ISEL | DEETC

Ambientes Virtuais de Execução

ChelasUIMaker (Engine Development)

Semestre de Verão de 2010/2011

Autores:
31401 – Nuno Cancelo
33595 – Nuno Sousa

Indicie

Introdução

O projeto proposto conceptualiza a concretização (e consequente utilização), de uma interface programática denominada de API fluente.

Esta interface traz muita elegância ao código, permitindo (quase) programar como se estivesse a escrever em linguagem corrente.

Acompanhando esta tecnologia pretende-se utilizar os conteúdos programáticos adquiridos durante este semestre, tal como a reflexão e os eventos.

Com estas ferramentas, é proposto pelo enunciado a criação de um "Controlador" conseguisse analisasse uma configuração passada por argumento e que a mesma fosse concretizada numa aplicação funcional.

Concretização

O projeto foi elaborado em algumas fases, sendo que a primeira fosse a analise de requisitos do enunciado.

Nesta analise constatou-se ;

- Especificação de configuração através da linguagem fluente
- Esta configuração permite a gerar a interface gráfica baseada na biblioteca System.Windows.Forms, assim como eventos.
- Segue um modelo reconhecido MVC
- Os nomes do handlers do eventos do controlador seguem uma norma
- Existem dois exemplos a demonstrar a utilização da API
- As tags utilizadas na Configuração são especificadas pelo System.Windows.Forms version = 4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
- Que o anexo fornecido têm já algumas implementações
- O código do enunciado não corresponde às imagens fornecidas
- Algumas frase não fazem sentido.

Na fase seguinte, foi efetuada uma estudo da forma como o código fornecendo estava escrito e de que forma os módulos se interligavam.

Como forma de analise, realizou-se o que é denominado reverse engineering para obter o diagrama UML (Illustration 1) da solução implementada de forma a ser evidente a sua utilização.

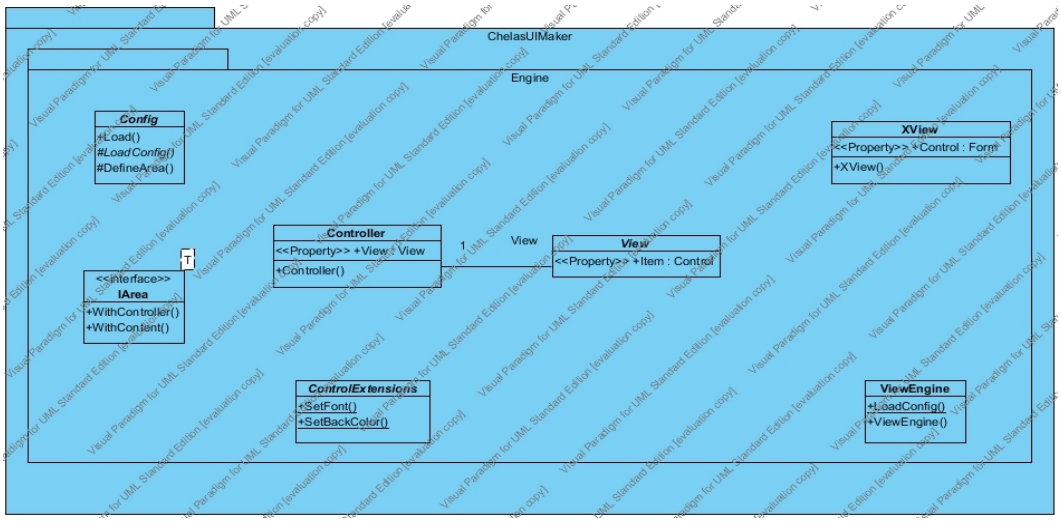

Illustration 1: UML da Solução Fornecida

Como se pode verificar excetuando a ligação entre a classe Controller e a classe View não existe uma ligação forte entre os objetos. É neste momento que o conceito MVC se torna fulcral na implementação da solução. Para entender o conceito ajuda, mais uma vez, ver o diagrama Illustration 2: Modelo MVC.

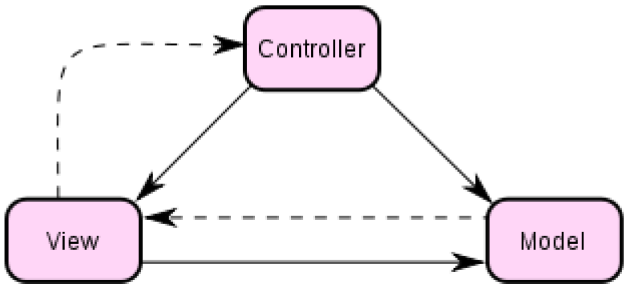

Illustration 2: Modelo MVC

Neste momento já se consegue começar a entender qual a interligação por entre os módulos fornecidos.

