Introdução ao Ambiente Visual Spider

Leonardo Mendonça de Moura e Carlos José Pereira de Lucena
Pontifícia Universidade Católica do Rio de Janeiro
Departamento de Informática
e-mail: moura@lids.puc-rio.br e lucena@inf.puc-rio.br

Resumo

Composição visual é o desenvolvimento interativo de aplicações através da manipulação direta de componentes reutilizáveis. Acreditamos que composição visual lida diretamente com a complexidade de grandes sistemas de software, fazendo o desenvolvimento mais fácil, flexível, e claro. Isto é alcançado pela implementação de abstrações reutilizáveis e conceitos de visualização. O projetista torna-se um criador de componentes e não um criador de grandes aplicações que são difíceis de manter e evoluir. As aplicações são construídas pela composição destes componentes.

Palavras-chave: programação visual, componentes, reuso de caixa preta

Abstract

Visual composition is an interactive development of applications by the direct manipulation of reusable components. We believe that the visual composition approach directly deals with the complexity of large software systems making the development easier, more flexible, and more understable. This is accomplished by implementing abstraction, reuse, and visualization concepts. The developer becomes a component builder and no longer creates large applications that are hard to maintain and enhance. The user will have the freedom to choose the components to build an application, and to mix and match components from different developers until the desired functionality is achieved.

Keywords: visual programming, components, black box reuse

1. Introdução

A linguagem Spider troca o paradigma textual de programação por um paradigma visual [8] de manipulação direta e edição de componentes de aplicação e interface com o usuário. Spider encoraja uma abordagem orientada a componentes [23] para o desenvolvimento de aplicações onde coleções de componentes reutilizáveis podem ser cuidadosamente trabalhados em um ciclo de vida de software evolutivo. A linguagem tem como objetivo final o suporte a construção de grandes aplicações a partir destes componentes pré-fabricados.

Programação visual provê um ambiente onde programas (não somente interfaces com o usuário) são produzidos pela manipulação direta de ícones ao contrário da programação textual convencional. Composição visual combina o paradigma visual com uma abordagem orientada a componentes. Também definimos composição visual como um desenvolvimento interativo de aplicações pela manipulação direta de componentes reutilizáveis. Esta abordagem

lida diretamente com a complexidade de grandes sistemas. A complexidade é controlada de tal forma que o sistema torna-se mais fácil de ser desenvolvido, testado e depurado, ganhando também flexibilidade e clareza. Isto é alcançado pelo uso de abstrações, reutilização e visualizações. O objetivo é ter o projetista como um criador de componentes e não mais como o criador de grandes aplicações que são dificeis e manter e evoluir. As aplicações são construídas pela composição de componentes criados por diferentes projetistas que juntos definem a funcionalidade desejada. Na composição visual a função do programador é minimizada. O foco é na reutilização e composição de software. Linguagens convencionais de programação são usadas para desenvolver componentes de software e não para o desenvolvimento de aplicações específicas. Estas entidades chamadas componentes constituem o principal elemento computacional do ambiente de programação Spider. Componentes são objetos de alto nível construídos através de programação textual ou visual. Componentes definidos textualmente (componentes primitivos) são programados em linguagens convencionais. Componentes construídos visualmente (componentes compostos) são implementados pela manipulação direta de outros componentes. Os componentes definem uma biblioteca de blocos de construção de aplicações.

Como descrito acima, existem duas funções importantes no desenvolvimento de software dentro do ambiente Spider. A primeira esta relacionada com o desenvolvimento de componentes reutilizáveis, esta função é exercida por um projetista especial denominado fabricante. A outra é relacionada com a construção de aplicações através de composição visual. Este papel é exercido por um projetista denominado construtor. Afirmamos que os fabricantes devem ser projetistas com alto nível de conhecimento e os construtores podem ser projetistas comuns. O paradigma visual funciona como uma ponte entre os fabricantes e construtores.

