Departamento de Engenharia Informática
Instituto Superior de Engenharia do Porto
Instituto Politécnico do Porto

Engenharia da Informação

4º ano da Licenciatura em Engenharia Informática

Introdução ao Desenvolvimento *roundtrip engineering* usando Rational XDE

(2ª edição)

Paulo Sousa

—♦—

Abril de 2005

Departamento de Engenharia Informática
Instituto Superior de Engenharia do Porto (ISEP/IPP)
Rua Dr. António Bernardino de Almeida, 431
4200-072 PORTO
Portugal
Tel. +351 228 340 500
Fax +351 228 325 219

Criado em Abril, 2004
Última modificação em 05 Abril, 2005 **(v 1.3)**
Email: psousa@dei.isep.ipp.pt
URL: http://www.dei.isep.ipp.pt/~psousa/aulas/EINF/GuiaoXDE.pdf

Índice

1 Interface Overview

Quick visual overview of the Rational XDE Developer interface in the Microsoft Visual Studio .NET IDE and the various user interface elements referred to in this guide.

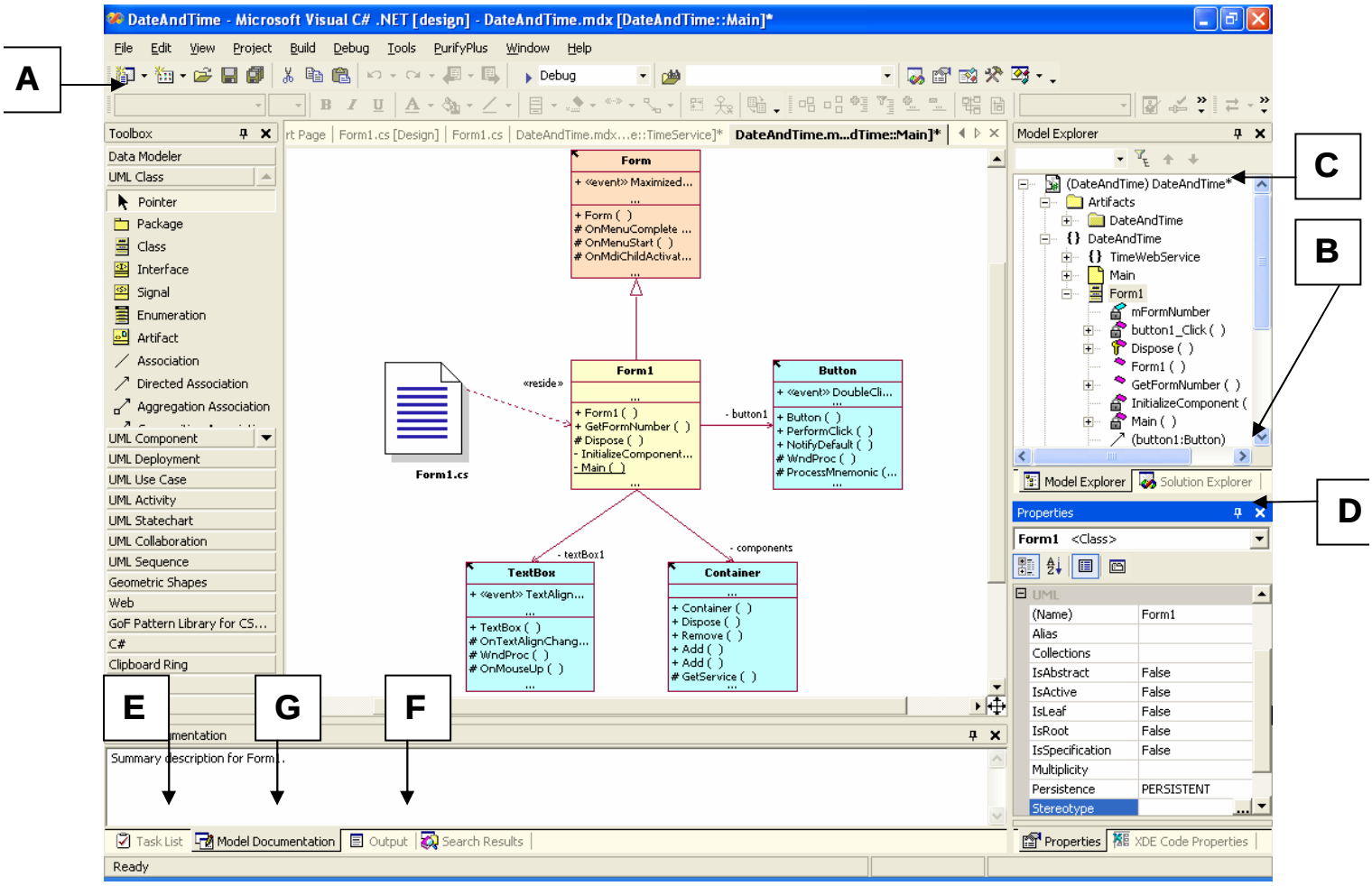

Figura 1 – interface gráfica do Rational XDE

A. The **Toolbox** is graphical representation of all the possible modeling elements the user can drag onto a modeling diagram. It is grouped by modeling categories such as UML diagram types and General objects used in free-form models. The up/down arrows provide navigation for lengthy lists of modeling elements. Users can add their own Toolbox categories as well as move elements between categories.

B. The **Solution Explorer** provides a view of the entire project or solution. Multiple projects can be viewed at any time. A project stores a collection of files that can include references, models, source code, storage units, text files, and other project-related artifacts.

C. The **Model Explorer** displays all elements that can be associated with a project, including pattern assets, diagrams, other models, and external documentation. Users can drag model elements from the Model Explorer onto

a diagram. Similar to the Solution Explorer, the Model Explorer enables you to view and modify multiple models at any given time.

D. The **Properties window** is context-sensitive and will display all the given properties for a selected element. These properties can be modified from within the Properties window. When you create patterns, a Properties window specific to pattern creation is opened.

E. The **Task window** shows model validation, compilation, and build errors. Users can also enter their own tasks. The tasks that appear in the window are context-sensitive: when you are viewing source code, the Task window will display compilation and build errors and warnings; when you are viewing model diagrams, model validation errors and warnings will be displayed.

F. The **Output window** is a log window that displays results of various program actions, such as the creation or modification of a new project, model file, model element, and so on.

G. The **Model Documentation window** displays documentation at the model element level.

- Para apagar elementos dos diagramas usar "Del". Para apagar um elemento do modelo usar "Ctrl + Del".

2 Exemplo 1

Criar um novo projecto no Visual Studio usando `File` → `New` → `Project` → `Visual C# : windows application` e denominar "HelloWorld".
Colocar um *label* no formulário como se vê na figura seguinte.

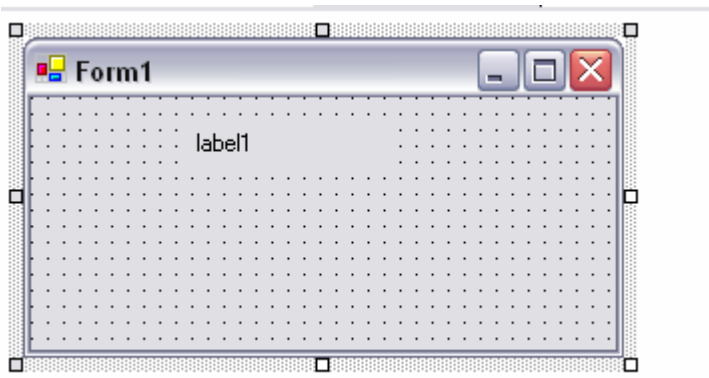

Figura 2 – interface da aplicação exemplo 1

No *handler* do evento de `Load` do formulário adicionar o seguinte código

```
private void Form1_Load(object sender, System.EventArgs e)
{
        label1.Text = "Olá Mundo";
}
```

Compilar e testar.

Gerar o modelo inicial desta aplicação efectuando uma aplicação de sincronização de código (neste caso *reverse engineering*) (Figura 3)

Figura 3 - Solution Explorer

No solution explorer escolher o separador Model Explorer.

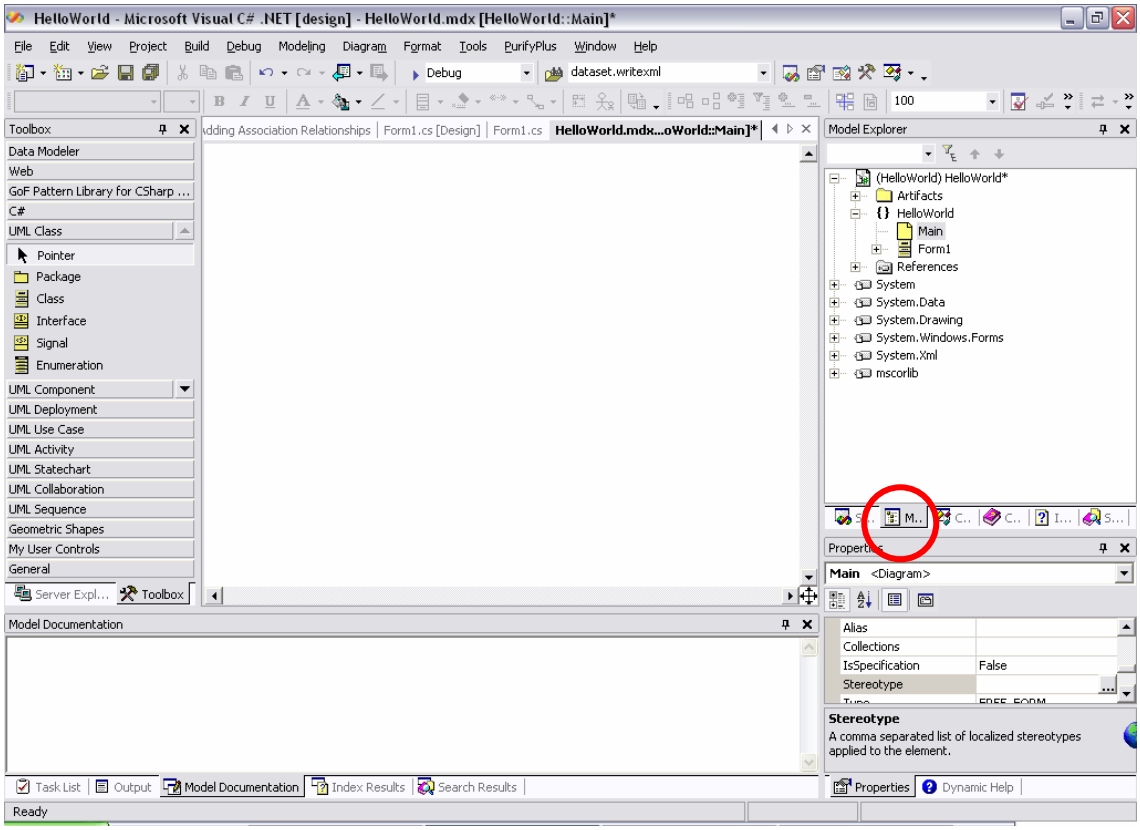

Figura 4

Duplo click no diagrama “Main”
Arrastar a classe “Form1” para o diagrama

Com o botão do lado direito no diagrama escolher `Add` → `C#` → `Class`

Figura 5 - Add C# class

Com o botão do lado direito em cima da nova classe escolher `Add` → `C#` → `Property`

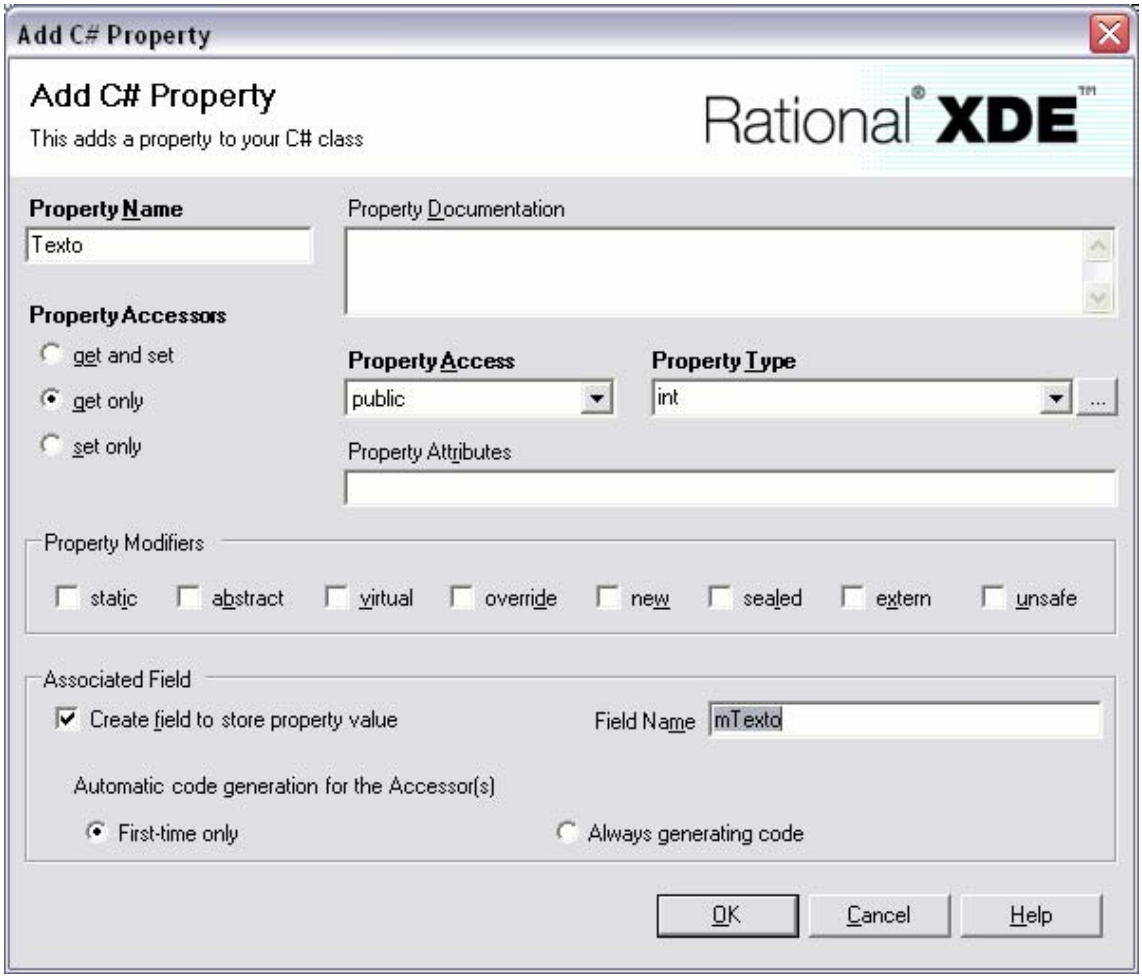

Figura 6 - Add C# Property

Na *toolbox* procurar o símbolo de associação de dependência e criar uma associação entre as duas classes.
O diagrama deve estar como na seguinte figura

