Princípios da Programação Orientada ao Componente

- Separação entre a interface e a implementação
- Compatibilidade binária
- Independência da linguagem
- Transparência da localização
- Gestão da concorrência
- Controlo de versões
- Segurança Baseada em Componentes

Common Language Infrastructure(CLI)

- **Common Type System (CTS)**
  - O conjunto dos tipos de dados e operações que são partilhados por todas as linguagens de programação que podem ser executadas por uma implementação de CLI, tal como a CLR.
- **Metadata**
  - Informação sobre os tipos presentes na representação intermédia
- **Common Language Specification (CLS)**
  - Um conjunto de regras básicas que as linguagens compatíveis com a CLI devem respeitar para serem interoperáveis entre si. As regras CLS definem um subconjunto do Common Type System.
- **Virtual Execution System (VES)**
  - O VES carrega e executa programas CLI-compatíveis, utilizando a metadata para combinar separadamente as partes em tempo de execução.

Características da plataforma .Net

- Portabilidade
  - Representação intermédia
  - Ambiente de execução virtual
  - Biblioteca de classes
- Interoperabilidade entre linguagens
  - Sistema de tipos comum
    - Capacidade de utilização de componentes de software realizados em linguagens diferentes
    - Linguagem intermédia com sistema de tipos independente da linguagem
- Serviços de execução
  - Gestão de memória
  - Segurança de tipos e controlo de acessos
  - Serialização (“serialization”) - conversão de grafos de objectos em sequências de bytes
- Funcionalidades
  - Biblioteca rica para desenvolvimento de diferentes tipos de componentes/aplicações
- Estes serviços estão disponíveis porque existe informação de tipos (METADATA) em tempo de execução

Metadata de um Managed Module

- Conjunto de tabelas com:
  - os tipos definidos no módulo - DefTables;
  - os tipos referênciados (importados) – RefTables;
- Informação sobre tipos
  - Sempre incluída no módulo pelo compilador
  - Descreve tudo o que existe no módulo

Vantagens da Metadata

- Dispondo da metadata não são necessários os ficheiros header e os ficheiros biblioteca para compilar e ligar o código das aplicações. Os compiladores e os linkers poderão obter a mesma informação consultando a metadata dos managed modules.
- O processo de verificação do código do CLR usa a metadata para garantir que o código realiza apenas operações seguras.
- A metadata suporta a serialização/desserialização automática do estado dos objectos
- Para qualquer objecto, o GC pode determinar o tipo do objecto e, a partir da metadata, saber quais os campos desse objecto que são referências para outros objectos.

Assembly

- Por omissão, os compiladores transformam o managed module num assembly.
  - manifest do assembly gerado contém a indicação de que o assembly consiste apenas de um ficheiro
  - Cada assembly é ou uma aplicação executável (exe) ou uma biblioteca (DLL).
- Um dos módulos constituintes do assembly contém obrigatoriamente um manifesto que define os módulos constituintes como um todo inseparável e contém o identificador universal do assembly.

Razões para a criação de Assemblies Multi-módulo

- O módulo só é carregado para memória quando é necessário (quando for usado algum tipo exportado no módulo)
- Torna possível a implementação de um assembly em mais que uma linguagem
- Separar fisicamente resources (imagens, strings) do código

A linguagem intermédia (IL)

- A linguagem IL é stack based (execução de uma máquina de stack)
- Não existem instruções para manipulação de registos
- Todas as instruções são polimórficas (dependendo do tipo dos operandos podem ter sucesso ou não, gerando, em caso de insucesso, uma excepção ou falhando a fase de verificação, se existir).

Loader

- O Loader é responsável por armazenar e inicializar assemblies, recursos e tipos;
- Utiliza um política de armazenar tipos (e assemblies e módulos) on demand
- O Loader faz cache da informação dos tipos definidos e referênciados no assembly, injectando um pequeno stub em cada método armazenado.
- O stub é utilizado para
  - Denotar o estado da compilação JIT.
  - Transitar entre código managed e unmanaged.
- O loader irá armazenar os tipos referênciados, caso os mesmos ainda não tenham sido armazenados.
- O loader utiliza a metadata para inicializar as variáveis estáticas e instânciar os objectos.

