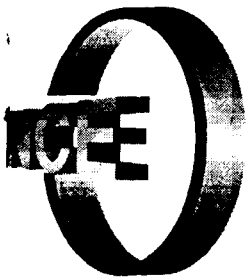


Relatório Técnico

Núcleo de Computação Eletrônica

Aplicação de Patterns no Desenvolvimento de Um Sistema CAD para Microeletrônica

Oliveira, C. E. T.
Duboc, A. L. C. L.
Pais, A. P. V.
Muniz, D. P.
Anido, M. L.

NCE - 17/99

Universidade Federal do Rio de Janeiro

Aplicação de Patterns no Desenvolvimento de Um Sistema CAD para Microeletrônica

Oliveira, C.E.T., Duboc A.L.C.L., Pais A.P.V., Muniz, D.P. and Anido, M.L.

Núcleo de Computação Eletrônica - Universidade Federal do Rio de Janeiro.
Caixa Postal 2324, Rio de Janeiro, RJ Brasil CEP 20001-970
E-mail: carlo@ nce.ufrj.br

Palavras chaves: Programação orientada a objetos, Padrões de Projeto, circuitos integrados

Resumo

Este artigo apresenta a aplicação de técnica de orientação a objetos para resolver problemas de integração em ferramentas de CAD para microeletrônica. A abordagem enfatiza o uso de *patterns* e componentização para obter homogeneidade e o reuso de *software*. Os componentes são responsáveis por realizar tarefas básicas como a interpretação de linguagens e edição gráfica. A divisão de arquitetura em camadas garante a modularidade e provê o acoplamento de ferramentas funcionais. A arquitetura também suporta a distribuição de objetos, aumentando o potencial de processamento em redes de computadores.

1. Introdução

Ferramentas CAD para microeletrônica são sistemas complexos que reunem partes desenvolvidas por grupos distintos. Esta diversificação é devido a grande especiliazação de cada parte, exigindo peritos que normalmente formam grupos afins. Uma ferramenta completa deve então integrar as partes de cada um destes grupos de peritos. Mesmo com uma prática de *marketing*, é sensato dividir a ferramenta em módulos que podem ser adaptados às necessidades dos usuários. O problema então passa para o nível de integração da ferramenta, no suporte de cada módulo e interoperabilidade entre eles. A ferramenta deve mostrar uma interface consistente para todos os módulos. Para isso a interface deve operar baseada em um modelo de interação comum a todos os módulos. Os módulos por sua vez tem que respeitar este modelo comum. Este modelo comum deve ser flexível o bastante para se adaptar às diversas naturezas de todos módulos que podem proceder de origens distintas.

Nossa experiência neste campo advém de problemas similares provenientes de *softwares* produzidos por estudantes. Ferramentas CAD de VLSI são uma fonte abundante de temas para o trabalho de estudantes universitários e de pós-graduação. Depois de servir ao seu propósito acadêmico, nós pretendemos fundir o trabalho dos estudantes em uma ferramenta integrada usada para o ensino de microeletrônica. Para este propósito nós desenvolvemos uma arquitetura modular capaz de empreender funcionalidades novas através de pequenos ajustes.

A solução proposta está centrada na componentização e em uma arquitetura modular em camadas. Esta baseia-se em uma coleção de técnicas avançadas de programação aplicadas ao problema de projeto de microeletrônica. Essas técnicas são soluções já utilizadas em muitas aplicações reais. Através do reuso destas soluções foi possível aplicá-las na programação de ferramentas de microeletrônica. Além disso, esta estratégia foi utilizada para coordenar a divisão de tarefa entre os estudantes. Interfaces compatíveis podem ser definidas, encapsulando o trabalho de cada estudante sob uma especificação

facilmente integrável. O uso de protocolos padrão de componentização como DCOM estende os resultados à distribuição de objetos remotos.

Este artigo primeiramente descreve o problema de projeto de circuitos integrados, que é o enfoque de todo o desenvolvimento. Propõe-se então uma arquitetura voltada para integração de ferramentas, expondo-se a sua organização em camadas. No detalhamento desta arquitetura, as técnicas de programação orientada a objetos são descritas para cada parte do problema. Um editor gráfico é o protótipo utilizado para exemplificar a aplicação desta arquitetura. Seguem-se as considerações e conclusões desta experiência.

2. Problema relacionado ao projeto de microeletrônica

Um projeto de microeletrônica pode ter três níveis básicos de representação : comportamental, estrutural e física. A idéia do nosso sistema é poder suportar estas diversas representações e possibilitar o intercâmbio entre elas. O tratamento do problema a partir de cada uma das representações faz parte do processo de construção de circuitos integrados. A partir de um comportamento desejado (representação comportamental), sua lógica é projetada (representação estrutural) e esta conduz à descrição do circuito de transistores (representação física) .

Todos os tipos de descrição de um circuito podem ser definidos por linguagens. Assim, foi construído um módulo no sistema que implementa uma gramática que se integra na arquitetura como um todo estabelecendo uma ligação da parte passiva do sistema com a ativa , ou seja , o sistema lê os códigos da linguagem) e fornece uma representação computacionalmente ativa do nível físico.

O fato de linguagens serem um ponto em comum a diversas representações leva-nos a propor uma solução componentizada para o tratamento delas. Foi então projetado um conjunto de componentes visuais que implementasse o padrão de projeto (*pattern*) *Interpreter*. O *pattern Interpreter* define uma representação para a gramática de uma dada linguagem. Esta representação é comparada com um dado texto para traduzir as sentenças desta linguagem. Este *pattern* é representado no sistema por diagrama sintático visual que no nosso exemplo foi programado como um interpretador de linguagem CIF (*Caltech Intermediate Form*). Um grupo de componentes, que atuam como produções, estações, *tokens*, terminais e trilhos, foi desenvolvido com o intuito de se construir tal interpretador. No diagrama mostrado na Fig. 2 , o código é interpretado e transformado em dados necessários para a construção da representação física de circuitos pelo modelo. A entrada do diagrama abaixo representa *Boxes*. Sua entrada tem primeiramente a letra B, que nos diz tratar-se de um *Box* e inteiros que retratam : seu comprimento, sua largura e as coordenadas de seu centro, respectivamente. Ao identificar a letra B , o sistema dispara o método CreateBox que atua como um *pattern* do tipo *Builder* .

