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

ÉMERSON BARBIERO HERNANDEZ

Macanudo: uma Abordagem Baseada em Componentes Voltada a Reuso de Projetos de Hardware

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação

Prof. Dr. Ricardo Augusto da Luz Reis
Orientador

Porto Alegre, dezembro de 2005.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Hernandez, Émerson Barbiero

Macanudo: uma Abordagem Baseada em Componentes Voltada a Reuso de Projetos de Hardware / Émerson Barbiero Hernandez – Porto Alegre: Programa de Pós-Graduação em Computação, 2005.

91 f.:il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação. Porto Alegre, BR – RS, 2005. Orientador: Prof. Dr. Ricardo Augusto da Luz Reis

1. CAD. 2. CBSE. 3. IDE. 4. Reuso. 5. VHDL. I. Reis, Ricardo Augusto da Luz. II. Título.

AGRADECIMENTOS

Conscientemente ou não, algumas pessoas tiveram participação na construção do trabalho a seguir. Indiferentemente de qual tenha sido a contribuição, faço desse momento um evento de rememoração e reverência. Para isso, tomo como base algumas das músicas que compõe um CD que no decorrer desses anos de mestrado, me acompanhou em diferentes momentos.

A meus pais, ofereço a primeira música: "O Bêbado e a Equilibrista".

A todos amigos que fiz na Universidade Federal, independente de sua colocação acadêmica, ofereço "Somos Todos Iguais Nessa Noite", pois se olharmos com atenção, existe realmente um elo que nos faz soar em uníssono. E há sempre um bar para ser freqüentado...

Aos Camagadas Caco, Márcio, Leomar, Gervini, Julius, Fernando, Rodrigo, Edgard, Felipe e Juan, mando "Encontros e Despedidas", lembrando que mesmo que o trem não tenha parado, as pessoas continuam se encontrando nas estações.

Aos professores Weber e Pimenta, por terem me mostrado que em nós sempre há algo a ser acrescentado, ofereço "Faltando um Pedaço", agradecendo por terem me influenciado positivamente a ler Tolkien e Borges.

Aos meus amigos do "Expresso 25", que souberam ouvir e despertar algumas das mais interessantes notas, envio "Minha Voz, Minha Vida".

E a você, que está lendo este trabalho, fecho essa pequena homenagem com "Redescobrir", na esperança de que em sua leitura, você acabe redescobrindo um pouco de todos que contribuíram, assim como um pouco mais sobre você mesmo e suas buscas.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

3C – Conceito, Conteúdo e Contexto.
AMI – Asynchronous Method Invocation
API – Application Program Interface
BDK – Bean Development Kit
CASE – Computer-Aided Software Engineering
CBD – Component Based Development
CBSE – Component Based Software Engineering
CI – Circuito Integrado
COM – Component Object Model
CORBA – Common Object Request Broker Architecture
COTS – Commercial off-the-Shelf
DCE – Distributed Computing Environment
DCOM – Distributed Component Object Model
DDE – Dynamic Data Exchange
DTD – Document Type Definition
EDA – Eletronic Design Automation
EJB – Enterprise Java Beans
GUI – Graphic User Interface
I18N – Internationalization
IDE – Integrated Development Environment
IDL – Interface Description Language
IEEE – Institute of Electrical & Electronic Engineers
IIOP – Internet Inter-ORB Protocol
IP – Intelectual Property
MDTC – Microsoft Distributed Transaction Coordinator
MSMQ – Microsoft Message Queuing
MTS – Microsoft Transaction Server

MYA – Model of the Year Architecture

OLE – Object Linking and Embedding.

OMG – Object Management Group

OO – Orientação a Objetos

ORB – Object Request Broker

OSF – Open Software Foundation

OSTG – Open Source Technology Group

OVHDL – Objective VHDL

PLI – Programming Language Interface

RAD – Rapid Application Development

RASSP – Rapid Prototyping Of Application Specific Signal Processors

RMI – Remote Method Invocation

RPC – Remote Procedure Call

RPD – Rapid Program Development

RSD – Rapid System Development

RTL – Register Transfer Level

Si2 – Silicon Integration Initiative

SLB – System-Level Blocks

SLM – System-Level Macros

SWT – Standard Widget Toolkit

VB – Visual Basic

VHDL – VHSIC Hardware Description Language

VHSIC – Very High Speed Integrated Circuit

VSI – Virtual Socket Interface

WSDL – Web Services Description Language

XML – Extensible Markup Language

XP – Extreme Programming

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Como as tecnologias de CI avançam através de melhoras de desempenho, maiores são as densidades e a complexidade de projetos. Esse avanço cria a necessidade de novas ferramentas CAD e metodologias de projeto para lidar com um ritmo aceitável de desenvolvimento. Inúmeras soluções foram propostas, entre elas está a utilização de conceitos de reuso, metodologia abordada nesse trabalho. O projeto reusável vem do pensamento de que funcionalidades realizadas são muitas vezes similares ou até mesmo idênticas em inúmeras aplicações. Circuitos como somadores e multiplicadores são exemplos de blocos comuns utilizados em diferentes soluções.

Este trabalho apresenta uma abordagem baseada em componentes para descrições de *hardware*, com o objetivo de maximizar reuso, através de composição e montagem gráfica de componentes. Para lidar com esse paradigma, é apresentado um processo chamado Macanudo. Mesmo que linguagens de descrição de hardware tenham ajudado os processos de projeto a alcançar reusabilidade, essa abordagem tem o objetivo de trazer uma forma mais eficiente de guiar o processo a esse resultado.

Esta abordagem é composta por um modelo de componentes que descreve como cada uma dessas entidades deve ser criada e suas interações, assim como sua evolução e distribuição. Juntamente a esse modelo, foi criada um ambiente de desenvolvimento integrado para dar suporte a esse paradigma. Esse programa trabalha com o conceito de projetos, pois a evolução do mesmo traz intrinsecamente uma forma de usabilidade de grande importância. Isso se deve ao fato de no decorrer do tempo, grupos de componentes serem melhorados e adaptados para satisfazer novos requisitos. Assim, o ambiente permite que um determinado componente seja modificado em seu interior para satisfazer necessidades específicas como desempenho, por exemplo.

O ambiente dá suporte a construção gráfica de componentes, usando como base a idéia de estrutura comumente encontrada em sistemas eletrônicos: conjuntos de entidades que se conectam. Cada uma dessas entidades pode igualmente ser formada de blocos interconectados ou de apenas um bloco básico de projeto, reutilizados em diferentes níveis de abstração e hierarquia. Alguns circuitos foram montados através desse processo, gerando novos componentes e códigos VHDL, sendo possível sua integração no ambiente e em outros projetos externos.

Palavras-Chave: CAD, CBSE, IDE, Reuso, VHDL

Macanudo: A Component-Based Approach to Reuse in Hardware Designs

ABSTRACT

As integrated circuit technologies advance towards higher performance, greater densities and increasing system complexity, CAD tools and design methodologies struggle to keep pace. Many solutions have been proposed, and one of them is the concept of reuse, which is the adopted methodology in this work. The reusable design comes from the idea that almost duplicated or even equal functionalities appear in several applications. Circuits such as adders and multipliers are examples of common building blocks needed in many different applications.

This work presents a component-based approach in hardware descriptions, with the main goal of maximizing reuse, through graphical building and assembly of components. To deal with this paradigm, it is presented a process called Macanudo. Even with hardware description languages leading to a major leap in design reusability, this new approach has the objective of bringing an efficient way to guide the process to a better result.

A component model describing how these entities must be created and assembled, as well as its evolution and distribution, composes this process. The model is followed with an IDE that supports this paradigm. The software deals with the concept of designs, because its evolution provides yet another form of a useful functionality, with great importance. This happens because over the time groups of products are updated and adapted to satisfy new requirements, such as performance.

The environment supports graphical component building, using the idea based upon an usual structure found in electronics systems: a set of entities that connect to each other. Interconnected blocks or a simple design block, reused in different levels of abstraction and hierarchy, can equally form each of these entities. Initial circuits have been assembled through this approach, generating new components and VHDL code, making its integration under the environment and others external designs possible.

Keywords: CAD, CBSE, IDE, Reuse, VHDL

1 INTRODUÇÃO

No final dos anos 90, projetistas dos meios empresarial e acadêmico começaram a observar que as tecnologias orientadas a objetos não eram suficientes para acompanhar as rápidas mudanças de requisitos de sistemas de *software*. O desenvolvimento orientado a objetos não tinha conduzido a um extensivo reuso, como originalmente era esperado. Uma das razões era a falta de produção de arquiteturas de software capazes de serem adaptadas aos requisitos que se modificavam. As metodologias orientadas a objetos não guiavam os projetistas a fazer uma clara separação entre os aspectos computacionais e os composicionais. Mesmo que as construções orientadas a objetos encorajassem o desenvolvimento de modelos que refletiam os objetos de domínio do problema.

Antes mesmo do surgimento da orientação a objetos, McIlroy (1968) propôs uma visão de blocos de construção de software. Eles seriam como famílias de rotinas baseadas em princípios bem definidos, propiciando ligação entre esses blocos, que foram chamados componentes. Uma das principais motivações por trás da tecnologia de componentes é a utilização do conceito de reuso. A independência de cada uma das partes é também uma característica fundamental, assim como sua evolução, flexibilidade, adaptabilidade e manutenibilidade. O projeto visando reuso exige do projetista a aplicação de conceitos e princípios de construção de software, levando em conta as características do domínio de aplicação.

Existe um compromisso entre a facilidade de reuso e a de uso de um componente. Tornar um bloco de software reutilizável implica fornecer um modelo de acesso ao componente genérico o suficiente para apresentar diferentes maneiras pelas quais o bloco possa ser acessado e utilizado. Criar um componente reutilizável implica também em fornecer uma interface fácil de compreender. A facilidade de reuso aumenta a complexidade e, desse modo, reduz a compreensão do componente, dificultando aos engenheiros a decisão de quando e como reutilizar esse componente. Fica a cargo dos projetistas encontrar o equilíbrio entre a generalidade e a facilidade de compreensão.

O objetivo da Engenharia de Software Baseada em Componentes, (*Component Based Software Engineering*, CBSE) é pegar elementos de uma coleção de componentes reusáveis de software e construir aplicações apenas fazendo as ligações necessárias entre esses componentes. Adicionalmente, visa à produção de sistemas de maior qualidade em ciclos de desenvolvimento menores e menos custosos. Reconfigurar componentes, adaptar componentes existentes e introduzir novos componentes são as atividades básicas do processo que tem o intuito de adaptar-se ao problema de requisitos

que se modificam, de uma maneira mais eficiente que a abordagem orientada a objetos. (NIERSTRASZ; DAMI, 1995)

Existem dois fatores fundamentais para a correta ligação entre componentes:

- A interface de cada componente deve satisfazer exatamente as expectativas do outro componente;
- Os contratos entre os componentes devem ser explícitos e bem definidos.

Portanto, o desenvolvimento de aplicações baseadas em componentes depende da aderência a interfaces restritas e padronização de protocolos de interação. Em orientação a objetos, o projeto nos leva a interfaces ricas, mas a protocolos de interação que não são padronizados. (NIERSTRASZ; DAMI, 1995) Dado o código fonte de uma aplicação orientada a objetos, a identificação dos componentes é trivial, entretanto, é difícil interpretar a composição do sistema. Essa análise deve-se ao fato de orientação a objetos expor a hierarquia de classes em detrimento à interação entre os objetos. Adicionalmente, a maneira com que os objetos são interconectados está distribuída entre esses objetos, o que esconde a separação entre os aspectos composicionais e computacionais, necessários para o desenvolvimento baseado em componentes (*Component Based Development*, CBD).

