ICIE Y2K

VI INTERNATIONAL CONGRESS ON INFORMATION ENGINEERING

APRIL, 26-28, 2000

Título do Artigo : **Aspectos do Projeto e Implementação de Ambientes Multiparadigmas de Programação**

Autores: **Aparecido Valdemir de FREITAS e João José NETO**

Escola Politécnica da Universidade de São Paulo
Depto. de Engenharia de Computação e Sistemas Digitais
Av. Prof. Luciano Gualberto, trav. 3, No. 158. Cidade Universitária
São Paulo – Brasil
e-mail: **avfreitas@imes.edu.br** e **joao.jose@poli.usp.br**

Área de Aplicação: **Computational Languages**

João José NETO

Formado em 1971 na Escola Politécnica da USP em Engenharia de Eletricidade, modalidade Eletrônica. Mestre em Eng. Elétrica pela EPUSP em 1975. Doutor em Eng. Elétrica pela EPUSP em 1980. Livre-Docente pela EPUSP em 1993. Ocupa atualmente o cargo de Professor Associado junto ao Depto. de Eng. de Computação e Sistemas Digitais da EPUSP. Ministra disciplinas de graduação e pós-graduação na área de Sistemas Operacionais, Eng. de Software, Linguagens de Programação, Compiladores e Teoria da Computação. Desenvolve junto à EPUSP a linha de pesquisa em Tecnologias Adaptativas, através da orientação de programas de Mestrado e Doutorado na área.

Aparecido Valdemir de FREITAS

Mestrando em Eng. de Computação – Sistemas Digitais pela EPUSP. Especialização em Engenharia de Computação – Ênfase Programação pela EPUSP-FDTE em 1986. Bacharel em Matemática Plena pela F.S.A. em 1974. Formado em Engenharia Civil pela E.E.Mauá em 1979. Ocupa atualmente o cargo de Professor-I no curso de Ciência da Computação do IMES – Instituto Municipal de Ensino Superior de São Caetano do Sul, onde ministra as disciplinas Sistemas Operacionais-I e Técnicas de Programação.

Aspectos do Projeto e Implementação de Ambientes Multiparadigmas de Programação

Aparecido Valdemir de Freitas e João José Neto

Escola Politécnica da Universidade de São Paulo
Depto. de Engenharia de Computação e Sistemas Digitais
Av. Prof. Luciano Gualberto, trav. 3, No. 158. Cidade Universitária
São Paulo – Brasil
e-mail: **avfreitas@imes.edu.br** e **joao.jose@poli.usp.br**
Palavras-chave: paradigma, multilinguagem, multiparadigma, ambiente, composição.

Abstract

The development of large-scale complex software is often made difficult by the lack of tools allowing the programmer to express his or her ideas in a neat way for all technical aspects involved. This is particularly the case when we deal with the expressive power of programming languages. Since large programs are usually composed by segments of different natures, it seems natural to provide specifical tools for the programmers in order to achieve good expressiveness in the corresponding code. Multiparadigm and multilanguage approaches are discussed here as alternative ways to satisfy this need, and the implementation of a multilanguage (multiparadigm) development system is proposed and discussed. The paper presents a significant bibliography on this subject to orient readers interested in both in design and philosophical issues on this approach.

Resumo

O desenvolvimento de software complexo de grande porte é muitas vezes dificultado pela carência de ferramentas adequadas para a clara expressão das idéias dos programadores em todos os aspectos técnicos do projeto. Isto é particularmente verdadeiro quando se lida com o poder de expressão de linguagens de programação. Como os grandes programas se compõem usualmente de segmentos com características técnicas diversificadas, parece natural disponibilizar ferramentas específicas para os programadores, de forma que uma boa expressividade seja obtida no código correspondente. As técnicas multiparadigma e multilinguagem são discutidas como formas alternativas de satisfazer essas necessidades, e a implementação de um ambiente de desenvolvimento multilinguagem (multiparadigma) é proposta e discutida em seus aspectos diversos. O artigo finaliza com uma bibliografia significativa sobre o tema, orientando os leitores interessados tanto nos aspectos de projeto como nos de caráter filosófico da técnica proposta.

1. Motivação

Quantas vezes, ao iniciarmos um trabalho, fomos impedidos de prosseguí-lo por falta de uma ferramenta particular? Conforme [Hailpern-86a], a falta de uma ferramenta adequada é muito freqüente na área de desenvolvimento de software. Ao escrevermos códigos, muitas vezes a ferramenta ideal pode não estar disponível, pode ser cara, ou ainda, pode não existir. Portanto, muitas vezes somos obrigados a codificar a despeito das limitações da linguagem de programação disponível.

É possível que, no desenvolvimento de uma aplicação, tenhamos que manipular diferentes problemas de programação. Tendo em vista que cada um destes problemas poderia ser mais facilmente resolvido com a utilização de um determinado paradigma de programação, idealmente poderíamos particionar o problema em subproblemas, cada qual associado ao paradigma mais apropriado.

Conforme [Spinellis-94], o termo paradigma é comumente utilizado para se referir a um conjunto de entidades que compartilham características comuns.

A programação multiparadigma tem, como meta, permitir que o programador implemente o código através do uso dos mais adequados paradigmas ao problema em questão.

A programação multiparadigma pode ser implementada através de linguagens que combinam mais de um paradigma de programação. Neste caso, podemos encontrar na literatura um número relativamente expressivo de linguagens multiparadigmas. No entanto, a maior parte destas combina um conjunto limitado de paradigmas de programação, conforme [Spinellis-94].

Uma outra forma de implementarmos a programação multiparadigma seria através da disponibilização de ambientes que combinariam códigos gerados por compiladores de paradigmas isolados. Nesta alternativa, os programadores desenvolveriam os módulos da aplicação em paradigmas independentes e um ambiente multiparadigma os auxiliaria como interface entre os módulos correspondentes.

A área da programação multiparadigma é relativamente nova, embora várias pesquisas mais atreladas ao projeto de linguagens multiparadigmas tenham sido desenvolvidas, conforme relatado em [Hailpern-86b].

Este artigo tem, como objetivo, especificar e apresentar uma proposta de implementação de um ambiente de programação que viabilize o emprego da programação multiparadigma.

2. Paradigmas

O uso da expressão "*novo paradigma*" tem crescido de forma muito intensa através dos anos 90, e como decorrência disto, muitos trabalhos de pesquisa têm usado o termo em seus títulos e *abstracts*.