O *pattern Builder* separa a complexa construção de um certo objeto da sua representação. Possibilitando assim, a criação de diferentes representações do objeto a partir de um mesmo processo de construção. No sistema, a partir do que é lido pelo interpretador, os comandos são identificados e repassados de uma maneira legível para o modelo. O modelo fica encarregado de formular cada tipo de representação dos dados e passá-las para a vista. Como podemos observar, o controle se abstrai totalmente de como o objeto é representado, se preocupando apenas com o processo de construção do mesmo. No diagrama da Fig. 2 , dentro do método CreateBox existem outros métodos que capturam as demais informações pertinentes ao *Box* e as enviam para o modelo.

O diagrama sintático visual tem como objetivo o entendimento das informações contidas num arquivo em linguagem CIF pelo modelo e, consequentemente, pela vista. Os dois *patterns* utilizados se

integram de forma a fornecerem dados legíveis para o restante do sistema. Na Fig. 01 ilustramos melhor o relacionamento entre eles.

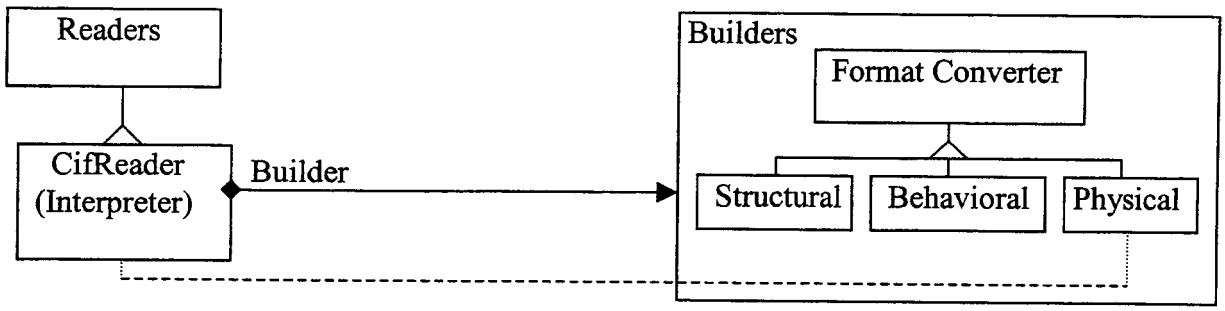


Fig. 01 : *Patterns Builder* e *Interpreter*

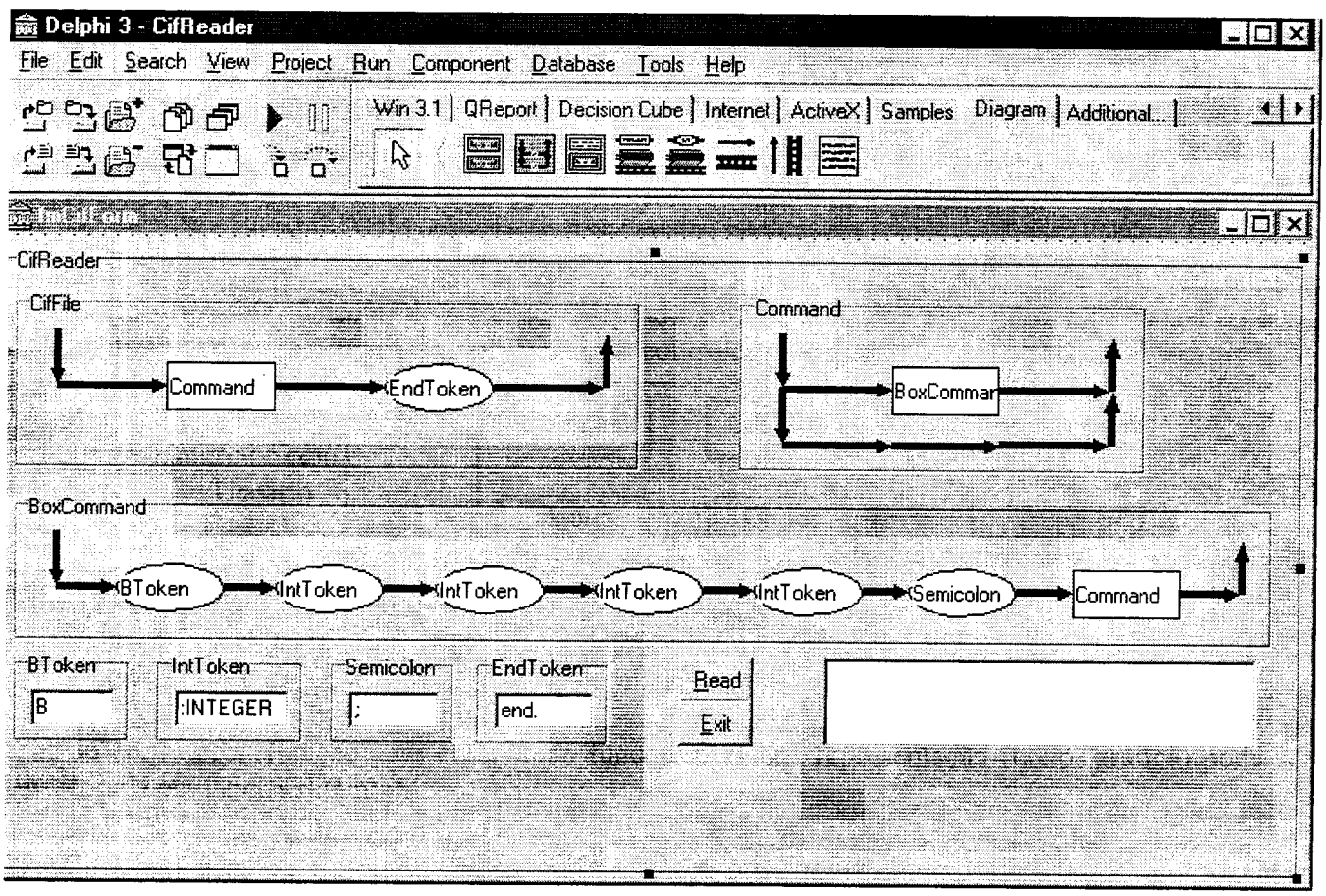


Fig. 02 : Diagrama Sintático Visual

3. Arquitetura em camadas

Ferramentas integradas de projeto para microeletrônica agregam partes que individualmente são especializadas e com alto teor tecnológico. Muitas vezes estas partes são desenvolvidas por diversos parceiros, cada um contribuindo com o seu conhecimento específico. A integração destas partes tão disjuntas requer um sistema flexível que se adapte a diversas especificações de cada parte. A arquitetura proposta usa técnicas de orientação a objetos para obter a flexibilidade necessária a este tipo

