UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

Blade: Um Editor de Esquemáticos Hierárquico voltado à Colaboração

por

LISANE BRISOLARA DE BRISOLARA

Dissertação submetida à avaliação,
como requisito parcial para a obtenção do grau de Mestre
em Ciência da Computação

Prof. Ricardo Reis
Orientador

Porto Alegre, dezembro de 2002.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Brisolara, Lisane Brisolara de
Blade: Um Editor de Esquemáticos Hierárquico voltado à Colaboração / por Lisane Brisolara de Brisolara. – Porto Alegre: PPGC da UFRGS, 2002.

136 f.: il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR-RS, 2002. Orientador: Reis, Ricardo Augusto da Luz.

1. Microeletrônica. 2. ferramentas de CAD. 3. Editor de esquemáticos. 4. Editor de diagramas. 5. Colaboração I. Reis, Ricardo Augusto da Luz. II. Título.

Aos meus pais, Maria Luci e Arlindo, que sempre acreditaram na minha capacidade.

Agradecimentos

Aos professores do Grupo de Microeletrônica pelo compartilhamento de seus conhecimentos. Em especial, ao meu orientador o professor Ricardo Reis, pelo apoio e a confiança depositada em mim, bem como pelo incentivo à produção científica e à participação nos eventos da comunidade acadêmica.

Aos colegas de projeto, Leandro Indrusiak, Sandro Sawicki e Emerson Barbiero, pelas contribuições para o desenvolvimento deste trabalho. Em especial, ao Leandro pelo apoio, pelas cobranças e pela sua amizade, e ao Sandro pelos trabalhos cooperativos desenvolvidos ao longo do curso e pelo carinho.

Gostaria de agradecer também, aos colegas da UFRGS, que mais do que colegas, tornaram-se também meus amigos. Agradeço o incentivo e a amizade.

Aos colegas do grupo *voleiGME* e do Coral da Informática, que me propiciaram muitos momentos agradáveis.

Não poderia deixar de agradecer a meus pais, Maria Luci e Arlindo, as minhas irmãs, Elisa e Cibele, e demais familiares pelas palavras de incentivo e pelo carinho. Além, da paciência e compreensão em minhas ausências.

A todos os amigos pelo carinho e amizade. Em especial, a Milena pelo apoio e companheirismo e ao Marcio pelos conselhos e pelas palavras de incentivo.

A todos os cidadãos brasileiros que contribuem para o desenvolvimento da pesquisa no país, muitos deles sem acesso às universidades e sem conhecer o valor da pesquisa desenvolvida nestas instituições.

A Deus, como esquecer dele nesta hora? Foi ele quem me indicou este caminho, no qual encontrei dificuldades, é verdade, mas também encontrei todas essas pessoas que me deram o apoio necessário para vencer todos os obstáculos e alcançar meus objetivos.

Sumário

Lista de Abreviaturas

ACM	Association for Computing Machinery
API	Application Programming Interfaces
CAD	Computer Aided Design
CSCW	Computer Supported Cooperative Work
CI	Circuito Integrado
EAD	Electronic Aided Design
EDIF	Electronic Design Interchange Format
FPGA	Field Programmable Gate Arrays
FSM	Finite State Machine
GUI	Graphical User Interface
HDL	Hardware Description Language
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IP	Intellectual Property
MVC	Model View Controller
OO	Object-Oriented
OODMS	Object-oriented Database Management System
PC	Personal Computer
PCB	Printed Circuit Board
RDBMS	Relational Database Management System
RTL	Register Transfer Level
SDL	System Description Language
TCP/IP	Transmission Control Protocol / Internet Protocol
UML	Unified Model Language
VHDL	Very High Integrated Circuits HSIC Hardware Description Language
VLSI	Very Level System Integration
WWW	World Wide Web

Lista de Figuras

Lista de Tabelas

Resumo

Este trabalho apresenta a proposta de um editor de diagramas hierárquico e colaborativo. Este editor tem por objetivo permitir a especificação colaborativa de circuitos através de representações gráficas. O Blade (*Block And Diagram Editor*), como foi chamado, permite especificações em nível lógico, usando esquemas lógicos simples, bem como esquemas hierárquicos. Ao final da montagem do circuito, a ferramenta gera uma descrição textual do sistema num formato *netlist* padrão. A fim de permitir especificações em diferentes níveis de abstração, o editor deve ser estendido a outras formas de diagramas, portanto seu modelo de dados deve ter flexibilidade a fim de facilitar futuras extensões.

O Blade foi implementado em Java para ser inserido no Cave, um ambiente distribuído de apoio ao projeto de circuitos integrados, através do qual a ferramenta pode ser invocada e acessada remotamente. O Cave disponibiliza um serviço de colaboração que foi incorporado na ferramenta e através do qual o editor suporta o trabalho cooperativo, permitindo que os projetistas compartilhem dados de projeto, troquem mensagens de texto e, de forma colaborativa, construam uma representação gráfica do sistema.

Objetivando fundamentar a proposta da nova ferramenta, é apresentado um estudo sobre ferramentas gráficas para especificação de sistemas, mais especificamente sobre editores de esquemáticos. A partir dessa revisão, do estudo do ambiente Cave e da metodologia de colaboração a ser suportada, fez-se a especificação do editor, a partir da qual implementou-se o protótipo do Blade.

Além do editor, este trabalho contribuiu para a construção de uma API, um conjunto de classes Java que será disponibilizado no Cave e poderá ser utilizado no desenvolvimento de novas ferramentas. Foram realizados estudos sobre técnicas de projeto orientado a objeto, incluindo arquiteturas de software reutilizáveis e padrões de projeto de software, que foram utilizados na modelagem e na implementação da ferramenta, a fim de garantir a flexibilidade do editor e a reusabilidade de suas classes.

Este trabalho também contribui com um estudo de modelagem de primitivas de projeto de sistemas. No modelo orientado a objetos utilizado no editor, podem ser encontradas construções muito utilizadas em diferentes ferramentas de projeto de sistemas, tais como hierarquia de projeto e instanciação de componentes e que, portanto, podem ser reutilizadas para a modelagem de novas ferramentas.

Palavras-chave: Microeletrônica, ferramentas de CAD, editor de esquemáticos, editor de diagramas, colaboração.

TITLE: "BLADE: A HIERARCHICAL DIAGRAM EDITOR TARGET TO COLLABORATION"

Abstract

This work presents a proposal of a hierarchical diagram editor for collaborative design. This tool has as objective to allow collaborative specification of circuits through graphical representations. This editor was called Blade (*Block And Diagram Editor*) and it allows specifications in logic level, using simple logic schemas or hierarchical schemas. After the circuit assemblage, the tool generates a textual description of the diagram using a netlist default format. In order to allow a specification in different abstract levels, this editor should be extended to handle other kinds of diagrams. Thus, its data model must have flexibility, facilitating future tool extensions.

Blade was implemented using the Java language and it was plugged into a distributed environment called Cave. This environment integrates several distributed CAD tools that are used to aid IC designers. Blade, as the other Cave's tools, can be accessed remotely. A collaborative service based on Pair Programming collaborative methodology was incorporated into Cave. Through this service, the Blade editor supports collaborative work, allowing designers to share design data, to exchange text messages and build cooperatively a graphical representation of a system.

Before the implementation of the Blade prototype, a specification of the tool was defined. This specification had as basis a study about graphical tools, more specifically schematic editors, a study about the Cave and also a study about the collaboration methodology to be supported. All these studies are presented in this work.

Besides the construction of the editor, this work contributed for the construction of an API, a set of Java classes that will be available in the Cave and that can be used in the development of new tools. Studies about techniques of object-oriented design, software reuse and design patterns were done in order to guarantee the editor flexibility and the reuse of its classes.

This work also contributes with a study about modeling design primitives. In the object-oriented approach used in the editor, constructions can be found that are widely used in several design tools, such as: design hierarchy and component instantiation. These constructions can be reused to build new tools.

Keywords: Microelectronics, CAD tools, schematic editor, diagram editor, collaboration.

1 Introdução

A síntese de um circuito integrado é descrita por uma seqüência de transformações incrementais aplicadas a uma descrição inicial da funcionalidade do circuito. A seqüência de operações aplicadas a esta descrição inicial determina um fluxo de projeto, no qual são utilizadas ferramentas de CAD (*Computer Aided Design*) que permitem a transformação incremental de uma descrição de projeto em outra mais detalhada ao longo do processo.

Circuitos integrados de grande complexidade não teriam sido desenvolvidos sem ferramentas computadorizadas que apoiassem os projetistas em todas as fases do projeto. Na automação de projetos são usadas diferentes ferramentas de CAD para as diversas fases de um projeto. Captura de esquemáticos, edição de leiaute, simulação, síntese de alto nível e verificação são exemplos de etapas utilizadas em um fluxo de projeto de CIs e que demandam ferramentas específicas.

O processo de síntese começa por uma captura da descrição inicial do circuito, ou seja, de sua especificação. O nível de detalhamento desta descrição inicial depende do projetista e da complexidade do circuito. As ferramentas nas quais o usuário pode editar uma descrição de entrada aceita por uma ferramenta de CAD integrada ao fluxo de projeto podem ser classificadas como ferramentas de edição e captura.

Uma ferramenta de edição e captura permite a entrada de uma descrição inicial do circuito ou a otimização de uma solução obtida através de uma outra ferramenta. Como ferramentas de edição e captura destacam-se os editores de leiaute, editores de esquemáticos e os editores de descrições de hardware que usam linguagens como VHDL e *Verilog*.

Os editores de leiaute foram muito utilizados nas etapas de especificação, porém hoje em dia, devido à complexidade dos sistemas, esta classe de ferramentas é normalmente utilizada para a visualização do leiaute gerado automaticamente por uma ferramenta de síntese, permitindo que o projetista analise e faça possíveis otimizações na solução obtida automaticamente, ou para a edição de células de pequena complexidade a serem geralmente inseridas em uma biblioteca.

Os editores de esquemáticos foram, e ainda são muito utilizados na especificação de sistemas elétrico / eletrônicos simples, através de componentes elétricos básicos (capacitores, resistores, transistores). Nestes casos, gerando descrições a nível elétrico, ou ainda, através de portas lógicas, gerando uma descrição a nível lógico.

A abordagem baseada em captura de esquemáticos para o projeto de circuitos integrados também foi muito adotada. Porém, a partir dos anos 90, com o aumento da

complexidade dos circuitos e com o conseqüente avanço nas ferramentas de automação, projetistas começaram a utilizar linguagens de programação para especificar sistemas complexos. Surgiram, então, as chamadas linguagens de descrição de hardware (HDL, do inglês, *Hardware Description Language*) [KUR 97]. As linguagens mais conhecidas são VHDL e *Verilog* [GER 99].

1.1 Motivação

Apesar do difundido e intenso uso de linguagens textuais para a entrada de projeto, pode-se vislumbrar possibilidades interessantes no uso de ferramentas que permitam a especificação gráfica de sistemas, tais como editores de esquemáticos, e que ainda não são totalmente exploradas pelos sistemas de CAD comerciais em se tratando de descrições com alto nível de abstração. Este trabalho apresentará uma nova classe de ferramentas objetivando explorar algumas destas possibilidades.

Uma representação visual de um projeto facilita o seu entendimento e acelera a montagem da descrição de um sistema, assim como, uma idéia representada na forma de um diagrama é mais facilmente compartilhada entre projetistas do que uma representação textual, facilitando a colaboração entre eles. Observa-se uma forte tendência para o uso de linguagens visuais no projeto de sistemas. Segundo Minas [MIN 95], notações gráficas e técnicas baseadas em diagramas são freqüentemente utilizadas em uma variedade de domínios. Na área de hardware / software *codesign*, por exemplo, pode ser observado o uso intenso de uma variedade de diagramas na especificação de sistemas.

Um circuito simples pode ser descrito em nível elétrico ou lógico através de um editor de esquemáticos. Porém, o projeto de circuitos complexos, normalmente exige a utilização de descrições em um nível mais alto de abstração, bem como o uso de modularização e hierarquia. Na especificação de um circuito complexo, através de um editor de esquemáticos hierárquico, podem ser utilizados blocos para representar graficamente os módulos que compõem o sistema. Assim, tem-se uma visualização de todo o sistema que pode ser compartilhada entre projetistas. Nesta abordagem, após a descrição do sistema através de blocos interconectados, deve-se definir o funcionamento destes blocos. No caso de circuitos simples, o seu funcionamento pode ser descrito através de esquemas lógicos ou elétricos. Porém, no caso de circuitos mais complexos, normalmente são utilizadas descrições em um nível mais alto de abstração, por exemplo, a nível comportamental.

Atualmente, muitos sistemas de CAD conjugam editores de descrições em HDL com editores de esquemático, de modo a obter uma descrição mista onde alguns dos módulos do esquemático podem ser descritos textualmente através de HDLs e a montagem final é feita graficamente no editor de esquemáticos [REI 2000]. Existem ferramentas comerciais de edição de esquemáticos que permitem a especificação gráfica de sistemas não somente através de diagramas a nível lógico, mas usando outras formas

de diagramas, tais como máquinas de estados finitos (FSM). Nestas ferramentas, ao final da descrição da FSM, é gerada uma descrição em uma linguagem HDL correspondente à máquina definida.

Analisando as ferramentas comerciais disponíveis e recentes investigações na área de CAD, observa-se o interesse no desenvolvimento de uma ferramenta que permita a especificação de sistemas utilizando diferentes representações gráficas/textuais, além de diferentes níveis hierárquicos. Permitindo descrições dos módulos do sistema em diferentes níveis de abstração, dependendo de suas complexidades e da experiência do projetista. Assim, descrições a nível lógico podem ser utilizadas para a especificação de módulos simples do sistema e descrições a nível comportamental para os módulos mais complexos. O mais alto nível da hierarquia de visualização do sistema pode ser representado por um diagrama composto por blocos conectados entre si, representando os módulos do sistema e as ligações existentes entre eles, ou por um único bloco que representa todo o sistema.

Além disso, existem muitos estudos que provam a necessidade de colaboração entre os projetistas no desenvolvimento de um sistema complexo. Porém, os ambientes convencionais de projeto de sistemas eletrônicos dificilmente oferecem suporte ao trabalho colaborativo. Neste contexto, o Cave2, atual versão do ambiente Cave proposto em [IND 98] tem um novo foco de pesquisa: o suporte ao trabalho colaborativo [SAW 2002] [SAW 2002a] [IND 2001]. O Cave é um ambiente distribuído que integra diversas ferramentas de CAD para projeto de CIs, que podem estar distribuídas em diferentes computadores. Atualmente, o Cave disponibiliza uma ferramenta de edição de leiaute, chamada Jale [OST 2001] [GRA 99], uma ferramenta de comunicação (*chat*) conhecida como Cadena, entre outras ferramentas desenvolvidas no contexto deste projeto.

No contexto do projeto Cave, um serviço de colaboração foi implementado a fim de suportar projetos colaborativos usando as ferramentas do ambiente. Este serviço utiliza tecnologia *Jini* e *Javaspaces* [SAW 2002b] e permite o armazenamento persistente dos dados de projeto, bem como o compartilhamento de dados e também a comunicação entre projetistas. Com a integração deste serviço ao ambiente, o suporte à colaboração poderá ser incorporado em qualquer ferramenta do ambiente, tanto ferramentas que manipulam representações textuais como ferramentas gráficas.

A fim de permitir o projeto colaborativo de sistemas VLSI complexos, atualmente, está sendo desenvolvida no GME uma ferramenta chamada Homero [HER 2001] [HER 2001a]. O Homero é um editor de texto voltado para descrições textuais de sistemas eletrônicos usando linguagens de descrição de hardware e foi utilizado como estudo de caso para o primeiro protótipo do serviço de colaboração implementado por Sawicki [SAW 2002]. Dois modos de colaboração foram propostos para o ambiente Cave, o visualmente acoplado e visualmente desacoplado [IND 2001], e são detalhados

no capítulo 3. Porém, o Homero, por trabalhar com representações textuais, permitiu validar apenas o modo visualmente acoplado.

Neste contexto, este trabalho apresenta o desenvolvimento de um editor de diagramas que visa a especificação gráfica de sistemas de forma colaborativa, onde dois ou mais projetistas podem colaborar na construção de diagramas. Assim como o Homero, esta ferramenta também será utilizada como estudo de caso para a colaboração no Cave. O Blade, por ser uma ferramenta gráfica, permite a validação dos dois modos de colaboração propostos por Indrusiak. Ainda, esta ferramenta permite validar a integração do serviço de colaboração em ferramentas gráficas.

Além disso, segundo Neuwirth [NEU 90], o uso de técnicas diagramáticas facilita a troca de idéias entre colaboradores de forma padronizada, bem como auxilia o acoplamento de componentes desenvolvidos separadamente. A partir desta afirmativa pode-se dizer que as ferramentas que utilizam representações gráficas são mais adequadas à colaboração.

Além da colaboração, no Cave está sendo criado um *framework* de software, uma biblioteca de classes Java que podem ser reutilizadas para o desenvolvimento de novas ferramentas para operação remota. O desenvolvimento de uma ferramenta de edição de diagramas utilizando esta biblioteca de classes também contribuirá para a validação deste *framework*. Além de incorporar ao ambiente Cave uma ferramenta para especificação gráfica de sistemas, que foi totalmente desenvolvida voltada para as propostas de colaboração planejadas para o Cave.

Outra motivação para este trabalho refere-se à independência de plataforma. Todas as ferramentas do ambiente Cave são desenvolvidas em Java, qualificando-o como um ambiente multi plataforma. Portanto, o novo editor de diagramas, desenvolvido no contexto deste trabalho, também implementado usando a linguagem Java, é uma ferramenta que pode ser utilizada em diferentes plataformas.

1.2 Proposta

A proposta deste trabalho é o desenvolvimento de um editor de diagramas hierárquico voltado para a colaboração que será utilizada como estudo de caso para o trabalho colaborativo no ambiente Cave. O protótipo desenvolvido foi chamado de Blade (*Block and diagram editor*).

Um editor de diagramas no contexto desta dissertação é uma ferramenta que permite a especificação de sistemas em diferentes níveis de abstração através da montagem de diagramas, gerando ao final uma descrição comportamental usando uma linguagem de descrição de hardware ou ainda, uma descrição em mais alto nível usando uma linguagem de descrição de sistemas (SDL, do inglês, *Systems Description Language*).

O Blade suportará, em um primeiro momento, somente diagramas a nível lógico, porém foi desenvolvido com o intuito de ser estendido a outras formas de diagramas permitindo o uso de especificações em outros níveis de abstração. Esta flexibilidade é um aspecto muito importante da ferramenta porque facilita a sua extensão, de modo a permitir a criação e edição de outras classes de diagramas, bem como, o reuso das classes do editor podem permitir o desenvolvimento de outros editores que utilizam outras representações gráficas, ou seja, outras formas de diagramas.

Futuramente, pretende-se integrar o Blade e o Homero. Esta integração visa suportar, dentro do ambiente Cave, a especificação de diferentes módulos de um mesmo sistema, de forma colaborativa, através da captura de esquemáticos ou de descrições textuais, neste caso usando uma linguagem de descrição de hardware. O editor fornecerá a visualização do sistema através de um diagrama onde estão representados os módulos do sistema e como estes estão interligados.

Com esta integração, permite-se suportar descrições usando representações diagramáticas e textuais para os módulos do sistema. Usando uma metodologia de projeto *top-down*, o projetista inicia o projeto montando graficamente uma descrição do sistema através do editor de esquemáticos. Esta descrição inicial pode ser um diagrama composto por blocos interconectados, os quais representam os módulos do sistema, fornecendo uma visualização de todo o sistema. Desta maneira, sempre haverá pelo menos uma representação gráfica do sistema para ser compartilhada entre os projetistas. Esta abordagem suporta a especificação de um sistema usando diferentes níveis de abstração na descrição de seus módulos. Por exemplo, um módulo mais simples pode ser descrito no nível lógico e um bloco mais complexo no nível comportamental. Podendo ainda, através da extensão desta ferramenta gráfica, permitir o uso de outras representações diagramáticas, tais como: máquina de estados e diagramas UML para a descrição dos módulos do sistema.

1.3 Objetivos

O presente trabalho apresenta o desenvolvimento de um editor de diagramas voltado para ambientes distribuídos que visa a especificação de sistemas através de representações gráficas, bem como a colaboração de representações diagramáticas entre projetistas de CIs. O Blade deve permitir a operação remota e deve ser independente de plataforma.

O ambiente Cave, no qual o Blade será inserido, disponibiliza um conjunto de classes Java reutilizáveis, que podem ser utilizadas na incorporação de novas ferramentas ao ambiente. Estas classes estão presentes no *framework* para uso por parte dos desenvolvedores de ferramentas e visam facilitar o desenvolvimento de novas ferramentas. Esta biblioteca de classes auxilia na definição da interface da ferramenta, bem como auxilia na modelagem de primitivas de projeto. Algumas funções de edição

de objetos gráficos (seleção, movimentação, redimensionamento) também já estão estruturadas, o que facilitou a implementação do Blade.

Além disso, o desenvolvimento do Blade serviu também para validar este *framework*. Durante o projeto do editor observou-se a necessidade de alterações na estrutura do *framework*, o resultado foi uma maior flexibilidade. Novas classes foram incorporadas ao *framework* visando facilitar a extensão do editor e o desenvolvimento de outras ferramentas de edição de diagramas para ambientes distribuídos. Durante o projeto, a flexibilidade da ferramenta foi um dos aspectos mais observados.

Para alcançar estes objetivos, inicialmente, realizou-se um estudo das ferramentas de edição de esquemáticos, fazendo um apanhado das funcionalidades requeridas a esta classe de ferramentas. Posteriormente, foi estudada a modelagem de dados, a fim de modelar as primitivas de projeto, ou seja, os dados a serem manipulados pela ferramenta.

Para o desenvolvimento da ferramenta, observou-se a necessidade de realização de um estudo sobre conceitos de orientação a objetos (OO), modelagem de dados OO, bem como padrões de projeto de software. A escolha da metodologia OO baseou-se em dois aspetos muito importantes para este projeto, a reusabilidade e a flexibilidade, que são inerentes a esta metodologia. O uso dos conceitos de orientação a objetos viabiliza o reuso e aumenta a flexibilidade, facilitando a extensão futura da ferramenta.

1.4 Organização do trabalho

A dissertação está organizada da seguinte forma:

O capítulo 2 traz uma revisão bibliográfica sobre ferramentas de edição de esquemáticos, discute o potencial do uso desta ferramenta na especificação de projetos e apresenta um estudo das características e funcionalidades desta classe de ferramenta.

O capítulo 3 aborda o ambiente Cave, apresentando a estrutura do ambiente, explicando como as ferramentas são executadas no Cave e como novas ferramentas podem ser adicionadas ao ambiente. Também são apresentadas as últimas investigações realizadas dentro do escopo do projeto Cave.

No capítulo 4, são apresentados estudos sobre modelagem orientada a objetos, padrões utilizados na construção de ferramentas de edição gráfica ou textual, juntamente com padrões utilizados para a definição de arquiteturas de software reutilizáveis. Estes estudos serviram como fundamentos para o desenvolvimento do Blade.

No capítulo 5, é feita a especificação da ferramenta, destacando-se funções desejadas à ferramenta. A descrição da implementação da primeira versão do Blade é apresentada no capítulo 6. As contribuições desta dissertação estão apresentadas no capítulo 7, onde também encontram-se as propostas para a continuidade do trabalho. Por

fim, no capítulo 8, são apresentadas as referências bibliográficas utilizadas para o desenvolvimento deste trabalho.

2 Editores de Esquemáticos

Este capítulo tem por objetivo apresentar um estudo sobre ferramentas de edição de esquemáticos. Este estudo, além de uma revisão bibliográfica, abrange também a análise de um grupo de ferramentas de edição de esquemáticos. Foram analisadas ferramentas desenvolvidas no GME, Grupo de Microeletrônica desta Universidade, e algumas ferramentas comerciais. A partir deste estudo, foi realizada uma listagem de funções suportadas por estes editores, bem como características desejáveis a esta classe de ferramentas.

O capítulo é organizado da seguinte maneira. Na seção 2.1, faz-se uma breve descrição dos editores de esquemáticos, com o objetivo de destacar seu uso e sua importância dentro do fluxo de projeto de circuitos eletrônicos. A seção 2.2 aborda os editores de esquemáticos hierárquicos. Na seção 2.3, descreve-se as primitivas de projeto utilizadas pelos editores de esquemáticos. Na seção 2.4, faz-se um apanhado das funcionalidades que normalmente estão disponíveis nas ferramentas de edição de esquemáticos e ainda, algumas funcionalidades desejáveis às mesmas. Na seção 2.5, apresenta-se algumas ferramentas com funcionalidades de edição de esquemáticos desenvolvidas pelo GME. Na seção 2.6, faz-se uma relação de algumas empresas que fornecem sistemas de CAD comerciais. A seção 2.7 apresenta uma análise comparativa das ferramentas estudadas.

2.1 Introdução

A concepção de um circuito ou sistema integrado é normalmente dividida em etapas devido à complexidade do problema. Cada uma destas etapas de síntese pode ser feita através de uma ou mais ferramentas de CAD (*Computer Aided Design*, projeto auxiliado por computador) específica.

A síntese de um circuito ou sistema integrado pode ser vista como uma seqüência de transformações incrementais aplicadas a uma descrição inicial da funcionalidade do circuito [REI 2000]. A seqüência de operações aplicadas a uma descrição inicial determina um fluxo de projeto que é baseado no uso de ferramentas de CAD, que permitem a transformação incremental da descrição de um projeto em outra mais detalhada ao longo do processo de síntese.

Os editores de esquemáticos são classificados como ferramentas de captura e edição e permitem a descrição de circuitos graficamente, através do posicionamento de símbolos referentes aos componentes que fazem parte do circuito e da ligação destes componentes. Após a especificação, a montagem do esquemático, o editor deve fornecer um *netlist* do circuito. O *netlist* é uma descrição textual do circuito que pode ser utilizada como entrada em uma ferramenta de simulação ou de síntese. Muitas vezes

descrições em formatos padrões para *netlist* são utilizadas para permitir a troca de informações entre as várias ferramentas que fazem parte do fluxo de projeto.

Os editores de esquemáticos foram muito utilizados, e ainda o são, na especificação de sistemas elétricos/eletrônicos simples. Utilizando componentes elétricos básicos (capacitores, resistores, transistores), o editor permite a realização de descrições a nível elétrico do sistema. Na especificação de sistemas digitais, um editor de esquemáticos permite a realização de descrições tanto a nível elétrico, quanto a nível lógico. Nas descrições a nível elétrico utilizam-se transistores e nas descrições a nível lógico utilizam-se funções lógicas, tais como AND, OR, NOT, etc.

2.2 Editores de Esquemáticos Hierárquicos

O crescimento da complexidade dos circuitos faz com que os desenvolvedores de CAD pesquisem novas formas de dominar esta complexidade. Algumas alternativas foram estudadas, aplicadas e utilizadas pelos projetistas, até hoje, a fim de dominar esta complexidade. Entre elas, pode-se citar a modularização e a hierarquia. Nos editores de esquemáticos, os princípios de modularização e hierarquia são utilizados a fim de permitir a especificação de sistemas complexos através de um esquemático.

A modularização é importante no processo de projeto de sistemas eletrônicos, pois permite que sistemas complexos sejam divididos em módulos, facilitando as tarefas de análise e síntese que são limitadas a módulos de pequeno porte, cuja funcionalidade pode ser facilmente compreendida e verificada pelo projetista [WAG 94].

A hierarquia é uma conseqüência imediata da modularidade. Módulos são divididos em sub-módulos, e assim sucessivamente, até que um sub-módulo tenha porte suficientemente pequeno para que sua função possa ser facilmente compreendida. A especificação de um sistema complexo utilizando uma abordagem hierárquica de módulos facilita o entendimento do sistema, bem como permite o trabalho em equipe.

Um editor de esquemáticos hierárquico permite a especificação de um sistema complexo através do uso do princípio de hierarquia. A hierarquia reduz o tamanho e a complexidade do esquemático. Numa abordagem hierárquica de projeto, no nível mais alto de hierarquia os módulos do sistema são representados por blocos. Nos níveis subseqüentes, estes módulos podem ter uma funcionalidade associada que representa um nível mais baixo de hierarquia, um nível de detalhamento maior.

Uma visualização hierárquica de um sistema complexo facilita o entendimento do projeto. A hierarquia de visualização é uma característica muito interessante numa ferramenta de edição de esquemáticos. À medida que se desce na hierarquia, a descrição dos blocos é mais detalhada. Uma vista mais abstrata, em nível de interface e pinos, também pode ser útil, pois pode ser utilizada por ferramentas de posicionamento e roteamento. Assim como uma visualização dos módulos do sistema em diferentes níveis

de abstração também pode ser interessante, permitindo que o projetista acompanhe todo o processo de síntese.

Na descrição de módulos simples pode-se utilizar um esquemático a nível lógico ou elétrico. No entanto, na descrição de módulos mais complexos podem ser utilizadas descrições em linguagens de descrição de hardware (HDL, *Hardware Description Language*) ou de descrição de sistemas. Portanto, no projeto de um sistema complexo é interessante o uso de um ambiente de CAD que permita o uso de diferentes representações gráficas / textuais, permitindo a construção de descrições em diferentes níveis de abstração.

Na seção 2.4.3, a hierarquia é abordada em mais detalhes, algumas figuras são utilizadas para ilustrar o uso deste princípio. Esta seção apresenta também uma lista das funções normalmente disponibilizadas nas ferramentas para permitir a especificação de projetos hierárquicos.

2.3 Primitivas de Projeto

Um editor de esquemáticos trabalha com primitivas de projeto que representam os componentes que podem ser utilizados na montagem de um esquemático. As primitivas utilizadas por um editor de esquemáticos são: símbolo, bloco funcional, pino de esquema (pino de entrada e pino de saída), conexão, ponto de solda e texto.

2.3.1 Símbolo

Um símbolo é uma primitiva que representa um componente eletrônico, cada componente possui sua representação gráfica, além de características intrínsecas ao mesmo, relacionadas com o comportamento ou funcionalidade do componente.

Normalmente, os símbolos utilizados para a representação das células num editor de esquemáticos podem ser buscados em uma biblioteca de células ou podem ser editados através de um editor de símbolos. No caso de um editor de esquemas lógicos, os símbolos são usados para representar portas lógicas, tais como and, or, inversor, etc.

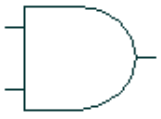


FIGURA 2.1 - Representação gráfica de uma porta lógica AND

2.3.2 Bloco funcional

Os blocos funcionais podem ser utilizados para representar diferentes módulos do sistema, portanto, o editor deve permitir a definição por parte do projetista das pinos

de entrada e saída dos blocos. Assim como, para uma melhor visualização dos blocos e de suas pinos, o editor deve permitir a redefinição do tamanho dos blocos pelo usuário.

Na figura 2.2, pode-se observar a representação de um bloco funcional. Normalmente, nos esquemáticos, costuma-se definir as entradas no lado esquerdo do bloco e as saídas no lado direito, de acordo com esta abordagem na figura 2.2 encontra-se a representação de um bloco com cinco pinos de entrada e cinco pinos de saída.

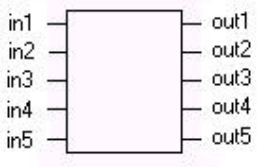


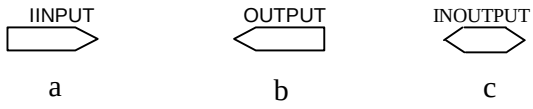
FIGURA 2.2 - Representação gráfica de um bloco funcional

Os blocos também podem ser usados para representar uma célula que não possua um símbolo definido, ou ainda representar um IP (*Intellectual Property*). Os IPs são núcleos pré-definidos que podem ser vistos como caixas pretas das quais somente a interface é visualizada.

Os blocos são elementos importantes em editores de esquemáticos hierárquicos. Na seção 2.4.2 aborda-se a hierarquia e ilustra-se o uso dos blocos funcionais na definição de um esquema hierárquico.

2.3.3 Pinos de esquema

Os pinos de esquema representam as entradas e saídas de um esquema elétrico, ou seja, definem a interface do circuito. Os pinos de esquema podem ser de dois tipos: pinos de entrada e pinos de saída. Na figura 2.3a e 2.3b são ilustradas, respectivamente, a representação gráfica de um pino de entrada e de um pino de saída.



(a) pino de entrada (b) pino de saída (c) pino de entrada e saída

FIGURA 2.3 - Representação gráfica dos pinos de esquema

Cabe observar que os pinos de uma células podem ser de entrada ou de saída. Porém, quando trata-se de pinos que fazem parte da interface de um chip, estes podem ser de entrada, de saída ou bidirecionais.

2.3.4 Conexão

A conexão representa a interconexão entre símbolos ou blocos, ou seja, uma conexão estabelece a ligação entre os pinos de símbolos, de blocos e pinos de esquema. Na figura 2.4 pode-se observar o exemplo de uma conexão, a qual representa a ligação entre a saída da porta AND2_0 e uma das entradas da porta OR2_1.

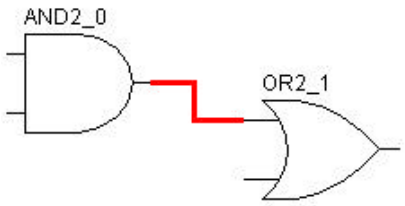


FIGURA 2.4 - Representação gráfica de uma conexão (sinal)

Uma conexão não pode ser modelada no editor simplesmente por um conjunto de segmentos de retas, uma conexão deve ser descrita através de redes de conexões. Uma rede é composta pelo conjunto dos pinos de componentes que estão conectados entre si, representando um mesmo sinal no circuito.

Conforme já foi mencionado, o editor de esquemáticos deve gerar um *netlist* a partir do esquemático do circuito. Uma descrição *netlist* representa todos os componentes que fazem parte do circuito e as interconexões existentes entre eles. Em outras palavras, um *netlist* é uma lista de redes de interconexões, na qual cada rede representa a ligação entre dois ou mais pinos de componentes.

A estrutura de conexões baseada em redes facilita a geração do *netlist*, bem como o uso do conceito de redes na modelagem das conexões facilita o desenvolvimento de funções do tipo detectar e eliminar laços (curto-circuito), ressaltar conexão, apagar conexão. A função de ressaltar conexão é muito útil quando se deseja observar uma determinada conexão dentro de um esquema complexo [MOR 90].

Existem duas classes de conexões: sinais e barramentos. O sinal é uma conexão “unária”, já um barramento representa o agrupamento de diversos sinais numa única conexão. A união de sinais é feita pelo ponto de solda e a união de barramentos é feita pelo elemento denominado derivador [MOR 90]. A figura 2.5 ilustra o uso de sinais, barramentos, pontos de solda e derivadores.

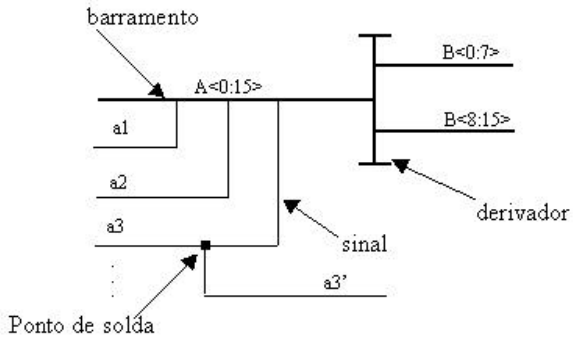


FIGURA 2.5 - Uso de sinais e barramentos [MOR 90]

2.4 Análise da funcionalidade dos editores de esquemáticos

Fez-se uma análise de várias ferramentas de edição de esquemáticos e foram identificadas as principais funções disponibilizadas por uma ferramenta de edição de esquemáticos. O grupo de ferramentas analisadas pode ser observado na tabela 2.1. No grupo encontram-se ferramentas desenvolvidas no GME, Grupo de Microeletrônica desta Universidade, e ferramentas comerciais.

Diversas características foram observadas, muitas delas estão presentes em todas as ferramentas analisadas, tais como inserção de símbolos, movimentação, conexão entre os pinos, etc. Outras características foram encontradas em apenas algumas das ferramentas. Através desta análise, foi possível obter uma lista das principais funções requeridas a esta classe de ferramentas, bem como a idealização de novas funções que facilitem o projeto de sistemas complexos.

TABELA 2.1 - Editores de esquemáticos analisados

Ferramenta	**Desenvolvedor**	**Classificação**
Design Architecture [MEN 2002]	Mentor Graphics	Ferramenta de síntese e simulação de circuitos
Virtuoso Composer [VIR 2001]	Cadence Design Systems	Ferramenta de síntese e simulação de circuitos
MaxPlus II [MAX 97]	Altera Corporation	Ferramenta de síntese e simulação de FPGAs
Quartus II [QUI 2002]	Altera Corporation	Ferramenta de síntese e simulação de FPGAs
Xilinx Foundation [GET 2000]	Xilinx	Ferramenta de síntese e simulação de FPGAs
Esqueleto [MOR 90]	GME - UFRGS	Editor de Esquemáticos
Ágata [CAR 97]	GME - UFRGS	Editor de Esquemáticos + Síntese
JASE I [REI 99]	GME - UFRGS	Editor de Esquemáticos
JASE II [REI 99a]	GME - UFRGS	Editor de Esquemáticos

2.4.1 Funções básicas

Primeiramente, são analisadas as operações gráficas e aquelas funções disponíveis em todas as ferramentas desta classe. Por fim, descreve-se algumas características desejáveis a esta ferramenta ou encontradas apenas em algumas das ferramentas analisadas.

A ferramenta de edição de esquemáticos possui uma área de trabalho, na qual os projetistas podem inserir componentes (portas lógicas e blocos) e define as conexões entre eles, ou seja, realiza a montagem do esquema. A ferramenta deve disponibilizar funções de inserção, exclusão e movimentação dos componentes na área de trabalho do editor para a montagem de uma descrição. O editor deve permitir também rotacionar e espelhar componentes, além de traçar conexões entre os pinos destes elementos.

?? Inserção dos componentes na área de trabalho

O editor deve permitir a inserção de símbolos, blocos e pinos de esquema na área de trabalho do editor. Já as conexões e pontos de solda são instanciados de forma diferente. Porque uma conexão deve ser criada ligando dois ou mais pinos de células, além disso, um ponto de solda deve ser criado quando uma conexão ligar mais de dois pinos de células.

?? Exclusão dos componentes na área de trabalho

Assim, como o projetista deseja inserir símbolos no esquema, o contrário também pode ocorrer, ou seja, o usuário da ferramenta pode desejar excluir um símbolo que faz parte do esquemático.

?? Movimentação dos símbolos na área de trabalho

?? Rotação dos componentes

A função de rotação permite alterar a orientação dos componentes. Esta função está ilustrada na figura 2.6, onde se pode observar quatro portas lógicas AND orientadas em diferentes direções.

FIGURA 2.6 - Rotação de Componentes

?? Espelhamento dos componentes

A função de espelhamento permite inverter a orientação dos componentes e está ilustrada na figura 2.7.

FIGURA 2.7 - Espelhamento de componentes

?? Criação de Conexões

O sistema deve permitir ao usuário conectar as células que fazem parte do esquema, através da interligação de pinos pertencentes a estas células. As linhas de conexão que ligam estes dois pontos podem ser geradas automaticamente pelo sistema, ou o usuário pode especificar o desenho das linhas através do mouse ou de um *tablet*.

?? Conexões formadas por segmentos de retas horizontais e /ou verticais;

Estas conexões devem ser formadas por segmentos de reta tanto orientados verticalmente como horizontalmente e ortogonais, ou seja, dois segmentos de reta adjacentes devem formar ângulos de 90° entre si. Na figura 2.8 pode-se observar que a conexão é formada por três segmentos, dois horizontais e um vertical.

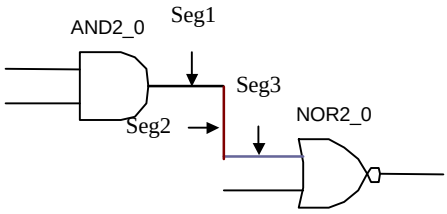


FIGURA 2.8 - Conexões formadas por segmentos

No caso de conexões ortogonais, o editor pode permitir ao usuário definir os pontos de quebra das linhas de conexão. Na figura 2.9, pode-se observar os pontos P1 e P2, sendo que estes pontos podem ser especificados pelo usuário. Neste caso, a posição de P1 e P2, juntamente com a posição dos pontos que devem ser conectados, definem o desenho das linhas de conexão.

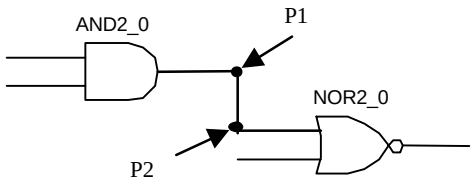


FIGURA 2.9 - Desenho de uma Conexão

?? Conexões vinculadas às células

As conexões devem ser vinculadas às células, de modo que na movimentação de uma célula, as conexões ligadas a esta célula adaptem-se ao seu novo posicionamento. Isso se dá pelo redimensionamento e reposicionamento dos segmentos que formam a conexão. Nestes casos, a ferramenta gera automaticamente a nova posição e/ou dimensão para os segmentos, podendo permitir que o usuário altere posteriormente o posicionamento destes segmentos.

Na figura 2.10 (a), pode-se visualizar uma conexão formada por três segmentos, Seg1, Seg2 e Seg3. Esta conexão está vinculada à saída da porta AND2_0 e a uma das entradas da porta NOR2_0, de modo que se a porta AND2_0 ou a porta NOR2_0 forem reposicionadas, os segmentos que formam esta conexão deverão ser redimensionados e reposicionados. A figura 2.10 (b) mostra a mesma conexão após o reposicionamento da porta NOR2_0.