Os fabricantes criam arquiteturas baseadas na análise de domínios [3, 9]. Usando a análise de domínio como ponto de partida os fabricantes criam e selecionam componentes. Quando o construtor utiliza uma arquitetura, ele seleciona que componentes irão ajudá-lo na construção de uma nova aplicação. A composição de componentes é restrita por um conjunto de regras definidas pelo fabricantes. Definimos arquitetura como um conjunto de componentes, regras de composição, *wizards* e ferramentas. *Wizards* são criados para ajudar os construtores na criação de aplicações, e as ferramentas são um tipo de macro que podem ser utilizadas para automatizar tarefas repetitivas. Podemos ver os fabricantes como um tipo de criador de ambientes para particulares domínios de aplicação, e os construtores como usuários destes domínios.

2. Elementos da Linguagem Spider

Componentes constituem o principal elemento de computação da linguagem Spider. Eles são caixas pretas que são construídas através de programação textual ou visual. Componentes possuem um conjunto de portas que constituem a sua interface. Estas portas definem o protocolo do componente.

Componentes compostos são formados por outros componentes (componentes internos). Um componente composto especifica as conexões entre um conjunto de componentes internos. A conexão entre as portas de dois componentes internos define um tipo de comunicação (Figura 1). Os componentes internos podem ser primitivos, ou outros componentes compostos. Componentes primitivos são implementados em linguagens convencionais como C++ ou Basic.

Se um componente composto é utilizado na construção de outro componente, somente um visão externa (caixa preta) do componente é fornecida.
Um componente composto encapsula um conjunto de componentes parcialmente conectados e determina que porta serão visíveis para outros componentes. Note que as portas de componentes compostos funcionam como ponte para as portas dos componentes internos. Portas de entrada dos componentes compostos são conectas as portas de entrada de seus componentes, e da mesma maneira acontece com as portas de saída.

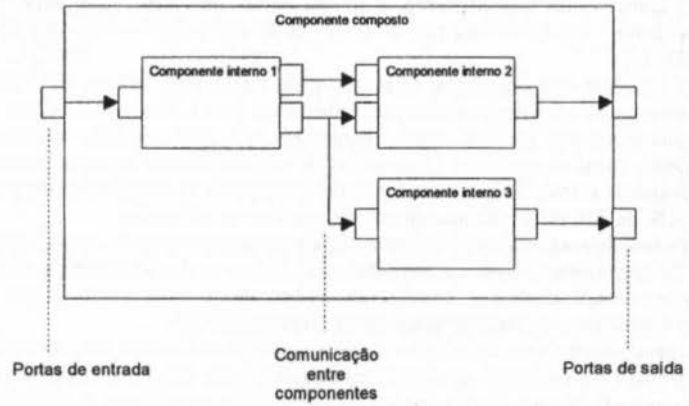

Figura 1 Componente Composto

Componentes compostos são criados para encapsular comportamentos e serviços particulares. Componentes compostos são como *designs* reutilizáveis. Eles economizam muito tempo do projetista não o obrigando a reconstruir um mesmo padrão diversas vezes. Componentes compostos podem também ser vistos também como exemplos de como se pode construir um componente com um determinada funcionalidade, servindo desta forma como um padrão que pode ser modificado pela adição, remoção e troca de componentes internos.
O fator de reutilização de componentes é aumentado pela introdução do conceito de arquiteturas. Arquiteturas permitem que componentes trabalhem de diversas formas. Normalmente assume-se que está bem definido quais componentes podem comunicar-se entre si. Definindo, desta forma, o conceito de compatibilidade entre componentes. No ambiente Spider não está definido de imediato que componentes podem comunicar-se entre si. Tal fato só é definido quando um componente é associado a um arquitetura. A arquitetura supre as regras de composição para a interação de componentes em um particular domínio de aplicação. Desta maneira, um componente pode ser compatível com diferentes conjuntos de componentes dependendo da arquitetura onde é utilizado. Um componente pode ser reutilizado em diferentes situações simplesmente modificando a arquitetura em vez de modificá-lo. Como exemplo, uma regra de composição de uma arquitetura poderia permitir que portas que retornam números inteiros sejam conectadas a portas que esperam *strings*, fazendo a conversão entre os dois tipos. Outros exemplos podem ser encontrados em [22, 21].
Bertrand Meyer define *weak coupling* ou *small interfaces* como [19]: "Se quaisquer dois módulos se comunicam, eles devem trocar a menor quantidade de informação possível." Ele

