# Apresentação

- Ambientes Virtuais de Execução
  - Semestre de Inverno, 10/11

# Objectivos e Programa

- Objectivos
  - Esta disciplina tem por objectivo tratar os ambientes virtuais para execução controlada de programas
  - Utiliza a plataforma Microsoft.NET como caso de estudo

- Programa
  - Requisitos e características dos ambientes de execução modernos
  - A *Common Language Infrastructure(CLI)* e a plataforma Microsoft .NET
  - Estudo do sistema de tipos (Common Type System – CTS) especificado pela CLI
  - Serviços de “run-time”
    - Gestão automática de memória
    - Introspecção
    - Serialização
    - Invocação de código nativo (*unmanaged*)
  - Criação, instalação (*deployment)* e configuração de componentes e aplicações
  - AppDomains

# Bibliografia

- Livro de referência
  - Jeffrey Richter, CLR Via C#, 3Ed, Microsoft Press, 2010
- Slides
- Bibliografia adicional
  - D. Box with C. Sells, “Essential .Net: The Common Language Runtime”, Addison-Wesley, 2002
  - ECMA Standard, http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-335.pdf
- Páginas de apoio
  - http://www.deetc.isel.ipl.pt/programacao/ave

# Avaliação

- 3 mini testes (1 por cada série de questões entregues antecipadamente) – Média mínima dos dois melhores superior ou igual a 8 valores.
- Trabalho prático de síntese no final do semestre.
- Teste final, realizado nas datas de exame (classificação mínima de 10 valores)

# Introdução ao CLI

- Ambientes Virtuais de Execução
  - Semestre de Inverno, 10/11

# Sumário

- Requisitos duma plataforma de desenvolvimento e execução “moderna”
  - Suporte à construção de software baseado em componentes
- Arquitectura da plataforma Microsoft.NET
  Informação de tipo – metadata
- Módulos “managed”
  - *Assemblies*
- Carregamento e Execução
  - Geração de código nativo
  - Verificação

# Normalização

- Common Language Infrastructure (CLI) - normas ECMA-335 e ISO/IEC 23271:2003. Organizado em secções que definem:
  - Representação intermédia - *Common Intermediate Language (CIL)*
  - Sistema de tipos - *Common Type System (CTS)*
    - *Common Language Specification* (CLS)
    - Metadata – informação sobre os tipos presente na representação intermédia
  - Conjunto de bibliotecas de classes
    - A maior parte das bibliotecas standard estão organizadas em perfis. Estão definidos dois perfis: O *kernel profile* que contém as funcionalidades necessárias a toda a implementação conforme e o *compact profile*, superconjunto do anterior que acrescenta as bibliotecas que suportam as API´s de *networking*, XML e introspecção.
    - *Framework Class Library* é o conjunto de classes fornecidas pela implementação da Microsoft para *desktops*
  - Formato binário dos módulos executáveis

# Propriedades do framework .NET (Resumo)

| Requisito | Suportado por |
|---|---|
| **Independência às linguagens** | ✓ Geração de código para uma máquina virtual orientada a objectos<br>• Representação intermédia<br>• Sistema de tipos independente das linguagens |
| **Portabilidade** | ✓ Geração de código para uma máquina virtual orientada a objectos<br>✓ Biblioteca de classes e *frameworks* |
| **Auto-Descrição** | ✓ *Metadata* |
| **Serviços de suporte à execução**<br>Carregamento dinâmico de código<br>Gestão *automática* de memória<br>Reflexão (ou Introspecção)<br>Serialização<br>Segurança<br>Excepções<br>Acesso a recursos nativos (*unmanaged*) | ✓ Informação completa de tipo em tempo de execução (metadata)<br>✓ *type safety* e permissões associadas a código (segurança) |
| **Controlo de versões** | ✓ *Deployment* privado<br>✓ *Deployment* partilhado<br>• Repositório de *assemblies* globais<br>• *Strong* names (coexistência, quer em disco quer em memória, de diferentes versões do mesmo componente) |

# Interoperabilidade entre linguagens

- Capacidade de utilização de componentes de *software* realizados em linguagens diferentes
  - Exemplo:
    - Realizar a classe X na linguagem A
    - Utilizar a classe X na linguagem B
    - Realizar, na linguagem C, a classe Y que deriva de X