Um significado conceitual do termo foi apresentado por [Kuhn–62], o qual utilizou o vocábulo para descrever teorias e procedimentos que, quando utilizados de forma conjunta, podem representar uma forma de organizar o conhecimento. A tese de *Kuhn* está sedimentada na idéia de que uma revolução na ciência ocorre apenas quando um paradigma antigo foi reexaminado, rejeitado e substituído por um novo paradigma, conforme[Budd-95].

Segundo *Kuhn*, paradigmas são essencialmente teorias científicas ou formas de observar o mundo que atendem a dois requisitos: devem ser "suficientemente sem precedentes" para atrair um grupo de seguidores ou pesquisadores e devem ser "suficientemente abertos" para permitir o surgimento de novas teorias que possam resolver problemas decorrentes ou correlatos. Um paradigma numa área de conhecimento pode ser retratado como um dogma ou crença em função da fidelidade com que seus seguidores o defendem. Trata-se, portanto, de algo mais do que, simplesmente, um conjunto de teorias ou modelos acerca de um problema.

Ainda segundo [Kuhn-62], cada um de nós visualiza o mundo através de um conjunto de parâmetros, ou padrões, que utilizamos para organizar nossa forma de pensar. Com esta interpretação, o termo paradigma está associado à percepção de como visualizamos uma determinada entidade, estendendo o conceito além do sentido físico. Estes paradigmas nos fornecem a base para determinarmos aquilo que acreditamos ou não.

[Bryan-99] relata um experimento em que, num jogo de cartas, as copas são pintadas de preto e as espadas de vermelho. Quando as cartas são exibidas aos participantes do jogo, estes incorretamente identificam as copas como espadas e as espadas como copas. Neste caso, podemos observar que suas mentes não estão em acordo com o novo paradigma das cartas, a eles apresentado.

Ainda segundo [Bryan-99], paradigmas podem bloquear nossa visão do futuro e afetar nossa capacidade de resolver problemas. Eles definem muito do que pensamos e do que fazemos. Há uma tendência natural em aceitarmos os paradigmas que condizem com nossa forma de pensar e, conseqüentemente, rejeitarmos aqueles que são novos, diferentes ou que de alguma forma conflitem com nossa forma de pensar. Quanto mais estivermos arraigados a uma dada forma de pensar, mais resistências ofereceremos a quaisquer evidências ou argumentos contrários.

Porque às vezes é tão difícil percebermos fatos que após a constatação dos mesmos nos parecem tão óbvios? Tal como ilusões de ótica, os paradigmas impedem que determinadas percepções nos sejam reveladas, uma vez que nossa forma de pensar pode estar muito vinculada à visualização de apenas algum particular aspecto do problema ou fato.

Dentre as várias formas de exteriorizar um paradigma ou forma de pensar, talvez uma das mais importantes na expressão de um paradigma seja a linguagem.

Conforme citado em [Budd-95], lingüistas têm enfocado que a linguagem na qual um pensamento é expresso reflete de forma fundamental a natureza de uma idéia. A associação entre pensamento e linguagem se torna ainda mais crítica se estendermos o conceito para o domínio das linguagens de programação. Assim, ainda conforme [Budd-95], a linguagem utilizada por um programador na resolução de um problema está fortemente relacionada à sua maneira de pensar, ou à sua forma de implementar a solução do problema.

De acordo com [Budd-95], a aceitação da *Tese de Church* tem uma importante e significativa implicação no estudo das linguagens de programação. *Máquinas de Turing* são mecanismos extremamente simples, e não requerem muitas construções de linguagens para simulá-las. Assim, com uma linguagem de programação que possua ao menos construções condicionais e de desvios, é possível simularmos uma *Máquina de Turing* qualquer. Por outro lado, para resolvermos um problema, deveremos encontrar uma *Máquina de Turing* que produza o resultado desejado, o qual, pela *Tese de Church,* deve existir. Em seguida, escolhemos a nossa linguagem favorita para simularmos a execução da *Máquina de Turing*.

Por que então há tantas linguagens de programação disponíveis? De acordo com [Budd-95], a experiência tem mostrado que a característica mais importante, e também a mais difícil de ser mensurada, é a facilidade de uso de uma linguagem. Certamente, poucos programadores se habilitariam a desenvolver programas de computadores caso o único mecanismo disponível fosse a *Máquina de Turing*. Da mesma forma, os programadores de hoje oferecem muitas restrições à escrita de programas em linguagem *assembly*. Embora não se questione a potencialidade de

execução de tarefas computacionais da linguagem *assembly* e da *Máquina de Turing*, a questão é de ordem operacional, pois estas são difíceis de serem utilizadas, difíceis de serem depuradas e requerem longos tempos de construção de programas, trazendo, como conseqüência, baixa produtividade.

3. Paradigmas e Linguagens de Programação

[Placer-91] cita que quando o escopo descrito por um dado paradigma contém os itens e relacionamentos que existem no domínio de interesse do problema, a tarefa de modelar uma solução dentro daquele domínio fica facilitada.

Por exemplo, o paradigma lógico tende a materializar a solução de um problema através da composição de predicados e relações, enquanto o paradigma funcional enfoca o uso e composição de funções. Se estivermos de posse de uma linguagem de programação na qual o modelo de *arrays* possa ser diretamente representado, teremos maior facilidade na resolução de problemas envolvendo aritmética de matrizes.

Por outro lado, caso a linguagem de programação a ser utilizada não abstraia diretamente as entidades do domínio do problema, (no caso matrizes), a solução deverá ser implementada através da simulação de tais entidades com a ajuda dos elementos disponíveis na linguagem. Esta tarefa dificulta o processo de implementação e diminui a expressividade da solução, uma vez que a forma de pensar do programador não estará diretamente mapeada na linguagem de implementação.

Com estas reflexões, podemos inferir que as linguagens de programação, nas quais as soluções dos problemas são escritas, direcionam a mente do programador, de forma tal que este trate o problema de acordo com uma determinada visão, imposta pelo tipo de linguagem.

É comum nos depararmos com programadores que trabalham há muitos anos com uma determinada linguagem de programação, e cujo paradigma está tão fortemente arraigado à sua forma de pensar, que estes quase sempre modelam suas soluções a partir das construções disponíveis no paradigma da linguagem. Tais programadores, quase sempre, têm dificuldades em quebrar o paradigma usual e empregar outros paradigmas de programação.