de integração. O primeiro passo é obter um isolamento completo entre interface e os módulos executivos de cada parte. A figura abaixo mostra um esquema de isolamento. A camada de isolamento provê um meio de se acoplar a interface com o diversos módulos, evitando a interferência mútua entre eles.

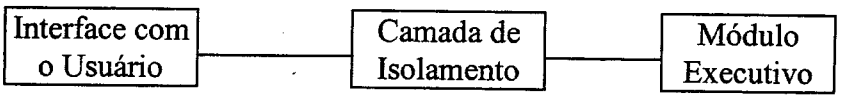


Fig. 3: Camada de isolamento

A presente arquitetura divide a camada de isolamento em duas partes principais, uma para intercâmbio de controle e outra para dados. O controle transfere comandos entre a interface e o módulo executivo. A ligação de dados provê uma correspondência entre a representação interna do módulo e a apresentação visual dos dados.

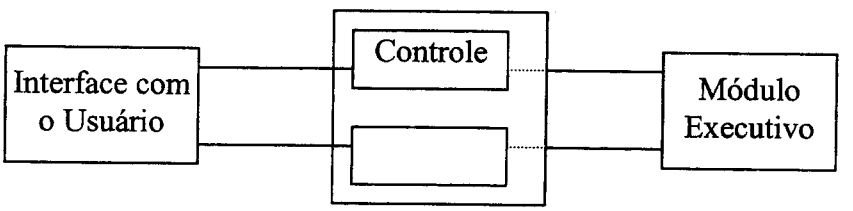


Fig. 4: Canais de controle e dados da camada intermediária

Para facilitar a adaptação dos variados módulos, a camada de isolamento é dividida entre a interface e o módulo. O controle pode ser adaptado para executar os comandos específicos de cada módulo mas, dentro da camada de isolamento, os controles conversam de modo padronizado.

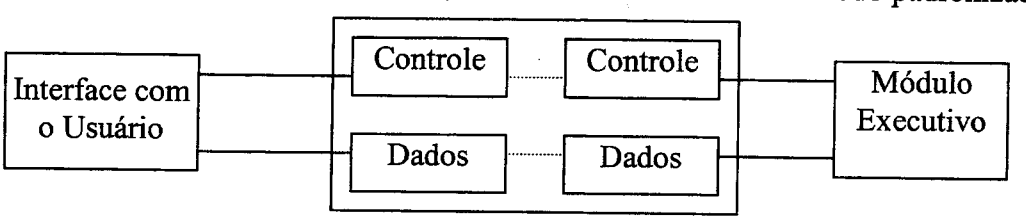


Fig. 5: Partição da camada intermediária

A separação entre controle e dados permite que se tenha diversas ligações para fazer a correspondência entre diferentes tipos de dados. Por exemplo: um digrama esquemático pode ser anotado com documentação para cada bloco funcional e cada representação (texto, gráfica) pode ser apresentada em janelas distintas.

4. Interface com o usuário

Para testar a modularidade da arquitetura, nós implementamos o protótipo de um editor gráfico. Este protótipo deverá ser integrado à nova versão da ferramenta de ensino de projetos de VLSI do Sistema TEDMOS. A interface e módulo funcional foram desenvolvidos independentemente por dois estudantes. A interface foi inicialmente testada acoplada a um módulo funcional muito simples. Em um segundo passo a mesma interface foi integrada a uma implementação de alto desempenho funcional sem qualquer necessidade de mudança de código.

No protótipo implementado são apresentadas duas formas de visualização de um circuito integrado: uma gráfica (através de retângulos) e uma textual. A representação gráfica foi organizada de forma a

facilitar as operações ocorridas na interface, uma classe retrata somente a parte do modelo que está sendo efetivamente alterada e a outra descreve o modelo como um todo. Os eventos do *mouse* na área de desenho são repassadas para a primeira, enquanto que outras operações, como *zoom* e *scroll*, são transferidas para a segunda.

A fim de estabelecer um isolamento ainda maior, a vista foi desenvolvida sob a forma de um *pattern* conhecido como *bridge*. Ele garante a independência entre a interface do objeto e sua implementação. Para isto foram criadas duas "árvores de classes" equivalentes, sendo que uma é totalmente abstrata, só representando a interface dos objetos, e a outra responsável pela implementação de suas funcionalidades. As classes abstratas referenciam as que contém a implementação. A utilização deste *pattern* faz com que alterações nesta parte do sistema, possam ser feitas em somente uma das duas "árvores de classes", sem que haja necessidade de se alterar a outra.