➢ Representação intermédia com sistema de tipos independente da linguagem

**Legenda:**
♦ Definição
➢ Suporte

# Portabilidade

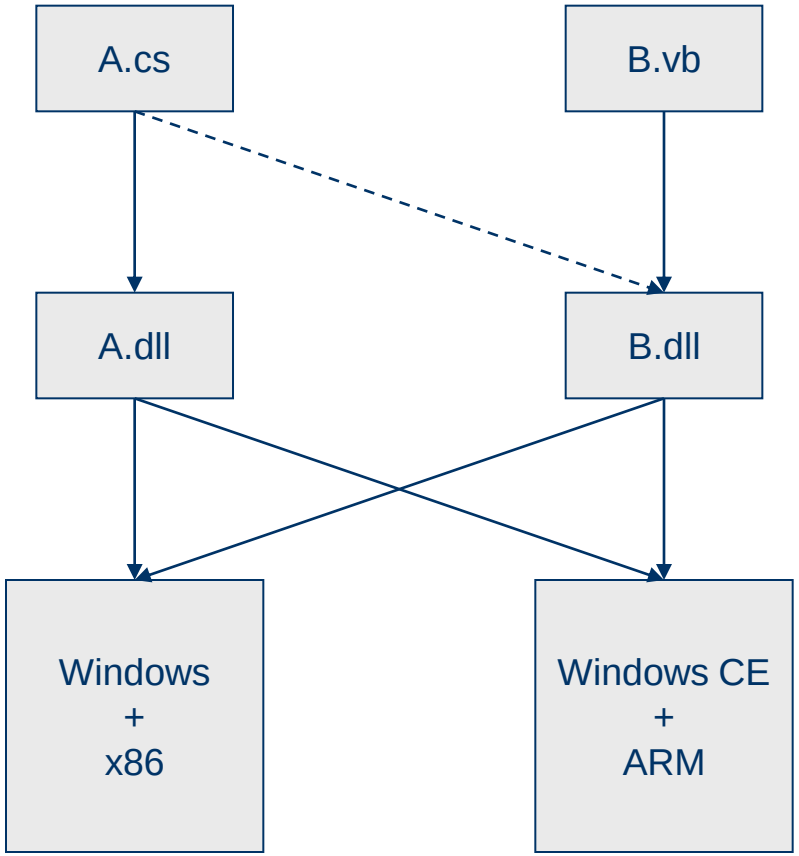
- Capacidade de execução em plataformas computacionais diferentes (OS, CPU)
- Exemplos: Windows+x86, WindowsCE+ARM, *Linux+PowerPC*

- Representação intermédia
  - Independente da arquitectura do processador
  - Mais compacta e fácil de analisar que o *assembly* nativo

- Ambiente de execução virtual
  - Tradução da representação intermédia para representação nativa
  - Abstracção do ambiente de execução fornecido pelo OS

- Biblioteca de classes
  - Conjunto de funcionalidades fornecidas de forma independente do sistema operativo

# Portabilidade e interoperabilidade




Representação Intermédia

Interoperabilidade entre linguagens

Portabilidade

# Serviços de Runtime

- Conjunto de serviços disponíveis em tempo de execução. Exemplos:
  - **Carregamento e ligação dinâmica de código**. Os componentes são carregados *on demand* (porque contém necessários ao código em execução)
  - **Reflexão (ou Introspecção).** Interface programática para ispecção alteração de objectos cujo tipo apenas é conhecido em tempo de execução
  - **Gestão automática de memória ("garbage collection")** – detecção e recolha de objectos não utilizados
  - **Serialização** - conversão de grafos de objectos em sequências de *bytes*
  - **Segurança** – "type safety" e acesso controlado a recursos
    - Verificação de "type safety" (utilização correcta de tipos) em tempo de carregamento
    - Controle de acesso baseado na identidade do utilizador ou do código
  - **Invocação de código nativo** (*unmanaged)*

- Informação de tipo em tempo de execução (Metadata)
  - Necessária para a implementação destes serviços
  - *Sempre* presente na representação intermédia

# Informação de tipo em ficheiros *header* (modelo *unmanaged*)

- Considere-se a seguinte classe em C++:

```
class UmaClasse {
  int umCampo;
  int UmMetodo(int arg1);
};

int UmaClasse::UmMetodo(int arg1){
   return arg1 + umCampo;
}
```

# Representação nativa

```
TITLE     C:\Cadeiras\MP3.ES\Progs\unmanaged_vs_managed\classe.cpp

PUBLIC    ?UmMetodo@UmaClasse@@AAEHH@Z
TEXT      SEGMENT
_this = -4
_arg1 = 8
?UmMetodo@UmaClasse@@AAEHH@Z  PROC

    push   ebp
    mov    ebp, esp

    mov    DWORD PTR      _this[ebp], ecx
    mov    ebx, DWORD PTR _this[ebp]
    mov    eax, DWORD PTR _arg1[ebp]
    add    eax, DWORD PTR [ebx]
    mov    esp, ebp
    pop    ebp
    ret    4

?UmMetodo@UmaClasse@@AAEHH@Z ENDP
_TEXT     ENDS
END
```

UmMetodo é um procedimento

**this** e **arg1** são *offsets* em relação ao *frame pointer*

**this** passado no registo ecx

Afectar [ebp-4] //**this**

Carregar [ebp-4] //**this**

Carregar [ebp-4] //**arg1**

Somar **arg1** com **this.umCampo**

# Representação nativa: propriedades

- Não existe informação sobre a classe **UmaClasse**
- O método **UmMetodo** é um procedimento
  - Não tem informação de tipo (classe, retorno, parâmetros)
- **this** é um *offset* em relação ao *frame pointer* (-4)
  - mov DWORD PTR _this[ebp], ecx
- **arg1** é um *offset* em relação ao *frame pointer* (+8)
  - mov ecx, DWORD PTR _arg1[ebp]
- **UmCampo** é um *offset* em relação a **this** (0)
  - add ecx, DWORD PTR [eax]

# Informação de tipo no modelo *managed*

- Considere-se a seguinte classe em C#

```
class UmaClasse {
  int umCampo;
  int UmMetodo(int arg1) {
       return arg1 + umCampo;
  }
}
```

# Representação intermédia: módulo “managed”

```
.class private auto ansi beforefieldinit UmaClasse
      extends [mscorlib]System.Object  {
  .field private int32 umCampo
  .method private hidebysig
        instance int32  UmMetodo(int32 arg1) cil manag
    .maxstack  2
    .locals init (int32 V_0)
    IL_0000: ldarg.1
    IL_0001: ldarg.0
    IL_0002: ldfld      int32 UmaClasse::umCampo
    IL_0007: add
    IL_0008: stloc.0
    IL_0009: br.s       IL_000b

    IL_000b: ldloc.0
    IL_000c: ret
  } // end of method UmaClasse::UmMetodo
```

