UNIVERSIDADE FEDERAL DE PERNAMBUCO - UFPE

CENTRO DE INFORMÁTICA - CIN

TRABALHO DE GRADUAÇÃO

ÁREA: COMPILADORES

DISCENTE: NELSON GUEIROS BARLOW

ORIENTADOR: PROF° ANDRÉ LUÍS DE MEDEIROS SANTOS

AVALIADOR: PROF° HERMANO PERRELLI DE MOURA

RECIFE – PE

2008.1

UNIVERSIDADE FEDERAL DE PERNAMBUCO - UFPE

CENTRO DE INFORMÁTICA - CIN

TRABALHO DE GRADUAÇÃO

ÁREA DE COMPILADORES

PROGRAMAÇÃO ORIENTADA A ASPECTOS NO MICROSOFT PHOENIX FRAMEWORK

DISCENTE: NELSON GUEIROS BARLOW

ORIENTADOR: PROF° ANDRÉ LUÍS DE MEDEIROS SANTOS

AVALIADOR: PROF° HERMANO PERRELLI DE MOURA

RECIFE – PE

2008.1

AGRADECIMENTOS

À Deus por ter me dado essa oportunidade;

Ao meu pai e minha mãe por me fazerem acreditar que nada está além do meu alcance;

À Rachel pela sua força que é uma grande inspiração;

À Amy que me ensina a usar a criatividade no meu mundo não-criativo;

À Marcela por me fazer buscar meus objetivos até quando eu não estou interessado;

À Nelsinho e Glorinha que me ajudam a perceber que a vida é muito maior do que eu posso sequer imaginar.

Finalmente ao Professor André Santos pela orientação.

RESUMO

Programação orientada a objetos tem sido apresentada como o paradigma que auxilia a engenharia de software. Isto porque a orientação a objetos tem sido o modelo que mais se adéqua a problemas reais. Porém, tem se encontrado vários problemas (aspectos) onde a orientação a aspectos não é o suficiente para capturar claramente todas as decisões de design que um programa deve implementar.

O propósito deste trabalho é apresentar uma ferramenta que representa o estado da arte em termos de processamento de linguagem- o Microsoft Phoenix Framework- e sua capacidade de auxiliar os desenvolvedores com problemas que desafiam os paradigmas comuns nos ambientes de desenvolvimento como o de orientação a objetos. Para isso, será desenvolvido um protótipo que permite que um crosscutting concern seja acrescentado após a implementação de um programa.

SUMÁRIO

1. INTRODUÇÃO

Programação orientada a objetos tem sido apresentada como o paradigma que auxilia a engenharia de software. Isto porque a orientação a objetos tem sido o modelo que mais se adéqua a problemas reais. Porém, tem se encontrado vários problemas (aspectos) onde a orientação a aspectos não é o suficiente para capturar claramente todas as decisões de design que um programa deve implementar.

O desenvolvimento das tecnologias de processadores de linguagens, e mais particularmente das tecnologias de compiladores, está muito ligada ao desenvolvimento da ciência da computação. Soluções ad hoc para os problemas da computação têm sido substituídos por soluções mais elegantes e genéricas que tem se tornado possível devido aos avanços dos processadores de linguagens.

O propósito deste trabalho é apresentar uma ferramenta que apresenta o estado da arte em termos de processamento de linguagem- o Microsoft Phoenix Framework- e sua capacidade de auxiliar os desenvolvedores com problemas que desafiam os paradigmas comuns nos ambientes de desenvolvimento como o de orientação a objetos. Para isso, será desenvolvido um protótipo que permite que um crosscutting concern seja acrescentado após a implementação de um programa.

2. PROGRAMAÇÃO ORIENTADA A ASPECTOS

No contexto de engenharia de software, o paradigma de programação orientado a aspectos(AOP) auxiliam aos programadores a separar as preocupações (concerns), mais precisamente os crosscutting concerns, como avanço na modularização. As principais técnicas de AOP utilizam-se de mudança na linguagem de programação para atingir esse objetivo (Kiczales et al, 1996).

A separação dos crosscutting concerns requer a separação do código em partes distintas com o mínimo de interseção nas suas funcionalidades. Todas as linguagens de programação dão suporte a separação e encapsulamento de concerns em entitades únicas. Por exemplo, procedimentos, pacotes, classes e métodos auxiliam ao programador encapsular concerns em entidades únicas. Porém alguns concerns desafiam estas formas de encapsulamento. Estes são os crosscutting concerns e são chamados assim porque eles cortam através de múltiplos módulos em um programa (Eaddy et al, 2007).

2.1 Conceitos Básicos

Nesta seção será aprofundado alguns conceitos referentes à programação orientada a aspectos.

2.1.1 Crosscutting Concerns

Como brevemente definido acima, crosscutting concerns são preocupações que se encontram espalhadas pelo código, isto é, não-modularizados (Eaddy et al, 2007). Esta não-modularização é responsável pela degradação da qualidade do código.

Infelizmente, identificar esses concerns é uma tarefa difícil e são necessários métricas bem definidas para determinar o quanto um certo concern é crosscutting e quanto será o benefício em modularizá-lo.

A especificação de um programa é uma simples descrição das funcionalidades do programa. A especificação pode se física ou lógica. Muitos autores definem candidatos a crosscutting concerns como sendo elementos contidos na especificação lógica do programa. Portanto, a especificação do programa contém

o domínio de concerns que o programa representa. Os crosscutting concerns representam um subdomínio contido no domínio dos concerns.

2.1.1.1 Scattering

Segundo Figueiredo et al, 2005, um concern é considerado scattered quando ele está presente em mais de uma entidade. No caso de orientação a objetos, um concern é scattered quando presente em mais de uma classe.

Uma característica que define um módulo é que sua implementação é local, portanto, um concern que é scattered, por definição, não é modular (Eaddy et al, 2007).

2.1.1.2 Tangling

Um concern é considerado tangled quando ele e ao menos um outro concern estão relacionados na mesma entidade (Figueiredo et al, 2005).

O exemplo em código Java a seguir ilustra um concern de tratamento de exceção tangled com concern de transação, concern de negócio, concern de segurança e concern de log.

```
void transfer(Account fromAccount, Account toAccount, int amount) throws Exception {
  if (!getCurrentUser().canPerform(OP_TRANSFER)) {
    throw new SecurityException();
  }

  if (amount < 0) {
    throw new NegativeTransferException();
  }

  Transaction tx = database.newTransaction();
  try {

    if (fromAccount.getBalance() < amount) {
      throw new InsufficientFundsException();
    }
    fromAccount.withdraw(amount);
    toAccount.deposit(amount);
```

```
    tx.commit();
    systemLog.logOperation(OP_TRANSFER, fromAccount, toAccount, amount);
  }
  catch(Exception e) {
    tx.rollback();
    throw e;
  }
}
```

Tabela 1: Exemplificação de Tangling

2.1.1.3 Guia de identificação de Concern

O guia de identificação proposto por Eaddy et al, 2007, auxilia na desambiguação do que é um concern.

O domínio de concerns deve ter objetivo e satisfazer a critérios pré-definidos.
O domínio de concerns deeve ser finito.

Tabela 2: Guia de Identificação de Concerns

Componentes Primitivos: Associar um concern a um componente primitivo se e somente se a remoção do concern requer a remoção completa ou modificação do componente e suas referências. Remoção completa implica que nenhum componente ficara associado a ele.
Componentes do tipo Container: Se todas as referências dele e os componentes contidos em um componente do tipo container tem a mesma associação de concern, o componente automaticamente recebe a associação.
Declaração: Se todas as referências para uma declaração têm a mesma associação, a declaração automaticamente recebe a associação. Caso não, não será associada.
Subclasses: Se todas as subclasses de uma classe base tem a mesma associação, a classe base automaticamente recebe a associação. Caso não, não será associada.