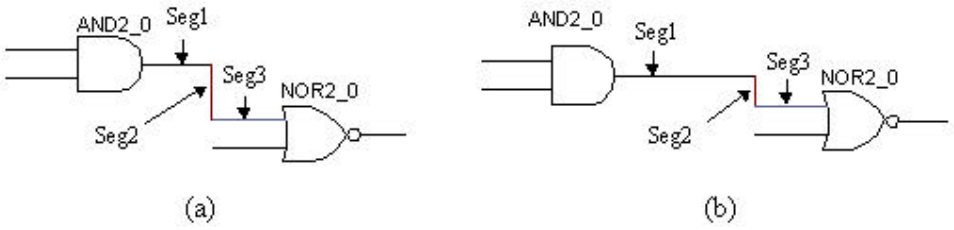


FIGURA 2.10 - Conexões Vinculadas às células

?? Textos vinculados às células

Assim como para os componentes, a ferramenta pode permitir a inserção, remoção, movimentação, rotação e espelhamento de textos. Da mesma forma que ocorre com as conexões, os textos devem ser vinculados às células, devendo o texto ter sua posição definida a partir da posição da célula a qual este se refere. Esta abordagem permite que quando a célula tiver sua posição alterada, a posição do texto também seja alterada. Na figura 2.11 pode-se observar o reposicionamento do texto referente à célula inversora quando esta célula teve sua posição alterada.

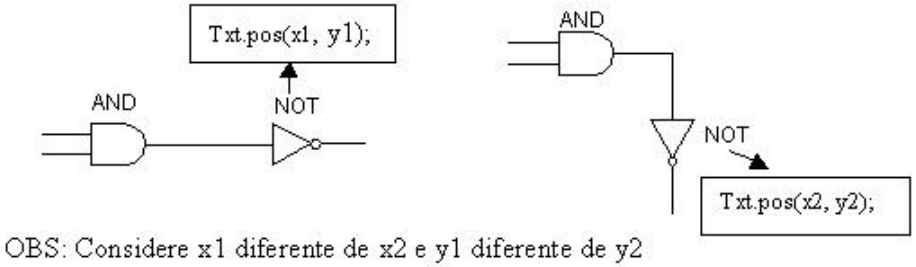


FIGURA 2.11 - Reposicionamento de textos vinculados às células

Uma ferramenta de edição de esquemáticos pode disponibilizar um modo de configuração no qual o usuário pode selecionar se quer ou não visualizar os textos vinculados às células.

?? Permitir a procura de nodos e conexões pelo nome, ressaltando-as no esquema;

?? Disponibilizar funções de *Zoom*

Assim, como em outras ferramentas de edição gráfica, num editor de esquemáticos é desejável a presença de diferentes modos de visualização, a fim de que o usuário possa ter visões em diferentes ampliações, permitindo, no caso de um circuito muito grande, ver todo ele na tela do computador, ou visualizar apenas uma parte selecionada do esquema com um nível de detalhamento maior.

?? Disponibilizar ao usuário uma forma de optar por visualizar ou não visualizar a grade na área de trabalho.

?? Possibilidade de definição da grade, ou seja, da distância entre os pontos na área de trabalho.

?? Pré-visualização dos símbolos antes de sua inserção na área de trabalho.

?? Limite de gravidade, ou seja, o número de *pixels* que representa a distância na qual atua uma força de gravidade, provinda da ponta dos pinos, e que atrai as conexões. Esta funcionalidade facilita a criação das conexões.

?? Ferramenta agregada para a edição de novos símbolos, permitindo a montagem de sua representação gráfica e a posição de seus pinos.

?? Permitir copiar e mover componentes ou grupo de componentes entre diagramas, facilitando o reuso (usando funções de copiar e colar).

2.4.2 Hierarquia

Numa abordagem hierárquica, os módulos do sistema são divididos em sub-módulos, e assim sucessivamente, até que um sub-módulo tenha porte suficientemente pequeno para que sua função possa ser facilmente compreendida.

Na figura 2.12, apresenta-se uma árvore da hierarquia de visualização adotada para o projeto de um *shift register* de 4 bits, onde o nível mais baixo da hierarquia é uma descrição lógica de um componente básico do circuito.

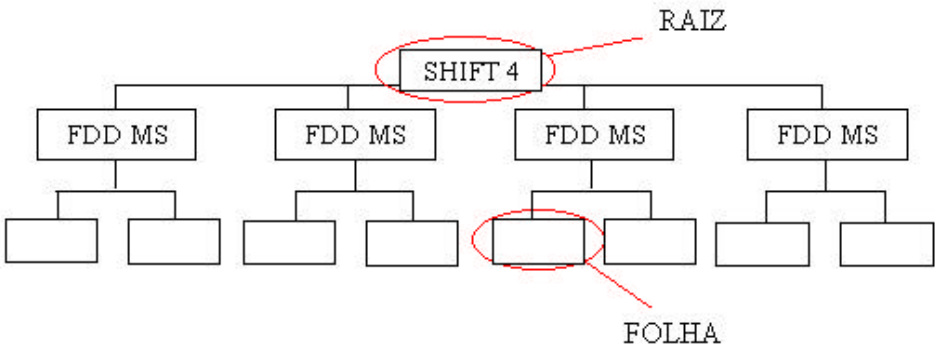


FIGURA 2.12 - Árvore de hierarquia de visualização [REI 83] [REI 99]

Nas figuras 2.13, 2.14 e 2.15, pode-se observar uma visualização hierárquica de um circuito. Na figura 2.13 tem-se um bloco chamado *shift_register* de 4 bits, que é descrito a partir de 4 *flip-flops* D (FFD) mestre-escravo (MS). O *shifter* é a raiz da árvore de hierarquia. Na figura 2.14 tem-se a visualização do FFD mestre-escravo a partir de dois FFD e na figura 2.15, tem-se um diagrama que representa um FFD no nível de portas lógicas. Esta figura representa, o esquema folha da árvore de hierarquia de visualização representada na figura 2.12.

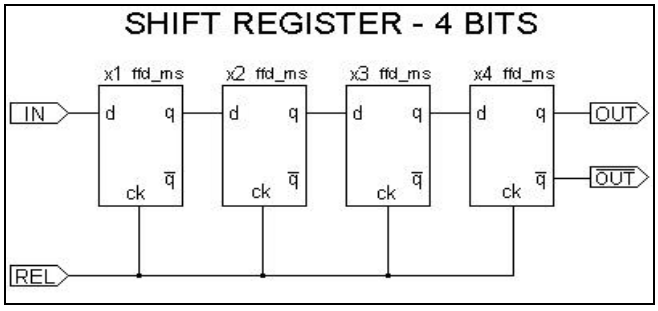


FIGURA 2.13 - *Shift-Register* de 4 bits [REI 99]

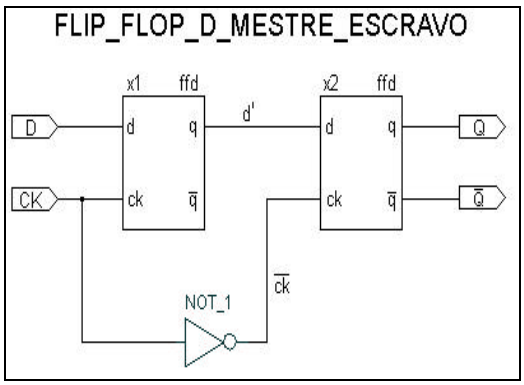


FIGURA 2.14 - FFD Mestre Escravo [REI 99]

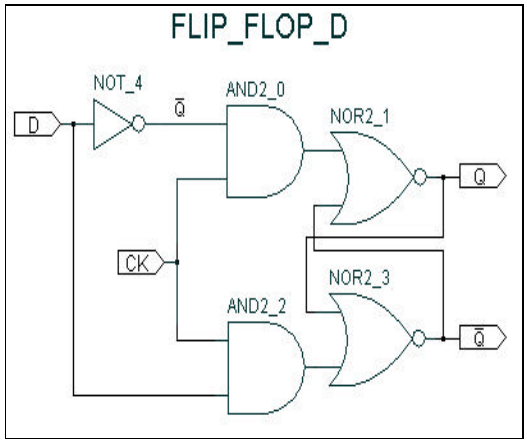


FIGURA 2.15 - *Flip-flop D* [REI 99]

As figuras 2.13, 2.14 e 2.15 fornecem uma visão hierárquica de um circuito, onde se pode observar o mesmo em um nível alto de visualização, com o uso de blocos. E tem-se também uma visão mais detalhada, onde se pode visualizar uma descrição de um bloco do circuito num nível mais baixo de visualização com portas lógicas, por exemplo. Porém, todas as representações do *shifter* encontram-se no nível lógico, pois considerou-se o nível lógico como o nível das folhas na árvore hierárquica de visualização.

Uma visão hierárquica, portanto, não implica necessariamente numa visão em diferentes níveis de abstração. Para este exemplo do *shift-register*, poderia ser mostrado ainda, um esquema elétrico do *flip-flop* D, ou seja, um esquema a nível de rede de transistores, ou ainda o leiaute para o mesmo, numa dada tecnologia, e assim teríamos descrições de um mesmo circuito em níveis diferentes de abstração. Estas duas características são interessantes num editor de esquemáticos. Tanto a hierarquia, quanto à visualização em diferentes níveis de abstração.

Abaixo é apresentada uma lista das funções disponibilizadas nos editores de esquemáticos de forma a permitir a entrada de projetos hierárquicos. Algumas destas funcionalidades são encontradas em todas as ferramentas comerciais, já outras não são tão comuns.

- Permitir ao projetista importar e/ou exportar blocos de circuitos pré definidos, permitindo o reuso destes blocos através da criação de instâncias dos mesmos.

- Para facilitar o reuso de blocos, o sistema utiliza uma única instância do circuito. Alterações nesta instância são refletidas nas demais instâncias do mesmo circuito utilizadas no projeto, de modo que há uma única instância a ser mantida pelo projetista.

- Suportar uma fácil navegação entre os diferentes níveis de hierarquia de um projeto hierárquico.

- Suportar a edição de múltiplas *sheets*, permitindo a edição de múltiplos níveis de hierarquia de um projeto.

- Permitir a criação de blocos funcionais que representam os módulos do sistema, podendo estes ter tamanho e número de pinos variável.

- Criação automática de um bloco para representar uma instância de um circuito pré definido sem precisar utilizar um editor de símbolos.

- Quando o projetista exclui um bloco funcional, se este bloco possui uma descrição, ou seja, um nível inferior de hierarquia, este também deve ser excluído.

- Alterações realizadas na interface de um bloco funcional devem refletir na sua descrição no nível inferior de hierarquia, através da inserção ou exclusão de pinos de esquema.

- Alterações nos pinos de esquema nos níveis inferiores de hierarquia também devem refletir na interface do bloco correspondente ao esquema alterado, através da inserção ou exclusão de pinos deste bloco.

Os editores hierárquicos comerciais permitem que se especifique um circuito e defina um símbolo para ele, alguns geram um símbolo automaticamente, bem como a interface (entradas e saídas) do circuito. Posteriormente, este esquema pode ser instanciado para montar um esquema mais complexo.

2.4.3 Descrição *netlist*

Após o término da especificação do sistema através da ferramenta de edição de esquemáticos, as ferramentas costumam trabalhar com representações num formato interno que representa o diagrama na estrutura de dados utilizada pela ferramenta. Esta descrição trata de aspectos tais como dimensão e posicionamento dos componentes do esquema e normalmente é armazenada no formato de um arquivo, permitindo a visualização e edição de um esquema gravado anteriormente. Além desta, uma descrição *netlist* deve ser gerada em um formato padrão, de forma que possa ser utilizada por outras ferramentas que fazem parte do fluxo de projeto adotado pelo projetista, tais como ferramentas de síntese, simulação, etc.

Na área de EDA, há um grande número de padrões que foram criados para a especificação de projetos de circuitos eletrônicos. Apresenta-se aqui, alguns formatos conhecidos e utilizados para descrições *netlist*, tais como SPICE, EDIF [ELE 2002] e VHDL.

Porém, convém salientar que além desses, existem outros formatos que também são bastante empregados para a descrição de sistemas, dentre eles pode-se destacar o formato BLIF [WAG 98] usado para *netlist* lógico e a linguagem *Verilog*, que assim como VHDL, é uma linguagem de descrição de hardware.

2.4.3.1 EDIF

O desenvolvimento de *frameworks* e de soluções de CAD integrados levaram ao desenvolvido de uma série de padrões que se tornaram elementos-chave no projeto e ciclo de desenvolvimento de produtos eletrônicos [RAM 95].

Neste contexto, o EDIF (*Electronic Design Interchange Format*) [ELE 2002] foi definido para resolver problemas de integração, migração e arquivamento de dados de projeto de sistemas digitais e analógicos. O EDIF é um formato neutro, de domínio público e bastante conhecido na área de EDA (*Electronic Design Automation*). A sua última versão é a 4.0.0, que suporta a descrição de PCBs, módulos multi-chips, e de regras de projeto (regras de desenho e de tecnologia), além de suportar esquemáticos, hierarquia, bibliotecas e configurações já suportadas pela versão anterior.

2.4.3.2 SPICE

SPICE (*Simulation Program with Integrated Circuit Emphasis*) foi desenvolvido por pesquisadores da Universidade de *Berkeley* durante a década de 70, com o objetivo de simular os circuitos antes do caro processo de fabricação. Atualmente, o SPICE é distribuído integrado a ferramentas gráficas para desenho de esquemáticos, bem como a ferramentas que permitem uma visualização gráfica dos resultados de simulação fornecidos pelo SPICE, através de formas de onda.

O coração de um arquivo de descrição SPICE é um *netlist*, que é simplesmente uma lista de componentes e as redes que conectam estes componentes uns aos outros. Este formato *netlist* usado numa descrição SPICE permite descrever o circuito em nível de rede de transistores, realizando uma prévia planificação do esquemático, mascarando desta forma a hierarquia. Uma descrição SPICE completa exige dados referentes à tecnologia e vetores de teste, que devem ser inseridos pelo projetista.

Um *netlist* no formato SPICE pode ser usado como entrada em ferramentas de simulação, de síntese física, bem como em ferramentas de estimativa de área e potência. Na figura 2.16, pode-se observar um exemplo de descrição no formato SPICE.

```
RELATORIO SPICE FLIP_FLOP_D_MESTRE_ESCRAVO
**--------------------------------------------------------------------------------------------------------------------**
DEFINICAO DOS SUBCIRCUITOS
**--------------------------------------------------------------------------------------------------------------------**
** CELULA: INVERSOR
.SUBCKT   INV   IN   OUT   vcc
MP1   OUT   IN   vcc   vcc  PMOS  L=2.0U  W=7.0U   AD=233P AS=180   PD=126U   PS=114U
MN2   OUT   IN    0     0    NMOS  L=3.0uW=4.0U   AD=12P   AS=48P   PD=14U    PS=38U
.ENDS
**--------------------------------------------------------------------------------------------------------------------**
CHAMADA DE SUBCIRCUITOS
**--------------------------------------------------------------------------------------------------------------------**
** CELULA: NAD2-1  NOR
.SUBCKT   AND21NOR  I1  I2  I3   OUT   vcc
MP1    1    I1   vcc   vcc  PMOS   L=2.OU  W=7.0U   AD=70P    AS=246P  PD=34u    PS=138U
MP2    1    I2   vcc   vcc  PMOS   L=2.OU  W=7.0U   AD= 210U AS=196P  PD=122P  PS=118P
MP3  OUT I3    1     vcc  PMOS   L=2.OU  W=7.0U   AD=120P  AS=246P  PD=82U   PS=138U
MN5   2     I1   0     0    NMOS   L=3.0U   W=4.0U   AD= 28P   AS=60P    PD=22U   PS=44U
MN6 OUT I2    2     0    NMOS   L=3.0U   W=4.0U   AD=28P    AS=60P    PD=22U   PS=44U
MN7 OUT I3    0     0    NMOS   L=3.0U   W=4.0U   AD=28P    AS=60P    PD=58U   PS=56U
.ENDS   AND21NOR
**--------------------------------------------------------------------------------------------------------------------**
SINAIS DEINTERFACE DO CIRCUITO
**--------------------------------------------------------------------------------------------------------------------**
X1        CK        ~ck_1     VCC       INV
X2        D         ~d_2      VCC       INV
X3        ~d_2      CK        ~q_2      d'_1      VCC       AND21NOR
X4        D         CK        d'_1      ~q_2      VCC       AND21NOR
X5        d'_1      ~d_3      VCC       INV
X6        ~d_3      ~ck_1     ~Q        Q         VCC       AND21NOR
X7        d'_1      ~ck_1     ~Q        Q         VCC       AND21NOR
**--------------------------------------------------------------------------------------------------------------------**
*Interface: CK
* Interface: D
* Interface: Q
* Interface: ~Q
.END
```

FIGURA 2.16 - Descrição SPICE relativa ao esquema FFD-MS

2.4.3.3 VHDL

Em 1980, avanços na tecnologia de circuitos integrados desencadearam esforços para a definição de padrões de práticas de projeto de circuitos digitais. Vários padrões foram desenvolvidos como parte deste esforço, dentre eles o VHDL. Esta linguagem tornou-se um padrão na indústria, sendo definida como um padrão IEEE. O padrão original ficou conhecido como IEEE 1076 [BRO 2000].

O VHDL foi originalmente criado para ser usado como uma linguagem de documentação para descrever circuitos digitais, além de fornecer um modo comum de documentar circuitos a ser utilizado pelos projetistas. Além disso, o VHDL permite a modelagem do comportamento de circuitos digitais. Posteriormente, VHDL tornou-se uma linguagem bastante popular entre os projetistas para ser usada como entrada de projeto em sistemas de CAD [BRO 2000].

A linguagem VHDL pode ser usada para representar um esquemático a nível lógico, bem como, descrever textualmente uma máquina de estados finita. Além disso, o VHDL pode ser usado para descrever projetos hierárquicos, em um nível estrutural ou comportamental. Na figura 2.13, pode-se observar uma descrição VHDL para um *flip-flop* D.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY flipflop IS
PORT (D, Clock: IN          STD_LOGIC
      Q         : OUT     STD_LOGIC) ;

END flipflop ;