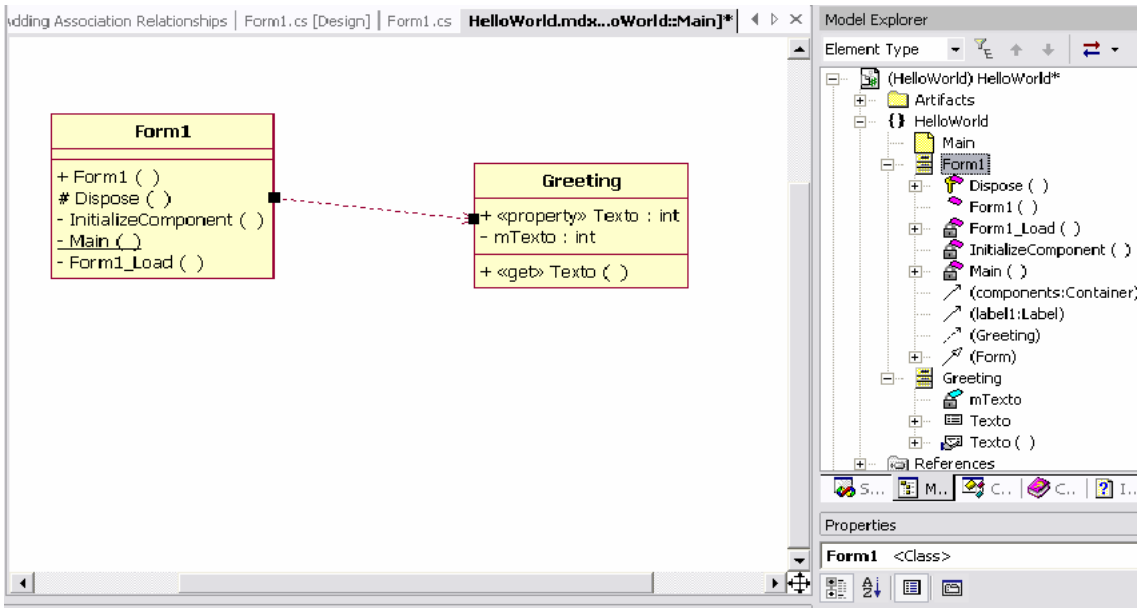

Figura 7 - diagrama de classes da aplicação exemplo 1

Com o botão do lado direito em cima do projecto no `Model Explorer` escolher `Generate code`.
Escolher a vista de `Class View`. E procurar o código da nova classe `Greeting` gerada automaticamente pelo XDE (*forward engeneering*)

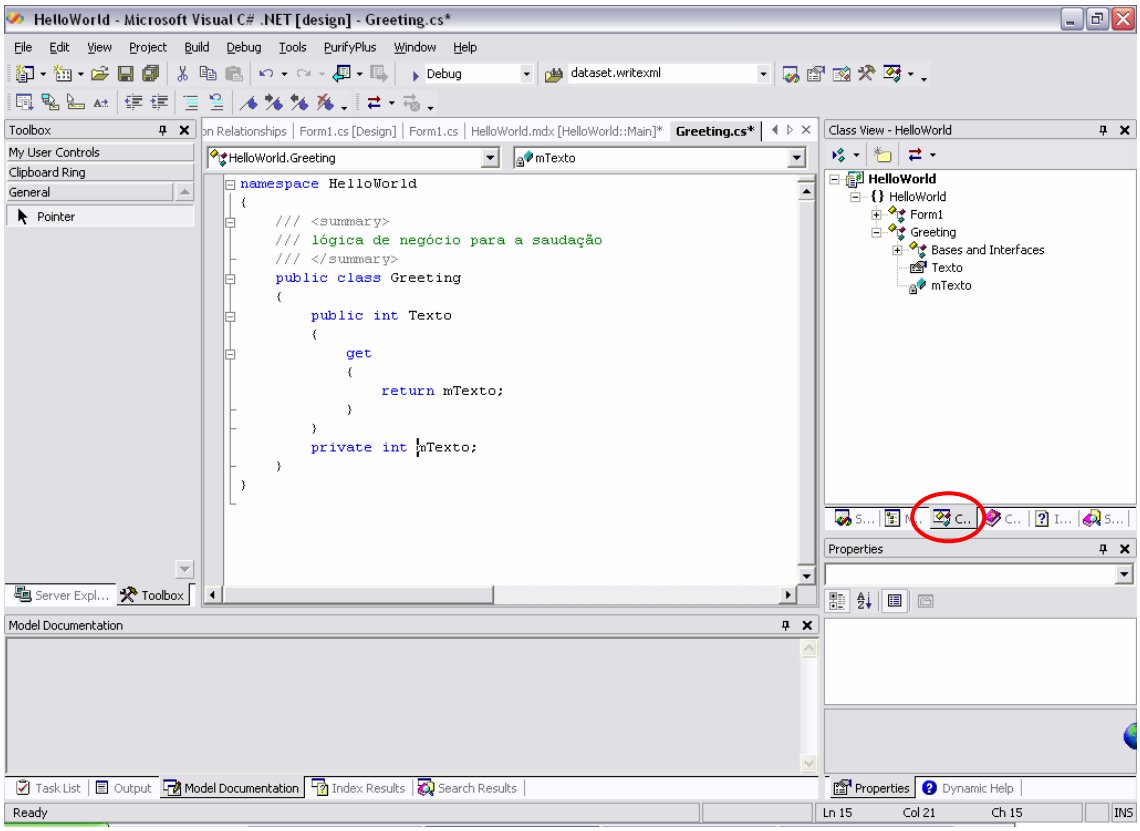

Figura 8

Alterar o tipo de dados da variável `mTexto` e da propriedade Texto para `string` e fazer as seguintes alterações

```
public class Greeting
{
        public string Texto
        {
                get
                {
                        return mTexto;
                }
        }
        private string mTexto = "Hello!!!!";
}
```

Na classe `Form1` fazer as seguintes alterações

```
private void Form1_Load(object sender, System.EventArgs e)
{
        Greeting bl = new Greeting();
        label1.Text = bl.Texto;
}
```

Compilar e testar.

Seleccionar de novo o `Model Explorer` e verificar que o diagrama ainda não está actualizado com as alterações efectuadas. Com o botão do lado direito escolher `syncronize`. O diagrama fica agora com as alterações efectuadas.

3 Exemplo 2 – Tarefas

3.1 Introdução

Modelar e codificar um sistema informático que possibilite ao utilizador introduzir e remover tarefas. A introdução consiste em três dados:

- *Prioridade* : Identifica o nível de prioridade para executar a tarefa;
- *Descrição* : Uma breve descrição do que consiste a tarefa;
- *Data Execução* : Data em que deverá ser executada a tarefa.

O sistema deverá sempre que seja criada uma tarefa, atribuir um identificador único e a data em que foi criada a tarefa.

A aplicação deverá permitir que só no fim da sessão o utilizador decida se o seu trabalho é gravado para posterior utilização ou não.
Durante a sessão de trabalho se o utilizador remover alguma tarefa deverá ter a possibilidade de a recuperar, sem que para isso tenha de cancelar todo o seu trabalho. Após a gravação da sessão deixa de ser possível recuperar as tarefas apagadas.

Ao iniciar a sessão deverão ser mostradas ao utilizador todas as tarefas existentes por ordem de entrada no sistema.

3.2 Passo 1 - Criar projecto base

Criar uma nova solução

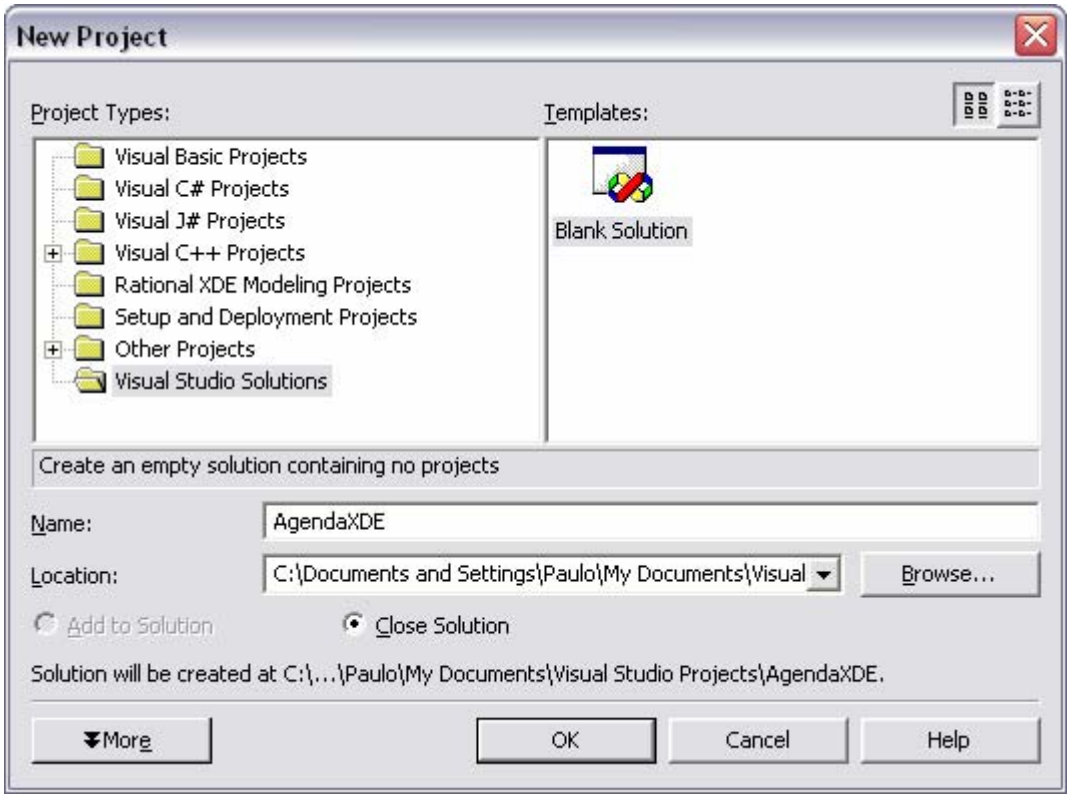

Figura 9

Adicionar um projecto windows à solução

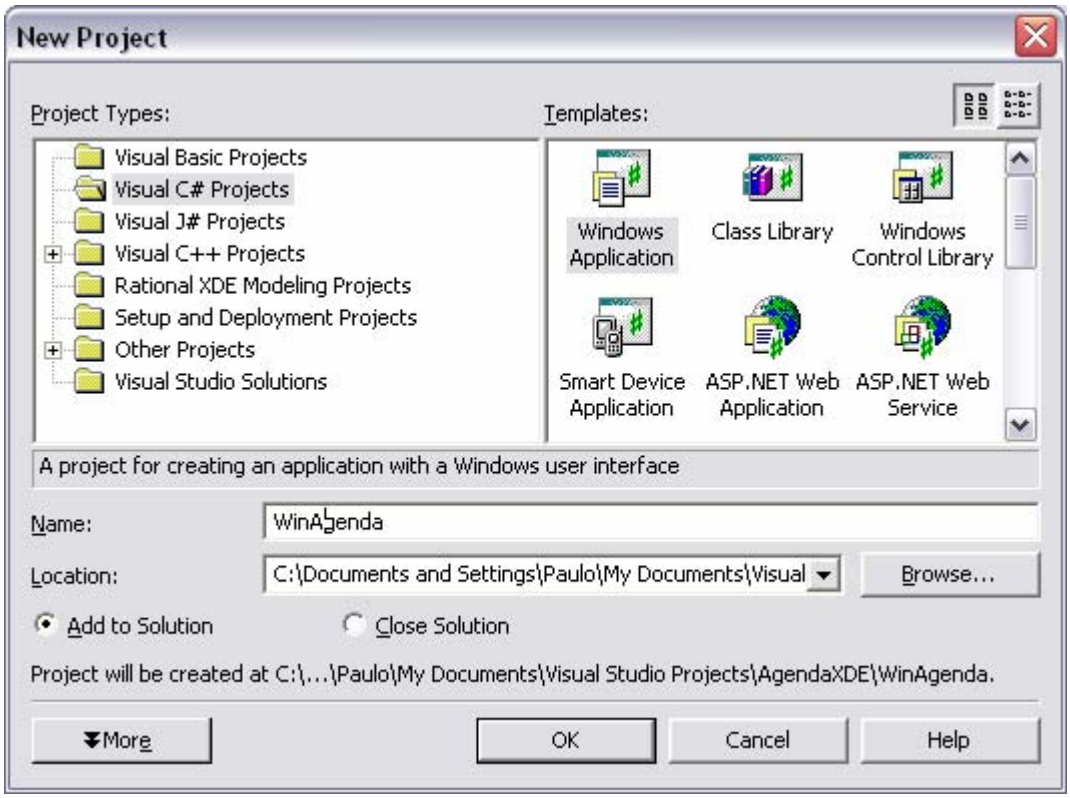

Figura 10

Efectuar sincronização de código e modelo para gerar o modelo inicial deste projecto

3.3 Passo 2 - Definir os use cases, os actores e o protótipo

Com o botão do lado direito adicionar um package denominado “Casos de uso” ao modelo. Nesse package criar um diagrama de use cases e chamar-lhe “Casos de Uso”.

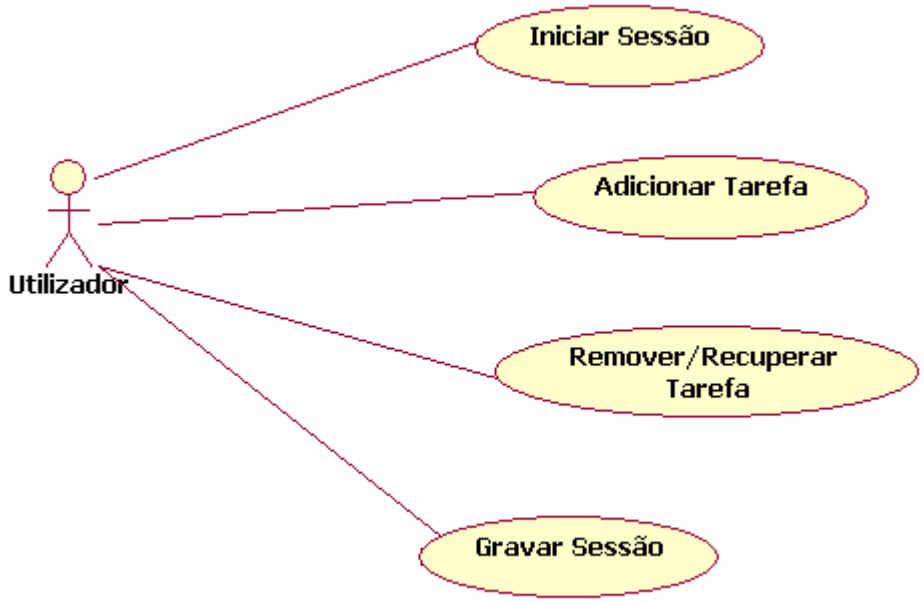

Figura 11 – diagrama de caso de uso da aplicação exemplo Tarefas

Para cada use case inserir a seguinte descrição usando a janela Model Documentation (`View` → `Other windows` → `Model documentation`) (Figura 12)