Podemos então inferir que o objetivo do projeto é criar uma biblioteca de classes que permita pegar numa configuração e aplicar numa View.

Seguindo o fluxo de trabalho de um dos exemplos fornecidos e que podemos generalizar para todos é a seguinte:

Illustration 3: Fluxo de Execução

No final desta fase está claro o fluxo de trabalho, o modelo MVC e os objetivos do projeto, o que perspetiva uma implementação eficaz.

Passo 1

O Senso Comum diz que "antes de aprender a andar, temos que gatinhar", então vamos começar a "descascar" o código fornecido, tentando efetuar o mínimo de alterações à API fornecida.

Analisando o código os seguintes não foram alterados, mantendo a mesma implementação fornecida:

- MyController.cs
- MyConfig.cs
- Program.cs

A razão pela qual estas funcionalidades não foram alteradas é evidente, estão

dependentes da concretização do objetivo pela qual existem, não dependem do motor programático.

Deste modo tentamos seguir o caminho sugerido da execução do programa.

Passo 2

Neste estagio do processo vamos começar a fazer desenvolvimentos do projeto e caminhar para a implementação.

A primeira classe a ser alterada é ***ViewEngine.cs***, que é o ponto de entrada no programa principal e cujo seu método único é responsável por carregar uma configuração e devolver uma "vista" que seja a representação gráfica dessa configuração. Como descrita em Text 1: Concretização da classe ViewEngine

```
public class ViewEngine
{
    public static XView LoadConfig(Config myConfig)
    {
        if (myConfig == null)
            throw new InvalidOperationException();

        myConfig.Load();
        return new XView(myConfig);
    }
}
```

Text 1: Concretização da classe ViewEngine

Nesta pequena concretização podemos constatar três ações:

1. Carregamento do objeto do tipo Config
2. Concretização do objeto do tipo Config numa View
3. Retorno desse objeto do tipo View

Estas ações estão diretamente relacionadas com as classes que definem as suas ações.
Temos então dois objetos a serem implementados: o Config.cs e Xview.cs

A implementação de Config.cs pode ser visualizada em i

```
public abstract class Config
{
  Object _area;
  Type _parameter;

  public void Load(){ LoadConfig(); }
  protected abstract void LoadConfig();
  protected IArea<T> DefineArea<T>(Action<T> a)
where T : new(){
    IArea<T> area = new ChelasUIArea<T>(a);
    _area = area;
   _parameter = typeof(T);
        return area;
    }

  public Object IArea
  { get { return _area; } }

  public Type ParameterType
  { get { return _parameter; } }

}
```

i

i Implementação de Config.cs

É relevante revelar esta classe em primeiro lugar para dar a conhecer um nova classe que implementa a linguagem fluente. Esta classe implementa a interface

IArea<T>. A esta interface foi adicionada dois métodos que permite a obtenção de objetos internos da própria. Estes objetos poderiam ter sido obtidos através de reflexão, mas uma vez que estamos a implementar o objeto torna-se mais elegante e eficiente o código à posterior noutras classes da solução.

A API desta interface pode ser visualizada em ii:

```
public interface IArea<T>
{
    IArea<T> WithController<C>();

    IArea<T> WithContent<U>(IArea<U> content);
    System.Type Controller();
    System.Windows.Forms.Control Control();
}
```

ii

ii Implementação de IArea<T>

Assim torna-se indispensável então falar das seguintes classes antes de revelar a sua realização:

ChelasUIArea.cs

Como foi referido antes a classe IArea<T> é uma interface com quatro métodos. Esta classe faz uma possível implementação dessa interface. Os métodos public IArea<T> WithContent<U>(IArea<U> content) e public IArea<T> WithController<C>() concretizam a funcionalidade da dita Linguagem Fluente[1] devolvendo a referência para a própria classe, permitindo que os seus métodos sejam encadeados.

A implementação dos outros métodos da interface facilitam a obtenção dos objeto e do tipos dos métodos permitindo uma interligação mais amigável.

Implementação em iii.

Controller.cs

Uma classe simples que somente guarda a View, facilitando a comunicação para os controladores específicos.

Implementação em iv

Xview.cs

Esta classe estende da classe View e implementa os seus métodos de obtenção e de afetação.