ARCHITECTURE Behavior OF flipflop IS

BEGIN
      PROCESS (Clock)
      BEGIN
            IF Clock ' Event AND Clock = ' 1 ' THEN
            Q <= D ;
            END IF;
      END PROCESS;
END Behavior;
```

FIGURA 2.17 - Código VHDL – Descrição de um *flipflop* D

2.4.4 Verificação de Regras Elétricas (ERC)

Algumas ferramentas disponibilizam verificadores de regras elétricas que checam o circuito, verificando se não há um curto-circuito ou um circuito aberto, assegurando, assim, a integridade do circuito. Por exemplo, em algumas ferramentas a operação de rotação não mantém as conexões do componente, gerando uma inconsistência no circuito, ou seja, ao rotacionar um componente que já está devidamente conectado a um outro componente do circuito, as conexões do componente rotacionado são perdidas. Nestes casos, se a ferramenta possui um verificador de regras elétricas, o erro gerado é notificado, indicando que o projetista deve refazer a conexão que foi perdida.

Algumas ferramentas possuem testes de integridade que executam uma análise do *netlist* do circuito, detectando erros e inconsistências. Quando são encontrados erros, estes são notificados ao projetista que pode ver uma descrição dos mesmos e fazer as correções necessárias no esquema. Os seguintes erros são freqüentemente encontrados:

- Conexões com mesmo nome, visto que os nomes das conexões geralmente são gerados automaticamente quando as linhas são desenhadas.

- Nomes de macros idênticas ao nome do projeto.

O Esqueleto [MOR 90] provê um verificador que indica erros do tipo símbolo sem pinos, bloco funcional sem pinos, barramento conectado a pino de símbolo, sinal conectado a mais de um barramento. Além dos erros, a ferramenta indica também *warnings* quando há um pino de bloco funcional desconectado, pino de símbolo desconectado ou mais de uma saída na mesma conexão.

2.4.5 Colaboração

A necessidade de projeto colaborativo pode ser facilmente justificada pelo aumento da complexidade dos sistemas, demandando maior número de projetistas e indicando a necessidade de troca de experiência entre eles [IND 2001].

Além da hierarquia, a possibilidade de trabalho em equipe é uma conseqüência positiva da modularização. Uma vez que interfaces entre os módulos tenham sido especificadas adequadamente, módulos distintos podem ser projetados independentemente, por pessoas distintas, embora interações entre os projetistas sejam necessárias durante os projetos dos módulos, o que indica que o trabalho em equipe pressupõe cooperação entre projetistas [WAG 94].

Segundo Neuwirth [NEU 90], o uso de técnicas diagramáticas facilita a troca de idéias entre colaboradores de forma padronizada, bem como auxilia o acoplamento de componentes desenvolvidos separadamente. Assim, as ferramentas tais como editores de esquemáticos e editores de diagramas, por trabalharem com representações gráficas, mostram-se adequadas ao trabalho colaborativo. Segundo Cayres [CAY 2001], existem muitas pesquisas que abordam a colaboração em documentos diagramáticos.

O Blade, editor desenvolvido no contexto desta dissertação, permite que mais de um projetista trabalhe num mesmo diagrama de forma colaborativa. No primeiro protótipo da ferramenta, apenas um projetista, o qual é chamado escritor, pode alterar o diagrama efetivamente a cada momento, mas toda alteração que é feita por este projetista é notificada para os demais. E os outros projetistas podem requisitar o direito de escrita quando desejarem alterar o diagrama.

2.4.6 Interface com o Projetista

O estudo e o desenvolvimento de interfaces entre usuário e computador vêm ocorrendo desde os primórdios da computação [THI 90], pois, da eficiência desta interface depende a satisfação e a produtividade do usuário em relação à tarefa efetuada. Essa eficiência, por sua vez, está ligada aos recursos de hardware e software disponíveis para a implementação dessa interface [LAU 90].

As interfaces amigáveis ao usuário são um importante fator para a aceitabilidade de um novo sistema. Nos anos oitenta, isso significava simplesmente projetar telas

dando atenção apropriada a cores, sublinhadas, campos piscantes, etc. Hoje, isso quer dizer menus suspensos, janelas instantâneas, entradas para mouse, ícones e todos os outros componentes disponíveis para a montagem de interfaces gráficas com o usuário [COAD 93].

Uma ferramenta de edição de esquemáticos possui um alto grau de interação com o usuário. Portanto, a interface é de extrema importância a esta classe de ferramentas. Destaca-se aqui, alguns aspectos relacionados à interface de uma ferramenta de edição de esquemáticos.

- Interfaces para as funcionalidades básicas do editor devem ser bastante intuitivas, de modo a diminuir o tempo gasto pelo usuário para aprender a utilizar a ferramenta.

- Uso de barras de ferramentas removíveis, disponibilizando as principais funções do editor.

- Visualização de *hints* que indicam a funcionalidade dos botões da interface.

- Presença de tutoriais para os usuários com pouca experiência com o uso das ferramentas. Algumas ferramentas disponibilizam um *help* on-line.

A análise destes aspectos nos aponta uma tendência de uso intensivo de interfaces gráficas, com janelas, menus, barra de ferramentas (botões), deixando apenas o absolutamente necessário na forma de entrada textual por parte do usuário. Também se pode destacar o uso de *pop up* menus e teclas de atalho que ajudam a acelerar as tarefas de edição.

Também convém ressaltar aqui, que algumas metodologias de desenvolvimento de software facilitam a construção de interfaces, como é o caso da prototipação. O uso desta metodologia, ajuda na montagem da interface gráfica do sistema, pois permite a realização de várias experiências de interação com o usuário.

2.5 Editores de esquemáticos desenvolvidos pelo GME

No grupo de Microeletrônica da UFRGS, foram desenvolvidas anteriormente, outras ferramentas de edição de esquemáticos. Duas destas ferramentas voltadas para plataforma PC, o Esqueleto e o Ágata. O Esqueleto [MOR 90] é uma ferramenta de edição de esquemáticos e o Ágata [CAR 97] é um sistema de CAD que permite a entrada de dados através de um editor de esquemáticos. Foram também desenvolvidos, parcialmente, dois editores de esquemáticos para WWW implementados em Java, o JASE [REI 99] e o JASE II [REI 99a]. Nas duas implementações do JASE, não se chegou a um editor totalmente funcional, estes trabalhos focaram a especificação das funções necessárias a esta classe de ferramentas.

2.5.1 ESQUELETO

O sistema Esqueleto é um sistema de CAD interativo para a edição de esquemas elétricos. Este foi desenvolvido no grupo de Microeletrônica da UFRGS, com suporte financeiro da SID Microeletrônica, onde o objetivo principal era a especificação e implementação de um software brasileiro para a edição de esquemáticos [MOR 90]. O sistema Esqueleto foi desenvolvido para plataforma PC.

2.5.2 ÁGATA

O ÁGATA é um sistema de CAD desenvolvido por pesquisadores do grupo de Microeletrônica da UFRGS para suportar o projeto de circuitos dedicados baseado em um *array* de células a ser personalizado no então CTI (Centro Tecnológico para Informática – Campinas - SP). O sistema provê ferramentas para descrição de circuitos (ferramenta de edição de esquemáticos), simulação (o simulador e o visualizador de formas de onda), síntese de circuitos (ferramentas de particionamento e roteamento) e visualização de leiaute (ferramenta de visualização de leiaute). A ferramenta foi desenvolvida para plataforma PC.

2.5.3 JASE

Com o advento da Internet surgiu a necessidade de desenvolvimento de ferramentas de CAD independentes de plataforma para serem disponibilizadas na WWW. Neste contexto, no Grupo de Microeletrônica da UFRGS, foram desenvolvidos dois protótipos de editores de esquemáticos, o JASE I e o JASE II.

Estes protótipos foram desenvolvidos usando a linguagem Java. Esta linguagem foi escolhida devido à sua independência de plataforma, simplicidade e portabilidade.

O JASE I baseou-se no sistema Esqueleto. Neste protótipo foram implementadas várias funções gráficas necessárias às ferramentas de visualização e edição de esquemáticos, porém nem todas as funcionalidades do editor chegaram a ser implementadas. O JASE I [REI 99] permite a edição de circuitos em nível de portas lógicas e de blocos funcionais. Este trabalho apresenta, também, um levantamento de requisitos bastante consistente, que foi utilizado, posteriormente, no desenvolvimento de uma nova versão do editor, que foi chamada de JASE II.

O JASE II [REI 99a] baseou-se na análise de requisitos apresentada em [REI 99]. Porém, neste projeto houve uma preocupação maior com a flexibilidade da ferramenta, a fim de tornar mais fácil o acréscimo de novas funcionalidades. De forma a atender este objetivo, utilizou-se uma metodologia de projeto orientado a objetos.

2.6 Ferramentas de edição de esquemáticos comerciais

A *Cadence* e a *Mentor Graphics*, grandes empresas na área de EDA, possuem sistemas completos de síntese de circuitos integrados, dentro dos quais há ferramentas para projeto de circuitos tais como, ferramentas de edição de esquemáticos, ferramentas que trabalham com descrições em linguagens de descrição de hardware, além de ferramentas de síntese e simulação. Estes ambientes permitem a especificação de sistemas em diferentes níveis de abstração e geram o leiaute do circuito, cobrindo todo o fluxo de projeto de um CI.

A Altera e a *Xilinx*, grandes fabricantes de FPGAs, possuem sistemas para prototipação em FPGAs. Nestes sistemas há ferramentas para edição de descrições textuais usando linguagens de descrição de hardware, bem como editores de esquemáticos para entrada gráfica, além das ferramentas para síntese e simulação em FPGA. A Altera fornece o *MaxPlux II* e o *Quartus*, e a *Xilinx* possui o *Xilinx Foundation* e o ISE.

2.7 Análise Comparativa de Ferramentas de edição de esquemáticos

Esta seção apresenta uma comparação entre ferramentas de edição de esquemáticos. Foram analisadas ferramentas comerciais de diferentes fornecedores (*Cadence, Mentor, Altera, Xilinx*), bem como ferramentas desenvolvidas anteriormente pelo Grupo de Microeletrônica da UFRGS (GME). O JASE I e JASE II não serão considerados nesta comparação por serem protótipos e não ferramentas completas de edição de esquemáticos. A comparação baseia-se na análise das funcionalidades disponibilizadas por estes editores.

As funções básicas para a construção de diagramas, tais como inserção, movimentação e remoção de símbolos foram encontradas em todas as ferramentas analisadas. Assim como funções de z*oom* que normalmente são disponibilizadas em ferramentas de edição de diagramas. Algumas das ferramentas permitem, além do *zoom*, a definição da grade (*grid*), ou seja, a definição da distância entre os pontos na área de trabalho. Assim como algumas ferramentas permitem ao usuário especificar se deseja ou não visualizar a grade na área de trabalho.

Abaixo, estão enumeradas algumas das funcionalidades encontradas na maioria das ferramentas analisadas. Convém ressaltar que esta é uma listagem resumida, resultado de uma pesquisa nos manuais das ferramentas e no próprio uso das mesmas.

- Inserção, seleção/movimentação e remoção de componentes;
- Redimensionamento dos componentes;
- Rotação e espelhamento;

- Funções de z*oom;*
- Visualizar ou não o *grid* ;
- Definir g*rid;*
- Pré-visualização de símbolos;
- Limite de gravidade;
- Textos vinculados às células;
- Conexões ortogonais;
- Conexões vinculadas as células;
- Permite definir pontos de quebra da conexão;
- Editor de símbolos;

Mesmo em ferramentas comerciais, observou-se a falta de algumas funções interessantes tais como: criar automaticamente um pino em um componente quando chegar uma conexão sobre ele, ou ainda, garantir que as conexões estejam fortemente ligadas aos componentes de forma que quando estes forem movimentados, rotacionados ou espelhados as conexões não sejam perdidas. Em algumas ferramentas comerciais este problema não é tratado. Por exemplo, quando um componente que está conectado a outros componentes é rotacionado, a conexão existente entre o componente rotacionado e os demais componentes é perdida, devendo ser refeita.

É importante salientar também, nesta análise comparativa, a questão dos tipos de entrada suportados pelas ferramentas, bem como a geração de descrições *netlist* dos circuitos. As ferramentas *Design Architecture* da Mentor e a ferramenta *Virtuoso* da *Candence* permitem a entrada através de um editor de esquemáticos, a nível elétrico e lógico. Já a ferramenta *ISE*, permite, além do nível elétrico e lógico, também a entrada de diagramas de máquinas de estados.

Descrição *Netlist*

O *Virtuoso* e o *Design Architecture* geram uma descrição em formato EDIF a partir de um esquemático, assim como a ferramenta Ágata desenvolvida no GME. A ferramenta *Xilinx Foundation* gera descrições num formato interno, que pode ser convertido para os formatos XNF, EDIF ou VHDL. O Esqueleto suporta descrições PINLIST (FUTURENET) e SPICE [MOR 90].

Colaboração

Nenhuma das ferramentas analisadas suportam o trabalho colaborativo. Este é um dos maiores diferenciais do Blade, ferramenta desenvolvida ao longo deste trabalho e que será detalhada nos capítulos 5 e 6. O Blade faz parte do Projeto Cave, que tem como objetivo atual disponibilizar um ambiente de projeto de CIs distribuído e colaborativo.

O Blade, juntamente com o serviço de colaboração incorporado ao Cave, permitirá aos projetistas colaborarem na construção de diagramas. A integração deste serviço ao Blade suporta o compartilhamento de diagramas entre projetistas, bem como a troca de mensagens de texto através de uma ferramenta de comunicação (*chat*), além de outras funcionalidades úteis para a realização de projetos colaborativos.

3 Projeto Cave

O trabalho de um projetista de circuitos integrados necessita, muitas vezes, do uso de ferramentas de diferentes origens, implementadas em diferentes plataformas. Isso obriga o projetista a alternar o ambiente de trabalho entre diferentes plataformas de hardware e/ou software, usando diferentes modelos de interfaces (gráficas ou não), o que torna o fluxo de trabalho mais lento. A integração das ferramentas de apoio ao projeto é então necessária para um fluxo de projeto mais eficiente [IND 97].

O uso de sistemas em rede para a integração das ferramentas diminuiu o tempo perdido pelo projetista, mas não o eliminou. Ainda se faz necessária a implementação de um ambiente que poupe o projetista de administrar os recursos distribuídos, bem como ofereça a ele uma interface gráfica simples e eficiente para que possa se dedicar ao projeto propriamente dito [IND 98].

O projeto Cave é uma iniciativa de pesquisa que tem por objetivo tornar possível a distribuição de recursos de CAD de forma transparente para o usuário. Em [IND 98a] apresentou-se uma nova arquitetura para a integração de ferramentas de apoio ao projeto de circuitos integrados, visando reduzir o tempo perdido pelo projetista com a administração dos recursos distribuídos, bem como com o aprendizado do uso das interfaces de integração.

Neste capítulo, é apresentada, inicialmente, a proposta original do projeto baseada em hiperdocumentos e WWW. Atualmente, a distribuição e compartilhamento de recursos, juntamente com a colaboração representam os principais objetivos do projeto. Portanto, antes de apresentar a atual arquitetura do Cave, o capítulo apresenta também uma revisão bibliográfica sobre trabalho colaborativo.

3.1 Proposta Original: Ambiente baseado na *World Wide Web*

Algumas pesquisas foram realizadas [BEN 96] [NEW 96] [SIL 96] a fim de criar um novo paradigma para integração de ferramentas de apoio ao projeto de CIs, rodando sobre uma plataforma distribuída com uma interface comum. Muitas destas pesquisas focam a execução remota. A arquitetura Cave, proposta em [IND 97] baseia-se nesta abordagem, mas permite também que o projetista carregue uma ferramenta e rode-a na máquina local.

O Cave em sua proposta original, teve por objetivo a distribuição de recursos de CAD usando o ambiente WWW como infra-estrutura básica, em outras palavras, foi proposto um modelo para distribuição de recursos de projeto entre cliente e servidor. O modelo cliente/servidor utilizado pode ser observado na figura 3.1. Neste modelo, a máquina cliente também pode rodar algumas das ferramentas, sendo necessário, nestes

casos, transferências de códigos e dados entre cliente e servidor. Geralmente, as ferramentas que apresentam intenso uso de saídas gráficas são executadas na máquina do projetista, de forma a assegurar uma melhor performance nas interações entre projetista e ferramenta.

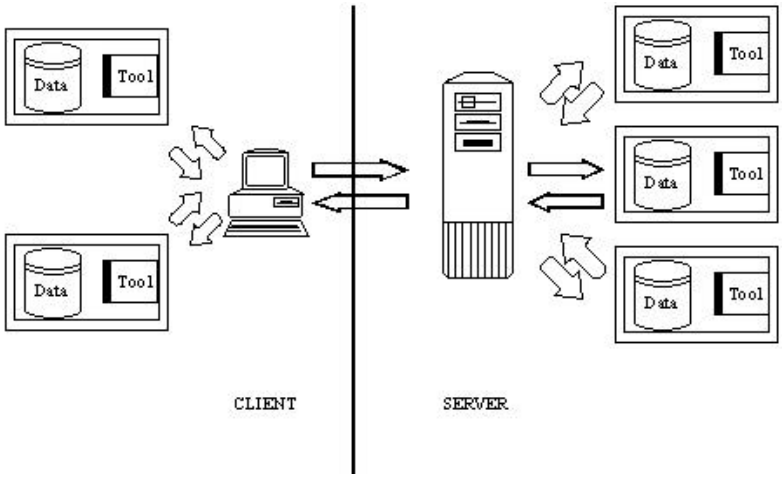


FIGURA 3.1 - Modelo Cliente servidor proposto para o Cave [IND 98]

O navegador (*web browser*) é usado como única interface entre o projetista e as ferramentas. Esta abordagem libera o projetista da gerência de diferentes plataformas de hardware e de software que podem estar envolvidas no fluxo de projeto e permite a execução de ferramentas tanto no servidor quanto na máquina cliente e em qualquer plataforma.

A fim de definir a distribuição das ferramentas de automação de projeto na rede, as ferramentas são divididas em dois grupos, de acordo com o nível de interação entre o projetista e a ferramenta. No grupo 1, encontram-se as ferramentas com alto grau de interação com o usuário, estas ferramentas caracterizam-se pelo uso intenso de interfaces gráficas. Exemplos claros de ferramentas deste grupo são os editores de esquemáticos e editores de leiaute. Estas ferramentas devem ser implementadas usando uma linguagem independente de plataforma e ligadas a hiperdocumentos e executadas na máquina cliente através de um *web browser*. O procedimento de execução das ferramentas do grupo 1 é descrito abaixo e ilustrado na figura 3.2.

- Quando o navegador requisita um hiperdocumento referente à ferramenta desejada, o servidor envia o hiperdocumento com a ferramenta anexada.

- O cliente recebe a aplicação e a executa.

- Os dados de projeto podem ser armazenados na máquina cliente ou no servidor. Neste último caso, é necessário abrir outra conexão de rede para o envio dos dados para o servidor.

Em contrapartida, no grupo 2 encontram-se as ferramentas com baixo grau de interação, que, por sua vez, são executadas no lado do servidor. As ferramentas do grupo 2 caracterizam-se por gerar resultados a partir de um arquivo de descrição do circuito e de parâmetros especificados pelo projetista e passados à ferramenta. O procedimento de execução das ferramentas do grupo 2 segue os passos descritos abaixo e também pode ser observado na figura 3.2.

- Quando o navegador requisita um hiperdocumento referente à ferramenta desejada, o servidor envia um formulário HTML, o qual é usado pelo projetista para definir parâmetros a serem passados à ferramenta.

- O usuário preenche o formulário e envia-o ao servidor.

- O servidor executa a ferramenta.

- Após a execução, o servidor pode enviar os resultados para o cliente ou armazená-los localmente.

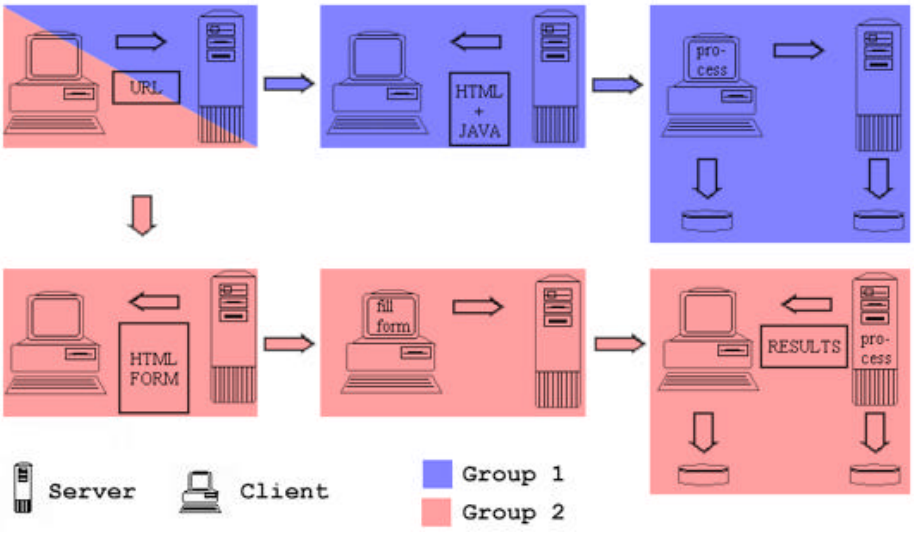


FIGURA 3.2 - Arquitetura de integração ferramentas - *framework* [IND 2002a]

Porém, o projeto de circuitos integrados é uma tarefa muito complexa que envolve muitas ferramentas de projeto. Então, além de prover ferramentas, um ambiente de projeto deve prover mecanismos para invocar estas ferramentas na ordem correta de acordo com a metodologia adotada. Assim como, deve permitir o compartilhamento de dados entre ferramentas e, em alguns casos, permitir também a conversão de dados. Todos estes aspectos apontam para a necessidade de definição de uma arquitetura de integração de ferramentas.

A arquitetura de integração de ferramentas proposta por Indrusiak em [IND 98a] tem forte ligação com os conceitos de WWW, na qual define-se uma metodologia de projeto como um canal de hiperdocumentos. Cada hiperdocumento ativa a interface de uma ferramenta de projeto, assim, os *links* entre hiperdocumentos conectam as ferramentas na ordem correta, permitindo que o usuário navegue entre as diferentes tarefas envolvidas no projeto de acordo com a metodologia escolhida. Esta arquitetura utiliza armazenamento e transmissão de dados de projeto baseados em arquivos. Nesta abordagem, os dados são armazenados no servidor junto com as ferramentas e com os hiperdocumentos que definem o fluxo de projeto. E a transmissão é feita usando um protocolo padrão do WWW, o HTTP.

Muitas vantagens descritas em [IND 98], defendem o uso desta arquitetura, tais como: facilidade de implementação e manutenção do ambiente de projeto, independência de plataforma, redução da sobrecarga cognitiva, possibilidade de uso remoto e de times distribuídos de projetistas através de projetos colaborativos.

Porém, apesar destas vantagens, algumas modificações na arquitetura original do Cave foram consideradas necessárias, principalmente devido ao modelo de integração e de armazenamento dos dados, que não se mostraram adequados para projetos distribuídos envolvendo diferentes projetistas ou quando múltiplas visões de blocos de projetos eram requisitadas.

Na próxima seção, são apresentados os resultados das investigações mais recentes no âmbito do projeto Cave, incluindo um estudo sobre o trabalho colaborativo. A seção apresenta também a nova arquitetura do Cave que comporta objetos de projeto, ao invés de documentos, e visa suportar a colaboração entre projetistas através de um ambiente distribuído.

3.2 Pesquisa Atual: Ambiente distribuído colaborativo

O processo de desenvolvimento de circuitos integrados, assim como o processo de desenvolvimento de software, apresenta muitas atividades inerentemente cooperativas, requerendo trabalho em equipe e conseqüentemente ferramentas de apoio [BOR 95][SOU 97][PIN 99a][PIN 99b]. Além disso, segundo Indrusiak [IND01], a necessidade de projeto colaborativo pode ser facilmente justificada pelo aumento da complexidade dos sistemas, demandando maior número de projetistas e indicando a necessidade de troca de experiência entre eles.

Como relata Willians [WIL 94], a distribuição das organizações tem obrigado seus profissionais a trabalharem com colegas distantes. Muitas vezes os projetistas de uma organização se encontram em diferentes localizações geográficas e precisam interagir e colaborar no desenvolvimento de um sistema, sendo necessárias ferramentas que suportem a colaboração entre eles.

Neste contexto, o principal objetivo do Cave2, atual versão do projeto Cave, é a criação de um ambiente distribuído e colaborativo de apoio ao projeto de CIs. Para atender estes objetivos, foi proposta uma nova arquitetura para o ambiente Cave que será apresentada na seção 3.2.2.

3.2.1 Trabalho Colaborativo

O trabalho colaborativo está associado aos termos CSCW (*Computer Supported Cooperative Work*) e *groupware*. De acordo com Borges [BOR 95], o termo CSCW, surgiu na década de 80, quando foi título de uma conferência da ACM (*Association for Computing Machinery*). De acordo com Dietrich [DIE 96], alguns autores diferenciam CSCW e *groupware.* Segundo estes autores, CSCW é a área de pesquisa que estuda o trabalho cooperativo suportado por computador, enquanto *groupware* referencia os sistemas que são frutos das pesquisas na área de CSCW.

Segundo Souza [SOU 96], CSCW é uma área de pesquisa que aborda o desenvolvimento de um suporte informatizado ao trabalho em grupo. Encontram-se destacados em [FER 98] [HOL 94] [SOU 96] alguns fatores que impulsionaram a pesquisa em CSCW, são eles: a disseminação de redes de computadores nos mais variados ambientes de trabalho, a adoção de sistemas distribuídos e a necessidade de compartilhamento de recursos. Sobretudo a disseminação das redes de computadores, que segundo Dietrich [DIE 96] são o meio pelo qual são implantados os mecanismos de comunicação, compartilhamento e troca de informações. Além disso, segundo [SIE 94], as aplicações baseadas em redes de alta velocidade e redes de comunicação públicas de baixo custo devem facilitar a expansão do trabalho colaborativo.

O suporte a um espaço de compartilhamento de dados também é uma questão fundamental em um ambiente de projeto colaborativo, permitindo o armazenamento persistente dos dados de projeto e o seu acesso facilitado. Pode-se destacar um conjunto de características básicas requeridas a este suporte, tais como, controle de concorrência, controle de versões, manutenção de um histórico do uso dos dados, acesso estruturado e não estruturado, bem como gerenciamento de informações sobre grupos e membros e gerenciamento das interações entre membros.

A comunicação é o aspecto mais importante da atividade em grupo, sem o qual não há colaboração. A comunicação entre participantes de um grupo pode ser apoiada por ferramentas de comunicação síncronas e assíncronas, através de redes de computadores. A comunicação síncrona ocorre com usuários que estão ativos no sistema num mesmo momento, podendo ser realizada através de trocas de mensagens textuais (como no *talk* e *chat*) ou através de áudio e vídeo (videoconferência). Já a comunicação assíncrona não exige esta coordenação temporal, portanto, há uma defasagem entre a ação realizada por um dos participantes e sua percepção pelos demais participantes da seção. A interação assíncrona pode ocorrer através de mensagens

textuais (como correio eletrônico, *news* e documentos/hiperdocumentos), ou através de áudio e vídeo gravados.

Quanto ao local, a comunicação pode se dar entre colaboradores localizados no mesmo local ou em locais diferentes. Vale salientar que trabalhando ou não em locais diferentes as redes são essenciais da atividade de CSCW.

As dimensões espaço e temporal são as principais utilizadas na análise de atividade colaborativa auxiliada por computador. Segundo Barros [BAR 94], combinando estas duas dimensões têm-se diferentes modalidades de interação. Na tabela 3.1, apresenta-se uma taxionomia espaço-temporal para a interação em um ambiente colaborativo. As ferramentas de grupo (*groupware*) também podem ser classificadas segundo os critérios temporal e espacial [DIE 96][SOU 96].

TABELA 3.1 - Taxonomia Espaço-Temporal [UNA 91]

Taxonomia Espaço-Temporal	Mesmo Tempo	Tempo Diferente
Mesmo Local	**Face a Face**	**Assíncrona**
Local Diferente	**Distribuída Síncrona**	**Distribuída Assíncrona**

Num ambiente colaborativo, ainda há outros aspectos a serem considerados, tais como a metodologia a ser utilizada para difundir as ações de um usuário para os demais participantes do grupo e como o custo e a eficiência devem ser analisados [WIL 94].

Na seção 3.2.3 apresenta-se a proposta de suporte à colaboração no ambiente Cave. Serão abordados aspectos referentes à metodologia de colaboração e à infra-estrutura para suporte ao compartilhamento de dados e à comunicação entre projetistas.

3.2.2 Arquitetura do Cave2

A arquitetura do Cave2 baseia-se numa estrutura de rede sob o padrão TCP/IP, não necessitando mais do uso de um servidor *web* (*web server*) para armazenar dados e ferramentas, estes podem ser armazenados em qualquer máquina da rede. Na primeira versão do Cave, o acesso era realizado via HTTP e navegadores (*web browsers*) compatíveis com a linguagem Java, porém na nova arquitetura, o acesso se dá através de *sockets*, conexões TCP/IP, não necessitando do uso de um *web browser*.

A nova proposta possui duas características básicas, que podem resolver alguns dos problemas encontrados na primeira arquitetura. A primeira delas é o intenso uso de GUI (*Graphical User Interface*), permitindo que a interface das ferramentas seja criada dinamicamente. Assim como, através da interface ativa do ambiente, chamada de *Tool Launcher*, o usuário pode carregar diversas ferramentas, permitindo o uso concorrente das mesmas e o fácil compartilhamento de dados entre elas, sem a necessidade de

geração de arquivos sucessivos e *parsing*, que são procedimentos normalmente sujeitos a erros e caros do ponto de vista computacional.

A segunda característica é a estrutura de dados baseada em objetos. A arquitetura do Cave2 utiliza um modelo orientado a objetos (OO, do inglês *oriented-object*) que substituiu o armazenamento e transmissão baseado em arquivos utilizados na primeira versão. O modelo orientado a objetos permite melhor integração entre as ferramentas e suporta um armazenamento de dados mais eficiente, usando um modelo de dados unificado. Este modelo unificado garante a consistência dos dados e, quando possível, também controla as replicações e redundâncias de dados indesejáveis, problemas que aparecem quando se acessa dados de projeto através de diferentes visões fornecidas pelas diferentes ferramentas ativas. A orientação a objetos facilita também a gerencia de projeto porque permite controle de versões e autenticação de usuário de baixa granularidade, ao invés de um controle de versão e autenticação no nível de arquivos, tem-se um controle no nível de objetos.

Na redefinição da arquitetura, foram considerados os seguintes aspectos: uso de objetos para a representação dos dados de projeto, necessidade de objetos distribuídos e necessidade de acesso multi-usuário para dados de projeto. Além disso, dois aspectos foram priorizados na redefinição da arquitetura, a extensão do sistema e o reuso.

Na arquitetura do Cave2, utiliza-se o conceito de *framework* sob uma diferente perspectiva, a visão da engenharia de software, utilizando-se das vantagens dos mais novos conceitos desta área que não eram conhecidos na primeira geração dos ambientes de projeto. No domínio da engenharia de software, um *framework* é uma arquitetura para a construção de sistemas de software reutilizáveis. Segundo Johnson [JOH 97], um *framework* é um projeto de software reutilizável expresso como um conjunto de classes abstratas e pelo modo como suas instâncias colaboram.

No campo da automação de projeto, tais *frameworks* de software facilitam a definição de modelos de dados (representação de primitivas de projeto), bem como facilitam o desenvolvimento de novas ferramentas de projeto.

Baseado neste conceito de *frameworks*, a arquitetura do Cave2 é focada numa biblioteca de códigos OO reutilizáveis e na extensão desta biblioteca, sendo as classes contidas na biblioteca utilizadas para modelar as ferramentas de automação de projeto e as primitivas de projeto [IND 2002a]. No capítulo 4, são abordados conceitos de *frameworks* e técnicas para a construção de um projeto de software reutilizável.

Além disso, a abordagem de *framework* de software facilita a modelagem dos dados de projeto usando objetos. A relação entre os objetos que irão representar os dados de projeto pode ser definida usando padrões de projeto de software [GAM 2000] [LAR 99]. O uso de padrões de projeto de software garante uma aplicação mais flexível, bem como aumenta as possibilidades de reuso e facilita a manutenção da aplicação.

Neste trabalho, o Cave2 foi dividido em três partes para permitir uma melhor compreensão da sua estrutura, são elas: *Cave Framework*, *Cave GUI* e *Service Space*.

O *Cave Framework* é um *framework* de software reutilizável que possui dois conjuntos de classes Java. O primeiro deles é uma biblioteca de classes para criação de ferramentas de projeto. Nesta biblioteca há classes que compõem os módulos das ferramentas de CAD já implementadas e incorporadas ao Cave, além de um conjunto extensível de primitivas GUI que são utilizadas para montar a interface entre o usuário e a nova ferramenta. Esta biblioteca facilita o desenvolvimento de novas ferramentas de projeto *internet-enabled*, bem como a incorporação destas ao ambiente Cave. Quando uma ferramenta é carregada, objetos destas classes são instanciados. O segundo conjunto que compõe este *framework* possui classes usadas para modelagem de primitivas de projeto e são instanciadas quando o usuário interage com as ferramentas de projeto.

A *Cave GUI* é composta pelo *Tool Launcher* e por uma interface de comunicação. Através do *Tool Launcher*, o usuário pode conectar-se a um servidor e invocar as ferramentas disponíveis neste servidor. A interface de comunicação permite a troca de mensagens entre os projetistas, representando o canal de comunicação do ambiente. Na seção 3.2.3.3 a comunicação entre projetistas é vista em mais detalhes.

O *Service Space* provê o controle necessário para a distribuição e o compartilhamento de recursos entre os projetistas. É um dos elementos mais importantes num ambiente de projeto colaborativo, pois permite o compartilhamento de dados e que uma alteração realizada por um projetista possa ser difundida aos demais projetistas que fazem parte do projeto.

O *Cave Framework Server* representa o *framework* de software reutilizável disponível para os desenvolvedores de ferramentas de automação de projeto. O *Cave Service Space* atualmente disponibiliza serviços de persistência de dados, de autenticação, de integração com ferramentas externas e de colaboração.

Na incorporação de novas ferramentas ao ambiente, utiliza-se dos conceitos de herança e carga dinâmica de classe de objetos. Para integrar uma nova ferramenta, o projetista deve implementar uma interface para a ferramenta, que deve herdar código de classes de objetos já implementadas e organizadas no *Cave Framework*. Fazendo isso, o projetista pode incorporar uma ferramenta no ambiente e usar as facilidades de comunicação entre ferramentas sem necessidade de escrever código adicional. Assim como, a nova ferramenta poderá usar o modelo OO e o serviço de persistência para o armazenamento dos seus dados.

A arquitetura do Cave 2 é apresentada na figura 3.3, na qual pode-se observar a presença de dois servidores, o *Cave Framework Server* e o *Cave Service Space*.

Nesta arquitetura, a biblioteca de classes do *framework* e os serviços encontram-se no *Cave Framework Server*, diferentemente do que ocorria na versão anterior, onde tínhamos a biblioteca e serviços num servidor HTTP.

A distribuição de recursos de projeto pode ser facilmente realizada, visto que a arquitetura suporta a cooperação entre vários *Framework Server* e *Service Space*. Porém, atualmente, utiliza-se uma abordagem não redundante, ou seja, tem-se um único *Framework Server* e um único *Service Space*. Assim, todas as classes da representação de projetos e módulos de ferramentas de projeto são armazenadas num único *Framework Server* e um protocolo interno é usado para o compartilhamento de tais recursos. Todos os serviços são disponibilizados num único *Service Space* e podem ser localizados através de um mecanismo de *lookup* descrito em [FRE 99], que é usado para procurar pelo serviço desejado entre os disponíveis no *Service Space*.

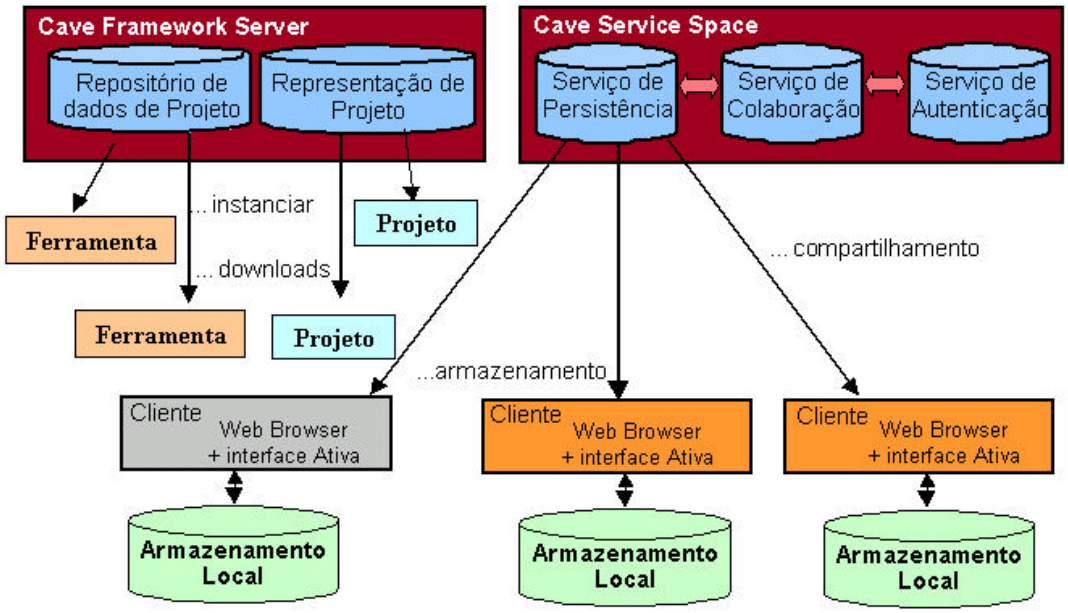


FIGURA 3.3 - Estrutura do ambiente de projeto [IND 2001]

Primeiramente, o cliente abre o *tool Launcher*, a partir da janela do *Tool Launcher* o usuário seleciona um provedor de serviços (*service provider)* e realiza sua autenticação usando seu *username* e *password.* Após, o provedor envia a lista de serviços disponíveis, podendo esta lista ser personalizada para o usuário. Ao selecionar uma ferramenta ou serviço da lista, uma conexão *socket* é aberta e o *Tool Launcher* linka dinamicamente as classes utilizadas pelo serviço solicitado e as instancia, ou seja, uma instância da ferramenta é criada na máquina cliente, possibilitando o seu uso pelo usuário. A partir da instanciação do serviço, com a interação com o usuário, as primitivas de projeto são instanciadas. A figura 3.4 representa a interação do usuário com o *Framework Server*, podendo-se observar como ocorre o processo de invocação das ferramentas e o uso das primitivas de projeto.

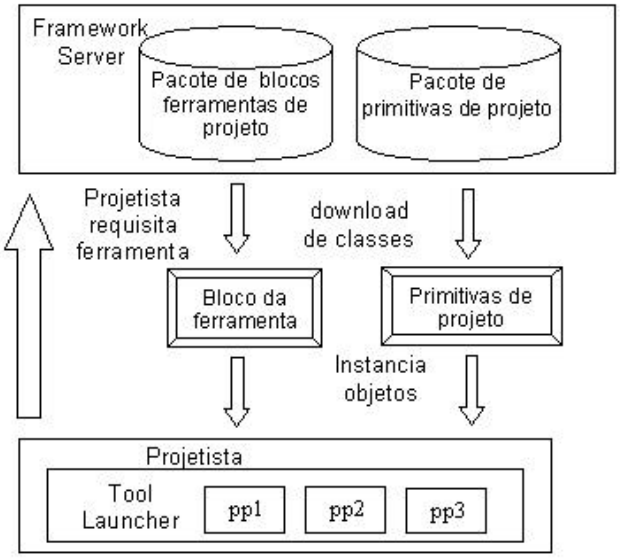


FIGURA 3.4 - Interação do usuário com o *Cave Framework Server* [IND 2002a]

Durante a interação do usuário com as ferramentas, objetos de projeto são instanciados pelo usuário. Ao término de uma sessão de projeto, estes objetos deverão ser armazenados de forma a tornarem-se persistentes, bem como acessíveis pelos projetistas envolvidos no projeto, a fim de permitir a colaboração entre os mesmos. Neste contexto, pode-se observar a necessidade de um repositório de dados, onde os dados serão armazenados e compartilhados. A persistência é um dos serviços disponibilizados pelo *Cave Service Space*. Diferentes abordagens para a implementação deste serviço foram estudadas e serão apresentadas e na seção 3.2.3.2 deste trabalho.

A figura 3.5 ilustra o processo de instanciação de primitivas de projeto, bem como o armazenamento e compartilhamento destas instâncias entre os projetistas.

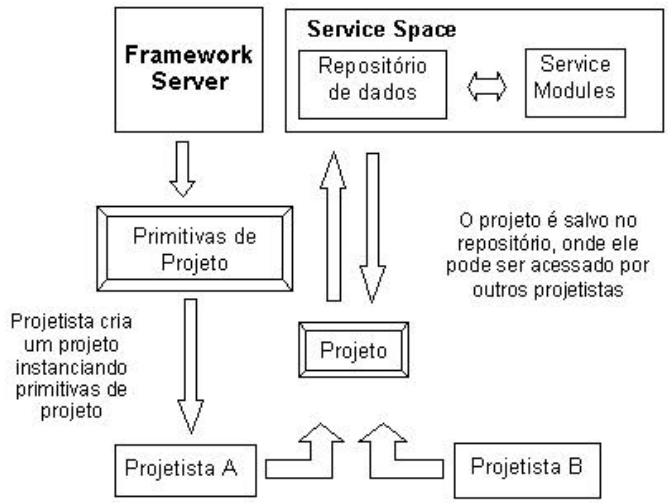


FIGURA 3.5 - Compartilhamento de Dados de projeto [IND 2002a]

Como já foi visto, o usuário cria um projeto instanciando primitivas de projeto representadas por classes Java no *Cave Framework Server*. Após a criação do projeto, é feito o *upload* dos dados para o *Cave Service Space*, onde estes são armazenados podendo ser acessados pelos demais projetistas.

A seguir é apresentada uma metodologia de colaboração e a infra-estrutura necessária para compartilhamento de dados entre projetistas.

3.2.3 Colaboração sobre o Cave

O suporte à colaboração, atualmente, é um objetivo importante do Projeto Cave e a principal motivação para o desenvolvimento da nova versão do ambiente. Algumas abordagens para suporte de trabalho colaborativo foram anteriormente apresentadas na área de EDA [BRG 2001] [KIR 2001] [LAV 97]; estes trabalhos focavam compartilhamento de IPs, treinamento remoto ou modelagem do fluxo de tarefas [IND 2002].

Porém, a abordagem utilizada para o Cave tem uma idéia diferente, na qual o modelo de colaboração deve ser genérico o suficiente para suportar a colaboração sobre diferentes tipos de objetos, visto que as ferramentas envolvidas no fluxo de projeto de um CI utilizam diferentes representações (gráficas ou textuais) para os dados de projeto. Portanto, o ambiente deve suportar o armazenamento e o compartilhamento de diferentes representações.

Um sistema colaborativo depende da metodologia de colaboração e da infra-estrutura para compartilhamento de dados e para comunicação adotadas para o ambiente Cave. Portanto, nas seções 3.2.3.1, 3.2.3.2. e 3.2.3.3 estes aspectos serão abordados.

Na seção 3.2.3.1, será apresentada uma proposta de metodologia de colaboração, porém o serviço de colaboração que será implementado e adicionado ao Cave deve ser genérico a fim de suportar outras metodologias de colaboração.

Atualmente, no projeto Cave, está sendo desenvolvida uma ferramenta chamada Homero para especificação de sistemas através de uma linguagem de descrição textual. O Homero é um editor de texto, cujo objetivo é permitir a descrição de um sistema usando linguagens de descrição de hardware ou linguagens de descrição de sistemas. O Homero foi utilizado como estudo de caso para a implementação do serviço de colaboração no ambiente Cave.

Porém, pesquisas na área de trabalho colaborativo apontam as ferramentas que utilizam representações gráficas como as mais adequadas para a implementação da colaboração. Segundo Neuwirth [NEU 90], um diagrama é mais facilmente compartilhado entre os projetistas do que uma descrição textual. Neste contexto, esta dissertação, apresenta o desenvolvimento do Blade, uma ferramenta de edição de diagramas. O Blade é um editor de diagramas hierárquico voltado para colaboração.

Assim como o Homero, o Blade também será utilizado como estudo de caso para a colaboração no ambiente Cave, principalmente, pelo fato desta ferramenta trabalhar com representações gráficas que são mais adequadas à colaboração, além de permitir a validação dos dois modos de colaboração, visualmente acoplado e visualmente desacoplado, propostos por [IND 2001] e que serão apresentados na seção a seguir.

Após a validação do suporte à colaboração, outras ferramentas do ambiente poderão utilizar este serviço de colaboração, permitindo que os projetistas trabalhem colaborativamente em todas as etapas do fluxo de projeto.

3.2.3.1 Metodologia de Colaboração

Um aspecto importante num sistema colaborativo é a metodologia de colaboração, enquanto o repositório de dados possibilita o compartilhamento de dados, a metodologia define a interação entre os usuários que irão desenvolver um trabalho em conjunto, bem como a interação entre o usuário e os dados compartilhados.

Nesta primeira versão do Cave colaborativo, a metodologia de colaboração baseia-se em *Pair Programming* [WIL 2000]. Esta escolha baseou-se nos sucessos reportados no domínio da engenharia de software referentes ao uso desta metodologia. *Pair Programming* [WIL 2000] é usado como metodologia de colaboração juntamente com a técnica conhecida como *EXtreme Programming* [BEC 99]. Porém, a fim de superar algumas limitações da proposta original de *Pair Programming*, uma adaptação chamada *Paar programming* foi proposta em [IND 2001]. Esta adaptação visa permitir a colaboração entre usuários remotos, bem como entre grupos maiores de usuários (colaboradores).

No contexto do ambiente Cave, estão sendo implementados dois modos distintos de colaboração: visualmente acoplado e visualmente desacoplado. Para a definição destes modos utiliza-se como estudo de caso uma ferramenta de edição de esquemáticos colaborativa, na qual um diagrama é compartilhado entre vários projetistas.

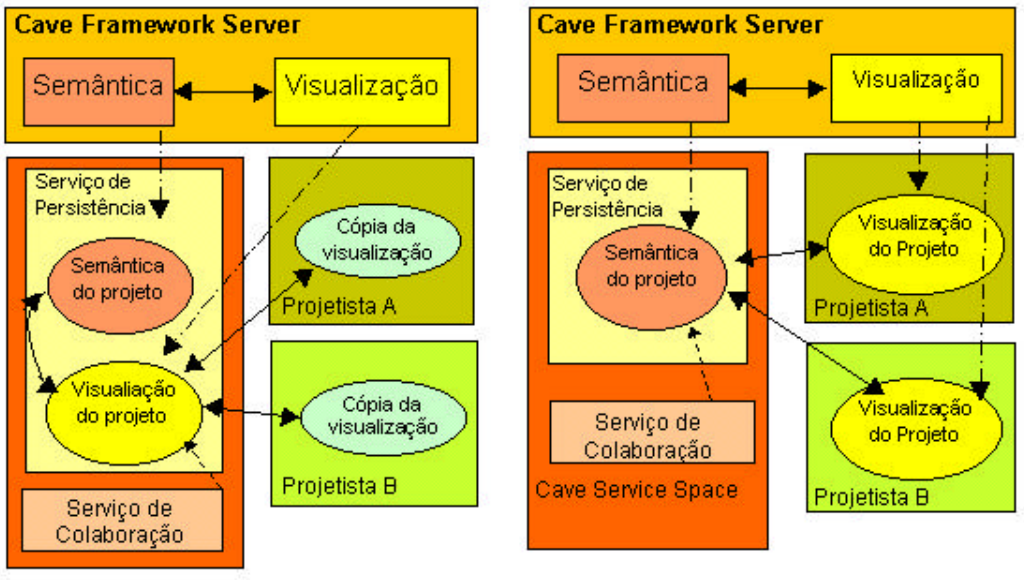
No modo visualmente acoplado, utiliza-se um único modelo, uma única visão e inúmeros controladores, um para cada projetista. Neste caso, os projetistas têm a mesma visão dos dados de projeto, qualquer alteração feita no diagrama por um dos projetistas deve ser repassada aos demais projetistas, mesmo que estas alterações não tenham alterado a semântica do diagrama, assegurando assim, a consistência das visualizações.

Na implementação do modo visualmente desacoplado, utiliza-se um único modelo e inúmeras visualizações e controles associados, uma visualização para cada projetista no grupo colaborativo. Neste caso, o projetista não compartilha a visão dos dados de projeto, permitindo que todos eles tenham visualizações distintas de um mesmo diagrama, mas todos os diagramas estão sob um mesmo modelo. Portanto,

quando um projetista faz uma alteração no diagrama, esta só será repassada para os demais projetistas do grupo se alterar também o modelo associado, ou seja, se alterar a semântica do diagrama.

A semântica do projeto e sua representação textual / gráfica são modeladas por objetos diferentes, prática que é chamada neste trabalho de separação de conceitos. Por exemplo, a semântica representa um modelo e a representação gráfica a visualização deste modelo. Usando um editor de esquemáticos, como estudo de caso, pode-se dizer que a semântica corresponde ao *netlist* de um circuito e a representação gráfica corresponde aos dados referentes à visualização do circuito. Esta abordagem permite que os projetistas tenham visualizações diferentes de um mesmo bloco de projeto, ou seja, possibilita a implementação do modo visualmente desacoplado.

Por exemplo, numa ferramenta colaborativa de edição de esquemáticos, operando no modo visualmente acoplado, qualquer alteração, mesmo que altere somente a posição de um bloco ou porta lógica, é repassada a todos os projetistas envolvidos no projeto. No modo visualmente desacoplado, somente as alterações que alteram o *netlist* gerado a partir deste esquemático são repassadas aos demais projetistas, por exemplo, quando uma porta ou conexão é excluída ou incluída. As figuras 3.6 (a) e 3.6 (b) ilustram respectivamente, o funcionamento dos modos visualmente acoplado e visualmente desacoplado.



(a) (b)

FIGURA 3.6 - (a) Visualmente Acoplado (b) Visualmente Desacoplado [IND 2001]

3.2.3.2 Compartilhamento de dados

A maioria das aplicações requer o armazenamento e a recuperação de informações em um mecanismo de armazenamento persistente, tal como um banco de dados relacional ou orientado a objetos [LAR 99]. No contexto das ferramentas de CAD isso não é diferente. Portanto, a estratégia de representação dos dados e a arquitetura do repositório de dados são pontos-chave para o desenvolvimento de um ambiente de projeto. Hoje em dia, um ambiente de projeto é composto por um conjunto heterogêneo de ferramentas. Recentes trabalhos padronizam um conjunto de funções para comunicação entre as ferramentas de projeto e o repositório de dados [IND 2002].

No contexto do suporte à colaboração, ou seja, à interoperabilidade entre os projetistas, um repositório de dados deve ser definido, e sua implementação deve suportar o compartilhamento dos dados, acesso a dados concorrentes, múltiplas visões, etc. Além disso, a implementação deve ser flexível o suficiente para suportar diferentes metodologias de colaboração, assim como a incorporação de novas metodologias, mesmo depois do repositório de dados já conter objetos persistentes.

Nesta seção, serão analisadas algumas alternativas para a implementação de um serviço de persistência que será disponibilizado através de um *framework* para persistência. Segundo Larman, um *framework* para persistência é uma estrutura de classes reutilizáveis, e usualmente extensíveis, que fornecem serviços para objetos persistentes [LAR 99].

De forma a tornar extensível o serviço de colaboração do Cave, o repositório deve trabalhar com objetos de qualquer tipo. Assim, não se limita o domínio de uso desta implementação e permite-se que outras ferramentas que manipulam objetos diferentes possam ser facilmente incorporadas ao Cave, bem como possam utilizar o seu serviço de colaboração.

Em [IND 2002] [IND 2002a] foram analisadas diferentes tecnologias para a implementação do repositório de dados: banco de dados relacional, chamados RDBMS (*Relational Database Management System),* banco de dados orientado a objetos, conhecidos OODMS (*Object-oriented Database Management System)* e espaço de compartilhamento de objetos. Foram analisados dois tipos de banco de dados OO, um deles utiliza cópias locais e o outro utiliza referências. Este último é também conhecido como banco de dados OO de instância única, porque as aplicações não tem uma cópia local dos dados, somente uma referência para os dados armazenados no repositório.

Após o estudo das tecnologias para implementação de repositório de dados, optou-se pela realização de dois protótipos do serviço de colaboração para o Cave. O primeiro deles desenvolvido por Sawicki [SAW 2002] baseia-se num serviço de persistência distribuída e utiliza *Jini* e *JavaSpaces* [FRE 99]. O segundo protótipo usa um banco de dados orientado a objetos de instância única e para sua implementação será utilizado o banco de dados *Ozone* [MEU 00]. As implementações destes protótipos

utilizam diferentes tecnologias para persistência de dados, visando uma posterior comparação entre as alternativas estudadas.

3.2.3.3 Comunicação entre Projetistas

Além do compartilhamento dos dados, um canal de comunicação é necessário para permitir a colaboração entre os projetistas. Atualmente no Cave há um serviço de troca de mensagens de texto síncronas, disponibilizado através da ferramenta Cadena. Assim como, pesquisas estão sendo realizadas com o intuito de criar um serviço de troca de mensagens de voz possibilitando uma maior interação entre os projetistas. Este serviço permitirá que os projetistas troquem mensagens de voz de forma assíncrona, sendo que as mensagens enviadas são armazenadas para posterior leitura pelo destinatário. Posteriormente, pretende-se ainda, possibilitar a troca de mensagens de voz síncronas.

4 Modelagem Orientada a Objetos

Na modelagem de dados do editor que foi desenvolvido ao longo deste trabalho, três aspectos foram fortemente considerados: a extensibilidade, a reusabilidade e a colaboração. Neste projeto, utilizou-se uma modelagem orientada a objetos, visto que o uso desta metodologia no desenvolvimento de uma aplicação facilita a reusabilidade tanto de código como de projeto, além de facilitar também a extensão, dando maior flexibilidade à aplicação.

O paradigma de projeto orientado a objetos é muito utilizado atualmente, principalmente porque suporta o projeto e manutenção de sistemas complexos. Este paradigma baseia-se nos conceitos de classe, objetos, herança, encapsulamento e ligação dinâmica [PRE 96].

Neste capítulo, apresenta-se um resumo da metodologia de projeto OO (orientado a objetos) e descrevem-se técnicas utilizadas no projeto de sistemas OO reutilizáveis, tais como: análise e projeto orientado a objetos, *frameworks* e padrões de projeto de software. Na descrição destas técnicas utiliza-se a linguagem de Modelagem Unificada (UML, em inglês, *Unified Modeling Language*) [FUR 98], que é a notação padrão usada para modelagem orientada a objetos.

Dentre as técnicas estudadas neste capítulo, destacam-se os padrões de projeto de software. Alguns destes padrões, sobretudo aqueles considerados de interesse para este domínio de aplicação, são apresentados neste capítulo e revisitados no capítulo 6, onde a aplicabilidade destes padrões é demonstrada no contexto da implementação da ferramenta.

4.1 Projeto Orientado a Objetos

O uso da tecnologia orientada a objeto está proliferando no desenvolvimento de software. Análise e projeto orientado a objetos são pontos críticos para criar sistemas orientados a objetos robustos e de fácil manutenção [LAR 99]. Além disso, aumenta a reusabilidade e flexibilidade da aplicação.

Durante a análise orientada a objetos, há uma ênfase na descoberta e na descrição dos objetos – ou conceitos – do domínio da aplicação. Enquanto, durante o projeto orientado a objetos existe uma ênfase na definição de elementos lógicos de software, os quais possuem atributos e métodos que em última instância serão implementados em uma linguagem de programação orientada a objetos. Finalmente, durante a construção, usando programação orientada a objetos, os componentes do projeto são implementados como classes em uma linguagem tal como, C++, *Java*, *Smalltalk*, etc.