justifica este princípio pelo critério de continuidade e proteção, atestando que a modificação de uma pequena parte da interface de um módulo deve resultar em um pequena quantidade de modificações em outras partes do sistema. Mas esta não é a única vantagem. *Weak coupling* também assegura a composição de componentes de software, pois pequenas portas de comunicação aumentam a probabilidade de encontrar componentes que podem participar de uma comunicação.

Em Spider, descrevemos o princípio de *weakest coupling*, que é similar ao usado em [25]: "Para componentes informarem outros componentes sobre algo, eles não precisam determinar os outros componentes explicitamente, e devem enviar mensagens com uma quantidade mínima de dados." Em Spider este tipo de comunicação entre objetos é baseada na teoria de ADV/ADO [11].

Com este princípio nos esperamos obter um grau máximo de composição mesmo para componentes que não foram planejados para trabalharem juntos. Havendo uma boa chance de existir combinações não previstas entre componentes se o dado recebido ou trasmitido for simples. Desta forma os projetistas (fabricantes) de um componente tem que pensar somente sobre as portas de entrada e saída, ficando para o construtor a tarefa de combinar estes blocos de construção de diferentes maneiras afim de obter diferentes aplicações.

Com o objetivo de manter o modelo simples, Spider atualmente suporta apenas comunicação síncrona. Se uma porta de saída é conectada a mais de um porta de entrada, é utilizada uma estratégia de execução *depth-first*. Usualmente a ordem das conexões é definida pela seqüência de criação das mesmas, mas o projetista pode modificar esta ordem.

Em linguagens visuais a área de trabalho torna-se confusa rapidamente com a proliferação de nodos e arcos. Para minimizar este efeito introduzimos dois conceitos: camadas (*layers*) e sombras (*shadows*). O conceito de camada é bem semelhante ao utilizado em aplicações CAD, onde o usuário pode dividir o design em grupos de elementos correlatos. Sombras são usadas quando queremos usar os mesmos componentes em diversas posições diferentes (ex.: em diferentes camadas). Sombras também podem ser utilizadas para organizar grafos complexos, evitando por exemplo a sobreposição de conexões. Acreditamos que este conceitos podem ajudar a organizar o código visual.

3. O Protótipo do Ambiente

Implementamos um protótipo do ambiente visual Spider. A versão atual suporta a conexão gráfica de componentes, e a execução interativa de aplicações. Diversos componentes primitivos e compostos foram implementados para motivos de demonstração.

3.1. Exemplo

Explicar uma linguagem visual sem figuras é uma tarefa quase que impossível. Desta forma, apresentamos nesta seção um exemplo de uma aplicação simples construída no ambiente Spider. Queremos neste exemplo demostrar como os diversos conceitos apresentados se enquadram dentro do ambiente. Outros exemplos podem ser encontrados em [21,22].

3.1.1. Descrição da Aplicação

Iremos ilustrar a funcionalidade do ambiente com um exemplo similar ao mostrado na documentação do ambiente Visual Age [26]. Este exemplo é denominado *shopping list*, a Figura 2 mostra a aplicação sendo executada. Esta aplicação exemplo permite que o usuário adicione e remova itens na lista de compras. Um elemento é adicionado na lista colocando-se seu nome no campo *New item* e apertando o botão *Add*. Para remover um elemento, o usuário deve selecionar o elemento na lista *Itens* e apertar o botão *Remove*.