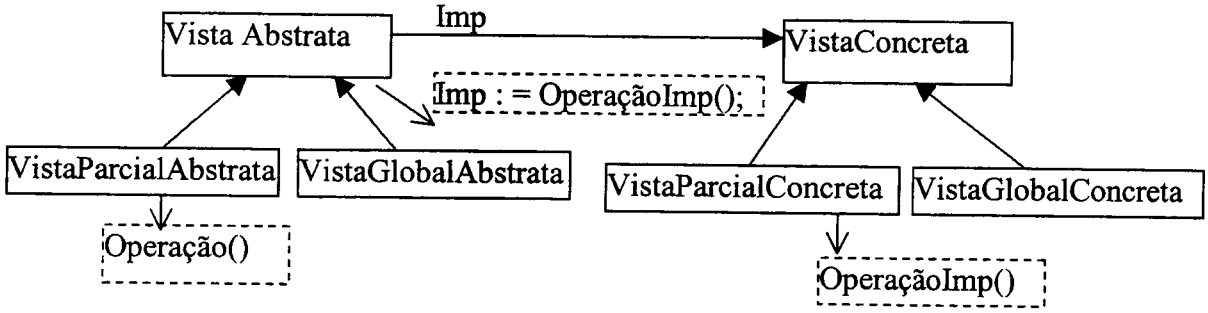


Fig.6 : *Pattern Bridge*

Os controladores também são divididos segundo suas funcionalidades – controlar a interface ou o modelo. O primeiro foi especializado ainda mais, definindo uma classe responsável pelos eventos do *mouse* e outra pelas funções do aplicativo. Assim, qualquer ação do *mouse* é tratada por um objeto de vista e repassada, no momento apropriado ao controle de aplicação. Este controle comanda operações como inserção, deleção e seleção de retângulos, utilizados na representação do circuito integrado. O controlador de aplicações gera comandos para o controlador de modelo, que interpreta esses comandos e provê a execução das operações necessárias para atender a requisição da interface.

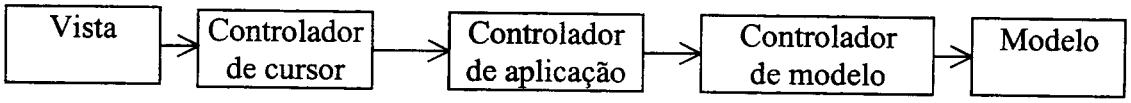


Fig.7 : Caminho das mensagens

Ao encaminhar uma operação ao controlador de modelo, o controlador de aplicação comunica à vista que mandou as mensagens de solicitação da alteração do modelo. Como o fluxo de mensagens é unidirecional, cabe as vistas se atualizarem, pedindo diretamente ao modelo um *refresh*. Esta comunicação só é possível graças ao um objeto intermediário que possibilita que mensagens, provenientes da vista, cheguem ao modelo. Esta classe também é responsável por garantir a integridade do sistema, ou seja, que todas representações da interface correspondam ao estado corrente dos módulos executivos. Isto pode ser feito utilizando-se do *pattern Observer*. Este objeto intermediário passa a ser observado por todas as vistas; assim, qualquer mensagem direcionada a ele é imediatamente avisada a todos os seus observadores. Por isto, quando uma das vistas sofre uma modificação que acarreta uma mudança nos dados da estrutura, o objeto que está sendo observado avisa a todas as outras vistas que elas precisam ser atualizadas.

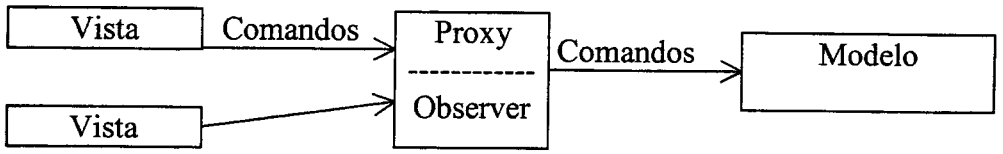


Fig. 8 :*Pattern Observer / Proxy*

Os controladores estão definidos segundo um *pattern* conhecido como *Chain of Responsibility*. Eles são organizados em uma cadeia onde cada controlador aponta para o próximo. Uma mensagem, quando direcionada a esta cadeia, percorre toda sua extensão até que seja tratada por um de seu integrantes. A vantagem proveniente desta implementação é que não se precisa definir, anteriormente, o destino de um comando.

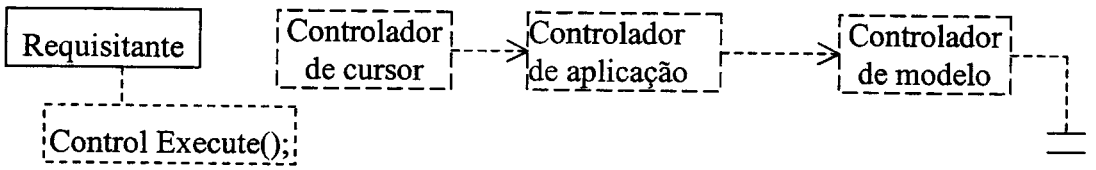


Fig. 9: *Pattern Chain of Responsibility*

Na arquitetura proposta, cada controlador será implementado segundo o *pattern State*. Este *pattern* permite que um objeto modifique o seu comportamento quando seu estado interno é modificado. Para tal, é construída uma classe abstrata que representa os estados que o controlador pode assumir, encapsulando, assim, o comportamento associado ao estado corrente. Este comportamento é implementado em suas subclasses concretas - cada uma delas representando um possível estado do controlador. Esta classe abstrata é referenciada por outra que define um interface compatível à vista. Esta mantém sempre uma instância de uma subclasse concreta. O controlador de cursor, por exemplo, terá subclasses que representam estados como "*Rubberbanding*"(enquanto os retângulos estão sendo desenhados) e "*CursorReady*" (quando o retângulo terminou de ser editado.) . Já o controlador de aplicação terá subclasses que implementam as operações necessárias em estados como inserção, seleção e cópia de dados no modelo.

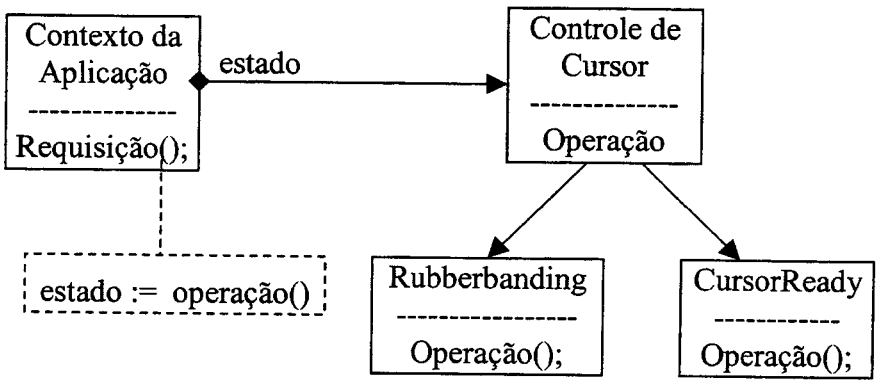


Fig .10: *Pattern State*

No entanto, ainda havia a necessidade de isolar a vista dos controladores para garantir a flexibilidade do sistema. Um objeto definido segundo o *pattern Proxy*, é responsável por esta separação. ele funciona "assumindo o papel" do objeto que deveria receber a mensagem. Este objeto implementa este *pattern* bidirecionalmente, ou seja, incorpora o papel de controlador quando a mensagem vem da vista e vice-versa. Um *Action* define uma ação que deve ser submetida ao modelo,