Existem muitas atividades e artefatos possíveis na análise e no projeto, bem como um rico conjunto de princípios e diretrizes. O primeiro passo é definir o que o sistema deve fazer. Em termos de metodologia de análise e projeto orientado a objetos, isso é análogo à análise de requisitos. Os requisitos são descrições das necessidades ou dos desejos para um produto que está sendo projetado. Utilizam-se descrições no formato de diagramas de caso de uso (*Use Case*) para expressar os requisitos do sistema, estes diagramas fazem parte da linguagem UML.

Antes de proceder a um projeto lógico de como uma aplicação de software funcionará, é necessário investigar e definir seu comportamento como uma "caixa-preta". O comportamento do sistema é uma descrição do que o sistema faz, sem explicar como ele faz. Uma parte dessa descrição é um diagrama de seqüência do sistema [LAR 99]. Os diagramas de seqüência também fazem parte da UML.

Posteriormente, deve-se definir quem faz o quê no sistema e como os componentes do sistema colaboram para realizar um processo. Esta atividade está ligada ao projeto orientado a objetos e tem por finalidade a atribuição de responsabilidades. A atribuição de responsabilidades pode ser destacada como a atividade que tem efeito mais profundo sobre a robustez, a facilidade de manutenção e a reusabilidade dos componentes de software. Outra atividade importante é encontrar objetos e abstrações adequadas. Ambos são críticos, porém a atribuição de responsabilidade é a habilidade mais difícil de dominar.

A atribuição de responsabilidades e a interação entre objetos de software são expressas, freqüentemente, através de diagramas de classes de projeto e diagramas de colaboração – diagramas que mostram respectivamente, a definição de classes e o fluxo de troca de mensagens entre objetos de software.

Na fase de construção do software, os diagramas de classe são traduzidos para as definições de classes e os diagramas de colaboração para métodos, sendo esta tradução relativamente direta. Ainda existe muito espaço para decisões, mudanças de projeto e exploração, durante a fase de programação, porém a arquitetura geral e as decisões principais, idealmente, devem ter sido completadas antes da fase de codificação.

4.2 Arquitetura de *Frameworks*

Segundo Johnson [JOH 92], um *framework* é constituído por um conjunto de classes abstratas e componentes que, em conjunto, definem um projeto abstrato para uma família de problemas relacionados. Gamma [GAM 95], por sua vez, define que um *framework* é um conjunto de classes cooperantes que constituem um projeto reutilizável para uma classe específica de *software*.

Um *framework* define a arquitetura de um domínio de aplicação. Ele define a estrutura global da aplicação, a sua divisão em classes e objetos, as responsabilidades

chave de cada parte, como as classes e os objetos que colaboram e a seqüência de controle. Ambos os autores citados acima destacam aspectos abstratos do projeto como o aspecto central que caracteriza os *frameworks*.

Pree [PRE 94] ressalta os aspectos estruturais de classes e de sua utilização por especialização para a construção de aplicações. Segundo o autor, um *framework* é constituído por um conjunto de blocos de construção prontos para serem utilizados e outros semi-acabados. A arquitetura global, isto é, a composição e interação de blocos, são predefinidas também. Produzir uma aplicação específica usualmente envolve a adaptação de componentes a necessidades específicas, implementando alguns métodos em subclasses das classes do *framework*.

Goldberg [GOL 95], por sua vez, centra-se nos aspectos operacionais, ou dinâmicos de um *framework*, tomando em consideração o aspecto da estrutura de controle que esse *framework* fornece para a construção de aplicações. Segundo ele, um *framework* é um conjunto de objetos que interagem e que fornecem um conjunto bem definido de serviços. É uma forma bem definida na qual o controle é transferido entre esses objetos.

Por último, [DEU 89] define o conceito de *framework* levando em consideração o contexto de utilização das abstrações e de extensão dessas abstrações. Para ele, um *framework* pode prover uma instância X de uma classe própria como parâmetro a uma instância Y de uma outra classe existente, com a expectativa de que Y enviará um conjunto de mensagens preestabelecidas a X. Neste caso, Y é considerado como o *framework*, enquanto X é considerado um cliente interno para diferenciá-lo dos clientes externos que interagem com a combinação. Um cliente pode definir uma subclasse XC de uma classe existente YC, fornecendo funcionalidade adicional ou especializada. Outra vez, a classe existente é considerada como o *framework* e o cliente como cliente interno.

Estas definições enfatizam diferentes aspectos ou perspectivas, tanto da estrutura como da natureza operacional dos *frameworks*. A definição de Deutsch [DEU 89], é a mais precisa em termos dos mecanismos que a orientação a objetos oferece para suportar reutilização de funcionalidade. Esta definição, entretanto, não destaca os aspectos de abstração inerentes aos *frameworks*, nem os aspectos de domínio de aplicação, fazendo referência implícita à predefinição de uma estrutura de controle. Já na definição de Goldberg [GOL 95], a estrutura de controle é ressaltada, mas, neste caso, o aspecto de abstração está implícito na definição. Pree, Gamma, Helm, Johnson e Vlissides [PRE 94], incluem o conceito de arquitetura como um dos aspectos essenciais que caracterizam um *framework*, o qual está intimamente relacionado com a noção de estrutura de controle bem definida, mencionada por Goldberg [GOL 95].

O conceito de domínio de aplicação está relacionado com o objetivo de um *framework*, que é o fornecimento de uma estrutura de classes reutilizável para produzir aplicações semelhantes. A abstração caracteriza o processo de construção de um

framework, enquanto que o conceito de arquitetura caracteriza o resultado deste processo.

Desta forma, pode-se dizer que um *framework* é um conjunto de classes concretas e abstratas. A interface entre elas tem por objetivo servir como uma aplicação semi-acabada, utilizada como base para o desenvolvimento de novas aplicações. Aplicações baseadas em um *framework* são construídas customizando suas classes [CAY 2001]. Segundo Pree [PRE 95], reusar um *framework* significa adaptar sua estrutura para necessidades específicas, sobrescrevendo métodos de algumas classes nas subclasses.

Para Deutsch, [DEU 89], a reutilização de elementos de *software* pode ocorrer ao nível de código, ou de projeto. A reutilização de código consiste na utilização direta de trechos de código já desenvolvidos. A reutilização de projeto consiste no reaproveitamento de concepções arquitetônicas de uma aplicação em outras, não necessariamente com a utilização da mesma implementação [SIL 96].

Um *framework* é um projeto genérico em um domínio que pode ser adaptado a aplicações específicas, servindo como um molde para a construção de aplicações. Um *framework* pode fornecer um alto nível de reutilização, pois além de fornecer reutilização de código, oferecido por classes e objetos, também oferece reutilização de projeto [CAY 2001].

4.2.1 Desenvolvimento de *Frameworks*

Segundo Pree, [PRE 95], a construção de um *framework*, utilizando-se da abordagem de projeto orientado a pontos adaptáveis, consiste na identificação dos pontos fixos (*frozen spots*) e dos pontos adaptáveis (*hot spots*) do domínio de uma aplicação. Os pontos fixos correspondem às partes comuns das aplicações correlatas, enquanto que os pontos adaptáveis são as partes que podem ser estendidas para cada aplicação específica, dando ao *framework* a capacidade de ser flexível e moldar-se a diferentes aplicações. Esta flexibilidade é obtida através da utilização dos metapadrões.

A etapa inicial consiste na identificação e definição da estrutura de classes do *framework*. A etapa seguinte consiste na identificação dos pontos adaptáveis. É preciso identificar os aspectos que variam de aplicação para aplicação, assim como o grau de flexibilidade desejado. A etapa final consiste em um refinamento da estrutura do *framework*. Ao final, o *framework* é avaliado para verificar se os pontos adaptáveis oferecem o grau de flexibilidade desejado. Caso isto não ocorra, retorna-se à etapa de identificação de pontos adaptáveis e reprojeta-se o *framework*. Reprojetar um *framework* consiste na modificação da estrutura definida inicialmente, de modo a obter a flexibilidade desejada.

Geralmente, na construção de *frameworks* reutilizam-se vários padrões de projeto. Os padrões de projeto são microarquiteturas reutilizáveis que contribuem para a definição da arquitetura global do sistema. Segundo Gamma [GAM 93], os padrões de projeto suportam o desenvolvimento de componentes de software orientado a objetos extensíveis. Estes padrões representam uma estratégia complementar aos métodos de análise e projeto orientado a objetos, pois capturam o objetivo de um projeto, identificando classes, instâncias, métodos, suas colaborações e a distribuição de responsabilidades.

Pree [PRE 95] aborda a utilização de metapadrões na construção de *frameworks*. Metapadrões são pequenas estruturas e técnicas genéricas que são repetidamente encontradas nos padrões de projeto de software. Estas técnicas são baseadas nos princípios básicos da orientação a objetos, tais como: herança, polimorfismo, ligação dinâmica (*dinamic binding*) e classes abstratas. Estes metapadrões reúnem pequenos princípios que expressam como construir *frameworks* e garantir *hot spots* flexíveis.

O uso de herança permite que uma subclasse herde todos os métodos e variáveis de instância de uma outra classe, chamada superclasse, podendo ser adicionadas novas variáveis de instâncias na subclasse. O mesmo acontece com os métodos: uma subclasse herda os métodos de sua superclasse, assim como pode estender estes métodos ou incluir novos métodos. Portanto, herança permite adaptação sem ter que alterar o código fonte já utilizado e dá compatibilidade.

Uma das principais metas das técnicas OO é o reuso de código. Porém, algumas operações podem exigir customização a fim de atender uma necessidade particular. O polimorfismo permite a sobreposição dos métodos herdados pelas subclasses, permitindo a redefinição de um método herdado. A palavra polimorfismo é usada para uma operação que assume muitas formas de implementação, dependendo do tipo de objeto [MAR 95].

Uma potencialidade do polimorfismo é que uma solicitação para uma operação pode ser feita sem saber qual método deve ser invocado. Estes detalhes de implementação ficam ocultos. No uso de linguagens OO que possuem ligação dinâmica o compilador não determina qual dos métodos será chamado; esta definição é feita em tempo de execução conforme o tipo do objeto que está sendo manipulado.

A figura 4.1 apresenta um diagrama de classes no qual tem-se uma superclasse A e duas subclasses A1 e A2. Subclasses herdam as características (atributos e métodos) da sua superclasse. No referido diagrama, as subclasses A1 e A2 herdam os métodos X () e Y() da superclasse A. Porém, através do polimorfismo a subclasse A1 redefine o método Y( ), permitindo que os objetos desta subclasse possuam um comportamento diferente dos outros objetos da superclasse.

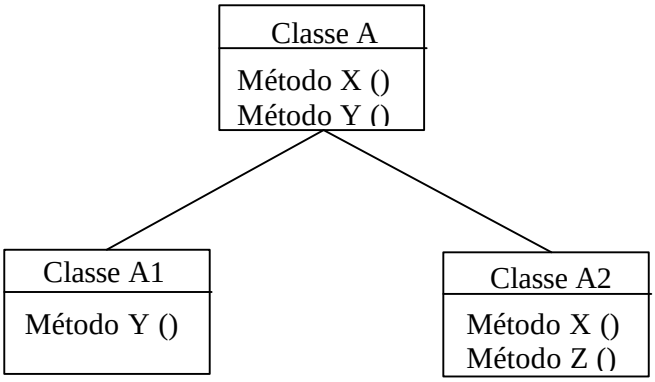


FIGURA 4.1 - Ilustração de polimorfismo [PRE 95]

Na construção de um software orientado a objetos reutilizável são necessárias combinações apropriadas de polimorfismo e ligação dinâmica através de classes abstratas e interfaces. Classe abstrata é aquela que não possui instâncias diretas, mas cujas classes descendentes, sim. Uma classe concreta é uma classe instanciável, ou seja, que pode ter instâncias diretas.

As classes abstratas organizam características comuns a diversas classes. Muitas vezes é útil criar uma superclasse abstrata para encapsular classes que participam da mesma associação ou agregação. Algumas classes abstratas surgem naturalmente no domínio da aplicação, outras podem ser introduzidas artificialmente como um mecanismo para facilitar a reutilização de código.

As classes abstratas são freqüentemente usadas para definir métodos a serem herdados pelas subclasses. Por outro lado, uma classe abstrata pode definir o protocolo para uma operação sem apresentar uma implementação correspondente, ou seja, definindo apenas o número e tipo de parâmetros e o tipo de resultado, bem como o significado semântico [RUM 94].

Um exemplo simples de uso de classes abstratas é apresentado na figura 4.2, onde se apresenta um esboço da modelagem de dados usada em um editor gráfico que manipula figuras geométricas. No referido exemplo têm-se uma superclasse chamada “Figura” e subclasses chamadas “Retângulo” e “Triângulo”. Apesar destas duas subclasses possuírem alguns atributos e métodos em comum, estas também possuem diferentes comportamentos. Por exemplo, a operação de desenho, representada pelo método desenha(), deve possuir implementações diferentes para estas duas subclasses. Portanto, a classe Figura pode ser definida como uma classe abstrata e o método desenha() pode ser um método abstrato, ou seja este não é implementado na superclasse. A definição de um método abstrato apresenta apenas seu nome, sua lista de parâmetros e o tipo de resultado, permitindo uma padronização de interface para o método.

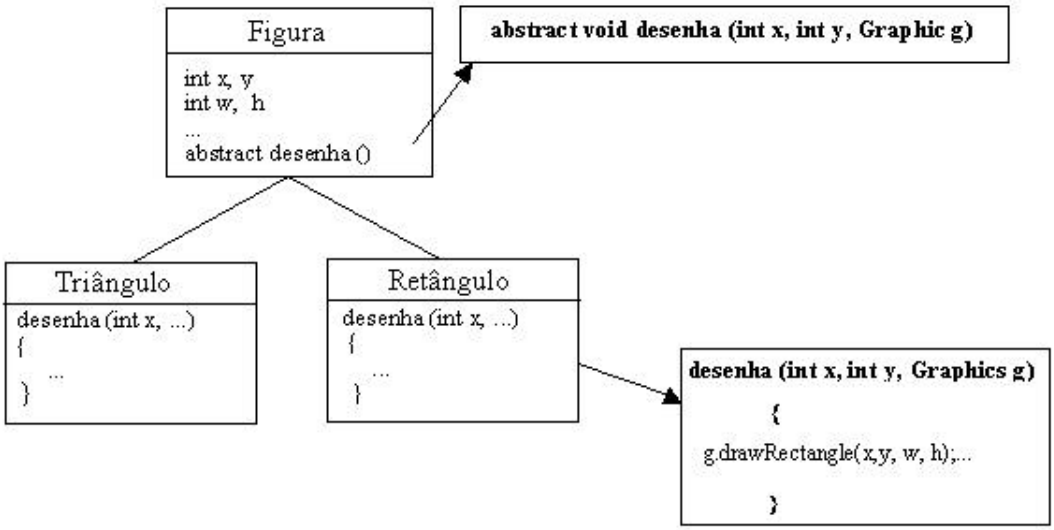


FIGURA 4.2 - Exemplo de Classes Abstratas

Interfaces são classes abstratas que descrevem o comportamento de um objeto. Porém, uma interface não possui nenhuma implementação, todos os seus métodos são abstratos e públicos por *default*. As interfaces são bastante usadas para despacho dinâmico de métodos (*dinamic binding*), ou seja, para permitir que o interpretador possa decidir qual a classe que será chamada em tempo de execução. Bem como, nas linguagens que não possuem herança múltipla, as interfaces são usadas nas situações onde a herança múltipla é necessária.

Além das técnicas apresentadas aqui, na próxima seção deste trabalho são apresentados alguns padrões catalogados por Gamma [GAM 95] e que são freqüentemente utilizados na construção de *frameworks*. Também foi analisada a listagem e classificação de padrões realizada por Cooper, que pode ser encontrada em [COO 98].

4.2.2 Uso de padrões de projeto

O projeto de um software orientado a objetos é uma tarefa árdua, principalmente quando se deseja projetar um software reutilizável. Durante o projeto, deve-se definir os objetos pertinentes, faturá-los em classes no nível correto de granularidade, definir as interfaces das classes e as hierarquias de herança e estabelecer as relações-chave entre eles. Embora o projeto deva ser específico para uma aplicação, este também deve ser genérico o suficiente para atender futuros problemas e requisitos.

Projetistas experientes costumam reutilizar soluções que funcionaram bem no passado. Conseqüentemente, pode-se encontrar padrões de classes e de comunicação entre objetos, que reaparecem freqüentemente em muitos sistemas. Estas soluções

podem vir a constituir-se em padrões de software que se bem documentados podem ser utilizados por projetistas menos experientes [GAM 2000]. Existem vários autores que se dedicam a catalogar estes padrões de projeto, registrando assim a experiência em projeto orientado a objeto. Dentre eles podemos citar E. Gamma, R. Helm, R. Johnson e J. Vlissides [GAM 95], K. Beck [BEC 94] [BEC 94a], P. Coad [COA 95], entre outros.

Os padrões de projeto tornam mais fácil reutilizar projetos e arquiteturas bem-sucedidas. Esta idéia baseia-se no fato que expressar técnicas testadas e aprovadas tornam-nas mais acessíveis para o desenvolvedores de novos sistemas. Os padrões de projeto auxiliam na escolha de alternativas de projeto que tornem o sistema reutilizável e a evitar alternativas que comprometam a reutilização. Além disso, os padrões podem melhorar a documentação e a manutenção do sistema ao estabelecer uma especificação explícita de interações de classes e objetos e o seu objetivo adjacente [GAM 2000].

Um padrão de projeto nomeia, abstrai e identifica os aspectos-chave de uma estrutura de projeto comum para torná-la útil à criação de um projeto orientado a objetos reutilizável. O padrão de projeto identifica as classes e instâncias participantes, seus papéis, colaborações e a distribuição de responsabilidades. Cada padrão de projeto focaliza um problema ou tópico particular de projeto orientado a objetos, descrevendo quando o padrão pode ser aplicado, as conseqüências e benefícios de sua utilização [GAM 2000].

A chave para maximização da reutilização está na antecipação de novos requisitos e de mudanças nos requisitos existentes e, também, em projetar sistemas de modo que eles possam evoluir de acordo. Portanto, para obtermos um sistema robusto e flexível é importante que durante o projeto leve-se em conta as mudanças que possam vir a ser necessárias ao longo da vida do sistema. Estas mudanças podem acarretar em redefinições, reimplementações e retestagem do sistema, processo conhecido como reprojeto ou reformulação. Esta reformulação afeta muitas partes do sistema de software e invariavelmente mudanças não antecipadas são muito caras. Em [GAM 2000], são apresentadas algumas causas comuns de reformulação juntamente com os padrões de projeto que as tratam.

Os padrões, portanto, podem ajudar a incorporar flexibilidade ao software. Quão importante esta flexibilidade é para o sistema, depende do tipo de software que está sendo construído. No caso do projeto de um *framework*, a flexibilidade é altamente desejada.

4.2.2.1 Modelo de Interação MVC

Um dos blocos de construção que um *framework* de interface normalmente utiliza é o modelo de interação MVC (*Model–View–Controller*) [GLE 88]. Esta tríade de classes Modelo / Visão / Controlador (MVC) é usada para construir interfaces para usuários em Smaltalk-80 [GAM 2000].

O modelo MVC utiliza três classes, conforme se pode observar na figura 4.3. A classe Modelo (*Model)* gerencia os dados de domínio específico que serão representados e manipulados pela aplicação GUI. A classe Vista (*View)* é responsável pela representação destes dados na tela. A classe Controlador (*Controller*) é responsável por aceitar entrada de dados (*mouse* e teclado) e por passar as mensagens apropriadas às classes Modelo e Vista para habilitar a edição do modelo de dados.

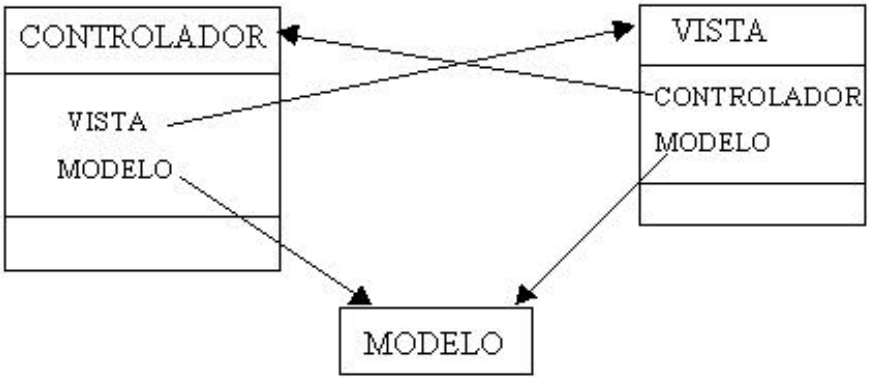


FIGURA 4.3 - Modelo de Interação MVC

Por exemplo, se a aplicação for um processador de textos: o Modelo armazena os dados textuais, a Vista representa o texto numa janela de edição e o Controlador gerencia a digitação. Já em uma aplicação gráfica o Modelo armazena informações sobre objetos gráficos, a Vista representa a figura numa janela de edição e o Controlador gerencia a edição destas figuras [CAY 2001].

Uma aplicação é criada a partir de um *framework* MVC gerando-se subclasses. Especializações nas subclasses promovem o refinamento das classes MVC para habilitar a visualização e edição dos dados do modelo. Vistas e Controladores devem ter apenas um Modelo, mas Modelos podem ter várias Vistas e Controladores.

Atualizações MVC

Vistas e controladores são geralmente fortemente acoplados. A razão torna-se óbvia se for considerada a diferença entre editar dados numa planilha ou num aplicativo de diagramas gráficos. Os tipos de interações envolvidas em uma vista de uma planilha são completamente diferentes das interações envolvidas numa vista gráfica.

A possibilidade de múltiplas vistas introduz o problema de manter todas as vistas consistentes com o estado dos dados, quando editados através de uma das vistas. Isto é chamado *problema de atualização MVC*, o qual é usualmente gerenciado mantendo-se uma lista de todas as vistas. Assim, uma atualização pode ser propagada por todas as vistas. A atualização MVC pode ser observada na figura 4.4. Cada vez que

uma vista é alterada, uma mensagem é propagada para todas as outras vistas através do modelo, esta mensagem é responsável pela atualização das demais vistas.

O Modelo contém dados de domínio específicos que são exibidos e manipulados por uma aplicação. Eles podem ser uma faixa de inteiros (representando contadores e termômetros), *arrays* de caracteres (editores de texto simples), listas dinâmicas, registros ou outras estruturas de dados complexas [CAY 2001].

Uma vista controla a representação visual de todo ou partes de um modelo específico. Funções comuns como *refresh* ou *scroll* de uma janela podem ser mantidas nesta classe, mas funções específicas das aplicações devem ser implementadas nas subclasses, pelos desenvolvedores da aplicação. Vistas podem representar todo o modelo ou somente certos aspectos. A vista deve saber sobre o modelo que está representando, mas não precisa de conhecimento sobre qualquer uma das outras vistas.

Controladores são associados tanto a vistas quanto a modelos. Um controlador aceita entradas do usuário através de vários dispositivos como teclado e *mouse* e envia mensagens apropriadas ao modelo e vista. Os controladores devem saber sobre o modelo e vista que estão associados, mas não precisam de informação sobre outros controladores.

Vistas e controladores são associados com seus modelos quando são criados. Cada vez que os dados do modelo são modificados, este envia uma mensagem *broadcast* de mudança para todos os seus dependentes. Cada vista e controlador dependentes podem acessar os dados do modelo e atualizar a si mesmos apropriadamente. Parâmetros passados com a mensagem de mudança permitem às vistas e controladores decidir se necessitam de atualização.

Comunicações entre o Modelo e o par Vista-Controlador são capturadas nas classes abstratas. Assim a arquitetura MVC pode ser reutilizada para novas vistas e aplicações. Isto pode economizar um considerável esforço de projeto cada vez que o *framework* MVC é utilizado.

Arquitetura MVC (Model-View-Controller)

O principal objetivo da arquitetura MVC é a separação de vistas e modelos do modo como estes interagem. Esta separação (dados e aplicação) gera maior flexibilidade e grande possibilidade de reuso [GAM 2000]. A figura 4.4 ilustra a arquitetura MVC.

Esta característica do MVC de separar vistas e modelo está diretamente ligada a um problema mais geral: separar objetos de maneira que mudanças ocorridas em um possam afetar um número qualquer de objetos dependentes sem que estes precisem conhecer o objeto que foi alterado. Este projeto mais geral é descrito pelo padrão *Observer* [GAM 2000]. Este padrão será apresentado na seção 4.2.2.2.

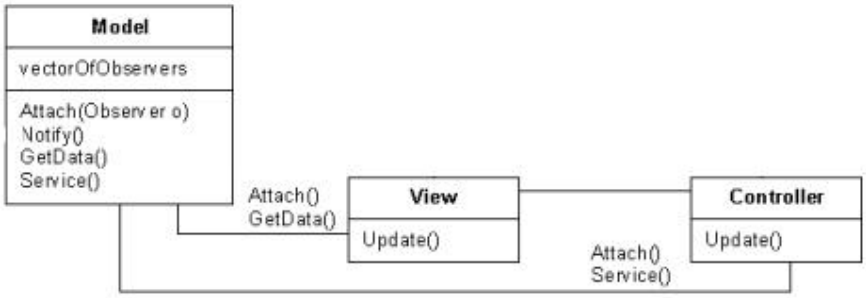


FIGURA 4.4 - Arquitetura MVC [GAM 95]

Outra característica do MVC é que as vistas podem ser encaixadas. O MVC suporta vistas encaixadas através do uso da classe *CompositeView*, uma subclasse de *View*. Os objetos da classe *CompositeView* funcionam exatamente como objetos da classe *View*. Portanto, uma vista composta pode ser usada em qualquer lugar que uma vista possa ser usada, mas ela também contém e administra vistas encaixadas. Esta característica do MVC também pode ser relacionada com um problema mais genérico: agrupar objetos e tratar o grupo como um objeto individual. Este projeto mais geral é descrito pelo padrão *Composite*.

O MVC usa outros padrões de projeto, tais como *Factory Method*, para especificar por *default* a classe controladora para uma vista e *Decorator*, para acrescentar capacidade de rolagem a uma vista. Mas os principais relacionamentos no MVC são fornecidos pelos padrões *Observer*, *Composite* e *Strategy* que serão apresentados nas próximas seções deste capítulo. Todos estes padrões foram encontrados em [GAM 2000].

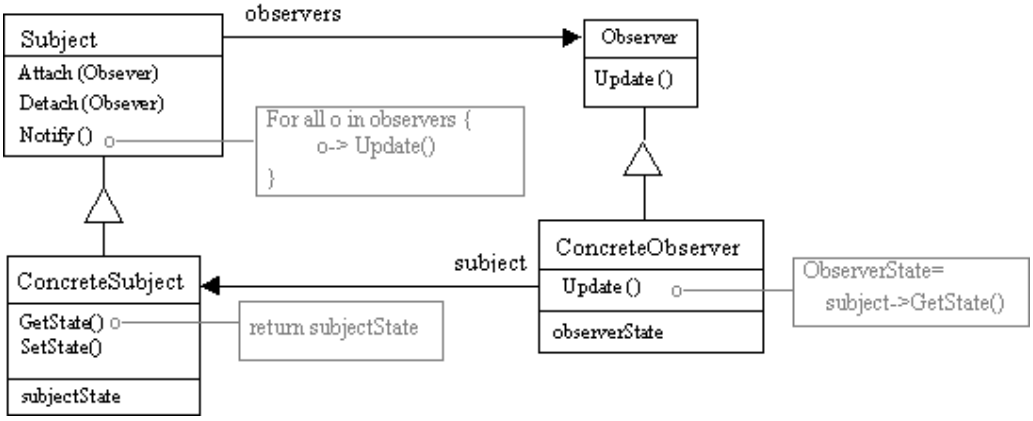
4.2.2.2 Padrão *Observer*

Quando se deseja sincronizar objetos de modo que a alteração em um objeto repercuta em outros objetos dependentes do primeiro, deve-se definir um mecanismo de acoplamento entre estes objetos. Porém, um acoplamento forte entre objetos compromete a reusabilidade dos mesmos [BEC 94]. O padrão *Observer* descreve como estabelecer estes relacionamentos sem comprometer a reutilização dos objetos envolvidos.

Os objetos-chave deste padrão são *subject* (assunto) e *observer* (observadores) e a sua estrutura é ilustrada pela figura 4.5. Um *subject* pode ter uma lista de observadores dependentes. Quando o *subject* é alterado todos os seus observadores são notificados. Quando um observador recebe uma notificação, ele decide se deve considerá-la, e neste caso, atualizar seu estado, ou se a notificação deve ser simplesmente ignorada.

Os objetos-chave deste padrão são *subject* (assunto) e *observer* (observadores) e a sua estrutura é ilustrada pela figura 4.5. Um *subject* pode ter uma lista de observadores dependentes. Quando o *subject* é alterado todos os seus observadores são notificados. Quando um observador recebe uma notificação, ele decide se deve considerá-la, e neste caso, atualizar seu estado, ou se a notificação deve ser simplesmente ignorada.

FIGURA 4.5 - Estrutura do Padrão *Observer* [GAM 2000]



O padrão *Observer* aparece em *Smalltalk Model/View/Controller* (MVC), que usa um *framework* para o ambiente de interface. Este define uma relação de dependência de objetos de um para muitos, de maneira que quando um objeto muda seu estado, todos seus dependentes são notificados e atualizados automaticamente. Este padrão define, basicamente, o funcionamento da arquitetura MVC (*Model-View-Controller*), que pode ser observado através da figura 4.6.

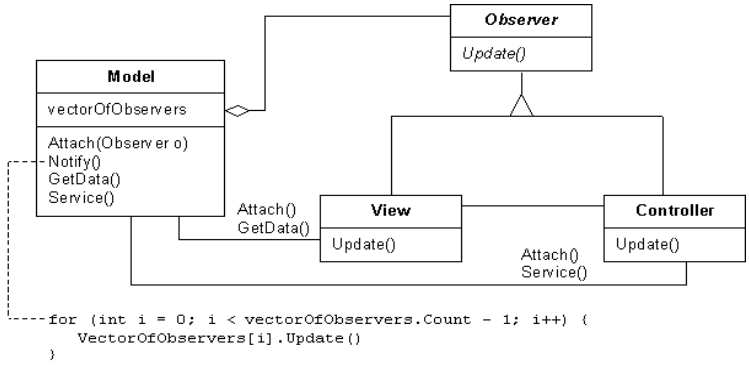


FIGURA 4.6 - Estrutura da Arquitetura MVC [GAM 95]

Podem existir múltiplas vistas de um mesmo modelo, uma vez que as vistas e os controladores estão separados da representação do modelo. A cada vista é associado um controlador. Se alguma vista altera o estado do modelo através do controlador, este notifica todas as outras vistas de que seus dados foram modificados.

Este padrão pode ser utilizado em aplicações para *web* na confecção de ferramentas que auxiliem na edição de documentos diagramáticos, pois disponibiliza mecanismos de atualização de vistas durante um processo de edição [CAY 2001].

4.2.2.3 Padrão *Composite*

O padrão *Composite* [GAM 95] é utilizado para compor vários objetos em estruturas hierárquicas. Tanto os objetos individuais como os objetos compostos são tratados de forma idêntica, como ilustra a figura 4.7.

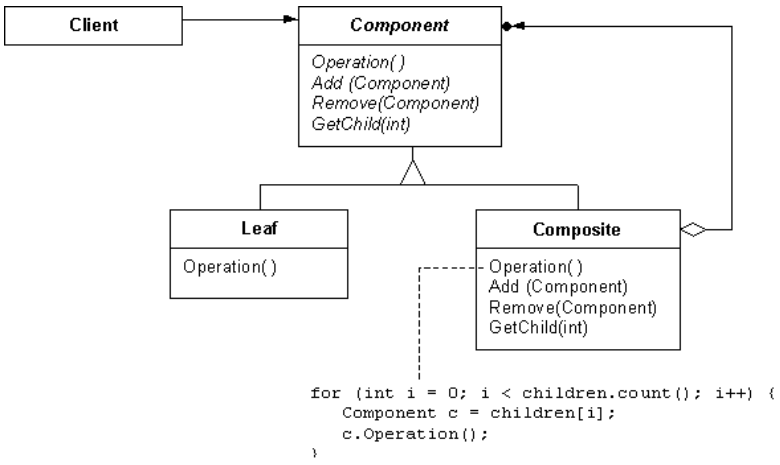


FIGURA 4.7 - Estrutura do Padrão *Composite* [GAM 95]

A classe abstrata *Component* declara uma interface abstrata para os objetos na composição, definindo o comportamento padrão para a interface comum a todas as classes. Da mesma forma, define o acesso a seus componentes filhos, e opcionalmente pode definir o acesso ao componente pai.

O objeto *Leaf* representa a folha dentro da árvore hierárquica, não possuindo filhos, definindo um comportamento para objetos primitivos. Já o objeto *Composite* define o comportamento de um objeto que possui descendentes, armazenando-os, como ilustrado na figura 4.8. As operações dos objetos compostos correspondem ao conjunto de operações executadas pelos seus filhos.

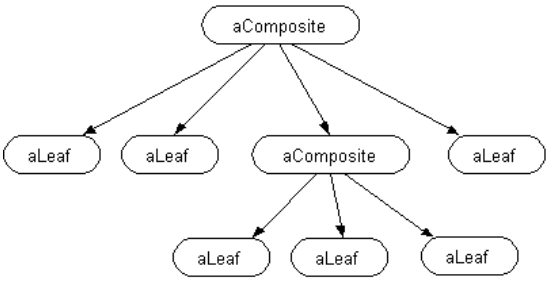


FIGURA 4.8 - Exemplo de composição de objetos [GAM 95]

O padrão torna o cliente simples, uma vez que ele não precisa saber se está tratando com um componente primitivo (*Leaf*) ou composto (*Composite*), tratando-os uniformemente. Facilmente novos objetos podem ser criados – através da composição de outros objetos existentes – sem que o código do cliente precise ser modificado.

Segundo Cayres [CAY 2001], através do padrão *Composite*, pode-se facilitar o processo de edição de documentos diagramáticos, uma vez que sua estrutura possibilita a utilização de objetos básicos, na construção de outros mais complexos, através de composição. Além disso, segundo Gamma [GAM 95], este padrão é freqüentemente combinado ao padrão *Flyweight*, apresentado na próxima seção, para representar estruturas hierárquicas, tal como um gráfico com nós de folhas compartilhadas.

4.2.2.4 Padrão *Flyweight*

O padrão *Flyweight* descreve como compartilhar objetos, permitindo o uso de objetos com pequena granularidade sem incorrer num custo proibitivo. Neste padrão, os objetos compartilhados são chamados de *flyweight* ou peso-mosca e podem ser utilizados em vários contextos simultaneamente, funcionando como um objeto independente em cada contexto.

Um *flyweight* não pode fazer hipóteses ou asserções sobre o contexto no qual opera. Ele armazena apenas as informações que são independentes de contexto, e que fazem parte de seu estado intrínseco. O estado extrínseco é dependente de contexto, portanto, não pode ser compartilhado e deve ser armazenado num objeto separado. Quando necessário, o estado extrínseco pode ser passado ao *flyweight* através de um dos seus clientes. Na figura 4.9 pode-se observar a estrutura deste padrão.

A classe *Flyweight* é uma classe abstrata que permite que os objetos *Flyweight* recebam os estados extrínsecos de seus clientes. Esta classe possui duas subclasses *ConcreteFlyweight* e *UnsharedConcreteFlyweight*. A subclasse *ConcreteFlyweight* acrescenta o armazenamento dos dados intrínsecos, se houver. Um objeto deste tipo pode ser compartilhado. Os objetos *UnsharedConcreteFlyweight* armazenam todos os seus estados inclusive estados extrínsecos, e portanto, não podem ser compartilhados.

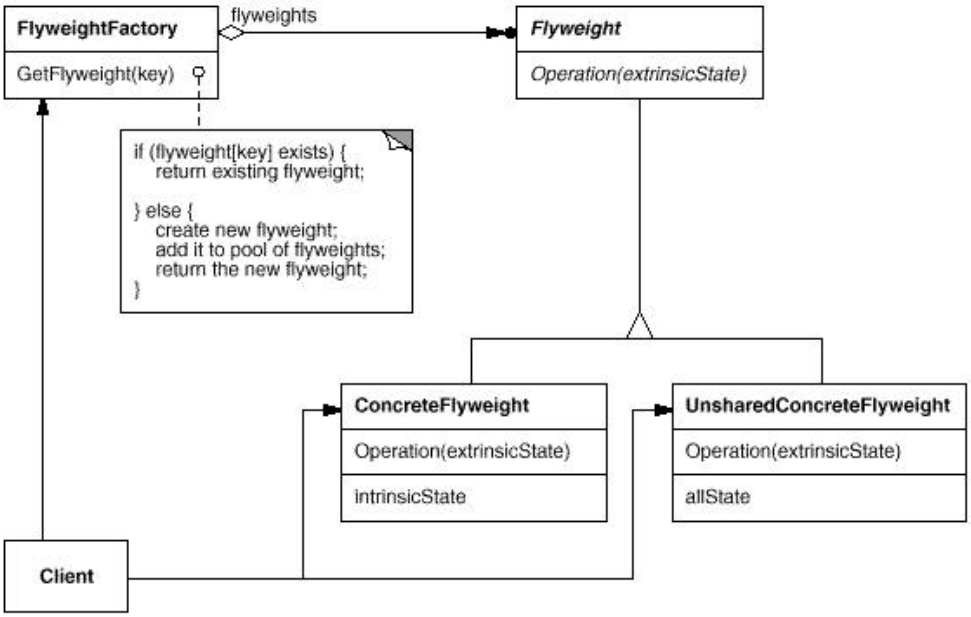


FIGURA 4.9 - Estrutura do padrão *Flyweight*

A classe *FlyweightFactory* é responsável pela gerência e criação de objetos do tipo *Flyweight*. É através dela que os objetos do tipo *Client* (cliente) podem solicitar o compartilhamento de objetos *Flyweight*. Um cliente possui referências para um ou mais objetos *Flyweights* e armazena e computa o estado extrínseco destes *Flyweights*. A figura 4.10 ilustra um diagrama no qual pode-se observar a ligação entre os objetos que fazem parte de uma estrutura *Flyweight*.

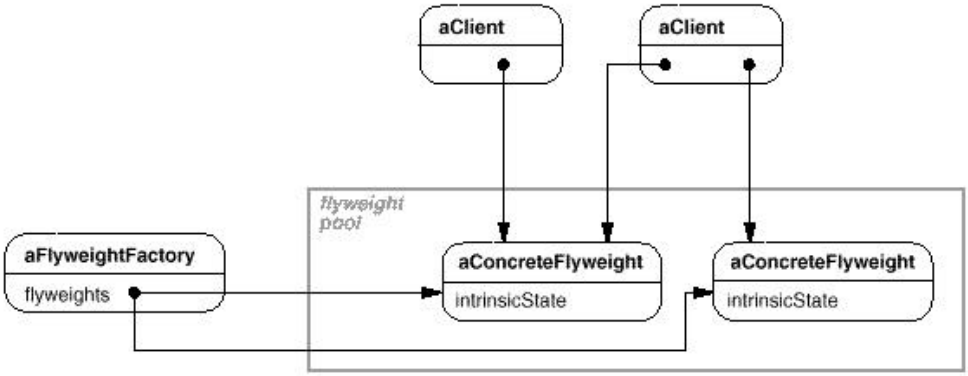


FIGURA 4.10 - Estrutura de uma *Flyweight Pool*

No diagrama de classes da figura 4.10, dois clientes compartilham um objeto *ConcreteFlyweight* e um *FlyweightFactory*. Esta estrutura pode ser utilizada para compor uma biblioteca de objetos compartilháveis, chamada de *Flyweight Pool*, onde estão armazenados os objetos *Flyweights*.

4.2.2.5 Padrão *Strategy*

O padrão S*trategy* é usado quando muitas classes diferem somente no seu comportamento. As estratégias fornecem uma maneira de configurar uma classe com um, dentre os muitos comportamentos suportados. Ou ainda, quando se necessita de variantes de um mesmo algoritmo. As estratégias podem ser usadas quando estas variantes são implementadas como uma hierarquia de classes de algoritmos.

Na figura 4.11 pode-se observar a estrutura usada para implementar o padrão S*trategy*. Onde *Strategy* (estratégia), define uma interface para todos os algoritmos suportados e *Context* (contexto) usa esta interface para chamar o algoritmo definido por uma *ConcreteStrategy*, a qual implementa o algoritmo usando a interface de *Strategy*. *Context*, por sua vez, é configurado por um objeto *ConcreteStrategy*, mantém uma referência para um objeto *Strategy* e pode definir uma interface para permitir que este acesse seus dados.

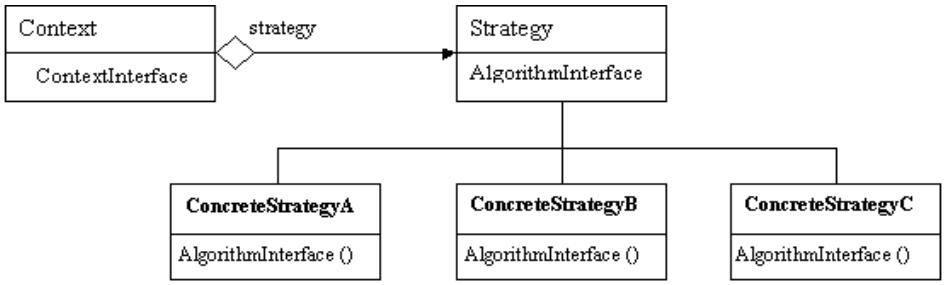


FIGURA 4.11 - Uso do padrão *Strategy* [GAM 2000]

Segundo Gamma [GAM 2000], o uso de *Strategy* é uma alternativa ao uso de subclasses e permite variar um algoritmo dinamicamente. O encapsulamento dos algoritmos permite variar o algoritmo independentemente de seu contexto, tornando mais fácil alterá-los, compreendê-los e estendê-los.

O relacionamento entre o *View* e *Controller* é um exemplo do padrão *Strategy*, também conhecido por *Policy*. Um objeto do tipo *Strategy* representa um algoritmo. Ele é útil quando se deseja substituir um algoritmo tanto estaticamente quanto dinamicamente, quando há muitas variantes do algoritmo ou quando há estruturas de dados complexas, as quais deseja-se encapsular [GAM 2000].

5 Modelo de Dados do Blade

Esta dissertação de mestrado propõe o desenvolvimento de uma ferramenta gráfica para projeto de CIs, um editor de diagramas hierárquico denominada Blade (*Block And Diagram Editor*). Este capítulo da dissertação detalha esta proposta, através de uma especificação da ferramenta e descreve a metodologia utilizada no seu desenvolvimento. O capítulo apresenta também uma análise dos requisitos e um modelo de dados inicial, correspondendo, portanto, à descrição das etapas de análise e projeto da ferramenta. Na descrição destas etapas do projeto, utiliza-se notações e diagramas da linguagem UML (*Unified Modeling Language*) [RAT 2001] [FUR 98].

5.1 Introdução

Para acompanhar o aumento da complexidade dos circuitos, os projetistas cada vez mais utilizam níveis de abstração mais altos para a especificação de sistemas. Além disso, assim como no projeto de software, o uso de especificações gráficas mostra-se também uma tendência no projeto de hardware. Neste contexto, os desenvolvedores de CAD estão constantemente desenvolvendo novas ferramentas e metodologias para apoiar o projeto de CIs.

O ambiente CAVE apresentado no capítulo 3, foi desenvolvido com intuito de integrar ferramentas de CAD, baseado principalmente na distribuição de recursos e na independência de plataforma. Atualmente, está sendo incorporado um serviço de colaboração, de modo que este ambiente possa suportar projetos colaborativos. A necessidade de projetos colaborativos pode ser justificada pelo aumento da complexidade dos sistemas, pela distribuição geográfica das empresas, entre outros fatores destacados em [IND 2002].

Este trabalho propõe o desenvolvimento de uma ferramenta gráfica para projeto de CIs, um editor de diagramas hierárquico, denominado Blade. O Blade está voltado para ambientes distribuídos, permite a operação remota e, por ser desenvolvido em Java, é independente de plataforma. O Blade visa a especificação de sistemas através de representações gráficas e está voltado ao suporte à colaboração inserido no ambiente Cave, suportando projetos colaborativos e o compartilhamento de representações diagramáticas.

Um editor de diagramas no contexto deste trabalho é uma ferramenta que permite especificar um sistema graficamente, através do uso de diagramas. A idéia do projeto é disponibilizar no Cave uma ferramenta que servirá como interface gráfica entre o projetista e outras ferramentas de CAD disponíveis no ambiente de projeto. Esta ferramenta pode ser integrada a outras ferramentas, por exemplo, a um editor de descrições HDL, desta forma, permitindo descrições tanto textuais como gráficas para

um sistema. O editor de diagramas ainda poderia ser integrado a uma ferramenta de síntese ou a uma ferramenta de estimativa de área ou potência.

Além disso, o editor de diagramas deve suportar diferentes tipos de diagramas, permitindo a entrada de um projeto utilizando descrições em diferentes níveis de abstração. Ao final da especificação, o editor gera uma descrição do sistema usando uma linguagem de descrição de hardware ou ainda, uma descrição em mais alto nível usando uma linguagem de descrição de sistemas (SDL, do inglês, *Systems Description Language*).

No contexto do projeto Cave, está sendo desenvolvido um editor de descrições HDLs, chamado Homero [HER 2001] [HER 2001a]. Futuramente, pretende-se integrar o Blade e o Homero. Esta integração visa suportar dentro do ambiente Cave, a especificação de diferentes módulos de um mesmo sistema, de forma colaborativa, através da captura de esquemáticos ou de descrições textuais, neste caso, usando uma linguagem de descrição de hardware. O editor fornecerá a visualização do sistema, através de um esquema onde podem ser visualizados os módulos do sistema e como estes estão interligados.

A integração do editor de diagramas com um editor de descrições HDL visa suportar descrições diagramáticas e textuais para os módulos do sistema. Usando uma metodologia de projeto *top-down*, o projetista inicia o projeto montando graficamente uma descrição do sistema através do editor de diagramas. Esta descrição inicial pode ser um diagrama composto por blocos interconectados que representam os módulos do sistema e suas interconexões. Desta maneira, sempre haverá pelo menos uma representação gráfica do sistema para ser compartilhada entre os projetistas. Esta abordagem suporta a especificação de um sistema usando diferentes níveis de abstração na descrição de seus módulos. Por exemplo, um módulo mais simples pode ser descrito no nível lógico e um bloco mais complexo em um nível de abstração superior. Podendo ainda, através da extensão desta ferramenta gráfica, permitir o uso de outras representações diagramáticas para a descrição dos módulos do sistema, tais como máquina de estados, diagramas UML, etc.

A visualização gráfica do sistema fornecida pelo Blade pode ser utilizada também para gerar uma visualização do *floorplanning* do CI. Neste caso, baseando-se na estimativa de área dos blocos do sistema geradas por ferramentas específicas, o Blade pode permitir a montagem da planta baixa do chip. Além disso, pode-se utilizar esta ferramenta gráfica para visualizar graficamente resultados gerados por ferramentas de síntese utilizadas durante o fluxo de projeto.

O Blade suportará primeiramente somente diagramas a nível lógico, porém foi desenvolvido com o intuito de ser estendido a outras formas de diagramas permitindo o uso de especificações em outros níveis de abstração. Esta flexibilidade é um aspecto muito importante da ferramenta porque facilita tanto a sua extensão como o reuso de suas classes. Assim, o editor pode ser estendido a fim de permitir a especificação de

sistemas usando novas classes de diagramas que venham a ser utilizadas na área de CAD e no projeto de CIs. Além disso, o reuso das classes do editor pode permitir o desenvolvimento de outros editores que utilizam outras representações gráficas, ou seja, outras formas de diagramas.

5.2 Metodologia

É importante destacar que a metodologia utilizada para o desenvolvimento deste trabalho baseia-se no paradigma orientado a objetos, aliado à criação de protótipos e seguindo o modelo espiral proposto por [PRE 96].

O paradigma de projeto orientado a objetos é muito utilizado atualmente, principalmente porque suporta o projeto e manutenção de sistemas complexos. Vários autores, entre eles Rumbaugh [RUM 94], Coad [COA 93], Pressmann [PRE 96], e MARTIN [MAR 95] destacam as vantagens do uso deste paradigma no desenvolvimento de sistemas complexos.

Além disso, a metodologia orientada a objetos (OO, do inglês, *object-oriented*) foi escolhida porque possui muitas vantagens, tais como reusabilidade, flexibilidade e facilidade de manutenção. O uso dos princípios de OO garante uma maior flexibilidade à aplicação, facilitando assim a manutenção do sistema. Além disso, o uso desta metodologia permite a modelagem de tipos complexos de dados, o que é uma característica interessante no contexto do desenvolvimento de ferramentas de CAD.

Em Coad [COA 93], são listadas várias razões para o uso de prototipação e também algumas recomendações para o uso de prototipação em projetos baseados em objetos. A prototipação foi escolhida no contexto deste projeto por diversos fatores. O principal deles é seu dinamismo. Nesta metodologia, implementa-se um protótipo inicial do sistema e ao longo do desenvolvimento esta implementação vai sendo refinada até alcançar os objetivos desejados.

Outro fator que foi considerado na escolha da metodologia é a possibilidade de testar a interface com o usuário ao longo do desenvolvimento, que é altamente desejável quando do desenvolvimento de uma ferramenta gráfica. A prototipação permite a realização de experiências com o componente de interação humana do projeto, permite testar o sistema, bem como descobrir requisitos esquecidos. Além disso, esta abordagem permite liberar partes funcionais tão cedo quanto for possível e, assim, facilitar o andamento do processo de desenvolvimento do sistema.

5.3 Requisitos Funcionais do Sistema

Um editor de esquemáticos trabalha com primitivas básicas, as quais representam graficamente as células (portas lógicas e blocos funcionais), os pinos de esquema e as conexões entre as células. Como já foi analisado no capítulo 2, o editor

deve permitir a inserção, remoção, edição destas primitivas básicas, e ainda a conexão entre células para permitir que o projetista especifique um circuito através de um esquema lógico, além de algumas operações gráficas comuns nesta classe de ferramentas.

A figura 5.1 mostra um diagrama UML de caso de uso (*Use Case*), representando as funções básicas do editor do ponto de vista do usuário. Um projetista pode inserir portas lógicas e blocos, remover portas e blocos, editar portas, blocos, pinos e conexões, a fim de construir um diagrama. Generalizando, todas essas primitivas gráficas foram chamadas de componentes.

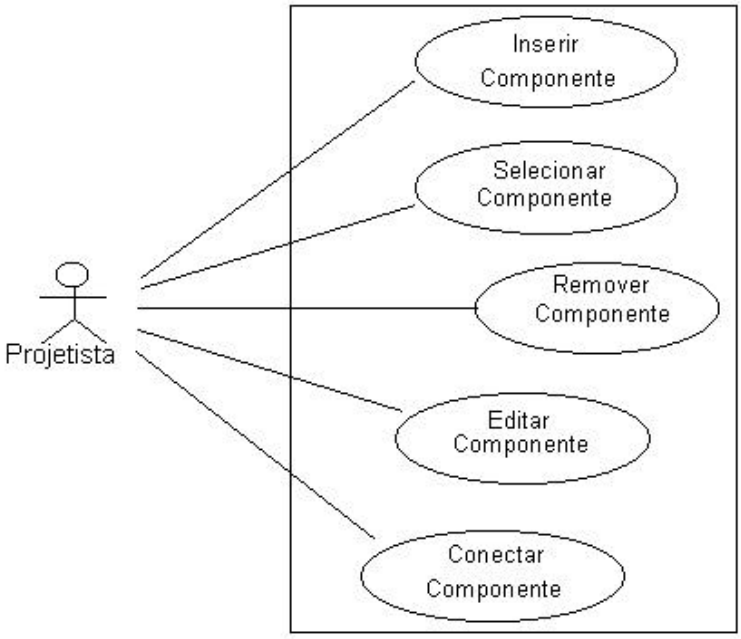


FIGURA 5.1 - Diagrama de Caso de Uso – Funções básicas do editor

A fim de permitir a execução destas funções descritas no diagrama de caso de uso da figura 5.1, dividiu-se as operações do editor em quatro modos: o modo criação (*CreateMode*), modo seleção/edição (*SelectMode*), modo seleção de área (*GroupArea*) e o modo conexão (*ConnectMode*).

No modo criação, o usuário pode inserir blocos ou portas lógicas no esquema, ou seja, inserir componentes no diagrama. Após a inserção, o usuário pode usar o modo seleção/edição para selecionar componentes do esquema, permitindo editá-los ou removê-los do diagrama. Para conectar componentes, o usuário utiliza o modo conexão. O modo seleção de área permite selecionar um grupo de componentes, através do *drag and drop* do mouse. Estes modos de operação são ativados através dos botões na barra de ferramentas do editor e somente um deles pode estar ativo a cada momento.

Além dos quatro modos básicos, há o modo hierárquico (*HierarchicalMode*) que é responsável pelas operações referentes à criação de projetos hierárquicos. Esse modo, diferentemente dos demais, não é ativado através da barra de ferramentas e não opera isoladamente e, portanto, pode estar ativo juntamente com um dos outros modos.

Através do modo hierárquico, a ferramenta deve permitir a criação de blocos funcionais que representam módulos do sistema, a definição de sua interface (pinos de entrada e saída), definição do comportamento ou da funcionalidade associada a este módulo, além de permitir ao projetista criar facilmente níveis de hierarquia, bem como excluí-los. Além disso, este modo deve permitir também uma fácil navegação entre os diferentes níveis de hierarquia do projeto.

Além dessas funções, a ferramenta deve suportar a reutilização de blocos de circuitos pré-definidos. Para suportar isto, a ferramenta deve permitir ao projetista salvar seus blocos de projeto em uma biblioteca, bem como permitir o reuso destes blocos através da criação de instâncias dos mesmos. Algumas ferramentas criam automaticamente um bloco ou um símbolo para representar uma instância de um circuito, sem que o projetista precise utilizar um editor de símbolos para fazer isso.

Para facilitar o reuso de blocos de circuitos pré-definidos, o sistema deve utilizar uma única referência para o bloco, desta forma qualquer alteração que ocorrer nesse circuito será refletida nas demais instâncias do mesmo circuito utilizadas no projeto e, assim, haverá uma única instância para ser mantida pelo projetista.

Um diagrama de caso de uso para o modo hierárquico é apresentado na figura 5.2, onde estão representadas as principais operações relacionadas à hierarquia.

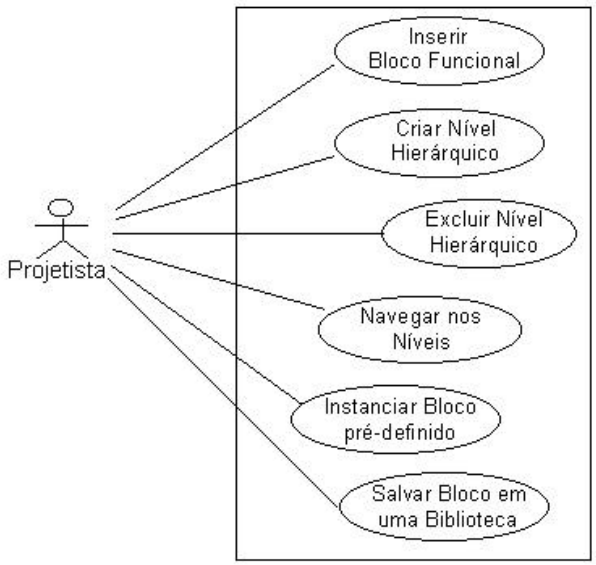


FIGURA 5.2 - Diagrama de Caso de Uso – Projetos hierárquicos

O editor deve permitir também que projetistas que trabalham em máquinas diferentes possam compartilhar um diagrama e colaborar para o desenvolvimento de um projeto. Para isso, definiu-se um outro modo de operação, o modo colaborativo que pode estar ativado e trabalhar em conjunto com os demais modos de operação da ferramenta.

O Blade utiliza o serviço de colaboração proposto por Sawicki [SAW 2002] e incorporado no ambiente Cave. A metodologia de colaboração utilizada neste serviço baseia-se na metodologia chamada *Pair Programming* [WIL 2000], na qual dois projetistas trabalham lado a lado, colaborando para o desenvolvimento de um sistema, sendo que apenas um deles tem a posse do teclado.

Em [IND 2002], foi proposta uma adaptação à esta metodologia que foi chamada de *Paar Programming*, na qual permite-se a colaboração entre máquinas remotas e a colaboração entre grupos maiores de projetistas. Assim como no *Pair Programming*, nesta nova metodologia, somente um participante pode alterar os dados de projeto a cada momento.

Na figura 5.3, pode-se observar um diagrama de caso de uso que apresenta as funções que devem ser disponibilizadas na ferramenta e que estão relacionadas à colaboração. Algumas destas funções estão extremamente ligadas ao serviço de colaboração que está sendo utilizado na ferramenta, sendo muitas vezes disponibilizadas pelo próprio serviço. Nestes casos, a ferramenta deve fazer somente o papel de interface entre o projetista e o serviço.

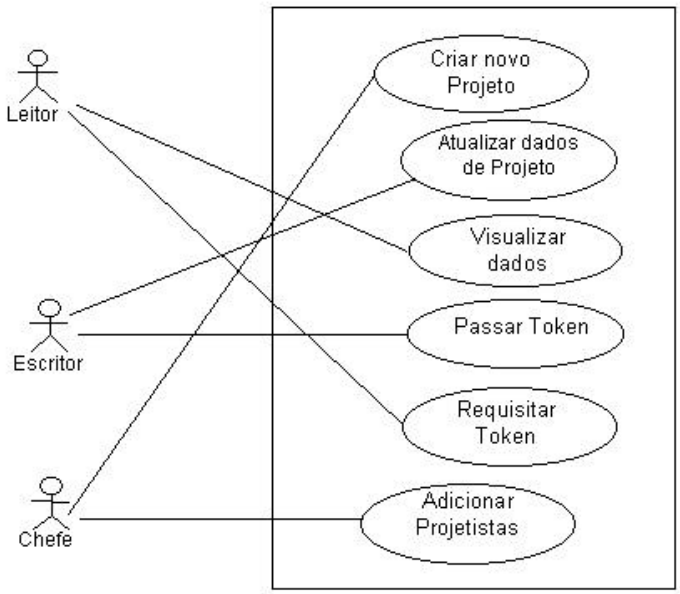


FIGURA 5.3 - Diagrama de Caso de Uso – Colaboração entre projetistas

No capítulo 6, a integração do serviço de colaboração ao Blade é descrita, são apresentadas também as funções disponibilizadas pelo serviço e um estudo de caso, no qual pode-se observar os projetistas trabalhando de forma colaborativa no Blade e, portanto, realizando as operações destacadas no diagrama da figura acima.

Na metodologia de colaboração adotada, como pode ser observado na figura 5.3, há três classes de participantes num projeto colaborativo: o leitor, o escritor e o chefe de projeto. A posse do bastão ou *token* dá o direito de escrita aos participantes, tornando-os escritores. De acordo com a metodologia de colaboração utilizada, apenas um participante pode ter a posse do *token* a cada vez. Os projetistas leitores podem visualizar os dados de projeto e requisitar ao corrente escritor o direito de escrita quando desejarem alterar os dados de projeto. O chefe do projeto é também um projetista, podendo ser escritor ou leitor, e tem funções de gerência de projeto. O chefe cria o projeto e adiciona os participantes que irão colaborar neste projeto.

Todos os modos de operação propostos neste estudo foram implementados e são detalhados ao longo do capítulo 6, onde são abordados aspectos relacionados à implementação do Blade. Após a análise da funcionalidade desejada, realizou-se um estudo dos dados manipulados pela ferramenta, que é apresentado na próxima seção.

5.4 Modelagem dos Dados

Nesta seção, aborda-se a modelagem dos dados manipulados pelo Blade, na qual considera-se a montagem de diagramas a nível lógico, além da especificação de projetos hierárquicos e a possibilidade de colaboração entre projetistas. No aspecto colaboração, a modelagem baseia-se na proposta de colaboração no Cave apresentada por Indrusiak [IND 2002] e Sawicki [SAW 2002].

Na modelagem dos dados utilizou-se um paradigma baseado em objetos, por ser atualmente um dos mais utilizados e por suportar a modelagem de sistemas complexos. Este paradigma utiliza conceitos de classes, objetos, herança, polimorfismo, encapsulamento e ligação dinâmica [PRE 96].

Além disso, uma das metas a serem alcançadas nesta ferramenta de edição de diagramas é a flexibilidade. É altamente desejável que o modelo de dados utilizado possa ser facilmente extensível, permitindo a manipulação de diferentes linguagens visuais, ou seja, diferentes formas de diagrama. Para alcançar esta flexibilidade, o modelo de dados proposto para o editor deve ser genérico, embora exista um compromisso importante a ser alcançado entre a generalidade do modelo de dados e sua adequação ao projeto de sistemas eletrônicos.

Como destacado anteriormente, o modelo que é apresentado aqui representa apenas os dados envolvidos na manipulação de diagramas lógicos, considerando o Blade como um editor de esquemáticos que trabalha a nível lógico. Conforme foi

mostrado no capítulo 2, um editor de esquemáticos manipula primitivas de projeto que representam os componentes que podem ser utilizados na montagem de um esquemático. As principais primitivas utilizadas por um editor de esquemáticos são: símbolo, bloco funcional, pino de esquema (pino de entrada e pino de saída), conexão.

As primitivas utilizadas em projetos lógicos estão representadas na figura 5.4. A figura esboça um diagrama de classes simplificado, no qual observa-se os tipos de "objetos" que precisam ser representados e, portanto, devem fazer parte do modelo de dados (estrutura de dados) de um editor de esquemáticos.

No diagrama da figura 5.4, pode-se observar uma superclasse nomeada "Primitivas_de_Projeto", a qual possui três subclasses "Objeto_com_pinos", "Conexão" e "Pino". A classe "Objeto_com_pinos" serve para modelar objetos que possuem pinos, tais como portas lógicas e blocos funcionais. A subclasse "Conexão", como o próprio nome indica, é usada para modelar as conexões entres os componentes do esquemático. A subclasse "Pino", por sua vez, representa os pinos de portas lógicas, blocos funcionais e pinos de esquema. Sendo uma conexão formada pela ligação de dois ou mais pinos pertencentes aos "objetos_com_pinos".

No modelo proposto, os "objetos_com_pinos" são divididos em três diferentes subclasses, "INPUT e OUTPUT", "Porta lógica" e "Bloco funcional". A subclasse chamada "INPUT e OUPUT" é utilizada para modelar os pinos de esquema que são utilizados para definir os pinos de entrada e saída de um circuito.

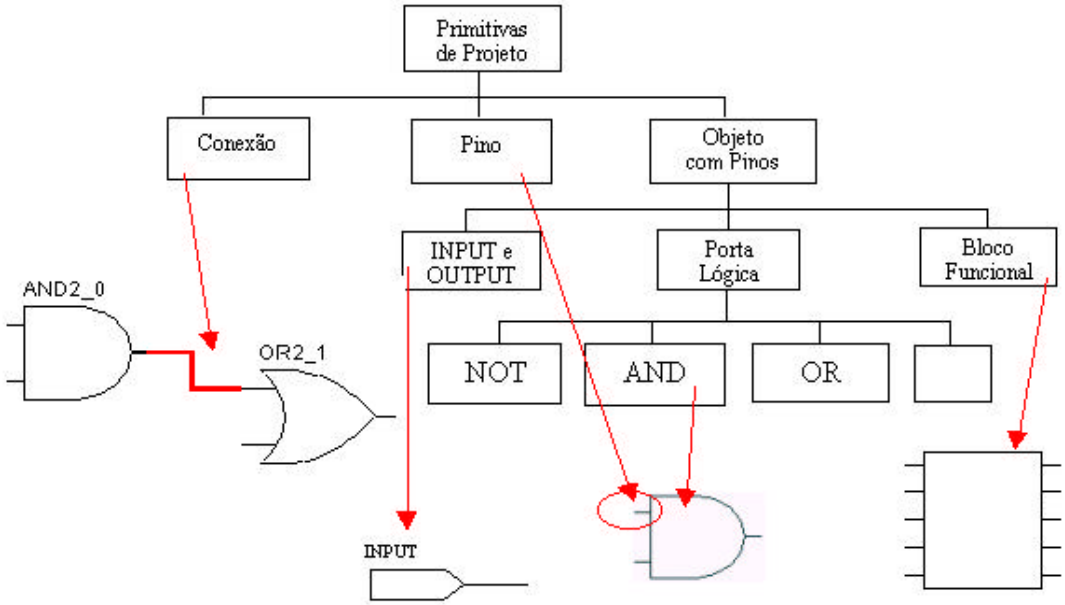


FIGURA 5.4 - Esboço das classes de um editor de esquemáticos

5.4.1 Separação dos dados