1.1 Reuso

O principal propósito de componentes de software é o reuso. Os dois principais tipos de reuso de software são caixa preta e caixa branca. Reuso caixa branca é aquele onde o código do componente é completamente disponível e pode ser estudado, adaptado e modificado. Esse tipo de reuso é o fator chave de *frameworks* orientados a objetos que se baseiam em herança entre as classes, onde desenvolvedores derivam novas subclasses e sobrecarregam métodos. (GAMMA et al., 2000) (JUNG et al., 2000) O problema do reuso caixa branca é que os usuários desses componentes dependem de seu interior e serão diretamente afetados quando das mudanças de seu estado interno. (HUIZING, 1999)

O reuso caixa preta é baseado nos princípios de acoplamento, onde o estado de seu componente deve ser minimamente revelado, e novas funcionalidades são adquiridas pela montagem e composição de componentes. Usuários desses componentes podem depender apenas das interfaces, que são descrições e/ou especificações do comportamento do componente. (HUIZING, 1999) (JUNG et al., 2000)

O reuso traz consigo algumas vantagens, como confiabilidade, redução de riscos e desenvolvimento acelerado. Componentes reutilizados que são empregados nos sistemas em operação tendem a ser mais confiáveis do que os componentes novos, pois já foram experimentados e testados em diferentes ambientes. Os defeitos de projeto e de implementação são descobertos e eliminados no uso inicial dos componentes, reduzindo, assim, o número de falhas quando o componente é reutilizado. Ao recorrer a um componente já existente, incertezas nas estimativas de custos de projeto, sobretudo em componentes grandes, como subsistemas, são reduzidas. O reuso de componentes acelera a produção, porque o tempo de desenvolvimento e o de validação são reduzidos. O exemplo mais claro disso são os componentes desenvolvidos para implementar interfaces gráficas com usuário. (SOMMERVILLE, 2003)

As técnicas de reuso nada mais são que as boas técnicas de programação levadas à consciência dos programadores. Keating e Bricaud (2002) e citam como principais a boa documentação, o bom código e o uso de ambientes de desenvolvimento eficientes. O reuso de componentes de software, quando apoiado por um ambiente, deve ser realizado por ferramentas que incluam os seguintes elementos:

- Uma base de dados de componentes capaz de guardar componentes de software e a informação de classificação necessária para recuperá-los.
- Um sistema de gestão de biblioteca que forneça acesso à base de dados
- Um sistema de recuperação de componentes de software que permita a uma aplicação cliente recuperar componentes e serviços do servidor da biblioteca.
- Ferramentas CBSE que suportam a integração de componentes reusados numa implementação ao projeto novo.

Cada uma dessas funções interage ou é incorporada aos limites de uma biblioteca de reuso. Essa biblioteca é um elemento de um repositório CASE maior e fornece facilidades para o armazenamento dos componentes, sendo formada pela base de dados e ferramentas necessárias para a consulta e recuperação dos elementos imersos nessa base.

A complexidade não se limita, então, apenas ao crescente número de funcionalidades de um sistema, mas também ao maior número de requisitos de um projeto. Entre eles destacam-se o *time-to-market*, e requisitos de desempenho e segurança. A complexidade, neste caso, está presente em todas as dimensões, desde o número de arquivos de projeto até o crescente número de portas de sistema. Com o objetivo de ajudar os projetistas a acelerar e simplificar o desenvolvimento de projetos de ponta é sugerido um trabalho composto por uma metodologia de projeto e um ambiente para criação, desenvolvimento e gerência de projetos de CI complexos.

Esse documento é composto por cinco capítulos, onde os de números dois e três mostram um panorama das áreas de engenharia baseada em componentes e reuso em descrições de hardware, com desenvolvimento RAD e ambientes de programação, respectivamente. O capítulo quatro mostra o processo Macanudo, composto por um modelo baseado em componentes. Apresenta ainda, um protótipo implementado com o objetivo de demonstrar a validade do modelo.

2 ENGENHARIA DE SOFTWARE BASEADA EM COMPONENTES

O objetivo da Engenharia de Software Baseada em Componentes (*Component Based Software Engineering*, CBSE) é o desenvolvimento de sistemas pela composição de componentes reusáveis, através da criação de blocos construtivos de software. Esses blocos construtivos são freqüentemente chamados de componentes e necessitam padrões para interação, composição, infra-estrutura e serviços. O desafio de CBSE é definir modelo de componentes com esses padrões e prover implementações para esses modelos possibilitando projeto, implementação e implantação dos componentes.

Os conceitos básicos em CBSE se originaram de diferentes áreas da engenharia de software e da ciência da computação, como programação orientada a objetos, reuso, arquitetura de software, linguagens de modelagem e especificações formais. (CRNKOVIC et al., 2002) Dentre essas áreas, CBSE é usualmente considerada a evolução do paradigma orientado a objetos. Assim sendo, muitos costumam relacionar os termos objetos e componentes, usando-os até mesmo como sinônimo. Entretanto, esses termos estão relacionados a conceitos independentes, mesmo que os componentes tenham sido implementados através de orientação a objetos.

Objetos são entidades que encapsulam estado e comportamento, possuindo uma identificação única. O comportamento e a estrutura dos objetos são definidos através das classes. Uma classe implementa o conceito de um tipo abstrato de dados e provê uma descrição abstrata para o comportamento de seus objetos. Adicionalmente, ela é usada para criar instâncias e seu nome é normalmente usado como nome de tipo em linguagens fortemente tipadas.

Os componentes possuem uma grande similaridade com as classes, quanto ao estado e comportamento, sendo comparáveis quanto ao encapsulamento que provém a seus estados e a interface que provêm para o mundo externo. Também podem ser criados a partir de outros componentes, como as classes podem ser formadas através da interação de diferentes objetos. Entretanto, o conceito de tipo e o conceito de implementação são completamente separados. A preocupação com a interface é mais importante para os componentes do que usualmente acontece com OO. Técnicas de contratos e pré e pós-condições acabam sendo renovadas e reutilizadas em desenvolvimento baseado em componentes (*Component Based Development*, CBD). Componentes precisam estar de acordo com as regras e padrões definidos pelo modelo de componentes. (WEINREICH; SAMETINGER, 2001)

Um dos desafios de CBSE é o projeto de componentes seguros e robustos, sem que, no entanto, percam características importantes como leveza e facilidade de uso. O segundo é como desenvolver componentes de uma maneira que sejam genéricos o suficiente para serem utilizados em vários ambientes diferentes. Padrões de projeto são soluções genéricas para problemas que ocorrem em diferentes contextos. Seu uso já mostrou ser bem sucedido no desenvolvimento orientado a objetos, e suas abstrações se mostram promissoras em CBD. (HUIZING, 1999)

Existem três requisitos fundamentais para o projeto e o desenvolvimento baseado em componentes:

- Deve ser possível encontrar componentes reutilizáveis apropriados. As organizações necessitam de uma base de componentes reutilizáveis adequadamente catalogados e documentados. Deve ser fácil encontrar componentes nesse catálogo, se ele existir.
- O responsável pelo reuso dos componentes precisa ter certeza de que os componentes se comportarão como especificado e de que serão confiáveis. Idealmente, todos os componentes no catálogo de uma organização devem estar certificados, a fim de confirmar que atingiram determinados padrões de qualidade. Na prática, essa situação não é realista e as pessoas em uma empresa aprendem de maneira informal sobre componentes confiáveis.
- Os componentes devem ter uma documentação associada para ajudar o usuário a compreendê-los e adaptá-los a uma nova aplicação. A documentação deve incluir informações sobre onde os componentes foram reutilizados e sobre quaisquer problemas de reuso que tenham sido encontrados.

Entre as metodologias utilizadas em CBSE está a Engenharia de Domínio. Seu objetivo é identificar, construir, catalogar e disseminar um conjunto de componentes que tem aplicabilidade num domínio de aplicação particular. A meta global é estabelecer mecanismos que permitam aos projetistas compartilhar os componentes de maneira a reusá-los, durante o desenvolvimento de sistemas novos e/ou existentes. A engenharia de domínio possui três principais atividades: análise, construção e disseminação.

2.1 Componente

Um componente de software é um elemento de software, uma parte não trivial do sistema, praticamente independente e substituível que preenche uma clara função seguindo as regras definidas por uma arquitetura, chamada de modelo de componentes. Pode ser implantado independentemente e acoplado a outros componentes de acordo com uma padronização de composição. O componente reflete abstrações estáveis de domínio, que são conceitos fundamentais no domínio de aplicações. Um componente é especificado pelas funções com as quais se comunica com o ambiente e pelos atributos que não são expressos pelas funcionalidades que disponibilizam, mas que definem o seu comportamento. (CRNKOVIC et al., 2002)

Os componentes são mais abstratos do que as classes de objetos e podem ser considerados provedores de serviços *stand-alone*. Quando um sistema precisa de algum serviço, ele chama um componente para fornecer esse serviço, sem preocupar-se a respeito de onde esse componente está sendo executado ou com a linguagem de programação utilizada para desenvolver o componente. Essa independência de ambiente

reflete uma maior possibilidade de adaptação e extensão. Assim sendo, o componente deve ocultar a maneira como seu estado é representado e deve fornecer operações que permitam que o estado seja acessado e atualizado. Um componente se comunica com o ambiente através de interfaces. Um componente deve ter suas interfaces claramente especificadas, enquanto que sua implementação deve ficar encapsulada, não sendo alcançada diretamente pelo ambiente. Isso é o que torna possível a composição por outras pessoas. (CRNKOVIC et. al, 2002)

Os componentes publicam sua interface e todas as interações são feitas por meio desse mecanismo. A interface do componente é expressa em termos de operações parametrizadas e seu estado interno nunca é exposto. Todas as exceções devem ser parte da interface do componente. Os componentes não devem manipular as próprias exceções, uma vez que diferentes aplicações terão diferentes requisitos para a manipulação de exceções. Em vez disso, o componente deve definir que exceções podem surgir e deve publicá-las como parte da interface. Eventualmente, um componente surge como entidade executável independente. O código-fonte não está disponível, de modo que o componente não é compilado com outros componentes do sistema.

Os componentes são definidos por suas interfaces, como podemos ver na figura 2.1, onde a primeira define os serviços fornecidos pelo componente, chamada de *provides*, e a segunda especifica que serviços devem estar disponíveis, chamada de *requires*.

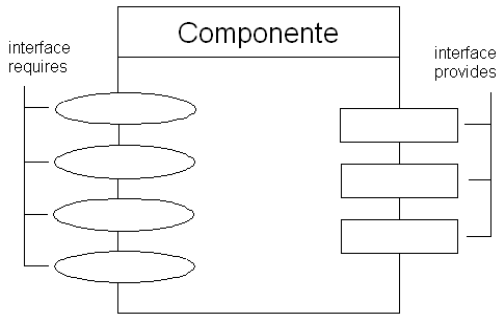


Figura 2.1: Exemplo de Representação de Componente e Interface

2.2 Descrição de Componentes Reusáveis

Um componente de software reusável deve ser descrito usando o modelo 3C – conceito, conteúdo e contexto, como pode ser visto na tabela 2.1. Para ser útil num local pragmático, o modelo 3C precisa ser traduzido num esquema concreto de especificação. Os métodos de classificação de componentes mais usados estão distribuídos em três grandes áreas: métodos de bibliotecas e ciência da informação, métodos de inteligência artificial e sistemas de hipertexto, onde a maioria dos trabalhos sugere o uso de métodos de biblioteconomia para a classificação de componentes. (PRESSMAN 2002)

Tabela 2.1: Modelo 3C

Conceito	Comunica o objetivo do componente, aquilo que ele faz. A interface é descrita e a semântica identificada, através das representações de pré e pós-condições.
Conteúdo	Descreve a realização do conceito. É toda aquela informação pertinente para projetistas que desejam manipular internamente o componente, através de modificações e/ou testes.
Contexto	Localiza o componente dentro de um determinado domínio de aplicabilidade. Permite ao projetista encontrar o componente adequado para satisfazer os requisitos da aplicação.