Métodos Virtuais: Se todos as sobrescrições do método virtual tem a associação, então o método virtual é automaticamente associado; caso não, não será associado.

Tabela 3: Guia de associação de concerns beseado à componentes de código

2.1.1.4 Métricas de Concerns

Degree of Scattering (DOS)

Concentração (CONC) mede quantas linhas de um concern específico estão contidas em um componente específico t. A fórmula de grau de scattering é dada por:

$$DOS(s) = 1 - \frac{|T| \sum_t^T \left(CONC(s,t) - {}^1/_{|T|}\right)^2}{|T| - 1}$$

Tabela 4: Fórmula de Degree of Scattering (DOS) (Eaddy, 2005)

As peculiaridades matemáticas desta fórmula fogem do escopo deste trabalho. Porém ela apresenta algumas propriedades importantes:

- O DOS é normalizado para ser 0 (completamente localizado) e 1(completamente deslocalizado, uniformemente distribuído) para que concerns possam ser comparados.
- DOS é proporcional ao número de componentes relacionados ao concern.
- DOS é inversamente proporcional a concentração, isto é, quanto menos concentrado um concern, mais espalhado ele será.

Após calculados os valores DOS dos diferentes concerns, pode-se determinar quais concerns são melhor candidatos a serem separados do código, isto é, quais aspectos estão não-modularizados e estão afetando a qualidade do código.

2.1.1.2 Weaving

Soares et al em 2002 definiu que o weaving (costura) consiste na ligação entre o código de componentes (orientado a objetos) com o código dos aspectos de maneira a produzir uma única saída com todas as funcionalidades da especificação implementadas.

A figura a seguir ilustra as diferentes fases do desenvolvimento Orientado a Aspectos:

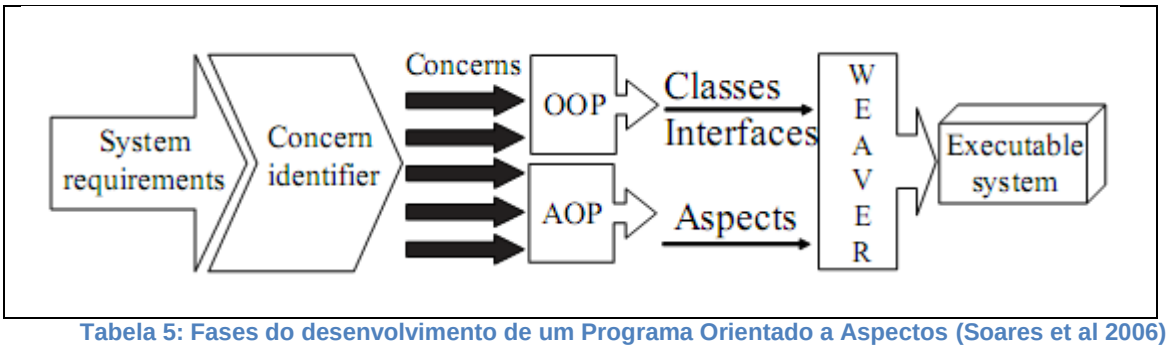

Tabela 5: Fases do desenvolvimento de um Programa Orientado a Aspectos (Soares et al 2006)

3. ARQUITETURA DO .NET FRAMEWORK

Para o entendimento completo do Phoenix, é necessário conhecer, mesmo que apenas superficialmente, a arquitetura do .NET Framework. O objetivo deste capítulo é introduzir os componentes da arquitetura .NET Framework dando uma visão geral do seu funcionamento.

3.1 Biblioteca de Classes

O .NET Framework é composto por duas partes. Uma dessas partes é a biblioteca de classes desenvolvida para solucionar problemas comuns de programação. Alguns exemplos de problemas comuns solucionados pela biblioteca são: leitura de arquivo, escrever em arquivo, interação com banco de dados, entre outros. Todas as linguagens de programação do .NET Framework podem utilizar

essa biblioteca de classes. Uma explicação mais detalhada sobre como a Microsoft atingiu a interoperabilidade de linguagens computacionais será dada mais adiante. Por enquanto, segue um detalhamento da divisão da biblioteca.

- Base Class Library (BCL): Um subconjunto da biblioteca de classes, a BCL contém apenas as classes essenciais para o correto funcionamento do programa. Todas as implementações do .NET Framework possuem a BCL (Box & Sells, 2002).
- Framework Class Library (FCL): É o conjunto de todas as classes do Framework. Exemplo de classes da FCL que não se encontra na BCL incluem: WinForms, ADO.NET, Language Integrated Query (LINQ), etc.

3.2 Ambiente Gerenciado

A segunda parte do .NET Framework é a máquina virtual capaz de executar código de diversas linguagens de programação diferente em um ambiente gerenciado. A visão da Microsoft com o Framework foi dar o poder de diferentes linguagens ao programador. A arquitetura do Framework foi desenvolvida de maneira que qualquer linguagem que o possua como alvo, poderá ser executada sobre um ambiente gerenciado, isto é, sobre a máquina virtual do Framework. Considerando que linguagens computacionais são apenas um meio de expressar instruções, cada uma possui um conjunto de regras de sintaxe e análise que examinam o código fonte e verifica se faz sentido. Linguagens diferentes permitem programar usando sintaxes diferentes. Esse poder não deve ser subestimado. Linguagens funcionais, por exemplo, permitem expressar programas matemáticos com maior produtividade que uma linguagem dinâmica do tipo Perl. Isso significa que o programador poderá escolher a linguagem que melhor se adapte e permita

maior produtividade ao seu problema sem se preocupar com interoperabilidade com outros módulos do seu sistema.

A visão da Microsoft para o .NET Framework foi alcançada com sucesso. Hoje, várias linguagens não-Microsoft já são executadas sob a máquina virtual do .NET Framework. Alice, APL, COBOL, Component Pascal, Eiffel, Fortran, Haskell, Mercury, ML, Mondrian, Oberon, Perl, Python, RPG, Scheme, e Smalltalk são apenas alguns exemplos dessas linguagens. A arquitetura do Framework foi essencial para seu sucesso.