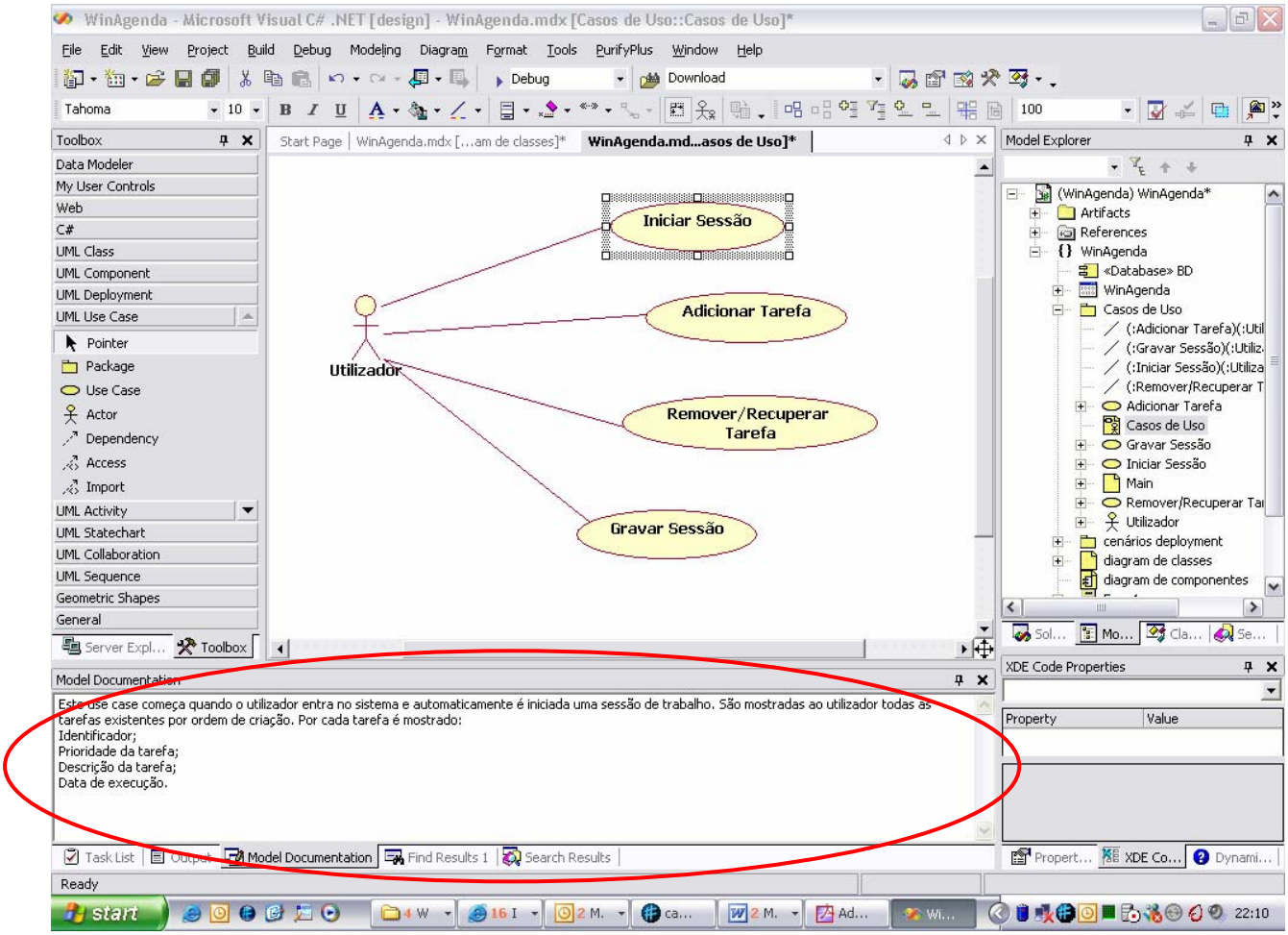

Figura 12

3.3.1 Use Case Iniciar Sessão

Este use case começa quando o utilizador entra no sistema e automaticamente é iniciada uma sessão de trabalho. São mostradas ao utilizador todas as tarefas existentes por ordem de criação. Por cada tarefa é mostrado:

- Identificador;
- Prioridade da tarefa;
- Descrição da tarefa;
- Data de execução.

3.3.2 Use Case Adicionar Tarefa

Este use case começa quando o utilizador decide introduzir uma nova tarefa.
Para isso introduz a prioridade, a descrição da tarefa e a data de execução. Em seguida o utilizador selecciona o botão adicionar.
O sistema não faz qualquer validação aos dados introduzidos, mas atribui um identificador único e uma data de criação. Após a introdução da tarefa esta é visualizada juntamente com as restantes por ordem de entrada.

3.3.3 Use Case Remover/Recuperar

Este use case começa quando o utilizador decide remover uma tarefa. O utilizador selecciona uma tarefa da lista e de seguida carrega no botão Remover/Recuperar, se a tarefa já estava removida então será recuperada, senão será removida.
Se não for seleccionada nenhuma tarefa então nada acontecerá.

3.3.4 Use Case Gravar Sessão

Este use case começa quando o utilizador decide gravar todo o trabalho que fez durante a sessão.
Todo o seu trabalho será guardado numa base de dados para posterior utilização: As novas tarefas serão guardadas e as apagadas serão removidas fisicamente da base de dados.

3.3.5 Actor Utilizador

O utilizador é a pessoa que pode executar todas as opções relacionadas com as tarefas, é o único perfil do sistema.

3.4 Passo 3 – definir protótipo de interface com utilizador

Abrir o *form* da aplicação e desenhar o interface

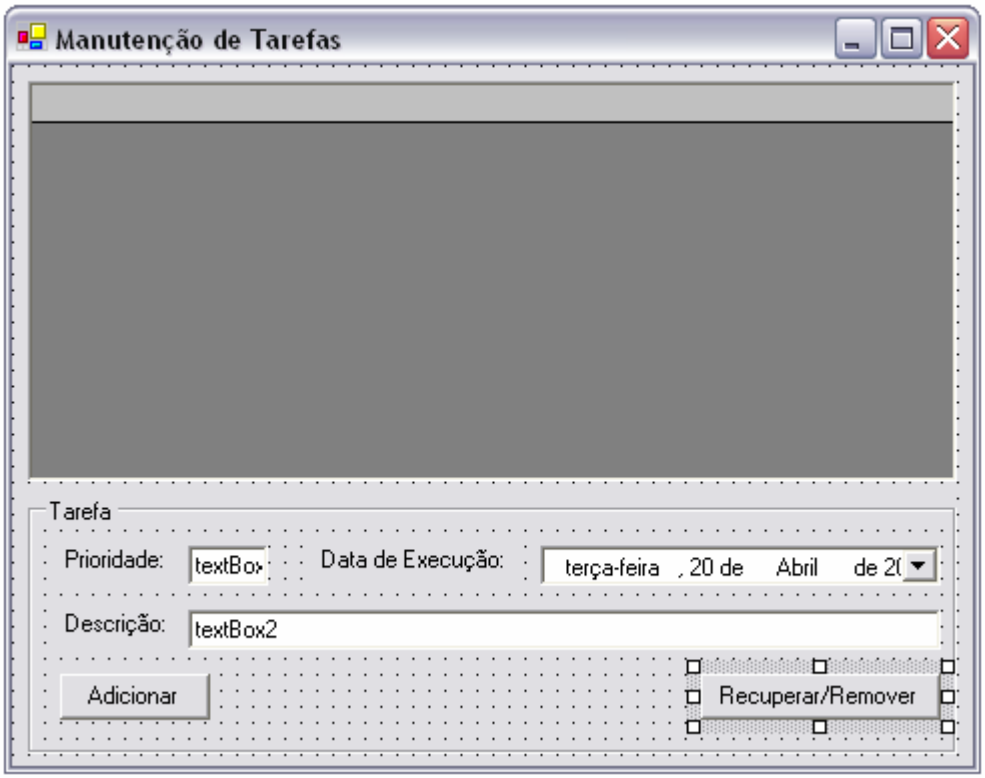

Figura 13 - interface gráfica da aplicação exmeplo Tarefas

Criar para este *form* um *handler* do evento `Load` fazendo duplo click numa zona livre do *form* ou na janela de propriedades seleccionar o evento. Criar igualmente os *handlers* para os botões fazendo duplo click em cima de cada botão.

Sincronizar o código com o modelo.

Este passo facilita a criação dos diagramas de sequência pois permite escolher estes métodos automaticamente. Caso fossem criados depois implicariam trabalho adicional para os indicar como *handlers*.

3.5 Passo 3 – criar diagrama de classes inicial

No `model explorer` seleccionar o diagrama de classes. Adicionar a classe `Form1` ao diagrama. Criar duas novas classes `ListaDeTarefas` e Tarefa para representar o núcleo do problema e adicionar uma terceira chamada `ServicoDados` para tratar da persistência da informação na BD.

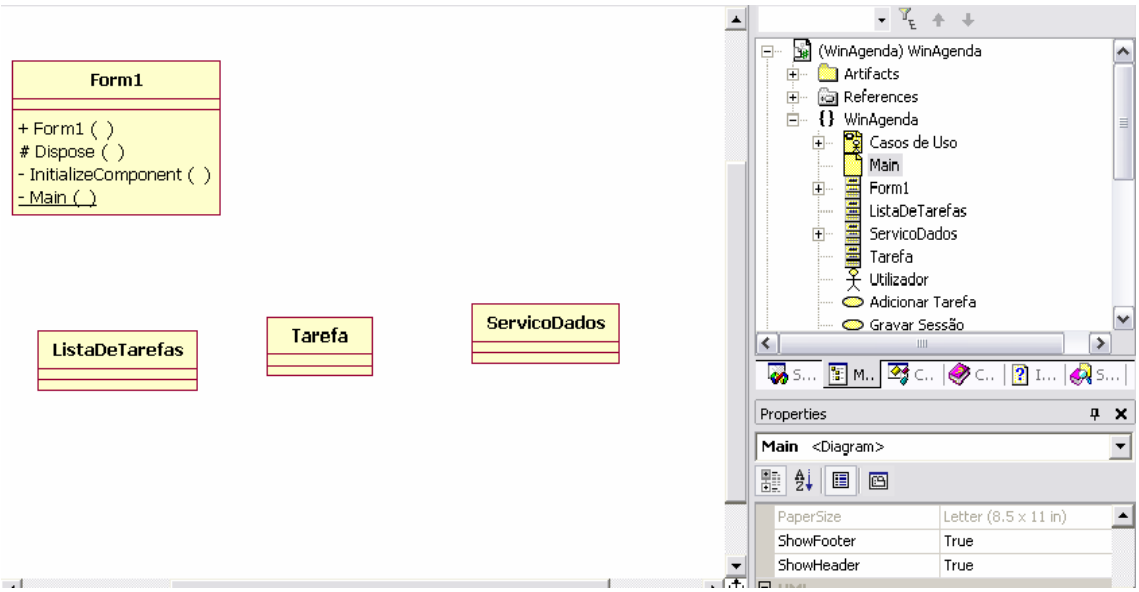

Figura 14

3.6 Passo 4 – definir processos

Com o botão do lado direito em cima de cada *use case* seleccionar `Add diagram` → `Sequence : Role`. Definir os diagramas de sequência para cada use case conforme o descrito em seguida:

- Para colocar objectos no diagrama arrastar do modelo para o diagrama.
- Para criar mensagens arrastar da *toolbox*.
- Em cada mensagem pode-se escolher da *drop down* qual o método a invocar ou colocar um texto correspondente a novo método.
- Para converter a mensagem num método, seleccionar com o botão do lado direito e escolher `Create Operation from Message`.
- Para as operações de criação de objectos (ex., `new`) existe uma mensagem específica na *toolbar* denominada "Create". Estas operações aparecem nos diagramas seguintes com a mensagem "`/CreateOperation/`"

3.6.1 Iniciar Sessão

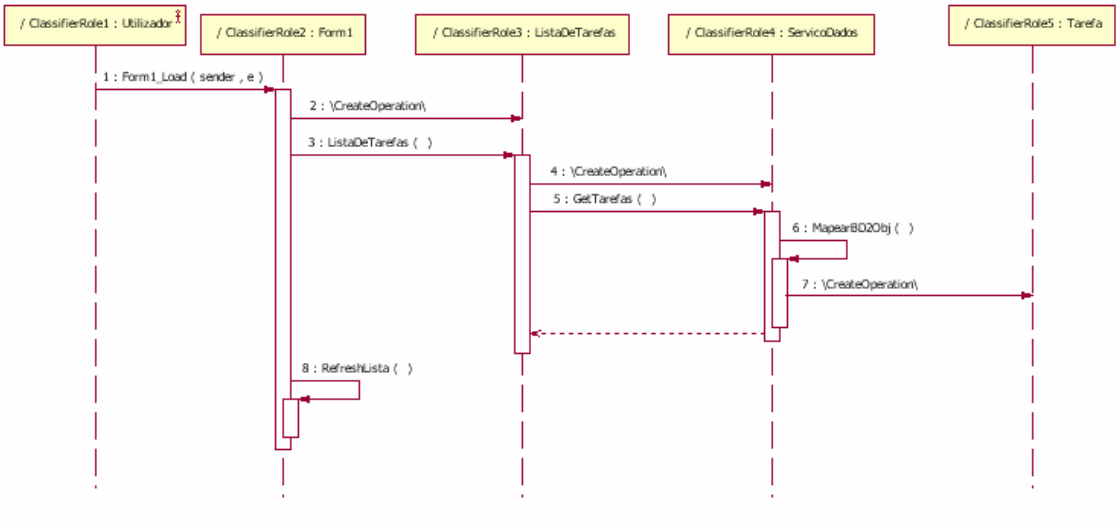

Figura 15 - diagrama de sequência para caso de uso Iniciar Sessão

3.6.2 Adicionar Tarefa

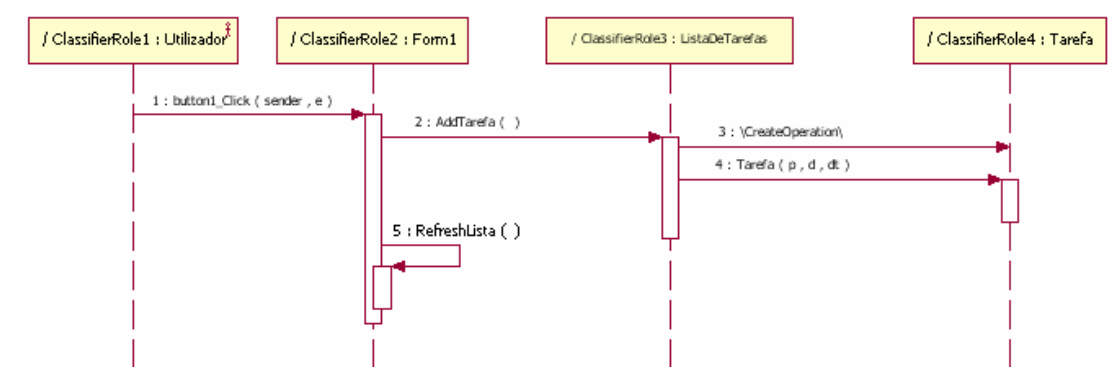

Figura 16 - diagrama de sequência para caso de uso Adicionar Tarefa

De notar que a mensagem n.º 4 possui parâmetros. Para colocar os parâmetros na mensagem o ideal é a operação já estar definida anteriormente com os respectivos argumentos. Para isso devem ir ao `Model Explorer` e com o botão do lado direito em cima da classe desejada (neste caso `Tarefa`) escolher `Collection Editor`.
Na janela de diálogo que vos aparece escolham o separador `Operations`. Se ainda não criaram a operação `Tarefa` (construtor da classe), criem essa operação (usando o botão identificado pelo círculo vermelho na Figura 17).

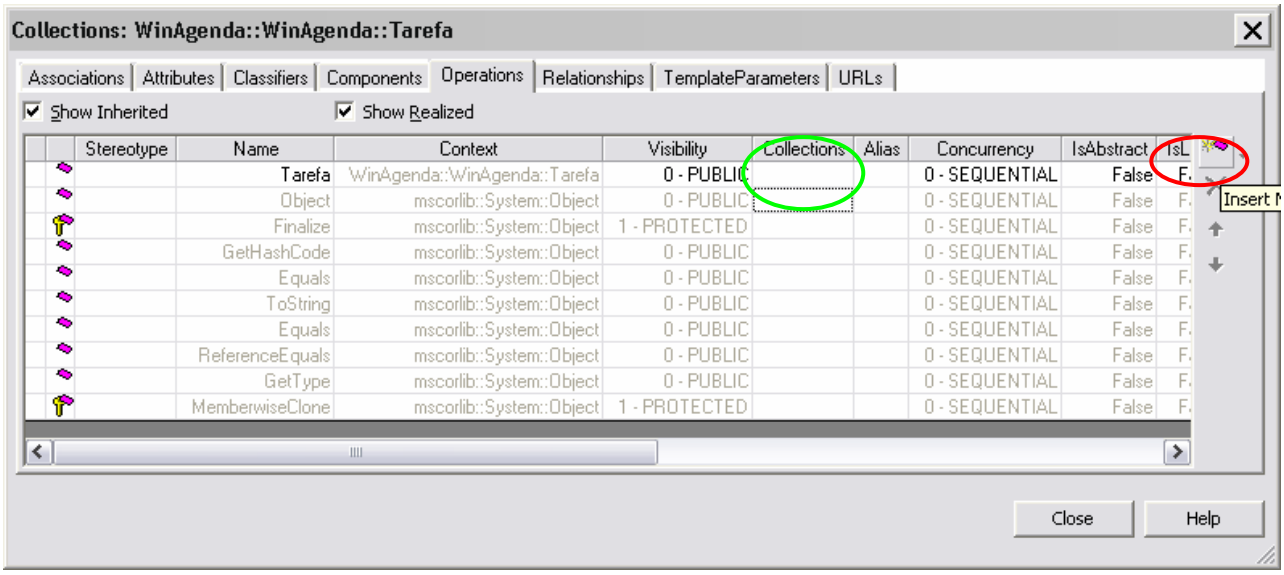

Figura 17 - criar nova operação numa classe

Após criarem a operação acrescentem os parâmetros escolhendo o botão identificado pelo círculo verde na Figura 17.