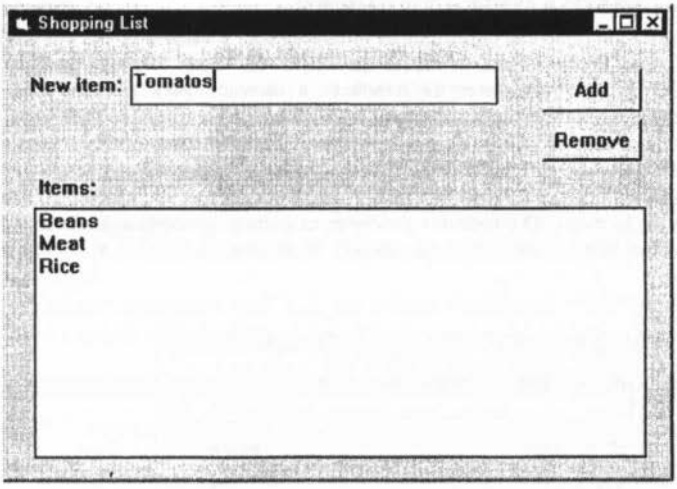

Figura 2 Aplicação *Shopping List*

3.1.2. Descobrindo Artefatos de Software Reutilizáveis

Em ambientes convencionais de desenvolvimento é difícil encontrar artefatos de software reutilizáveis, e entender os recursos disponíveis contidos neles. O ambiente Spider torna esta tarefa um pouco mais fácil através da inclusão de *palettes* de componentes compatíveis em um formato facilmente acessível para os construtores (Figura 3).

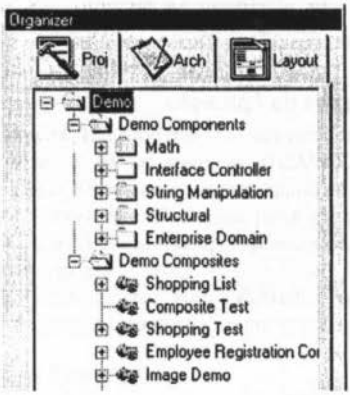

Figura 3 *Organizer*

3.1.3. Construindo a Interface com o Usuário

Spider possui um poderoso construtor de interfaces com o usuário que provê um conjunto de *widgets* como botões, listas, e caixas de edição de texto. Usando conceitos da teoria ADV/ADO [11], o construtor de interfaces é utilizado para a construção dos ADVs (interfaces) de uma aplicação. É importante notar que o usuário pode reutilizar ADVs previamente criados na criação de novos ADVs. A Figura 4 mostra o construtor de interfaces do Spider editando a interface da aplicação *Shopping List*. Os *widgets* também estão organizados em *palettes* (Figura 3), e podem ser selecionados e utilizados pelo usuário através de manipulação direta. O construtor de interfaces permite a modificação de propriedades dos *widgets* como largura, altura e nome, através do mouse e/ou acesso a lista de propriedades (Figura 4).

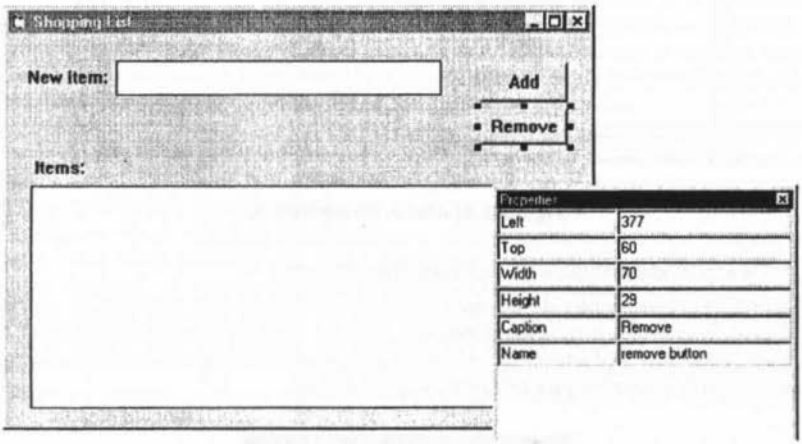

Figura 4 Construtor de interfaces com o usuário do ambiente Spider

3.1.4. Construindo a Lógica da Aplicação

O ambiente permite a composição de componentes através do *visual builder*. Usando conceitos da teoria de ADV/ADO, o *visual builder* é usado para construir novos ADOs (componentes) a partir de componentes previamente criados, sejam estes primitivos ou compostos. A Figura 5 mostra a implementação do recurso de adição de novos elementos na aplicação *shopping list*. Neste exemplo, podemos observar a utilização de três camadas (*Add Item Layer, Remove Item Layer* e *Main Layer*) para organizar o código visual. Deve ser notado que há não há diferença entre componentes primitivos e compostos no desenvolvimento de novos componentes. Este exemplo pode ser resumido da seguinte forma:

Descrição das conexões:

- (*click, create*): uma conexão evento-ação (*event-action*)
- (*source, text*): uma conexão pergunta-responde (*ask-answer*)
- (*click, exec add*): uma conexão evento-ação
- (*new instance, elem to add*): uma conexão pergunta-responde (*ask-answer*)

Interpretação:

- Quando o usuário aperta o botão *add*, o evento *click* é disparado.
- As ações associadas ao evento *click* são executadas.
- A primeira ação (*create*), cria um novo elemento para ser adicionado na lista.
  - Para criar um novo elemento, a *factory* necessita de uma matriz, cujo valor será copiado para o novo elemento.
  - A *factory* **pergunta** (*ask*) ao *entry field* por esta informação via a conexão (*source, text*).
- A segunda ação (*exec add*), adiciona o elemento na *collection*
  - Para adicionar um elemento a *collection* precisa saber qual elemento será adicionado.
  - A *collection* **pergunta** (*ask*) a factory pelo novo elemento criado via a conexão (*new instance, elem to add*)

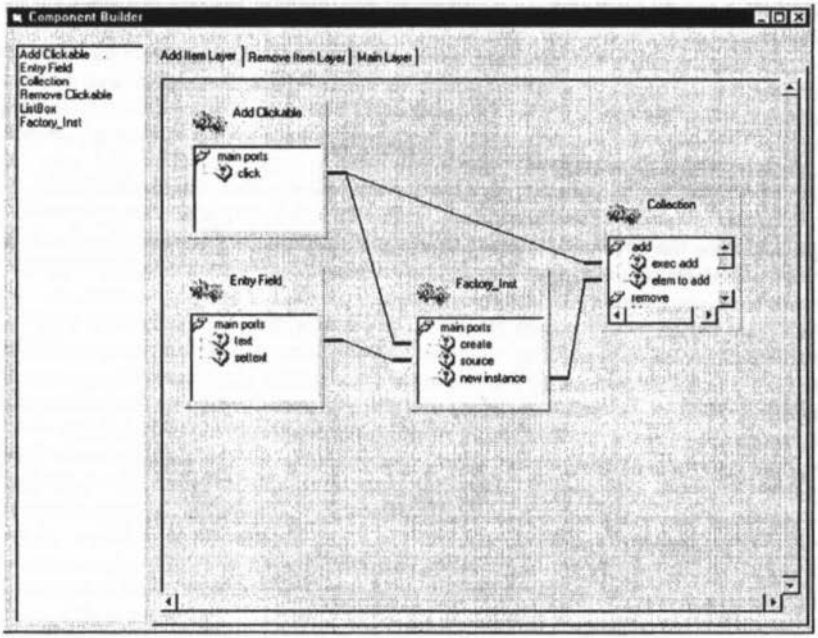

Figura 5 Lógica da aplicação para adicionar um item

3.2. *A Implementação do Protótipo*

O protótipo do ambiente Spider foi implementado em Visual C++ [28] e Visual Basic [27]. As arquiteturas, componentes compostos, e aplicações são armazenadas em um banco de dados relacional. Os componentes compostos são representados como grafos. O componentes

primitivos podem ser implementados em Visual C++, Delphi [6], Visual Basic, ou qualquer outro ambiente que suporte OLE Automation [7].
O protótipo tem como objetivo servir como um protótipo evolutivo. A interface com o usuário do ambiente está continuamente sendo modificada para a adição de novos recursos, e para atender a pedidos dos usuários do ambiente. Alguns recursos ainda não foram implementados no protótipo, mas estamos trabalhando numa versão integral (1.0) do ambiente com a experiência adquirida na implementação da protótipo.