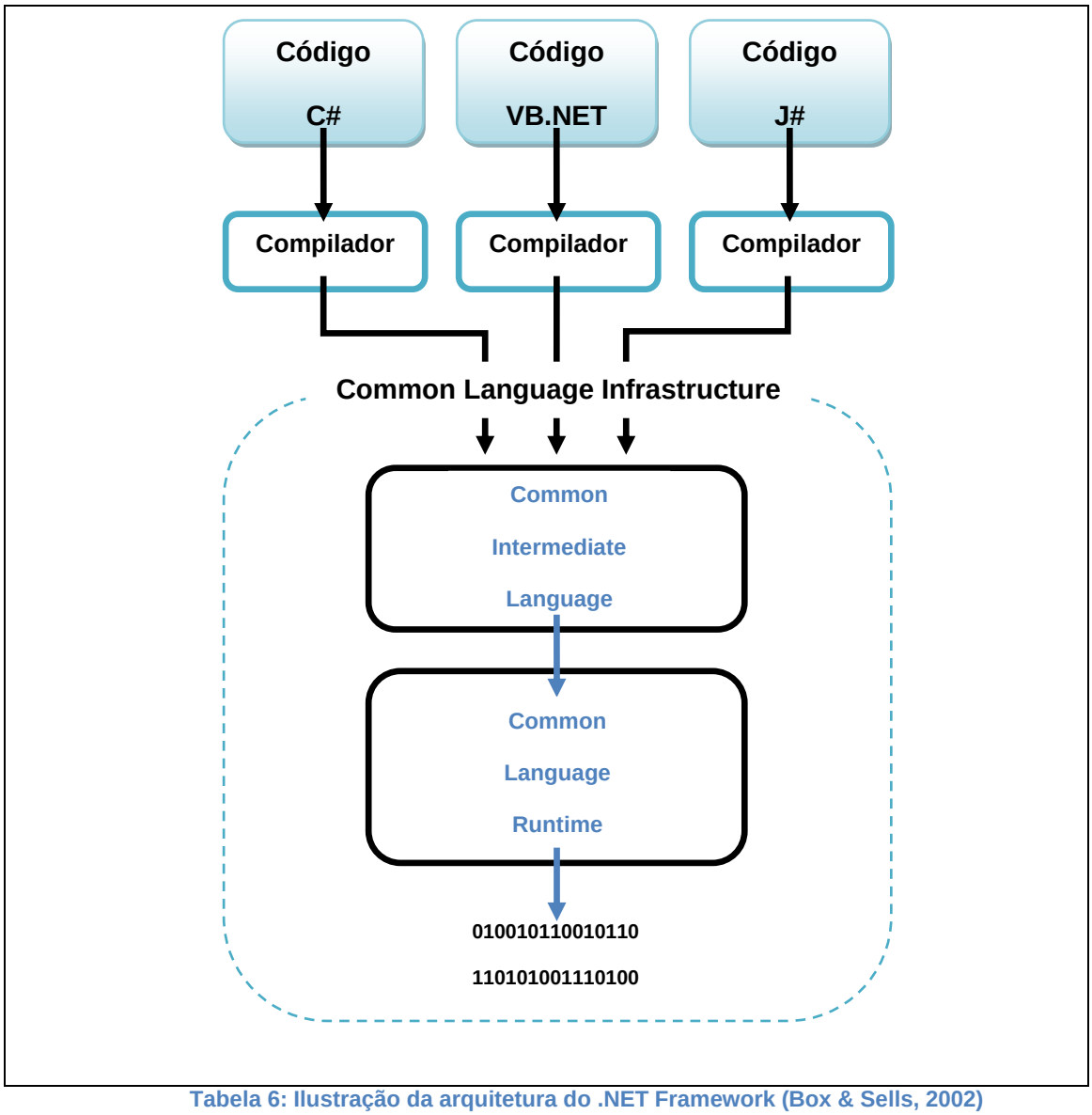

Tabela 6: Ilustração da arquitetura do .NET Framework (Box & Sells, 2002)

3.3 Common Language Infra-Structure (CLI)

Em agosto de 2000, em uma parceria da Microsoft com a Hewlett-Packard e Intel, houve um esforço para padronizar a CLI. Em dezembro de 2001, a CLI foi ratificada pela primeira edição da Ecma International (Common Language Infrastructure, 2006). Em abril de 2003 a Organisation internationale de normalisation (ISO) também a ratificou.

O ECMA-335 e o ISO/IEC 23271:2006 definem o padrão internacional da CLI em que aplicações escritas em diferentes linguagens de alto nível podem ser executadas em sistemas de ambientes diferentes sem a necessidade de reescrever essas aplicações para satisfazer as características únicas desses ambientes.

Com a padronização da CLI, a Microsoft não é a única empresa que pode implementar a mesma. A empresa deseja que ela seja apenas uma de várias empresas que implementam a CLI. Certamente que a Microsoft deseja desenvolver a melhor implementação em termos de desempenho e funcionalidades, porém, com a padronização, outras empresas poderão competir com a Microsoft.

3.3.1 Common Language Runtime (CLR)

A CLR é a implementação da Microsoft do padrão da CLI. A CLR é compatível apenas com a o sistema operacional Windows. A CLR executa um tipo de bytecode chamada Common Intermediate Language CIL. Em tempo de compilação, o código fonte é transformado em CIL e em tempo de execução, o respectivo código CIL é compilado para linguagem de máquina pelo compilador Just-in-time (compilador JIT) da CLR.

3.3.1.1 Common Type System (CTS)

A CLR é fortemente tipada. Tipos expõem funcionalidades para aplicações e componentes e são o mecanismo pelo qual uma linguagem de programação consegue reutilizar código escrito em outra linguagem de programação. Como os tipos estão na raiz da CLR, a Microsoft criou uma especificação formal- o CTS- que descreve como tipos são criados e como se comportam (Richter, 2002).

O CTS especifica que um tipo pode ter zero ou mais membros. Os membros podem ser dos seguintes tipos:

- Campo: Uma variável de dado que faz parte do estado do objeto. São identificados pelo nome e tipo.
- Método: Uma função que realiza uma operação no objeto. Mudando o estado do objeto. Métodos têm nome, assinatura e modificadores. A assinatura especifica a chamada do método, isto é, o número de parâmetros e suas seqüência, os tipos dos parâmetros e o tipo do valor retornado pelo método.
- Propriedade: Para quem chama, esse método parece um campo. Mas para quem implementa, parece um método. Propriedades permitem ao implementador validar a entrada e estado do objeto antes de acessar o valor e/ou calcular o valor apenas quando necessário. Ele permite que quem use o tipo utilize sintaxe simplificada. Propriedades também permitem a criação de campos somente leitura e somente escrita.
- Evento: Permite a notificação entre objetos interessados. Por exemplo, um clique do mouse pode gerar um evento que notifica um objeto.

O CTS também especifica as regras de visibilidade e de acesso a membros do tipo. A lista a seguir demonstra as opções de controle de acesso de um método ou campo do tipo.

- Private: O método só pode ser chamado por outros métodos do mesmo tipo de classe.
- Family: O método só pode ser chamado por classes derivadas, independentemente de estarem no mesmo assembly ou não. Observação: Em algumas linguagens chamado de *protected*.
- Family e Assembly: O método só pode ser chamado por tipos derivados, mas apenas se os tipos derivados se encontrarem no mesmo assembly.
- Assembly: O método só pode ser chamado por código dentro do assembly. Não necessariamente do mesmo tipo. Observação: Algumas linguagens o chamam de *internal*.
- Family ou Assembly: O método só pode ser chamado de tipos derivados em qualquer assembly ou tipos do assembly que o contém. Observação: em C# chamado de *protected internal*.
- Public: Pode ser chamado de qualquer código de qualquer assembly.

Além disso, o CTS define regras que definem a hierarquia de tipos, funções virtuais, tempo de vida do objeto, entre outros. Essas regras foram desenvolvidas para garantir compatibilidade com a semântica das linguagens de programação modernas. Desenvolvedores de linguagens .NET não precisam se preocupar com detalhes de regras do CTS pois as linguagens utilizadas irão expor sua própria

sintaxe e regra de tipos e o compilador irá mapear a sintaxe especifica da linguagem para a CIL quando for gerado o modulo gerenciado.

Uma das regras mais importantes para o entendimento do CTS é a regra que todos os tipos herdam do tipo predefinido System.Object. Esse Object(objeto) é a raiz de todos os tipos e portanto garante que todos as instâncias de tipos tenham um mínimo de comportamento semelhante. Especificamente, o tipo System.Object permite que se faça o seguinte:

- Compare a igualdade de duas instâncias.
- Obter um código hash para o objeto.
- Fazer uma cópia bit-a-bit da instância.
- Acessar o tipo da instância.
- Obter uma representação do objeto do tipo string.

3.3.1.2 Common Language Specification

A tecnologia antecessora à .NET da Microsoft, a Component Object Model(COM), permitia que objetos criados em diferentes linguagens se comunicassem. O CLR, no entanto, permite que objetos criados em uma linguagem sejam tratados como irmãos por uma linguagem completamente diferente. Ou seja, os objetos não só se comunicam mais podem ser estendidos e modificados em outra linguagem.

Embora esta integração seja ideal, na prática as diferenças das linguagens de programação não permitem o cumprimento perfeito desta integração. Isso se deve ao escopo atingido pelas diferentes mesmas. Por exemplo, algumas linguagens não dão suporte à sobrecarga de operadores, métodos com número variável de parâmetros, e até a inteiros sem sinais.