No seu construtor recebe uma configuração que o programador deseja ver concretizada numa aplicação gráfica. No decorrer da implementação recorre-se à reflexão para gerar a aplicação compatível com System.Windows.Forms.

1 http://en.wikipedia.org/wiki/Fluent_interface

View.cs

É uma classe abstrata que menciona somente dois métodos de afetação de propriedades.
Implementação em v

Dada a breve descrição podemos verificar na conceção dos objetos que os realizam tarefas simples mas com elevado potencial demonstrando a poder da reflexão da Framework .NET.

Implementações

Aqui temos as possíveis concretizações das classes mencionadas anteriormente.

ChelasUIArea<T>

```
class ChelasUIArea<T> : IArea<T> where T : new()
{
  private Control _control;
  private Type _controller;
  private T _type;

  public ChelasUIArea(Action<T> action)
  {
   _type = new T();
   if (action != null) action(_type);
     _control = _type as Control;
  }
  public IArea<T> WithController<C>()
  {
    _controller = typeof(C);
    return this;
  }
  public IArea<T> WithContent<U>(IArea<U> content)
  {
    if (content != null)

      {
```

```
      Control c = content.Control();
      if (_control != null && c != null)
      {
        _control.Controls.Add(c);
      }
        }
      return this;
  }
  public Type Controller()
    {  return _controller;  }
  public Control Control()
    { return _control;  }
  public T Type()
    { return _type; }
  public override string ToString()
    { return
        String.Format
        ("Name: ${0}",this.GetType().FullName);
    }
 }
```

iii

iii Implementação ChelasUIArea<T>

Controller:

```
public class Controller
{
    private View view;
    public View View { get { return view; } set { view = value; } }
}
```

iv

iv Implementação de Controller

View:

```
public abstract class View
{
    public abstract Control this[String s] { get; set; }
}
```

v

v Implementação de View

XView.cs:

```
public class XView : View
{
private Form _formControl;
public System.Windows.Forms.Form Control { get { return _formControl; } set { _formControl = value; } }

Config _config;

public XView(Config config)
{
    if (config == null) return;
    _config = config;
    //Obtém o objeto do tipo IArea que foi guardado pelo DefineArea
    IArea<Form> area = (IArea<Form>)_config.IArea;

    //Obtém o objeto do tipo Control que foi guardado pelo DefineArea
    Control c = area.Control();

    if (c != null)
    {
        _formControl = new Form();
        processProperties(c);
    }
    Type t = area.Controller();

    object o = t.GetConstructor(new Type[] { }).Invoke(null);

    processController(o);

    //set view in the controller
    if (t != null) t.GetProperty("View").SetValue(o, this, null);
}

public override System.Windows.Forms.Control this[string s]
{
    get
    {
        Control c = null;
        fetchControlWithName(s, _formControl.Controls.Cast<Control>(), ref c);
        return c;
    }
    set
    {
        Control c = null;
        fetchControlWithName(s, _formControl.Controls.Cast<Control>(), ref c);
        if (c != null) c.Name = s;
    }
}

private void fetchControlWithName(string s, IEnumerable<Control> controls, ref Control c)
{
    foreach (Control ctrl in controls)
    {
        if (ctrl.Name.Equals(s))
        {
            c = ctrl;
            return;
        }

        if (ctrl.Controls.Count > 0) fetchControlWithName(s, ctrl.Controls.Cast<Control>(), ref c);
    }
}
//Caso haja control copia as suas definições e controlos
private void processProperties(Control control)
{
    _formControl.Controls.AddRange(control.Controls.Cast<Control>().ToArray());
    _formControl.Name = control.Name;
    _formControl.Text = control.Text;
    _formControl.Width = control.Width;
    _formControl.Height = control.Height;
}
```

```
//Processa o Controller. Associa e Regista os eventos
private void processController(Object object_)
{
    if (object_ == null) return;

    Type t = object_.GetType();
    MethodInfo[] m =
        t.GetMethods(BindingFlags.NonPublic | BindingFlags.Instance | BindingFlags.DeclaredOnly);
    String[] bnfHandler;
    int size;
    String event_, element_, context_;
    foreach (MethodInfo mi in m)
    {
        /*
            * BNF Controller Handler
            * <Handler Name> ::= [<context>'_'][<element>'_']<event>
            * <Handler Name> ::= [Context][Element][Event]
            *
            */

        bnfHandler = mi.Name.Split('_');
        size = bnfHandler.Length;
        event_ = bnfHandler[bnfHandler.Length - 1];
        //element_ = (bnfHandler.Length < 2) ? null : bnfHandler[bnfHandler.Length - 2];
        context_ = (bnfHandler.Length >= 3 &&  bnfHandler[bnfHandler.Length -
3].Equals( _config.ParameterType.Name))? _config.ParameterType.Name:null;

        LinkedList<Control> control = new LinkedList<Control>();

        processControls(_formControl.Controls.Cast<Control>(), ref control, context_);
        AddHandler(control, object_, event_, mi);
    }
}


//Trata dos Eventos
private void AddHandler(IEnumerable<Control> control, Object object_, String name, MethodInfo
methodInfo)
{
    EventInfo event_;
    foreach (Control c in control)
    {
        event_ = typeof(Control).GetEvent(name);
        if (event_ != null)
        {

            event_.AddEventHandler(c, Delegate.CreateDelegate(event_.EventHandlerType, object_,
methodInfo, true));
        }
    }
}
//Stub para rotinar o processamento dos controlos
private void processControls(IEnumerable<Control> control, ref LinkedList<Control> list, String name
=null)
{
    if (control == null || list == null) return;

    foreach (Control c in control)
    {

        if (name == null || name.Trim().Equals("") || c.Name.Equals(name) ||
c.GetType().Name.Equals(name))
        {
            list.AddFirst(c);
            if (c.Controls.Count > 0) processControls(c.Controls.Cast<Control>(), ref list, name);
        }
    }
}

}
```