4. Trabalhos Correlacionados

Prograph [12, 14] usa um modelo de computação baseado em fluxo de dados. Este ambiente permite que o programador crie código utilizando representações icónicas de chamadas de métodos pela conexão de linhas, que representam o fluxo de dados de um método para outro. Não está claro neste ambiente porque é mais fácil programar ou aprender a linguagem do que nos sistemas textuais tradicionais. O usuário tem que desenvolver aplicações utilizando as mesmas técnicas utilizadas em linguagens textuais. Desta forma, o usuário tem que ser um bom programador afim de conseguir criar aplicações complexas. Entretanto, Prograph consegue substituir com sucesso o código textual, por representações icónicas que representam ações. O ambiente força o usuário a gerar um código mais claro e modular, pois a área de trabalho rapidamente fica confusa com arcos e nodos. Acreditamos que os conceitos de camadas e sombras existentes no ambiente Spider facilitariam a organização do código dentro do ambiente Prograph.
Vista [23, 18] é uma ferramenta de composição visual que é parte do projeto ITHACA [1, 17]. As principais diferenças entre Vista e o ambiente Spider são: Vista precisa ser recompilado toda a vez que um novo componente é adicionado; a criação de componentes primitivos em Vista é uma tarefa complexa e pode ser feita apenas em C/C++; Vista não explora conceitos de programação visual mais avançados como camadas e sombras.
Visual Age [26] não é exatamente um projeto de pesquisa, mas é um dos únicos produtos de programação visual do mercado para o desenvolvimento de aplicações. A principais diferenças são para o ambiente Spider são: o projetista tem que possuir um conhecimento profundo da linguagem onde os componentes foram construídos; os componentes construídos em uma linguagem não podem ser utilizados com componentes criados em outras linguagens; no Visual Age não é possível construir novos componentes utilizando-se apenas programação visual. O Visual Age é muito mais dirigido para a conexão entre os elementos de lógica da aplicação e os elementos de interface com o usuário, do que para a construção da lógica da aplicação através de programação visual.
Sibal [24] é uma linguagem visual para a especificação de interfaces gráficas, que separa corretamente os conceito de: definição do *layout* da interface, e definição do comportamento da interface. Esta linguagem também se baseia na idéia de componentes reutilizáveis que podem ser conectados, e nos paradigmas de *data-flow* e *control-flow*. O sistema possui um mecanismo de biblioteca de componentes, que permite a procura de um determinado componente. Desta forma, reutilização e trabalho cooperativo são pontos abordados pelo sistema. Sibal não explora conceitos de programação visual mais avançados como camadas e sombras.
Escalante [20] é um ambiente que suporta design interativo, prototipação, e geração automática de linguagens visuais complexas, sem muito esforço. Este sistema permite que o usuário especifique uma aplicação e sua interface pela definição do seu modelo de dados, e do modelo de visualização correspondente (usando um ambiente de especificação visual). O modelo de dados são grafos genéricos, enquanto que o modelo de visualização são gráficos.

Uma vez que o modelo de dados e visualização estejam criados, Escalante irá gerar um programa que implementa o modelo de dados e um mecanismo de visualização usando mecanismos fixos de controle. O programa resultante pode ser aperfeiçoado pela adição de código textual. Apesar do sistema possuir um mecanismo de definição de linguagens visuais bastante interessante e fácil de ser utilizado, este não aborda temas como reutilização, e adição de novos componentes.

VPlus [29] é um sistema de programação visual voltado para a criação de aplicações de tempo real, que processam dados, e os devolvem em tempo real. Este tipo de aplicação é geralmente um processador de sinal ou sistema de controle. Desta forma, o conceito de processador é fundamental neste sistema. Apesar deste sistema ser bastante interessante, este cobre apenas um pequeno grupo de aplicações.

Application Visualization System (AVS) [4] é um sistema que ajuda o projetista a criar aplicações gráficas de visualização e análise de dados. O coração do AVS é o Network Editor, o ambiente de programação visual propriamente dito. O Network Editor permite a conexão de diverso módulos AVS afim de gerar uma aplicação, que pode ser reutilizada e modificada posteriormente. Nós podemos afirmar que neste sistema também há uma separação dos projetistas em dois grupos: criação de módulos em C/C++ e Fortran (fabricantes), e composição de módulos através do Network Editor (construtores).