Para se utilizar tipos que possam facilmente ser utilizador por outras linguagens de programação, é necessário utilizar apenas características da linguagem de programação que são garantido de funcionar com qualquer outra linguagem. Para auxiliar esse requisito, a Microsoft criou o Common Language Specification (CLS) que detalha o mínimo conjunto de funcionalidades que um compilador precisa suportar caso o alvo seja a CLR.

O CLR/CTS suporta mais recursos que o subconjunto definido pelo CLS, então caso a interoperabilidade entre linguagens não seja crucial, a complexidade dos tipos é limitada apenas pelas funcionalidades da linguagem de programação. Especificamente, o CLS define regras para tipos e métodos que serão acessíveis externamente para qualquer linguagem de programação que tenha a CLR como alvo.

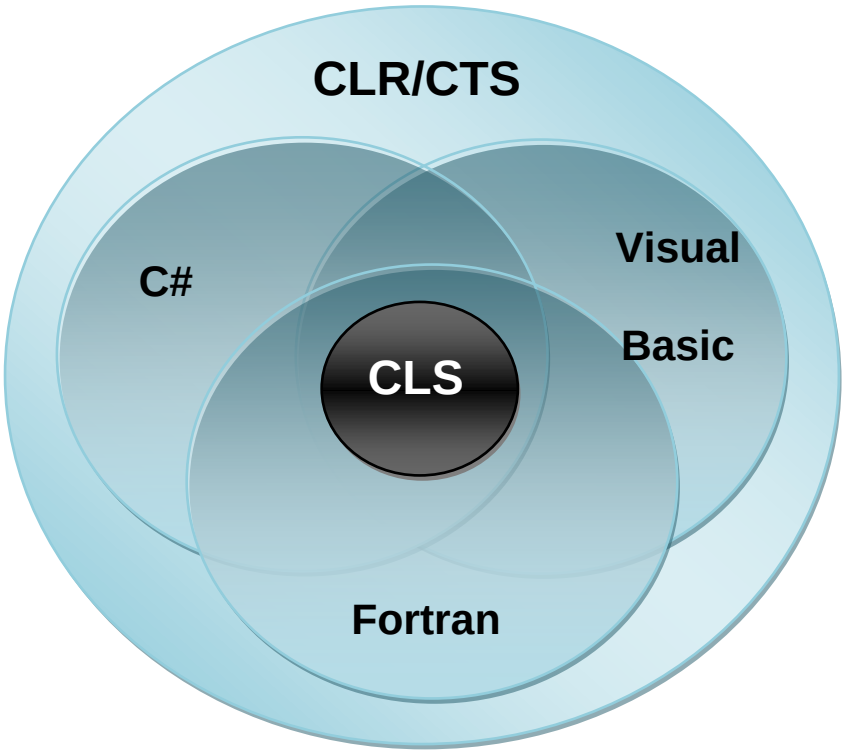

Figura 2: Ilustração dos domínios das linguagens em relação CLR/CTS e o CLS.

3.3.1.3 Metadata

Metadata é simplesmente um conjunto de tabelas de dados que descreve o que é definido no arquivo compilado, como tipos e membros. Além disso, metadata também tem tabelas que indicam o que o código compilado referencia, como tipos importados e seus membros. Destaca-se que a metadata sempre está contido no mesmo arquivo que o código e não é possível separá-los em arquivos diferentes. Como o compilador produz metadata e o código ao mesmo tempo e os combina no arquivo resultante, a metadata e o código gerado estão sempre sincronizados.

Alguma vantagens da utilização de metadata no .NET Framework:

- Metadata torna possível a eliminação de arquivos de biblioteca e arquivos cabeçalho já que toda informação sobre os tipos e membros referenciados estão contidos nos arquivos que implementam os tipos e membros referenciados.
- A IDE do Visual Studio se utiliza do metadata para auxiliar o desenvolvedor. O recurso de IntelliSense analisa o metadata para dizer que métodos um tipo oferece e que parâmetros este método espera.
- A verificação de código da CLR utiliza metada para assegurar que o código realiza apenas operações seguras.
- Metadata permite que o garbage collector (GC) gerencie a vida do objeto. Para qualquer objeto o GC verifica o tipo do objeto e, através do metada, verifica que campos do objeto são referenciados por outros objetos.

3.3.1.4 Common Intermediate Language

A Common Intermediate Language(CIL), anteriormente conhecida por Microsoft Intermediate Language (MSIL), é a linguagem de mais baixo nível do .NET Framework que ainda independente de arquitetura. Ela é executada pela máquina virtual CLR. Linguagens de programação que tem a CLR como alvo, precisam ser compiladas primeiramente para CIL.

Características CIL:

- Orientada a objetos.
- Baseado em pilha.
- Verificação de tipos em tempo de execução.

3.4 Assembly

Assembly é um conceito abstrato e é composto por um agrupamento lógico de um ou mais módulos gerenciados(arquivos de c código) ou arquivos de recurso. No .NET Framework, assembly é a menor unidade de reuso, segurança e versões. Dependendo da maneira que se utiliza o compilador, o assembly pode ser composto por um ou mais arquivos.

Características importantes de um assembly:

- Assembly define os tipos reusáveis.
- Assembly é marcado com um número de versão
- Assembly pode ter informações de segurança associados.
- Assembly é composto pela linguagem CIL

4. PHOENIX FRAMEWORK

Phoenix é um projeto conjunto entre as equipes do Visual C++, Microsoft Research e .NET Common Language Runtime da Microsoft. Seu objetivo é se tornar a base para a próxima geração de compiladores. A visão do Microsoft com o Phoenix é aumentar as inovações com geração de código e ferramentas de software.

O Phoenix foi desenvolvido com o propósito de satisfazer a dois públicos diferentes:

- **Acadêmicos**: Pesquisadores que desenvolvem ferramentas de programação e ferramentas de linguagens.
- **Desenvolvedores**: Necessitam de compiladores e ferramentas de nível industrial.

Porém, Phoenix representa um potencial impressionante e pode atingir ainda outros públicos como:

- **Indústria de Hardware**: Ao lançar um novo processador, há uma demora de aproximadamente dois anos até que os compiladores convencionais se adéqüem e façam uso das diferenças na arquitetura. Com o Phoenix será possível que a indústria de hardware lance o processador juntamente com um plug-in que permitirá a utilização de todas as capacidades do processador no dia do seu lançamento.
- **Educadores**: O processo de criação de um compilador pode ser demasiadamente complexo e os resultados não satisfatórios. O Phoenix

permite a criação de soluções para problemas do mundo real, o que tem ajudado bastante os educadores.

Phoenix é um Framework para um backend de um compilador, onde otimização e geração de código são realizados de maneira customizada. No Phoenix, as tarefas do compilador são divididas em fases (phases) que são executadas seqüencialmente. A separação em fases permite ao Phoenix a customização de qualquer parte da seqüência. Ferramentas de análises podem ser desenvolvidas e incluídas como uma fase específica do processo. Uma nova fase é inserida no Phoenix por meio de um módulo de biblioteca (DLL) que é um plug-in para o compilador.