Verifier

- Aquando da compilação JIT o CLR executa um processo designado por verificação
- O verificador é responsável por verificar que
  - A metadada está bem definida, isto é válida.
  - O código IL é type safe, isto é as assinaturas dos tipos estão a ser utilizadas correctamente verificando deste modo a segurança das instruções.

Geração de código “just in time”

- As invocações de métodos são realizadas indirectamente através de “stubs”
  - O “stub” de cada método é apontado pela tabela de métodos
- Inicialmente o “stub” aponta para o JIT
- Na primeira chamada do método é invocado o “JIT compiler”
  - Usa o código IL do método e a informação de tipo para gerar o código nativo
  - Altera o “stub” para apontar para o código nativo gerado
  - Salta para o código nativo
- As restantes chamadas usam o código nativo

As restantes chamadas usam o código nativo

- Desvantagens
  - Peso computacional adicional para a geração do código nativo
  - Memória necessária para a descrição intermédia e código nativo
- Vantagens
  - A geração de código tem informação sobre a plataforma nativa de execução
    - Optimização para o processador nativo
    - Utilização de informação de “profiling” (características de execução do código)

Sistema de tipos

- O Common Type System especifica a definição, comportamento, declaração, uso e gestão de tipos.
- Suporta o paradigma da Programação Orientada por Objectos.
- Desenhado por forma a acomodar a semântica expressável na maioria das linguagens modernas.
- Define:
  - Hierarquia de tipos
  - Conjunto de tipos “built-in”
  - Construção de tipos e definição dos seus membros
  - Utilização e comportamento dos tipos

Common Language Specification

- Conjunto de restrições ao CTS para garantir a interoperabilidade entre linguagens
  - Define um sub-conjunto do CTS
  - Contém as regras que os tipos devem respeitar por forma a serem utilizados por qualquer linguagem “CLS-compliant”

Common Type System (CTS)

- Sistema de tipos (orientado aos objectos) que permite representar tipos de várias linguagens.
- Define:
  - categorias de tipos
  - hierarquia de classes
  - conjunto de tipos “built-in”
  - construção e definição de novos tipos e respectivos membros genéricos

Tipos Valor (Value Types)

- Os tipos valor contêm directamente os seus dados, e as suas instâncias estão ou alocadas na stack ou alocadas inline numa estrutura.
- São passados por valor
- Podem ser definidos novos tipos valor, definindo uma nova classe como derivada da classe System.ValueType
- Os tipos valor são sealed, isto é, não podem ser tipos bases para outro tipo valor ou tipo referência

Tipos Refêrencia (Reference Types)

- Os tipos refêrencia representam referências para objectos armazenados no heap.
- São passados por referência
- Os tipos referência incluem classes, interfaces, arrays e delegates

System.Object

- Todos os tipos derivam de System.Object.

Unificação do sistema de tipos (resumo)

- As variáveis
  - de Tipos Valor contêm os dados (valores)
  - de Tipos Referência contêm a localização dos dados (valores)
- As instâncias de “Self-describing Types” (designadas objectos)
  - são sempre criadas dinamicamente (em heap)
    - explicitamente (com o operador new)
    - implicitamente (operação box)
  - a memória que ocupam é reciclada automaticamente (GC)
  - é sempre possível determinar o seu tipo exacto (memory safety)
- A cada Tipo Valor corresponde um “Boxed Value Type”
  - Suporte para conversão entre Tipos Valor e Tipos Referência (box e unbox)

Sumário de padrões de override de Equals

- Override de Equals implica override de GetHashCode de forma a manter o invariante
- A implementação deve verificar se o tipo do objecto passado como parâmetro é igual ao tipo do this
- Value Types
  - Override de Equals em ValueType deve sempre ser feito para evitar a penalização da implementação de ValueType
  - Deverá haver a sobrecarga em value type de nome VT: bool Equals(VT v);
- Reference Types
  - Verificar situações de referências nulas
  - Invocar Equals da classe base caso esta também tenha Equals redefinido

Namespaces e Assemblies

- O CLR não tem a noção de namespac:es. Quando acede a um tipo, o CLR necessita de conhecer o nome completo do tipo e qual o assembly que contém a respectiva definição.
- Não existe nenhuma relação entre assemblies e namespaces.