4. Paradigmas de Programação

Comentam-se a seguir seis paradigmas de programação, que constituem um conjunto significativo de paradigmas observado nas linguagens de programação usuais: os paradigmas imperativo, orientado a objetos, funcional, lógico, concorrente e adaptativo.

Paradigma Imperativo. Dentre os mais conhecidos paradigmas, a programação imperativa (estiloVon-Neumann) é a mais largamente utilizada. O vocábulo *imperativo* é de origem latina (*imperare*) o qual significa comandar, ordenar.

O paradigma imperativo tem uma história relativamente longa, uma vez que os primeiros projetistas das linguagens que o compõem, idealizaram o modelo de forma que variáveis e comandos de atribuição constituíssem uma simples, mas útil, abstração de consultas e atualizações à memória, através de seqüências de instruções de máquina.

Conforme citado em [Watt-90], em função do estreito relacionamento com arquiteturas de máquina, pelo menos em tese, as linguagens imperativas podem ser implementadas de forma muito eficiente.

Conforme [Budd-95], o paradigma imperativo é usualmente visto como o modelo *tradicional* de computação. A computação é vista como uma tarefa executada por uma *unidade de processamento* que modifica e manipula a *memória*, esta uma seqüência de posições ou células que armazenam valores. Cada célula é denotada por um número ou por um ou mais nomes simbólicos, e os valores nela contido podem ser modificados em tempo de execução.

Um programa em execução passa por uma seqüência de estados. A transição de um estado para o próximo é determinada por operações de atribuição e de seqüenciamento.

Neste modelo, o computador é um manipulador de dados segundo um padrão de instruções. Cada instrução extrai valores da memória, transforma-os, e em seguida armazena os resultados em posições de memória. Quando a computação terminar, os valores armazenados na memória deverão conter a solução desejada.

Assim sendo, o modelo imperativo é caracterizado por uma seqüência de mudanças de estado no qual um nome pode ser associado a um valor num determinado ponto do programa, podendo receber posteriormente um valor diferente. Tendo em vista que a ordem de execução destas atribuições afeta o valor final das expressões, uma importante característica do modelo é a sua sensibilidade à seqüência em que estas atribuições são efetuadas.

Paradigma OOP. O paradigma orientado a objetos tem como base a definição de estruturas que possam ser reutilizadas. De acordo com [Budd-95], este conceito está associado à noção de *design recursivo*, o qual visualiza um computador como sendo uma estrutura formada por diversas unidades computacionais integradas (chamadas objetos) compostas, como o próprio computador, de unidade de processamento e memória.

Conforme citado em [Sethi-96] a programação orientada a objetos tem, como meta, tornar mais fácil a a tarefa de estruturar, gerenciar e construir aplicações complexas.

Os objetos são descritos através de *classes*, que são compostas por *métodos* (ações) e *estados* (valores de dados). Dessa forma, a base para o suporte à programação orientada a objetos são as *classes*, usualmente organizadas em *hierarquias* e *instâncias*.

Ao invés de escrever procedimentos que manipulam estruturas de dados, as operações e os dados são vistos como um todo orgânico, uma unidade que provê um serviço para outras porções da aplicação.

Conforme citado em [Budd-95], a programação orientada a objetos se baseia na utilização de unidades autônomas que interagem com o objetivo de resolver um problema. Portanto, diferentemente da programação imperativa, a aplicação dos conceitos associados à orientação a objetos tendem a ser mais naturais na resolução de problemas.

Ainda de acordo com [Budd-95], a programação orientada a objetos é usualmente vista como uma extensão da programação imperativa, onde a computação também é desenvolvida promovendo-se mudanças de estado nos objetos.

Paradigma Funcional. Freqüentemente pensamos num programa em execução como sendo uma forma de implementar um relacionamento, no qual determinados valores de entrada são

mapeados para valores de saída. Na programação imperativa, este mapeamento é efetuado de forma indireta, através de comandos que lêem valores de entrada, manipulam tais valores, e escrevem os resultados de saída. Na programação funcional, o citado mapeamento é atingido de forma mais direta, uma vez que o programa é uma função, tipicamente composta por funções mais simples. Os relacionamentos entre as funções permitem que o resultado de uma função possa ser usado como argumento para uma outra função. Os programas funcionais são basicamente escritos em uma linguagem definida por expressões, funções e declarações.

Conforme [Budd-95], na programação funcional, valores são tratados como entidades simples, não como "*células com val*ores". Valores podem ser criados, mas uma vez criados não mais são modificados. Em lugar de se fazer pequenas mudanças incrementais às estruturas existentes, estes valores são transformados em novos valores, independentes dos valores originais. Por exemplo, da adição de um valor a uma lista resulta uma nova lista, e não uma alteração da lista original.

Ainda de acordo com [Budd-95], no paradigma funcional, as variáveis não correspondem a posições fixas de memória com tamanhos definidos, mas são simplesmente artifícios lingüísticos através dos quais os valores podem ser referenciados. Esta visão é quase a antítese da programação imperativa, uma vez que os valores não têm um "estado" para ser modificado com o decorrer do tempo. Na verdade, a noção de "tempo" tem diferentes conotações quando tratada na programação funcional e na programação imperativa. Em outras palavras, a restrição de que uma função somente possa ser aplicada quando seus argumentos estiverem disponíveis, faz com que as funções possam ser executadas em qualquer seqüência. Se uma função produz um certo resultado quando ativada com um dado conjunto de argumentos, este resultado não se altera, independentemente da época ou da freqüência com que a função é executada.

Paradigma Lógico. De acordo com [Budd-95], a programação lógica recebeu um grande impulso a partir do trabalho da comunidade de inteligência artificial, e ainda é usualmente descrita por analogia à prova de teoremas.

Conforme [Watt-90], o paradigma lógico está associado à noção de que um programa implementa uma relação, e está fundamentado no cálculo de predicados. Assim, um programa lógico é um conjunto de *cláusulas de Horn*, conforme relatado em [Sebesta-96].