Além da representação visual, todo diagrama possui uma representação semântica. Uma importante decisão de projeto relacionada à modelagem de dados foi a separação destas duas representações em duas hierarquias de classes distintas. Esta abordagem é apresentada e defendida nesta seção.

Normalmente, em uma representação visual são utilizadas informações referentes à posição, cor e desenho dos símbolos. Já a representação semântica de um diagrama está fortemente ligada à linguagem visual que o mesmo representa e ao domínio ao qual este diagrama é aplicado. Uma representação semântica de um esquemático é responsável pela geração de um *netlist* do circuito, no qual estão listadas suas redes de interconexões.

Como se pode observar, as informações nestas duas representações são muito diferentes e devem ser manipuladas e mantidas consistentes. Por exemplo, uma porta lógica AND, pode ser representada visualmente por sua posição (x,y), sua cor, sua largura, sua altura e pelo método responsável pelo desenho da porta. Já na representação semântica de uma AND, têm-se informações muito diferentes, tais como, a identificação da função lógica da porta e o conjunto de pinos que a porta possui. Uma conexão, no domínio semântico, pode ser representada por um vetor de pinos, seguindo esta abordagem, os pinos de um componente é que possuem a informação de conexão deste componente com os demais componentes do circuito. Enquanto, no domínio visual, uma conexão pode ser representada pelo conjunto de pontos que definem os segmentos de reta que formam a conexão.

Sob esta visão, optou-se por separar as informações manipuladas pelo editor em dois domínios distintos: visual e o semântico. Todas as primitivas de projeto possuem duas ou mais classes para representá-las, sendo pelo menos uma destas classes no domínio semântico. Foram criadas duas superclasses no *Cave Framework Server:* a *CaveVisualObject* e *CaveDesignObject*. A *CaveVisualObject* é a superclasse da representação visual e a *CaveDesignObject* a da representação semântica. Estas duas superclasses são classes abstratas, cujos métodos devem ser definidos nas subclasses e podem ser usadas na modelagem das primitivas de projetos das ferramentas do Cave.

Usando esta abordagem de separação de domínios (semântico e visual), a representação da semântica e a representação gráfica de um diagrama são modeladas por diferentes objetos. Isto possibilita inúmeras visualizações de um mesmo bloco de projeto, sendo que cada uma destas visualizações pode interagir com diferentes projetistas.

Esta prática é muito utilizada em engenharia de software porque aumenta a reusabilidade e facilita a manutenção e o entendimento do sistema. Esta abordagem permite que classes coletivas relacionem-se sem perder reusabilidade. Vários *toolkits* usados para a construção de interfaces gráficas separam aspectos da representação dos

dados dos aspectos da aplicação. Assim, as classes que definem a aplicação e as classes utilizadas na representação dos dados podem ser reutilizadas independentemente [GAM 2000]. Esta abordagem também é usada em editores de diagramas para facilitar a validação dos diagramas [SER 95].

No caso do editor de diagramas, a separação dos dados de projeto e dados de visualização permite que os projetistas possuam diferentes visualizações de um mesmo bloco de projeto. No contexto de projeto colaborativo, esta possibilidade é muito interessante e permite a implementação do modo visualmente desacoplado, definido por Indrusiak [IND 2001] no modelo de colaboração proposto para o ambiente Cave.

A figura 5.5 ilustra esta abordagem de separação dos dados em dois domínios, o semântico e o visual, no contexto de um editor de esquemáticos. Nesta ilustração, pode-se observar no domínio da semântica, a presença de métodos referentes à geração do *netlist* do circuito e no domínio visual, a presença de métodos referentes ao desenho das primitivas e à visualização do circuito.

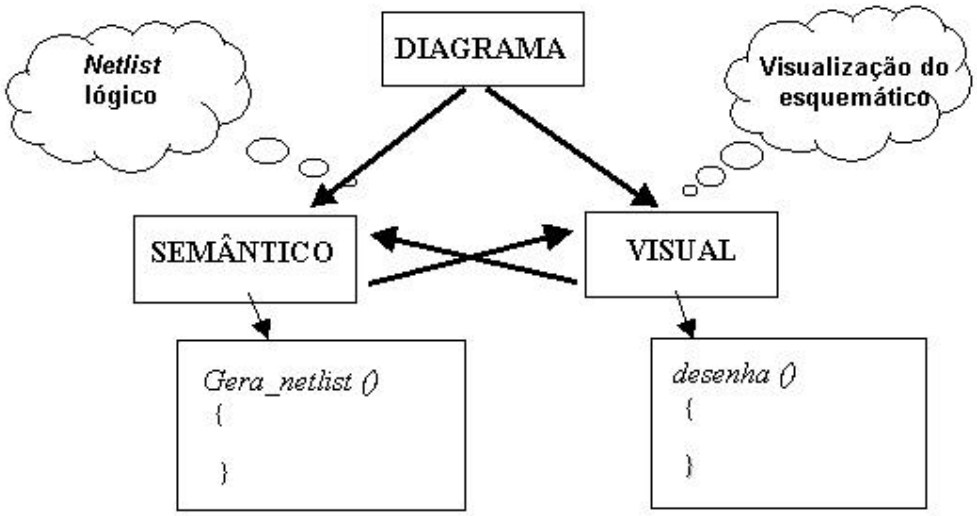


FIGURA 5.5 - Separação dos dados

No próximo capítulo, a implementação da ferramenta é abordada. Ao longo do capítulo, são apresentados diagramas de classe que seguem esta proposta de modelagem e de separação de dados discutida aqui e, portanto, no referido capítulo podem ser encontradas maiores informações do modelo de dados. Os diagramas são usados para ilustrar e apoiar a descrição da implementação das funcionalidades da ferramenta.

6 Implementação do Blade

Seguindo a especificação realizada no capítulo anterior, um protótipo do editor foi implementado. Este capítulo aborda aspectos relativos à implementação do Blade e utiliza diagramas de classes da notação UML para descrever a etapa de implementação. Esses diagramas, assim como a linguagem UML, são bastante difundidos entre desenvolvedores de software orientados a objetos.

O protótipo do Blade foi desenvolvido usando uma metodologia orientada a objetos e usando a linguagem Java. Seguindo uma metodologia de prototipação, primeiramente, desenvolveu-se um protótipo no qual somente foram implementadas as funcionalidades básicas do editor, a fim de permitir a montagem e edição de diagramas lógicos. Após, este protótipo foi alterado a fim de suportar projetos hierárquicos. E por fim, integrou-se o serviço de colaboração ao protótipo da ferramenta.

Primeiramente, apresenta-se as razões para a escolha da linguagem Java para o desenvolvimento deste projeto, após apresenta-se o modelo de dados utilizado no Blade, sua integração ao ambiente Cave e, ao longo do capítulo, são apresentados também detalhes da implementação das funcionalidades do editor. Dentre estas funcionalidades podem ser destacadas a hierarquia, a geração *netlist* e a colaboração que são abordadas nas seções deste capítulo.

6.1 Linguagem Java

A linguagem Java é uma linguagem computacional completa, foi criada pelo grupo liderado por Janmes Gosling na *Sun Microsystems* com o objetivo de reduzir a complexidade do processo de programação [HOR 2001]. Um dos principais motivos para a escolha da linguagem Java é a portabilidade dos códigos escritos em Java, bem como a facilidade de integração WWW encontrada nesta linguagem. Segundo Campione [CAM 96], Java é uma linguagem computacional completa, adequada para o desenvolvimento de aplicações baseadas na rede Internet, redes fechadas ou ainda programas *stand-alone.*

Dentre as características da linguagem Java, algumas diretamente ligadas ao fato desta linguagem ser baseada em objetos, pode-se destacar a reusabilidade e simplicidade, pois permitem que uma aplicação desenvolvida nesta linguagem tenha um tempo de desenvolvimento menor que uma aplicação desenvolvida em uma linguagem convencional.

A *Sun Microsystems* fornece um pacote chamado JDK, do inglês *Java Development Kit* [KRA 97], onde são disponibilizadas várias ferramentas como o interpretador, o compilador, o gerador de documentação, entre outras. Juntamente com

o JDK são fornecidas bibliotecas básicas, chamadas APIs, que disponibilizam códigos para integração de hiperdocumentos, construção de interfaces gráficas, estabelecimento de conexões de rede, além de outras funcionalidades, que podem ser reutilizadas pelos desenvolvedores Java. O uso destas bibliotecas e o fato da linguagem ser orientada a objetos fazem com que as aplicações em Java tenham um tempo de desenvolvimento menor do que aplicações desenvolvidas em linguagens convencionais.

Além disso, os últimos esforços do projeto Cave visam, entre outros aspectos, disponibilizar um *framework* de software, ou seja, um conjunto de pacotes de classes Java que podem ser reutilizáveis pelos projetistas de ferramentas, facilitando a implementação e integração de novas ferramentas. Portanto, a possibilidade de reuso de classes do Cave no desenvolvimento do Blade também motivou o uso da linguagem Java. Assim como o desenvolvimento do Blade também contribuiu para a definição deste *framework.*

6.2 Modelo de Dados

Na seção 5.3 deste trabalho, a modelagem de dados do Blade é discutida e são apresentados os requisitos que devem ser atendidos pelo modelo de dados. Na figura 5.4, pode-se observar as principais primitivas utilizadas para a construção de diagramas lógicos, tais como portas lógicas, pinos de esquema, conexões, etc. Estas primitivas foram modeladas a fim de permitir a construção e a edição de diagramas lógicos no Blade. Apesar do modelo apresentado só representar primitivas relacionadas à edição e montagem de diagramas a nível lógico, o modelo deve ser facilmente extensível a outras formas de diagrama.

O modelo de dados do Blade foi dividido em dois domínios: Visualização e Projeto. Esta abordagem permite que um projetista possa ter diferentes visões do mesmo bloco de projeto, ou ainda, que os projetistas possam ter visões diferentes de um diagrama, sendo todas essas visões correspondentes a um mesmo modelo. A separação de dados de projeto e visualização é discutida na seção 5.3.1 do capítulo 5 e ilustrada na figura 5.5.

No repositório de classes do Cave, foram definidas duas *packages* (pacotes), ***cave.graphic*** e ***cave.design***. Em *cave.graphic* encontram-se as classes referentes a aspectos gráficos das ferramentas, enquanto que no pacote *cave.design* encontram-se as classes que modelam aspectos referentes ao projeto propriamente dito.

Assim como na criação do modelo de dados do editor de diagramas, foram definidas duas hierarquias de classes uma referente ao **domínio visual** e outra referente ao **domínio semântico**. A hierarquia referente ao domínio visual faz parte do pacote *cave.graphic* e a hierarquia referente ao domínio semântico faz parte do pacote *cave.design*.

A hierarquia do domínio visual modela aspectos referentes à visualização dos componentes do diagrama e será apresentada na seção 6.2.1. A hierarquia do domínio semântico representa as informações referentes à semântica do diagrama e é apresentada na seção 6.2.2. No caso de um diagrama lógico, informações semânticas seriam aquelas necessárias à geração do *netlist* lógico do circuito.

6.2.1 Domínio Visual

Na figura 6.1, pode-se observar um diagrama de classes, onde a classe *Rectangle* que faz parte do pacote *Java.awt* é a classe mãe e todas as outras classes representadas no diagrama. Isto permite que todos os objetos visuais que compõem o diagrama estejam envolvidos por um retângulo que serve como um envelope nas operações de seleção, redimensionamento e movimentação dos componentes. Na figura 6.2 pode-se observar o envelope de uma porta lógica NAND.

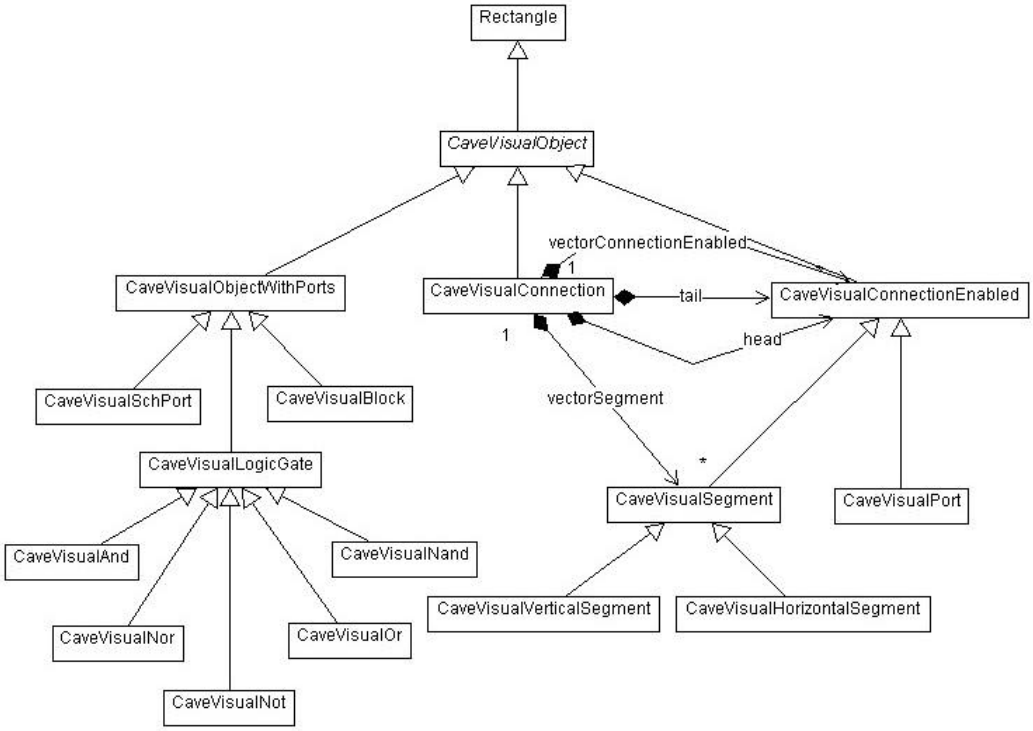


FIGURA 6.1 - Classes domínio visual

A classe *CaveVisualObject* é abstrata e foi criada para garantir a consistência entre todos os objetos visuais que compõem um diagrama. As classes *CaveVisualObjectWithPorts*, *CaveVisualConnection* e *CaveVisualConnectionEnabled* são suas subclasses. Na classe *CaveVisualObject* são definidos alguns métodos para suportar a seleção, o redimensionamento e a movimentação dos objetos visuais. Além disso, são definidos alguns métodos abstratos e que, portanto, devem ser implementados nas subclasses, entre estes se encontra o método *draw* que é responsável pelo desenho

dos componentes. Como o desenho de uma porta lógica AND é diferente do desenho de uma porta lógica OR, as implementações deste método dependem do tipo de componente e devem ser especificadas nas subclasses *CaveVisualAnd* e *CaveVisualOr*, respectivamente.

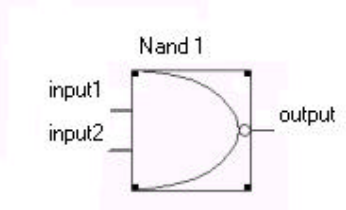


FIGURA 6.2 - Envelope envolvendo a porta lógica NAND

A classe *CaveVisualObjectWithPorts* representa todos os componentes que possuem pinos e podem ser conectados a outros componentes. Entre estes encontram-se as portas lógicas, blocos e pinos de esquema. Portanto, como subclasses de *CaveVisualObjectWithPorts* tem-se *CaveVisualLogicGate*, *CaveVisualBlock* e *CaveVisualSchPort*.

A classe *CaveVisualLogicGate* possui subclasses *CaveVisualAnd*, *CaveVisualOr*, *CaveVisualNot*, *CaveVisualNand* e *CaveVisualNor* que modelam, respectivamente, as portas lógicas tipo AND, OR, Inversor, NAND e NOR.

Objetos da classe *CaveVisualBlock* representam as primitivas conhecidas como blocos funcionais, que são utilizados para representar módulos do sistema. Objetos desta classe são usados para a construção de projetos hierárquicos.

A classe *CaveVisualSchPort* é usada para modelar os pinos de entrada e saída de um esquema lógico. Um dos atributos desta classe é o tipo de pino, para definir se o objeto é do tipo pino de entrada (INPUT) ou do tipo pino de saída (OUPUT).

Uma conexão no domínio visual pode ser representada por um segmento de reta ou por um conjunto de segmentos de reta, que devem ser desenhados para interligar dois componentes. A classe *CaveVisualConnection* representa uma conexão no domínio visual e foi criada a fim de permitir o desenho e a edição das conexões. Os segmentos são modelados pela classe *CaveVisualSegment*. Para permitir o desenho de conexões ortogonais, foram criadas as subclasses *CaveVisualHorizontalSegment* para modelar os segmentos horizontais e *CaveVisualVerticalSegment* para modelar os segmentos verticais. A fim de permitir a criação de conexões ortogonais, definiu-se que uma instância de *CaveVisualConnection* possui um vetor de segmentos (*CaveVisualSegment*), geralmente dois horizontais e um vertical, ou vice-versa.

Na figura 6.3, pode-se observar uma conexão, uma instância de *CaveVisualConnection*, que é formada pelos segmentos *Seg1*, *Seg2* e *Seg3*. Onde *Seg1* e *Seg3* são instâncias de *CaveVisualHorizontalSegment* e *Seg2* é uma instância de *CaveVisualVerticalSegment*.

A fim de permitir a criação de uma conexão entre dois pinos de componentes ou entre um pino e uma conexão já existente, foi criada a classe *CaveVisualConnectionEnabled*. Esta classe modela os componentes que podem ser início e fim de uma conexão, ou seja, os pinos e os segmentos. São subclasses de *CaveVisualConnectionEnabled*, as classes *CaveVisualPort* e *CaveVisualSegment*.

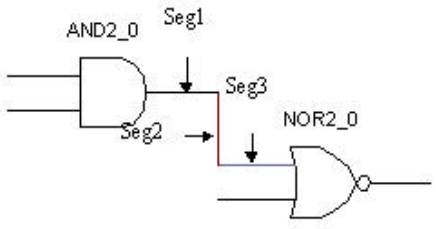


FIGURA 6.3 - Conexão no domínio visual

Esta estrutura foi montada para permitir o desenho de conexões ortogonais, assim como possibilitar conexões entre mais de dois elementos, além de permitir que quando um componente estiver conectado a um outro e for movimentado, o desenho da conexão que há entre eles possa ser refeito a fim de se adaptar ao novo posicionamento dos componentes.

O modelo apresentado nessa seção faz parte do domínio visual, a modelagem das conexões no domínio semântico será discutida na seção 6.2.2, sendo que as duas estruturas são importantes para a implementação das funcionalidades de conexão do editor. A implementação do modo conexão será discutida na seção 6.5.4, onde mais detalhes podem ser encontrados.

6.2.2 Domínio Semântico

A hierarquia de classes do domínio semântico pode ser visualizada na figura 6.4. No topo desta hierarquia encontra-se a classe *CaveDesignObject*. Esta superclasse é abstrata e implementa a interface *Serializable* que faz parte do pacote *Java.io*.

A interface *Serializable* permite que qualquer objeto Java seja transformado em uma seqüência de bytes que pode mais tarde, ser completamente restaurada para gerar o objeto original. Este processo é conhecido como serialização e permite também compensar automaticamente as diferenças de sistemas operacionais. Assim, pode-se serializar um objeto numa máquina *Windows* e enviá-lo para uma máquina *Unix* onde este poderá ser corretamente reconstruído.

A classe *CaveDesignObject* tem como subclasses: *CaveDesignConnection*, *CaveDesignPort* e *CaveDesignObjectWithPorts.* A classe *CaveDesignConnection* representa a estrutura para as informações referentes à conexão do ponto de vista semântico, em outras palavras, quem está ligado com quem. A classe *CaveDesignPort* é usada para representar as informações dos pinos de componentes, no ponto de vista semântico, estas informações se referem ao tipo de porta e a que componente esta porta pertence. A classe *CaveDesignObjectWithPorts* modela os objetos com pinos (portas lógicas, blocos e pinos de esquema) no lado semântico e possui como subclasses *CaveDesignLogicGate*, *CaveDesignBlock* e *CaveDesignSchPort*. Um objeto da classe *CaveDesignObjectWithPorts* possui um vetor de objetos do tipo *CaveDesignPort*, este vetor mantém os pinos referentes a este objeto.

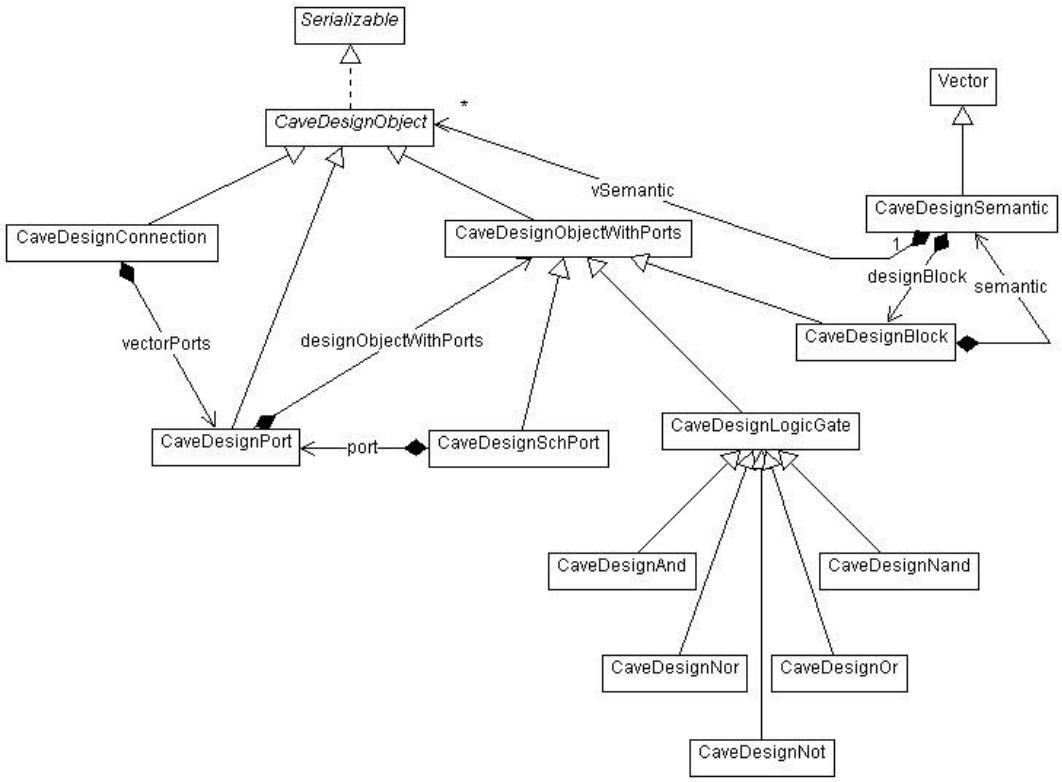


FIGURA 6.4 - Diagrama de classes do pacote *Design*

Os objetos do tipo *CaveDesignObjectWithPorts* possuem objetos do tipo *CaveDesignPort*. Na classe *CaveDesignObjectWithPorts*, para modelar o relacionamento entre um componente e seus pinos foram criados quatro vetores: *eastp*, *westp*, *sothp* e *northp*. Cada vetor armazena os pinos de um dos lados do componente, este, podendo possuir pinos em qualquer dos seus lados. Cada vetor é uma associação **um para muitos**, entre a classe *CaveDesignObjectWithPorts* e a classe *CaveDesignPort.* Estes vetores não estão ilustrados na figura 6.4, mas ela mostra uma

outra associação entre as duas classes chamada *designObjectWithPorts* e que serve para que o pino conheça o componente ao qual ele pertence.

As conexões no domínio semântico são representadas por um vetor de pinos. Este vetor é representado por uma associação de cardinalidade **um para muitos** entre a classe *CaveDesignConnection* e *CaveDesignPort* que pode ser observada na figura 6.4.

A classe *CaveDesignBlock* possui uma referência para um objeto do tipo *CaveDesignSemantic*, que possui um vetor de objetos do tipo *CaveDesignObject.* A classe *CaveDesignBlock*, juntamente com a classe *CaveDesignSemantic*, serve para a modelagem de hierarquia. Com a criação destas classes e das associações entre elas, permite-se que um componente do tipo bloco (*CaveDesignBlock*) seja composto por vários outros componentes, inclusive por outros blocos, permitindo, assim, a criação de projetos hierárquicos. Maiores detalhes sobre a implementação da hierarquia no protótipo serão apresentados na seção 6.5.

6.3 Integração de uma ferramenta no ambiente Cave

Conforme foi apresentado no capítulo 3, a interface ativa do ambiente Cave, chamada de *Tool Launcher*, permite ao usuário instanciar as ferramentas disponíveis no servidor através de conexões *sockets*. Após a conexão ser aberta, o *Tool Launcher* liga dinamicamente as classes que compõem a ferramenta solicitada e as instancia, ou seja, cria uma instância da ferramenta na máquina cliente, possibilitando o uso da ferramenta pelo usuário. Com a interação com o usuário, primitivas de projeto são também instanciadas.

Para que uma ferramenta possa ser invocada a partir do *Tool Launcher* do Cave, a ferramenta deve ser inserida na lista de serviços disponíveis em um dos servidores utilizados no ambiente. Além disso, a ferramenta deve ser criada como uma subclasse de *CaveGUI* e também deve ser criada uma subclasse da classe *CaveHandler* que será utilizada na invocação da ferramenta. A existência destas duas classes e suas corretas configurações são requisitos para a invocação da ferramenta através do *Tool Launcher*.

A classe *CaveGUI* estende a classe *JFrame* que faz parte do pacote Java Swing. Esta classe trata da comunicação com o servidor do *framework* e estabelece as conexões *sockets* que permitem a invocação da ferramenta. A classe *CaveGUI* faz parte do pacote *cave.graphics*. Na figura 6.5, pode-se observar um diagrama de classes de *CaveGUI*.

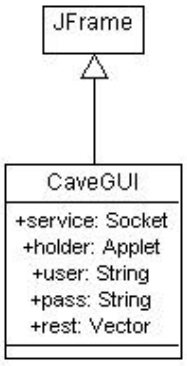


FIGURA 6.5 - Classe *CaveGUI*

A classe *CaveHandler* define um *listener* que fica no servidor e é instanciado para ouvir a comunicação do servidor com a ferramenta. A figura 6.6 esboça o diagrama de classes de *CaveHandler*, que faz parte do pacote *cave.protocol*. Pode-se observar no diagrama que a classe *CaveHandler* estende a classe *Thread*, que é definida no pacote *Java.Net*.

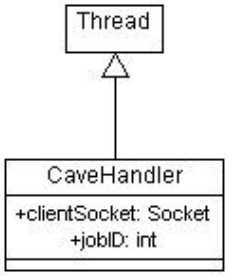


FIGURA 6.6 - Classe *CaveHandler*

Nas bibliotecas disponíveis no Cave, há uma classe chamada *CaveGraphicEditor* que é uma subclasse de *CaveGUI*. Esta classe foi criada com o intuito de facilitar a implementação de ferramentas gráficas, tais como editores de leiaute, de esquemáticos e de máquinas de estados. Algumas funcionalidades básicas tais como, seleção, movimentação e redimensionamento de componentes na área de trabalho do editor, são definidas nesta classe, podendo ser reutilizadas para o desenvolvimento de outras ferramentas. Na figura 6.7, um diagrama de classes que esboça a definição de *CaveGraphicEditor* é apresentado.

A classe *CaveGraphicEditor* define também uma barra de menu padrão para as ferramentas, bem como alguns outros elementos para a montagem da interface, tais como botões de zoom, barra de ferramentas, etc. Além disso, esta classe define três modos de operação *CreateMode, SelectMode* e *GroupMode*. Na figura 6.8, pode-se observar a definição destes modos.

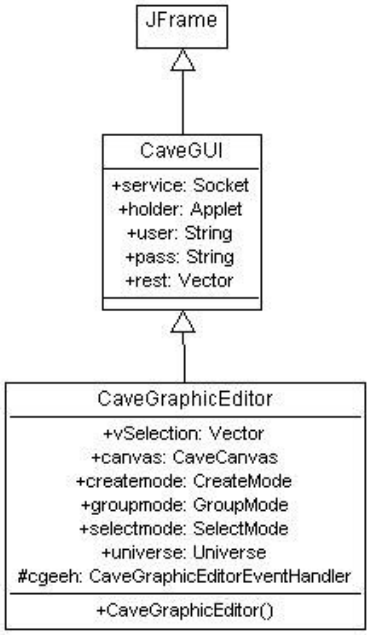


FIGURA 6.7 - Classe *CaveGraphicEditor*

O *CreateMode* permite ao usuário instanciar primitivas, possibilitando através do clique do mouse em um dos botões da barra de ferramentas e do clique na área de trabalho, a inserção de um componente. O *SelectMode* permite selecionar componentes presentes na área de trabalho. Após a seleção, os componentes podem ser redimensionados ou movimentados. O *GroupMode* permite a seleção de um grupo de componentes.

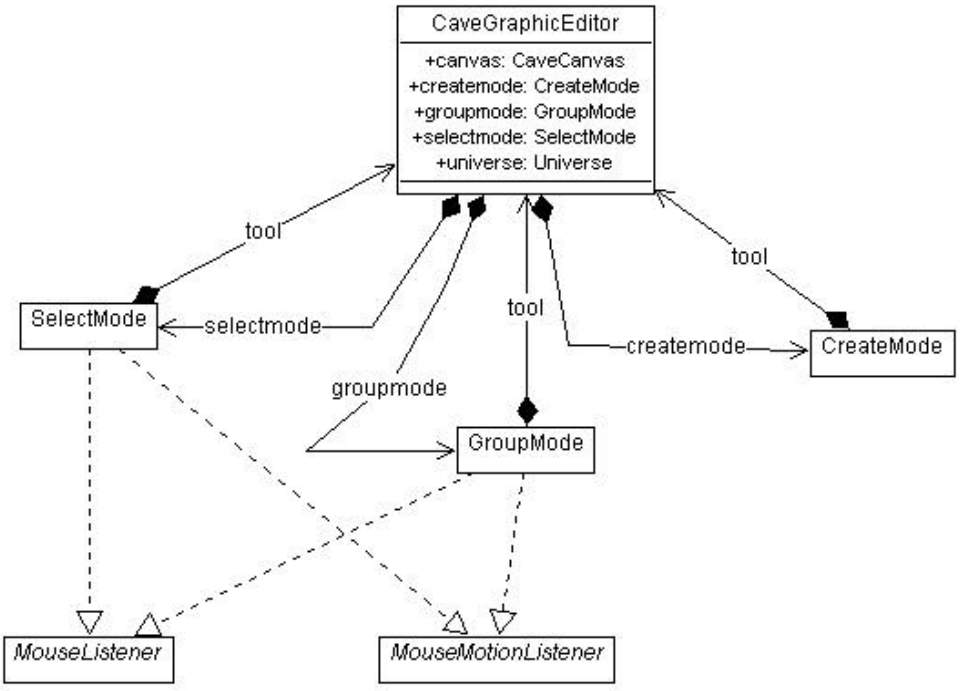


FIGURA 6.8 - Modo de Operação definidos em *CaveGraphicEditor*

Na definição da classe *CaveGraphicEditor*, criou-se duas classes importantes *CaveCanvas* e *Universe.* A classe *CaveCanvas* representa a área de trabalho da ferramenta. A classe *Universe* foi criada para representar todos os objetos visuais que são criados durante a interação entre usuário e ferramenta.

A classe *CaveCanvas* é uma subclasse de *Jpanel* que faz parte do pacote *Java Swing*. Esta classe possui uma referência para a classe *Universe*, que visa a representação dos elementos que são inseridos na área de trabalho, ou seja, a representação de todas as instâncias das primitivas do editor, criadas ao longo da interação entre usuário e ferramenta. Um esboço da definição da classe *CaveCanvas* pode ser observado na figura 6.9.

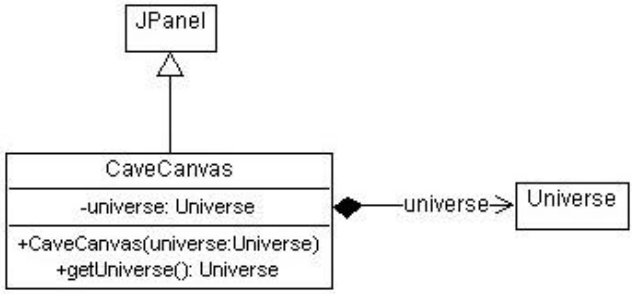


FIGURA 6.9 - Classe *CaveCanvas*

As classes *CaveGraphicEditor, CaveCanvas* e a classe *Universe* fazem parte do pacote *cave.graphics*. Na figura 6.10 pode-se observar as relações entres estas classes.

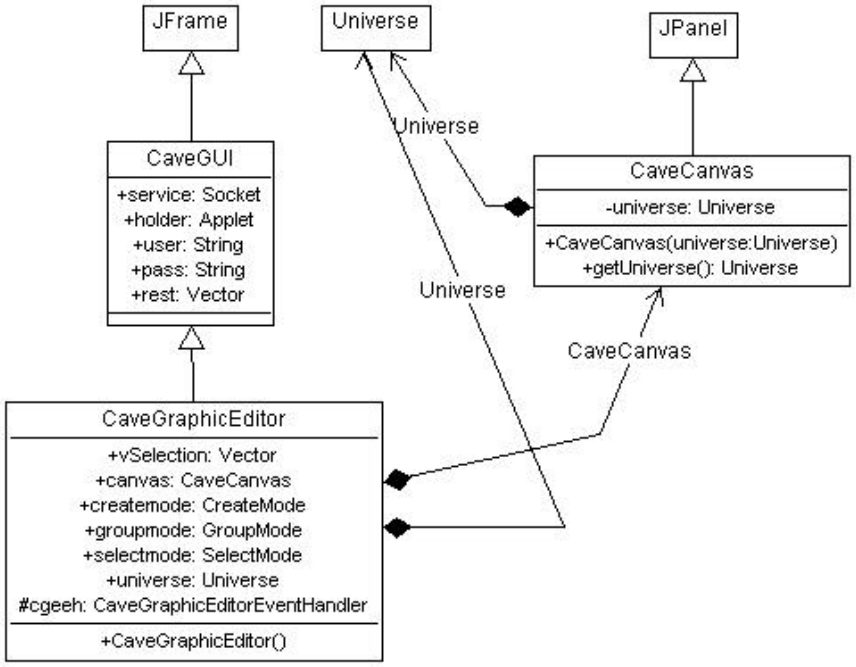


FIGURA 6.10 - Relação entre as classe *CaveGraphicEditor, CaveCanvas* e *Universe*

As classes *CaveGUI*, *CaveHandler*, *CaveGraphicEditor*, *CaveCanvas* e *Universe* fazem parte do *framework* de software do Cave e foram criadas visando facilitar o desenvolvimento e a inserção de novas ferramentas neste ambiente.

6.4 Integração do Blade

Durante o desenvolvimento do Blade, decidiu-se reutilizar funções de edição gráfica definidas em *CaveGraphiEditor*, portanto, a classe principal do Blade foi definida como subclasse de *CaveGraphicEditor*. Desta forma, o Blade herda algumas funcionalidades, tais como: seleção, movimentação e redimensionamento de componentes já definidas em *CaveGraphicEditor*. Além disso, para que o Blade possa ser invocado pelo *Tool Launcher*, a ferramenta deve ser uma subclasse de *CaveGUI*. Esta relação de herança entre *CaveGaphicEditor* e Blade garante também este requisito necessário à invocação da ferramenta pelo *Tool Launcher*. Na figura 6.11, pode-se observar a interface do *Tool Launcher*, a partir da qual o Blade pode ser invocado.

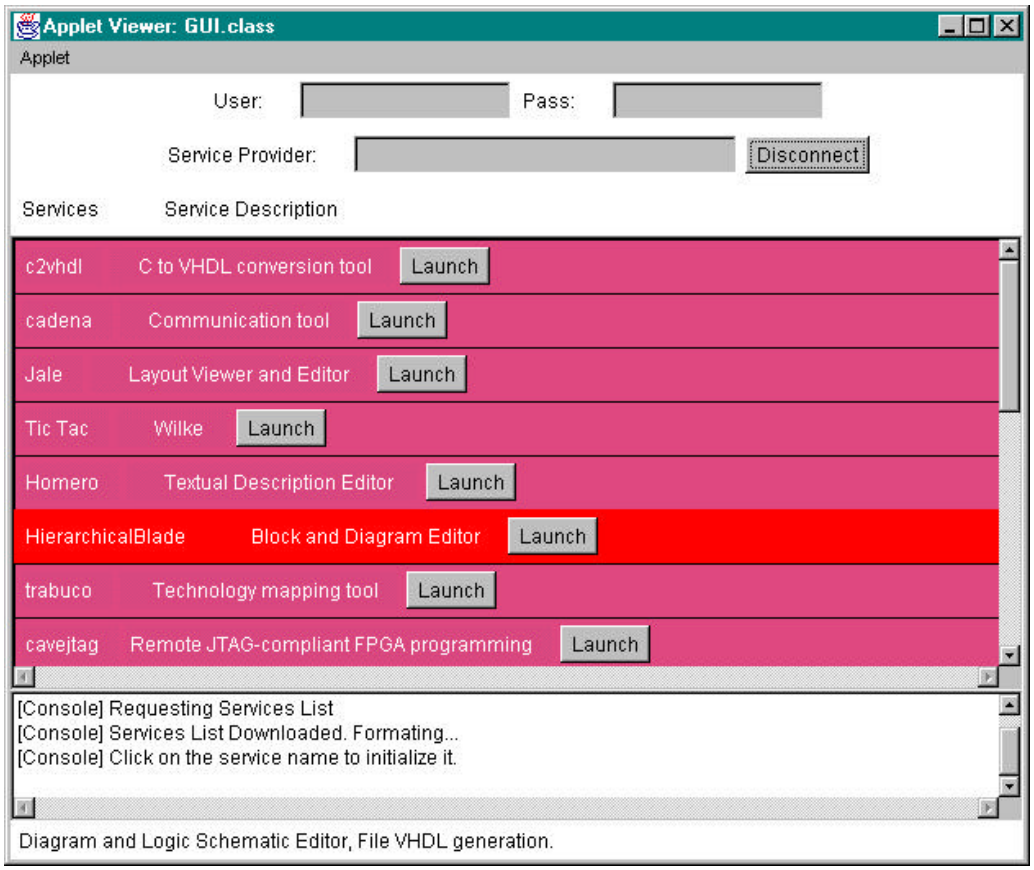


FIGURA 6.11 - Interface do *Tool Launcher*

Portanto, a classe Blade herda algumas definições de *CaveGraphicEditor*, como a definição dos modos de operação, a referência para o *Universe* e para *CaveCanvas*. Além disso, a classe *CaveGraphicEditor* define uma barra de menu padrão para as

ferramentas, bem como alguns outros elementos para montagem da interface, tais como botões de zoom e barra de ferramentas, que foram também reutilizados na implementação do Blade. Um esboço da classe *Blade* pode ser observado na figura 6.12.

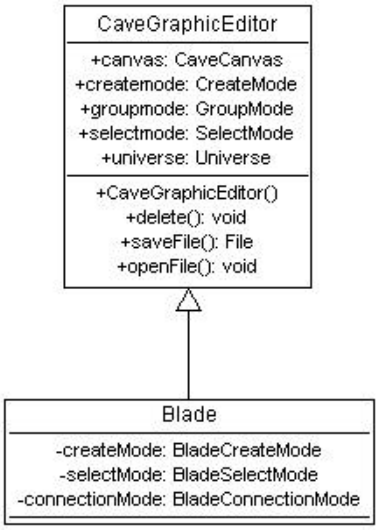


FIGURA 6.12 - Classe *Blade*

Conforme foi especificado no capítulo anterior, as funcionalidades da ferramenta foram divididas em modos de operação. Em *CaveGraphicEditor*, esta abordagem já era utilizada, onde foram definidos os modos criação, seleção e seleção de grupo. Porém, para o Blade, alguns destes modos tiveram de ter sua implementação alterada, assim como um novo modo foi definido para tratar das conexões. Portanto, foram criadas novas classes, *BladeCreateMode, BladeSelectMode* e *BladeConnectionMode*. Na figura 6.13, pode-se observar os modos básicos de funcionamento do Blade, cujas implementações serão apresentadas na seção 6.5.

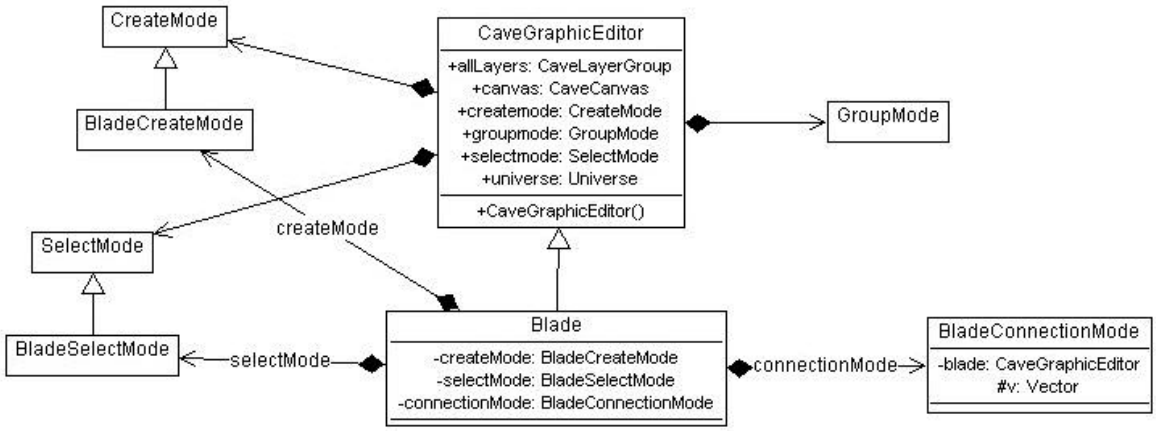


FIGURA 6.13 - Modos básicos de operação

Todos os modos de operação são classes que implementam as interfaces *MouseListener* e *MouseMotionListener,* permitindo o tratamento dos eventos do mouse.

6.5 Implementação das Funcionalidades Básicas do Blade

As implementações dos modos Criação, Seleção, Seleção de Grupo e Conexão são apresentadas nas seções 6.41, 6.4.2, 6.4.3 e 6.4.4, respectivamente. A Ativação destes modos de operação ocorre através da seleção de um botão na barra de ferramentas do editor e estes modos nunca estão ativados simultaneamente. A figura 6.14 apresenta a barra de ferramentas do Blade.

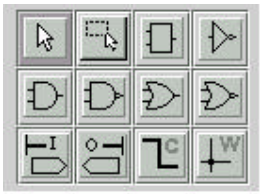


FIGURA 6.14 - Barra de ferramentas do Blade

6.5.1 Modo Criação

No *framework* do Cave foi definida a classe *CreateMode* que foi utilizada na implementação do editor de leiaute chamado *Jale* [OST 2001] [GRA 99]. Nesta classe, de acordo com o botão selecionado na barra de ferramentas, pode-se inserir componentes na área de trabalho do editor. Na implementação do Blade, a classe CreateMode foi estendida, sendo definida uma nova classe, chamada *BladeCreateMode*, que herda atributos e métodos de *CreateMode*.

A barra de ferramentas do Blade pode ser visualizada através da figura 6.14. Os componentes que podem ser instanciados, nesta versão, são as portas lógicas NOT, AND, OR, NAND e NOR, blocos funcionais e pinos de esquema (INPUT e OUTPUT). Portanto, para ativar o modo criação um destes botões deve ser selecionado:

Por exemplo, para inserir uma porta AND na área de trabalho, o usuário seleciona o botão  da barra de ferramentas e clica na área de trabalho. Com isso, criam-se uma instância da classe *CaveVisualAnd* e uma instância da classe *CaveDesignAnd*. Criam-se também instâncias para os pinos da porta lógica. Padronizou-se a definição de dois pinos de entrada e um de saída para a criação de portas lógicas e de blocos funcionais, portanto, neste caso, devem ser criadas três instâncias de *CaveVisualPort* e três instâncias de *CaveDesignPort*.

Porém, o número de entradas é um dos parâmetros que podem ser alterados pelo usuário. Após a inserção de um componente, o usuário pode modificar a sua interface. Para fazer isso, o usuário deve selecionar o componente, clicar com o botão direito e ao visualizar um *popup* menu, deve selecionar a opção ***Properties*** deste menu. No caso das

portas lógicas, só podem ser inseridos pinos de entrada, nestes casos o usuário deve informar apenas o nome do pino. No caso dos blocos, o usuário pode definir pinos de entrada e de saída, além de poder definir qual o lado em que o pino será inserido.

A figura 6.15 apresenta a janela de configuração de pinos do Blade, onde se pode visualizar um bloco no qual foram inseridos novos pinos, ficando este bloco com dois pinos no lado direito, dois pinos no lado esquerdo, um pino no lado superior e um no lado inferior. Pode-se observar na janela de configuração que para criar um pino de um bloco, deve-se definir o nome do pino, seu tipo (INPUT, OUTPUT, INOUT) e o lado onde ele será inserido (WEST, EAST, NORTH e SOUTH).

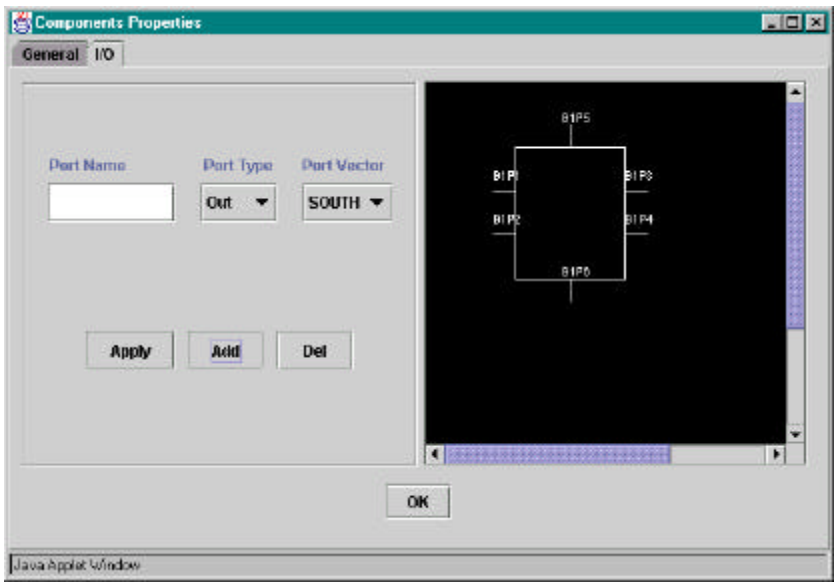


FIGURA 6.15 - Janela de Configuração de Pinos

6.5.2 Modo Seleção

O modo seleção permite selecionar os componentes na área de trabalho. Este modo é ativado a partir do botão  na barra de ferramentas.

Da mesma forma, na implementação da seleção dos componentes, foi reutilizada a implementação de seleção já presente no Cave na classe *SelectMode*, também usada no *Jale*. A classe *SelectMode* implementa as interfaces *MouseListener* e *MouseListenerMotion* para permitir selecionar ou desmarcar um componente na área de trabalho, através do clique do mouse.

No contexto do desenvolvimento do Blade, criou-se uma subclasse de *SelectMode*, a qual foi chamada *BladeSelectMode*. Esta nova classe reescreve alguns métodos de *SelectMode* a fim de suportar algumas particularidades da nova ferramenta. Esta classe permitiu também a implementação de um *popup* menu que é ativado através

do botão direito do mouse sobre um componente selecionado. As opções disponíveis neste menu dependem do componente selecionado.

A seleção permite o redimensionamento, movimentação e remoção de componentes presentes na área de trabalho, bem como a alteração das propriedades dos componentes. Todos os componentes inseridos na área de trabalho são instâncias de *CaveVisualObject* e portanto, são representações gráficas das primitivas de projeto. Todas essas representações possuem um retângulo que as envolve, denominado envelope que é visualizado quando um componente é selecionado. Na figura 6.16, observa-se que a porta lógica *Nand1* está selecionada e portanto, seu envelope está visível.

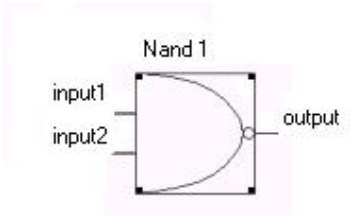


FIGURA 6.16 - Seleção de componentes

Os cantos extremos do envelope possuem pequenos quadrados que servem para redimensionar o componente, estes podem ser observados na figura 6.16. Por exemplo, quando após a seleção de um componente, clica-se no canto superior esquerdo do envelope que o envolve e arrasta-se o mouse para cima, o componente terá seu tamanho aumentado. Se nenhum dos cantos do envelope for selecionado e arrasta-se o mouse sobre a área de trabalho, o componente é simplesmente reposicionado, acompanhando o movimento do mouse.

6.5.3 Modo Seleção de Grupo

A classe *GroupMode*, presente no *framework* Cave, permite a seleção de um grupo de componentes através do *drag and drop* do mouse. Após a seleção, pode-se reposicionar ou deletar os componentes do grupo. Este modo é ativado pelo botão da barra de ferramentas.

6.5.4 Modo Conexão

O modo conexão tem por objetivo permitir conectar componentes. Este modo é ativado a partir do botão (“*connect*”) na barra de ferramentas. Para entender melhor a implementação do modo conexão, deve-se observar a modelagem das conexões que é brevemente abordada na seção 6.2 e que será revista nesta seção.

Como pode-se observar no capítulo 3, uma ferramenta de edição de esquemas lógicos deve permitir a criação de conexões ortogonais. Além disso, observou-se que as

conexões devem acompanhar o movimento dos componentes, garantindo assim, a consistência do esquema. Estes requisitos foram fortemente considerados e influenciaram na modelagem das conexões e na implementação do modo conexão.

A figura 6.17 apresenta a modelagem das conexões no domínio visual, apresentando inclusive alguns dos atributos e métodos definidos nas classes que fazem parte do modelo.

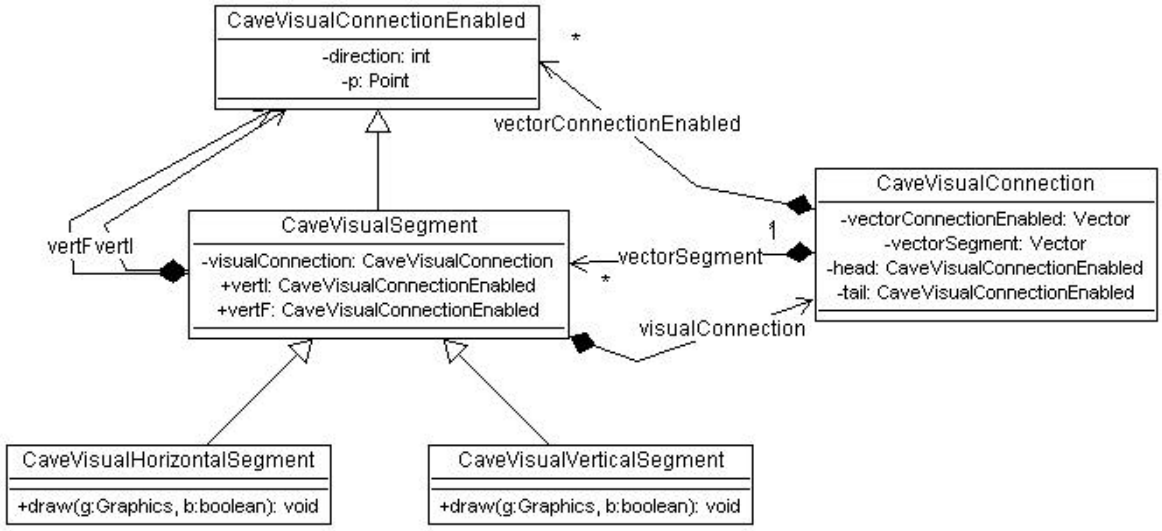


FIGURA 6.17 - Modelagem das conexões no domínio visual

A fim de permitir conexões que ligam três ou mais pinos, foi criada a classe *CaveVisualConnectionEnabled*, que tem como subclasses, *CaveVisualSegment* e *CaveVisualPort*. Assim, uma conexão pode fazer a ligação entre duas instâncias de *CaveVisualConnectionEnabled*, permitindo que uma conexão seja criada para conectar pinos, instâncias de C*aveVisualPort*, ou segmentos, instâncias de *CaveVisualSegment*.

O usuário especifica através do clique do mouse dois pontos, correspondendo a duas instâncias de *CaveVisualConnectionEnabled*, os quais deseja conectar. Quando o botão do mouse é solto, o sistema gera automaticamente os segmentos que formam a conexão. Portanto, uma conexão é criada quando o modo conexão está ativado e o usuário clica em dois pontos pertencentes a pinos de componentes ou pertencentes a segmentos que fazem parte de outras conexões. Posteriormente, deseja-se permitir ao usuário especificar os pontos que formam os segmentos, assim dando maior liberdade ao usuário no desenho das linhas de conexão.

Cada conexão no domínio visual, ou seja, cada instância de *CaveVisualConnection*, possui um vetor de *CaveVisualSegment* e um vetor de *CaveVisualConnectionEnabled* usados para o desenho das linhas que formam a conexão. Uma conexão no domínio semântico (*CaveDesignConnection*) possui um vetor de *CaveVisualConnection* e um vetor de *CaveDesignPort*. A figura 6.18 mostra os

relacionamentos entre as classes do Blade que modelam as conexões no modelo visual e semântico.

Um segmento, instância de *CaveVisualSegment*, possui duas referências para *CaveVisualConnectionEnabled* indicando os extremos do segmento (*VertI* e *VertF*). Quando um componente é movimentado, as conexões que envolvem os seus pinos devem ser reposicionadas. Para que isso ocorra, toda vez que o universo é redesenhado, as instâncias de *CaveVisualSegment* atualizam as coordenadas de seus vértices, através das referências *VertI* e *VertF*, antes de serem redesenhadas.

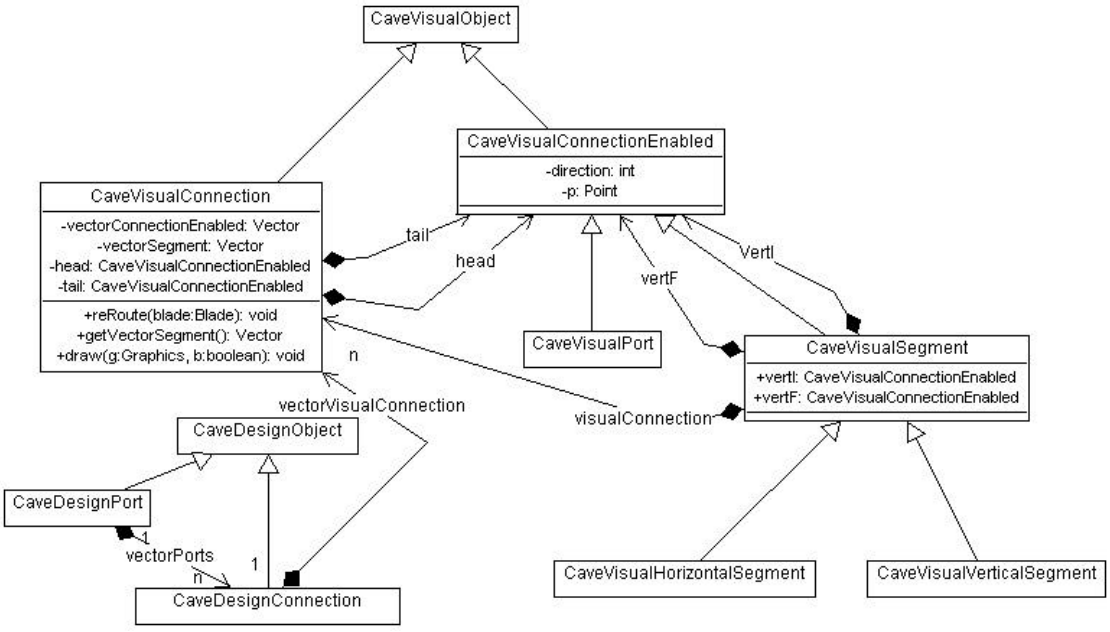


FIGURA 6.18 - Modelagem das conexões no domínio visual e semântico

Todos os *CaveVisualConnectionEnabled* possuem o método *getPoint( )* que retorna o ponto onde as conexões podem ser ligadas. Este ponto é determinado a partir da direção (*direction*), outro atributo dos objetos *CaveVisualConnectionEnabled*. Por exemplo, na figura 6.19, a porta *out* está do lado direito do bloco e portanto, sua direção é *east*. A partir da informação de direção, das coordenadas e da dimensão da porta, pode-se determinar as coordenadas do ponto P, que será o vértice de uma conexão que liga esta porta *out* à outra porta qualquer. Este método é usado na criação de uma nova conexão, bem como para atualizar a posição dos vértices dos segmentos quando algum dos pinos que fazem parte da conexão, forem movimentadas.

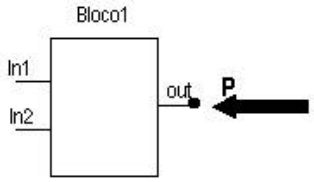


FIGURA 6.19 - Ponto para conexão

Para o desenho de uma conexão é preciso determinar os pontos intermediários usados na geração dos segmentos que fazem parte da conexão, um exemplo pode ser visualizado na figura 6.20. Para a determinação dos pontos intermediários implementou-se um algoritmo recursivo baseado nos pontos e na direção dos segmentos que serão conectados. Este algoritmo baseia-se em procedimentos encontrados em uma biblioteca de classes Java chamada Diva [REE 98], que foi usada na implementação da ferramenta *Ptolemy* [LEE 2002]. Diva é uma arquitetura de software para visualização e interação com espaços de informações dinâmicas, sua versão atual foca infraestruturas de visualização de grafos.

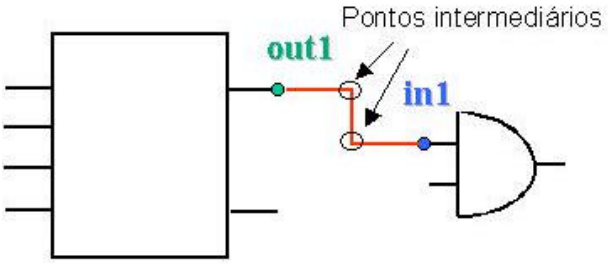


FIGURA 6. 20 - Pontos intermediários

No contexto do Blade, este algoritmo foi implementado em uma nova classe chamada *CaveOrthogonalConnection* e é dedicado ao desenho de conexões ortogonais. Visando uma maior flexibilidade na implementação das conexões, foi criada uma classe abstrata chamada *CaveConnectionPolicy* da qual a classe *CaveOrthogonalConnection* é filha. Esta abordagem facilita a criação de novas estratégias de roteamento dos segmentos que formam as conexões, facilitando a extensão do editor para outras formas de diagrama.

A figura 6.21 apresenta um diagrama de classes do modo conexão, no qual observa-se os relacionamentos entre as classes Blade, *BladeConnectionMode* e *CaveOrthogonalConnection*. Observa-se que a classe Blade possui uma referência para um objeto da classe *BladeConnectionMode*, classe que define o funcionamento do modo conexão. E por sua vez, a classe *BladeConnectionMode* tem uma referência para um *CaveConnectionPolicy* que define a política utilizada para o desenho das conexões.

Nesta implementação, utilizou-se o padrão de projeto *Strategy* que foi apresentado na seção 4.2.2.5 deste trabalho. Fazendo uma analogia entre as classes que fazem parte deste padrão e as classes utilizadas na implementação, pode-se dizer que a classe *BladeConnectionMode* faz o papel da classe "*Context*" do padrão *Strategy*, a classe *CaveConnectionPolicy* representa a classe "*Strategy*" e a classe *CaveOrthogonalConnection* pode ser considerada uma "*ConcreteStrategy*".

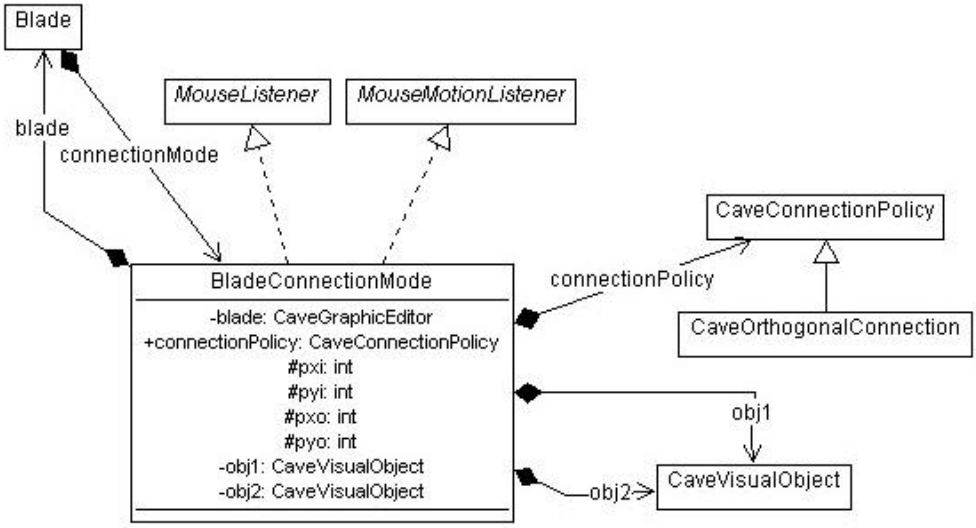


FIGURA 6.21 - Implementação Modo Conexão (*BladeConnectionMode*)

A figura 6.22 apresenta um *snapshot* do primeiro protótipo Blade, que possuía as funcionalidades básicas do editor. Pode-se observar nesta figura que este protótipo permite a montagem de diagramas lógicos, através da inserção, edição e remoção de componentes, bem como através da criação de conexões entre eles.

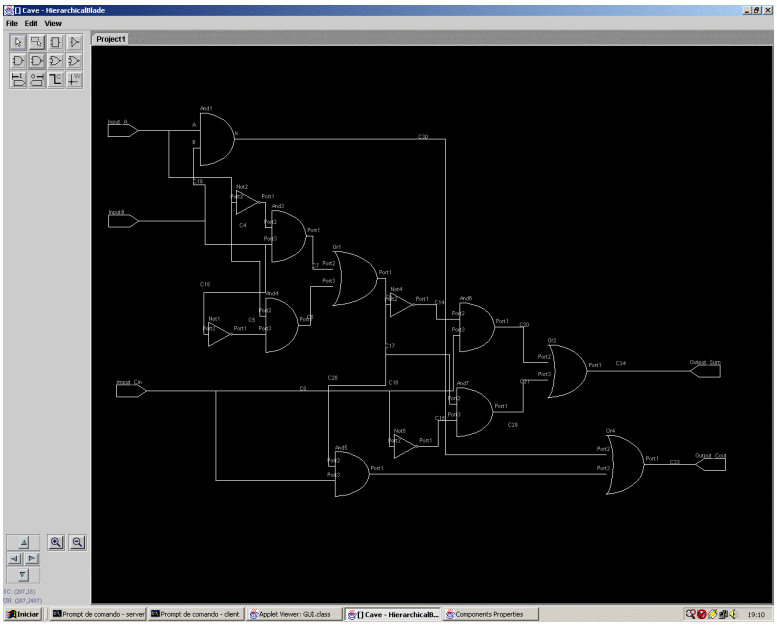


FIGURA 6.22 - Protótipo do Blade

6.6 Configurações do Usuário

É importante que uma ferramenta permita que o usuário defina algumas configurações, tais como preferência de cores, fontes, idioma. No Blade, de forma a permitir que o usuário configure algumas propriedades, foi criado o item de menu ***Options***, o qual possui as seguintes opções: ***Colors***, ***User Directory*** e ***Netlist Format***. Na figura 6.23 pode-se observar a interface do Blade, após a inserção deste item menu.

O item ***Netlist Format*** permite a especificação do formato do *netlist* que o usuário irá gerar ao final da especificação do diagrama. A geração *netlist* é acionada através do item de menu ***Netlist*** da barra de menu da ferramenta. A opção ***Colors*** é responsável por permitir a especificação das cores da área de trabalho. O item ***User Directory*** permite a especificação do diretório de trabalho, esta informação é usada para definir onde são armazenados os projetos. As configurações de cores e de diretório de trabalho não estão disponíveis nesta versão.

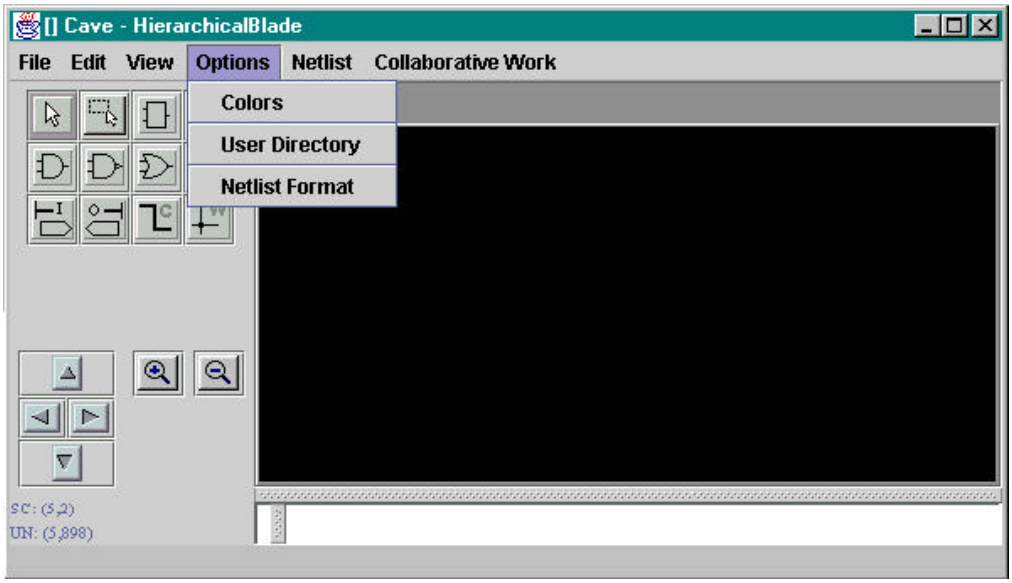


FIGURA 6.23 - Barra de Menu *Options*

Através do item ***Netlist Format***, o usuário pode selecionar em uma lista de formatos padrões suportados pela ferramenta, o formato que será utilizado na geração do *netlist*. A janela que permite esta configuração pode ser observada na figura 6.24, na qual pode-se observar dois formatos sendo disponibilizados para seleção. Porém, nesta versão da ferramenta, apenas a geração de descrições usando a linguagem VHDL são suportadas. A geração do *netlist* implementada no protótipo é abordada na seção 6.8.

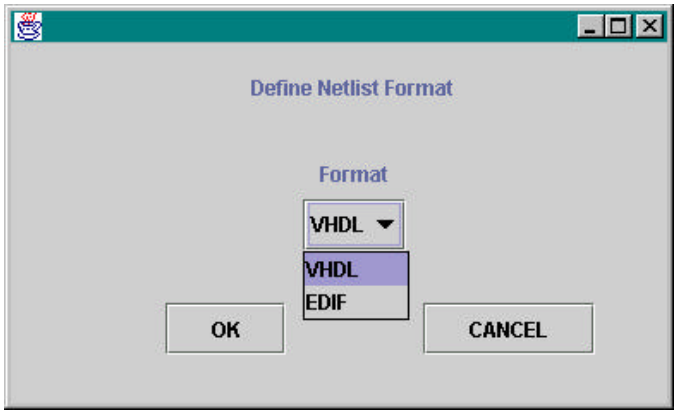


FIGURA 6.24 - Janela de configuração do formato *netlist*

6.7 Hierarquia

Além das funcionalidades apresentadas na seção 6.4, conforme a especificação apresentada no capítulo 5, o editor também permite a montagem de projetos hierárquicos. Esta seção aborda a implementação do suporte a projetos hierárquicos no Blade. Primeiramente, vamos ver em detalhe o modelo de dados que é utilizado para permitir a implementação da hierarquia. Na figura 6.25, pode-se observar um diagrama de classes, no qual estão representadas as relações que foram criadas entre as classes *CaveDesignBlock* e *CaveDesignSemantic*, a fim de possibilitar a criação de projetos hierárquicos no Blade.

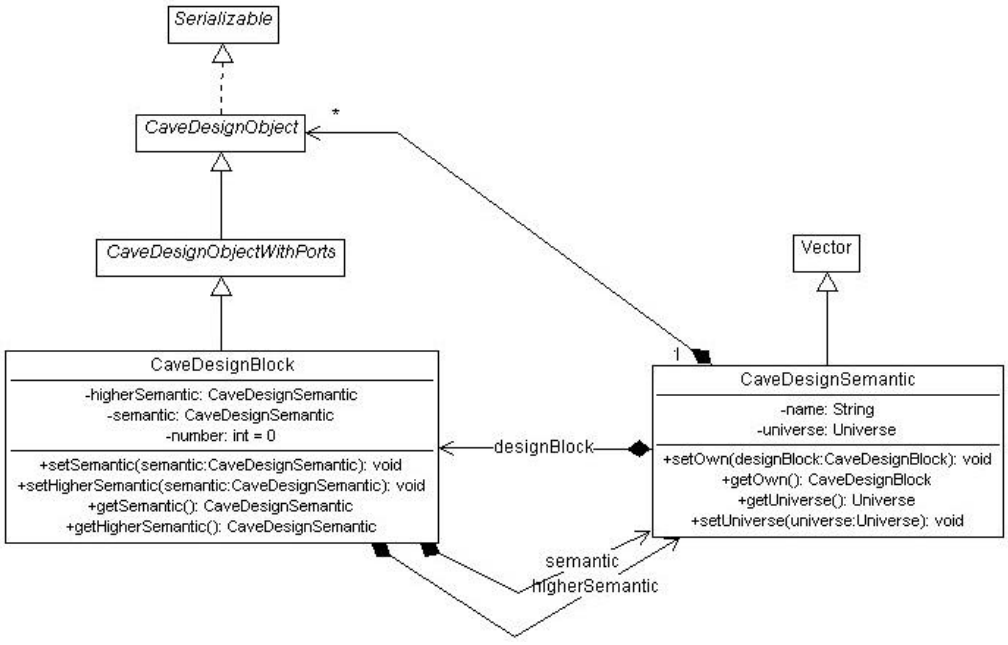


FIGURA 6.25 - Relação entre as classes *CaveDesignSemantic* e *CaveDesignBlock*

Um objeto do tipo *CaveDesignSemantic* representa um vetor de objetos de projeto (*CaveDesignObject*), vetor chamado *semantic*, e possui uma referência para o *Universe*, que é o vetor de objetos visuais (vetor de *CaveVisualObject*). Em projetos hierárquicos, um objeto do tipo *CaveDesignSemantic* é associado a um objeto da classe *CaveDesignBlock*, assim o vetor *semantic* pode ser usado para representar o funcionamento de um bloco. Quando um bloco possui uma especificação, ou seja, uma representação em um nível de hierarquia mais baixo, ele terá uma referência para um objeto do tipo *CaveDesignSemantic*, que representará o seu funcionamento.

Além da implementação da hierarquia, a classe *CaveDesignSemantic* é usada também para a geração de uma descrição *netlist*. Veja na seção 6.8 como esta classe é usada para gerar uma descrição VHDL do circuito.

Quanto à interface da ferramenta, também foram requisitadas algumas mudanças para suporte ao projeto hierárquico. Uma desta alterações deu-se na área de trabalho do editor. A fim de permitir a edição e visualização de múltiplos níveis de hierarquia de um projeto, foi inserido na área de trabalho da ferramenta um componente conhecido como *JTabbedPane*. Este componente, também chamado de “abas” ou “cortinas”, faz parte do pacote *Java Swing* e permite que o usuário navegue de um painel para outro. Assim, toda vez que um nível de hierarquia é criado no projeto, uma nova aba é inserida na área de trabalho. Esta interface permite também a fácil navegação entre os diversos níveis de hierarquia e pode ser observada na figura 6.26.

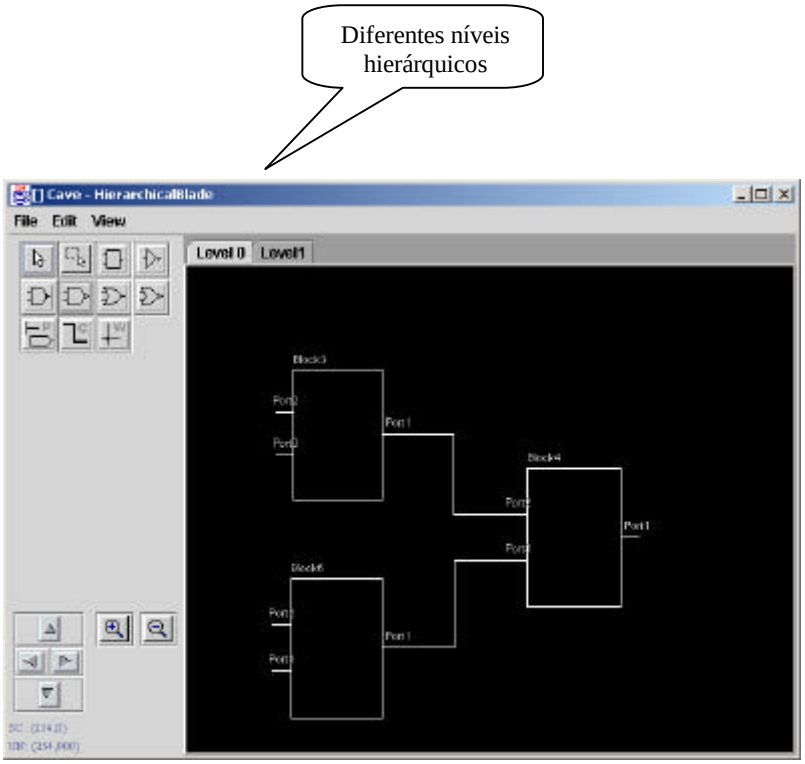


FIGURA 6.26 - Protótipo do Blade com Hierarquia – Nível Superior

A figura 6.27, abaixo, mostra a especificação de um dos blocos que compõem o esquema da figura anterior, onde pode ser visualizado um nível de hierarquia inferior.

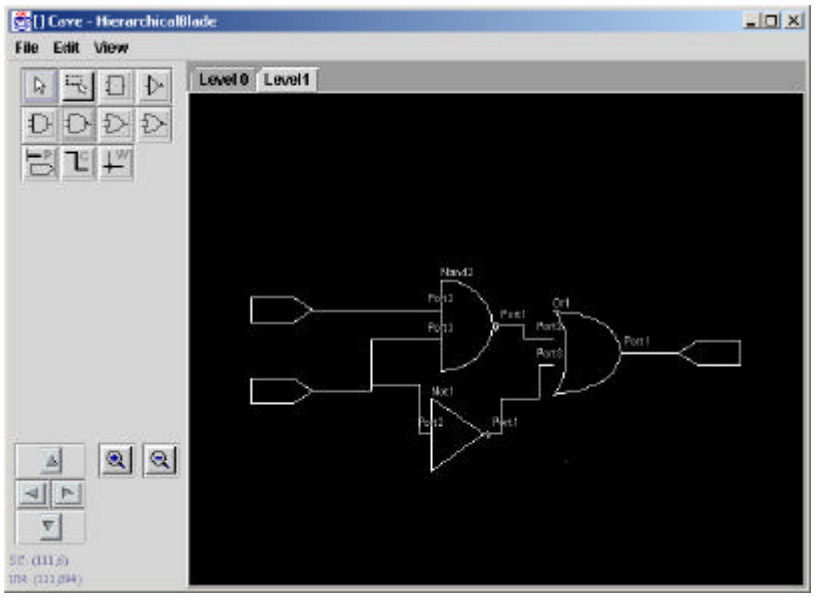


FIGURA 6.27 - Nível Inferior do Projeto

Cada um dos níveis de hierarquia representa uma instância da classe *Canvas* (área de trabalho da ferramenta) e possui uma referência para um objeto do tipo *Universe* (universo) e para um objeto do tipo *CaveDesignSemantic* (semântica). Portanto, quando a ferramenta está operando no modo hierárquico, ela deve manter um vetor de canvas, um vetor de universos e um vetor de semânticas.

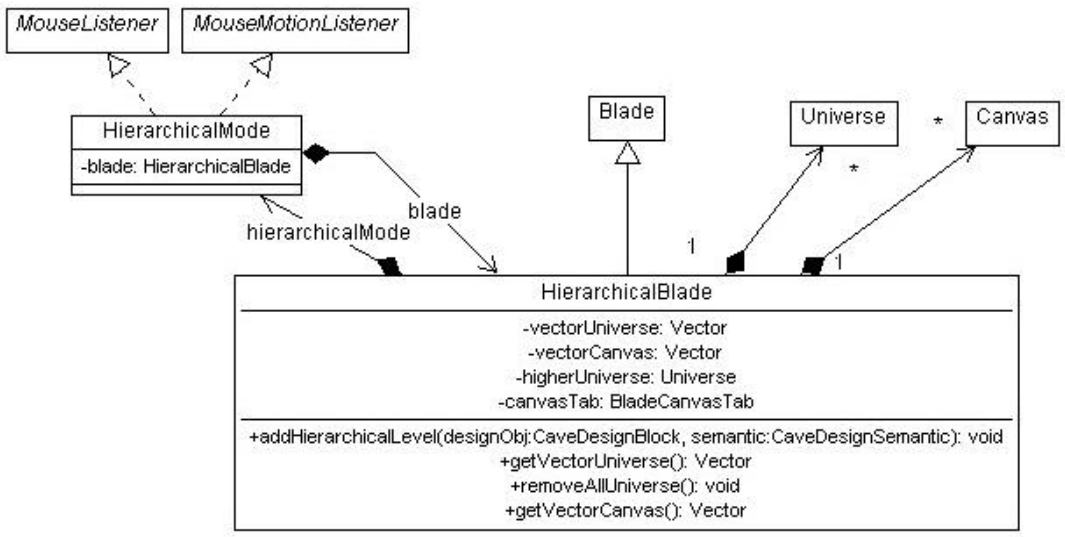


FIGURA 6.28 - Implementação da Hierarquia

Quando um projeto hierárquico é salvo, tanto em disco local quanto em um repositório de dados remoto, todos os objetos do vetor de universos (*vectorUniverse*) e do vetor de semânticas (*vectorSemantic)* são serialiazados e tornados persistentes. Assim, todas as visualizações e todos os dados de projeto, referentes aos diferentes níveis de hierarquia do projeto, são salvos. Desta forma, num outro momento, quando este projeto for recuperado, o projetista poderá visualizar e navegar entre os seus diferentes níveis de hierarquia.

Um nível de hierarquia é criado, quando o projetista dá um duplo clique em um bloco funcional que foi inserido no esquema e que ainda não possui uma semântica definida. Neste caso, uma nova "aba" deve ser adicionada no *JTabbedPane*, na qual um novo *Canvas* será inserido e permitirá a edição do esquema referente ao funcionamento deste bloco. A exclusão de um nível de hierarquia ocorre quando o bloco cuja descrição se encontra neste nível for excluído. Neste caso, a "aba" do *JTabbedPane* referente a este nível de hierarquia e, portanto, referente à semântica do bloco, deve ser removida.

6.8 Geração do *Netlist*

Nesta seção, aborda-se a implementação da geração do *netlist* de um circuito, ou seja, a sua descrição textual usando o formato selecionado pelo usuário através do menu ***Options,*** que pode ser visualizado na figura 6.23. É importante salientar aqui que, nesta primeira implementação do Blade, somente são geradas descrições em formato VHDL para os diagramas lógicos construídos através do Blade. Esta geração é acionada através do item de menu ***Netlist*** inserido na barra de menu. Este menu pode ser observado na figura 6.29 e possui dois subitens: ***Export Sheet*** e ***Export Project***.

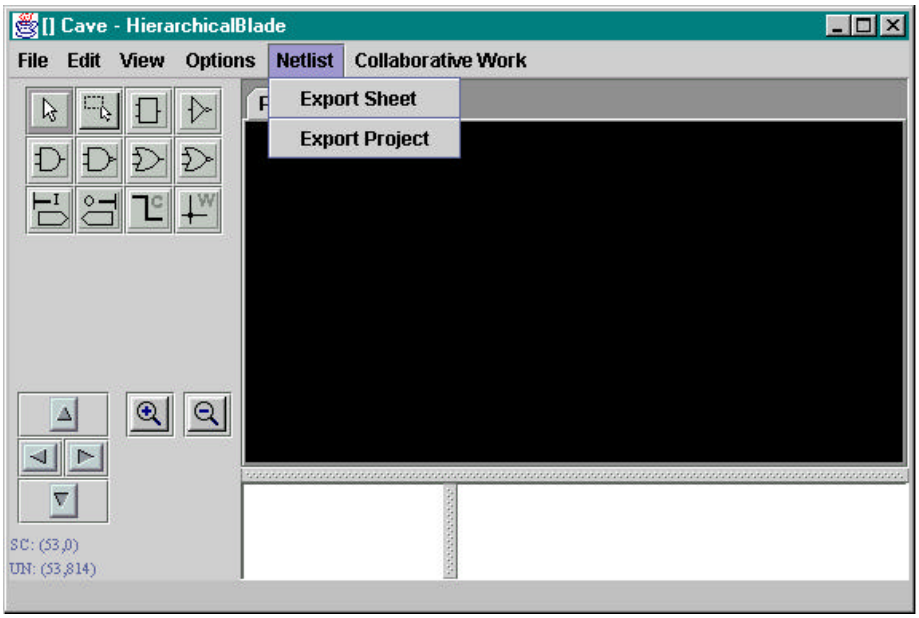


FIGURA 6.29 - Item de Menu *Netlist*

Como foi mostrado na seção 6.7, a ferramenta permite que o usuário trabalhe com projetos hierárquicos e portanto, visualize na área de trabalho mais de um nível de hierarquia do projeto. Para permitir que o usuário indique para qual dos níveis ele deseja gerar a descrição, foi criada a opção ***Export Sheet*** que gera a descrição apenas para o nível de hierarquia que estiver selecionado. A opção ***Export Project*** é usada quando o usuário deseja gerar a descrição de todo o projeto, ou seja, de todos os níveis de hierarquia envolvidos na especificação.

6.8.1 Geração VHDL

Os dados necessários para a geração VHDL do circuito estão todos no domínio semântico, são eles basicamente: a lista de componentes, as interconexões entre os componentes. No caso das portas lógicas, as suas funções lógicas também são utilizadas. No caso de projetos hierárquicos, há também a relação de hierarquia entre os componentes do projeto.

A linguagem VHDL permite a descrição de projetos hierárquicos, através da primitiva chamada *component*, que permite a criação de várias instâncias de um mesmo circuito. Quando esta primitiva é usada, vários arquivos VHDL são utilizados, um para cada circuito diferente que faz parte do projeto, além de um outro arquivo que representa o nível mais alto de hierarquia do projeto e é responsável pela interconexão entre as instâncias que fazem parte do projeto.

A fim de facilitar a geração, optou-se por considerar que todos os componentes que fazem parte do esquema são instâncias de componentes definidos em códigos VHDL. Esta abordagem foi utilizada até mesmo para as portas lógicas, assim, facilitando a geração da descrição, visto que tanto blocos como portas lógicas são tratados da mesma maneira. Para cada componente que faz parte do esquema, é gerado um outro arquivo VHDL que descreve o comportamento do componente. Na descrição de mais alto nível de hierarquia, cria-se as instâncias dos componentes e faz-se a interconexão entre elas.

Seguindo esta idéia, para gerar uma descrição VHDL de um esquema lógico, primeiramente, deve-se percorrer o vetor de objetos de projeto (instância de *CaveDesignSemantic*), descrevendo os componentes presentes no vetor e suas interconexões. Posteriormente, deve-se gerar a descrição para todos os componentes que fazem parte do esquema em arquivos separados. No protótipo do Blade, os nomes dos arquivos são gerados automaticamente, baseados no nome do projeto e no nome dos blocos e demais componentes que fazem parte do projeto. Assim, o usuário fica liberado da nomeação dos arquivos, que pode ser uma tarefa enfadonha se houver um grande número de arquivos a serem gerados.

No caso de projetos hierárquicos, observe na seção 6.7, há um vetor de *CaveDesignSemantic*, um para cada nível de hierarquia. Portanto, primeiro deve-se

gerar a descrição correspondente ao nível superior de hierarquia a partir do objeto do tipo *CaveDesignSemantic* (vetor de objetos de projeto), correspondente a este nível de hierarquia. Os componentes do tipo *CaveDesignBlock* (bloco funcional) possuem uma referência para um *CaveDesignSemantic* que representa o comportamento do bloco e portanto, um nível hierárquico inferior. Portanto, a geração da descrição VHDL de um bloco é realizada a partir do objeto *CaveDesignSemantic* ao qual este faz referência.

Seguindo este estilo de descrição, todo o componente deve ser capaz de gerar sua descrição VHDL. Porém, é importante ressaltar que a descrição do VHDL de uma porta lógica é diferente da descrição de um componente do tipo bloco, visto que um bloco pode ter uma descrição em outro nível de hierarquia. Portanto, criou-se o método abstrato *generateVHDL()* na classe *CaveDesignObjectWithPorts*, assim as suas subclasses, *CaveDesignLogicGate* e *CaveDesignBlock*, devem implementar este método. Desta forma, todos os componentes, sejam eles portas lógicas ou blocos, geram suas descrições VHDL através deste método.

Um outro método denominado *generateInterface( )* também foi definido na classe *CaveDesignObjectWithPorts*. Ele é usado para gerar a interface de um componente, ou seja, a definição dos pinos de entrada e saída, que faz parte da descrição VHDL de um circuito. Este método é invocado através do método *generateVHDL ( )*. Na figura 6.30, pode-se observar a localização destes métodos em um diagrama de classe simplificado da implementação do Blade.

Como dito anteriormente, a descrição de um bloco é realizada a partir de sua referência para um objeto do tipo *CaveDesignSemantic*, que é um vetor de objetos de projeto. Este vetor é chamado de *semantic* e representa todos os componentes presentes naquele nível de hierarquia. Porém, no caso da descrição de uma porta lógica, basta conhecer a função lógica correspondente àquela porta. Portanto, foi criado um método abstrato chamado *getLogicFunction( )* na classe *CaveDesignLogicGate* que retorna a função lógica da porta. Este método foi implementado nas subclasses e retorna um *string* que é usado na descrição do comportamento da porta. Por exemplo, retorna o *string* "AND" para uma instância da classe *CaveDesignAnd*.

Na classe *CaveDesignObjectWithPorts*, foi criado um método chamado *OrganizePortListForType()* que verifica se uma porta do componente é de entrada (INPUT), de saída (OUTPUT), ou bidirecional (INOUT) e reorganiza a lista de pinos do componente de acordo com o tipo da porta. Este método fez-se necessário porque os pinos (*CaveDesignPort*) estão organizadas em vetores conforme sua localização. Por exemplo, os pinos do lado esquerdo estão armazenadas no vetor *westp* e os pinos do lado direito estão no vetor *eastp*. O método *organizePortListForType( )* é invocado a partir do método *generateInterface( )*, que por sua vez é chamado a partir do método *generateVHDL( )*.

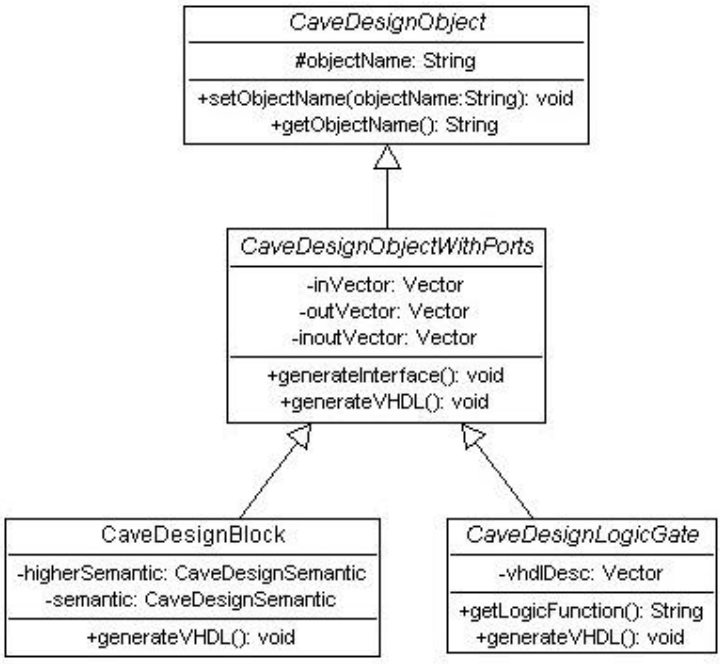


FIGURA 6.30 - Diagrama de Classes – Implementação da geração VHDL

Na classe *HierarchicalBladeHandler*, onde são tratados os eventos do menu, foi definido o método *generateNetlist( )* que verifica qual o formato selecionado pelo usuário e chama o método adequado para a geração da descrição. Um esboço deste método pode ser visualizado na figura 6.31.

