

```
  protected void GetPossibleFilters()
  {
    possibleFilters.Clear();
    possibleFilters.Add("title");
    possibleFilters.Add("year");
    possibleFilters.Add("type");
    possibleFilters.Add("region");
  }
```

Quadro 20 – Configurando os possíveis filtros da classe customizada.

Uma outra customização foi o *drop down menu* gerado quando um vinho é clicado com o botão direito. Nos métodos *GetCustomClassFunctions()* e *GetCustomClassFunctionsNames()*. Nestes foram adicionadas a quais facetas este vinho pertence e o clicar em um desses itens do menu gera um novo grupo com todos os vinhos pertencentes a faceta escolhida. Esta implementação apesar de ter requerido um esforço irrisório resulta em uma poderosa funcionalidade de navegação facetada.

```
  public void GetCustomClassFunctions(ArrayList functions)
  {
    ArrayList facetasNomes = Map.getFacetas(this as PItem);
    foreach (string facetaTitulo in facetasNomes)
      functions.Add(new System.EventHandler(GenericFunctionMenu));
  }
```

Quadro 21 – Configurando nomes das funções da classe customizada.

```
public void GetCustomClassFunctionsNames(ArrayList functionNames)
{
  ArrayList facetasNomes = Map.getFacetas(this as PItem);
  foreach (string facetaTitulo in facetasNomes)
    functionNames.Add(facetaTitulo);
}
```

Quadro 22 – Configurando chamadas das funções da classe customizada.

Finalmente, as últimas customizações necessárias são as referentes às ações de manipulações diretas. O método *doAction()* deve ser estendido para identificar a interação e fazer as chamadas necessárias. O nome da interação é o mesmo do selecionado no arquivo de configuração, como por exemplo: copiar e colar, recortar e colar, colar especial, entre outros.

```
…
else if (ActionName == "copypaste")
{
  resultItem = this.PasteToGroup(RootLayer, oldParent, targetGroup, true, false);
}...
```

Quadro 23 – Configurando manipulação direta da classe customizada.

Neste trecho, é feita a identificação da ação e então é chamado o método *PasteToGroup()*. Este método faz as devidas execuções para chamar e retornar um método equivalente (*copypaste*) na classe *Map*.

5.3 Implementando a Classe Map

Os métodos de mapeamento são feitos na classe *Map*, pois esta é a única que enxerga tanto as classes do framework quanto as classes do modelo. Todas as operações feitas sobre a interface, que alteram de alguma forma a estrutura hierárquica dos grupos ou itens, acaba passando por algum destes métodos. Por exemplo, as ações de adicionar item e remover item na interface. Quando um item é adicionado em um grupo o método *AddItem()* da classe *Map* recebe o item e o grupo nos tipos *PItem* e *PGroup*. Com o identificador destes, é feita a busca no

modelo pelo elemento equivalente ao item e pelo conjunto equivalente ao grupo. Feito isso o elemento é adicionado ao conjunto.

O método de remover, *RemoveItem()*, segue a mesma lógica. Este recebe qual o item deve ser removido e através dos identificadores localiza de qual conjunto deve ser retirado.

```
public static void AddItem(PItem item, PGroup group)
{
  Sets sets = Sets.Open(false);

  Set.Set conjunto = sets.getSetByIdentifier(group.GroupID);
  Set.Element elemento = sets.getElementByIdentifier(item.ItemID);
  conjunto.addElement(elemento);
  sets.UpdateElement(conjunto);

  sets.Close();
}

public static void RemoveItem(PItem item)
{
  Sets sets = Sets.Open(false);

  PGroup parentGroup = MainForm.mainForm.ConvertToGroupType(item.Parent);
  if(parentGroup != null)
  {
    Set.Set conjunto = sets.getSetByIdentifier(parentGroup.GroupID);
    Classes.Vinho elemento = conjunto.getElementByIdentifier(item.ItemID) as Vinho;
    conjunto.removeElement(elemento);
    sets.UpdateElement(conjunto);
  }
  sets.Close();
}
```

Quadro 24 – Mapeamento dos métodos de adicionar e remover itens.

Da mesma forma este mesmo mapeamento é feito para as operações de adicionar e remover grupos.

```
public static void AddGroup(PGroup group, PGroup cloneGroup)
```

```
        {
          Sets sets = Sets.Open(false);

          Set.Set conjunto = new Set.Set();
          conjunto.Identificador = cloneGroup.GroupID;
          for(int i =0; i<group.ChildrenCount;i++)
          {
            string x = (group.GetChild(i) as PItem).ItemID;
            Set.Element elemento = sets.getElementByIdentifier(x);
            conjunto.Elementos.Add(elemento);
          }
          sets.UpdateElement(conjunto);
          sets.Close();
        }

        public static void RemoveGroup(PGroup group)
        {
          Sets sets = Sets.Open(false);
          Set.Set conjunto = sets.getSetByIdentifier(group.GroupID);

          sets.RemoveElement(conjunto);
          sets.UpdateElement(conjunto);

          sets.Close();
        }
```

Quadro 25 – Mapeamento dos métodos de adicionar e remover grupos.

Para o mapeamento das operações mais complexas de manipulação foram criados dos métodos genéricos que reconhecem qual a chamada. Um para ações sobre os itens *doItemAction()* e um para grupos *doGroupAction()*. Estes métodos recebem como parâmetro qual função deve ser executada e os objetos pertinentes a estas funções. O quadro abaixo descreve um pequeno trecho dentro do método *doGroupAction()* onde é reconhecida a função de união e são feitas algumas codificações para então ser feita a chamada do método *União* definido pelo modelo na classe *Set*.

```
...
if( action == "union" )
{
```

```
  resultConjunto = conjunto1.Uniao(conjunto2);
  sets.AddElement(resultConjunto);
  if(clearStaticID)
    sets.removeSetByIdentifier(staticID);
  staticID = resultConjunto.Identificador;
  staticGroupDescription = "("+staticGroupDescription+" U "+group1.GroupDescription+")";

  resultGroup = convertSetToGroup(sets.getSetByIdentifier(staticID));
  if(lastGroup)
  {
    staticID = "";
    resultGroup.GroupDescription = staticGroupDescription;
    staticGroupDescription = "";
    return resultGroup;
  }
}
...
```

Quadro 26 – Mapeamento da união de grupos.