De acordo com [Budd-95], a proposição clássica da prova de teoremas contempla três partes: um conjunto de axiomas (que são assumidos como verdade ou, pelo menos, aceitos como verdades para os propósitos do teorema), um conjunto de regras de inferência (pelas quais novas informações podem ser derivadas de fatos já conhecidos), e uma questão ou consulta (*query*). A prova consiste de um conjunto de produções que se inicia a partir dos fatos assumidos, (usando-se regras de inferência) e se encerra mostrando que a questão pode, ou não, ser derivada a partir da informação fornecida.

O que é mais importante no processo de programação lógica é que a programação é declarativa, ou seja, não-procedimental. De acordo com [Mellish-94], o programador simplesmente apresenta um conjunto de cláusulas (fatos ou regras) e um conjunto de metas a serem atingidas. Estas cláusulas conhecidas serão utilizadas para satisfazer as metas. Se uma meta não puder ser atingida através de uma seqüência de cláusulas, inicia-se um mecanismo conhecido por *backtracking*, no qual se revisará o que foi feito e se tentará novamente satisfazer as metas através de cláusulas alternativas ainda não consideradas.

De acordo com [Budd-95], programas puramente lógicos são aplicáveis a um domínio limitado de problemas, e assim, quase todas as linguagens lógicas de programação são multiparadigmas, uma vez que incorporam técnicas de programação de outros paradigmas.

Paradigma Concorrente. Programas seqüenciais especificam a execução seqüencial de uma lista de comandos cuja execução é denominada processo. Um processo pode ser considerado como uma seqüência de operações, executadas uma de cada vez.

De acordo com [Santos-84], uma boa parte dos problemas computacionais exige uma ordenação apenas parcial de suas operações no tempo, ou seja, não há necessidade do estabelecimento de uma ordenação de todos os passos da computação. Assim, alguns programas exigem que determinadas operações sejam feitas antes de outras, enquanto que outras operações podem ser feitas paralelamente.

Operações que devem ser executadas numa ordem seqüencial estrita irão constituir processos. Um conjunto de processos executando paralelamente e com algum mecanismo de intercomunicação entre os mesmos, irá caracterizar o que denominamos programa concorrente.

Programas concorrentes podem ser executados desde através do uso compartilhado de um único processador entre os vários processos, até através do uso dedicado de um processador para cada processo.

Para evitarmos que a concorrência de processos produza anomalias de execução, torna-se necessário empregar mecanismos de comunicação e sincronização (por exemplo, semáforos), os quais servem não apenas como formas de estruturação de programas concorrentes, mas também como instrumentos para introduzir funcionalidade nesses tipos de programas.

Paradigma Adaptativo. De acordo com [Neto-87], as linguagens *livres* de contexto podem ser reconhecidas pelos autômatos de pilha estruturados, os quais podem ser vistos como conjuntos de sub-máquinas, que operam de forma semelhante aos autômatos finitos, com o auxílio de uma pilha de estados. Na operação do autômato, ao ocorrer uma chamada de transição para sub-máquina, antes de se desviar para o estado inicial da sub-máquina chamada e, por falta de possibilidade de continuar a executar transições internas, empilha-se uma indicação do estado para onde o retorno deverá ser efetuado ao final da operação da sub-máquina chamada.

Conforme [Neto-94], a descrição das linguagens dependentes de contexto pode ser feita através de dispositivos de reconhecimento, chamados autômatos adaptativos, cuja operação corresponde a uma seqüência de reconhecedores intermediários, de sucessivas *subcadeias* do texto de entrada, as quais são tratadas por correspondentes autômatos finitos ou de pilha.

Conforme [Pereira-99], as dependências de contexto da linguagem reconhecida pelos autômatos adaptativos são tratadas através de transições adaptativas. À medida em que estas transições adaptativas forem sendo executadas, informações relativas à cadeia de entrada (novos estados ou transições) são incorporadas à própria estrutura do autômato adaptativo.

A implementação das transições adaptativas é feita através da associação, às transições usuais do autômato, de ações adaptativas, as quais são chamadas de funções construídas a partir de ações adaptativas elementares, que podem determinar a consulta ao conjunto de transições existentes, ou a inclusão de novas transições, ou, ainda, a eliminação de transições existentes.

Um autômato adaptativo, portanto, corresponde a uma seqüência de evoluções sucessivas de um autômato de pilha estruturado inicial, processadas como fruto da execução de ações adaptativas. A cada ação adaptativa, uma nova topologia é obtida para o autômato, o qual dará continuidade ao tratamento da cadeia de entrada.

Esta característica de auto-modificação dos autômatos adaptativos nos possibilita propor um novo paradigma de construção de software denominado paradigma adaptativo, o qual tem como meta suportar o desenvolvimento de software incremental ou evolucionário, ou extensível.

5. Composição de Paradigmas

Um paradigma de programação pode ser considerado como a base para a caracterização de uma classe de linguagens de programação, ou ainda como uma forma de pensar acerca do estilo empregado na solução de problemas, conforme descrito em [Zave-89].

Se um paradigma é uma forma de organizar idéias ou formas de pensar, e se as várias linguagens disponíveis nos habilitam a exprimir estas idéias, qual vantagem há em empregarmos uma linguagem que suporte diversos paradigmas? Isto pode, a princípio, caracterizar uma contradição.

No entanto, conforme [Zave-96], a tarefa de resolver problemas com a utilização de um único paradigma de programação se torna extremamente complexa quando estivermos trabalhando com sistemas de grande tamanho e complexidade. Assim, tais sistemas podem ter aspectos heterogêneos que dificultam a utilização de um único paradigma ou estilo de programação. Conforme [Zave-89], um dado paradigma apresenta uma visão nem sempre suficientemente abrangente, para permitir a descrição de todos os aspectos de um grande e complexo sistema.

Analogamente, de acordo com [Zave-89], os engenheiros têm pesquisado novas matérias-primas para o desenvolvimento de produtos. Nenhuma montadora de veículos tentaria persuadir seus clientes a adquirir um carro que fosse inteiramente construído de vidro, aço, ou bronze. Muitos diferentes materiais são necessários, e muitas diferentes operações também são necessárias para manuseá-los e compô-los.

Como engenheiros de software, necessitamos de um repertório similar de operações para manipular e compor as muitas linguagens de programação que deveriam ser utilizadas para diferentes aplicações.

De acordo com [Spinellis-94], cada paradigma oferece um conjunto diferente de características que devem ser consideradas ao se avaliar os diversos aspectos de implementação de uma aplicação.