```
public void generateNetlist (String format)

/* Método chamado pelo menu Netlist, verifica o formato e chama o método
correspondente à geração naquele formato                                  * /
{
        if (format == "VHDL")
        {
                generateVHDL (allSemantic);
        }
....
...
..
}
```

FIGURA 6.31 - Método *generateNetlist( )*

O método *generateNetlist( )* é invocado a partir do item de menu ***Netlist,*** através das opções ***Export Sheet*** e ***Export Project***. Quando aciona-se a geração através do item *Export Sheet*, o parâmetro *allSemantic* será definido como *false*, indicando que a geração deve ser realizada somente no nível de hierarquia selecionado pelo usuário.

Caso a geração seja disparada através do item *Export Project*, *allSemantic* é definido como *true* e portanto, a geração deve ser feita para todos os níveis de hierarquia.

A classe *HierarchicalBladeHandler* disponibiliza também o método *generateVHDL( )* cujo esboço pode se visualizado na figura 6.32. Este método tem como parâmetro um valor lógico (*allSemantic*), que indica se a geração deve se dar sobre todo o projeto ou somente em um dos seus níveis de hierarquia. Portanto, ele verifica o valor do parâmetro *allSemantic* e dispara a geração das descrições, fazendo chamadas aos métodos *generateVHDL( )* de cada um dos componentes que fazem parte da especificação.