AppWare [2] autodenomina-se uma linguagem de quinta geração, que permite a criação de aplicações de maneira gráfica, a partir de componentes previamente construídos. Este sistema pode criar aplicações de acesso a banco de dados com grande facilidade, através da interconexão de funções. Estas funções são componentes de software denominados AppWare Loadable Modules (ALMs). ALMs podem ser reutilizados em diversas aplicações, e podem ser dispostas em diferentes servidores NetWare, afim de aumentar a performance. AppWare pode ser utilizado nas plataformas Windows e Machintosh ao mesmo tempo. Neste sistema também existe a separação dos projetistas em dois grupos como no ambiente Spider.

Em VIPR [10] o fluxo do programa é visualizado como uma série de anéis telescópicos. Entretanto, há muita confusão e ambigüidade nos elementos visuais para convencer alguém que o sistema possui alguma utilidade. VIPR somente provê uma representação gráfica para linguagens imperativas, desta forma, não é claro que construir programas em VIPR é mais fácil do que construir programas em linguagens textuais convencionais.

Visual Basic [27]/ Delphi [6] são ambientes de programação de propósito geral com suporte a criação interativa de interfaces com o usuário. Depois que os elementos de interface são compostos, o programador tem que escrever código para tratar os eventos de interface. A lógica da aplicação continua sendo escrita em uma linguagem de programação textual. Estes tipos de ambientes de programação tornam mais fácil a construção de interfaces com o usuário, mas não abordam os problemas de desenvolvimento de software em geral.

5. Trabalho Atual/Futuro

Atualmente, estamos trabalhando na versão 1.0 do ambiente Spider que será livremente distribuída. Para construir esta versão estamos desenvolvendo um framework [13] para editores de linguagens visuais, que chamamos MetaSpider. MetaSpider será utilizado em diversos projetos no Laboratório de Engenharia de Software (LES) da PUC-Rio.

Na versão 1.0 também estaremos expandindo o ambiente com um novo tipo de relacionamento denominado *views-a* que é parte da teoria de ADV/ADO [11]. Outra extensão diz respeito a personalização da linguagem visual em função do domínio de aplicação, pois acreditamos que as aplicações de um determinado domínio podem ser mais facilmente expressados por uma linguagem visual personalizada. Esta idéia é uma extensão da idéia de linguagens orientadas a

domínio [5, 15]. Um trabalho futuro será a definição de uma linguagem orientada a domínio para a personalização de linguagens visuais, iremos utilizar uma abordagem similar a de [16].

6. Conclusão

Nós estamos convencidos de que combinando o paradigma de programação visual com uma abordagem orientada a componente consiste em um passo na direção certa. O paradigma de programação visual provê uma interface gráfica e de manipulação direta para a construção interativa de aplicações para projetistas comuns. Os programadores com alto nível de conhecimento são construtores de componentes e ambientes que criam soluções para um domínio de aplicação que são utilizadas por diversos projetistas comuns. O mecanismo de comunicação definido permite a criação de blocos de construção (componentes) de alto nível e *weakly coupled*, aumentando, desta forma, a probabilidade de encontrar-se componentes reutilizáveis. Acreditamos que o ambiente visual Spider torne a programação menos frustrante e mais produtiva.

7. Agradecimentos

Os autores gostariam de agradecer ao resto da equipe de implementação do ambiente Spider: Bruno Bondarovsky, Luidi Fortunato, e Pedro Ripper. É também importante agradecer ao CNPq pelo seu suporte financeiro.

8. Refêrencias

[1] Ader, M.; Nierstrasz, O.M.; McMahon, S.; Miller, G.; Profrock, K.; "The ITHACA Technology: A Landscape for Object-Oriented Application Development"; Proceedings, Esprit 1990 Conference; pp. 31-51; Kluwer Academic Publishers; Dordrecht; 1990.