A multiplicidade de soluções oferecidas pelos diversos paradigmas de programação reflete a diversidade de técnicas que podem ser aplicadas na resolução de problemas, conforme [Budd-95].

A composição de paradigmas provê ao programador a habilidade para selecionar o enfoque mais apropriado ao problema em questão, permitindo assim, a escolha da solução mais expressiva e elegante.

Conforme relatado em [Spinellis-94], poderemos viabilizar a implementação das aplicações multiparadigmas através dos seguintes esquemas:

- Escrever a aplicação em uma única linguagem multiparadigma, que ofereça ao usuário o acesso aos recursos de todos os paradigmas necessários.
- Utilizar diversas linguagens isoladas como representantes de paradigmas simples, e integrar os programas assim desenvolvidos através de um *linkeditor* multiparadigma, encarregado da resolução de referências entre os módulos componentes assim obtidos.
- Implementar a aplicação através de processos do sistema operacional, representados por módulos executáveis que se comunicam através de recursos do sistema operacional, tais como, chamadas de API's (*Application Program Interface*) em ambientes Win32, para a intercomunicação dos mesmos.
- Empregar o método das transformações lingüísticas nas linguagens componentes da aplicação, até se atingir uma linguagem-alvo que será processada através de um compilador único, conforme descrito em [Spinellis-94].

6. Especificação de Requisitos para um Ambiente Multiparadigma

Os requisitos abaixo descritos foram levantados por [Spinellis-94] e apresentam algumas diretrizes para a especificação do projeto de um ambiente multiparadigma de programação.

- Acomodar os diversos modelos e mecanismos de execução das várias linguagens, tais como, interpretadores de máquinas abstratas, compilação ou interpretadores.
- Acomodar as diferentes notações sintáticas das várias linguagens, de forma a assegurar que o programador irá empregar os mesmos construtos usualmente codificados. O ambiente não deverá adicionar qualquer restrição sintática nas linguagens componentes. Assim, o programador não necessitará aprender novas linguagens relacionadas ao ambiente.
- Suportar o problema de transferência de controle, e transferência de dados entre as linguagens pertencentes à aplicação.
- Gerenciar os recursos empregados pelos diversos mecanismos de execução presentes numa aplicação multiparadigma (multilinguagem), tais como, memória, espaço de nomes, etc.

7. Arquitetura do Ambiente Multiparadigma proposto

Em nossa proposta, a aplicação multiparadigma é formada por um conjunto de processos que se comunicam, cada qual utilizando uma diferente linguagem de programação. Estas diferentes linguagens poderão ou não pertencer a um mesmo paradigma de programação. O ambiente proposto provê algumas funções com o objetivo de facilitar a integração dos módulos componentes, as quais são obtidas através de chamadas do sistema operacional, para assegurar a efetiva troca de mensagens entre os módulos, de acordo com a Figura-1.

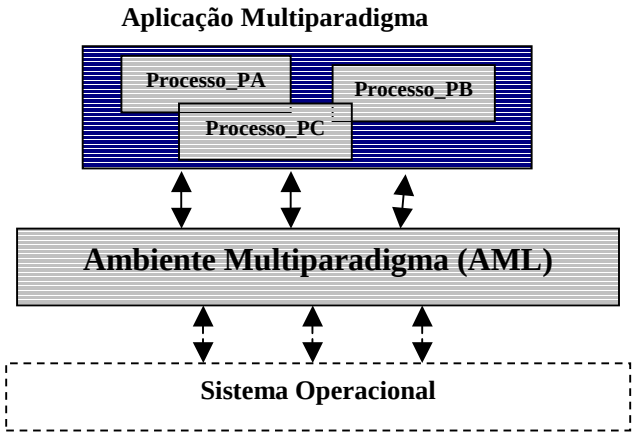

Figura-1 Arquitetura do Ambiente Multiparadigma

Para facilitar a apresentação da arquitetura proposta, utilizaremos a sigla AML para denotar o nosso ambiente multiparadigma de programação. Utilizaremos o ambiente proposto sob a plataforma *Win32*, mas as idéias são aplicáveis a outras plataformas, bastando que sejam substituídas as chamadas de API's para o sistema operacional desejado.

Para implementarmos um adequado esquema, que permita a interoperação dos paradigmas componentes, deveremos tratar os problemas de transferência de dados e de controle entre os processos que irão compor a aplicação.

Com o objetivo de projetarmos as primitivas de interoperabilidade, vamos idealizar três processos denotados por P_A , P_B e P_C. Admitiremos que P_A represente o processo principal da aplicação.

Para que a interoperabilidade entre os processos P_A , P_B e P_C aconteça, são projedadas algumas primitivas do ambiente responsáveis pelo atendimento aos serviços de interação entre os processos componentes.

Descreveremos, a seguir, alguns esquemas possíveis de interoperabilidade de processos, levando em conta os aspectos de transferência de dados e de controle.

7.1 Interação de Paradigmas sem dependência de dados.

Para esquematizarmos esta alternativa, admitiremos que o processo P_A necessite chamar o processo P_B. Para uniformizar este procedimento, o ambiente deve fornecer a primitiva AMLCALL que irá implementar a chamada entre processos. Tendo em vista que cada paradigma corresponde a um processo, cabe ao ambiente criar o processo a ser chamado, e, em seguida, providenciar a transferência de controle.

De acordo com [Zave-89], o mecanismo de transferência de controle proposto se refere a uma das formas bem conhecidas e aplicáveis de composição de módulos, uma vez que a maioria das linguagens de programação oferecem mecanismos de chamada (*calls*). A figura-2 abaixo ilustra o esquema de transferência de controle proposto. A eventual comunicação neste caso envolve apenas a transferência de dados através de arquivos fechados.