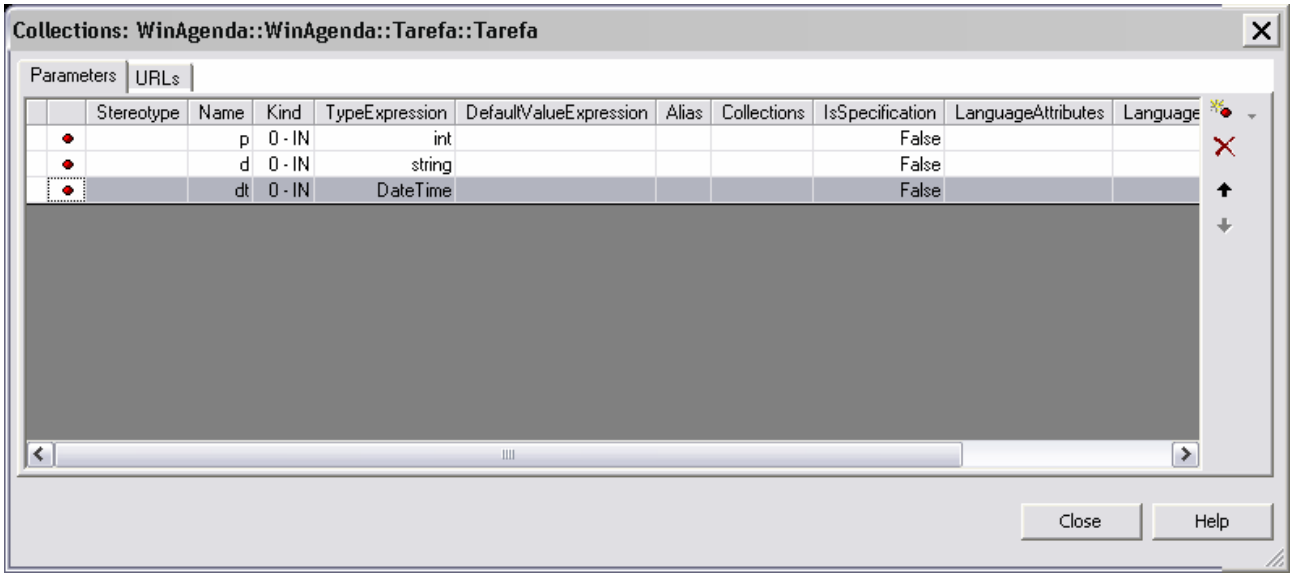

Figura 18 - parâmetros de uma operação

3.6.3 Remover/Recuperar

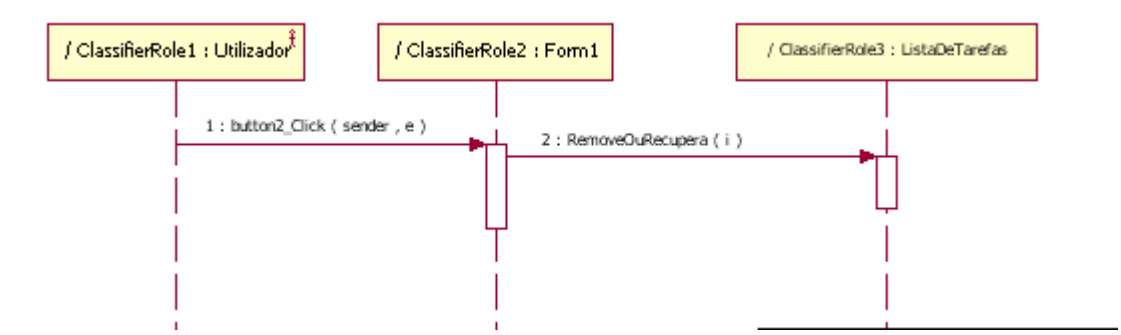

Figura 19 - diagrama de sequência para caso de uso Remover/recuperar

3.6.4 Gravar Sessão

Seleccionar o ficheiro com o formulário da aplicação. Na janela de propriedades seleccionar o separador de eventos e criar um *handler* para o evento `closed`.

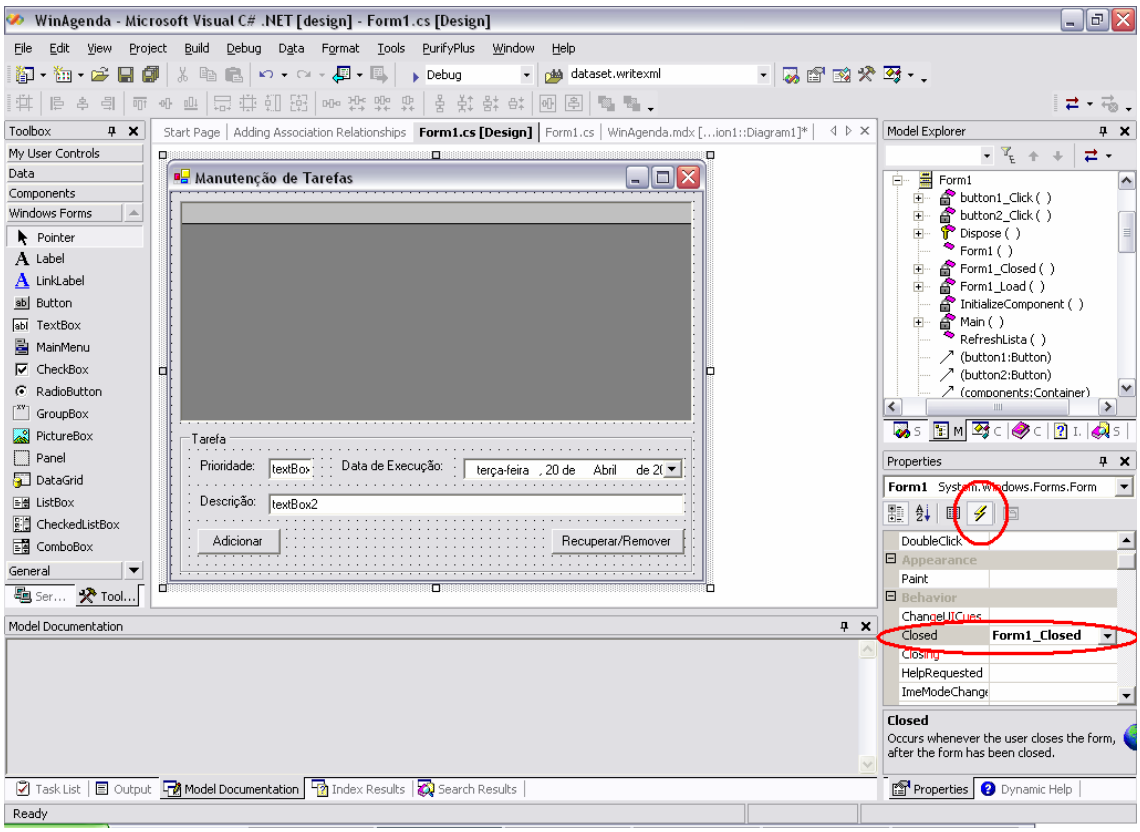

Figura 20

Sincronizar o código com o modelo.
Criar o seguinte diagrama de sequência

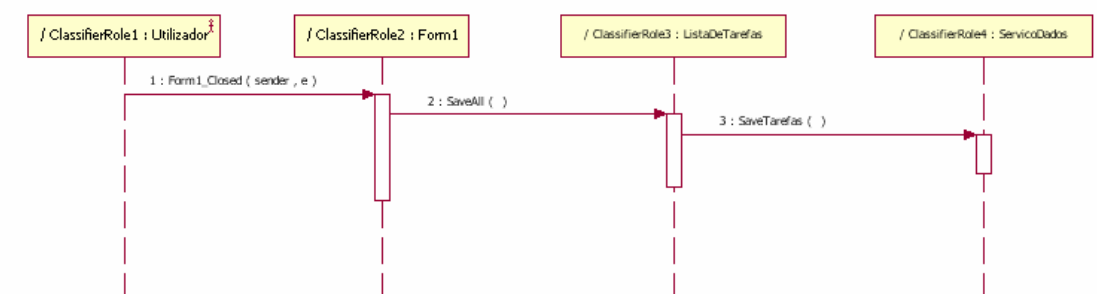

Figura 21 - diagrama de sequência para caso de uso Gravar Sessão

3.7 Passo 5 – definir diagrama de classes completo

Criar as associações existentes entre as diversas classes.

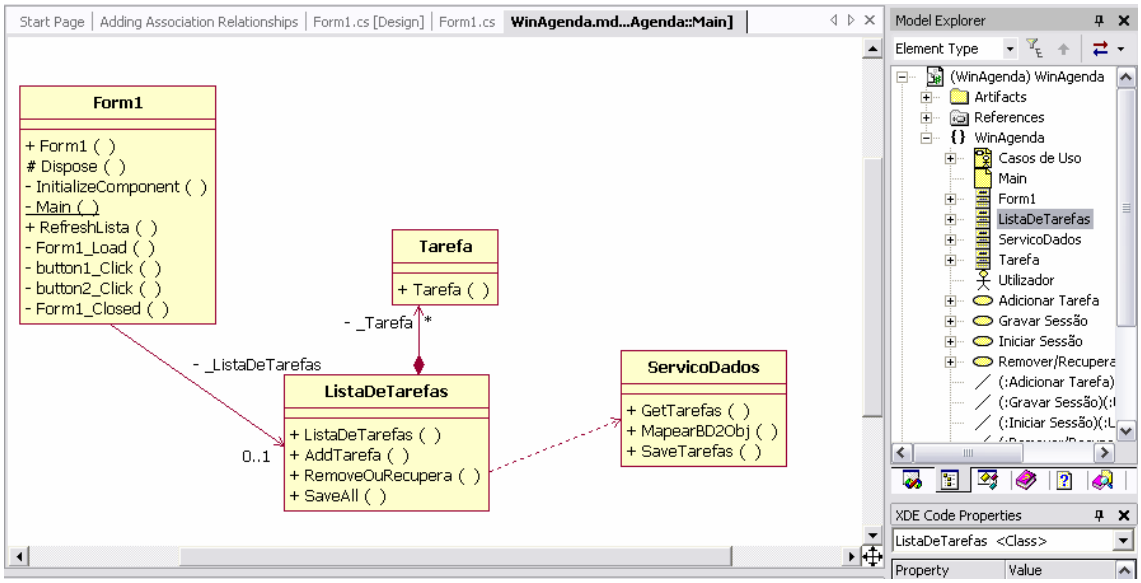

Figura 22- diagrama de classes para aplicação exemplo Tarefas

Seleccionar o diagrama de classes. Com o botão do lado direito sobre a classe `ServicoDados` escolher `Collection Editor`.

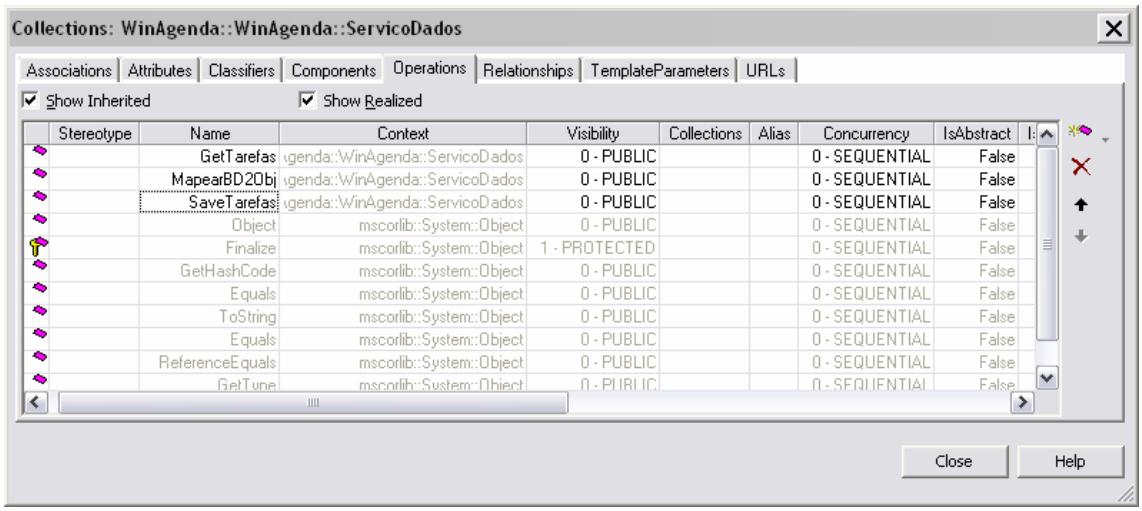

Figura 23

Seleccionar a célula chamada `collections` para a operação `SaveTarefas`. Acrescentar um parâmetro chamado lista com tipo `IList`

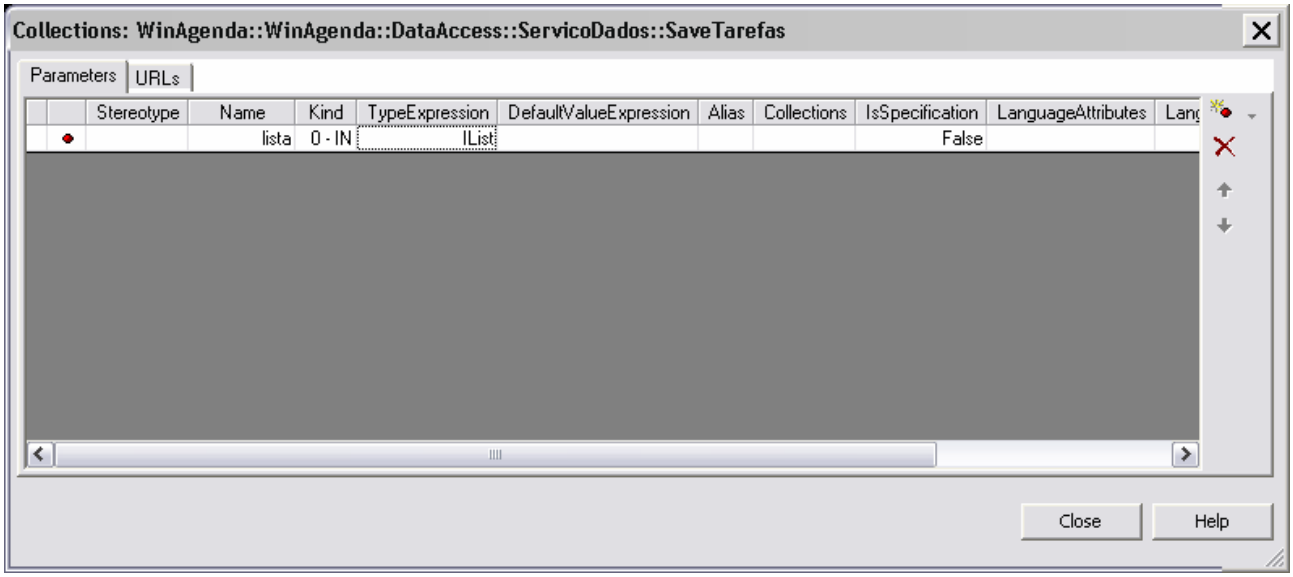

Figura 24

Sincronizar código. Compilar.