Tabela 2.2: Classificação de Bennatam

Componentes de prateleira	Software existente que pode ser adquirido de terceiros ou que foi desenvolvido internamente para um projeto anterior. Componentes COTS (*comercial off-the-shelf*) são comprados de terceiros, estão prontos para uso no projeto atual e são plenamente validados.
Componentes de experiência-plena	Especificações, projeto e código ou dados de teste existentes desenvolvidos para projetos anteriores, que são similares ao software a ser construído para o presente projeto. Os membros da equipe atual de software têm plena experiência da área de aplicação representada por esses componentes. Assim, as modificações necessárias para os componentes de experiência-plena serão de risco relativamente baixo.
Componentes de experiência-parcial	Especificações, projeto e código ou dados de teste existentes desenvolvidos para projetos anteriores, que são relacionados ao software a ser construído para o presente projeto, mas que vão exigir substancial modificação. Os membros da atual equipe de software têm apenas experiência limitada na área de aplicação representada por esses componentes. Assim, as modificações necessárias nos componentes de experiência parcial têm um grau de risco considerável.
Componentes novos	Componentes de software que precisam ser construídos pela equipe de software especificamente para as necessidades do presente projeto.

Tabela 2.3: Classificação de Meyer

Abstração Funcional	O componente implementa uma única função, como uma função matemática. Essencialmente, a interface *provides* é a própria função.
Agrupamentos Casuais	O componente é uma coleção de entidades inadequadamente relacionadas, que podem ser declarações de dados, funções, entre outras. A interface *provides* consiste em nomes de todas as entidades do agrupamento.
Abstrações de Dados	O componente representa uma abstração de dados ou classe em uma linguagem orientada a objetos. A interface *provides* consiste em operações para criar, modificar e acessar a abstração de dados.
Abstrações em Clusters	O componente é um grupo de classes relacionadas que trabalham em conjunto. Elas são chamadas, às vezes, de *framework*. A interface *provides* é a composição de todas as interfaces *provides* dos objetos que constituem o *framework.*
Abstração de Sistema	O componente é um sistema inteiramente autocontido. Reutilizar abstrações de nível de sistema é, às vezes, chamado de reuso de produtos COTS. A interface *provides* é chamada de API (*Application Program Interface* – Interface de Programação de Aplicações), que é definida para permitir que os programas acessem os comandos e as operações do sistema.

Muitas divisões e classificações de componentes foram criadas. Bennatan (1992) fez uma das primeiras, sugerindo quatro categorias de componentes, considerando a quantidade de informação e o impacto que pode causar no projeto, mostrada na tabela 2.2. Meyer (1999) cria uma separação em cinco níveis de abstração que pode ser vista na tabela 2.3. Pressman (2002) cria uma divisão mais abrangente, vista na tabela 2.4, levando em consideração aspectos mostrados desde Bennatan, ampliando a análise.

2.2.1 COTS

O termo COTS (produtos de prateleira) pode, em princípio, se aplicar a qualquer componente oferecido por um terceiro, sendo utilizado para se referir a produtos de hardware ou software. As funcionalidades dos COTS são mais amplas do que a de componentes mais especializados, obtendo um maior potencial de retorno por reuso.

Utilizar um sistema COTS em larga escala é o mesmo que utilizar qualquer outro componente específico. É preciso compreender as interfaces dos sistemas e utilizá-las exclusivamente para comunicar-se com o componente. É necessário um compromisso entre requisitos específicos em relação ao desenvolvimento rápido e ao reuso; deve-se projetar uma arquitetura de sistemas que permita aos sistemas COTS operarem juntos.

Os benefícios do reuso dos produtos COTS são muito significativos, porque esses sistemas oferecem muita funcionalidade para o usuário. Meses e, algumas vezes, anos de esforço de implementação poderão ser economizados, se um sistema existente for

reutilizado e os prazos de desenvolvimento do sistema puderem ser drasticamente reduzidos. Uma vez que a entrega rápida do sistema é o principal requisito de muitos sistemas essa forma de reuso pode provavelmente ser cada vez mais praticada. (PRESSMAN, 2002)

Tabela 2.4: Classificação de Pressman

Componente de software em execução	Um pacote dinamicamente constituído de um ou mais programas, geridos como uma unidade, ao qual se tem acesso através de interfaces documentadas, que podem ser descobertas durante a execução.
Componente de software	Uma unidade com dependências de contexto apenas explícita e contratualmente especificadas.
Componente de negócio	Implementação, em software, de um conceito de negócio ou processo de negócio "autônomo".
Componentes qualificados	Avaliados por engenheiros de software para garantir que não apenas a funcionalidade, mas também o desempenho, a confiabilidade, a usabilidade e outros fatores de qualidade satisfazem os requisitos do sistema ou do produto a ser construído.
Componentes adaptados	Adaptados para modificar características não requeridas ou indesejáveis.
Componentes montados	Integrados no estilo arquitetural e interconectados com uma infra-estrutura adequada para permitir que os componentes sejam coordenados e geridos efetivamente.
Componentes atualizados	Para substituir o software existente à medida que novas versões de componentes se tornam disponíveis.

Entretanto, algumas características devem ser levadas em conta sobre COTS, como a falta de controle sobre a funcionalidade e o desempenho, assim como problemas com interoperabilidade e controle de evolução. Embora a interface publicada de um produto possa aparecer para oferecer os recursos requeridos, esses podem não estar adequadamente implementados ou podem operar de maneira inadequada. O produto pode ter operações ocultas que interferem em sua operação. Resolver esses problemas pode ser uma prioridade para o integrador de produtos COTS, mas pode não ser uma preocupação real para o fabricante do produto. Os usuários podem simplesmente precisar encontrar um meio de contornar os problemas, se desejarem reutilizar o produto COTS. O nível de suporte técnico disponível a partir dos fabricantes de COTS varia muito. Por serem sistemas de prateleira, o suporte técnico do fabricante é

particularmente importante quando surgem problemas, porque os desenvolvedores não têm acesso ao código-fonte e à documentação detalhada do sistema. Embora os fabricantes possam se comprometer em fornecer suporte técnico, as mudanças de mercado e as circunstâncias econômicas podem tornar difícil que eles cumpram com seu compromisso. Adicionalmente, são os fabricantes de produtos COTS que tomam suas próprias decisões sobre as mudanças nos sistemas, em resposta às pressões de mercado. Essa falta de suporte, juntamente com uma interface pobremente implementada ou documentada, torna difícil a integração dos COTS.

2.3 Modelo de Componentes

Um modelo de componentes define padrões de interação e composição específicos, e opera em dois níveis. Primeiro, um modelo de componentes define como construir um componente individualmente. Em segundo lugar, modela o comportamento global de como o conjunto de componentes em um sistema baseado em componentes se comunica e interage entre si. Chama-se montagem quando componentes são compostos entre si, enquanto que a composição dada entre um componente e um outro elemento de *software* é chamada de conexão.

Tabela 2.5: Elementos Básicos de um Modelo de Componentes.

Padrão	**Descrição**
Interface	Especificação das propriedades e do comportamento dos componentes. Definição de uma linguagem de descrição de interface (*Interface Description Language*, IDL).
Nomenclatura	Nomes globais únicos para componentes e interfaces.
Metadados	Informação adicional sobre componentes e interfaces, assim como suas relações.
Interoperabilidade	Comunicação e troca de dados entre componentes de diferentes fontes, implementadas diferentemente.
Adaptação e Evolução	Interfaces para a personalização de componentes. Regras e serviços para modificações de versões de componentes.
Composição	Regras e interfaces para a criação de componentes maiores. Substituição e adição em estruturas existentes através de combinação de componentes existentes.
Empacotamento e Implantação	Implementação de pacotes e recursos necessários para instalação e configuração do componente.

O modelo de componentes deve definir mecanismos de personalização capazes de estender componentes sem modificá-los. Pode-se considerar, assim, personalização como uma forma avançada de interação. Adicionalmente, define propriedades fundamentais do componente, como formato de código e padrões de documentação. Usualmente, componentes são implantados independentemente e estão sujeitos a composição por terceiros. Isso faz necessário o surgimento de padrões para a comunicação entre componentes de diferentes produtores.

Uma infra-estrutura de componentes de software é o conjunto de componentes de software projetado para assegurar que os sistemas construídos usando esses componentes e interfaces satisfazem claramente as especificações do modelo de componentes. Alguns deles são finas camadas que executam no topo de um sistema operacional, possibilitando a múltiplas plataformas utilizar o máximo de aplicabilidade do modelo de componentes.

Um projeto baseado em componentes reflete o processo de padronização dos serviços disponibilizados, de uma maneira mais geral chegando a domínios específicos. Serviços e infra-estruturas horizontais provêm funcionalidades adicionais a múltiplos domínios, como o serviço de gerência de interfaces. Serviços e infra-estruturas verticais dão suporte a um domínio mais particular, como serviços de telecomunicações ou financeiros. (WILLIAMS, 2001)

Skyperski (2002) fala que o primeiro exemplo de sucesso de sistemas baseados em componentes são os sistemas operacionais. Fazendo uma analogia, os sistemas operacionais são as implementações de modelos de componentes para aplicações, que podem ser vistos como componentes de grão-grosso. A partir de um desenvolvimento e documentações da implementação, diferentes projetistas podem desenvolver aplicações que usem os serviços providos pelo modelo de componentes. No nível de granularidade de aplicações existe um dos mercados mais amplos, onde é possível a compra de diferentes aplicações de variadas fontes que coexistem e interagem aos padrões estabelecidos por aquele sistema operacional.

Entre os muitos mecanismos para a criação de uma infra-estrutura efetiva está um conjunto de características arquiteturais que devem estar presentes para conseguir a criação de componentes. O primeiro deles é o modelo de intercâmbio de dados, que são mecanismos que permitem que usuários e aplicações interajam e transfiram dados, seja homem-software, componente-componente, ou também entre recursos do sistema.

Existe uma grande dúvida quanto à forma de localização de um determinado componente que realize uma tarefa específica. Mesmo que o mercado de componentes já venha há algum tempo tentando gerar algumas alternativas (UDELL, 1994), ainda não existe uma abordagem sistemática. Assim, os componentes acabam sendo registrados em serviços específicos, usando seus metadados e interfaces e as buscas acabam sendo feitas nesse nível de abstração.

Existem três grandes modelos de componentes usados atualmente: OMG/CORBA, a família Microsoft COM e os JavaBeans/Enterprise JavaBeans da Sun. Alguns autores ainda citam os serviços Web como uma quarta frente.

2.3.1 OMG/CCM

O CORBA surgiu em Outubro de 1991, como um negociador de solicitação entre objetos. Tinha em sua base um modelo de objetos, a linguagem de definição de interface (IDL), um núcleo de API para a gerência dinâmica e um repositório de interfaces. Em sua última especificação, traz consigo a versão 3.0 do CCM (CORBA *Component Model*), lançada também como uma especificação solo. Suas maiores novidades são uma maior integração com outras tecnologias de componentes e com Java. Essas modificações trazem consigo uma clara demonstração de preocupação com os programadores e seu uso de CORBA.

Um componente é um meta-tipo básico de CORBA, sendo uma extensão do meta-tipo objeto. Os tipos de componentes são especificados pela IDL e representados no repositório de interfaces. (OMG, 2002) Cada tipo de componente está associado a um conjunto de comportamentos bem-definido. Fica a cargo das implementações desses tipos prover o comportamento esperado para cada plataforma de hardware ou sistema operacional, assim como versões em diferentes linguagens de programação. Um tipo acaba sendo "instanciado" com o objetivo de construir entidades concretas, possuidoras de estado e identificação.

Dividem-se os componentes em dois níveis: básico e estendido. Os componentes básicos essencialmente transformam um objeto CORBA em componente, enquanto que o segundo grupo, como o nome diz, estende esses objetos mais simples a um nível maior de funcionalidades. Um componente básico acaba sendo muito similar em funcionalidade a um EJB. Contudo, não pode fazer parte de hierarquia, não participando das vantagens de herança, sendo declarados a partir de uma versão simplificada, vista na figura 2.2. (OMG, 2002)

```
"component" <identifier> [<supported_interface_spec>]
                "{" <attr_dcl> ";"} * "}"
```

Figura 2.2: Declaração de Componente Básico. (OMG, 2002)

Componentes CCM possuem um mecanismo chamado *porta* para a comunicação com artefatos CORBA. Esse mecanismo especifica diferentes visões e interfaces expostas por componentes a seus clientes. São quatro os tipos de portas. Componentes do tipo básico oferecem apenas um tipo: atributos. Componentes do tipo estendido oferecem qualquer um dos quatro tipos, listados na tabela 2.6.