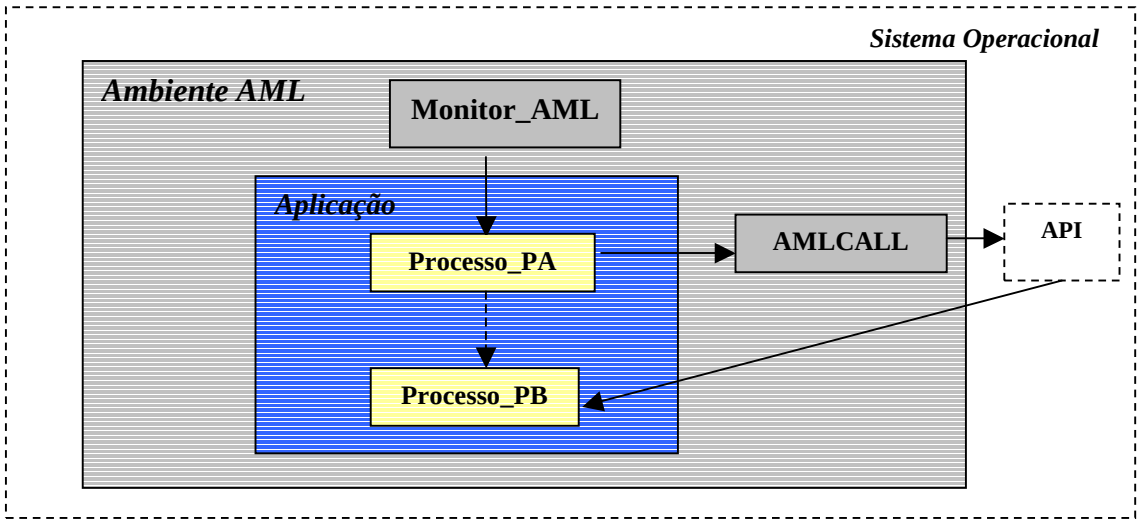

Figura-2 Interação de Paradigmas sem dependência de dados

7.2 Interação de Paradigmas com dependência de dados.

Com esta alternativa, além da transferência de controle, a interoperabilidade entre processos ocorre também com a transferência explícita de dados. Para a implementação desta interoperabilidade, o ambiente multiparadigma provê mecanismos responsáveis pela transferência de dados entre os processos componentes. Nosso ambiente será responsável pela alocação e gerenciamento de áreas compartilhadas de dados, disponibilizadas aos processos através de serviços do ambiente.

O módulo monitor do ambiente é responsável por iniciar as áreas compartilhadas que deverão corresponder aos tipos de dados usualmente disponíveis nas linguagens suportadas pelo ambiente. Cabe ao ambiente cuidar para que os devidos ajustes de representação sejam implementados, de forma que se torne transparente para a aplicação o formato interno dos dados.

Por simplicidade de implementação, estamos assumindo que a troca de dados entre linguagens apenas se dará através de tipos compativelmente representados em todas as linguagens componentes da aplicação.

A tabela a seguir foi obtida de [Spinellis-94], e sugere uma representação de dados comum entre diversos paradigmas.

Tipo de Dados	***Representação comum***
Valor Inteiro	Dois, quatro ou oito bytes
Valor String	Vetor de Caracteres
Caractere	Um byte (ASCII)
Número Ponto Flutuante	Seqüência de bytes IEEE 488
Booleano	Bit ou byte simples
Estrutura de dados	Seqüência de bytes

O nosso ambiente disponibiliza as funções primitivas *AML_IMPORT* e *AML_EXPORT* que são chamadas pelos processos que necessitem de troca de informações. A chamada dessas primitivas de ambiente deve estar acompanhada do nome interno da área no processo, o nome da área externa a ser gravada no ambiente, o tipo de dados e tamanho. Cabe ao ambiente gerenciar o espaço de nomes a ser compartilhado entre os vários processos componentes da aplicação, além de garantir que o uso das áreas compartilhadas não acarrete anomalias de atualização devido ao seu possível uso simultâneo por processos concorrentes. Assim, mecanismos de sincronização destas áreas compartilhadas são implementados automaticamente pelo ambiente, através de semáforos, mutex, regiões críticas ou outros.

O ambiente multiparadigma também se encarrega da liberação de áreas não mais utilizadas pelos processos. Assim, o ambiente deve estar permanentemente vigilante para impedir o acesso a quaisquer áreas não mais utilizadas por processos já encerrados.

A figura-3, a seguir, ilustra o esquema de transferência de dados proposto.

A interoperabilidade entre os processos ocorre, como vimos, através de áreas compartilhadas, as quais são alocadas em nosso ambiente proposto, através de três mecanismos.

O primeiro se refere a uma área de dados compartilhada a todos os processos componentes da aplicação, e criada pela primitiva de ambiente AML_ALLOC_DATA_AREA.

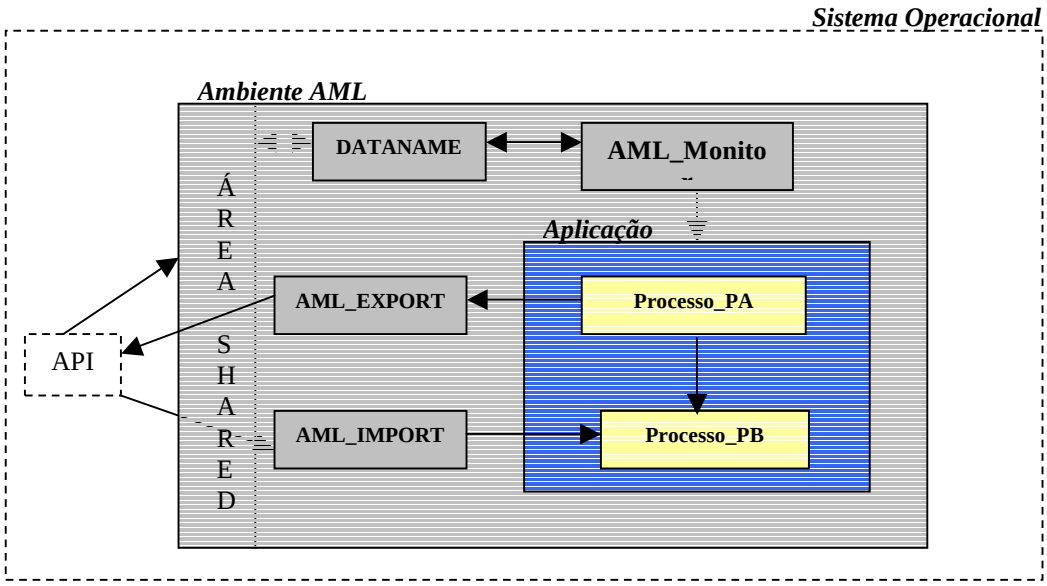

Figura-3 Interação de Paradigmas com dependência de dados

As primitivas de ambiente AML_EXPORT e AML_IMPORT são usadas pelos processos que desejarem efetuar leitura ou escrita nesta área compartilhada.