Deve dar um erro na classe `ServicoDados` por não reconhecer o tipo `IList`.
Adicionar ao ficheiro a seguinte linha de código (no início)

```
using System.Collections;
```

Caso não tenham especificado os tipos dos argumentos de algumas mensagens irão aparecer *warnings* e esses argumentos serão tratados como inteiros que poderão ser depois alterados e efectuar nova sincronização.

3.8 Passo 6 – modelar estado das principais entidades

A entidade principal do sistema é a classe `Tarefa`. Tendo em conta que é uma entidade para a qual é necessário gerir o seu estado por estarmos a trabalhar com sessões vai-se adicionar um diagrama de estados.
Para tal, com o botão do lado direito do rato em cima da classe `Tarefa` no `Model Explorer` seleccionar a opção `Add Diagram` → `statechart`.
No menu `Tools` → `Options` escolher o separador `Rational XDE` → `Appearance` → `Uml Activities/states` e activar a opção `Show Names` para as transições.

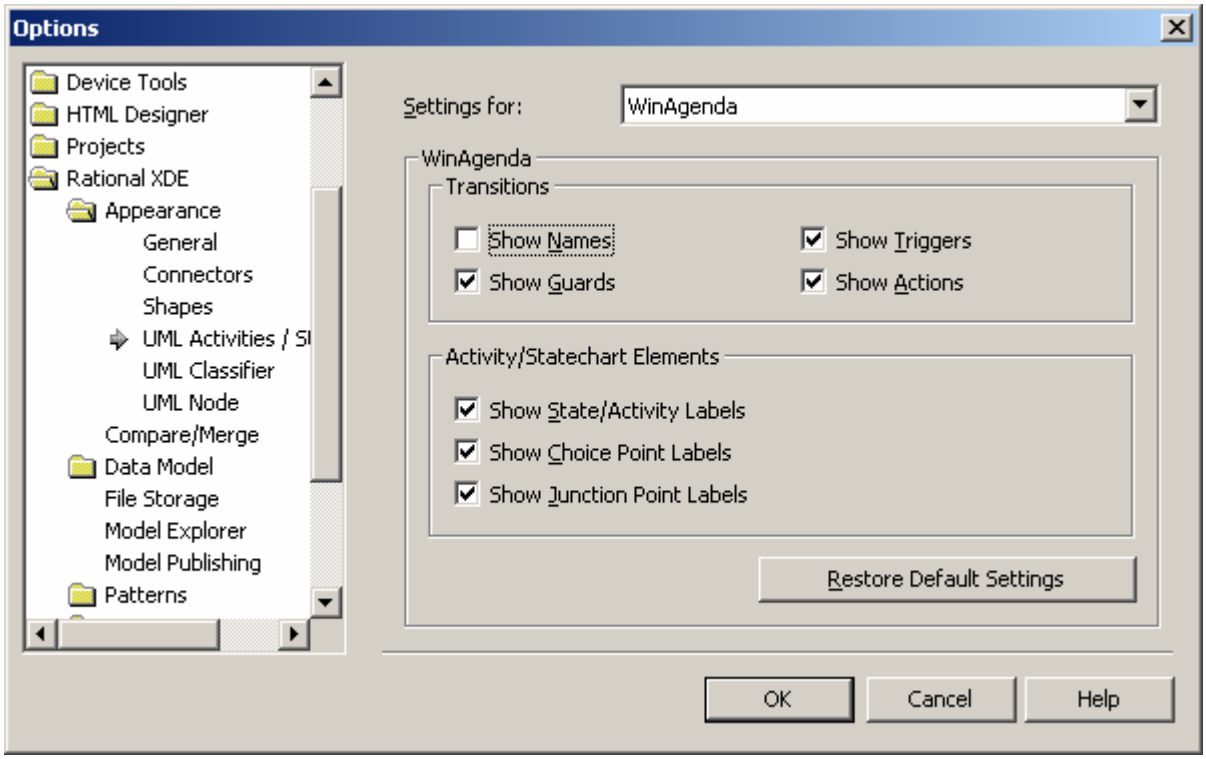

Figura 25

Modelar o estado da classe `Tarefa` como se mostra no seguinte diagrama

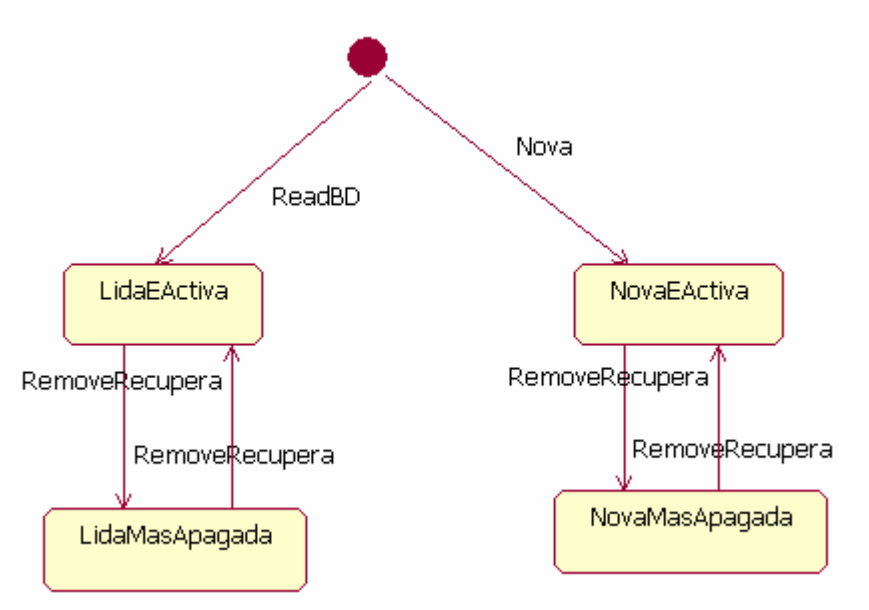

Figura 26 - diagrama de estado da classe Tarefa

3.9 Passo 7 – implementar código em falta

Seleccionar `Class view` no `solution explorer`. Verificar o código fonte gerado após sincronização para as operações e associações definidas no diagrama de classes.

```
public class Tarefa
{
      public Tarefa(int p, string d, DateTime dt)
      {
      }
}

public class ListaDeTarefas
{
      public ListaDeTarefas()
      {
      }
      public void AddTarefa()
      {
      }
      public void RemoveOuRecupera(int i)
      {
      }
      public void SaveAll()
      {
      }
      private Tarefa _Tarefa;
}

public class ServicoDados
{
      public void GetTarefas()
      {
      }
```

```
            public void MapearBD2Obj()
            {
            }
            public void SaveTarefas(IList lista)
            {
            }
      }

      public class Form1 : System.Windows.Forms.Form
      {
            private System.Windows.Forms.DataGrid dataGrid1;
            private System.Windows.Forms.GroupBox groupBox1;
            private System.Windows.Forms.Label label1;
            private System.Windows.Forms.TextBox textBox1;
            private System.Windows.Forms.Label label2;
            private System.Windows.Forms.TextBox textBox2;
            private System.Windows.Forms.Label label3;
            private System.Windows.Forms.DateTimePicker dateTimePicker1;
            private System.Windows.Forms.Button button1;
            private System.Windows.Forms.Button button2;
            /// <summary>
            /// Required designer variable.
            /// </summary>
            private System.ComponentModel.Container components = null;

            public Form1()
            {
                  //
                  // Required for Windows Form Designer support
                  //
                  InitializeComponent();

                  //
                  // TODO: Add any constructor code after InitializeComponent
call
                  //
            }

            public void RefreshLista()
            {
            }

            [...]

            private void Form1_Load(object sender, System.EventArgs e)
            {
            }
            private void button1_Click(object sender, System.EventArgs e)
            {
            }
            private void button2_Click(object sender, System.EventArgs e)
            {
            }
            private void Form1_Closed(object sender, System.EventArgs e)
            {
            }
            private ListaDeTarefas _ListaDeTarefas;
      }
```

Conforme se pode ver o tipo gerado para a associação entre `ListadeTarefas` e `Tarefa` não é adequado, pois não permite cardinalidade superior a um. Voltando ao diagrama

de classes, seleccionar a associação entre `ListaDeTarefas` e `Tarefa` e no menu de contexto seleccionar `Code Properties`. Isto faz aparecer a janela de propriedades de código (`XDE Code Properties`); alterar o tipo para `IList` e o valor inicial para `new ArrayList()`.

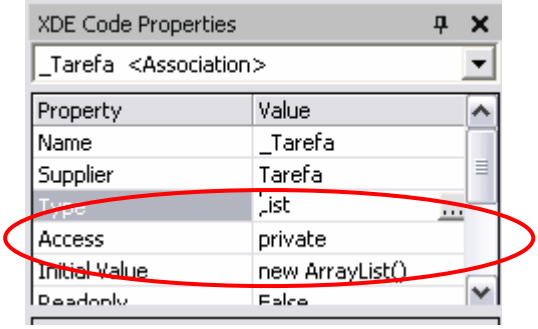

Figura 27

Sincronizar e compilar. Corrigir os erros colocando as directivas de `using` correctas (`System` para a classe `DateTime` e `System.Collections` para a interface `IList`).

Tabela 1

Nome da propriedade	get	set
Descricao	V	V
Priporidade	V	V
Data	V	V
Nova	V	
Apagada	V	V
DataCriacao	V	

No diagrama de classes, seleccionar a classe `Tarefa` e criar as propriedades públicas C# listadas na Tabela 1, criando para cada propriedade uma variável como se vê na figura seguinte.

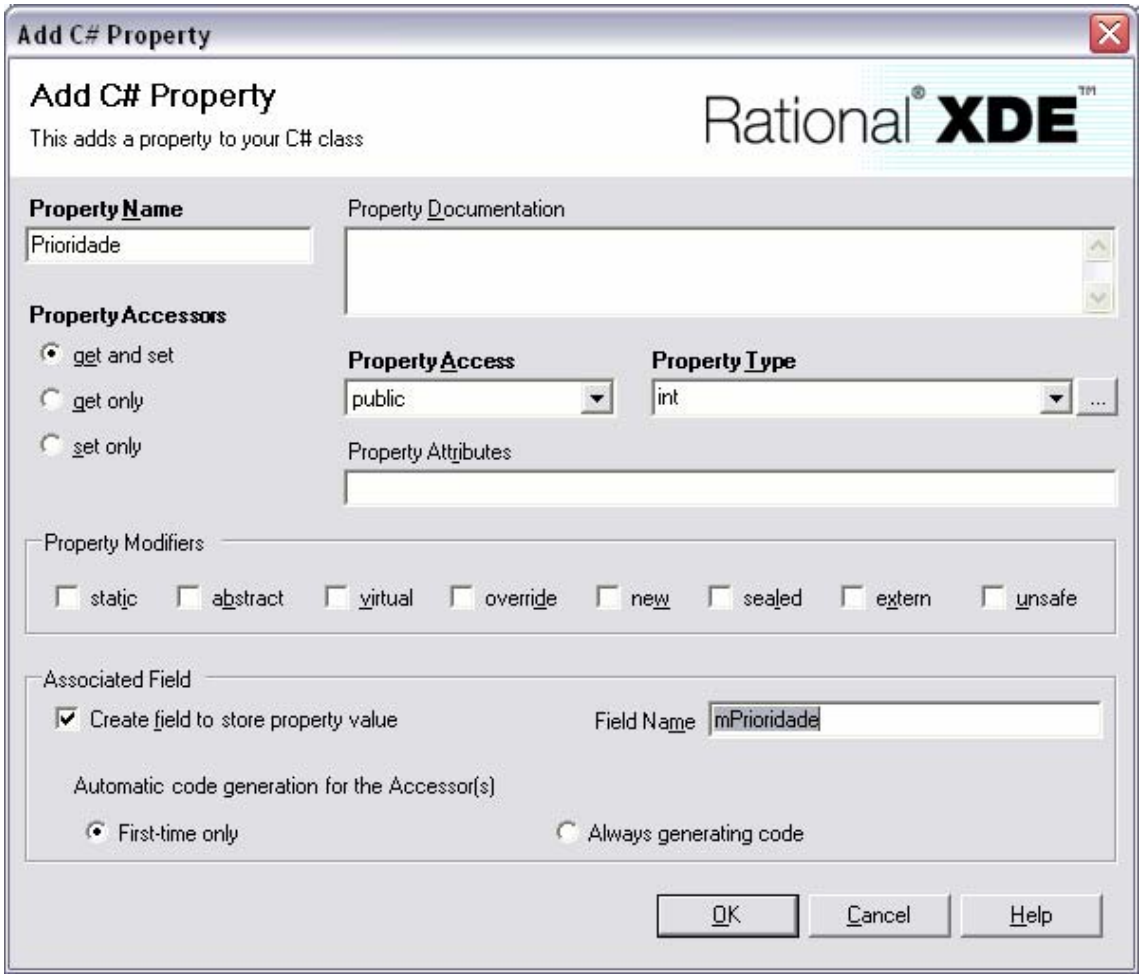

Figura 28

A classe deve ter o seguinte código:

```
public class Tarefa
{
        public Tarefa(int p, string d, DateTime dt)
        {
        }

        public string Descricao
        {
                get
                {
                        return mDescricao;
                }
                set
                {
                        mDescricao = value;
                }
        }

        public int Prioridade
        {
                get
                {
                        return mPrioridade;
                }
```

```
            set
            {
                mPrioridade = value;
            }
        }
        public DateTime Data
        {
            get
            {
                return mData;
            }
            set
            {
                mData = value;
            }
        }
        public bool Nova
        {
            get
            {
                return mNova;
            }
        }
        public bool Apagada
        {
            get
            {
                return mApagada;
            }
            set
            {
                mApagada = value;
            }
        }
        public DateTime DataCriacao
        {
            get
            {
                return mDataCriacao;
            }
        }
        private string mDescricao;
        private int mPrioridade;
        private DateTime mData;
        private bool mNova;
        private bool mApagada;
        private DateTime mDataCriacao;
    }
```

Alterar o construtor da classe

```
    public Tarefa(int p, string d, DateTime dt)
    {
        mPrioridade = p;
        mDescricao = d;
        mData = dt;

        mNova = true;
        mApagada = false;
        mDataCriacao = DateTime.Now;
    }
```

3.10 Passo 8 – base de dados

Criar um base de dados (por exemplo, em Access) contendo uma tabela denominada Tarefas com os seguintes campos.

Tabela 2

Campo	Tipo de dados
ID	AutoNumber
Prioridade	Long Integer
Designacao	Text (50)
Data	Date/Time
DataCriacao	Date/Time

3.11 Passo 9 – implementar código de cada processo

Os processos existentes no sistema foram já identificados nos diagramas de sequência correspondentes a cada caso de utilização.