Invocação de métodos em IL – call

- é utilizada para invocar métodos estáticos, de instância e virtuais

Invocação de métodos em IL – callvirt

- A instrução callvirt pode ser utilizada para invocar métodos de instância ou virtuais

Passagem de parâmetros: por referência

- Em IL indicado por & e em C# por ref
- ref versus out
  - The **out** keyword causes arguments to be passed by reference.
  - This is similar to the ref keyword, except that ref requires that the variable be initialized before being passed. To use an **out** parameter, both the method definition and the calling method must explicitly use the **out** keyword.

Sobrecarga de métodos

- Número de parâmetros;
- Tipo dos parâmetros;
- Tipo de retorno; (não em C#)
- Passagem de parâmetro por valor ou referência.

Enumerados

- Não podem definir métodos, propriedades ou eventos.

Interfaces

- suportam herança múltipla de outras interfaces
- Não podem conter campos de instância nem métodos de instância com implementação.
- a implementação de uma interface resulta, por omissão, em métodos sealed, excepto se for declarado como virtual
- An interface contains only the signatures of methods, properties, events or indexers. A class or struct that implements the interface must implement the members of the interface that are specified in the interface definition.

Genéricos

- Permite criar colecções homogéneas, validadas em tempo de compilação(Type Safety)
- Aumento da legibilidade - não necessita de cast´s explícitos
- Aumento de performance - Instâncias de tipo valor não precisam de ser boxed para serem guardadas em colecções genéricas

Compilação de Genéricos

- O código genérico é compilado para IL, que fica com informação genérica de tipos
- Quando um método que usa parâmetros do tipo genérico é compilado pelo JIT, o CLR
  - substitui o tipo genérico dos argumentos de cada método por cada tipo especificado, criando código nativo específico para operar para cada tipo de dados especificado
  - CLR fica com código nativo gerado para cada combinação método/tipo

Constraints (Restrições)

- Por omissão, os tipos parâmetro só podem ser usados através da interface de object, já que é a única que é garantidamente implementada
- Podem ser aplicadas restrições (constraints) aos tipos-parâmetro:
  - classe base
  - interfaces implementadas
  - existência de construtor sem parâmetros (new())
  - tipo-referência (class) ou tipo-valor (struct)