como, por exemplo, uma inserção, deleção ou seleção. Ele recebe as mensagens da vista, encaminhando-as aos controladores juntamente com uma instância de si mesmo, que é passada como parâmetro. Quando as mensagens vêm em sentido oposto, o *Proxy* as recebe como se fosse um objeto de interface e as direciona para quem efetivamente deve tratá-las.

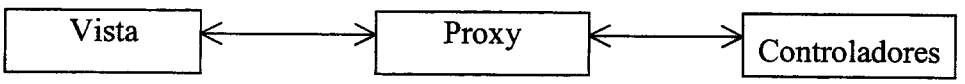


Fig. 11: *Pattern Proxy*

Na proposta, este objeto irá implementar o *pattern Command*. Este *pattern* encapsula um pedido como um objeto. É feita uma classe abstrata que declara a interface de uma operação chamada Execute. As subclasses concretas especificam um par Receptor-Estado e, na implementação do Execute, repassam o pedido a uma instância do receptor. O cliente é o objeto responsável por criar o objeto e setar o seu receptor. Em nosso sistema, a princípio o cliente será a vista. Ela criará comandos simples, como "*MouseDown*" e "*MouseUp*", que por sua vez serão passados aos controladores. Estes passarão a ser então os clientes responsáveis por criar comandos mais complexos, como inserção e deleção, dependendo do estado em que a aplicação se encontra.

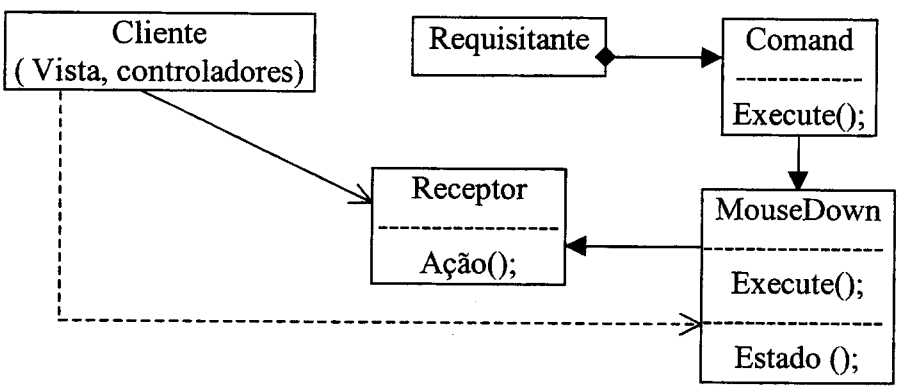


Fig.12 : *Pattern Command*

A utilização deste *pattern* possibilita a realização de operações de *UNDO* e *REDO*. Para tanto, as subclasses concretas terão que guardar informações extras como qualquer campo do receptor que pode ter seu valor alterado com a execução do pedido. Assim, a aplicação deverá guardar os comandos para que se possa desfazer operações. O número de *UNDO*'s e *REDO*'s vai depender da quantidade de comandos armazenados.

Como foi mencionado a princípio, desejava-se obter um isolamento para o intercâmbio de controle e dados. Sendo assim, o primeiro é transferido, conforme explicado acima, sempre indo da vista em direção ao modelo, enquanto que os dados são enviados nas duas direções. Como sabemos, um objeto intermediário que é observado pelas vistas é responsável por esta comunicação. Para permitir o canal de transferência de dados, aplicamos o *pattern Proxy* fazendo-o interagir com um objeto do modelo que implementa o *pattern Facade*. Este *pattern* define uma "fachada", permitindo que a vista não precise sequer conhecer a interface dos objetos do modelo, garantindo ainda mais o isolamento entre as partes. As mensagens são mandadas ao *Facade* e ele trata de "traduzi-las" de forma a serem tratadas pelo modelo, ou seja: ele as repassa usando chamadas equivalentes às recebidas. Os dados são transferidos como objetos que podem assumir diferentes formas. Este objeto pode encapsular um retângulo, uma *string*, um inteiro ou um ponto. Utilizando-se do polimorfismo, a vista transfere esse objeto ao *facade*, que "entende" a mensagem e manda inserir o dado no modelo.

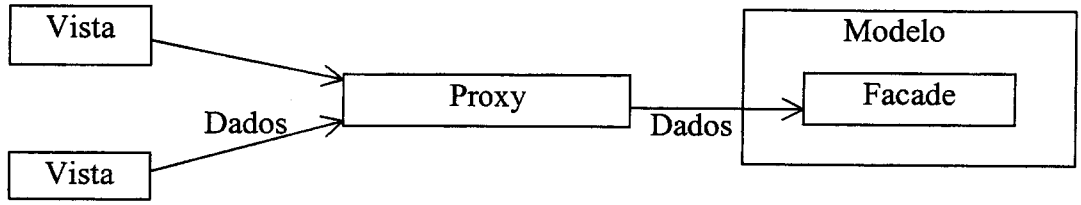


Fig.13 : *Pattern Facade*

A transferência, em sentido oposto, é feita no momento em que o *refresh* é pedido. As vistas, com o objetivo de se atualizarem, mandam uma mensagem *Refresh*, através do *Facade*, ao modelo. O modelo, através da aplicação de *pattern* como o *iterator* e o *decorator*, se encarrega de atualizar as vistas.

O sistema também faz uso de um outro *pattern* conhecido como *Factory*. Ele concentra a construção de todos os objetos. Estas "fábricas" contém procedimentos que evocam todos os construtores, como também estabelecem as relações entre os objetos – como é o caso da *Chain of Responsibility*. Assim, ao invés de se instanciar um objeto em diferentes pontos do programa, pede-se que a "fábrica", responsável por ele, o faça. Se no futuro forem criadas novas classes para substituir outras já existentes, somente a chamada dos construtores nas "fábricas" será modificada. Para uma melhor transparência do sistema, foram criadas "fábricas" de controle, de vista e de modelo.