As implementações de interface são encapsuladas pelos componentes. A estrutura interna fica então a cargo do projetista, e escondida de seus clientes que a tomarão como existente, tirando por base o contrato estabelecido pela IDL. Adicionalmente, alguns componentes podem ter a si associados uma identificação chamada de "chave-primária" (*primary key*). Esse valor é atribuído por um *component home* para que seja identificada cada instância dentro de um contexto. Um *component home* é um metatipo que age como gerente de instâncias.

Uma implementação de um componente é um conjunto de artefatos inter-relaciodos. Essa agregação é descrita através de uma descrição CIDL (Component Implementation Definition Language). A CIDL é uma linguagem declarativa para descrever a estrutura e o estado das implementações de componentes.

O processo de separação entre interfaces e implementações pode ser visto na figura 2.3. A descrição IDL é compilada em *stubs* (dos clientes) e *skeletons* (dos objetos) que servem como proxies. São também definidos os próprios clientes e objetos. Como os *stubs* e *skeletons* são definidos através da mesma linguagem de descrição, não existem problemas de incompatibilidade entre uma requisição de um cliente a um objeto, mesmo que tenham sido compilados com linguagens distintas. (OMG, 2005)

Tabela 2.6: Tipos de Porta do CCM

Atributos	Para habilitar a configuração dos componentes, CCM estende a definição de atributos do modelo de objetos CORBA. Esses atributos são alcançados através de operadores acessores. Ferramentas de configuração podem usar os atributos para pré-configurar valores ao componente. Fica a cargo dos projetistas a escolha entre manter as implementações de atributos de forma transiente ou persistente. CCM permite ainda que atributos estejam associados a exceções.
Facet	São as interfaces que um componente provê. Podem não estar relacionadas com as interfaces adquiridas via herança. As *facet* disponibilizam ao componente uma exposição de visões diferentes de suas interfaces a seus clientes. Isso pode acontecer ao invocar de maneira síncrona pelas operações de dois sentidos do CORBA, ou ainda pelo AMI (*Asynchronous Method Invocation*) que permite a utilização assíncrona dos serviços.
Receptacles	Antes que um componente delegue operações a outros componentes, ele deve obter referências às instâncias usadas por esses componentes. Para o CCM essas referências são os conectores dos objetos e os nomes de portas dessas conexões são chamados de *receptacles*.
Event sources/sinks	Outra forma de interação entre componentes é através do monitoramento de eventos assíncronos. Um componente declara explicitamente os eventos disponibilizados ou requeridos em sua definição. Esse tipo de comportamento é baseado no padrão Observer (GAMMA et al, 2000).

Um pacote CCM mantém uma ou mais implementações de componentes. Ele pode ser instalado separadamente ou em grupos interconectados chamados *assembly*. Um pacote é representado por um descritor e um conjunto de arquivos, agrupados em conjunto por um arquivo do tipo zip. XML é usado como base para descrever cada um dos pacotes. O descritor é formado por informações gerais sobre o componente, seguida de uma ou mais seções descrevendo sua implementação. Um arquivo descritor tem a extensão ".csd", que vem da abreviatura de *CORBA Software Descriptor* (CSD) e quando é parte integrante de um pacote fica localizado em uma pasta chamada "meta-inf", na raiz do documento.

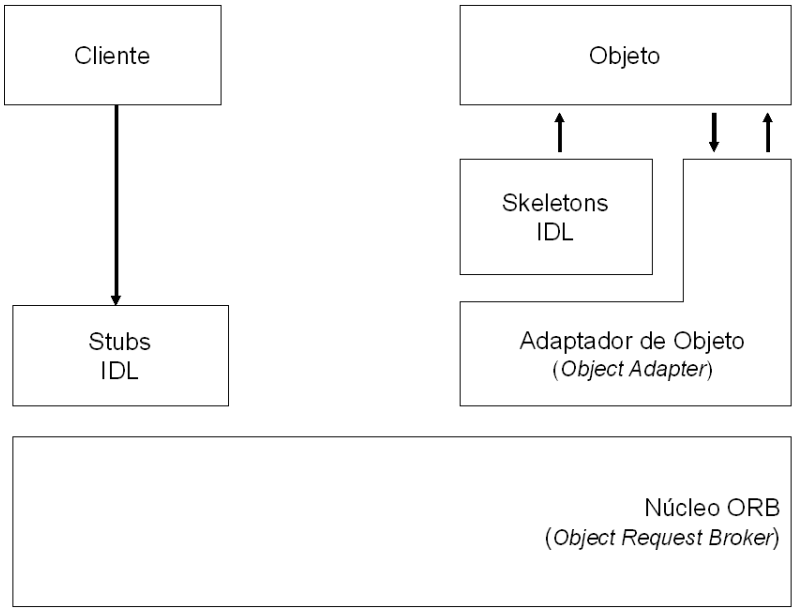


Figura 2.3: Modelo de Objetos CORBA (WANG; SCHMIDT; O'RYAN, 2001)

Wang, Schmidt e Ryan (2001) citam alguns problemas do CCM:

- Falta de padronização na maneira de implantar componentes;
- Padronização limitada a suporte de padrões de programação de servidores CORBA;
- Expansão de funcionalidade de componentes limitada;
- Disponibilidade de serviços CORBA indisponíveis antecipadamente;
- Falta de padronização de gerência de ciclo de vida de objetos.

2.3.2 Microsoft/COM

A Microsoft desenvolveu um produto de objeto de componentes (*Component Object Model*, COM) que fornece uma especificação para usar componentes produzidos por diversos fornecedores em uma única aplicação, rodando sob o sistema operacional Windows. COM contém dois elementos: interfaces COM e um conjunto de mecanismos para registrar e passar mensagens entre essas interfaces. Do ponto de vista da aplicação o enfoque não é como os objetos são implementados, mas sim que ele usa um sistema que disponibiliza uma interface para comunicação entre objetos. (EDDON; EDDON, 1998) Para facilidade de expressão, o termo COM será usado para a família COM/DCOM/COM+, sendo que a diferenciação ocorrerá apenas em situações onde se faça necessária mostram características específicas.

Com o ambiente Windows multitarefa, a Microsoft disponibilizou a possibilidade de troca de informação entre as diferentes aplicações. A área de transferência (*clipboard*) e a DDE (*Dynamic Data Exchange*) são as primeiras iniciativas incorporadas ao sistema operacional com o intuito de facilitar a comunicação entre os processos. A DDE era um protocolo muito complexo de ser implementado pelos projetistas e acabou sendo pouco

usado. A área de transferência introduziu o paradigma do copiar-colar (*copy-paste*), fácil de implementar pelos programadores e de fácil uso pelos usuários finais.

A área de transferência é usada na criação de documentos compostos. Entretanto, esses dados são compostos de maneira estática, fazendo com que alterações nos documentos originais não sejam propagados nas composições. Para resolver esse problema, em 1992, com o Windows 3.1 foi criado o OLE (*Object Linking and Embedding*).

Uma segunda versão do OLE foi lançada em 1993, refinando o conceito de ligação de múltiplas fontes para uma edição visual em única janela. Mesmo não visualizada pelos projetistas que a usavam, essa abordagem já era o embrião de um paradigma baseado em componentes. Isso é refletido em 1996, com o lançamento do Windows NT 4, onde o COM é lançado, podendo não só invocar objetos de fontes distintas do mesmo ambiente, como também através de uma rede. A evolução do uso dessa tecnologia desemboca no lançamento do DCOM junto com o Windows 95 em 1997. (EDDON; EDDON, 1998)

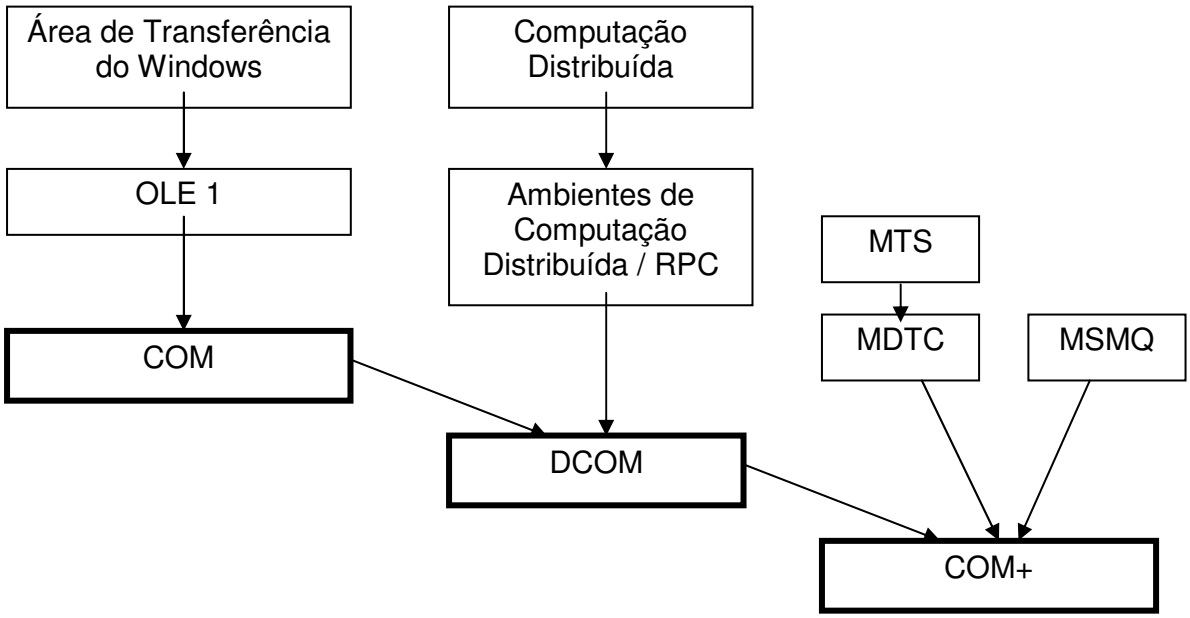


Figura 2.4: Evolução do Modelo COM

Pelo outro caminho, DCOM surge a partir da computação distribuída. No final da década de 80, vários grupos de indústrias juntavam esforços para a criação de padrões em várias áreas da computação. Um dos principais grupos a surgir e tomar força foi o OSF (*Open Software Foundation*). Esse grupo cria a especificação de um ambiente de computação distribuído (DCE, *Distributed Computing Environment*) com o objetivo de prover a base para o desenvolvimento de sistemas distribuídos. Para a comunicação entre diferentes aplicações entre computadores distintos, a OSF cria um protocolo chamado RPC (*Remote Procedure Call*). Esse protocolo é usado pela DCOM para a comunicação entre os computadores. (EDDON; EDDON, 1998)

Em 1997, foi lançado o Servidor de Transações da Microsoft (MTS, *Microsoft Transaction Server*) que mais tarde daria origem a um coordenador distribuído de transações (MDTC, *Microsoft Distributed Transaction Coordinator*). Ainda em 1997, a

Microsoft desenvolveu o *Microsoft Message Queue* (MSMQ). Em 1999 esses serviços são agrupados com a base COM em um ambiente de tempo de execução chamado COM+. Esse ambiente acaba sendo disponibilizado juntamente com o Windows 2000. (MICROSOFT, 2005) (CHAND, 2000) A evolução desses modelos pode ser vista através da figura 2.4.

O modelo de identificação global única (GUID, *Global Unique Id*) é usado pela família COM. Um GUID é um número de 128 bits que combina um identificador de local, hora de criação e um número gerado randomicamente.