Co-variância e Contra-invariância nos Genéricos

- Um parâmetro do tipo genérico por ser
  - Invariante : Significa que não pode ser mudado
  - Contra-variante:
    - Pode mudar de uma classe para uma classe derivada
    - Isso é indicado em C# pela palavra in
    - Apenas pode aparecer como input, tal como os argumentos de métodos.
  - Co-variante
    - Significa que pode mudar de uma classe para uma das suas classes base
    - Isso é indicado em C# pela palavra out
    - Apenas pode aparecer como output, tal como o tipo de retorno dos métodos

```
public static bool IsMethodToTest(MethodInfo mi){
Type t = Type.GetType("MethodTestAttribute");
object[] atrib = mi.GetCustomAttributes(t, false);
if (atrib.Length != 1) return false;
return (mi.GetParameters().Length == 0 &&
(! mi.ReturnType.Equals(Type.GetType("System.Void")))
&&(! mi.ReturnType.IsArray) &&(!
mi.ReturnType.BaseType.Equals(Type.GetType("System.Mult
icastDelegate"))));
}
class Stock{
private Double price;
public event EventHandler<StockPriceChangeEventArgs>
priceChanged;
public Stock(Double p){price = p;}
public Double Price{
   get{return price;}
   set{ StockPriceChangeEventArgs args = new
StockPriceChangeEventArgs(price, value);
   price=value; OnPriceChanged(args);
   }}
virtual protected void OnPriceChanged
(StockPriceChangeEventArgs args)
{
if (priceChanged != null){priceChanged(this,args);}}}
class StockPriceChangeEventArgs : System.EventArgs{
private double _oldPrice;private double _newPrice;
public StockPriceChangeEventArgs
(double old_price, double new_price)
{_oldPrice = old_price; _newPrice = new_price;}
public Double OldPrice { get { return _oldPrice; } }
public Double NewPrice { get { return _newPrice; } }
}
class StockPriceObserver {
public StockPriceObserver(Stock s){
s.priceChanged += this.PriceChanged;}
public void PriceChanged(Object sender,
StockPriceChangeEventArgs tmp){Console.WriteLine();}
}
class BrokerStockPriceObserver{
public BrokerStockPriceObserver(Stock s){
s.priceChanged += this.PriceChanged;}
public void PriceChanged(Object sender,
StockPriceChangeEventArgs tmp){
if(tmp.NewPrice - tmp.OldPrice > 10)Console.WriteLine;
}}
public static bool IsDelegate(this MemberInfo mi)
{return mi.DeclaringType.Equals(typeof(Delegate));}
public static bool IsSubclassOf(Type t1, Type t2)
{return t1.IsSubclassOf(t2);}
public static bool IsAssignableFrom(Type tvar, Type
tval){return tvar.IsAssignableFrom(tval);}
private static void DumpGenericObject(int level, object o)
{
 Type t = o.GetType(); BindingFlags bf =
BindingFlags.NonPublic|BindingFlags.Public |
BindingFlags.Instance;
while (t != typeof(System.Object)) {
FieldInfo[] fields = t.GetFields(bf);
foreach (FieldInfo f in fields) {
    object v = f.GetValue(o);
    indent(level); Console.Write(f.Name + ": ");
    if (v==null) {Console.WriteLine("null");}
   else if (v.GetType().IsPrimitive || v.GetType() ==
typeof(System.String))
        {Console.WriteLine(v.ToString());}
    else{
          Console.WriteLine();
          DumpObject(level + 1, v);
        }}
  t= t.BaseType;
 }}
  private static void showArray(int level, Array a)
  {
    bool first = true;
    indent(level); Console.WriteLine('[');
    foreach (object o in a)
    {if (first) first = false; else { indent(level+1);
Console.WriteLine(", "); }
     DumpObject(level + 1, o);
    } indent(level); Console.WriteLine(']'); }

  private static void showDelegate(int level, Delegate d)
  { indent(level); Console.WriteLine("[[ {0} Delegate:
{1} listeners ]]",
d.GetType().Name,d.GetInvocationList().Length);
  }
  public static void DumpObject(int level, object o) {
    Type t = o.GetType();
    Array a = o as Array;
    if (a != null) showArray(level,a);
    else {
      Delegate d = o as Delegate;
      if (d != null) showDelegate(level, d);
      else DumpGenericObject(level, o);
    }}
public static IEnumerable<TResult> myZip<TFirst,
TSecond, TResult>(this IEnumerable<TFirst> first,
IEnumerable<TSecond> second, Func<TFirst, TSecond,
TResult> resultSelector){
IEnumerator<TFirst> f = first.GetEnumerator();
IEnumerator<TSecond> s = second.GetEnumerator();
while (f.MoveNext() && s.MoveNext())
{yield return resultSelector(f.Current, s.Current);}
}
public static IEnumerable<int> InfiniteIntegers(int
start){while (true)yield return start++;}
public static IEnumerable<string> GetFileLines(string
fileName){
var l = System.IO.File.ReadAllLines(fileName);
return l.myZip(InfiniteIntegers(0), ((o2, o1) => o1 +
o2)); }
```

```
[AttributeUsage(AttributeTargets.Class, Inherited = false)]
[Serializable]
public sealed class AttributeUsageAttribute : Attribute {
  internal AttributeTargets m_attributeTarget = AttributeTargets.All;
  internal Boolean m_allowMultiple = false;
  internal Boolean m_inherited = true;
  public AttributeUsageAttribute(AttributeTargets validOn) {
    m_attributeTarget = validOn;
  }
  public AttributeTargets ValidOn {
    get { return m_attributeTarget; }
  }
  public Boolean AllowMultiple {
    get { return m_allowMultiple; }
    set { m_allowMultiple = value; }
  }
  public Boolean Inherited {
    get { return m_inherited; }
    set { m_inherited = value; }
  }
}
class MethodInfo : MethodBase {
        public  Type[] GetGenericArguments();
        public  ParameterInfo[] GetParameters();
        public  MethodImplAttributes GetMethodImplementationFlags();
        public  Object Invoke(Object obj, Object[] parameters);
         Type ReturnType{ get; }
        ParameterInfo ReturnParameter{ get; }

}
public static void SetX(Object target, int value) {
  // Write to field: int x
  Type type = target.GetType();
  FieldInfo field = type.GetField("x");
  field.SetValue(target, value);
}
double CallDoIt(Object target, double x, double y)  {
  // Call method: double Add( double, double )
  Type type = target.GetType();
  MethodInfo method = type.GetMethod("DoIt");
  if( method != null ) {
    object[] args = { x, y };
    object result = method.Invoke(target, args);
    return (double)result;
  }
  return(0);
}
```