**UmaClasse** é uma classe completamente caracterizada (visibilidade, herança, ...)

**umCampo** é um campo completamente caracterizado (visibilidade, tipo, nome)

**UmMetodo** é um metodo completamente caracterizado (visibilidade, tipos, ...)

Código IL

Carregar primeiro argumento

Carregar **this**

Carregar **this.umCampo**

# Representação intermédia: propriedades

- Definição completa da classe **UmaClasse**
  - ".**class** private auto ansi beforefieldinit **UmaClasse** extends [mscorlib]System.Object"
  - Visibilidade, nome, herança, ...
- Definição completa do campo **umCampo**
  - ".field private int32 umCampo"
  - Visibilidade, tipo, nome
- Definição completa do método **UmMetodo**
  - **".method** private hidebysig instance int32 **UmMetodo**(int32 arg1) cil managed"
  - Visibilidade, tipo de retorno, nome, argumentos, ...
- O código utiliza nomes e não localizações em memória
  - "ldarg.1" – Carregar o *primeiro* operando (e não [ebp+4])
  - "ldfld int32 UmaClasse::umCampo" – Carregar o campo umCampo (e não [this+0])

# Representação nativa (durante a execução)

- Representação nativa resultante da representação intermédia

```
00000000  push    ebp
00000001  mov     ebp,esp
00000003  sub     esp,0Ch
00000006  push    edi
00000007  push    esi
00000008  push    ebx
00000009  mov     edi,ecx
0000000b  mov     esi,edx
0000000d  xor     ebx,ebx
0000000f  mov     eax,dword ptr [edi+4]
00000012  add     eax,esi
00000014  mov     ebx,eax

...
```