O modelo COM é independente de linguagem de programação. Componentes podem ser desenvolvidos em diferentes linguagens de programação, sem problemas pra interoperabilidade dos mesmos. Adicionalmente, um componente pode ser desenvolvido com uma linguagem e sua evolução ocorrer através de uma outra, diferente da primeira. Adicionalmente, os mecanismos de evolução para seus componentes são flexíveis. Clientes podem consultar dinamicamente as funcionalidades disponibilizadas pelos componentes. Ao invés de expor sua interface em um único agrupamento de métodos e atributos, um componente COM pode se mostrar diferentemente para clientes distintos. (MICROSOFT, 1996)

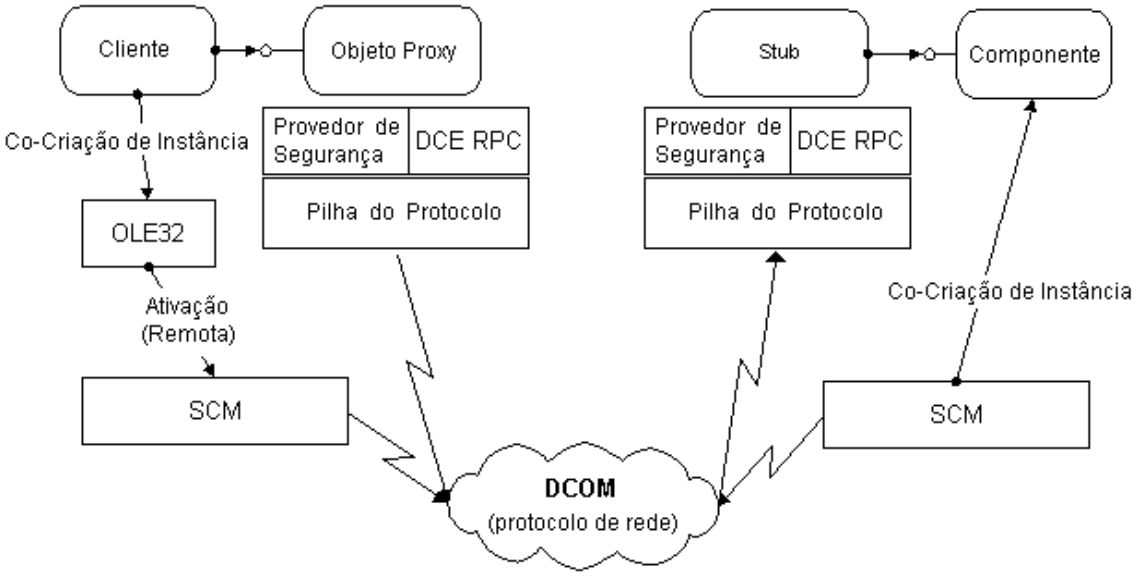


Figura 2.5: Arquitetura DCOM (MICROSOFT, 2005)

Detalhes de implantação não são especificados no código-fonte, pois o modelo esconde a localização de um componente, seja no mesmo processo do cliente ou em uma máquina geograficamente distante. Em qualquer um dos casos, a maneira com que o cliente se conecta com um componente e chama seus métodos é idêntica. Essa forma de comunicação, vista na figura 2.5, não se baseia em mudanças no código-fonte do componente, mas em reconfigurações no DCOM. (MICROSOFT, 1996)

COM gerencia as conexões aos componentes que são dedicadas a um único cliente, assim como componentes que são compartilhados a múltiplos clientes, pela manutenção a um contador de referência. Quando um cliente estabelece uma conexão com um

componente, DCOM incrementa o contador de referências. Quando um cliente libera a sua conexão, COM decrementa o contador. Se o contador chegar ao valor zero, o componente pode ser liberado. COM possui uma transparência de segurança pela disponibilização a desenvolvedores e administradores do poder de configuração das características de cada componente. Assim como o sistema de arquivos do Windows NT deixa que o administrador estabeleça uma lista de controle de acesso para arquivos e diretórios, COM armazena uma lista para os componentes. Essas listas indicam quais usuários ou grupos de usuários tem o direito de acesso a componentes de uma determinada classe e são configuradas a partir de uma ferramenta de configuração (DCOMCNFG) ou através do registro do Windows NT. Sempre que um cliente solicitar um método ou criar uma instância de um componente, COM obtêm os dados de usuário e processo corrente desse cliente. O Windows NT garante que essa credencial é autêntica. COM então passa essas informações para o componente que valida essa informação e consulta a lista de controle de acesso.

Uma especificidade do modelo de programação COM+ é que esse modelo requerer que cada componente configurado exista no escopo de uma aplicação COM+. Essa restrição limita a associação de um componente a mais de uma aplicação COM+ por computador. (PATTISON, 2000)

2.3.3 Sun/JavaBeans

O objetivo de JavaBeans é definir um modelo de componentes de software para Java. (SUN, 1997) Os Beans (como também são chamados os JavaBeans) podem ser divididos em dois grupos, quanto a sua utilização. O primeiro, sendo usado como blocos de composição em aplicações de montagem. O segundo, para a criação de documentos compostos. Mesmo que os dois grupos eventualmente possam estar interligados, fazendo um componente ter um comportamento distinto em uma determinada situação, existe um fator em comum a eles: um bean é criado para ser manipulado visualmente. (SUN, 1997).

Para tanto, os beans são construídos de maneira a satisfazer a condições claras de introspecção e personalização, assim como suporte a eventos. Uma rápida visão sobre as funcionalidades que distinguem os beans pode ser vista na tabela 2.7. Um bean não precisa herdar nenhuma característica de alguma classe básica ou interface. Entretanto, bean visuais devem poder ser manipulados por contêineres visuais, sendo subclasses de java.awt.Component.

Os beans devem, então, ser capazes de rodar dentro de uma ferramenta de construção, geralmente chamada de ambiente de desenvolvimento. É a partir desse ambiente que o bean disponibiliza informação de projeto, permitindo personalização. Adicionalmente, o bean deve ser capaz de rodar fora desses ambientes. A Máquina Virtual Java (*Java Virtual Machine*, JVM) usada estará associada a aplicação que serve de container para o bean. A comunicação externa é particular a cada servidor, como vista no exemplo da figura 2.6.

Tabela 2.7: Funcionalidades dos JavaBeans

Introspecção	Permitir ao construtor analisar como funciona o bean.
Personalização	Permitir a modificação de aparência e comportamento. Essas alterações são feitas a partir das propriedades do bean.
Eventos	Permitir uma comunicação simples para conectar beans.
Persistência	Permitir que as alterações de personalização sejam capturadas e salvem as propriedades do bean, sendo possível um carregamento posterior.

Os beans são empacotados e distribuídos em arquivos no formato JAR. Um arquivo JAR é um arquivo compactado que engloba recursos como arquivos de classes, imagens, arquivos de ajuda e outros arquivos de recursos. Existe um componente opcional chamado de arquivo de manifesto (*manifest file*), com metadados sobre o arquivo JAR. Ele deixa de ser opcional quando o JAR corresponde a um bean. Os objetos Java são criados usando um modelo hierárquico de nomes. A SUN aconselha o uso de um domínio registrado de Internet como a raiz do nome para os componentes. Essa mesma política de nomes é usada para os JavaBeans.

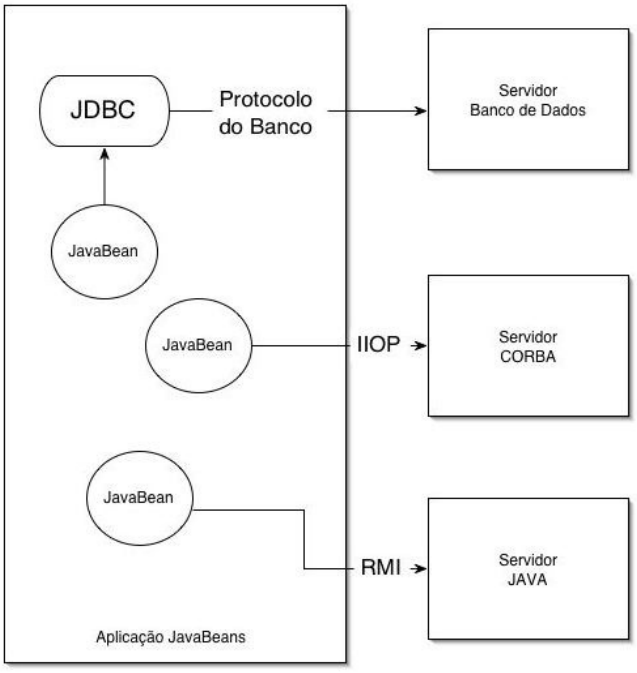


Figura 2.6: Exemplo de Aplicação Utilizando JavaBeans (SUN, 1997)

O Bean Developer Kit (BDK) é um conjunto de ferramentas que permite aos desenvolvedores analisar como os componentes Beans existentes funcionam (introspecção), adaptar seu comportamento e aparência (personalização), estabelecer

mecanismos para coordenação e comunicação (suporte a eventos), assim com testar e avaliar o comportamento do Bean.

2.3.4 Web-Services

Recentemente, os *web-services* ganharam atenção, sendo considerados como os portadores das resoluções dos problemas apresentados por outros padrões de componentes, principalmente nos níveis de integração de software e interoperabilidade. Um serviço é uma instância de configuração de sistema que roda a partir de uma organização provedora. Essas organizações provedoras de serviços instalam, rodam e mantém as infra-estruturas de hardware, software e componentes.

Szyperski é um grande entusiasta do que ele chama de serviços de software baseados em XML na web (SZYPERSKI, 2003) e relaciona os benefícios das propriedades de serviço sem a perda de vantagens dos componentes. Por outro lado, Gokhale et al (2002) fazem uma comparação entre *web-services* e CORBA e chegam à conclusão que conceitualmente são idênticos. As divergências entre os autores estão basicamente em duas características. A primeira é o uso de XML ao invés da IDL usada pelo CORBA, enquanto que a segunda fala em estados observáveis disponíveis para componentes CORBA, geralmente inexistentes nos *web-services*.

2.4 Desenvolvimento Baseado em Componentes

O processo começa quando uma equipe de software estabelece os requisitos para um sistema a ser construído usando técnicas de coleta de requisitos convencionais. Um projeto arquitetural é estabelecido, mas ao invés de passar imediatamente a tarefas de projeto mais detalhadas, a equipe examina os requisitos para determinar que subconjunto é mais adequado à composição, do que à construção.

Em princípio, CBD propicia o desenvolvimento de componentes que implementam completamente uma solução tecnológica ou um aspecto de negócio. Esses componentes estão aptos a serem usados em qualquer lugar. Funcionalmente, são implementados apenas uma vez, ficando claras suas vantagens sobre os pontos de vista de manutenibilidade, robustez e produtividade. Esse reuso pode acontecer dentro do mesmo ambiente de desenvolvimento como também através de reuso de componentes de terceiros.

A equipe tenta modificar ou remover os requisitos do sistema que não podem ser implementados com componentes, próprios ou COTS. Se os requisitos não puderem ser mudados ou descartados, os métodos convencionais ou de OO podem ser aplicados para desenvolver aqueles componentes novos, que precisam ser trabalhados para satisfazer os requisitos. A existência de componentes reusáveis não garante que esses componentes possam ser integrados efetivamente na arquitetura escolhida para a aplicação. Por essa razão, desenvolveu-se uma seqüência de atividades que visam identificar a aplicabilidade dos componentes.

No entanto, como toda migração, sair de uma metodologia para CBD não ocorre da noite para o dia e alguns fatores devem ser levados em conta. Especificamente no caso de componentes, é continuamente necessário relembrar que o reuso é um investimento de longo prazo. É natural que exista uma certa resistência na utilização de uma nova abordagem, e em CBD um dos principais problemas para se instituir um uso efetivo de

reuso é a síndrome do "Não Inventado Aqui" (*Not Invented Here* – NIH). (ALLEN, FROST, 2001)

2.4.1 Fases do Desenvolvimento Baseado em Componentes

Na produção do componente, o tempo despendido é direcionado a tarefas de análise regras de negócio, modelagem, e projeto e análise. Menos tempo acaba sendo gasto com desenvolvimento, enquanto que os testes acontecem em todas as etapas. O ciclo de vida de produção de um componente pode ser visto na figura 2.7.