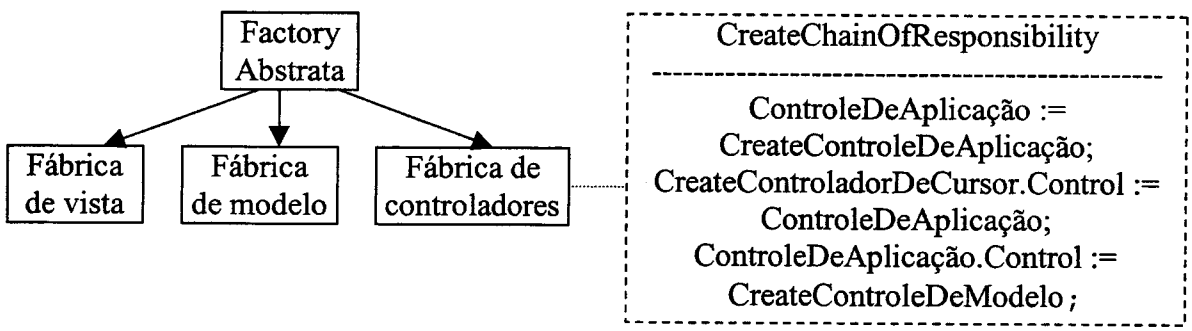


Fig. 14 : *Pattern Factory*

Observando-se que vários dos objetos utilizados deveriam ser únicos, achou-se interessante a utilização do *pattern Singleton*. Para que o sistema se torne confiável não podemos permitir, por exemplo, duas instâncias de um controlador. Aplicando-se um *singleton* a ele garantimos que só uma instância será criada e que só ela receberá as mensagens correspondentes, não se correndo o risco do sistema tornar-se inconsistente.

5. Protótipo de Plano físico: Modelo

O Modelo do TEDMOS retrata a representação interna de um circuito eletrônico. Este modelo deve ter flexibilidade suficiente para se adequar às diversas representações de um projeto VLSI, e suportar algoritmos complexos. Além disso, deve ter uma estrutura desenvolvida de tal forma que possibilite a integração com módulos de diferentes desenvolvedores. Para isso foram usados *patterns* que promovem uniformidade e isolamento ao modelo. O isolamento promovido pela orientação a objetos facilitou a implementação com objetos distribuídos.

O TEDMOS retrata a representação física de um circuito integrado sob a forma de retângulos. Porém, não é a desta forma que os dados são armazenados internamente. Seria muito custoso varrer a estrutura, principalmente porque podem haver milhões de retângulos, e esta operação é freqüentemente realizada. Como solução foi adotado o conceito de *Spans*. Um *Span* é um intervalo nas coordenadas X e Y. Necessitando-se de, pelo menos, um *X-Span* e um *Y-Span* para obter-se um retângulo. Uma coleção de retângulos representa um certo material usado na fabricação de circuitos e é descrita em nosso sistema por um *Layer*. Os *Spans* são definidos dividindo-se o *Layer* em ambos os eixos, e *Spans* consecutivos no eixo X são unidos em um só. A coleção de *Y-Spans* e *X-Spans* é chamada de Plano.

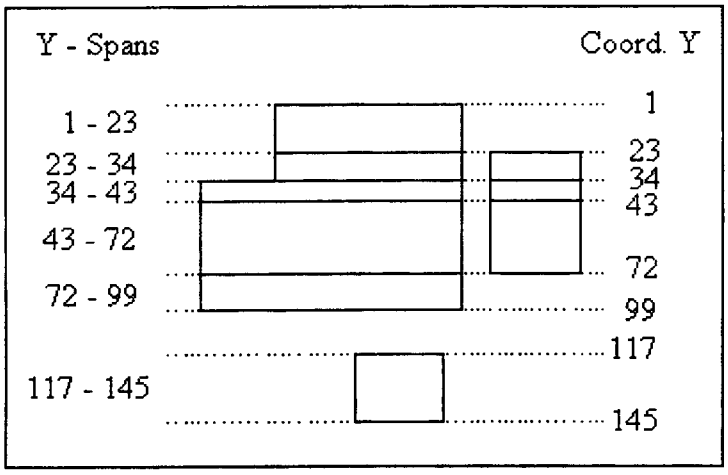


Fig. 15 : *Spans*

A varredura é facilitada porque os *Y-Spans* são mantidos numa coleção ordenada, e cada um deles mantém sua própria coleção de *X-Spans*. Com o uso desta estrutura, algoritmos de busca têm complexidade O(n), o que é uma grande vantagem porque é linear. O tempo consumido em operações realizadas num *Layer*, que envolve uma grande quantidade de dados, pode ser otimizado através desta nova abordagem.