Acesso ao campo **umCampo**

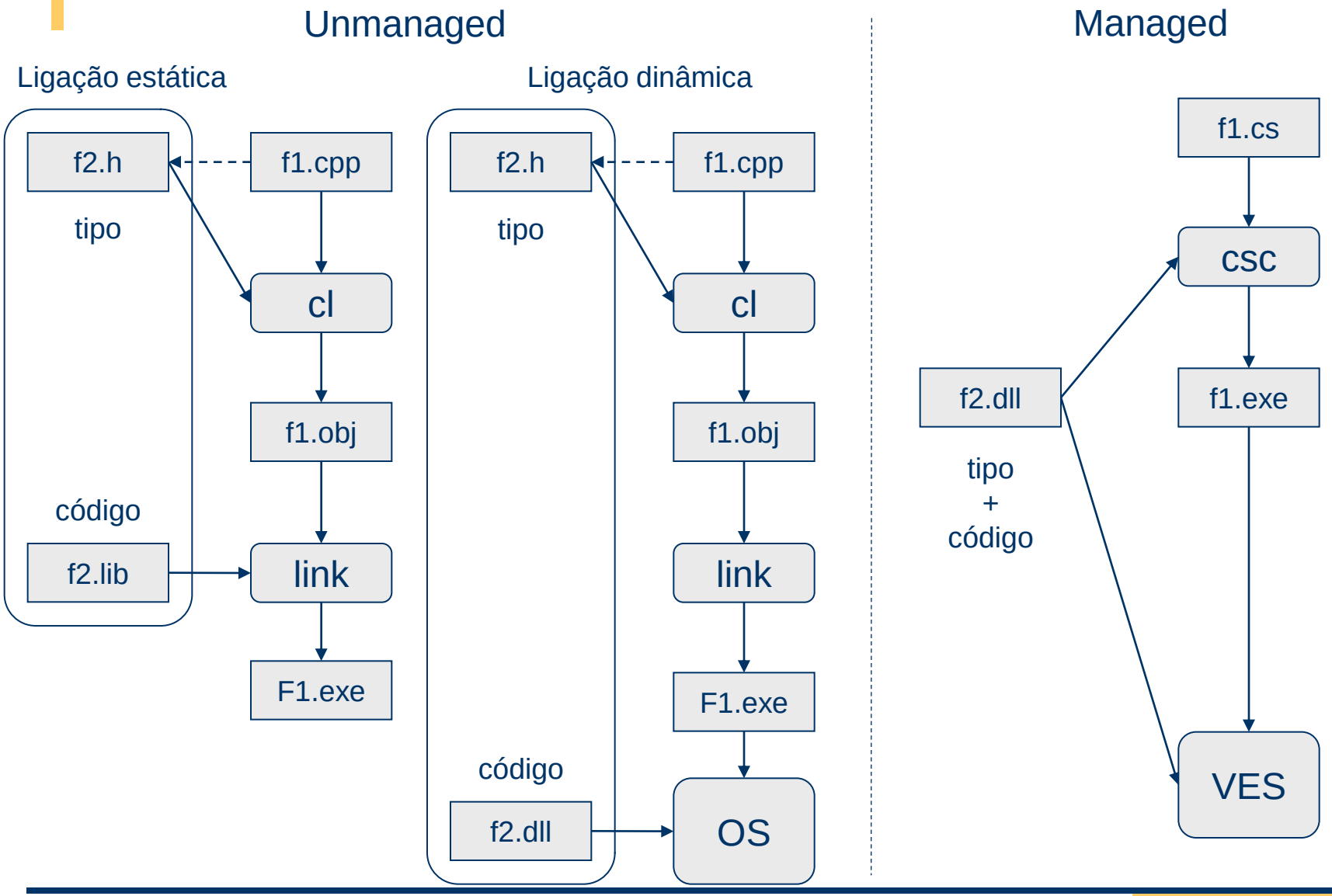
Realização da soma

# Comparação entre modelos “managed” e “unmanaged”

- Modelo “unmanaged” – Contrato físico
  - Informação de tipo (declarações) presentes no ficheiro *header*
  - Informação de tipo específica da linguagem
  - Implementação (instruções) presentes no ficheiro biblioteca
  - Problema: sincronização entre *header* e biblioteca
  - Instruções na linguagem nativa
  - Normalmente associado à construção de aplicações “monolíticas” – ligação estática

- Modelo “managed” - Contrato lógico
  - Informação de tipo (metadata) e implementação (linguagem intermédia) presentes no mesmo ficheiro
  - A ligação entre componentes é sempre dinâmica

# Modelos *unmanaged* e *managed*

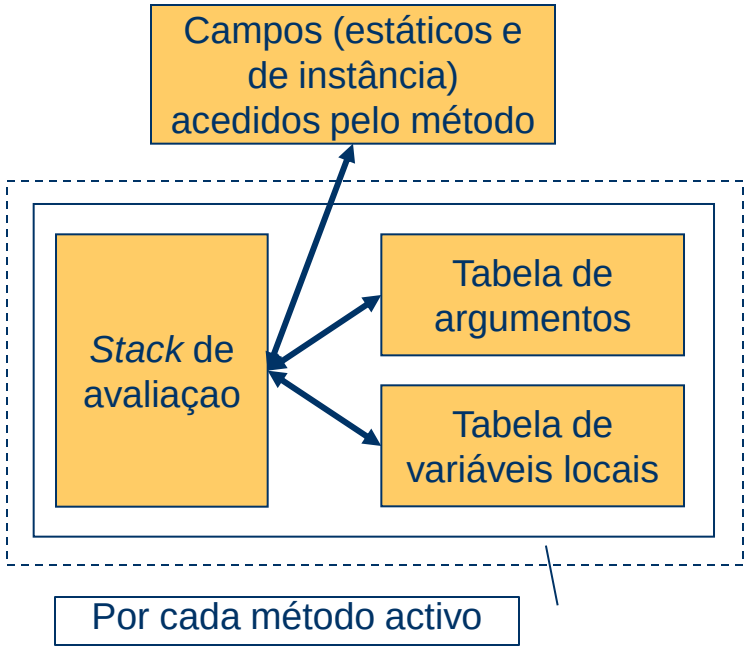


# A linguagem IL

- A linguagem IL é *stack based* (execução de uma máquina de *stack*)
  - Todas as instruções empilham os operandos (`load`*)* no *stack* e obtêm os resultados do topo do *stack* (`store`)
  - Não existem instruções para manipulação de registos
    - O que implica uma diminuição significativa no número de instruções necessárias
  - 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*).
    - Exº `add` - adiciona os dois operandos presentes no topo do *stack* e retorna o resultado no topo do *stack*

# A linguagem IL (II)

- Conjunto de instruções duma “máquina virtual”
- Quatro espaços de endereçamento
  - Tabela de argumentos do método (local)
  - Tabela de variáveis locais do método (local)
  - *Stack* de avaliação(local)
  - Campos acedidos pelo método (global)




- Formas de endereçamento
  - Tabelas de argumentos e variáveis – índice
    - **ldarg.1**
    - **stloc.0**
  - Campos – object + id. do campo
    - **ldfld int32 UmaClasse::umCampo**
      (usa como referência para o objecto o conteúdo do topo do *stack* de avaliação)
  - *Stack* – relativo ao topo
    - **add**

# A linguagem IL (III)

- Inclusão de instruções para o suporte ao paradigma da “orientação por objectos”
  - Noção de campo de objecto
    - **ldfld** e **stfld**
  - Chamada virtual
    - **callvirt**
  - Criação de instâncias
    - **newobj**, **initobj**
  - *Casting*
    - **castclass**, **isinst**
  - Excepções
    - **throw**, **rethrow**

# Módulo “managed”: constituição

- Cabeçalho PE (*Portable Execution*)
  - Tipo de ficheiro
  - Marca temporal
- Cabeçalho CLR
  - Ponto de entrada (opcional)
  - Directório para as secções *managed* *(constituídas por metadata e código intermédio)*
- Metadata
  - Descrição dos tipos definidos no módulo
  - Referência para os tipos utilizados no módulo
- Código intermédio

# a Metadata de um Managed Module

- Conjunto de tabelas com:
  - os tipos definidos no módulo - *DefTables*
  - tipos referenciados (importados) – *RefTables*

- Informação sobre tipos
  - **Sempre** incluída no *módulo* pelo compilador
    - Inseparável do módulo
    - gravada em formato binário
  - Descreve tudo o que existe no módulo *:*
    - Tipos, classes, métodos, campos, etc.