A figura a seguir ilustra as possíveis entradas e saídas do Phoenix Framework:

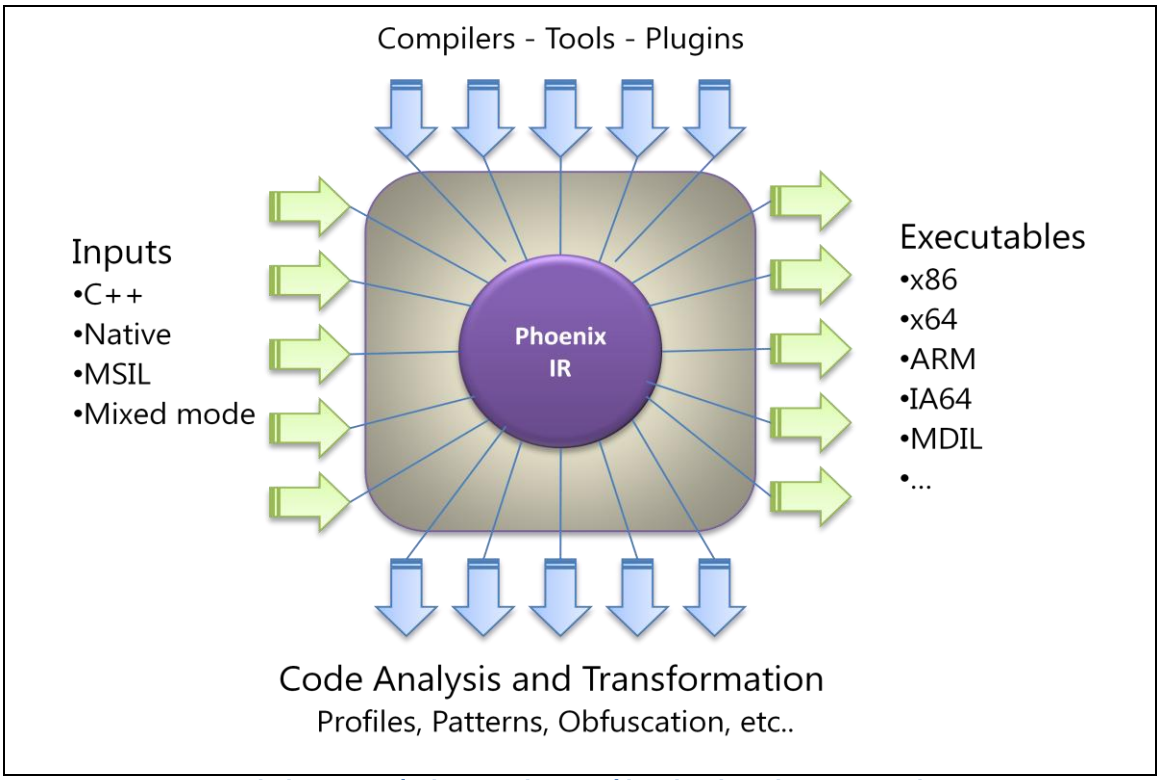

Tabela 7: Possíveis entradas e saídas do Phoenix Framework

4.1 Intermediate Representation

Observa-se na figura que o núcleo do Phoenix é composto pelo Intermediate Representation (IR). O IR nada mais é que uma representação do fluxo de instruções de uma função como uma série de operação de fluxo de dados. O IR representa uma função em diversos níveis de abstração, de alto-nível, até baixo-nível. O IR explicitamente representa todo o controle de fluxo e de dados de um fluxo de instruções (Phoenix Documentation, 2008).

Uma vantagem bastante positiva do IR em relação à outras linguagens intermediárias é a representação única. No caso do CIL, para se entender o comportamento de uma determinada função em um determinado contexto, é necessário recorrer à pilha. Não há uma pilha no IR e todas as informações necessárias para entendimento do contexto já se encontra no operando.

Os níveis de abstração do IR e suas características com relação a dependência de arquitetura e dependência do ambiente de execução seguem:

Nível IR	**Dependente de arquitetura**	**Dependente de Ambiente de Execução**
High Level IR (HIR)	Não	Não
Mid-level IR (MIR)	Não	Sim
Low-Level IR (LIR)	Sim	Sim
Encoded IR (EIR)	Sim	Sim

4.1.1 High-Level Intermediate Representation

- A forma mais abstrata do IR
- Bom alvo para análises gerais e do front-end

- Opcodes independente de arquitetura, isto é, opcodes abstratos.
- Operações específicas do ambiente de execução também abstrata.

4.1.2 Mid-Level Intermediate Representation

- Operações específicas de ambiente de execução se tornam explícitas
- Opcodes continuam sendo independente de arquitetura

4.1.3 Low-Level Intermediate Language

- Esse nível do IR contém alguns sub-estados bastante significativos, são eles:
    - Antes de alocar nos registros
    - Depois de alocar nos registros
- Instruções devem casar com formato da arquitetura alvo
- Fluxo de controle deve ser explícito

4.1.4 Encoded Intermediate Language

- Formato para linkers
- Não recomendado para análise e manipulação de instruções

4.1.5 Operandos

Os operandos do IR são a chave da representação dos recursos no IR. No gráfico do IR, cada nó representa um operando, e como todos os efeitos das intruções são representados explicitamente, operando refletem todos os recursos utilizados.

A classe abstrata Phx.IR.Operand representa os operandos. As classes derivadas representam os diferentes recursos na representação IR do Phoenix. A tabela a seguir apresenta os operandos e uma breve descrição.

Operandos	Descrição
VariableOperand	• Representa parâmetros, variáveis comuns e temporárias, endereço de variáveis e registros. • Em HIR/MIR os VariableOperands são entidades lógicas, isto é, ainda não foi definido se serão armazendados em registros ou memória. • Na transição ao LIR, VariableOperands são examinados e alguns se tornam candidatos a ser armazenados em registros.
MemoryOperand	Representa acesso à memória ou endereço na memória
LabelOperand	Além de aparecer em branches e switches, LabelOperands também representam fluxo de exceções, portanto, podem aparecer em qualquer instrução.
AliasOperand	Representa efeitos colaterais de instruções em um registro ou memória ou outro estado externo.
FunctionOperand	Representa uma chamada direta a uma função ou um símbolo de uma função.

4.1.6 Instruções

O Phoenix armazena a IR de uma função em uma lista duplamente ligada de instruções. Cada instrução na IR expressa fluxo de dados, fluxo de controle ou uma operação. Instruções são classificadas com pseudo ou real. Cada instrução contém um opcode que especifica a operação, a lista de operandos fonte(operandos que serão lidos pela instução), e a lista de operandos destino(operandos que serão gravados pela instrução).

Importantes características dos operandos e instruções:

- Cada Operando é associado a exatamente uma instrução. Caso se anexe um operando a mais de uma instrução, ele será automaticamente copiado.
- A copia de um MemoryOperand copia qualquer operando de modo de endereço também.

4.1.6.1 Instruções Reais

As instruções reais são subdivididas da seguinte maneira:

- **Simple**: Instrução aritmética ou lógica que produz um valor.
- **Complex**: Uma chamada direta ou indireta a um procedimento.
- **Control flow**: Operações de fluxo de controle como desvio condicionais e incondicionais.
- **Summary**: Instruções para código removido do fluxo principal de instruções.

A maioria das instruções reais são mapeadas para uma ou mais instrução de máquina. A tabela a seguir descreve as instruções reais diponíveis no IR:

Instrução	**Descrição**
ValueInstruction	Uma operação lógica ou aritmética que produz um valor.
CallInstruction	Chamada de procedimento, direta ou indiretamente.
CompareInstruction	Uma instrução de comparação que gera um código de comparação.
BranchInstruction	Fluxo de controle tanto para desvios tanto condicional como incondicional, e para desvios diretos e indiretos.
SwitchInstruction	Fluxo de controle para um switch.
OutlineInstruction	Uma instrução para retirada de código do fluxo principal.

4.1.6.2 Pseudo Instruções

Pseudo instruções representam elementos que não alteram o comportamento do programa diretamente. Podem ser dos seguintes tipos:

Operandos	Descrição
LabelInstruction	Descreve labels e pontos de união para controle de fluxo e fluxo de código criado pelo usuário.
PragamaInsruction	Descreve diretivas fornecidas pelo usuário.
DataInstruction	Guarda informação alocada estaticamente.