No caso da área de dados, a leitura das informações é não-destrutiva, ou seja, sucessivas operações de leitura são disparadas na mesma área, sem que se perca o conteúdo da mesma.

Uma segunda forma de implementação da interoperabilidade de processos se dá através da primitiva do ambiente *AML_ALLOC_QUEUE*, a qual aloca uma estrutura na forma de pilha ou fila que é acessada de forma compartilhada por todos os processos da aplicação.

Neste caso, a gravação/leitura dos dados na fila de dados é feita pelas primitivas do ambiente *AML_EXPORT_QUEUE* e AML_IMPORT_QUEUE. Com este mecanismo, a leitura dos dados é destrutiva, significando que após a leitura dos dados, estes não mais residirão na fila.

Finalmente, uma terceira forma de implementarmos a comunicação de dados se dá através da implementação de áreas booleanas (*flags*) que são criadas através da primitiva de ambiente *AML_ALLOC_SWITCH*. Estes interruptores têm um comportamente análogo à variáveis booleanas de ambiente e também têm acesso compartilhado à todos os processos da aplicação.

As primitivas de ambiente *AML_UPDATE_SWITCH* e *AML_READ_SWITCH* são as responsáveis pela gravação e leitura dos interruptores.

8. Observações relativas à implementação do ambiente proposto

Para validar a nossa proposta de projeto do ambiente multiparadigma de programação (AML), utilizamos um protótipo construído com as seguintes características:

- As aplicações multiparadigmas construídas sob o ambiente AML proposto, se utilizam das seguintes linguagens e seus respectivos compiladores ou interpretadores:
  1. Imperativo – MS Visual C++ 6.0
  2. Orientado a Objetos – Java SUN - JDK 1.1 (http://www.sun.com)

3. Lógico – SWI Prolog Version 3.2.9 (http://www.swi.psy.uva.nl/projects/SWI-Prolog/)
4. Funcional – NewLisp (http://www.nuevatec.com/newlisp)

- A plataforma proposta neste protótipo é *MS-Win32*.
- Todos os módulos correspondentes aos processos componentes da aplicação, são gerados através da execução dos respectivos compiladores ou interpretadores.
- Tendo em vista que todos os compiladores/interpretadores utilizados no protótipo proposto se utilizam da linguagem C como linguagem-base de interface, esta é utilizada como linguagem base no qual são construídas as primitivas do ambiente de programação AML.

Para a execução da aplicação multiparadigma, o ambiente AML é inicialmente carregado para a alocação de áreas compartilhadas. Em seguida, o usuário informa o nome do processo principal de modo que o ambiente inicie a aplicação. O usuário também pode passar parâmetros de configuração inicial, tais como, quantidade e tamanho de áreas compartilhadas.

Para a implementação da transferência de controle entre os diversos processos componentes da aplicação, a primitiva AMLCALL é gerada através de DLL's do ambiente *Win32*. Nosso protótipo, utiliza chamadas de funções de ambientes armazenadas na forma de DLL's, sob *Win32*.

Assim, a primitiva AMLCALL é implementada na forma de DLL, o qual conterá a chamada da API *CreateProcess*, que se encarrega de iniciar o processo desejado. Segue abaixo, um exemplo de uma chamada do paradigma lógico efetuado a partir do paradigma imperativo, com o uso da API *CreateProcess* sob *Win32*. Detalhes da API *CreateProcess* poderão ser encontrados em [Brain-96] ou [Richter-97].

```
#include <windows.h>
#include <iostream.h>
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nShowCmd)
{
 CHAR cmdStr[MAX_PATH] = "C:\\Program Files\\pl\\bin\\PLWIN.EXE";
 STARTUPINFO startUpInfo;
 PROCESS_INFORMATION procInfo;
 BOOL success;
 GetStartupInfo(&startUpInfo);
 success=CreateProcess(0, cmdStr, 0, 0, FALSE, CREATE_NEW_CONSOLE, 0, 0, &startUpInfo, &procInfo);
 if (!success)  cout << "Erro na criacao do processo:"  << GetLastError() << endl;
 return 0;}
```

O código seguinte, obtido de [Richter-97], demonstra como exportar uma função chamada *Aml_exp* e uma variável compartilhada inteira chamada *com_area* a partir de uma DLL:

```
__declspec (dllexport)  int   Aml_exp ( int nA1, int nA2) {
                        return (nA1 + nA2);
}
__declspec (dllexport)  int com_area = 0;
```

O compilador reconhece a declaração __declspec(dllexport) e a incorpora no arquivo OBJ resultante. Esta informação também é processada pelo *linker* quando todos os arquivos OBJ e DLL forem *linkeditados*. Maiores detalhes para a construção de DLL's poderão ser encontrados em [Richter-97], [Brain-96], [Rector-97], [Petzold-99], ou ainda em [Solomon-98].

Para a alocação de memória compartilhada, empregamos a técnica descrita em [Richter-97], conhecida por *memory-mapped*, a qual emprega, sob *Win32*, os mesmos procedimentos que são usados pelo sistema operacional para implementar a gerência de memória virtual. Assim, os arquivos *memory-mapped* permitem ao programador reservar uma região do espaço de endereçamento e efetuar um *committment* de memória física para esta região. Segue abaixo, um exemplo de uma possível implementação de uma DLL do ambiente para a alocação da área compartilhada.

```
#include "AML_exp"
#include <iostream.h>
#include <windows.h>
EXPORT  void AML_exp(int variavel)
{
        HANDLE hmmf;
        int * pInt;
        hmmf = CreateFileMapping((HANDLE) 0xFFFFFFFF, NULL,
                                        PAGE_READWRITE, 0, 0x1000, "AMLDATA");
        if (hmmf == NULL) {  cout << "Falha na alocacao da memoria compartilhada.\n";
                exit(1); }
        pInt = (int *)MapViewOfFile(hmmf, FILE_MAP_WRITE,0,0,0);
        if (pInt == NULL) {  cout << "Falha no mapeamento de memoria compartilhada!\n";
          exit(1); }
       if (*pInt == 0) { cout << "\nAmbiente AML fora do ar!\n Impossivel exportar  variaveis!!!!\n";
                exit(1); }
        *(pInt + 1) = variavel;
        cout << "Exportando Variavel: " << *(pInt+1)  << '\n';
        UnmapViewOfFile(pInt);
}
```

Para maiores detalhes na implementação de memória compartilhada através da técnica *memory-mapped*, poderemos consultar [Richter-97] e [Kruglinski-98].