Efectuar as seguintes alterações nas classes de acordo com os diagramas de sequência:

3.11.1 Iniciar Sessão

Na classe `Form1` alterar o método `Form1_Load`

```
private void Form1_Load(object sender, System.EventArgs e)
{
	// criar lista de tarefas
	_ListaDeTarefas = new ListaDeTarefas();

	//visualizar a lisat de tarefas
	if (_ListaDeTarefas.Tarefas.Count != 0)
		RefreshLista();
}
```

Para implementar o método `RefreshLista` será necessário que a classe `ListaDeTarefas` exponha o seu conteúdo. Para tal alterar a classe `ListaDeTarefas` criando a seguinte propriedade:

```
public IList Tarefas
{
	get
	{
		return _Tarefa;
	}
}
```

O método `RefreshLista` da classe `Form1` deve então ter o seguinte código:

```
public void RefreshLista()
{
	// remover conteúdo anterior
```

```
            dataGrid1.DataSource = null;
            // visualizar o conteúdo actual
            dataGrid1.DataSource = _ListaDeTarefas.Tarefas;
      }
```

O construtor da classe `ListaDeTarefas` deve ler todas as tarefas existentes na BD.

```
      public ListaDeTarefas()
      {
            // criar componente de acesso a dados
            ServicoDados bd = new ServicoDados();

            try
            {
                  // obter lista de tarefas da BD
                  IEnumerable lidas = bd.GetTarefas();

                  // copiar para lista interna
                  foreach (Tarefa t in lidas)
                        _Tarefa.Add(t);
            }
            catch (ApplicationException ex)
            {
            }
      }
```

A classe `ServicoDados` deve ser alterada da seguinte forma*:

```
using System.Data;
using System.Data.OleDb;

      private const string CONN = @"Provider= Microsoft.Jet.OLEDB.4.0; Data
Source=../../agenda.mdb";
      private const string SQL_GETTAREFAS = "SELECT * FROM Tarefas";

      public IEnumerable GetTarefas()
      {
            try
            {
                  OleDbConnection cnx = new OleDbConnection(CONN);
                  OleDbCommand cmd = new OleDbCommand(SQL_GETTAREFAS, cnx);
                  cnx.Open();
                  OleDbDataReader rd =
cmd.ExecuteReader(CommandBehavior.CloseConnection);

                  ArrayList lst = new ArrayList();
                  while (rd.Read())
                  {
                        // ler e mapear
                        Tarefa t = MapearBD2Obj(rd);
                        //adicionar
                        lst.Add(t);
                  }
                  rd.Close();
                  return lst;
            }
            catch (OleDbException ex)
```

* Para relembrar conceitos de ADO.net aconselha-se a consulta ao guia de Referência Rápida ADO.net disponível em http://www.dei.isep.ipp.pt/~psousa/aulas/ADAV/op_adonet.html .

```
            {
                  throw new ApplicationException("Erro no serviço de dados",
ex);
                  //return null;
            }
      }

      private Tarefa MapearBD2Obj(OleDbDataReader rd)
      {
            int p = (int)rd["Prioridade"];
            string d = (string)rd["designacao"];
            DateTime dt = (DateTime)rd["data"];
            return new Tarefa(p, d, dt);
      }
```

Alterar a visibilidade do método `MapearBD2Obj` para `private`.
Compilar e experimentar a aplicação.

3.11.2 Adicionar Tarefa

Para adicionar uma tarefa será necessário tratar o click do botão. Dessa forma colocar o seguinte código na classe `Form1`.

```
      private void button1_Click(object sender, System.EventArgs e)
      {
            // obter valores
            int p = int.Parse(textBox1.Text);
            string d = textBox2.Text;
            DateTime dt = dateTimePicker1.Value;

            // adicionar
            _ListaDeTarefas.AddTarefa(p, d, dt);

            // visualzar lista
            RefreshLista();
      }
```

Na classe `ListaDeTarefas` alterar o método `AddTarefa`.

```
      public void AddTarefa(int p, string d, DateTime dt)
      {
            Tarefa t = new Tarefa(p, d, dt);

            _Tarefa.Add(t);
      }
```

Compilar e experimentar a aplicação.

3.11.3 Remover/Recuperar

Na classe `Form1` colocar o seguinte código no *handler* do evento click para o botão de Remover/Recuperar

```
      private void button2_Click(object sender, System.EventArgs e)
      {
            // obter célula seleccionada na DataGrid
            System.Windows.Forms.DataGridCell cur = dataGrid1.CurrentCell;
```

```
        // remover ou recuperar a tarefa indicada
        // (baseado na ordem na lista)
        _ListaDeTarefas.RemoveOuRecupera(cur.RowNumber);

        // visualizar lista
        RefreshLista();
    }
```

Na classe `ListaDeTarefas` colocar o seguinte código no método `RemoveRecupera`.

```
    public void RemoveOuRecupera(int i)
    {
        // obter o elemento
        Tarefa t = (Tarefa)_Tarefa[i];

        // remover ou recuperar logicamente
        t.Apagada = !t.Apagada;
    }
```

Compilar e experimentar a aplicação.

3.11.4 Gravar Sessão

Finalmente para gravar a agenda, colocar o seguinte código no *handler* do evento `close` da janela da aplicação

```
    private void Form1_Closed(object sender, System.EventArgs e)
    {
        // gravar a agenda na BD
        _ListaDeTarefas.SaveAll();
    }
```

Na classe `ListaDeTarefas` implementar o método `SaveAll`.

```
    public void SaveAll()
    {
        // criar componente de acesso a dados
        ServicoDados bd = new ServicoDados();

        // para cada elemento da lista aplicar a operação necessária
        foreach (Tarefa t in _Tarefa)
        {
            //novas tarefas
            if (t.Nova && !t.Apagada)
            {
                bd.Inserir(t);
            }
            //lidas da BD e apagadas
            if (t.Apagada && !t.Nova)
            {
                bd.Apagar(t);
            }
        }
    }
```

Como se pode ver, este código não obedece totalmente ao diagrama de sequência especificado. Como tal vamos alterar o diagrama de sequência para o caso de uso “Gravar Sessão”.

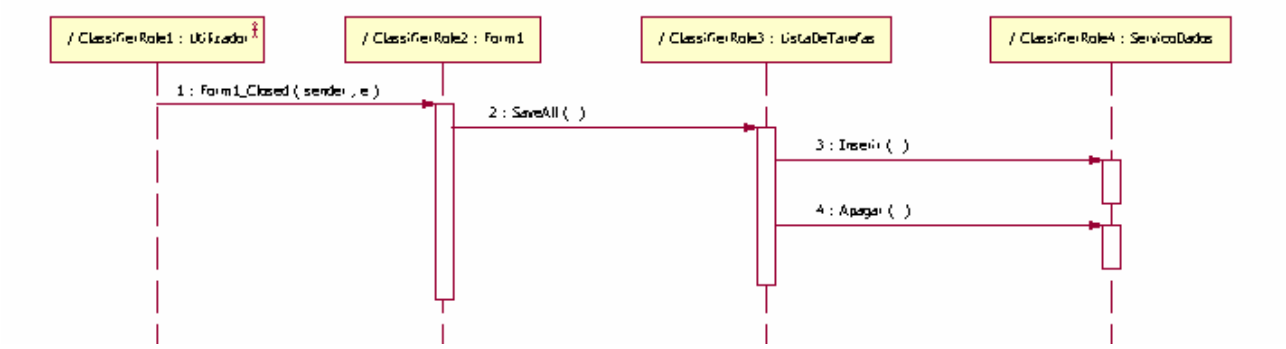

Figura 29 – diagrama de sequência para caso de uso Gravar Sessão

Em seguida, na classe `ServicoDados` implementar os seguintes métodos

```
private const string SQL_INSERTTAREFA = "INSERT INTO Tarefas (Prioridade, Designacao, Data, DataCriacao) VALUES (@Prioridade, @Designacao, @Data, @DataCriacao)";

public void Inserir(Tarefa t)
{
	try
	{
		OleDbConnection cnx = new OleDbConnection(CONN);
		OleDbCommand cmd = new OleDbCommand(SQL_INSERTTAREFA, cnx);

		MapearObj2BD(t, cmd);

		cnx.Open();
		cmd.ExecuteNonQuery();
		cnx.Close();
	}
	catch (OleDbException ex)
	{
		throw new ApplicationException("Erro no serviço de dados ao inserir registo", ex);
	}
}

private void MapearObj2BD(Tarefa t, OleDbCommand cmd)
{
	cmd.Parameters.Add("@Prioridade", t.Prioridade);
	cmd.Parameters.Add("@Designacao", t.Descricao);
	OleDbParameter parm = cmd.Parameters.Add("@Data", OleDbType.DBDate);
	parm.Value = t.Data;
	parm = cmd.Parameters.Add("@DataCriacao", OleDbType.DBDate);
	parm.Value = t.DataCriacao;
}
```

Retirar da classe `ServicoDados` o método anteriormente gerado `SaveTarefas`. Compilar a aplicação, efectuar sincronização de código com modelo.
Verificar o modelo e as alterações introduzidas na sessão de codificação (ou seja, verificar que o processo de *reverse engineering* funcionou). Testar a aplicação.

Relativamente à operação de remoção de tarefas da BD, a equipa de desenvolvimento constata que afinal dava jeito ter a chave primária de cada registo nos objectos, no entanto essa informação não existe nos objectos Tarefa em memória mas apenas na

BD. Vamos pois por isso proceder a algumas alterações na classe Tarefa criando mais uma propriedade para o ID na BD.

```
private int mID;
public int ID
{
	get
	{
		return mID;
	}
}
```

Em seguida é necessário criar um novo construtor da classe `Tarefa` usado apenas para criar objectos quando são lidos da BD.

```
public Tarefa(int id, int p, string d, DateTime dt, DateTime dtCricao)
{
	mID = id;
	mPrioridade = p;
	mDescricao = d;
	mData = dt;
	mDataCriacao = dtCricao;

	mNova = false;
	mApagada = false;
}
```

Alterar o método de mapeamento na classe `ServicoDados` para usar o novo construtor

```
private Tarefa MapearBD2Obj(OleDbDataReader rd)
{
	int id = (int)rd["ID"];
	DateTime dtCriacao = (DateTime)rd["DataCriacao"];
	int p = (int)rd["Prioridade"];
	string d = (string)rd["designacao"];
	DateTime dt = (DateTime)rd["data"];

	return new Tarefa(id, p, d, dt, dtCriacao);
}
```

Compilar e experimentar a aplicação.

Para implementar a operação de remoção da BD é necessário acrescentar o seguinte código à classe `ServicoDados`.

```
private const string SQL_DELETETAREFA = "DELETE FROM Tarefas WHERE id =
@TarefaID";

public void Apagar(Tarefa t)
{
	try
	{
		OleDbConnection cnx = new OleDbConnection(CONN);
		OleDbCommand cmd = new OleDbCommand(SQL_DELETETAREFA, cnx);
		cmd.Parameters.Add("@TarefaID", t.ID);

		cnx.Open();
		cmd.ExecuteNonQuery();
```

```
                cnx.Close();
            }
            catch (OleDbException ex)
            {
                throw new ApplicationException("Erro no serviço de dados ao
inserir registo", ex);
            }
        }
```

Compilar e experimentar a aplicação.
Sincronizar código e modelo. Verificar no diagrama de classes a inclusão das novas propriedades e operações.

3.12 Passo 10 – modelação de componentes da aplicação

Vamos agora criar um diagrama de componentes no `Model Explorer` e adicionar um componente chamado "WinAgenda".

Figura 30 - diagrama de compoentnes da aplicação exemplo Tarefas

Definir o estereotipo `Executable` para o componente criado na janela de propriedades.

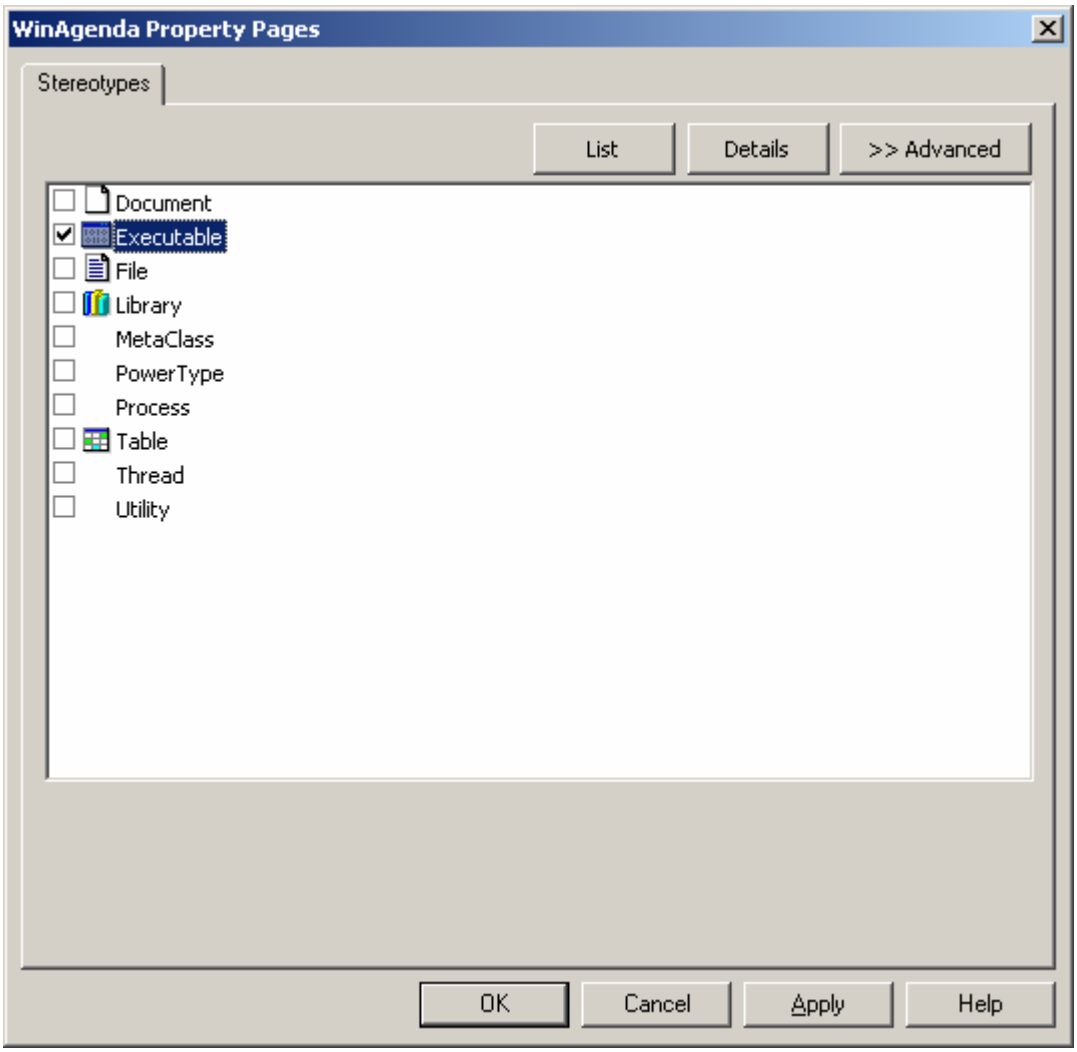

Figura 31

Reparar que a representação gráfica do componente altera-se.