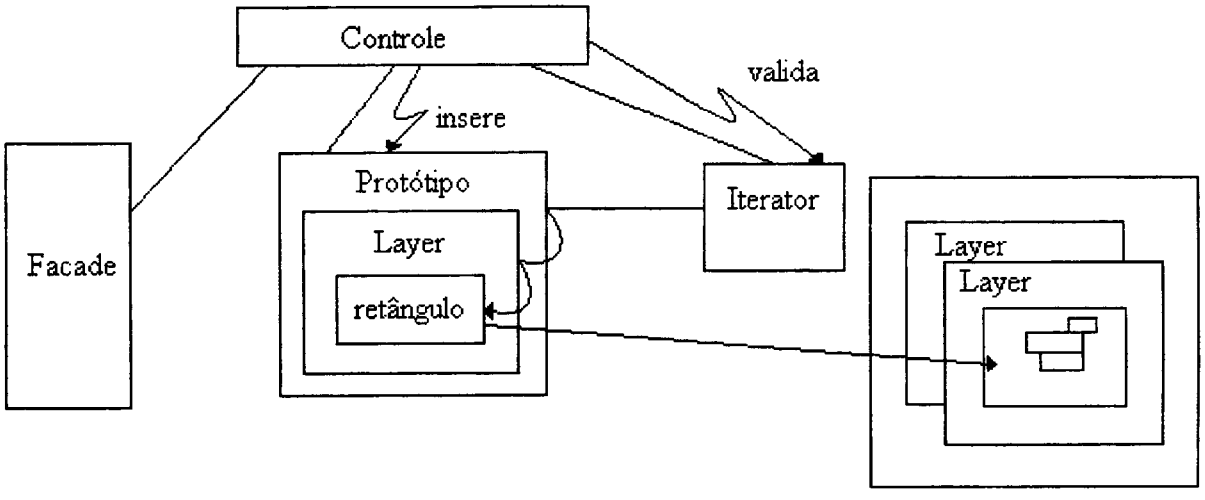


Fig. 16 : Inserção de um retângulo

Já que a arquitetura garante a independência entre modelo e vista, este módulo funcional pode substituir estruturas que não são otimizadas. O Modelo recebe os comandos do Controle na forma de mensagens. O *Layer*, que implementa o *pattern Prototype*, instancia um protótipo, ou seja, uma cópia de si mesmo. Os elementos do *Layer*, que sofrerão modificações provocadas pela execução das mensagens, são inseridos no protótipo. Posteriormente, estes elementos são passados ao Plano, onde são realizadas as modificações. Para que este procedimento torne-se mais claro, a seguir, é explicado como realiza-se a operação de inserção de um retângulo no *Layer*.

A mensagem, que corresponde a ação do usuário de desenhar um retângulo na tela, é enviada ao Controle da Vista. Este remete a mensagem ao *Facade*, que a repassa ao Controle do Modelo. Este Controle envia a mensagem ao *Layer* para que este instancie uma cópia - o protótipo. O retângulo é inserido no protótipo. Quando a mensagem de inserção é validada pelo Controle, o objeto que implementa o *pattern Iterator* percorre o protótipo. O *Iterator* é programado para enumerar uma coleção de objetos. Primeiramente, ele aponta o primeiro objeto, depois, conforme é solicitado, retorna cada elemento da coleção até encontrar o último. O *Iterator* retorna cada elemento do protótipo (neste caso, um único elemento) ao Plano. Este insere cada elemento na sua estrutura. É o Plano que se encarrega de converter o elemento recebido em *Spans*.

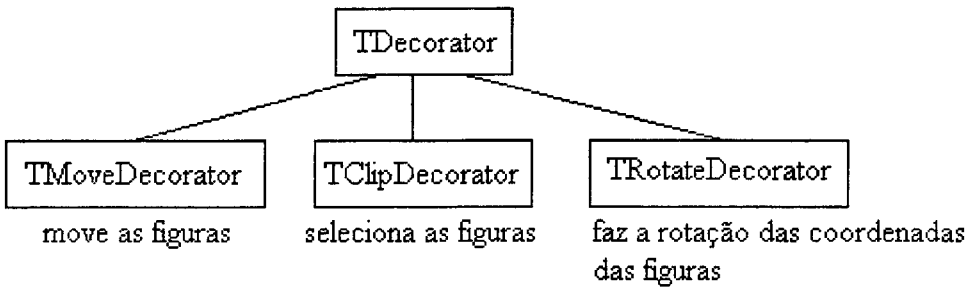
Na arquitetura de modelo-vista-controle adotada, as mensagens são passadas apenas na direção da Vista para o Modelo. As modificações ocorridas no Modelo são notificadas à Vista, que responderá com a solicitação de *refresh*. Quando este pedido é feito, são retornados dois objetos, estes implementam, respectivamente, os *patterns Iterator* e *Visitor*, colocando-os sob o comando da Vista. A Vista solicita ao *Iterator* que retorne o próximo elemento a ser representado na tela.

Para que a operação de representar cada elemento do *Layer* na Vista seja executada, usamos o *pattern Visitor*. O *Visitor* representa uma operação a ser realizada sobre os elementos da estrutura de um objeto. Isto significa que o *visitor* executa uma mesma operação para os diferentes tipos de elementos do Modelo. Neste caso, a operação é a de pintura na tela. A classe responsável por esta operação - o *visitor*- é programada para examinar a representação interna do *Layer* e converter cada elemento na sua apresentação visual correspondente.

A representação visual pode ser feita de diferentes maneiras. Como por exemplo, rotação de figuras do *layout*, ou apenas mudá-las de localização. Para isto foi usado o *pattern Decorator*. Este consiste numa alternativa flexível para o uso de subclasses, já que as responsabilidades são adicionadas ao objeto dinamicamente. Cada classe do *Decorator* implementa uma funcionalidade básica, que neste caso diz respeito a forma de realizar a representação visual.

Como exemplo podemos citar a operação de selecionar uma área, que contém vários elementos, arrastar esta seleção para um determinado lugar e, em seguida realizar uma rotação da área que foi movida. Inicialmente poderíamos criar uma classe para executar cada combinação destas funções. Como o número de combinações possível é grande, cresceria demasiadamente o número de classes. Usando *decorator*, evitamos este tipo de situação. Pois é necessário apenas criar as classes, descendentes do *decorator*, responsáveis por: pintar uma seleção de figuras, pintar figuras deslocando-as de sua posição original e pintar figuras aplicando uma rotação nas suas coordenadas originais.

Com estas três classes acima citadas, é possível executar apenas cada uma das três funcionalidades ou qualquer combinação delas. Isto é feito adicionando-se ao objeto as funcionalidades dinamicamente:




```
Painter := TRotateDecorator.Ceate ( axe,
                    TMoveDecorator.Create ( XOffset, YOffset,
                        TClipDecorator.Create ( Rect, TDecorator.Create ) ) );
```

Fig. 17 : Pattern Decorator

Observando-se a linha de código - em Pascal - acima, temos que "*Painter*" é a instância de um objeto com três funcionalidades, que foram unidas dinamicamente. E a classe *TDecorator* implementa a funcionalidade básica, neste caso, pintar figuras na tela. Como foi exemplificado, unimos funcionalidades em tempo de execução, sem ter que para isso aumentar o número de classes.