9. Conclusão

Este artigo apresentou as motivações e uma proposta de realização concreta de ambiente para programação multiparadigma. Uma evidente aplicação deste conceito pode considerar a presença de mais de uma linguagem de um mesmo paradigma no sistema, bem como a eventual ausência de outros paradigmas, ou ainda uma combinação qualquer dessas situações.

A proposta aqui apresentada foi implementada e testada em situações simples, e está sendo complementada para formar um ambiente experimental em que possam ser desenvolvidos programas segunda a técnica da decomposição multiparadigma ou multilinguagem. Os resultados até aqui obtidos encorajam o prosseguimento do trabalho e sua aplicação pedagógica no ensino de linguagens de programação, engenharia de software, ambientes de programação e sistemas operacionais.

Prevê-se para breve a publicação de dissertação de mestrado do primeiro autor deste trabalho, no qual os detalhes de seu projeto e realização estão descritos.

Uma importante contribuição, prevista para um futuro próximo, refere-se ao desenvolvimento e incorporação de recursos para a utilização do paradigma adaptativo rapidamente mencionado no item 4 deste trabalho. Além de sua contribuição intrínseca, isto deverá oferecer um importante recurso adicional para o desenvolvimento da linha de pesquisa sobre tecnologias adaptativas, em curso no Departamento de Engenharia de Computação e Sistemas Digitais da Escola Politécnica da Universidade de S. Paulo.

Um grande número de desdobramentos deste trabalho pode ser vislumbrado, envolvendo atividades com banco de dados e com a rede de computadores, no sentido de tornar mais prático e versátil a ferramenta aqui proposta. Trabalhos nesta direção deverão ser iniciados em breve.

9. Referências Bibliográficas

[Brain-96] Marshall Brain, *"Win32 System Services"*, Prentice Hall PTR, Second Edition, 1996, ISBN: 0-13-324732-5.

[Bryan-99] Jeff Bryan. *"The Paradigm Effect*". Fly Fisherman; Harrisburg. Jul 1999, Volume 30, ISSN: 00154741.

[Budd-95] Timothy A. Budd, *"Multiparadigm Programming in LEDA"*, Oregon State University, Addison-Wesley Publishing Company, Inc, 1995, ISBN: 0-201-82080-3.

[Hailpern-86a] Brent Hailpern - *"Multiparadigm Languages and Environments*", IBM - IEEE Software - January 1986 – Guest Editor's Introduction.

[Hailpern-86b] Brent Hailpern, *"Multiparadigm Research: A Survey of Nine Projects"* - IEEE Software - January 1986.

[Kruglinski-98] David Kruglinski, George Shepherd, e Scot Wingo, *"Programming Microsoft Visual C++"* – Fifth Edition, Microsoft Press, 1998, ISBN: 1-57231-857-0.

[Kuhn–62] Thomas S. Kuhn, *"The Structure of Scientific Revolutions"*, University of Chicago Press, Chicago, 1962.

[Mellish-94] C.S. Mellish e W.F. Clocksin, "*Programming in Prolog*", Springer-Verlag Berlin Heidelberg, Fourth Edition, 1994, ISBN: 3-540-58350-5.

[Neto-87] João José Neto, *"Introdução à Compilação"*, Editora LTC, Rio de Janeiro, 1987

[Neto-94] João José Neto, "*Adaptive automata for context-dependent languages*", ACM SIGPLAN Notices, Volume 29, Número 9, Setembro 1994.

[Pereira-99] Joel Camargo Pereira, "*Ambiente Integrado de Desenvolvimento de Reconhecedores Sintáticos, baseado em Autômatos Adaptativos*", Dissertação de mestrado apresentada ao Departamento de Engenharia de Computação e Sistemas Digitais da Escola Politécnica da Universidade de São Paulo, São Paulo, 1999.

[Petzold-99] Charles Petzold, "*Programming Windows*", Fifth Edition, Microsoft Press, Microsoft Programming Series, 1999, ISBN: 1-57231-995-X.

[Placer-91] John Placer, *"Multiparadigm Research: A New Direction in Language Design"*, ACM Sigplan Notices, Volume 26, Número 3, páginas:9-17, March 1991.

[Rector-97] Brent E. Rector e Joseph M. Newcomer, "*Win32 Programming*", Addison Wesley Longman, Inc., Addison-Wesley Advanced Windows Series, Alan Feuer, Series Editor, 1997, ISBN: 1-57231-995-X.

[Richter-97] Jeffrey Richter, "*Advanced Windows*", Third Edition, Microsoft Press, 1997, ISBN: 1-57231-548-2.

[Santos-84] Sueli Mendes dos Santos – "*Programação Concorrente: Mecanismos de Sincronização e Comunicação de Processos"* – IV Escola de Computação – Instituto de Matemática e Estatística – USP – 1984.

[Sethi-96] Ravi Sethi, *"Programming Languages Concepts & Constructs"*, Addison Wesley, Second Edition, 1996, ISBN: 0-201-59065-4.

[Solomon-98] David A. Solomon, "Inside Windows NT", Second Edition, Microsoft Press, Microsoft Programming Series, 1998, ISBN: 1-57231-677-2.

[Solomon-98] David A. Solomon, "*Inside Windows NT*", Second Edition, Microsoft Press, Microsoft Programming Series, 1998, ISBN: 1-57231-677-2.

[Spinellis-94] Diomidis D. Spinellis, *"Programming Paradigms as Object Classes: A Structuring Mechanism for Multiparadigm Programming"*, February 1994, A thesis submitted for the degree of Doctor of Philosophy of the University of London and for the Diploma of Membership of Imperial College.

[Watt-90] David A. Watt, "*Programming Language Concepts and Paradigms*", Prentice Hall International Series in Computer Science – C.A. R. Hoare Series Editor – Europe 1990, ISBN: 0-13-728866-2.

[Zave-89] Pamela Zave, "*A compositional Approach to Multiparadigm Programming*", AT&T Laboratories, IEEE Software - September 1989

[Zave-96] Pamela Zave and Michael Jackson, *"Where Do Operations Come From? A Multiparadigm Specification Technique"*, I EEE Transactions on Software Engineering, Vol. 22 Número 7 - July/1996, pp 508-528.