Metadata de um Assembly

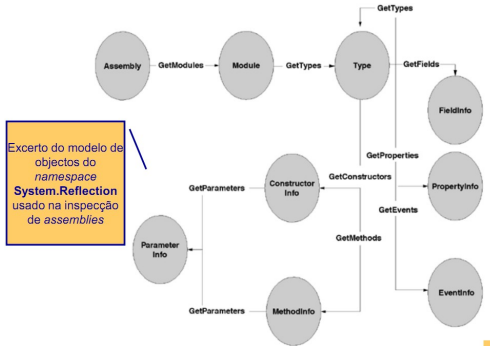
- Um dos módulos constituintes do Assembly, contém também a *manifest metadata table*.

Informação	Descrição
Nome	String com o nome "amigável" (*friendly name*) do *assembly*. Corresponde ao nome do ficheiro (sem extensão) que inclui o manifesto.
Número versão	*Major, minor, revision* e *build numbers* da versão
Cultura	Localização do *Assembly* (língua, cultura). Usada somente em *assemblies* com resources (*strings*, imagens). Os *assemblies* com a componente *cultura* denominam-se assemblies satélite
Nome criptográfico (*strong* name)	Identifica o fornecedor do componente. É uma chave criptográfica. Garante que não existem 2 assemblies distintos com o mesmo nome e que o *assembly* não foi corrompido
Lista de módulos	Pares (nome, *hash*) de cada módulo pertencente ao assembly
Tipos exportados (Também existe em cada um dos módulos)	Informação usada pelo *runtime* para associar um tipo exportado ao módulo com a sua descrição/implementação
Assemblies referenciados (Também existe em cada um dos módulos)	Lista dos *assemblies* de que o *assembly* depende

Atributos pré-definidos aplicáveis a métodos (2)

CLR Term	C# Term	Visual Basic Term	Descrição
NewSlot	new (default)	Shadows	O método não deve redefinir um método virtual definido pelo seu tipo base; o método esconde o método herdado. NewSlot aplica-se apenas a métodos virtuais.
Override	override	Overrides	Indica que o método está a redefinir um método virtual definido pelo seu tipo base. Aplica-se apenas a métodos virtuais.
Abstract	abstract	MustOverride	Indica que um tipo derivado tem de implementar um método com uma assinatura que corresponda a este método abstracto. Um tipo com um método abstracto é um tipo abstracto. Aplica-se apenas a métodos virtuais.
Final	sealed	NotOverridable	Um tipo derivado não pode redefinir este método. Aplica-se apenas ao métodos virtuais.