[2] AppWare User's Guide, Novel Inc., 1995.

[3] Arango, A.; "Domain Analysis Methods"; Software Reusability; Harwood; London; Março; 1993.

[4] AVS User's Guide; Advanced Visual Systems Inc; 1994.

[5] Bell, J.; et al; "Software design for reliability and reuse: A proof-of-concept demonstration"; In TRI-Ada '94 Proceedings; pp. 396-404; ACM; Nov. 1994.

[6] Delphi User's Guide; Borland Inc.; 1995.

[7] Brockshmidt, K.; "Inside OLE"; Second Edition; Microsoft Press; 1995.

[8] Shu, C. N.; "Visual Programming"; Van Nostrand Reinhold; New York; NY. 1988

[9] Clements, P.; "From Domain Model to Architectures"; A. Abd-Allah et al., eds.; Focused Workshop on Software Architecture; pp. 404-420; 1994.

[10] Citrin, W.; Doherty, M.; Zorn, B.; "Control constructs in a completely visual imperative programming language"; Techinical report CU-CS-673-93; Department of Computer Science; University of Colorado; Boulder; 1993.

[11] Cowan, D. D.; Lucena, C. J. P.; "Abstract Data Views: An Interface Specification Concept to Enhance Design for Reuse"; IEEE Transactions on Software Engineering; Março 1995.

[12] Cox, P. T.; Giles, F. R.; Pietrzykowski, T.; "Prograph: A step towards liberating programming from textual conditioning"; IEEE Workshop on Visual Languages; pp. 150-156; Rome; 1989.

[13] Gamma, E.; Helm, R.; Johnson, R. E.; Vlissides, J.; "Design Patterns, Elements of Reusable Object-Oriented Software"; Addsion-Wesley; 1995.

[14] Prograph Reference Manual; The Gunakara Sun Systems Ltd.; 1992.

[15] Hudak, P.; "Building Domain-Specific Embedded Languages"; Computing Surveys; 28A(4); ACM; 1996.

[16] Kleyn, M. F. E.; "A High Level Language for Specifying Graph Based Languages and Their Programming Enviroments"; PhD. Thesis; The University of Texas at Austin.

[17] Mey, V.; Nierstrasz, O.; "The ITHACA Application Development Environment"; Visual Objects; pp. 267-280; D. Tsichritzis (Ed.); Centre Universitaire d'Informatique; University of Geneva; 1993.

[18] Mey, V.; "Visual Composition of Software Applications"; Ph.D. thesis (no. 2660); Dept. of Computer Science; University of Geneva; 1994.

[19] Meyer, B.; "Object-Oriented Software Construction"; Prentice Hall International; 1988.

[20] McWhirter, J. D.; Nutt G. J.; "Escalante: An Environment for Rapid Construction of Visual Language Application"; Technical Report CU-CS-692-93; Department of Computer Science; University of Colorado; 1993.

[21] Moura, L. M.; "Uma Linguagem de Programação Visual"; Tese de Mestrado; Departamento de Informática; PUC-Rio; 1996.

[22] Moura, L. M.; Lucena, C. J. P.; "The Visual Language Spider"; to appear.

[23] Nierstrasz, O.; Gibbs, S.; Tsichritzis, D.; "Component-Oriented Software Development"; Communications of the ACM; 35(9); pp. 160-165; 1992.

[24] Rogers, I.; Cunningham, J.; "Towards a Visual Notation, and Editor, for User Interface Design"; UIDE Technical Report DTI/SERC: IED 4/1/1577; School of Cognitive and Computing Sciences; Sussex University; 1993.

[25] Schiffer, S.; Fröhlich, J. H.; "Visual Programming and Software Engineering with Vista"; Visual Object-Oriented Programming; Prentice Hall; 1995.

[26] Visual Age User's Guide and Reference; IBM Inc.; 1994.

[27] Visual Basic User's Guide; Microsoft Inc.; 1993.

[28] Visual C++ 4.0 User's Guide, Microsoft Inc., 1995.

[29] VPlus User's Guide, SimPhonics, Inc., 1995.