4.1.7 Assemblies e Unit Structures

Um assembly é uma das unidades fundamentais na programação Phoenix. Como descrito anteriormente no capítulo do .NET Framework, um assembly agrupa módulos ou arquivos de recursos e é a menor unidade de reuso, segurança e versões. Uma unit structure encapsula uma coleção de estruturas de elementos de um programa, como fluxo de instruções, variáveis inicializadas, e funções. Todas as classes de unit structure derivam de Unit.

4.1.7.1 Hierarquia de Unidade(Unit)

O Phoenix Framework cria unidades de compilação à medida em que processa as fases. Essas unidades de compilação consistem em código e um conjunto de estrutura de dados relacionados. Pode-se então utilizar essas unidades para realizar análise do programa em diversos níveis.

A seguinte figura ilustra a hierarquia de unidade do Phoenix:

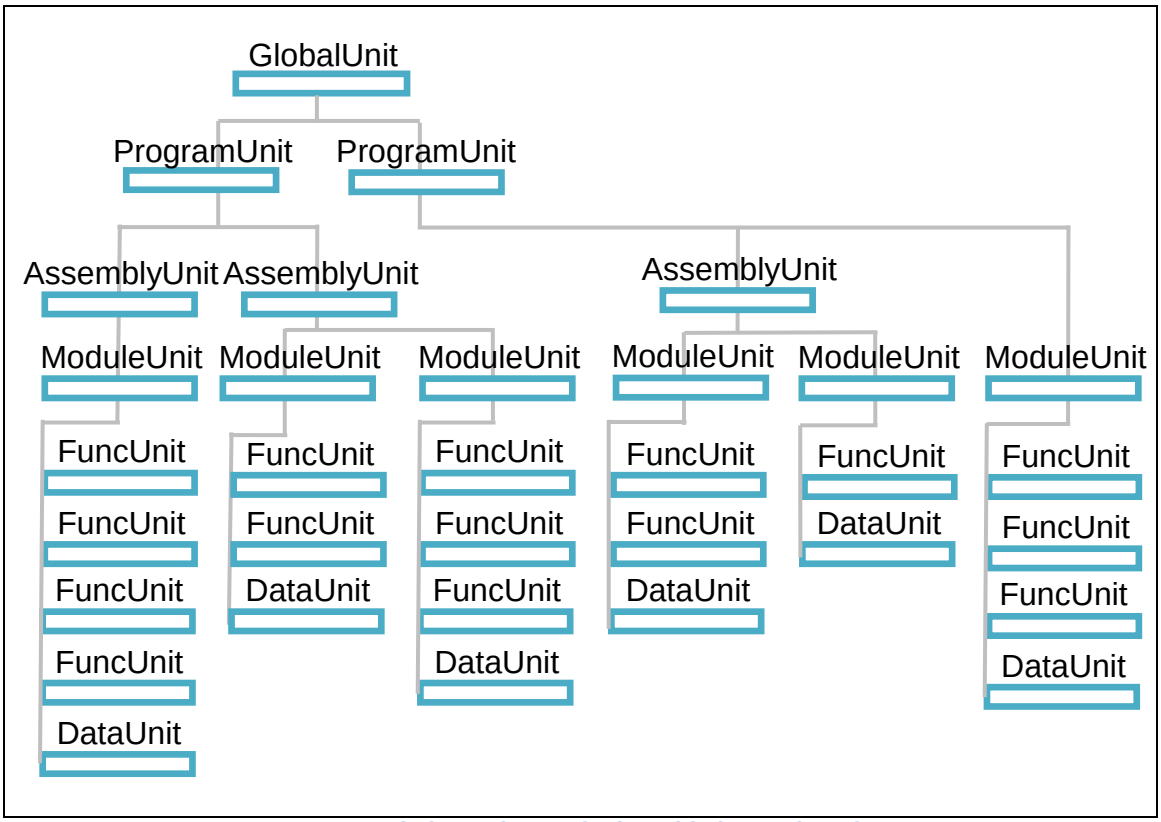

Tabela 8: Hierarquia de Unidade no Phoenix

GlobalUnit é a unidade de hierarquia mais alta. Um único GlobalUnit pode conter um ou mais objetos ProgramUnit. Cada objeto ProgramUnit, por sua vez, pode conter não só AssemblyUnit como também objetos ModuleUnit. Um ModuleUnit contém FunctionUnit e DataUnit. O FunctionUnit é a unidade fundamental na compilação e manipulação utilizada no Phoenix Framework.

Unidade (unit) do IR	Descrição
FunctionUnit	Contém o fluxo de instruções, tabela de símbolos, gráfico de fluxo, gráfico de região, e informações específicas de um método ou função.
DataUnit	Coleção de dados relacionados, como um conjunto de variáveis inicializadas.

ModuleUnit	Coleção de funções.
PEModuleUnit	O resultado de uma compilação que é uma imagem portable executable (PE), como um executável do Windows (EXE) ou uma biblioteca dinâmica(DLL).
AssemblyUnit	Uma compilação de uma unidade assembly do .NET Framework.
ProgramUnit	Uma imagem executável, isto é, um DLL ou EXE.
GlobalUnit	Unidade de compilação mais externa.

4.2 Symbol System

Os símbolos no Phoenix são associados com entidades como variáveis, labels, tipos, nome de funções, endereços, metada, e módulos importados/exportados. Coletivamente, símbolos provêem uma representação estrutural do relacionamento entre entidades dos programas, como o relacionamento entre uma função e certos argumentos e variáveis locais. Símbolos também são o mecanismo para referenciar essas entidades na IR (Phoenix Documentation, 2008).

Os símbolos do Phoenix se encontram no namespace Phx.Symbols e tem em comum a classe básica Symbol. Phoenix fornece uma variedade de símbolos que podem ser agrupados da seguinte maneira:

- **Basic Symbols**
- **PE Module Symbols**
- **Metadata Symbols**

A tabela a seguir descreve os símbolos básicos utilizados frequentemente durante o desenvolvimento com o Phoenix Framework.

Símbolo	Descrição
LocalVariableSymbol	Parâmetro, variável local, ou armazenamento temporário relacionado a um registro de ativação.
GlobalVariableSymbol	Armazenamento global de variável associado a um módulo executável.
NonLocalVariableSymbol	Referência Proxy a um GlobalVariableSymbol na representação IR de uma função.
ScopeSymbol	Um novo escopo de bloco.
FunctionSymbol	Ponto de entrada em um código executável.
ConstantSymbol	Constante
FieldSymbol	Campo nomeado em tipo de dados agregados.
StaticFieldSymbol	Campo estático em tipo de dados agregados.
LabelSymbol	Determinado ponto nomeado em código executável ou dados.
TypeSymbol	Tipo em uma entidade externa.
TypeDefinitionSymbol	Nome do tipo.
NamespaceSymbol	Determinação de escopo por namespace.
SectionSymbol	Nome de uma seção de um módulo executável.

Em termos de PE Module Symbols o Phoenix Framework apresenta:

Símbolo	Descrição
ImportSymbol	Variável ou ponto de entrada de código importado de módulo executável.
ImportModuleSymbol	Módulo externo que é a fonte das importacões.
ExportSymbol	Varável ou ponto de entrada de código de um módulo executável.

Em termos de Metadata Symbols o Phoenix provê:

Símbolo	Descriçào.
AssemblySymbol	Assembly do .NET Framework.
ResourceSymbol	Recurso gerenciado.
FileSymbol	Especificação de arquivo para recursos.
AttributeSymbol	Atributo customizado.
PermissionSymbol	Permissão.
PropertySymbol	Propriedade.
EventSymbol	Evento.

Os símbolos são agrupados dentro de tabelas Symbols.Table. Toda tabela de símbolo é associada a uma unidade(Unit). A hierarquia de unidades provê um mecanismo de escopo para os símbolos. Portanto, símbolos que são definidos no nível de módulo se encontram na tabela de símbolos ModuleUnit, e símbolos que existem no nível da função se encontram na tabela de símbolos FunctionUnit.