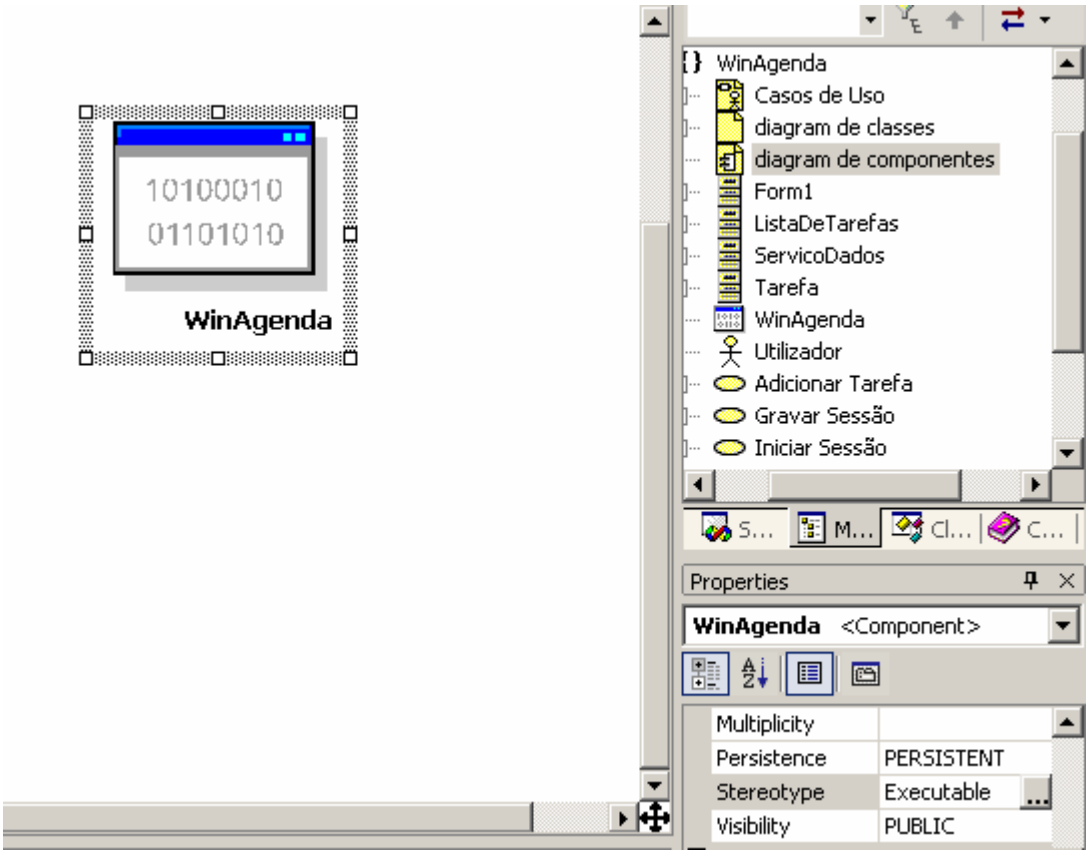

Figura 32

Por fim vamos indicar quais as classes residentes neste componente escolhendo na janela de propriedades a propriedade `Collections`.

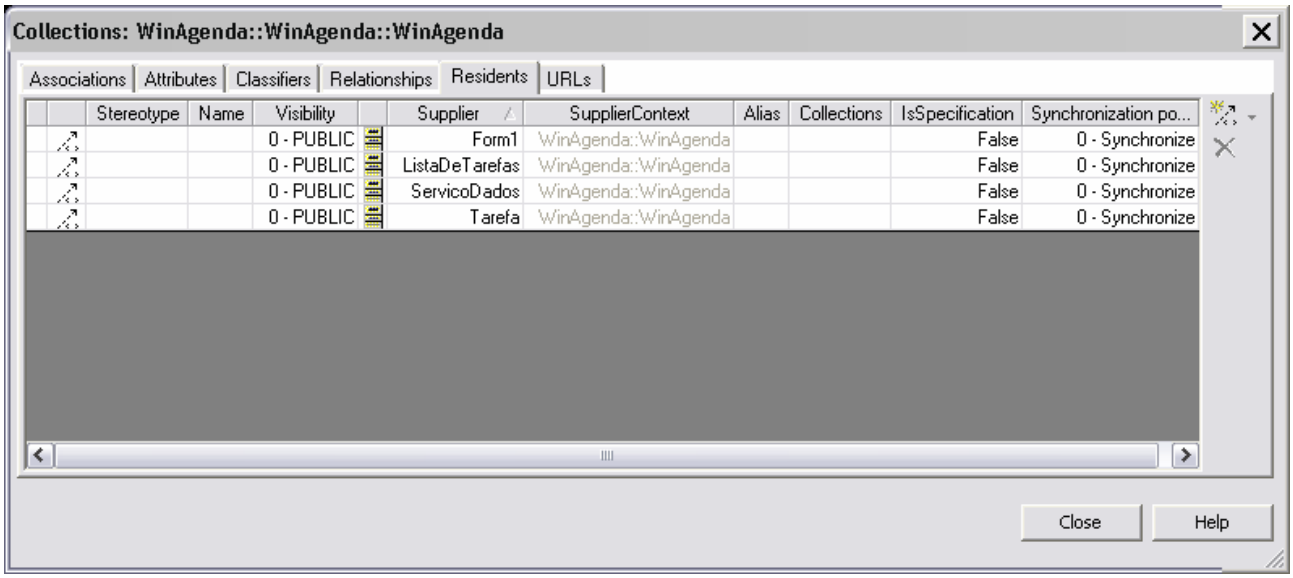

Figura 33

Um outro componente da aplicação é a base de dados. Adicionar ao diagrama um novo componente e especificar o estereotipo `Database`. Em seguida adicionar uma relação de dependência entre os dois componentes como se vê na figura seguinte.

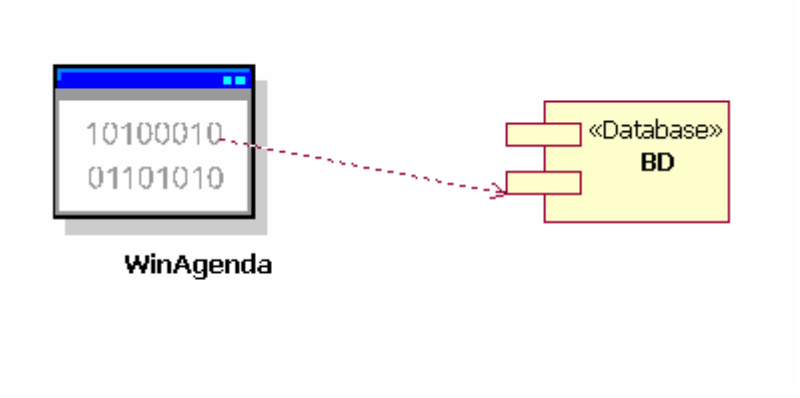

Figura 34 – diagrama de componentes da aplicação exemplo tarefas

3.13 Passo 11 – modelação da instalação da aplicação

Para finalizar a modelação da aplicação vamos então definir cenários de instalação. Neste caso existirão dois cenários possíveis:

1. a aplicação e a base de dados residem na mesma máquina; ou
2. a aplicação e a base de dados residem em máquinas diferentes

No `Model Explorer` criar um diagrama de *deployment* e chamar-lhe "Deployment single host". Adicionar um nó do tipo "tower".

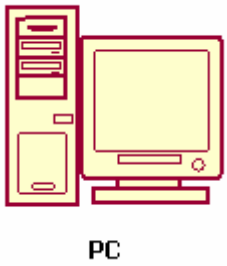

Figura 35

Na propriedade `collections` desse nó identificar quais os componentes instalados neste nó.

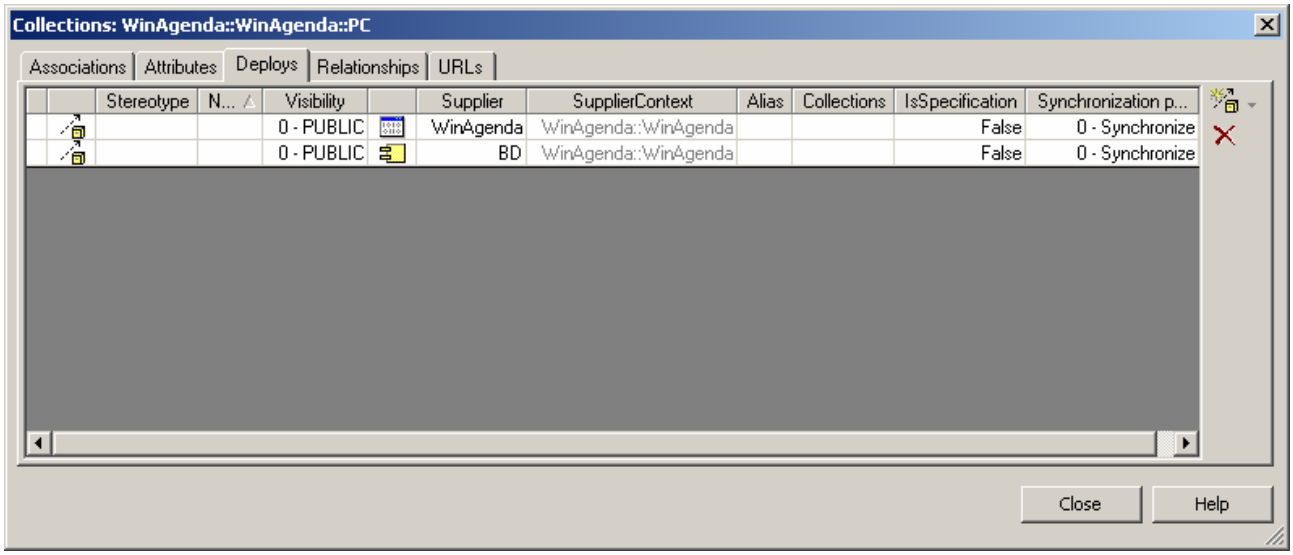

Figura 36

Para o segundo cenário vamos criar um novo diagrama de *deployment* chamado "deployment client-server" ao qual se vai adicionar vários nós e uma infra-estrutura de rede.

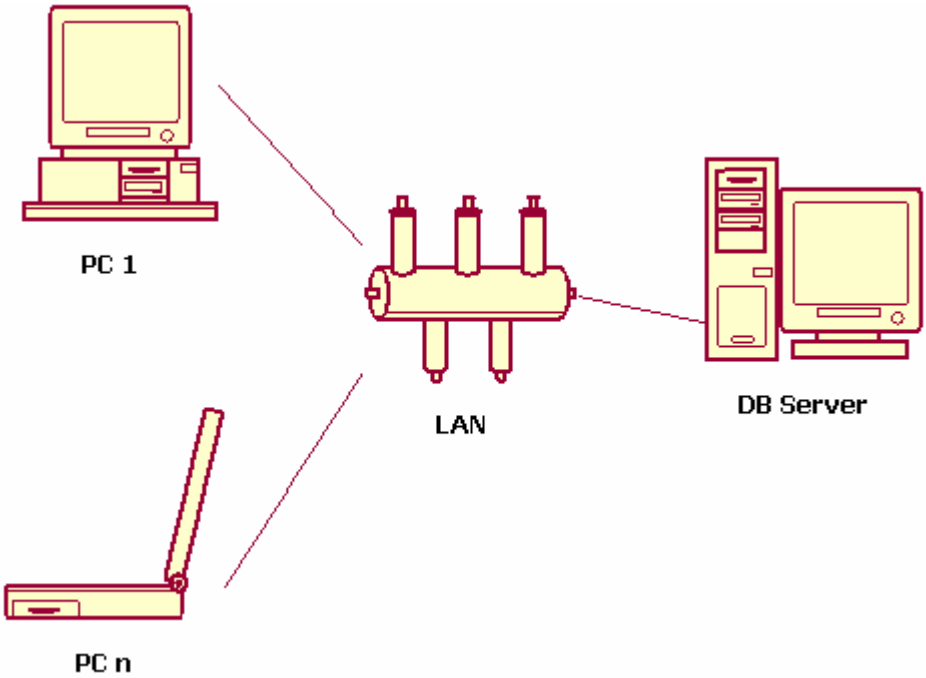

Figura 37 – diagrama de deployment para cenário cliente-servidor

Em seguida indica-se em cada nó quais os componentes instalados.

3.14 Melhorias à aplicação

A aplicação exemplo pode ser melhorada incluindo por exemplo as seguintes funcionalidades:

- Melhorar a visualização no DataGrid
  - Permitir ordenação das colunas

    - Visualizar apenas os campos que fazem sentido
- Edição directa do DataGrid no formulário
- Usar um enumerado (ex., Alta, Normal, Baixa) para a prioridade das tarefas em vez de um campo inteiro
- Permitir alteração de tarefas
    - Marcação das tarefas já realizadas
    - Acrescentar um campo para a data real de execução/conclusão da tarefa
- Indicação visual (por exemplo noutra cor) das tarefas em atraso tendo por referência a data de hoje e a data suposta de execução da tarefa

4 Melhor organização dos *layers*

4.1 Introdução

Na secção anterior vimos como criar uma aplicação a partir do modelo e usando *round-trip engineering* ir efectuado a sincronização entre código e modelo. Embora a aplicação anterior fosse logicamente dividida em três camadas essas camadas estavam num único componente e num mesmo "espaço". Vamos agora ver uma alternativa de organização mais lógica para as camadas.

4.2 Utilização de packages

Consiste na criação de *packages* separados (na mesma) aplicação para colocar as classes de cada camada.

4.2.1 Passo 1 – criar packages

Com o botão do lado direito no *model explorer* escolher `Add UML` → `Package` (Figura 38). Criar três *packages* denominados "Presentation", "BusinessRules" e "DataAcess" conforme se pode ver na Figura 39.

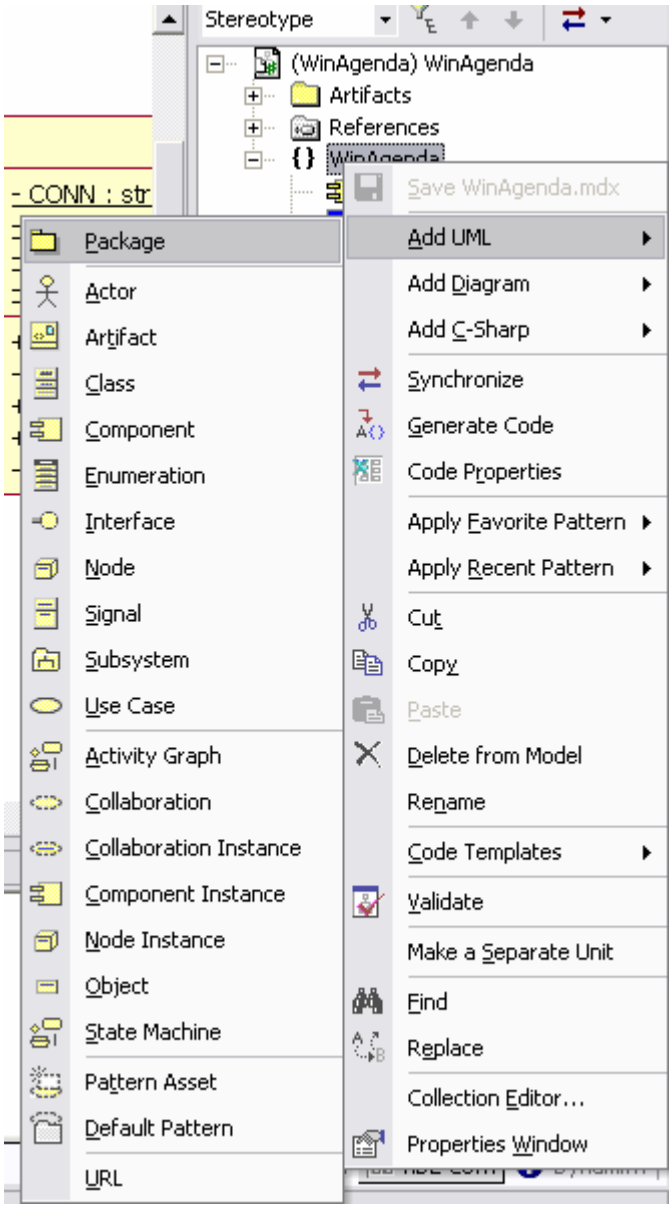

Figura 38- criar package

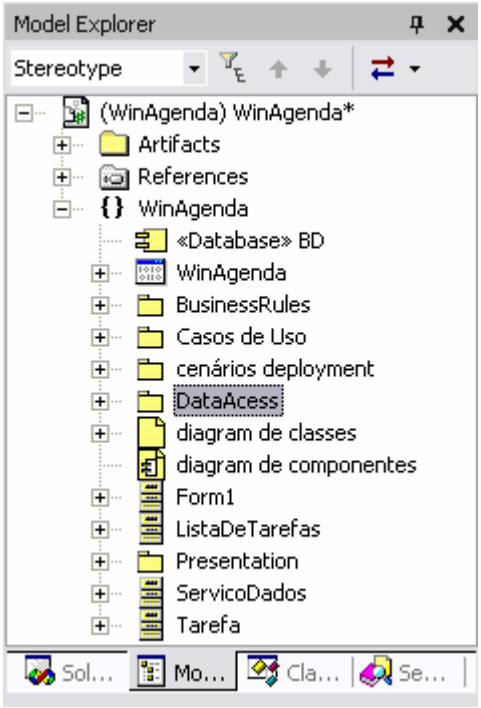

Figura 39 - packages criados

4.2.2 Passo 2 – colocar classes nos packages

Em seguida colocar as classes correspondentes a cada camada dentro do *package* correspondente (Figura 40).