```
public void generateVHDL (boolean allSemantic)

/* Método usado para gerar a descrição VHDL do esquema, é chamado pelo nétodo
generateNetlist(), baseado no valor de allSemantic e gera o netlist para todas as
semânticas ou apenas para a semântica selecionada *?                    * /
{

        if (allSemantic)
        {
                        ....
        }

....
...
..
```

FIGURA 6.32 - Método *generateVHDL( )*

6.9 Implementação do Serviço de Colaboração

A colaboração sobre o ambiente Cave foi abordada no capitulo 3, na seção 3.2.3, onde apresenta-se o serviço de colaboração proposto por Sawicki [SAW02]. Este serviço disponibiliza um repositório de dados seguro e também uma ferramenta de comunicação, que permite a troca de mensagens entre os membros de um grupo de projetistas.

O repositório de dados permite o armazenamento persistente dos dados de projeto. Na implementação deste serviço de persistência, utiliza-se tecnologia *JavaSpace*, na qual os dados são armazenados como objetos Java. Portanto, neste repositório, podem ser armazenados diferentes formatos de dados, manipulados por diferentes ferramentas de CAD, desde que estes possam ser convertidos para um objeto da classe *Object* da linguagem Java. Nas ferramentas desenvolvidas na linguagem Java, esta conversão não é necessária.

O serviço de colaboração disponibiliza um serviço de transação, assim todas as atualizações ocorrem dentro de uma transação, o que garante a consistência dos dados de projeto. Um serviço de autenticação e um controle a acessos concorrentes também são disponibilizados.

Conforme a metodologia que está sendo utilizada neste serviço, apenas um projetista tem o direito de escrita a cada momento, sendo este chamado de escritor (*writer*) e os demais denominados leitores (*listeners*). As atualizações nos dados do repositório seguem o modelo de atualização e notificação (*update / notify*), portanto, há um agente que observa o repositório e quando uma alteração é feita pelo projetista escritor, este notifica todos os leitores.

6.9.1 Integração do serviço de colaboração ao Blade

A integração do serviço de colaboração implementado por Sawicki à ferramenta Blade mostrou-se uma tarefa simples. Principalmente devido à flexibilidade do serviço, característica que foi priorizada em sua implementação, permitindo incorporar o serviço em diferentes ferramentas.

Nesta proposta de colaboração, os dados de projeto são armazenados no repositório de dados e compartilhados pelos projetistas. O serviço de colaboração disponibiliza um método para recuperar os dados armazenados no repositório, porém a ferramenta deve implementar um método a fim de permitir a visualização dos dados recuperados do repositório. Assim como, a ferramenta deve implementar também um método que indica os dados que serão encapsulados e armazenados no repositório. Isso porque este trata todos os dados de projeto de uma mesma forma, sejam estes representações textuais ou gráficas. Desta forma, somente a ferramenta precisa conhecer o seu formato de dados interno. Portanto, têm-se um serviço genérico que pode ser incorporado a qualquer das ferramentas do *framework* Cave.

Assim como, na opção de salvar para arquivo que é suportada pelo editor de diagramas através do menu ***File***, opção ***Save to Disk***, os dados de projeto são serializados a fim de serem armazenados no repositório de dados. A serialização dos dados de projeto é feita através da interface *Serializable* disponibilizada no pacote j*ava.io*.

Num projeto hierárquico colaborativo, quando uma alteração é feita em um dos níveis de hierarquia, esta alteração deve ser repassada aos demais projetistas. As questões ligadas à hierarquia não precisam ser tratadas dentro do serviço de colaboração, ficando somente a cargo da ferramenta, visto que, quando um projeto hierárquico é salvo, todos os níveis de hierarquia do projeto são salvos no repositório. Quando o mesmo projeto for recuperado pela ferramenta, esta é quem deve estar preparada para montar as visualizações dos diferentes níveis de hierarquia do projeto.

O desenvolvedor da ferramenta precisa implementar a interface ***DocumentListener*** na classe *Handler* – a estrutura do *framework* Cave padroniza a utilização de uma classe *Handler*, o que pode ser observado na seção 6.2, que aborda a integração de uma ferramenta ao Cave. Na definição da classe *Handler* é necessário também que sejam definidos os pacotes do módulo de colaboração, chamados ***Collaborative.connect***, ***Collaborative.chat*** *e* ***Collaborative.service***. Essa declaração é muito importante, pois permite que a ferramenta localize o conjunto de classes do serviço de colaboração.

Portanto, no Blade, a classe *HierarchicalBladeHandler* implementa a interface *DocumentListener* e define o uso dos pacotes do módulo de Colaboração como pode-se observar na figura 6.33. Nesta classe também são implementados os métodos ***OpenProject*** e ***SaveProject*** que definem respectivamente o que a ferramenta faz para mostrar os dados retornados do repositório e que dados a ferramenta salva no mesmo.

A interface *DocumentListener* faz parte *javax.swing.event* da *Sun* e implementa três métodos abstratos chamados ***insertUpdate()***, ***changeUpdate()*** e ***removeUpdate()***. Estes métodos recebem as mensagens de notificação do módulo de cooperação, tais como: permissões de escrita / leitura, atualizações de projeto e atualizações nos participantes e as mensagens do *chat*.

```
import cave.collaborative.connect.*;
import cave.collaborative.chat.*;
import cave.collaborative.service.*;