Figura 2.7: Etapas do Ciclo de Desenvolvimento Baseado em Componentes

2.4.1.1 Qualificação do Componente

Os requisitos e a arquitetura do sistema definem os componentes que serão necessários. Os componentes reusáveis normalmente são identificados pelas características de suas interfaces. Entretanto, a interface não fornece um panorama completo do grau em que o componente vai satisfazer a arquitetura e os requisitos. O engenheiro de software precisa usar um processo de exploração e análise para adequar cada componente. Essa etapa garante que um componente candidato vai realizar a função necessária, vai "se encaixar" adequadamente no estilo arquitetônico especificado para o sistema, e vai exibir as características de qualidade necessárias para a aplicação.

A descrição da interface fornece informação útil sobre a operação e uso de um componente de software, podendo tornar difícil determinar o funcionamento interno de COTS ou componentes de terceiros, porque a única informação disponível pode ser a própria especificação da interface.

2.4.1.2 Adaptação de Componente

A arquitetura de software representa padrões de projeto que são compostos de unidades de funcionalidade, conexões e coordenação. Considerando um componente

como unidade de funcionalidade, a arquitetura, de fato, define as regras de projeto para todos os componentes, identificando os modos de conexão e coordenação. Em alguns casos, os componentes reusáveis existentes podem estar em desacordo com as regras de projeto da arquitetura. Esses componentes devem ser adaptados para satisfazer às necessidades da arquitetura ou descartados e substituídos por outros componentes, mais adequados.

Mesmo depois de um componente ter sido qualificado para uso dentro da arquitetura de uma aplicação, ele pode conter algum tipo de conflito. Para amenizar esse problema de adaptação, uma técnica chamada *empacotamento de componente* é freqüentemente usada. Quando uma equipe de software tem pleno acesso ao projeto e ao código interno de um componente, o empacotamento caixa-branca é aplicado. Como o seu correspondente no teste de software, o *empacotamento caixa-branca* examina os detalhes de processamento interno do componente e faz modificações de código para remover qualquer conflito. O *empacotamento caixa-cinza* é aplicado quando a biblioteca de componentes fornece uma linguagem para extensão de componentes ou API que permite a remoção ou o mascaramento dos conflitos. O *empacotamento caixa-preta* exige a introdução de pré e pós-processamento na interface do componente para remover ou mascaras os conflitos. A equipe de software deve determinar se o esforço necessário para empacotar adequadamente um componente é justificado ou se um componente sob encomenda deve, em vez disso, ser construído.

2.4.1.3 Construção de Novos Componentes

A construção de novos componentes pode ser dada através de duas formas, basicamente: pela composição de componentes existentes e pela criação através de formas clássicas de construção de software.

Em sistemas baseados em componentes um trecho de código é idealmente implementado apenas uma vez. Essa característica leva a uma manutenção mais fácil, assim como a baixo custo e aumento do ciclo de vida desses sistemas. De fato a diferença entre a construção e a manutenção torna-se vaga e tende a desaparecer ao longo do tempo. Novas aplicações tendem a consistir em grande parte de componentes previamente existentes, fazendo com que a construção se assemelhe muito mais a uma montagem do que propriamente uma construção.

A tarefa de composição de componentes combina componentes qualificados, adaptados e construídos para compor a arquitetura estabelecida para uma aplicação. Para conseguir isso uma infra-estrutura deve ser estabelecida para aglutinar os componentes em um sistema em operação. A infra-estrutura fornece um modelo para a coordenação de componentes e serviços específicos, que permitem aos componentes se coordenarem entre si e realizarem tarefas comuns.

2.4.1.4 Atualização dos Componentes

Ao final de cada etapa existe a criação de novos componentes. Eles serão realmente reutilizados quando forem adicionados a uma entidade chamada Biblioteca de Componentes. É ela a responsável pelo gerenciamento de componentes. São suas atribuições as operações de investigação de localização física, assim como análise de estado, escondendo todas essas informações dos clientes que requisitam serviços de componentes. (ZEID, 2004) Quando são implementados sistemas com componentes COTS ou para bibliotecas muito volumosas, a atualização pode ser de difícil

manutenção. Esse tipo de situação cria a necessidade de criação de papéis, como pode ser visto na seção 2.4.4.

2.4.2 Utilização de CBD

Desenvolvimento Baseado em Componentes[1] foi proposto para confrontar os desafios de construção de sistemas complexos e de grande porte de maneira eficiente quanto a custos e tempo, (CRNKOVIC, 2002) e é usado em muitos setores distintos, tendo exemplos de desenvolvimento empresas como IBM, Microsoft e Sun (ZEID, 2004).

Tabela 2.8: Atividades envolvendo CBD.

Programadores usam componentes GUI para reduzir complexidade	70%
Programadores fazem melhor uso de interfaces para encapsular detalhes de implementação	53%
Analistas modelam conceitos de negócios como componentes, de forma independente de qualquer implementação especifica	51%
Existe reuso de componentes de negócios através de diferentes projetos e de grupos de negócios	40%
Componentes são identificados como parte de uma abordagem de planejamento arquitetural	36%

Paul Allen (2002) sintetiza em um artigo, uma pesquisa mais ampla realizada pela Cutter Consortium que entrevistou 118 empresas da Europa, Ásia e América do Norte quanto à sua utilização de CBD. Em duas tabelas o autor sintetiza o estado da arte no quesito prática (tabela 2.8) e a cobertura dos projetos que envolvem componentes (tabela 2.9). Mesmo que a esperada utilização dos componentes de interface apareça como a opção mais usada, os outros itens demonstram que o uso desses artefatos não está apenas na fase de construção, mas sim em etapas anteriores como a análise. Ou seja, projetistas já enxergam suas soluções através de construções que utilizem componentes.

Os bons prognósticos apresentados na tabela anterior, entretanto são reavaliados quanto levamos em consideração os dados apresentados pela tabela a seguir. Mesmo que a idéia de componentes já apareça em diferentes níveis do ciclo de vida de desenvolvimento, é baixo o número de companhias que utilizam CBD em todos os projetos onde estão envolvidos. Isso nos demonstra que esse tipo de desenvolvimento está longe de ser unanimidade. Entretanto, os índices de projetos selecionados nos mostram que as empresas já vêm CBD como uma possibilidade real e factível.

[1] Alguns autores também se referem a CBD como CBSD *(Component-Based Software Development*)

Tabela 2.9: Projetos envolvendo CBD.

Projetos Selecionados	53%
Projetos Piloto	50%
Experimentação de Pesquisa e Desenvolvimento	42%
Todos os Projetos	12%

2.4.3 Ambientes de Desenvolvimento em CBD

Ambientes específicos a certos domínios proporcionam a seus usuários um maior controle sobre suas aplicações, pois lhes é dada uma abordagem mais eficiente para que explorem os domínios através de sua experiência e criatividade. (MORCH, 2004)

Ambientes de desenvolvimento de software foram criados originalmente com o intuito de integrar e gerenciar as coleções de ferramentas e os artefatos produzidos por e com elas. Em um segundo momento os ambientes foram sendo centrados em processos definidos que guiavam o ciclo de desenvolvimento. Lüer (LUER, 2001) apresenta os ambientes de desenvolvimento baseados em componentes (*Component-based development environments,* CBDE), que têm como objetivo integrar a idéia original dos primeiros ambientes com as vantagens de um processo que guie o desenvolvimento. Para tanto, apresentam sete requisitos necessários para um CBDE: projeto modular, auto-descrição de componentes, nomenclatura global de interfaces, desenvolvimento de componentes de e aplicações, conectividade e adaptação, múltiplas visões e reuso por referência. Os pontos levantados na verdade são muito mais atribuições do modelo de componentes do que propriamente dos ambientes. As ferramentas devem, na realidade, possibilitar a seus usuários o desenvolvimento de acordo com o modelo.

Padrões de projeto e frameworks possuem um papel central na implementação e implantação de componentes. Em CBSE, software é construído através da junção de peças. Essa característica torna muito importante a existência de um contexto que suporte a utilização dessas peças. Em geral, frameworks descrevem uma situação de reuso típica em nível de modelo. Assim, frameworks de componentes constituem *circuit boards* com espaços vazios a espera de componentes para preenchê-los. (CRNKOVIC et. al, 2002)

2.4.4 Papéis nas Equipes de CBD

Como toda metodologia de desenvolvimento, também o uso de CBD não acontece sem um plano de migração. Isso envolve desde motivações psicológicas – como lembrar que reuso é um investimento de longo prazo – até definições mais práticas sobre o processo como um todo. Nesse segundo aspecto, é fundamental a definição de papéis dentro da equipe. Eles serão usados com o intuito de focar a atenção no processo de acordo com as capacidades individuais de cada membro do grupo.

Allen e Frost (2001) definem um papel como um conjunto de habilidades e responsabilidades dentro de um time, trazendo objetividade ao processo. Como em outras metodologias de trabalho em grupo, os papéis são dinâmicos, podendo ser exercidos por inúmeros indivíduos, assim como cada pessoa pode caracterizar mais de um papel. Muitos podem ser os papéis possíveis, e quanto mais utilizado for o processo, maior a necessidade de novas especializações. Entretanto, alguns deles tendem a ser genéricos o suficiente para serem usados em inúmeros processos, e por isso serão citados aqui na tabela 2.10.

Tabela 2.10: Papéis em CBD

Arquiteto de Reuso	É o responsável por identificar e adquirir componentes. Assegura consistência do projeto e promove o valor do reuso através do Patrocinador de Reuso. Esse papel requer conhecimento sobre padrões e *frameworks*, assim como bons contatos com fabricantes de componentes.
Avaliador de Reuso	É aquele que identifica áreas para melhorias de reuso. Esse papel requer uma boa consciência sobre sistemas existentes, pacotes, banco de dados e componentes disponíveis. Esse conhecimento é necessário para ampliar o reuso, identificando e avaliando as oportunidades. Como também é responsável por difundir a prática do reuso, são necessárias habilidades interpessoais.
Bibliotecário de Componentes	É o responsável por controlar o repositório de componentes. Suas atribuições incluem verificar componentes, publicar capacidades de cada componente e gerenciar suas configurações. Esse papel requer a combinação de habilidades administrativas e técnicas, e deve ser especialista em programas de gerência de componentes.
Coordenador de Processo de Negócio	Papel responsável por demonstrar o valor dos componentes dentro de um determinado processo, através da coordenação da adequação do desenvolvimento ou da criação e novos processos. É uma posição que requer conhecimento especialista em modelagem de processos de negócio, onde é necessário compreender a relevância de cada componente de acordo com as necessidades dos usuários.
Desenvolvedor de Componentes de Dados	É aquele que modela e implementa componentes para interpretar dados armazenados. Requer um conjunto de habilidades que incluem da análise ao teste, com um particular conhecimento na linguagem de programação apropriada ao banco.
Desenvolvedor de Componentes de Negócio	É o papel que modela e desenvolve componentes de serviços para determinada área de negócios. Para isso, é necessário um grande conhecimento de práticas de negócio, assim como conhecimentos na linguagem de programação apropriada para o desenvolvimento.
Gerente de Reuso	É aquele responsável por planejar e controlar as atividades de projetos envolvendo componentes, criando e conservando as políticas de reuso.
Patrocinador de Reuso	Papel responsável por autorizar recursos para projetos de componentes. Esse papel deve ser atribuído para pessoas em postos altos da hierarquia.

.

3 HDL: CONCEITOS, EXTENSÕES E FERRAMENTAS

Linguagens de Descrição de Hardware (Hardware Description Language, HDL) são linguagens usadas para descrever as funções de um circuito eletrônico com objetivos de documentação, simulação e síntese lógica. Existem diversos tipos de HDL, mas as duas mais usadas pela indústria são Verilog e VHDL.