Constraints	Descrição
where T: <className>	**T** tem de derivar de <className>
where T:<interfaceName>	**T** tem de implementar <interfaceName>
where T : class	**T** tem de ser um tipo referência
where T: struct	**T** tem de ser um tipo valor
where T : new()	**T** tem de ter construtor sem parâmetros



Excerto do modelo de objectos do *namespace* **System.Reflection** usado na inspecção de *assemblies*

```
using Reflection;

public static void ListAllMembersInEntryAssembly() {
  Assembly assembly = Assembly.GetEntryAssembly();
  foreach (Module module in assembly.GetModules())
    foreach (Type type in module.GetTypes())
      foreach (MemberInfo member in type.GetMembers(flags))
        Console.WriteLine("{0}.{1}", type, member.Name );
}
```

```
class Type : MemberInfo  {

        Type DeclaringType{ get; }
        Type ReflectedType{ get; }
        Module Module{ get;}
        Assembly Assembly{ get; }
        String FullName{ get; }
        String Namespace{ get; }
        Type BaseType{ get; }
        Boolean IsNested{ get; }
        Boolean IsNotPublic{ get; }
        Boolean IsPublic{ get; }
        Boolean IsLayoutSequential{ get; }
        Boolean IsClass{ get;}
        Boolean IsInterface{ get;}
        Boolean IsValueType{ get;}
        Boolean IsAbstract{ get; }
        Boolean IsSealed{ get; }
        Boolean IsEnum{ get; }
        Boolean IsSerializable{ get; }
        Boolean IsArray{ get; }
        Boolean IsGenericType{ get; }
        Boolean IsPrimitive{ get; }
}
```

```
class Type : MemberInfo  {
 public static Type GetType(String typeName);
  public  Boolean IsSubclassOf(Type c);
 public  Boolean IsAssignableFrom(Type c);
 public  ConstructorInfo GetConstructor(Type[] types);
 public  ConstructorInfo[] GetConstructors();
 public  ConstructorInfo[] GetConstructors(BindingFlags bindingAttr);
 public  MethodInfo GetMethod(String name, BindingFlags bindingAttr);
 public  MethodInfo GetMethod(String name);
 public  MethodInfo[] GetMethods();
 public  MethodInfo[] GetMethods(BindingFlags bindingAttr);
 public  FieldInfo GetField(String name, BindingFlags bindingAttr);
 public  FieldInfo[] GetFields();
 public  FieldInfo[] GetFields(BindingFlags bindingAttr);
 public  Type GetInterface(String name);
 public  Type[] GetInterfaces();
 public  EventInfo GetEvent(String name);
 public  EventInfo GetEvent(String name, BindingFlags bindingAttr);
 public  EventInfo[] GetEvents();
 public  EventInfo[] GetEvents(BindingFlags bindingAttr);
 public  PropertyInfo GetProperty(String name, BindingFlags bindingAttr);
 public  PropertyInfo GetProperty(String name, Type returnType, Type[] types);
 public  PropertyInfo GetProperty(String name, Type[] types);
 public  PropertyInfo GetProperty(String name, Type returnType);
 public  PropertyInfo GetProperty(String name);
 public  PropertyInfo[] GetProperties(BindingFlags bindingAttr);
 public  PropertyInfo[] GetProperties();
 public  Type[] GetNestedTypes();
 public  Type GetNestedType(String name);
 public  Type[] GetGenericArguments();
```

Nome	Descrição
All	Attribute can be applied to any application element.
Assembly	Attribute can be applied to an assembly.
Class	Attribute can be applied to a class.
Constructor	Attribute can be applied to a constructor.
Delegate	Attribute can be applied to a delegate.
Enum	Attribute can be applied to an enumeration.
Event	Attribute can be applied to an event.
Field	Attribute can be applied to a field.
Interface	Attribute can be applied to an interface.
Method	Attribute can be applied to a method.
Module	Attribute can be applied to a module.
Parameter	Attribute can be applied to a parameter.
Property	Attribute can be applied to a property.
ReturnValue	Attribute can be applied to a return value.
Struct	Attribute can be applied to a value type.

```
class PropertyInfo : MemberInfo  {
        public  ParameterInfo[] GetIndexParameters();
        public  Object GetValue(Object obj, Object[] index);
        public  Void SetValue(Object obj, Object value, Object[] index);
        public  MethodInfo[] GetAccessors();
        public  MethodInfo GetGetMethod();
        public  MethodInfo GetSetMethod();
        Type PropertyType{ get; }
        Boolean CanRead{ get; }
        Boolean CanWrite{ get; }
}
```