public class HierarchicalBladeHandler … implements … DocumentListener
{
…
   public void insertUpdate(DocumentEvent e) {}
   public void changedUpdate(DocumentEvent e){}
   public void removeUpdate(DocumentEvent e) {}
…
}
```

FIGURA 6.33 - Inserção de colaboração no Blade - Classe *HierarchicalBlade*

Na implementação do serviço, apenas o método ***InsertUpdate()*** está sendo utilizado. Este método recebe todas as notificações e atualizações dos projetos e dos participantes. Os métodos *removeUpdate* e *changeUpdate()* são definidos como métodos vazios, sem funcionalidade.

Além disso, a ferramenta precisa também criar uma interface apropriada para projetos colaborativos. Sawicki propôs, juntamente com o serviço de colaboração, uma interface a ser incorporada nas ferramentas e que consiste em um item de menu ***Collaborative Work***, um painel com a lista de participantes on-line e informações relativas a direitos de escrita dos mesmos e uma barra de *status* que permite visualizar o direito de escrita. A barra de *status* é verde quando o projetista tem direito de escrita,

caso contrário, é vermelha. A interface proposta por Sawicki foi incorporada ao Cave e pode ser observada na figura 6.34.

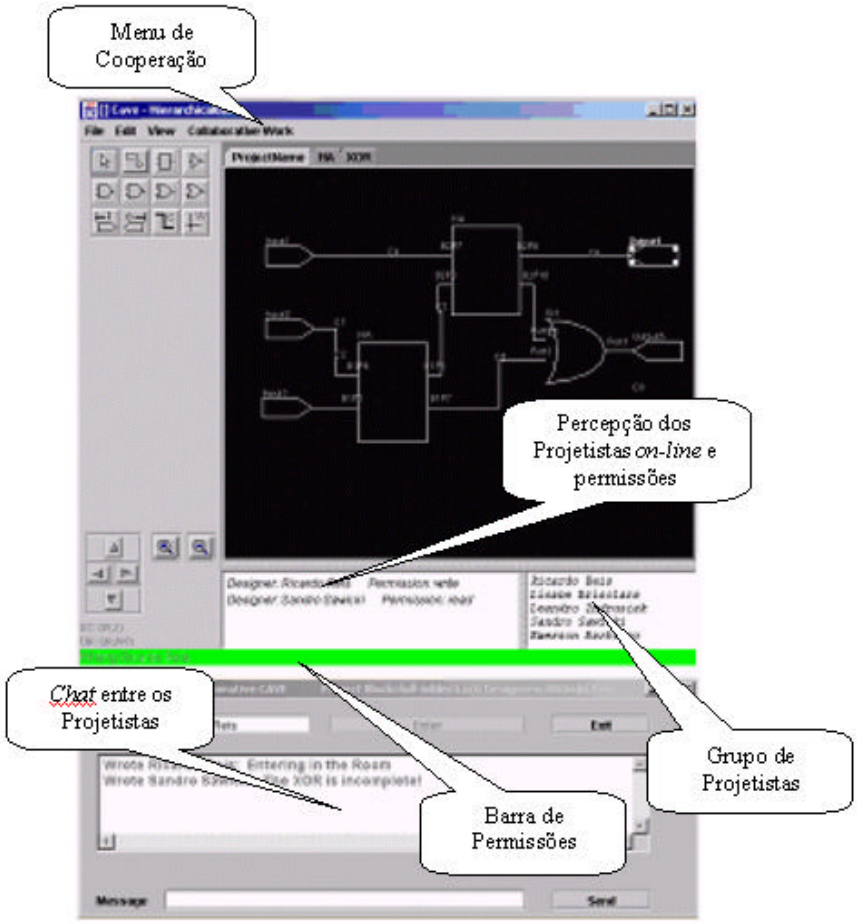


FIGURA 6.34 - Interface voltada a Cooperação

Juntamente com o serviço de colaboração são disponibilizadas duas classes que podem ser utilizadas para a montagem da interface proposta por Sawicki. São elas: *CollaborativeExtendMenu* e *CollaborativeExtendedService*. A classe *CollaborativeExtendMenu* é a encarregada de adicionar a interface gráfica do módulo de cooperação. Para que isso aconteça, é necessário executar um método estático chamado *createMenu()*, passando como parâmetro o *menubar* padrão da ferramenta e sua interface *ActionListener*. A interface *ActionListener* será usada para tratar dos eventos de clique do mouse no menu criado.

Na figura 6.35, pode-se visualizar a interface do Blade após a inserção do novo item de menu, chamado *Collaborative Work*, e as opções disponíveis neste menu.

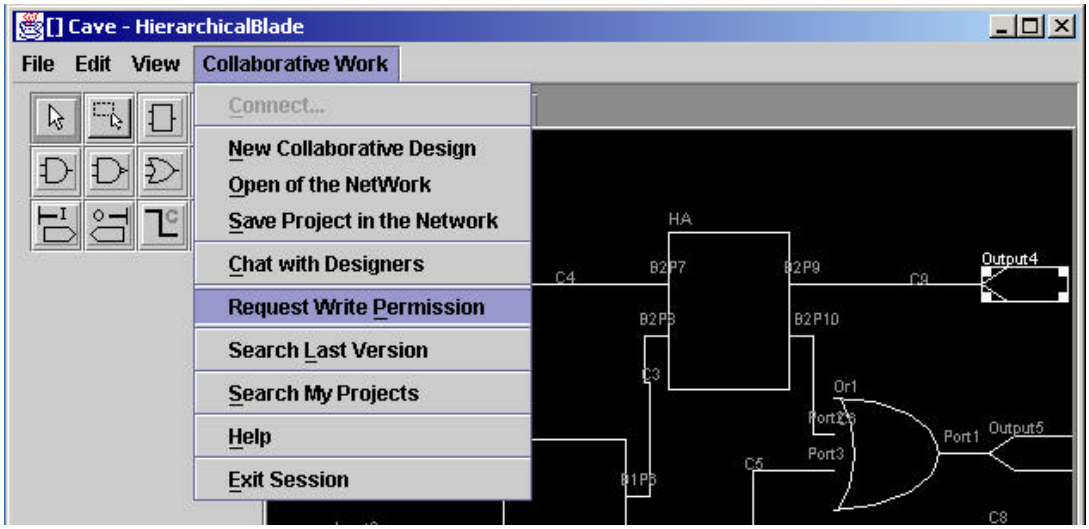


FIGURA 6.35 - Interface do Blade - Menu *Collaborative Work*

O item de menu ***Collaborative Work*** permite a execução de funcionalidades relacionadas com a colaboração, tais como: criar projeto colaborativo, abrir projeto colaborativo, salvar projeto, requisitar direito de escrita, entrar no *chat* e deixar a sessão de projeto. Para facilitar a criação e incorporação deste menu nas ferramentas, as classes ***CollaborativeExtendService*** e ***CollaborativeExtendMenu*** foram disponibilizadas juntamente com o serviço de colaboração.

A opção ***Connect*** faz a comunicação da ferramenta com a tabela de locação de serviços (TLS), que retorna a referência dos serviços de transação e persistência, deixando a ferramenta pronta para a cooperação. Antes de estabelecer a conexão, somente as opções ***Connect*** e ***Help*** estão disponíveis, conforme pode ser observado na figura 6.36.

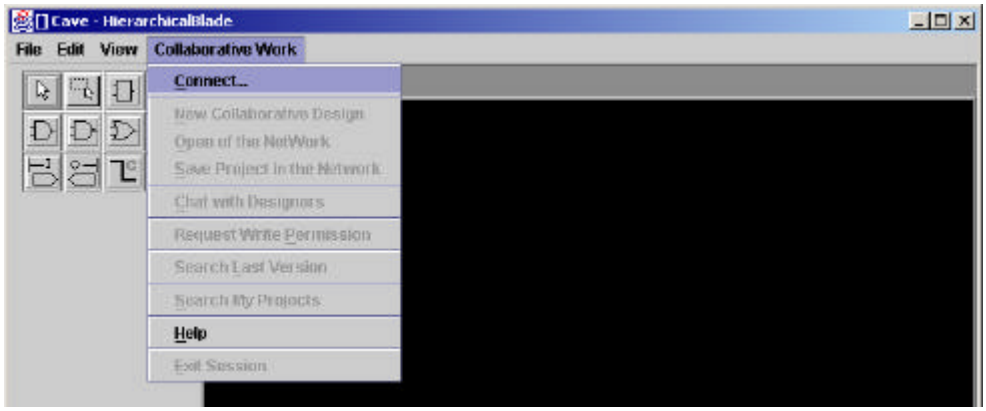


FIGURA 6.36 - Conectando com o Servidor

Após o estabelecimento da conexão, são habilitadas as opções ***Open Coollaborative Project*** e ***New Collaborative Project***, conforme pode-se observar na figura 6.37. As demais opções que podem ser visualizadas na figura 6.35 somente serão

disponibilizadas quando o projetista estiver com um projeto aberto ou que o mesmo tenha criado um novo projeto.

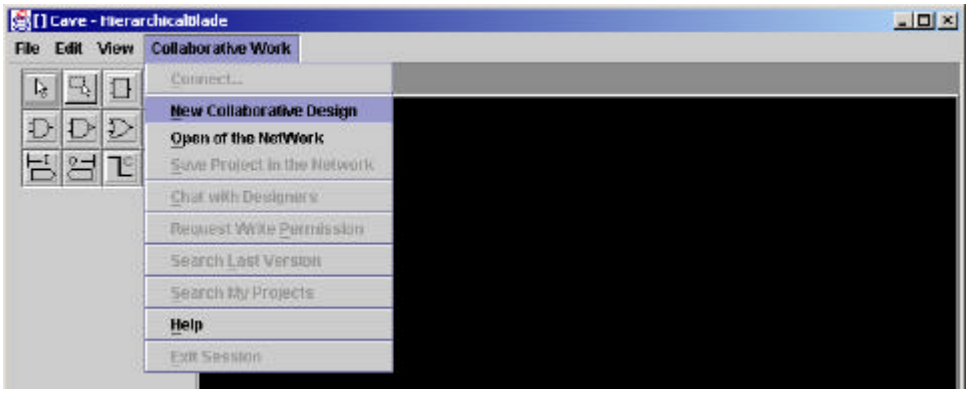


FIGURA 6.37 - Projetista Conectado

A opção ***New Collaborative Design*** cria um novo projeto cooperativo e o armazena no repositório de projetos. Esse procedimento, assim como as atualizações, é realizado dentro de uma transação. A figura 6.38 ilustra a criação de um novo projeto cooperativo, adicionando o nome do projeto, seu responsável, uma senha de acesso, os blocos que compõem esse projeto, e por último a inserção do grupo de projetistas que irão trabalhar nesse projeto.

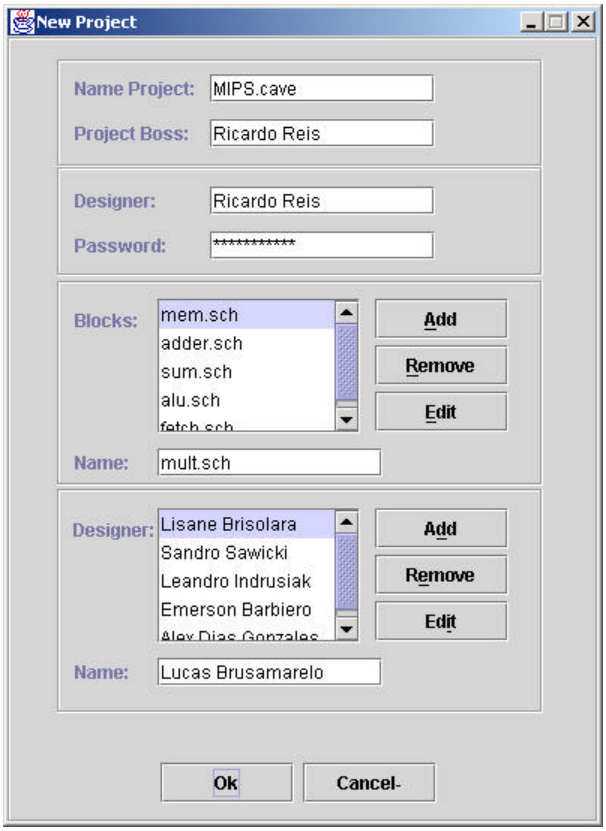


FIGURA 6.38 - Criação de um novo projeto cooperativo

A opção ***Open of the Network*** recupera projetos armazenados no repositório. A recuperação de um projeto e seus blocos é realizada através do nome do projeto e nome do líder do projeto, *Project Name e Project Líder Name*, parâmetros que são inseridos através da interface gráfica apresentada na figura 6.39.

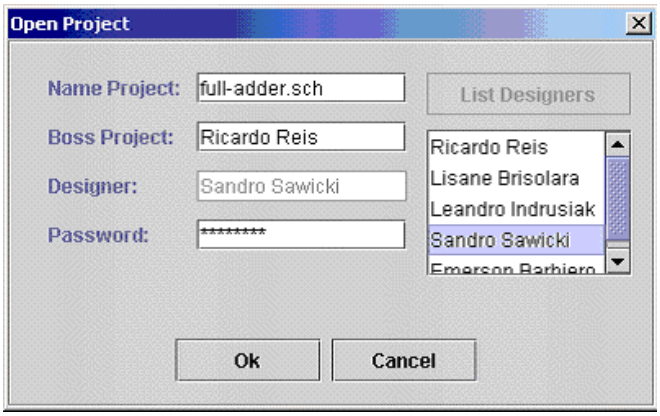


FIGURA 6.39 - Interface para recuperação de um projeto cooperativo

A opção ***Save Project in the Network,*** atualiza um projeto no repositório. Ela somente estará disponível, se o projetista estiver com a permissão de escrita. Caso contrário, este projetista não poderá atualizar os dados de projeto no repositório. É importante ressaltar novamente que todas as atualizações são realizadas dentro de uma transação, a fim de garantir a consistência dos dados de projeto.

A opção ***Chat with Designers*** permite a interação entre os projetistas através de um *chat* de texto. A figura 6.40 mostra a troca de mensagens de texto entre dois projetistas. Como mencionado anteriormente, pesquisas estão sendo realizadas para inserir voz como opção na comunicação entre os projetistas.

A opção ***Request Write Permission*** permite a um projetista requisitar a permissão de escrita. Para que isso aconteça, um método é disparado, e uma mensagem chega até o projetista portador da permissão de escrita que, então, passa a permissão para o projetista que fez a requisição.

A opção ***Search Last Version*** recupera a última versão do projeto e está disponível somente para o projetista que detém a permissão de escrita.

A opção ***Help*** executa um *help on-line*, onde há uma especificação detalhada do funcionamento do serviço de colaboração.

A opção ***Exit Session*** permite que um projetista saia de uma sessão cooperativa. Ao sair da sessão, os demais projetistas são notificados e atualizados. Caso a saída da sessão aconteça com quem detém a permissão de escrita, a permissão é passada ao próximo projetista da lista.

FIGURA 6.40 - Interação entre projetistas via *chat*

A figura 6.41 apresenta a tela do Blade após a inserção do serviço de colaboração. Observa-se a inserção do item de menu *Collaborative work*, da área com a lista de projetistas on-line. A área usada para *chat* só é visível quando o projetista está participando de uma seção de *chat*.

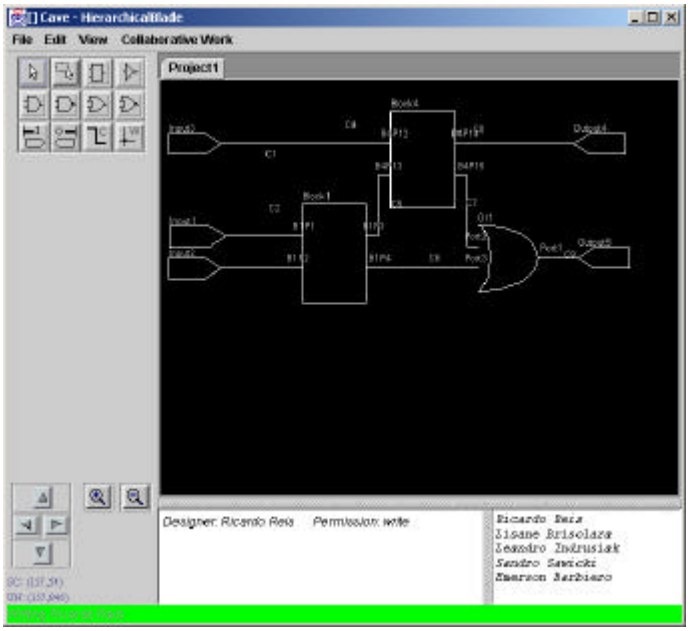
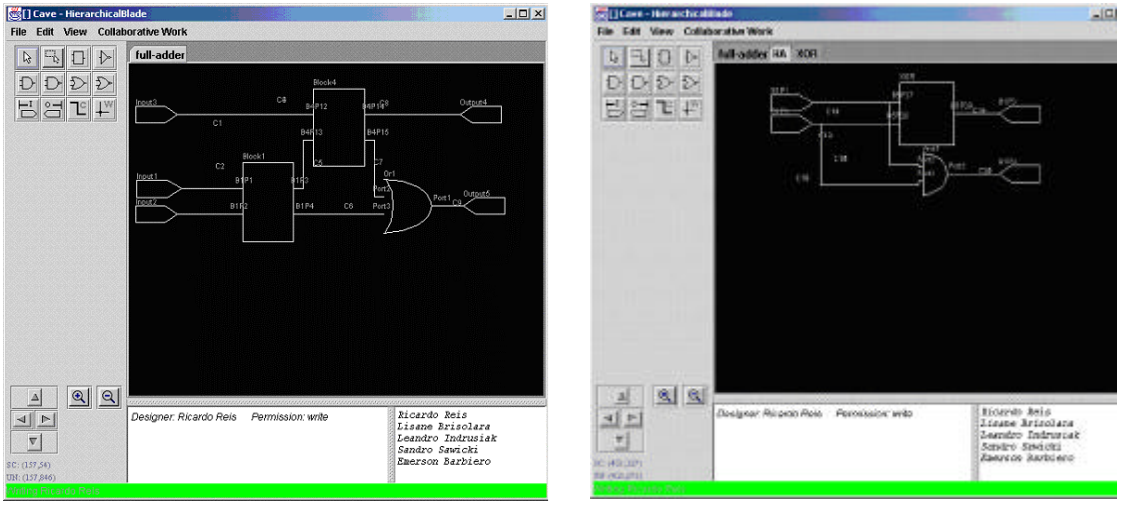


FIGURA 6.41 - Tela do Blade após a integração com o serviço de colaboração

6.9.2 Estudo de Caso: Projeto Colaborativo através do Blade

A fim de demonstrar o uso do serviço de colaboração na ferramenta, apresenta-se nesta seção, um estudo de caso, no qual o projeto de um *full-adder* é utilizado. O *full-adder* é um circuito bastante simples, e portanto, o uso de hierarquia não é um requisito para seu projeto. Porém, neste estudo de caso, utilizou-se uma abordagem hierárquica a fim de permitir a validação do serviço de cooperação junto ao suporte a projetos hierárquicos disponibilizado pelo Blade. Foram utilizados três níveis de hierarquia na especificação do circuito exemplo. O primeiro nível descreve o *full-adder* a partir de dois *half-adder*, o segundo nível descreve o *half-adder* e, por fim, o último nível foi utilizado para a especificação da porta XOR, que é utilizada na especificação do circuito e não faz parte das primitivas básicas disponíveis nesta versão do editor.

Na figura 6.42, seqüência (a) e (b) podem ser visualizados dois dos níveis de hierarquia usados no projeto de um *full-adder*.



(a) (b)

FIGURA 6.42 - Representação dos níveis hierárquicos em um *full-adder*

Nos painéis de cooperação que foram inseridos no Blade, estão ilustrados os projetistas participantes da sessão cooperativa que estão *on-line* e o tipo de permissão que cada um deles têm (painel da esquerda), assim como, também pode ser visualizada a lista de projetistas vinculados ao projeto (painel da direita). Convém ressaltar que na seqüência de figuras acima, um único projetista está trabalhando no projeto e, portanto, ele tem a permissão de escrita, indicada pela cor verde na barra de *status.*

A figura 6.43 ilustra dois projetistas interagindo no projeto, onde ambos estão visualizando o nível mais alto de hierarquia do projeto. É possível perceber que ambos tem a mesma visão do projeto e de seus níveis hierárquicos, o projetista que entrou por

último na sessão cooperativa obteve a permissão de leitura, ilustrada pela cor vermelha na barra de status. Os dois projetistas trocam idéias através do *chat* de texto, que pode ser visualizado na parte inferior da janela do editor.

As figuras 6.43 a 6.45 mostram a interação de dois projetistas usando como exemplo o projeto do *full-adder*, seguindo a metodologia de cooperação adotada, *Pair-programming*. É importante ressaltar que essa interação foi executada em uma única máquina para fins ilustrativos, mas é possível executá-la em diferentes plataformas de hardware e/ou software espalhadas remotamente pela rede.

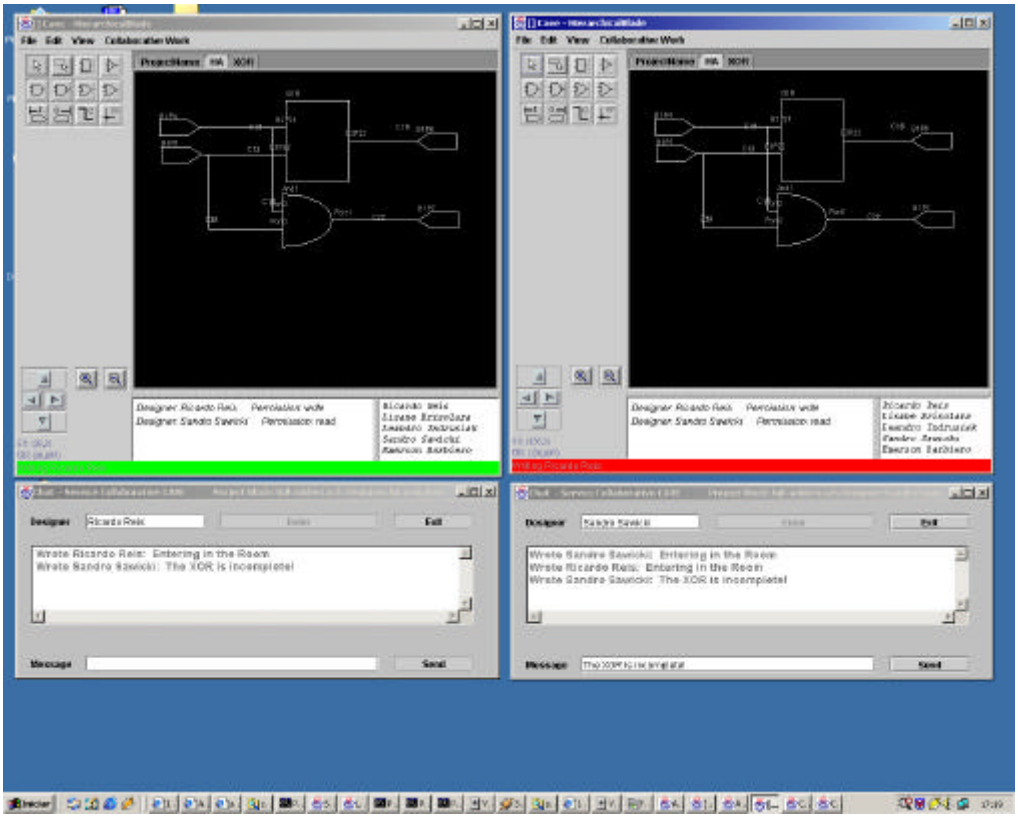


FIGURA 6.43 - *Full-Adder* - Nível 1: Interação entre dois projetistas

Conforme foi destacado anteriormente, este projeto possui três níveis hierárquicos que podem ser acessados através de “abas” na parte superior da área de trabalho do Blade. Na figura 6.43, pode-se observar o nível mais alto de hierarquia, sendo composto por dois blocos que representam meio somadores (*half-adders)* e uma porta lógica OR.

O segundo nível de hierarquia do projeto *full-adder*, descreve o funcionamento de um *half-adder* e pode ser visualizado na figura 6.44. Este circuito é composto por uma porta AND e uma porta lógica EX-OR. A porta EX-OR não está disponível nesta versão do *Blade*. Por isso, a Ex-OR foi representada neste nível de hierarquia por um bloco funcional e depois descrita usando portas lógicas do tipo OR, NOT e AND.

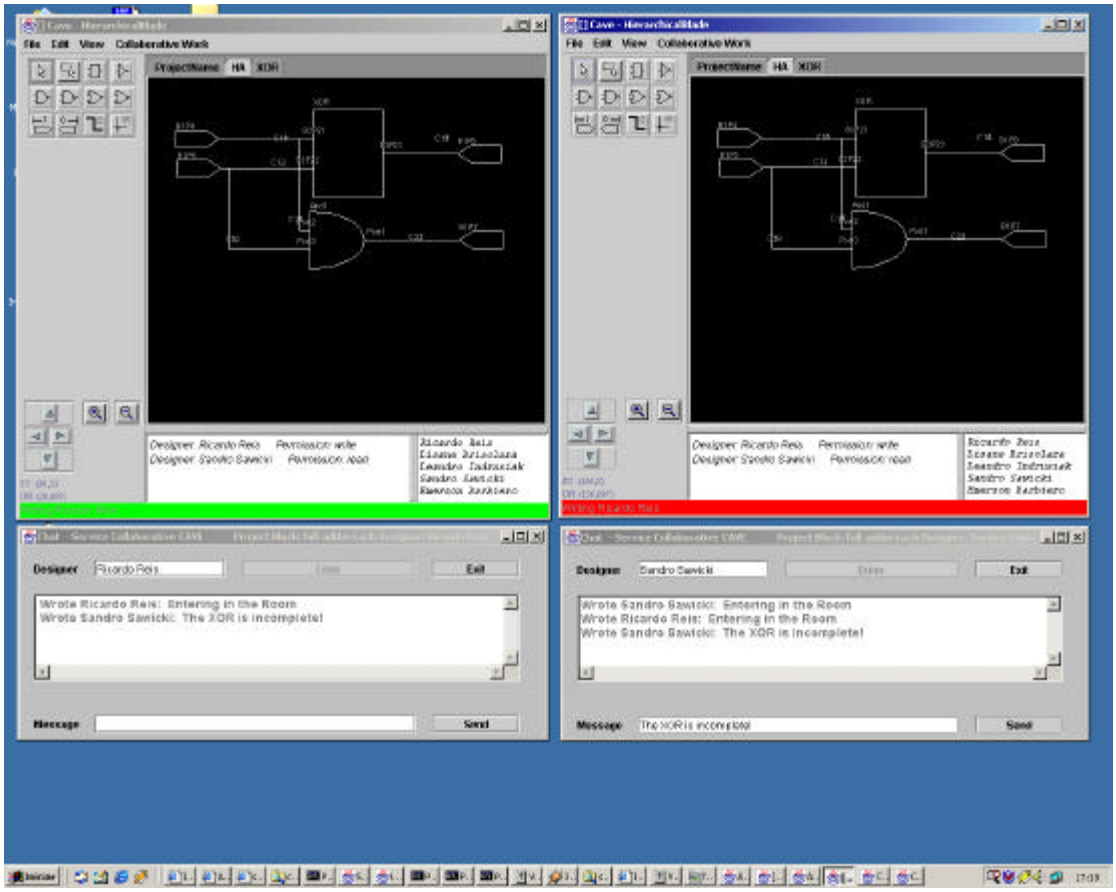


FIGURA 6.44 - Nível 2 – Projeto do *FullAdder*

Quando o projetista escritor realiza uma alteração no projeto, ele pode atualizá-lo no repositório de dados, isso faz com que todo o grupo de projetistas on-line envolvidos no projeto tenham suas visualizações atualizadas, de forma a refletirem as alterações realizadas pelo escritor.

Na metodologia utilizada, somente um projetista pode alterar o projeto. Os demais participantes são considerados leitores (ou ouvintes). Entretanto, caso um participante queira editar o projeto, pode requisitar a escrita ao projetista escritor. A figura 6.45 ilustra a requisição do direito de escrita feita por um projetista leitor para o projetista escritor. Após a autorização do projetista escritor, os direitos dos dois projetistas são alterados. Na interface gráfica do editor, isso pode ser visualizado através da barra de status que indica o direito de escrita do projetista.

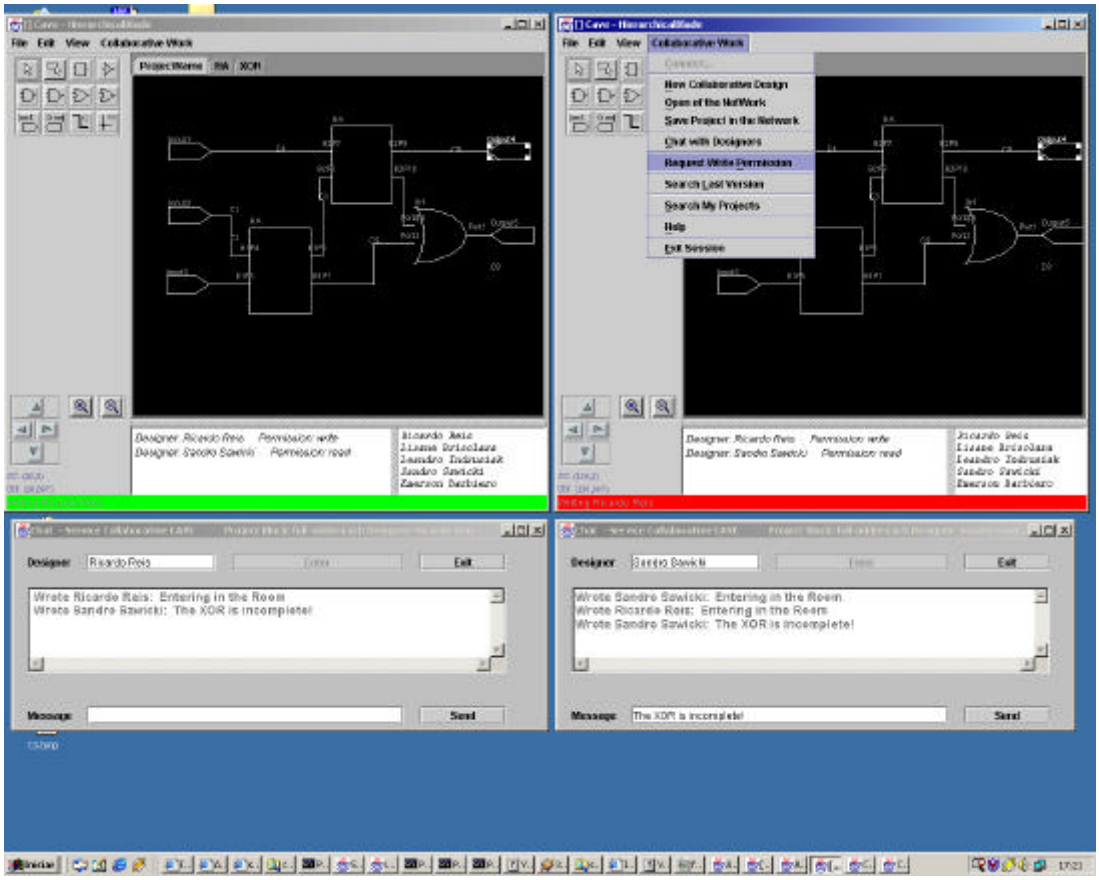


FIGURA 6.45 - Requisitando permissão de escrita

Além, das mudanças na barra de *status* da ferramenta, a lista com os projetistas on-line e suas permissões também devem ser modificadas, sendo que estas alterações devem ser repassadas a todos projetistas participantes da sessão colaborativa. Em outras palavras, todo projetista on-line deve estar ciente de quem está interagindo com ele no momento, e quem é o projetista que detém o direito de escrita. Essas informações podem ser observadas na interface do editor que é visualizada pelos projetistas.

7 Conclusões

Neste trabalho foram estudadas ferramentas para a especificação de projeto de CIs, em específico as ferramentas de edição de esquemáticos. O objetivo principal deste estudo foi a especificação e implementação de uma nova ferramenta gráfica, chamada Blade (*Block and diagram editor*), dedicada à especificação de sistemas.

O Blade foi implementado para ser incorporado ao ambiente Cave, o qual tem por objetivo integrar ferramentas de CAD para circuitos integrados num ambiente distribuído. A linguagem Java é utilizada no projeto Cave e foi utilizada também no desenvolvimento do Blade, principalmente devido à sua independência de plataforma.

Para o desenvolvimento da ferramenta foram estudados conceitos de modelagem orientada a objetos, juntamente com conceitos de desenvolvimento de arquiteturas de software reutilizáveis e uso de padrões de projeto de software. Tudo isso, com objetivo de desenvolver uma ferramenta flexível que possa ser facilmente estendida a outras formas de diagrama e cujo projeto pode ser reutilizado no desenvolvimento de novas ferramentas. O Blade deve ser extensível a novas classes de diagrama, de forma que o mesmo possa ser utilizado para especificação de sistemas em diferentes níveis de abstração.

Atualmente está sendo incorporado um suporte a trabalho colaborativo no ambiente Cave. Neste contexto, além das características de reusabilidade e extensão, na modelagem dos dados do Blade, considerou-se também o suporte à metodologia de colaboração suportada no Cave. Desta forma, facilitou-se a incorporação do serviço de colaboração, disponibilizado no ambiente, ao Blade.

7.1 Contribuições

O Cave disponibiliza uma biblioteca de classes Java, um *framework* no domínio da engenharia de software, que visa facilitar a implementação e incorporação de novas ferramentas ao ambiente. O desenvolvimento do Blade permitiu validar este *framework* de software, bem como estendê-lo para o desenvolvimento de editores de diagramas. As classes da arquitetura de software do Blade foram incorporadas a este *framework*, de modo que elas possam ser reutilizadas para o desenvolvimento de outros editores de diagramas que trabalham com diferentes linguagens visuais. Portanto, este trabalho contribuiu para a construção desse *framework* que está sendo disponibilizado no ambiente Cave e que poderá ser utilizado por desenvolvedores de ferramentas de CAD.

Além do desenvolvimento do editor, o presente trabalho contribui também com um estudo sobre modelagem de primitivas de projeto de sistemas eletrônicos. Os dados manipulados pelo editor de esquemáticos foram modelados usando conceitos da

orientação a objetos e de padrões de projeto de software. Neste modelo, encontram-se representadas relações freqüentemente utilizadas por diferentes ferramentas de CAD, tais como hierarquia de projeto e criação de instâncias. Assim, com a incorporação das classes do Blade a este *framework* disponível no Cave, desenvolvedores de ferramentas podem reutilizar as idéias de modelagem propostas neste trabalho para o desenvolvimento de novas ferramentas.

7.2 Trabalhos Futuros

No capítulo 5, apresenta-se uma especificação da ferramenta Blade, na qual baseou-se a implementação da ferramenta. É importante ressaltar que nem todas as funções descritas na especificação foram implementadas no protótipo da ferramenta e, portanto, serão implementadas futuramente. Dentre elas pode-se destacar a rotação e espelhamento de componentes. Assim como, durante a implementação, observou-se novas funcionalidades que poderiam ser incorporadas à ferramenta Blade, algumas destas serão descritas nas próximas seções.

7.2.1 Integração com outras ferramentas

Como trabalhos futuros pode-se destacar o estudo da integração do Blade com outras ferramentas de CAD dedicadas ao projeto de CIs (ferramentas de síntese, simulação, estimativa de área e de potência, etc). Isto poderia ser feito através do desenvolvimento de um módulo de importação / exportação, que inserido no Blade realizaria a leitura, tradução e adaptação de descrições. Para o desenvolvimento deste módulo, uma análise dos formatos de entrada / saída utilizados pelas ferramentas envolvidas no fluxo de projeto a ser suportado deve ser realizada.

Poderia ser proposta ainda, uma integração mais forte, na qual as estruturas de dados das ferramentas estejam mais acopladas. Neste caso, precisa-se criar um modelo de dados único, genérico, que possa ser utilizado em diferentes ferramentas de CAD. Para fazer isso, deve-se realizar um estudo aprofundado sobre modelagem de primitivas de projeto.

7.2.2 Integração com Homero

Como já foi ressaltado anteriormente, o Blade permite a descrição de um circuito através de um digrama lógico, usando portas lógicas, ou ainda, através de um diagrama de blocos. Os blocos são utilizados na ferramenta para permitir o projeto hierárquico. Ao final da descrição, o editor gera uma descrição VHDL do sistema especificado. Enquanto, o Homero foi desenvolvido por Hernandez [HER 2001] [HER 2001a] e permite a edição de descrições textuais usando linguagens de descrição de hardware.

Com a integração das duas ferramentas, Blade e Homero, pode-se suportar a visualização e edição das descrições VHDL geradas pelo Blade, através da invocação do Homero pela interface do Blade. Desta forma, o projetista pode fazer alterações no diagrama e visualizar os reflexos destas alterações na descrição VHDL do sistema.

7.2.3 Visualização da Planta Baixa do chip

A visualização gráfica do sistema fornecida pelo Blade pode ser utilizada também para gerar uma visualização do *floorplanning* do CI. Isto poderá ser feito através do uso do resultado de estimativa de área dos blocos do sistema fornecido por ferramentas de estimativa específicas. Com base nessas informações de estimativa de área, o Blade pode ser utilizado para a montagem da planta baixa do chip.

7.2.4 Geração de outros formatos *netlist*

Neste primeiro protótipo, o Blade gera descrições *netlist* do circuito usando a linguagem VHDL. Porém, é desejado que a ferramenta possibilite a geração de descrições em mais de um formato padrão. Padrões como EDIF [ELE 2002] e BLIF [WAG 98] devem ser estudados e a geração de descrições nestes formatos deve ser inserida na ferramenta. Isso aumentaria as possibilidades de troca de informações entre o Blade e outras ferramentas, permitindo, por exemplo, que as descrições geradas no Blade possam servir de entrada para um maior número de ferramentas.

7.2.5 Extensão a outras formas de diagrama

Com o aumento da complexidade dos sistemas, cada vez mais estão sendo utilizados níveis de abstração mais altos para a especificação de sistemas, sendo necessário que as ferramentas suportem estes novos níveis de abstração. Nos níveis de abstração mais altos, estão sendo utilizadas representações gráficas para especificação dos sistemas em um nível mais conceitual. Nestes casos, uma ferramenta gráfica pode permitir a montagem de diagramas, bem como a geração de uma descrição num formato padrão que possa servir como entrada para outras ferramentas usadas no projeto. A ferramenta *Ptolemy* [LEE 2002], por exemplo, permite especificações em nível de sistema, através da montagem de diagramas que representam diferentes modelos de computação.

O Blade pode ser estendido para permitir o uso de diagramas de máquinas de estados (FSM) para especificação de sistemas. Segundo Lee [LEE 2002], este é um modelo computacional excelente para expressar lógica de controle e portanto, são utilizados com freqüência na especificação de sistemas. Encontram-se também, trabalhos que apontam a modelagem de sistemas usando a linguagem UML [MAR 2001] [FER 2001] [FER 2000] [WAG 99]. Logo, para suportar modelagem de sistemas sob o paradigma orientado a objetos, o Blade poderia ser estendido de forma a permitir

a edição de diagramas UML, tais como: diagramas de objeto, diagramas de classe, diagrama de seqüência, etc.

Bibliografia

[BAR 94] BARROS, Ligia A. **Suporte a Ambientes Distribuídos para Aprendizagem Cooperativa**. 1994. Tese (Doutorado em Ciência da Computação) - COPPE/UFRJ, Rio de Janeiro.

[BEC 94] BECK, K.; CUNNINGHAM, W. Patterns and Software Development. Dr. **Dobbs Journal**, [S.l], Feb. 1994.

[BEC 94a] BECK, K.; JOHNSON R. Patterns Generate Architectures. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, ECOOP, 8., Bologna, Italia**. Object-Oriented Programming**. Berlin: Springer Verlag, 1994.

[BEC 99] BECK, K. **Extreme Programming Explained**. Reading: Addison Wesley, 1999.

[BEN 96] BENINI, L.; BOGLIOLO, A.; DE MICHELI, G. Distributed EDA tool integration: the PPP paradigm, In: IEEE INTERNATIONAL CONFERENCE ON COMPUTER DESIGN: VLSI IN COMPUTERS AND PROCESSOR, ICCD, 1996, Austin. **Proceedings...** Los Alamitos: IEEE Computer Society, 1996. p. 448-453.

[BOR 95] BORGES, M. R. da S. **Suporte por Computador ao Trabalho Cooperativo.** Porto Alegre: Instituto de Informática da UFRGS, 1995. 45p. Curso Ministrado na 14. Jornada de Atualização em Informática.

[BRG 2001] BRGLEZ, F.; LAVANA, H. A Universal Client for Distributed Networked Design and Computing. In: DESIGN AUTOMATION CONFERENCE, DAC, 38., 2001, Las Vegas. **Proceedings...** New York: ACM, 2001.

[BRO 2000] BROWN, S.; VRANESIC, Z. **Fundamentals of Digital Logic with VHDL Design.** Boston: McGraw-Hill, 2000.

[CAM 96] CAMPIONE, M.; WALRATH, K. **The Java Tutorial:** object-oriented programming for the internet. [S.l.]: SunSoft Press, 1996.

[CAR 97] CARRO, L. et al. Ambiente ÁGATA de projeto Versão Beta 2.0. In: WORKSHOP IBERCHIP, 3., 1997, México. **Anais...** México: Departamento de Ingineria Electrica, 1997. p. 497-503.

[CAY 2001] CAYRES, P. **Editor Distribuído de Manipulação Colaborativa de Documentos Diagramáticos**. 2001. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

[COA 95] COAD, P. **Object Models:** strategies, patterns and applications. Englewood Cliffs, NJ.: Prentice-Hall, 1995.

[COA 93] COAD, P.; YOURDON, E. **Projeto Baseado em Objetos.** Rio de Janeiro:

Campus, 1993. 195p.

[COO 98] COOPER, J. W. **Design Patterns – Java Companion**. [S.l.]: Addison-Wesley, 1998. Design Patterns Series.

[DEU 89] DEUTSCH, P. Frameworks and reuse in the Smalltalk 80 System. In: BIGGERSTAFF, T.; PERLIS, A. J. **Software reusability:** application and experience. New York: ACM Press, 1989. v.2, p. 57-71.

[DIE 96] DIETRICH, E. **Projeto de um Sistema de Suporte a Autoria Cooperativa de Hiperdocumentos**. 1996. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

[ELE 2002] ELECTRONIC INDUSTRIES ALLIANCE. **Electronic Design Interchange Format.** Disponível em: <http://www.edif.org>. Acesso em: maio 2002.

[FER 2001] FERNANDES, J. M.; MACHADO, R. J. System-Level Object-Orientation in the Specification and Validation of Embedded Systems. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, SBCCI, 14., 2001, Pirenópolis. **Proceedings...** Los Alamitos: IEEE Computer Society, 2001. p. 8-13.

[FER 2000] FERNANDES, J. M.; MACHADO, R. J.; SANTOS, H. Modeling Industrial Embedded Systems with UML. In: INTERNATIONAL SYMPOSIUM ON HARDWARE / SOFTWARE CODESIGN, CODES, 8., 2000, San Diego. **Proceedings…** USA: ACM Press, 2000.

[FER 98] FERNANDES, J. H. C. CIBERESPAÇO: modelos, tecnologias, aplicações perspectivas. In: JORNADA DE ATUALIZAÇAO EM INFORMÁTICA, JAI, 17., Belo Horizonte. **Anais...** Belo Horizonte: SBC, 1998.

[FRE 99] FREEMAN, E.; HUPFER, S.; ARNOLD, K. **JavaSpaces:** principles, patterns, and practice. Reading: Addison Wesley, 1999.

[FUR 98] FURLAN, J. D. **Modelagem de Objetos através de UML – the Unified Modeling Language**. São Paulo: Makron Books, 1998. 329p.

[GAM 95] GAMMA, E. et al. **Design Patterns:** elements of reusable object-oriented software. Reading: Addsion Wesley, 1995.

[GAM 93] GAMMA, E. et al. Design Patterns: abstraction and reuse of object-oriented design. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, ECOOP, 7., Kaiserslautern. **Object-Oriented Programming**. Berlin: Springer Verlag, 1993. (Lecture Notes in Computer Science, v. 707).

[GAM 2000] GAMMA, E. et al. **Padrões de Projeto:** soluções reutilizáveis de software orientado a objetos. Tradução Luiz M. Salgado. Porto Alegre: Bookman, 2000.

[GER 99] GEREZ, S. H. **Algorithms For VLSI Design Automation**. Chichester:

John Wiley & Sons, 1999.

[GET 2000] GETTING Started with the XILINX Foundation F2.1i Tools. 2000. Disponível em: <http://www.seas.upenn.edu/~ee201/foundation/foundation_intro.html>. Acesso em: 23 mar. 2001.

[GLE 88] GLEN, E. K.; STEPHEN, T. P. A Cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. **Journal of Oriented-Object Programming**, [S.l.], v. 1, n. 3, p. 26-49, Aug. / Sept. 1988.

[GOL 95] GOLDBERG, A.; RUBIN, K. **Succeeding with Objects**: decision frameworks for project management. Reading: Addison-Wesley, 1995.

[GRA 99] GRALEWSKI, D.; WINCKLER, M. A. A.; INDRUSIAK, L. S.; REIS, R. A. L. JALE Layout Editor. In: UFRGS MICROELECTRONICS SEMINAR, SIM, 14., 1999, Pelotas. **Anais...** Porto Alegre: SBC, 1999. p. 51-53.

[HER 2001] HERNADEZ, E. B. **Homero:** editor de descrições textuais para trabalho colaborativo. 2001. 39f. Projeto de Diplomação (Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

[HER 2001a] HERNANDEZ, E. B.; SAWICKI, S.; INDRUSIAK, L. S.; REIS, R. A. da L. Homero: um editor VHDL cooperativo via web. Poster apresentado no 7. Workshop IBERCHIP, 2001, Montevideo.

[HOL 94] HOLLINSWORTH, David; WHARTON, Peter. An Architecture for Developing CSCW. In: SPURR, K. et al. **Computer Support for Co-Operative Work**. Chichester, EUA: John Wiley, 1994.

[HOR 2001] HORSTMANN, C.; CORNELL, G. **Core Java 2**. São Paulo: Makron Books, 2001. v.1.

[IND 2002] INDRUSIAK, L. S.; GLESNER, M.; REIS, R. A. da L. Comparative Analysis and Application of Data Repository Infrastructure for Collaboration-enabled Distributed Design Environments. In: DESIGN AUTOMATION AND TEST IN EUROPE, DATE, 39., 2002, Paris. **Proceedings...** Los Alamitos: IEEE Computer Society, 2002.

[IND 2002a] INDRUSIAK, L. S. **A Review on the Framework Technology Supporting Collaborative Design of Integrated Systems.** 2002. Exame de Qualificação (Doutorado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

[IND 2001] INDRUSIAK, L. S.; BECKER, J.; GLESNER, M.; REIS, R. A. da L. Distributed collaborative design over Cave2 framework. In: IFIP INTERNATIONAL CONFERENCE ON VERY LARGE SCALE INTEGRATION, VLSI, 11., 2001, Montpellier. **Proceedings...** Montpellier: LIRMM, 2001.

[IND 98] INDRUSIAK, L. S.; REIS, R. A. da L. A Case Study for a WWW Based

CAD Framework – Cave Project. In: BRAZILIAN SYMPOSIUM ON INTEGRATED CIRCUITS DESIGN, SBCCI, 11., 1998, Armação de Búzios. **Proceedings...** Los Alamitos: IEEE Computer Society, 1998. p. 31-36.

[IND 98a] INDRUSIAK, Leandro S.; REIS, Ricardo A. L. **Ambiente de Apoio ao projeto de Circuitos Integrados baseado na *World Wide Web***. 1998. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

[IND 97] INDRUSIAK, L. S.; REIS, R. A. da L. A WWW approach for EDA Tool Integration. In: BRAZILIAN SYMPOSIUM ON INTEGRATED CIRCUITS DESIGN, SBCCI, 10., 1997, Gramado. **Proceedings...** Los Alamitos: IEEE Computer Society, 1997.

[JOH 97] JOHNSON, R. E. **Components, frameworks, patterns**. Feb. 1997. Disponível em: <ftp://st.cs.uiuc.edu>. Acesso em: 15 abr. 2001.

[JOH 92] JOHNSON. R. Documenting Frameworks Using Patterns. In: CONFERENCE ON OBJECT-ORIENTED PROGRAMMING: Language and Application, ECOOP, 8., 1992, Washington DC. **Tutorial Notes...** [S.l.: s.n.], 1992.

[KIR 2001] KIROWSKI, D. R.; POTKONJAK M.; DRINIC, M. Hypermedia Aided Design. In: DESIGN AUTOMATION CONFERENCE, DAC, 38., 2001, Las Vegas. **Proceedings...** New York: ACM, 2001.

[KRA 97] KRAMER, D. **JDK Documentation**. [S.l.]: Sun Microsystems, 1997.

[KUR 97] KURUP, P.; ABBAS, T. **Logic Synthesis Using Synopsys.** 2 nd ed. Boston: Kluwer Publishers, 1997.

[LAR 99] [LAR 99] LARMAN, Craig. **Applying UML and patterns – An Introduction to Object-Oriented Analysis and Design**. [S. l.]: Prentice Hall, 1999.

[LAU 90] LAUREL, B. **The Art of Human-Computer Interface Design**. [S.l.]: Addison-Wesley, 1990.

[LAV 97] LAVANA, H. et al. Executable workshops: A Paradigm for Collaborative Design on the Internet. In: DESIGN AUTOMATION CONFERENCE, DAC, 34., 1997, CA, USA. **Proceedings...** New York: ACM, 1997.

[LEE 2002] LEE, E. A. et al. **Ptolemy II heterogeneous Concurrent Modeling and Design in Java**. Berkeley: UC Berkeley EE, 2002. (Technical Memorandum UCB/ERL M02/23).

[MAR 2001] MARTIN, G.; LAVAGNO, L.; GUERIN, J. L. Embedded UML: a merger of real-time UML and co-design. In: INTERNATIONAL SYMPOSIUM ON HARDWARE / SOFTWARE CODESIGN, CODES, 2001, Copenhagen. **Proceedings…** Los Alamitos: IEEE Computer Society Press,

2001.

[MAR 95] MARTIN, J.; ODELL, J. J. **Análise e Projeto Orientados a Objeto**. São Paulo: Makron Books, 1995.

[MAX 97] MAX+PLUS II Getting Started. Feb. 1997. Disponível em: <http://www.altera.com/support/software/sof-maxplus2.html>. Acesso em: 20 set. 2000.

[MEN 2002] MENTOR GRAPHICS. **Design Architecture DataSheet**. Disponível em: <http://www.mentor.com/designarchitect/datasheet>. May 2002. Acesso em: 20 ago. 2002.

[MIN 95] MINAS, M.; VIEHSTAEDT, G. *DiaGen:* A Generator for Diagram Editors Providing Direct Manipulation and Execution of Diagrams. In: IEEE SYMPOSIUM ON VISUAL LANGUAGE, 11., 1995, Darmstadt. **Proceedings...** Los Alamitos: IEEE Computer Society, 1995.

[MOR 90] MORAES, F. G.; OSÓRIO, F. S.; SUZIM, A.; BARONE, D. A. C. Esqueleto: Editor de Esquemas Elétricos. In: CONFERÊNCIA LATINO AMERICANA DE INFORMÁTICA, CLEI, 16., 1990, Assuncion. **Anais...** CLEI, 1990. v. 2, p. 589-605.

[NEU 90] NEUWIRTH, C. M. et al. Issues in the Design of Computer-Support for Co-Authoring and Commenting. In: CONFERENCE OF COMPUTER SUPPORTED COOPERATIVE WORK, CSCW, 3., 1990. **Proceedings...** Los Angeles: ACM, 1990.

[NEW 96] NEWTON, A. R. et al. **WELD Project - Web-Based Electronic Design**. Berkeley: U.C. Berkeley, 1996. Disponível em: <http://www-cad.eecs.berkeley.edu/Respep/Research/weld/>. Acesso em: 20 set. 2001.

[OST 2001] OST, L.C. et al. Jale3D - Platform-independent IC/MEMS Layout Edition Tool. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, SBCCI, 14., 2001, Pirenopolis. **Proceedings...** Los Alamitos: IEEE Computer Society Press, 2001.

[PIN 99] PINHEIRO, M. K. **Edição cooperativa de hiperdocumentos na WWW**. 1999. 53f. Trabalho Individual – Instituto de Informática, UFRGS, Porto Alegre.

[PIN 99a] PINHEIRO, M. K. Mecanismo de monitoramento e contextualização em ambientes cooperativos. In: SIMPÓSIO NACIONAL DE INFORMÁTICA, 4., 1999, Santa Maria. **Anais...** Santa Maria : Multipress, 1999. p. 46-47.

[PRE 95] PREE, Wolfang. **Design Patterns for Object – Oriented Software Development**. Reading: Addison-Wesley, 1995.

[PRE 94] PREE, W. Meta-Patterns: abstracting the essentials of object-oriented frameworks. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, ECOOP, 8., 1994, Bolonga, Italia. **Object-Oriented**

Programming. Berlin: Springer Verlag, 1994. p.150-164.

[PRE 96] PRESSMAN, R. S. **Software Engineering:** a practitioner's approach. [S. l.]: McGraw-Hill, 1996.

[QUI 2002] QUICK Start Guide For Quartus II Software Version 2.2 Manual. 2002. Disponível em: <http://www.altera.com/literature/lit-qts.html>. Acesso em: 03 dez. 2002.

[RAM 95] RAMMING, F. J.; WAGNER, F. **Electronic Design Automation Frameworks Volume 4**. [S. l.]: Chapman & Hall, 1995.

[RAT 2001] RATIONAL SOFTWARE CORPORATION. **Unified Modeling Language.** Notation Guide. Version 1.0. Santa Clara, 1997. Disponível em: <http://www.rational.com>. Acesso em: 17 set. 2001.

[REE 98] REECKIE, J.; SCHILMAN, M. **Diva:** Dynamic, Interactive Visualization. 1998. White-paper. Disponível em: http://www.gigascale.org/diva/>. Acesso em: nov. 2001.

[REI 2000] REIS, R. A. da L.; REIS, A. I. Ferramentas de CAD. In: REIS, R. A. da L. (Org.). **Concepção de Circuitos Integrados**. Porto Alegre: Instituto de Informática da UFRGS: Sagra Luzzato, 2000. p. 99-118.

[REI 99] REINHARDT, K. F. **JASE:** Um editor de esquemáticos para World Wide Web. 1999. Projeto de Conclusão de Curso (Bacharelado em Informática) – UFPel, Pelotas.

[REI 99a] REIS, J. P. **JASE II:** um editor de esquemáticos para World Wide Web. 1999. Projeto de Diplomação (Bacharelado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

[REI 83] REIS, R. A. da L. **TESS Evaluater Automatique des Plans de Masse de Circuits VLSI**. 1983. Tese (Doutorado) - Institute National Polytechnique, Grenoble.

[RUM 94] RUMBAUGH J. et al. **Modelagem e Projeto baseado em objetos**. Rio de Janeiro: Campus, 1994.

[SIE 94] SIEMIENIUCH, C. E.; SINCLAIR, M. A. Concurrent Engineering and CSCW: the human factor. In: SPURR, K. et al. **Computer Support for Co-Operative Work**. Chichester, EUA: John Wiley, 1994.

[SAI 95] SAIT, Sadiq M.; YOUSSEF, Habib. **VLSI Physical design Automation:** theory and practice. [S. l.]: McGraw Hill, 1995.

[SAW 2002] SAWICKI, S.; INDRUSIAK, L. S.; REIS, R. A. da L. Projeto Cooperativo no Ambiente Cave. In: WORKSHOP IBERCHIP, 8., 2002, Guadalajara, México. **Proceedings...** Guadalajara:[s.n.], 2002.

[SAW 2002a] SAWICKI, S.; BRISOLARA, L.; INDRUSIAK, L. S.; REIS, R. A. da L. Collaborative Design using a Shared Object Spaces Infrastructure. In:

SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, SBCCI, 15., Porto Alegre. **Proceedings...** Los Alamitos: IEEE Computer Society, 2002.

[SER 95] SERRANO, J. A. The Use of Semantic Constraints on Diagram Editors. In: INTERNATIONAL IEEE SYMPOSIUM ON VISUAL LANGUAGES, 11., 1995, Darmstadt. **Proceedings...** Los Alamitos: IEEE Computer Society Press, 1995.

[SIL 2000] SILVA, R. P. **Suporte ao desenvolvimento e uso de *frameworks* e componentes**. Porto Alegre: PPGC da UFRGS, 2000.

[SIL 96] SILVA, R. P. **Avaliação de metodologias de análise e projeto orientados a objetos voltados ao desenvolvimento de aplicações, sob a ótica de sua utilização no desenvolvimento de *frameworks* orientados a objetos**. 1996. Trabalho Individual – Instituto de Informática, UFRGS, Porto Alegre.

[SOU 97] SOUZA, C. R. B.; WAINWE, J.; RUBIRA, C. M. F. Um modelo de Anotações para o Desenvolvimento Cooperativo de Software. In: WORKSHOP ON HYPERMEDIA E MULTIMEDIA APPLICATIONS, 3., 1997, São Carlos. **Anais...** São Carlos: [s.n.], 1997. p. 143-154.

[SOU 96] SOUZA, A. S. de; GOLENDZINER, L. G. **Um Estudo Sobre Trabalho Cooperativo Suportado por Computador (CSCW)**. 1996. Trabalho Individual (Mestrado em Ciência da Computação) - Instituto de Informática, UFRGS, Porto Alegre.

[SUZ 89] SUZIM, A. A. **Sistemas Digitais:** uma visão integrada do processo de síntese. Porto Alegre: Departamento de Eletricidade da UFRGS, 1989.

[THI 90] THIMBLEBY, H. **User interface Design.** [S.l.]: Addison-Wesley, 1990.

[UNA 91] UNAMAKER, J.F. Electronic meeting systems to support group work. **Communications of the ACM**, New York, v. 34, n. 7, p. 40-61, July 1991.

[VIR 2001] VIRTUOSO Schematic Composer Datasheet. 2001. Disponível em: <http://www.cadence.com/datasheets/virtuoso_composer.html> Acesso em: 03 abr. 2002.

[WAG 99] WAGNER, F. et al. M. Object-Oriented Modeling and Co-Simulation of Embedded Systems. In: IFIP INTERNATIONAL CONFERENCE ON VLSI, 1999, Lisboa. **Proceedings…** Los Alamitos: IEEE Computer Society Press, 1999.

[WAG 98] WAGNER, A. **Berkeley Logic Interchange Format (BLIF)**. Berkeley: University of California, 1998. Disponível em: <http://www.bdd-portal.org/docu/blif/>. Acesso em 26 nov. 2000.

[WAG 94] WAGNER, F. R. **Ambientes de Projeto de Sistemas Eletrônicos**. Recife: UFPE, 1994. 155p. Curso ministrado na 9. Escola de Computação, 1994, Recife.

[WIL 2000] WILLIAMS, L.; KESSLER, R.R. All I Really Need to Know about Pair Programming I Learned In Kindergarten. **Communications of the ACM**, New York, v. 43, n. 5, p. 108-114, 2000.

[WIL 94] WILLIANS, N. et al. The Impact of Distributed Multimedia systems on CSCW. In: SPURR, K. et al. **Computer Support for Co-Operative Work**. Chichester, EUA: John Wiley, 1994.