Uma linguagem de descrição de hardware permite uma descrição da estrutura de um projeto e sua decomposição em sub-módulos. Com essa hierarquia fica facilitada a implementação de cada módulo em separado e como funciona a comunicação entre eles. Através desse tipo de abstração fica mais fácil também a especificação com maior precisão de parâmetros dos circuitos, sem a necessidade de se conhecer com exatidão como foram construídos.

Verilog é uma HDL, padrão IEEE número 1364. Sua primeira versão foi publicada em 1995, sendo revisada em 2001. A organização por trás do desenvolvimento de Verilog chama-se Accellera e publicou uma extensão chamada SystemVerilog, com o objetivo de elevar o nível de abstração em modelagem e validação de Verilog. Adicionalmente, o padrão 1364 define uma interface de linguagem de programação (*Programming Language Interface*, PLI) que nada mais é que uma coleção de rotinas de software capazes de permitir uma interface bidirecional entre Verilog e linguagens de programação como C. (DOULOS, 2005)

3.1 Reuso em Hardware

A decomposição de um problema em partes menores é uma abordagem usual em diferentes áreas do conhecimento. Quando especificamos no campo de sistemas digitais, a decomposição é normalmente guiada por regras estruturais que objetivam esconder decisões de projeto, fazendo com que apenas as interfaces fiquem visíveis. Kission, Ding e Jerraya (1995) dividem uma metodologia de projeto estruturado em três etapas:

- Particionamento da totalidade do sistema em subsistemas;
- Síntese de cada subsistema resultante;
- Abstração de cada subsistema sintetizado para serem usados como componentes durante a síntese de sistemas em níveis mais altos da hierarquia.

Jacone e Peixoto (2001) citam três níveis de abstração para definir elementos de reuso:

- Modelos de Desempenho e Arquitetura: provêm comportamento para nodos de processadores, barramentos e interconexões que compõem uma topologia de rede.
- Modelos de Comportamento Abstrato: provêm comportamento funcional em nível algorítmico e de arquitetura de conjunto de instruções.
- Modelos Funcional e de Interface Completos: provêm funcionalidade total a nível de sinal, com fidelidade de *timing* em nível de *clock*, incluindo RTL e modelos lógicos.

Ainda segundo Jacone e Peixoto (2001), existem três tipos de informação que definem um componente reusável: documentação de reuso, modelo de componentes e suporte a verificação. A documentação de reuso é artefato produzido com o objetivo de guiar o projetista pelo processo de reutilização. As etapas que compõem esse documento especificam como um programa pode ser refinado, através de especificações de parâmetros, assim como mostram as possibilidades de verificação e teste suportados antes do processo de síntese. Todos esses relacionamentos são claramente inspirados e permitidos pelo modelo de componentes ao qual o componente pertence.

Cada um desses componentes reusáveis ou sistemas pode ser visto como um bloco, e é visto como uma forma de propriedade intectual (*Intellectual Property*, IP) e é usualmente referenciado apenas como IP. O termo IP pode designar diferentes elementos, como núcleos(*core*), blocos em nível de sistema (*System-Level Blocks*, SLB), macros e macros em nível de sistema (*System-Level Macros*, SLM) (VSIA; 2005). A abordagem baseada em projetos de IP segue a idéia que um modelo arquitetural pode ser implementado montando componentes reusáveis de hardware e software, sem distinção de origem. Essa abordagem se baseia em componentes conhecidos e testados, possibilitando que os projetistas apenas se preocupem em satisfazer seus requisitos. (WAGNER et. al., 2004) A integração destes núcleos é uma tendência que cresce cada vez mais, onde esta idéia de reuso é focada na construção funcional de componentes integráveis. (JACONE; PEIXOTO, 2001).

Pela separação da comunicação do comportamento é possível a verificação de um bloco IP com o uso de uma definição abstrata de interface. O usuário do IP pode então usar um nível de descrição mais alto para o bloco, avaliando as interações entre blocos, cobrindo a comunicação sem precisar uma análise profunda sobre a funcionalidade daquele bloco. (ROWSON, SANGIOVANNI-VINCENTELLI, 1997)

Uma metodologia de projeto baseada em blocos para ser efetiva necessita de uma extensa biblioteca de componentes para reuso. Os desenvolvedores devem, por outro lado, desenvolverem os blocos para que possam ser reutilizados. Para tanto, (KEATING; BRICAUD, 2002) citam dois princípios que devem ser seguidos:

- O bloco deve ser fácil de ser integrado à totalidade do projeto;
- O componente deve ser robusto para garantir que a integração não necessite de testes adicionais sobre as funcionalidades internas do bloco.

Especificamente quando se fala sobre reuso de blocos de hardware, Keating e Bricaud (2002) ainda citam algumas características que devem ser consideradas quanto ao projeto. Componentes devem ser projetados visando a resolução de problemas gerais, ou seja, devem ter a capacidade de serem configurados para serem aplicados em diferentes soluções. Outro ponto é quanto a seu uso em diferentes tecnologias. Os *soft*

IP devem possuir scripts de síntese que produzam resultados satisfatórios para diferentes bibliotecas, enquanto que para *hard IP* devem existir estratégias de portabilidade para o mapeamento em novas tecnologias.

Tabela 3.1: Tipos de IP

Hard	Blocos em nível de leiaute.
Firm	Uma descrição RTL acompanhada de algum planejamento de posicionamento.
Soft	Um bloco composto apenas por uma representação RTL.

Os IP podem vir em três formas distintas, como pode ser visto na tabela 3.1. A integração dos diferentes tipos tem uma ligação direta com os conceitos de reuso caixa-preta e caixa-branca. Analogamente àquela abordagem, aqui um *hard IP* não mostra seu conteúdo. No entanto, eles possuem a vantagem de terem melhores estimativas de desempenho, potência e área. (WAGNER et. al., 2004)

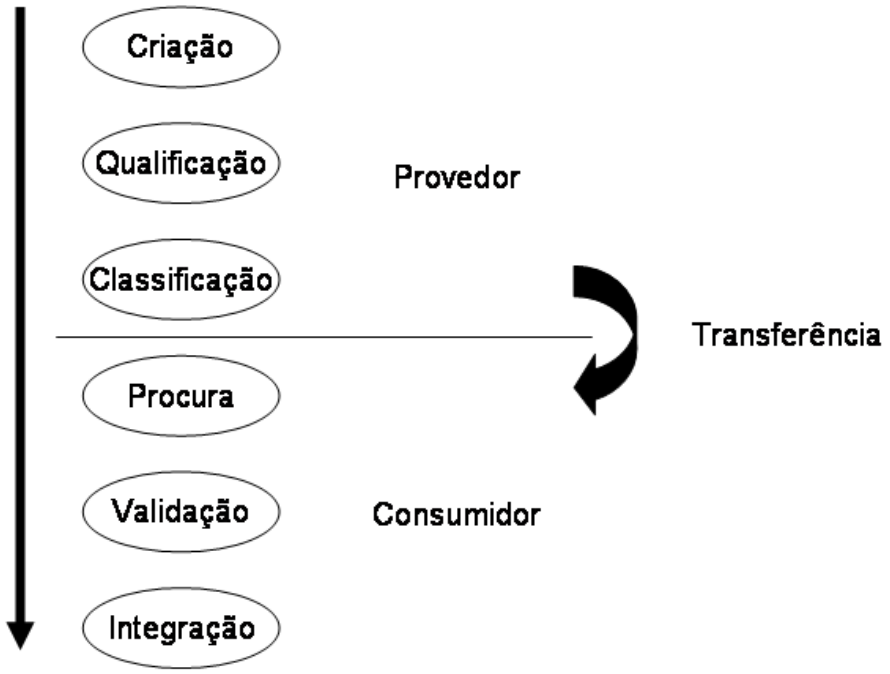


Figura 3.1: Processo de Reuso de IP (WAGNER et. al., 2004)

A derivação de um IP só pode ser usada quando da disponibilidade do código-fonte (HDL para hardware, linguagem de programação para software), possibilitando uma alteração direta. Esse tipo de modificação é bastante utilizado em situações onde é necessária uma adaptação de interfaces. Para os casos de *hard IP*, usam-se adaptadores

de interfaces, também chamados *wrappers*. *Wrappers* encapsulam componentes reusáveis em nível de arquitetura, com o objetivo de dar suporte a abstrações funcionais. Com isso, asseguram interoperabilidade e independência de tecnologia.

Tabela 3.2: Etapas do Fluxo de Desenvolvimento de IP

Criação e Qualificação	Componentes devem ser criados de acordo com diretrizes visando reuso. É aconselhável o uso de padrões de linguagem e projeto. Os produtores dos componentes devem seguir uma metodologia e utilizar tecnologias que garantam as qualidades esperadas pelos consumidores.
Classificação e Busca	Os consumidores devem encontrar blocos IP capazes de satisfazerem suas necessidades através de mecanismos *online* de busca. Fica a cargo dos produtores desses blocos uma efetiva classificação de seus produtos por meio de critérios adequados.
Integração e Validação	A validação de um componente ocorre em dois tempos. No primeiro, ele é testado em ambientes que possibilitem sua simulação, para serem posteriormente integrados ao projeto. Então são novamente testados e avaliados em definitivo.

3.2 Extensões em VHDL

Hansen, Bringmann e Rosenstiel (1999) dividem as abordagens de projeto para HDL em extensões orientadas a objetos e projetos estruturais.

Uma linguagem OO é tecnologia fundamental para computação reativa, o que é um nível de abstração mais alto do que as HDL. Jacone e Peixoto (2001) afirmam que o uso desse tipo de modelo em estágios iniciais do processo de desenvolvimento é extremamente útil para verificar a corretude de decisões de alto-nível antes de se investir esforço em projetos mais detalhados. Por isso, quando se fala em extensões para HDL, uma das abordagens mais utilizadas é a adição de conceitos OO, como em (ASHENDEN; WILSEY; MARTIN, 1998) e (BENZAKKI; DJAFRI, 1997). A abordagem mais utilizada e que é citada na seção 3.2.1, sendo usado como base para o padrão do Comitê de Padrões para Automação de Projetos da IEEE.

Quando se fala em projetos estruturais, temos exemplos em Preis et. al. (1997), onde propuseram uma metodologia de reuso baseada em VHDL independente de plataforma, através de um ambiente utilizando hipertexto. Uma outra abordagem, baseada em RAD, utilizando ferramentas automatizadas e repositórios de componentes distribuídos foi o MYA (*Model of the Year Architecture*) (PRIDMORE et. al.; 1997). O programa RASSP (*Rapid Prototyping Of Application Specific Signal Processors*) introduziu esta abordagem que possuia foco na evolução de aplicações embarcadas de larga escala. O MYA era formado por três elementos básicos: a arquitetura funcional, componentes encapsulados para serem compartilhados em bibliotecas e guias e constantes de projeto.

Indiferente a abordagem escolhida, um dos maiores problemas para extensões em HDL é a portabilidade. Isto é causado quando fabricantes criam extensões próprias para seus códigos sem nenhum tipo de padronização. Isto pode ocorrer desde tipos de dados

a funções, fazendo com que o código desenvolvido fique restrito a quem o criou. (MCFARLAND, 1993) Por isso, mais uma vez, fica clara a necessidade da dupla metodologia e automação.

3.2.1 Objective VHDL

Barna e Rosenstiel (1999) propuseram uma extensão a VHDL baseada em orientação a objetos, com o claro intuito de alcançar reuso, a que chamaram Objective VHDL. A idéia consistia em introduzir os conceitos de OO, como classe, herança, definição e polimorfismo de tipos. A idéia é que *wrappers* são criados em Objective VHDL, encapsulando IP em VHDL nativo.

Os objetos de entidade OVHDL são baseados na idéia que componentes de hardware estrutural representado por uma entidade de projeto VHDL podem ser considerados como objetos no contexto OO. O modelo OVHDL é centrado em *firm* IP, que são armazenados em bibliotecas de objetos reusáveis. A estrutura de cada objeto é visualizada na figura 3.2 e representa o reuso caixa-cinza, onde são permitidas modificações controladas, que não afetam diretamente o código reusável (BARNA; ROSENTIEL, 1999).