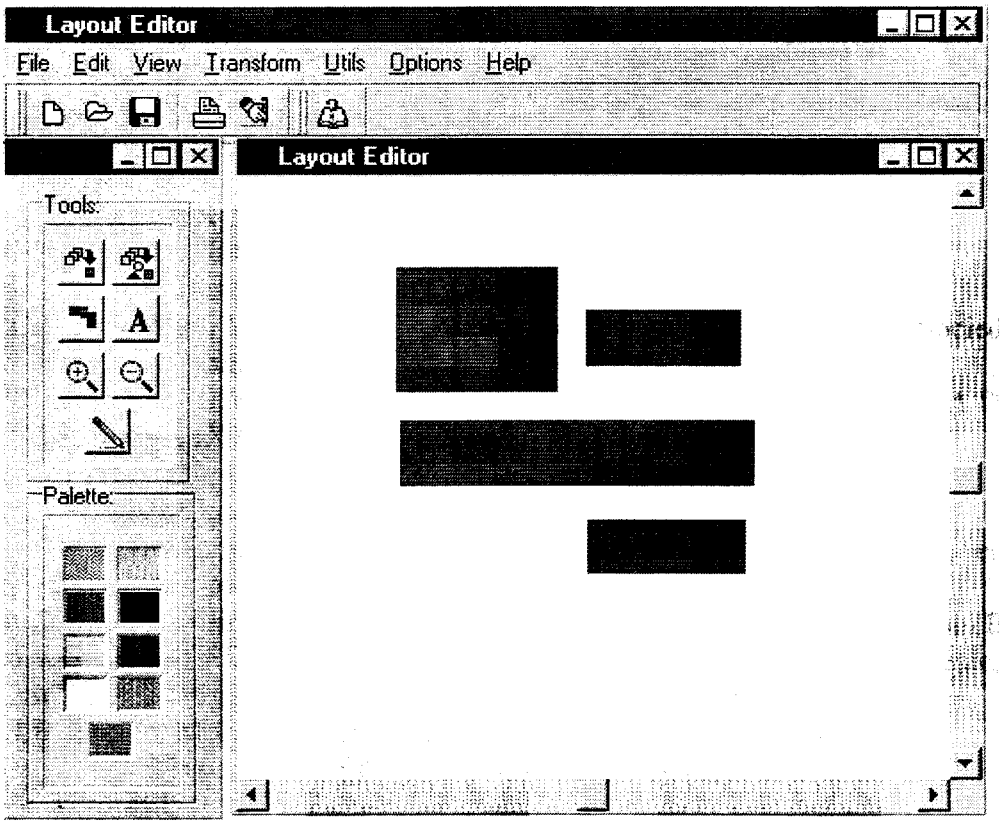


Fig. 18: Implementação do Protótipo

Assim foi obtida uma estrutura modular para modelos físicos e de fácil acoplamento com a interface, com potencial de distribuição em redes de computadores. Isto permitiria, por exemplo, delegar a execução dos algoritmos a diferentes máquinas, tornando o sistema mais eficiente e seguro.

6. Conclusão

O processo de projetar um circuito integrado é complexo o suficiente, de tal modo que qualquer outra dificuldade adicionada é indesejável. O artigo sugere uma forma de integração entre ferramentas que não repasse a heterogeneidade de tratamento entre os módulos para a interface. Além disso, propõe uma arquitetura com facilidade de adaptar-se rapidamente a exigências novas.

A arquitetura modular projetada para o TEDMOS está preparada para se acoplar facilmente a módulos independentes. Até mesmo se a arquitetura não está embutida no módulo, um adaptador funcional pode trazer módulos legados ou partes estrangeiras para fundirem-se ao sistema. O uso de técnicas avançadas de orientação a objetos pavimentou o caminho para construir, em algumas semanas, um protótipo que demonstra a flexibilidade da proposta. O caminho está agora aberto a experiência com objetos distribuídos. O mecanismo de isolamento básico implementado na arquitetura cumpre as exigências para encapsulamento de objeto remotos. Na próxima experiência, o módulo funcional será distribuído por várias máquinas e o ganho de desempenho será avaliado.

7. Referências:

[1] Furlan, J.D., "Modelagem de Objetos através da UML – The Unified Modeling Language", MAKRON Books, 1998.

[2] Gamma, E., Helm, R., Johnson, R., Vlissides, J., "Design Patterns : Elements of Reusable Object - Oriented Software", Addison-Wesley, 1998.

[3] Eddon, G., Eddon, H., "Inside Distributed COM" – Microsoft Press, 1998.

[4] Weste, N., Eshraghian, K., "Principles of CMOS VLSI Design", Addison-Wesley,1988.

[5] Mead, C., Conway, L., "Introduction to VLSI Systems", Addison-Wesley,1980.

[6] "Developer's Guide - Borland - Jbuilder 2", Borland, 1998.

[7] Oliveira, C.E.T. e Anido, M.L., "TEDMOS para Windows", IX Congresso da Sociedade Brasileira de Microeletrônica, Campinas, pp. 65-73, Agosto, 1994.

[8] Nunes, R.B., Anido, M.L. e Oliveira, C.E.T., "Circuit Verification Using Spans – A DataStructure with O(n) Algorithms", IX Congresso da Sociedade Brasileira de Microeletrônica, Campinas, pp. 65-73, Agosto, 1994.

[9] Nunes, R.B., Anido, M.L. e Oliveira, C.E.T., "A New Approach to Perform Circuit Verification Using O(n) Algorithms" , IEEE Procedings of the EUROMICRO'94 conference, Liverpool, IEEE Computer Society Press, pp. 428-434, 1994.

[10] Alcântara, J.M.S., Oliveira, C.E.T. e Anido, M.L., "A Novel Circuit Extration Tool Based on X-Spans and Y-Spans", IEEE Procedings of the 21st EUROMICRO Conference, Prague, Tcheck Republic, Setembro, 1996.

[11] Nunes, R.B., Anido, M.L. e Oliveira, C.E.T., "A New Approach to Perform Circuit Verification Using Spans" , IEEE Procedings of the 38th Midwest Symposium on Circuits and Systems, Agosto, Rio de Janeiro, Brasil, 1995.