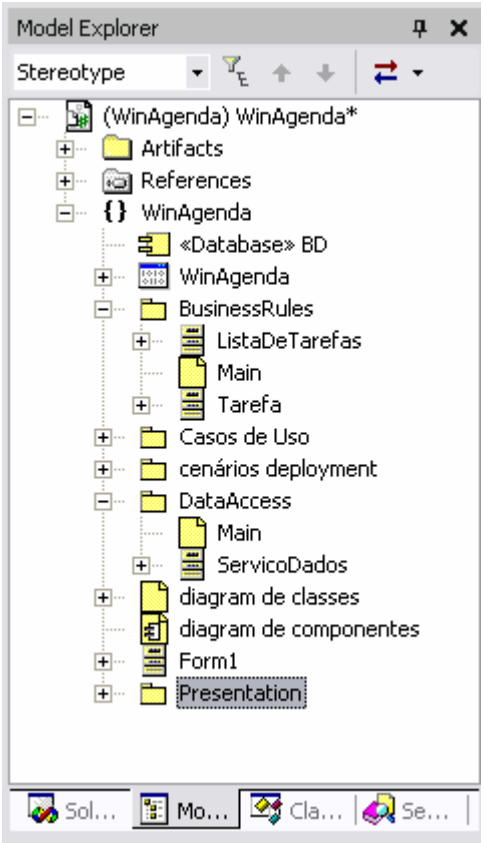

Figura 40 - classes nos packages

De notar que a classe `Form1` não foi colocada em nenhum *package*. Essa alteração será feita no código posteriormente para evitar alguns problemas do XDE ao alterar classes geradas pelos wizards do visual studio.

4.2.3 Passo 3 – converter packages em namespaces .net

Para cada um dos packages criados, na janela de propriedades modificar o estereotipo para "namespace" (Figura 41).

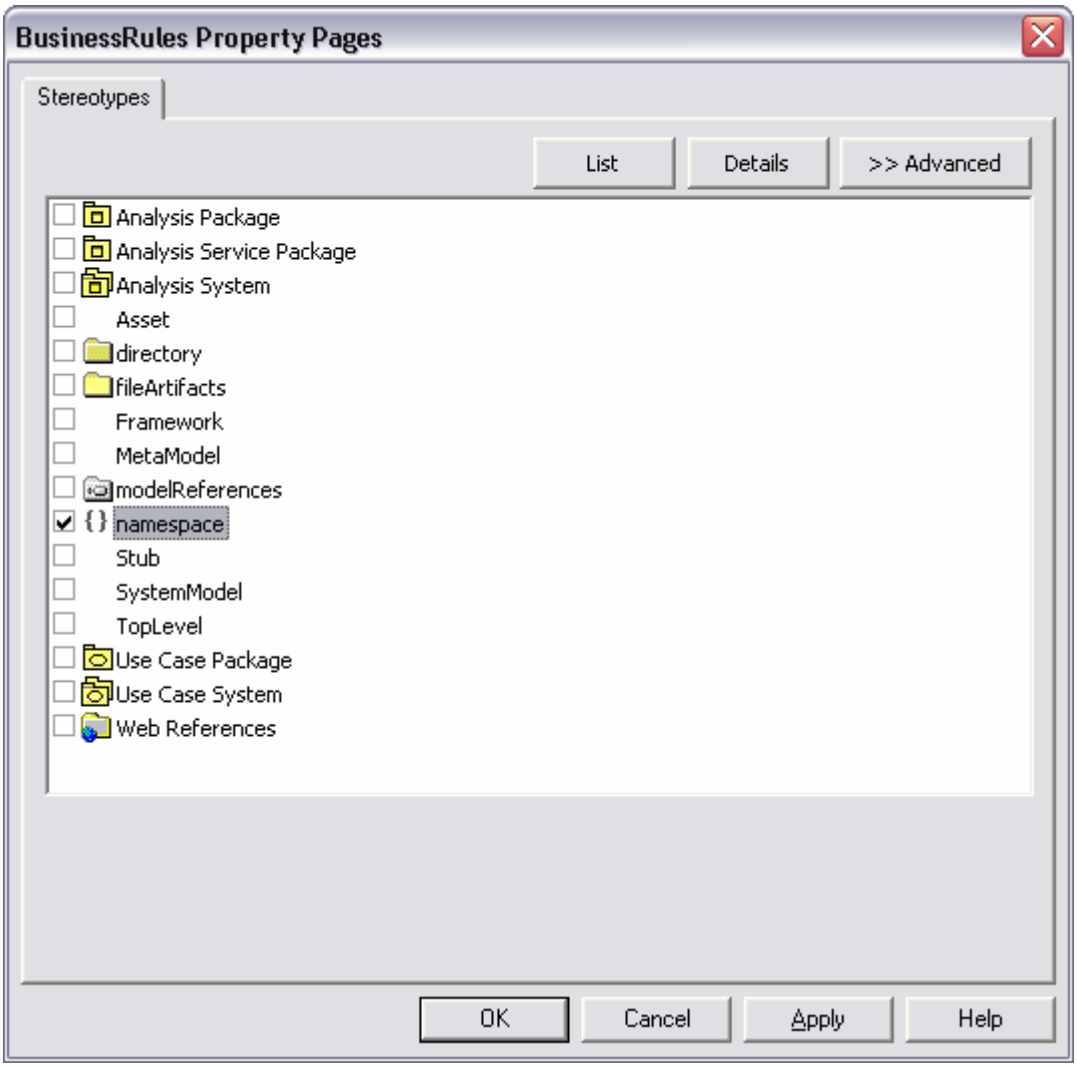

Figura 41 - estereótipos para packages

De notar que a representação gráfica do *package* modifica-se (Figura 42)

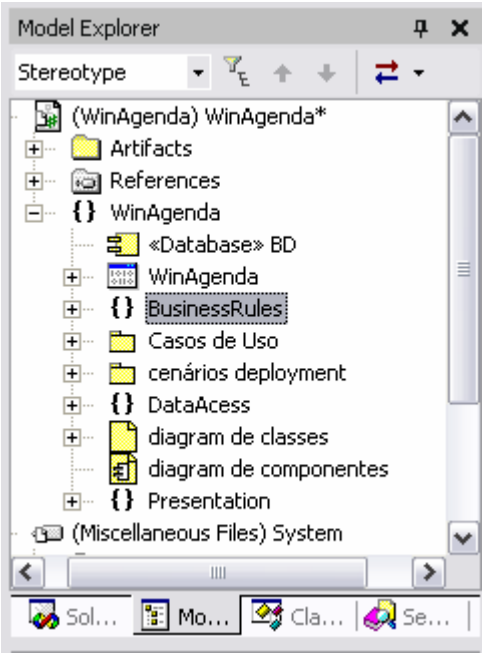

Figura 42 – namespaces

4.2.4 Passo 4 – criar diagrama de packages

No *model explorer* adicionar um diagrama do tipo "Freee Form" e dar-lhe o nome "diagrama de packages". Arrastar os *packages*/*namespaces* criados nesta secção para o diagrama e colocar as dependências entre eles (Figura 43).

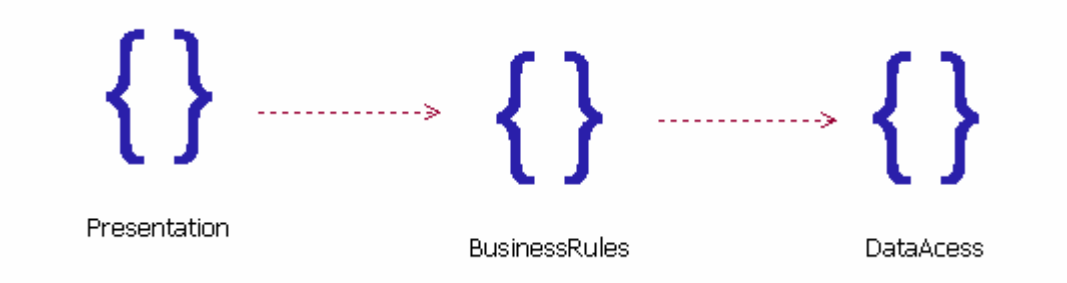

Figura 43 - diagrama de packages

4.2.5 Passo 5 – sincronizar código

Sincronizar o código e o modelo e verificar que as classes estão agora dentro do *namespace* correspondente (Figura 44).

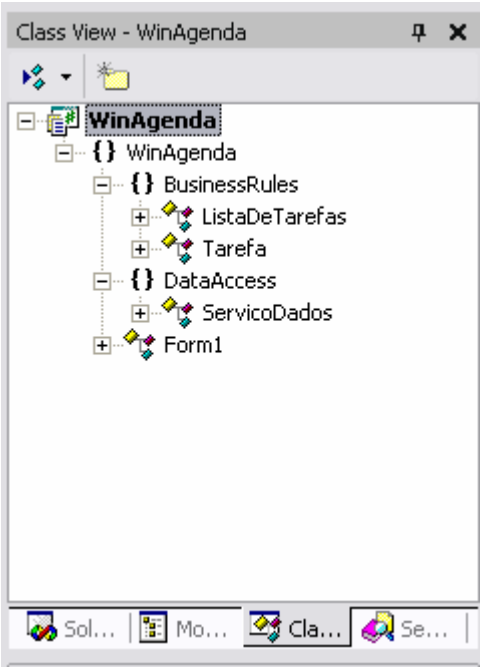

Figura 44 - class view

No ficheiro "Form1.cs" alterar a declaração de *namespace* para incluir a classe `Form1` no *namespace* "Presentation".

```
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

namespace WinAgenda.Presentation
{
	/// <summary>
	/// Summary description for Form1.
	/// </summary>
	public class Form1 : System.Windows.Forms.Form
	{
		private System.Windows.Forms.DataGrid dataGrid1;
		private System.Windows.Forms.GroupBox groupBox1;
```

Ao compilar dará erros de referências pois não encontra os tipos declarados nas classes devido a estes estarem em *namespaces* separados. Para corrigir, acrescente as seguintes linhas de código no início dos ficheiros indicados.

Tabela 3 - declarações "using"

Ficheiro	Linhas a acrescentar
Form1.cs	`using WinAgenda.BusinessRules;`
ListaDeTarefas.cs	`using WinAgenda.DataAcess;`
ServicoDados.cs	`using WinAgenda.BusinessRules;`

Ao compilar já não deve dar erros.

4.2.6 Passo 6 – corrigir modelo

Sincronizar o modelo e o código.

Reparar que existe uma dependência entre a camada de acesso a dados e a camada de lógica de negócio (última linha da Tabela 3) o que não é desejado (já que as camadas devem ter apenas dependências num único sentido).
Se analisarem o código, essa dependência tem a ver com a criação de objectos `Tarefa` lidos da BD. Há várias maneiras de fazer desaparecer esta dependência indesejada; uma delas apoia-se na constatação que a classe `Tarefa` é na realidade uma entidade de negócio que necessita ser partilhada por todas as camadas. Nesse caso, cria-se um novo *package*/*namespace* denominado "BusinessEntities" e coloca-se a classe Tarefa nesse *package* (Figura 45).

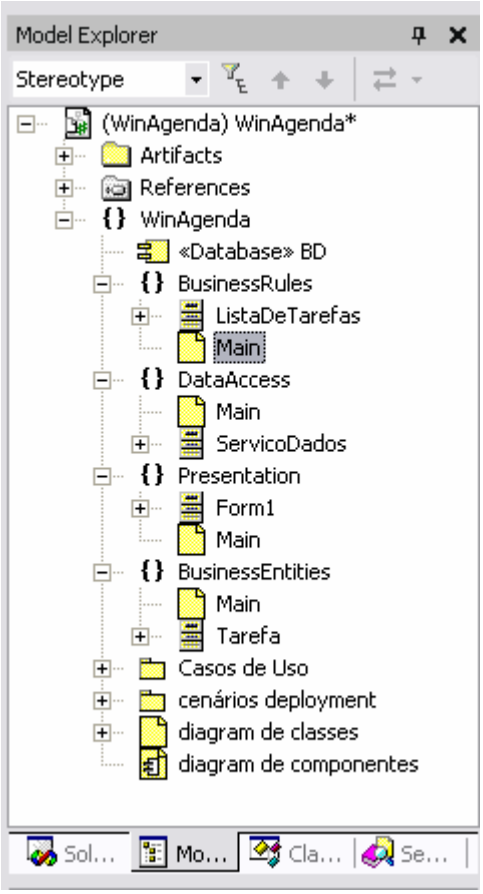

Figura 45 - nova estrutura de camadas da aplicação

Após actualizar o diagrama de *packages* (Figura 46) é possível sincronizar novamente com o código e actualizar as dependências entre classes colocando as declarações `using` nos ficheiros correspondentes (Tabela 4).

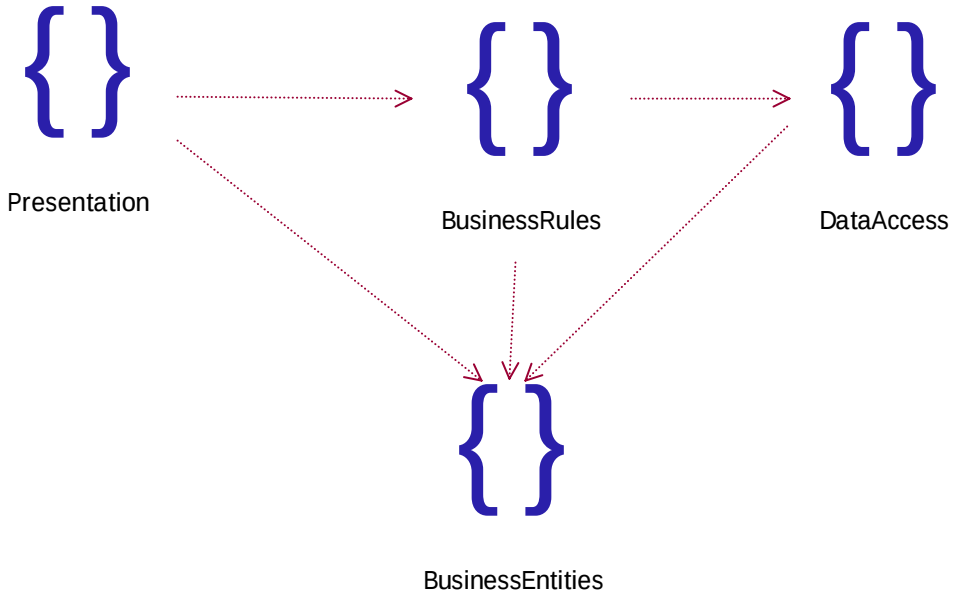

Figura 46 - novo diagrama de packages

Tabela 4 – novas declarações "using"

Ficheiro	Linhas a acrescentar
Form1.cs	`using WinAgenda.BusinessRules;`
ListaDeTarefas.cs	`using WinAgenda.BusinessEntities;` `using WinAgenda.DataAcess;`
ServicoDados.cs	`using WinAgenda.BusinessEntities;`

5 Conclusão

Neste guião foi brevemente apresentado o processo de desenvolvimento utilizando uma ferramenta CASE que permite *roundtrip engineering*. Exemplificou-se como criar um modelo a partir do qual se gera código, bem como actualizar o modelo com base nas alterações efectuadas no código.
Efectuou-se uma análise inicial do sistema em causa (*use cases* e diagramas de sequência), a partir da qual se modelou a estrutura estática da aplicação (diagrama de classes) sobre a qual se partiu para a geração de código.
Após o código funcional de acordo com os requisitos especificados nos use case, modelou-se os cenários de instalação da aplicação.