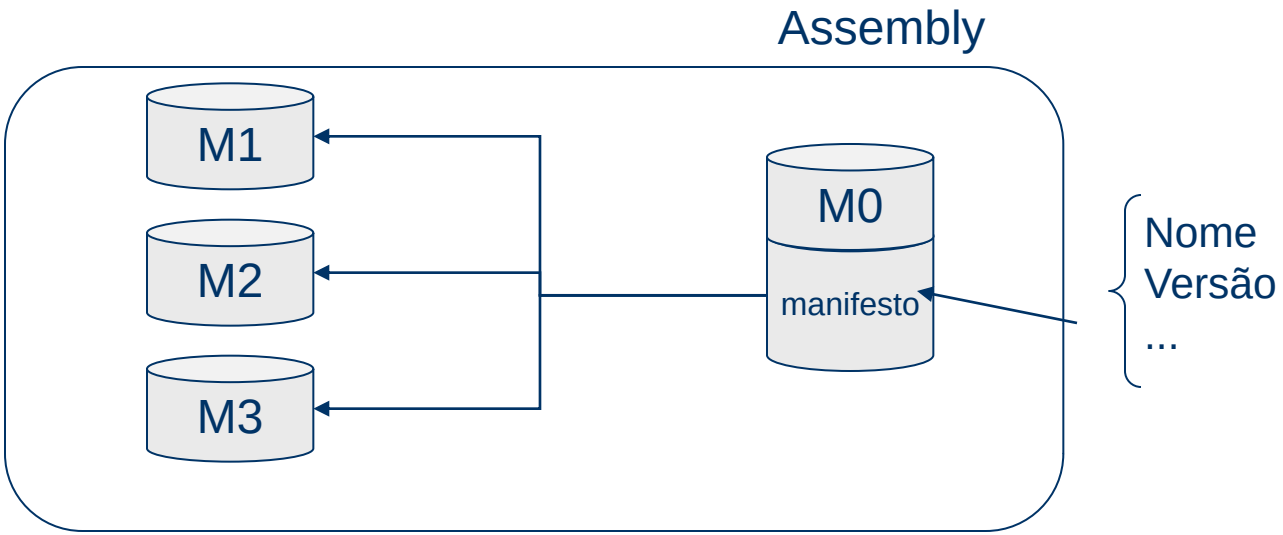
# Vantagens da Metadata

- Pelo facto dos compiladores emitirem ao mesmo tempo a *metadata* e o código dos módulos, estas duas entidades nunca podem deixar de estar em sintonia.
  - Dispondo da *metadata* não são necessários os ficheiros *header* para compilar o código das aplicações. Os compiladores poderão obter a mesma informação consultando a *metadata* dos *managed modules*.
  - O *Visual Studio .NET* utiliza a *metadata* para auxiliar o programador durante a escrita do código. A *feature* «*IntelliSense*» analisa a *metadata* e informa o programador sobre os métodos que cada tipo define e sobre os parâmetros de cada método.
  - O processo de verificação do código do CLR usa a *metadata* para garantir que o código realiza apenas operações seguras.
  - Usada para suportar os serviços de *runtime*. São exemplos:
    - 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.

# A unidade de distribuição (componente) em .NET é o Assembly

Um *Assembly* é:

- um agrupamento lógico de um ou mais *Managed Modules* e *Resource Files*
- É a unidade básica de utilização, controle de versões  e sujeita a restrições de segurança
- 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.*



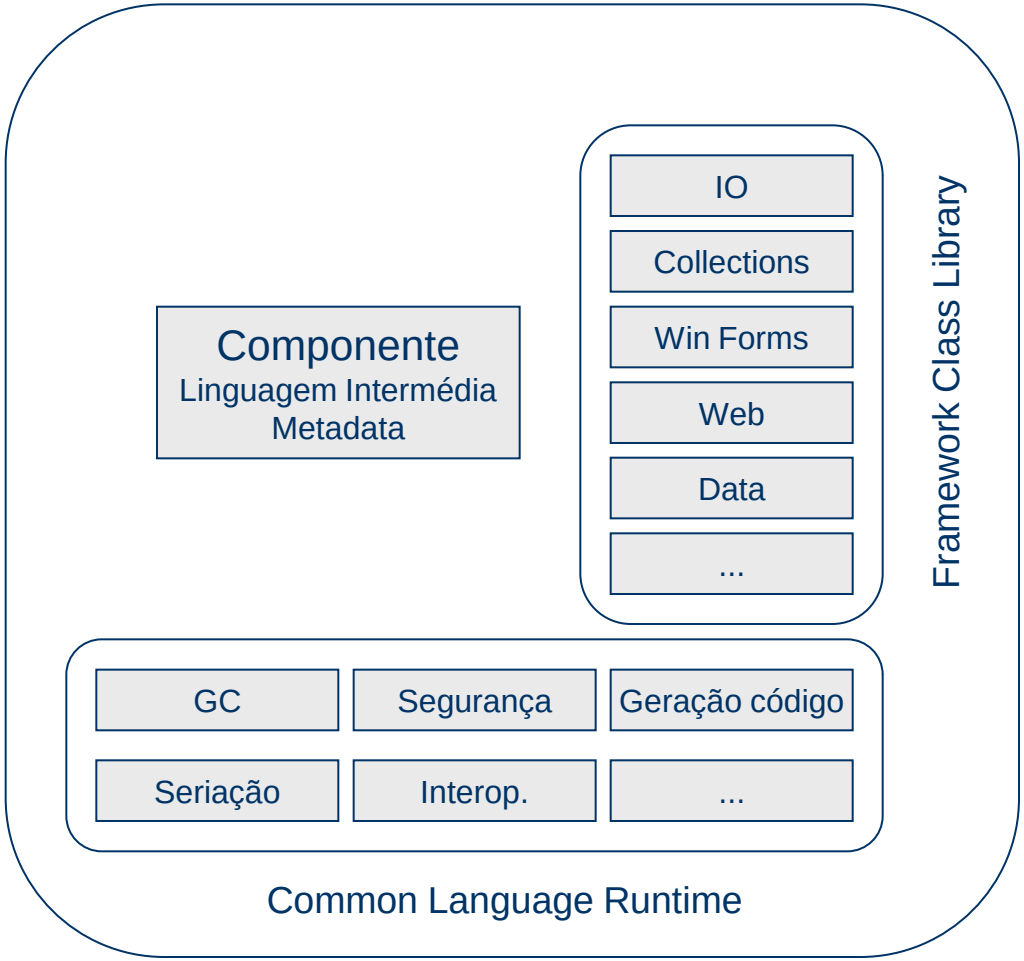
# O que contém o Manifesto de um Assembly