Passo 3

A facilidade de estender funcionalidades sem alterações no código do motor é uma ambição de qualquer programador.
O Visual Studio®[2] aliada à Framework .NET facilita a implementação de projetos do tipo class library.
Com o código fonte fornecido existe um ficheiro ControlExtensions.cs, no qual é fornecido já algum suporte para tipos complexos.
No sentido separar as funcionalidades, optou-se por realizar um projeto de biblioteca de classe e contem somente o referido ficheiro.
A concretização da classe sofreu uma pequena alteração:

```
public static class ControlExtensions
{
    public static Control SetFont(this Control ctrl, string font)
    {
        var x = font.Split(',');
        if (x.Length >= 2)
            ctrl.Font = new Font(x[0],float.Parse(x[1]));
        return ctrl;
    }

    public static Control SetBackColor(this Control ctrl, int colorArgb)
    {
        ctrl.BackColor = Color.FromArgb(colorArgb);
        return ctrl;
    }

}
```

vi

vi Implementação do ControlExtensions

2 http://www.microsoft.com/visualstudio/en-us/products/2010-editions

Considerações Finais

Ao longo do relatório não é notório as dificuldades que foram passadas na implementação da solução proposta.

Foi desperdiçado demasiado tempo na interpretação do enunciado e na assunção de que o código fonte era a API final e que não se podia mexer. Neste pressuposto chegou-se a uma ponto da implementação que não havia hipótese de continuar a produzir um bom código.

Outra situação que levou a uma perca de tempo foi a interpretação tida com o tratamento da sintaxe BNF[3], que após esclarecimento demonstrou que o fraco discernimento na interpretação do enunciado.

Após de todas as dúvidas esclarecida e implementação foi bastante simples.

Relativamente ao projeto os objetivos foram largamente alcançados, alcançado ainda dois objetivos extra:

- Conhecer um pouco a API System.Windows.Form
- Conhecer a fraca documentação (no sentido da qualidade do conteúdo da documentação) do MSDN referente à plataforma .NET.

3 http://pt.wikipedia.org/wiki/Formalismo_de_Backus-Naur

Agradecimentos

No decorrer do desenvolvimento do projeto houve algumas pessoas que de uma forma ou de outra ajudar a ultrapassar barreiras que estavam a bloquear o desenvolvimento.

Quero deixar o meu agradecimento aos seguintes colegas, que durante as suas férias abdicaram do seu tempo oferecendo o seu tempo, sendo que os mais importantes foram:

- Cláudia Crisóstomo (aluna 32142)
- Fábio Dias (aluno 33138)
- Mónica Rodrigues (aluno 31094)

Bibliografia

- Jeffrey Richter, CLR Via C#, 3Ed, Microsoft Press, 2010
- D. Box with C. Sells, “Essential .Net: The Common Language Runtime”, Addison-Wesley, 2002

Referências

- http://msdn.microsoft.com/library/default.aspx
- http://www.google.com

Illustration Index