As tabelas de símbolos utilizam de mapas (maps) para descrever os símbolos que contêm. Tabelas não contêm um mecanismo de busca; os mapas provêem essa funcionalidade às tabelas. Toda tabela de símbolo tem pelo menos um objeto do tipo LocalIdMap. Pode-se utilizar esse mapa para buscar qualquer símbolo que ele contém utilizando a propriedade LocalId do símbolo, que é única na tabela. Pode-se ainda acrescentar mapas à tabela para se utilizar de características como endereço ou busca por nome.

4.2.1 Proxies

Proxy é uma forma especial de símbolo que permite o símbolo aparecer em mais de uma tabela de símbolos. Por exemplo, uma variável estática que é definida em uma função utiliza o Proxy para indicar que ela é tanto um membro do escopo da função quanto uma variável global.

Proxies são geralmente necessários quando se deseja referenciar um símbolo a partir de uma unidade que não seja a unidade real do símbolo, isto é, a unidade que mantém a tabela de símbolos em que o símbolo existe.

4.3 Sistema de tipos

O sistema de tipos no Phoenix provê um alicerce para flexibilidade e extensibilidade para importante funcionalidades como geração de código, otimização de alto-nível e depuração de código objeto. Além disso, para melhorar a robustez dos compiladores e ferramentas que utilizam a infra-estruturara do Phoenix, compiladores e ferramentas podem realizar checagem de tipos em código HIR e MIR, e checagem de consistência em LIR (Phoenix Documentation, 2008).

A sistema de tipos no Phoenix garante provê tipos e um meio de construir regras para checagem de tipos. Um compilador ou ferramenta pode criar um

conjunto customizado de tipos e provê um conjunto de regras customizadas para checagem de tipos. A especificação da semântica do sistema de tipos é independente de linguagem, mas dá suporte a propriedades customizadas e regras de checagem de tipos que são apropriadas para um ambiente específico.

O sistema de tipos também suporta mapeamento dos tipos do Phoenix para tipos da linguagem fonte original. O sistema de tipos pode expressar tanto tipos de alto nível quanto tipos de nível de máquina.

4.3.1 Classes do sistema de tipos

Há no Phoenix classes para armazenamento de informações dos tipos. A classe abstrata Type é a classe base para todos os tipos em Phoenix. Ela contém propriedades divididas por todos os tipos no Phoenix, como tamanho e informações sobre o campo. A classe tabela(Table)é usada para conter informações para todos os tipos. Por conveniência, a classe Table contém propriedades para obtenção de tipos primitivos freqüentemente utilizados, como inteiros e tipos de ponto flutuante.

Há na tabela a seguir algumas das classes de tipos frequentemente utilizados e que deriva de Type.

Classe	Descrição
PrimitiveType	O tipo mais simples que contém tipo e informação sobre o tamanho.
PackedType	Este tipo é utilizado em operações multimídia.
PointerType	Descreve tipos de referência, gerenciado, e nativo.
AggregateType	Descreve todos os tipos agregados

	exceto array. Tipos agregados são tipos que contêm membros.
EnumTypes	Descreve tipos de enumeração. (Enumeration)
TypeVariableType	Descreve variáveis de tipo em tipos genéricos.
ManagedArrayType	Descreve array gerenciado. Esta classe herda de AggregateType.
UnmanagedArrayType	Descreve tipos de arrays nativos.
FunctionType	Descreve tipos de função.

4.4 Plug-in

Um plug-in é um módulo externo executado por um compilador do Phoenix (como o C2.exe, o compilador back end para C++), ou a ferramenta de análise do Phoenix.

Suponha que um usuário criou um plugin chamado MeuPlugin.dll para o C2.exe. Com o comando –d2plugin, C2 irá carregar e executar o código contido dentro de MeuPlugin.dll como parte da compilação.

O código de MeuPlugin terá acesso total a estrutura interna de dados do C2. Como resultado, MeuPlugin pode modificar o comportamento de C2, adicionando fases, removendo fases ou substituindo fases. Por exemplo, MeuPlugin pode realizar as seguintes tarefas:

- Prover uma fase alternativa de alocação de registros que substitui a contida em C2.

- Inserir código extra em cada função compilada.
- Imprimir a representação intermediária (IR) da função enquanto ela é compilada. Pode-se verificar desta maneira como o IR evolui após otimizações.
- Imprimir informações sobre operações do C2 como uso de CPU ou memória.

A figura a seguir ilustra a operação de um plug-in no compilador C2:

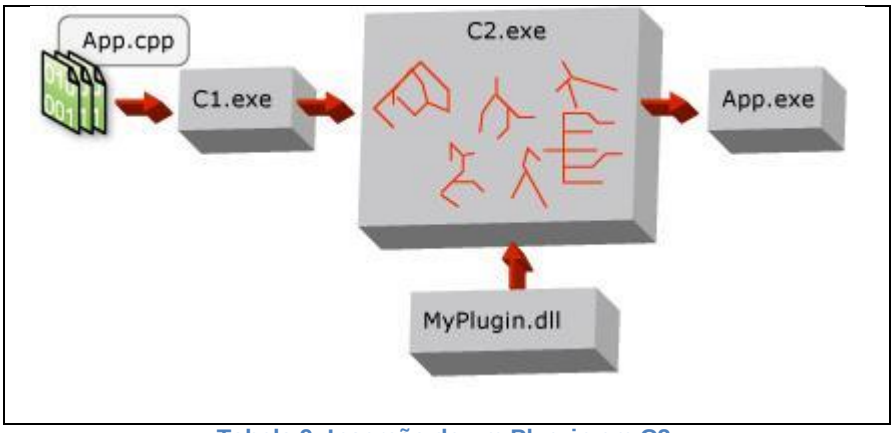

Tabela 9: Inserção de um Plug-in em C2.

Na esquerda da figura há uma coleção de arquivos .cpp que compõem a aplicação App.exe. Esses arquivos são compilados pelo front-end, C1.exe, e depois processados pelas fases do compilador back-end, C2.exe. Quando C2 é executado, ele dá acesso completo a todos os dados internos ao plug-in. Assim, o MeuPlugin terá acesso a todos os recursos necessários para modificar o comportamento de C2 podendo até afetar a geração de código do App.exe.

4.4.1 Responsabilidades de um Plug-in

Um plug-in deve ser implementado como uma assembly DLL gerenciado. Deve-se definir exatamente uma classe que implementa a classe Plugin. Essa classe deve conter um corpo para os seguintes métodos de interface: RegisterObjects e BuildPhases.

O Host chama o método RegisterObjects após a DLL do plugin é carregada no processo do Host. O método é utilizado para registrar os controles do plug-in. O Phoenix então reexamina a linha de comando fornecida na inicialização do Host, desta vez levando em consideração os controles passados para o plugin.

Após o Host ter criado suas próprias fases, ele chama o método BuildPhases. A chamada retorna um PhaseConfiguration, um objeto que auxilia o plug-in a inspecionar as fases do Host e toda a sua estrutura de dados, modificando-os de acordo com a especificação do plug-in. As modificações variam de acordo com o design do plug-in.

Para ser carregado pelo Host, ou qualquer outro compilador do Phoenix, o plug-in deve ser passado para o compilador. Isso é atingido através do opção de linha de comando **–plugin.** A exemplificação segue:

```
cl -nologo -d2plugin:FuncNames.dll funcs.cpp
```

Tabela 10: Passando um Plugin ao compilador.

4.4.2 Responsabilidades do Host

Para dar suporte aos plug-ins, o host precisa ser compilado utilizando o Phoenix Framework, deve-se incializar o Phoenix, chamando o método BeginInitialization e EndInitialization, e deve ser implementado como um assembly gerenciado. Todas as outras funcionalidades são fornecidas automaticamente pelo Phoenix.