| **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<br>(Também existe em cada um dos módulos) | Informação usada pelo *runtime* para associar um tipo exportado ao módulo com a sua descrição/implementação |

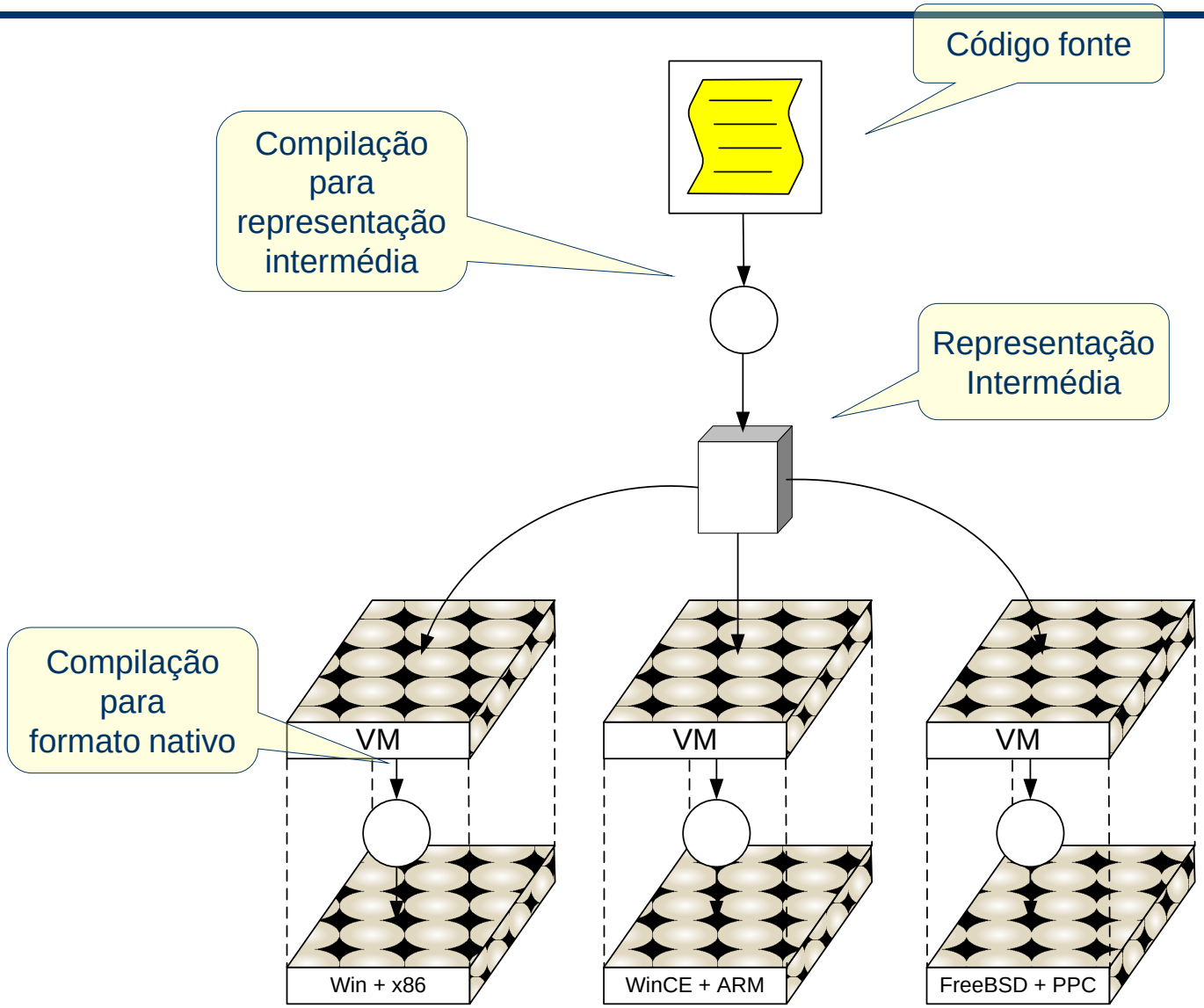
# Razões para a criação de Assemblies Multi-módulo

- 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

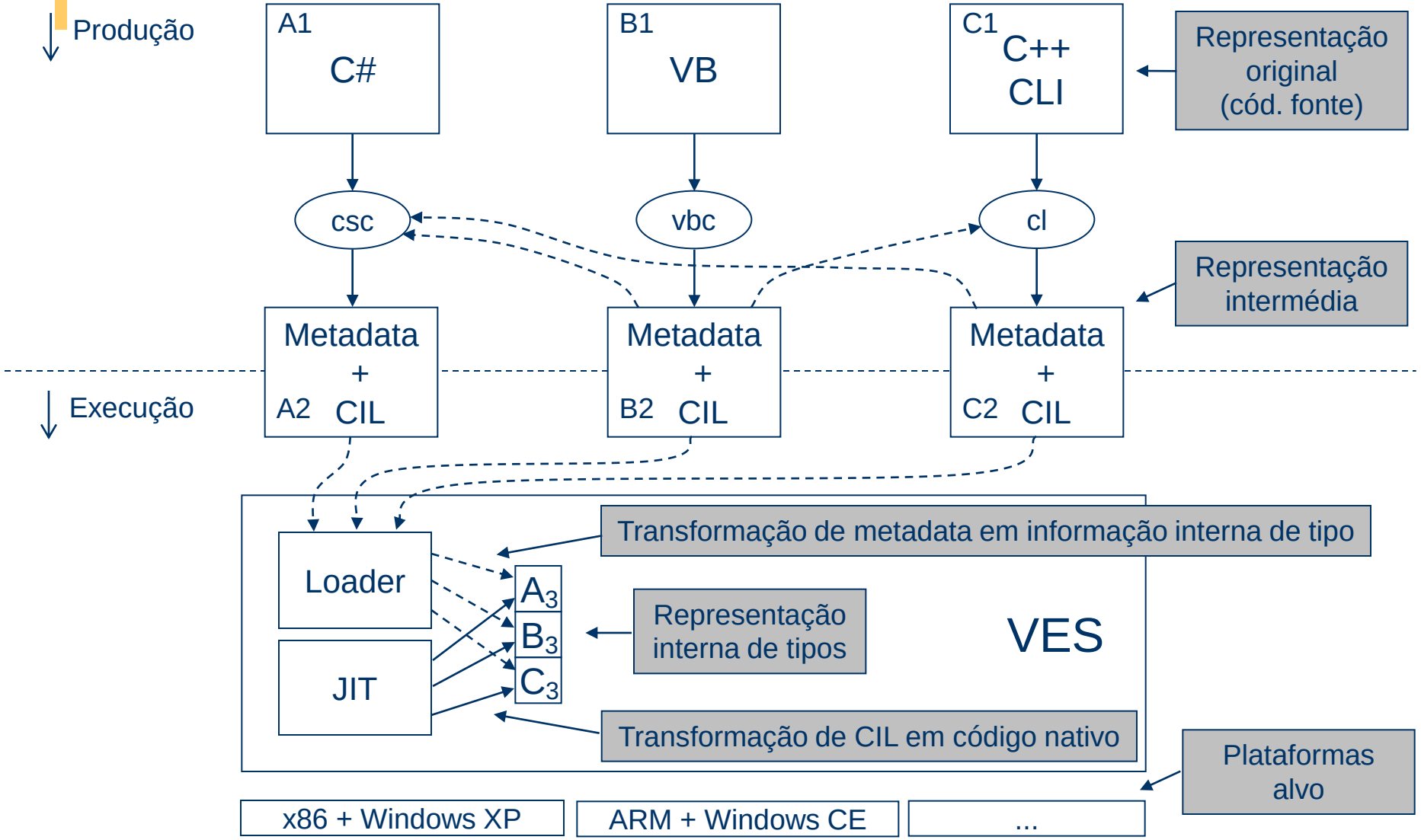
# Ambiente de execução



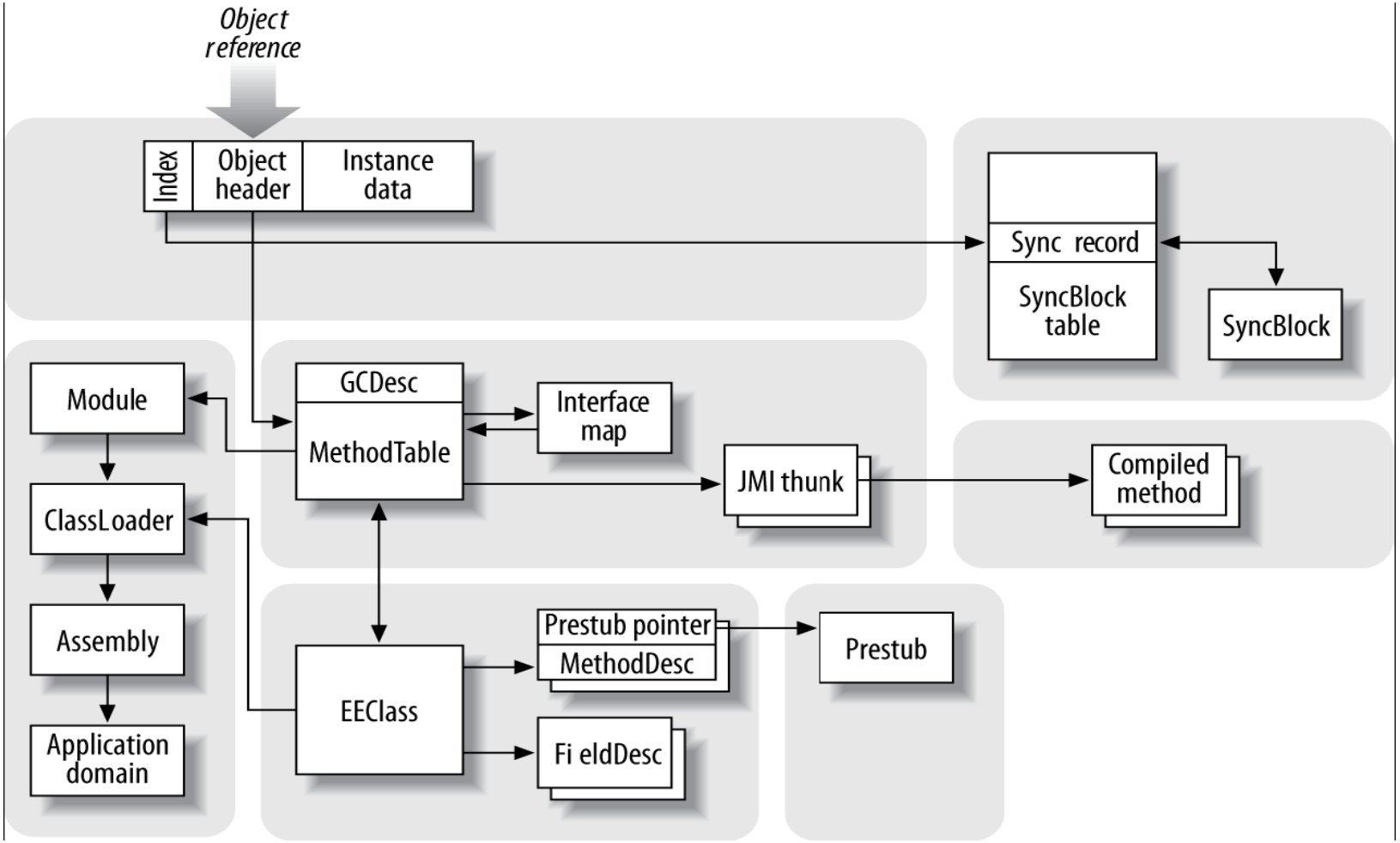
# Compilação a dois níveis



# Produção e execução de componentes



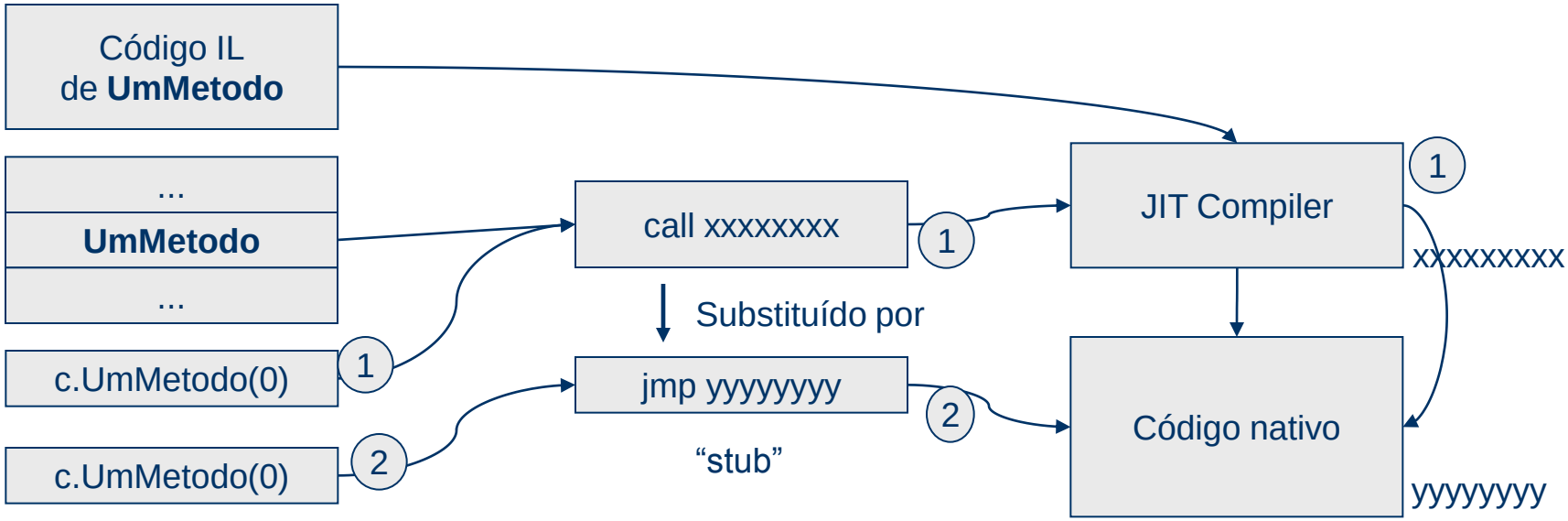
# RTTI em detalhe (1)




(1) – Retirado do livro Shared Source CLI 2.0 Internals

# 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 o fio, 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



# Consequências do modelo “JIT”

- 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)

# Verifier

- Uma das grandes vantagens da máquina virtual é a robustez do código executado
  - Aquando da compilação JIT o CLR executa um processo designado por **verificação** (pode ser executada manualmente através da ferramenta `PEVerify.exe`)
  - A verificação analisa o IL e garante que todas as instruções a realizar são seguras, p.ex.:
    - Verifica se todos os métodos são chamados com o número correcto de parâmetros e que cada parâmetro é do tipo correcto
    - No acesso a um campo verifica se o objecto acedido é do tipo correcto
    - Que o tipo de retorno de um método é usado correctamente
    - Numa operação aritmética, verifica: compatibilidade dos operandos
  - A verificação garante código seguro no sentido de robusto (*type safety*).

# 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