4.5 Phases

Tipicamente, um processo de compilação consiste em diversas fases diferentes, onde cada fase realiza uma atividade diferente sobre o programa que compila. Uma fase pode, por exemplo, ser responsável por alocar registros, e

outra realiza inlining de funções. Deve-se construir uma lista de fases que o Phoenix irá executar quando o programa é processado.

Phoenix contém uma classe chamada Phase que pode ser derivada para construção de novas fases. Quando se escreve uma ferramenta utilizando o Phoenix, se define as fases e se adiciona a lista de fases. Quando a ferramenta processa uma unidade (Unit), o framework vai percorrer a lista de fases e executar as fases na unidade.

A infra-estruturara de fases no Phoenix consiste em quatro principais pedaços:

- PhaseConfiguration. Encapsulamento que contém um PhaseList e eventos que são levantados antes e depois de cada fase.
- PostPhaseEvent e PrePhaseEvent. Cada PhaseConfiguratioin mantém esses dois eventos associados, um que é levantado antes da execução e outro que é executado após a execução.
- PhaseList. Uma lista de fases derivada de Phase, e portanto uma PhaseList pode conter elementos que são objetos PhaseList. Como resultado, pode-se ter listas de listas. PhaseList também implementa o método Execute que chama a execução em cada objeto Phase contido na lista.
- Phase. A classe que se deriva para implementar uma nova fase. Em particular, na classe derivada deve-se sobrescrever o método Execute, que é o método que a infra-estruturara invoca.

Segue uma ilustração da infra-estrutura de fases:

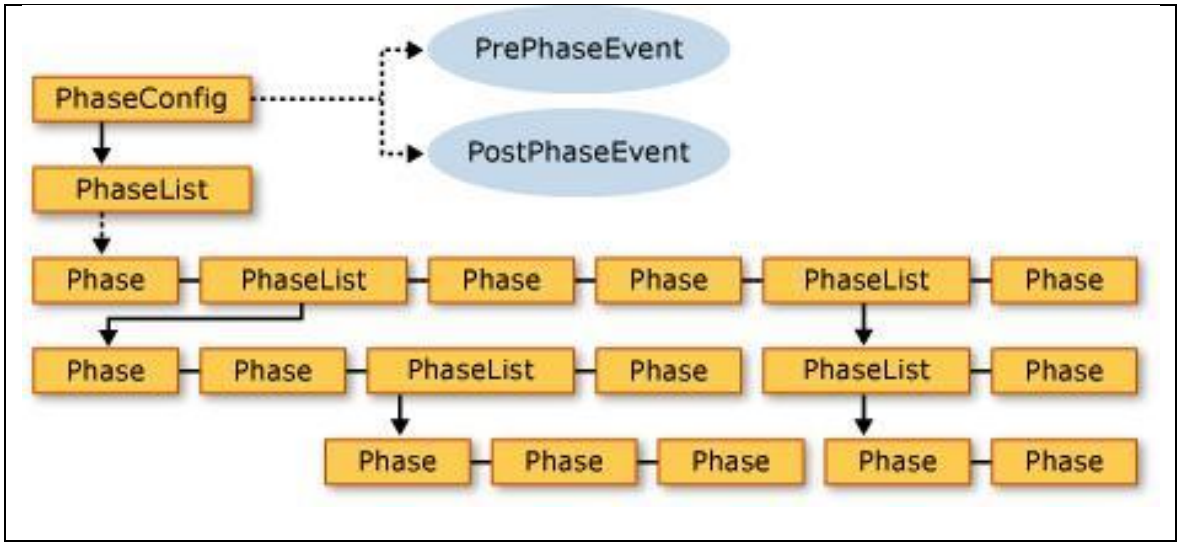

Tabela 11: Ilustração da infra-estrutura de fases.

Após a construção do objeto PhaseConfiguration, e todas as seus objetos Phase e PhaseList, pode-se executar as fases chamando o método DoPhaseList. Este método percorre a lista de fases e executa os sobre a unidade.

Uma fase é geralmente associada a um controle de componente (ComponentControl), que permite a habilitação ou desabilitação da fase a partir da linha de comando.

5. POTENCIAL DO PHOENIX COMO WEAVER

Como demonstrado no capítulo de Programação Orientada a Aspectos, a maioria das ferramentas que fazem a costura dos aspectos com o código, necessitam de uma modificação na sintaxe da linguagem original.

O Phoenix, no entanto, oferece uma alternativa: alterar o código binário diretamente e acrescentar os aspectos sem alteração da sintaxe da linguagem original. Esse potencial se torna interessante por não haver curva de aprendizado para os que utilizam a linguagem de programação. Eles continuarão escrevendo

seus códigos despreocupados com os aspectos identificados pelos arquitetos de software, e após a geração do código, o Phoenix entrará em cena e acrescentará as chamadas aos métodos de maneira a implementar os aspectos.

5.1 Plug-in AddCall.dll

O plug-in do Phoenix fornece um excelente ponto de partida para implementação de um Weaver para aspectos.

Este plug-in abre um arquivo binário nativo (não-gerenciado) e após ser executado, fornece um arquivo que contém uma chamada para um método local específico em uma DLL externa.

Com esse plug-in, um aspecto bastante comum nas aplicações do mundo real já se torna implementável: o aspecto de Logging. Pode-se acrescentar nas chamadas de método outro método que registra em um arquivo de log a sua utilização.

6. CONCLUSÃO

Embora o Phoenix não se encontre ainda em fase de comercialização, já se apresenta como uma ferramenta bastante útil para o desenvolvimento de ferramentas que auxiliam na modularização de um programa. Tornando-o mais reusável, e assim aumentando sua qualidade.

As características do Phoenix permitem que qualquer código que forneça um assembly, isto é, qualquer código do .NET Framework sejam modificados e acrescentados com advices. Isso amplia, em relação às implementações de linguagens orientada a aspectos, as possibilidades de reuso de código. Um mesmo aspecto pode ser utilizado em diversos módulos de código independente

de linguagem escrita. Percebe-se ainda a vantagem de poder modificar código de terceiros sem necessariamente ter acesso ao código fonte, visto que é possível editar um arquivo binário.

Portanto, a construção de uma implentação AOP no Phoenix não é uma evolução em termos de paradigmas de linguagens, mas também uma revolução em termos de funcionalidades de um compilador. O Phoenix tira o compilador de uma caixa preta e o fornece ao desenvolvedor para geração de código de melhor qualidade em termos de reuso e desempenho.

7. REFERÊNCIAS

Box, D., e Sells, C. “Essential .NET: The Common Language.” Vol. 1. Addison Wesley. 2002.

Eaddy, M., Aho, A, e Murphy, G. “Identifying, Assigning, and Quantifying Crosscutting concerns.” ICSE Workshops. Nova Iorque. 2007.

ESTADOS UNIDOS. Phoenix Documentation. Edição de Março/2008. Redmond: Microsoft Corporation. 2008.

Figueiredo, E., Garcia, A., Sant'Anna, C., Kulesza, U., e Lucena, C. "Assessing Aspect-Oriented Artifacts: Towards a Tool-Supported Quantitative Method," Workshop on Quantitative Approaches in OO Software Engineering, 2005.

Kiczales, G., Irwin, J., Lamping, J., Loingtier, J.-M., Lopes, C. V., Maeda, C., e Mendhekar, A. "Aspect-oriented programming," ACM Computing Surveys, 28(4es):154, 1996.

Richter, J. “Applied Microsoft .NET Programming”. 1ª. edição. Redmond: Microsoft Press. 2002.

Soares, S.,.Borba, P., e Laureano, E. “Distribution and Persistence as Aspects.” Software Practice Experience. 2000; 00:1–6.

SUÍÇA. “Common Language Infrastructure.” 4ª. edição. Genebra: ECMA International. 2006.