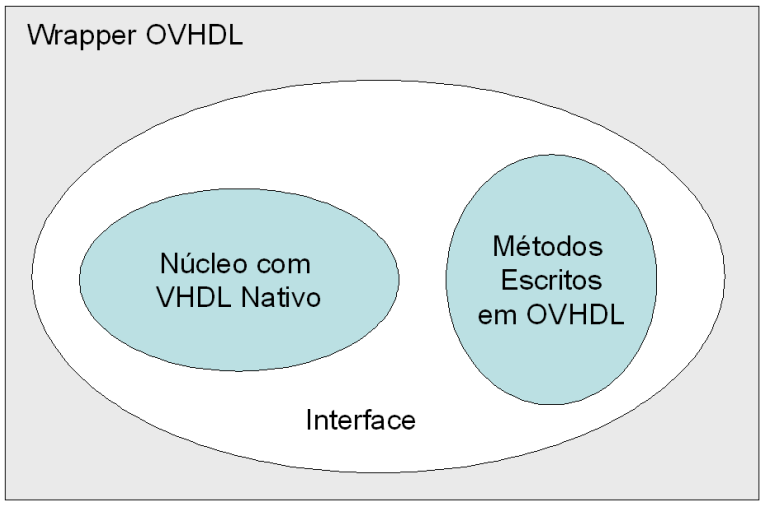


Figura 3.2: Objetos de Reuso OVHDL

OVHDL foi projetada com ênfase em possibilitar a extensão de síntese para o código orientado a objetos, causando a inclusão das etapas de síntese na metodologia de projeto, sempre se preocupando em preservar a habilidade de síntese em eventuais modificações na linguagem. A extensão de uma linguagem OO com conceitos adequados de comunicação e com uma metodologia que define como uma comunicação é especificada ou refinada, requer uma correspondência conceitual das extensões com os conceitos fundamentais da linguagem. A especificação de um código de comunicação (um protocolo, por exemplo) assim como dos componentes que utilizarão esta comunicação devem ser especificados de maneira ortogonal. Isto requer uma clara separação entre os códigos, provendo então a possibilidade do reuso em nível de composição e de armazenamento em repositórios. (PUTZKE-RÖMING; NEDEL, 2000)

Essa abordagem está claramente preocupada em todos os conceitos que envolvem comunicação, e citam a necessidade de um processo completo de projeto, capturado a

partir de uma mesma metodologia de seu início a seu fim. Para isso, devem ser incluídas especificações e refinamentos na comunicação, pois são requisitos fortemente relacionados a questões de projetos de linguagem. Complexidades excessivas devem ser evitadas, fazendo com que técnicas de modelagem e especificação baseiem a linguagem numa visão de aplicabilidade ortogonal, pois integrações não-ortogonais potencializam os problemas de incompatibilidade. (PUTZKE-RÖMING; NEDEL, 2000)

Algumas ferramentas foram produzidas para trabalhar com esse paradigma, como o RMS (*Reuse Management System*), que é um ambiente para armazenamento, adaptação e recuperação de objetos de reuso.

OVHDL não foi eficiente ao relatar seus resultados pela falta de suporte a ferramentas que estendam a linguagem, assim como bibliotecas com protocolos compatíveis para a comunicação. Mesmo assim, provou que a interação entre objetos pôde ser modelada de maneira mais eficiente. Putzke-Röming e Nedel (2000) ainda frisam a importância de uma biblioteca que disponibilize o maior número de classes de componentes e protocolos de comunicação.

3.2.2 JHDL

JHDL é um método de descrever componentes e suas conexões em um circuito lógico digital usando classes Java para a construção das estruturas destes circuitos. O conjunto de classes Java tem sua funcionalidade dividida em duas áreas: simulação de circuitos e suporte em tempo de execução de CCM (*Configurable Computing Machines*). JHDL inclui um ambiente com um conjunto de ferramentas para simulação, testes e *debugging*. Essas ferramentas interagem com o modelo de circuitos JHDL através de API. Foi projetado para dar suporte a reconfiguração em tempo de execução, tanto parcial quanto global, escondendo detalhes de configuração do usuário. (BELLOWS; HUTCHINGS, 1998)

Baseado em Java, e por isso em OO, JHDL usa os conceitos de construtores e destrutores para descrever as estruturas dos circuitos e suas modificações no decorrer do tempo. Essas estruturas podem ser tanto de software quanto de hardware, co-existindo no mesmo projeto e sendo possível sua execução e simulação pelo mesmo ambiente. Assim, extensões das classes JHDL são executadas em hardware, enquanto que trechos escritos puramente em Java rodam sobre a máquina virtual. Os circuitos são criados através de construtores para o objeto JHDL correspondente com a passagem de objetos de ligação como parâmetros que serão ligados a esse circuito. Na Figura 3.3, vemos a criação de uma porta AND com duas entradas correspondendo a **`a`** e **`b`** e tendo **`c`** como saída. (HUTCHINGS et al., 1999).

```
new and_o(a, b, c)
```

Figura 3.3: Exemplo de Criação de Objeto JHDL

Atualmente, o ambiente conta com ferramentas para depuração gráfica, gerador de esquemáticos e um posicionador gráfico, sendo possível sua aquisição na forma de ferramenta ou de código-fonte. (BYU, 1998).

3.3 Ferramentas

O desenvolvimento de projetos nas diferentes áreas da computação é baseado tradicionalmente na abordagem especifica-projeta-constrói-usa. Esse paradigma é normalmente apoiado por um conjunto de ferramentas que atuam em todas essas fases do ciclo de vida, incluindo desde gerenciamento de projetos até compiladores, passando por ferramentas de análise, projeto e testes. A essas ferramentas é dado o nome de CASE (*Computer-Aided Software Engineering*). Fugetta (1993) divide, mais especificamente, essas ferramentas em três categorias:

- Ferramentas: apóiam tarefas individuais;
- *Workbenchs*: apóiam fases ou atividades de processo e
- Ambientes: conjunto de workbenches e/ou ferramentas que apóiam o todo ou uma grande fatia do processo.

O uso de ferramentas de maneira isolada não traz, entretanto, garantia de sucesso. Deve-se igualmente agregar valor ao projeto que está se construindo através de metodologias. Está entre as grandes necessidades de hoje em dia a preocupação com a eliminação de todo o trabalho que não acrescente valor aos produtos. Um dos paradigmas que se dispõem a isso é o Desenvolvimento Rápido de Aplicações (RAD).

3.3.1 RAD

O Desenvolvimento Rápido de Aplicações (*Rapid Application Development*, RAD) é uma técnica de programação que permite um desenvolvimento veloz de software. As implementações de RAD podem ser em forma de ferramentas visuais para desenvolvedores, ou *frameworks* de geração de código através de "*wizards*". Soluções desenvolvidas via técnicas de RAD podem não ser soluções ótimas para determinados conjuntos de problemas. Entretanto, muitas aplicações não têm como prioridade a melhor solução, mas sim uma boa o suficiente e factível em um determinado prazo.

RAD surgiu com o objetivo de ser uma alternativa eficaz para reduzir o tempo de desenvolvimento de software. Essa promessa não se resumia a diminuição do *time-to-market* mas também a diminuir os custos dos métodos tradicionais e a entregar produtos que melhor alcançassem seus requisitos. Para isso, os desenvolvedores criavam diagramas de requisitos usando ferramentas automáticas ao invés de processos manuais. Ainda, incentivavam as entrevistas coletivas, onde usuários e projetistas discutiam as aplicações em conjunto, minimizando a distância desses dois papéis. (REILLY; CARMEL; 1995) O primeiro grande difusor de RAD foi Martin Mcmillan, quando em 1991 lançou um livro chamado *"Rapid Application Development"*. De lá para cá RAD tem alternado altos e baixos e feito legiões de seguidores e de inimigos. Os defensores (HOWARD, 2002) argumentam que os princípios modernos de qualidade, como se adequar aos propósitos, evitar perdas, alcançar os objetivos na primeira vez, responsabilizar-se individualmente sobre a qualidade e ir de encontro aos requisitos do cliente, são inerentes a RAD.

As técnicas RAD são centradas em uma abordagem de negócio, com o claro objetivo de quebrar as barreiras entre a tecnologia de informação e as comunidades de negócios. Todas as perspectivas são voltadas a ajudar os desenvolvedores a obter de maneira mais eficiente os requisitos do sistema. O time de desenvolvimento é responsável por organizar os componentes, controlar o desenvolvimento e verificar requisitos que não foram totalmente contemplados. Para isso, a comunidade de usuários

provê a informação que será usada para essa validação, representando assim uma cooperação que tem como resultado uma melhor compreensão de ambos os lados de como é o sistema em sua totalidade. (REILLY; CARMEL, 1995) (HUGHES, 2000) (HOWARD, 2002) Howard (2002) divide RAD em duas formas de abordagem: *Rapid Program Development* (RPD) e *Rapid System Development* (RSD).

RPD começa com a especificação do projeto. O objetivo então é implementar essa especificação no menor tempo possível. Esse paradigma acaba necessitando de programadores com grandes habilidades, pois se defrontam com a necessidade de rapidamente entender uma especificação e produzir código de alta-qualidade. Não é necessário nenhum tipo de comunicação com os usuários-final.

RSD começa com um núcleo que pode ser uma nova idéia ou a manutenção de um sistema já existente. Para isso, é preciso que aconteça uma grande interação entre a equipe de produção e os usuários, pois não existe nenhuma especificação inicial. Durante o progresso do projeto, protótipos devem ser desenvolvidos para análise e definição de escopo, tomando-se os requisitos. Um projetista usando essa abordagem precisa então possuir habilidades nas áreas técnica e social. Howard (2002) vê o processo de uma forma muito mais caótica, tendo nas técnicas RAD a tentativa de trazer ordem ao incerto. O sucesso do projeto acaba passando pela habilidade do time de transformar um conceito inicial em um sistema que agregue valor a uma operação de negócios.

Zubeck (1997) afirma que para alcançar as vantagens oferecidas por RAD, um desenvolvedor deve encontrar uma combinação entre a ferramenta e a necessidade de negócio. No entanto, quanto mais os desenvolvedores distorcem os componentes que acompanham a ferramenta para alcançar seus requisitos, menos adequados eles se tornam para a construção de sistemas maiores.

O principal problema de implementações RAD no universo OO é sua facilidade exclusiva a usuários experientes. Esses usuários memorizaram a hierarquia de objetos e suas relações, enquanto que novos usuários normalmente sentem-se perdidos para encontrar determinadas classes que satisfaçam funcionalidades procuradas. Esse problema é potencializado se o fornecedor da ferramenta não fizer um esforço para manter as hierarquias consistentes quando da criação de novas versões do ambiente de desenvolvimento. A biblioteca de componentes deve documentar quando os componentes sofrem adaptação, mostrando a distância percorrida por essa evolução, desde suas funcionalidades nativas. (ZUBECK, 1997)

Zubeck (1997) afirma que algumas ferramentas RAD reduzem ou eliminam a necessidade de programação com algum tipo de código-fonte. Adiciona ainda, que experimentos com ferramentas RAD demonstraram serem mais eficazes para configurações e manipulações de ordem visual, como construção de diagramas ou consultas *drag-and-drop*. O autor cita, ainda, características fundamentais para uma ferramenta RAD eficaz:

- Funcionalidades que diminuam ou eliminem linguagens de programação ou script.
- Uma biblioteca extensível que guarde diferentes variações dos componentes.
- As *widgets* devem ser fáceis de serem entendidas.

- A disposição visual dos projetos deve ser feita de modo que o programador trabalhe em módulos lógicos visualizados através de uma organização modular, facilitando a manutenção de seu trabalho.

3.3.2 IDE

Um ambiente de desenvolvimento integrado (*Integrated Development Environment*, IDE) é um software que possui como função ajudar programadores a desenvolver programas. Normalmente consiste em um conjunto de ferramentas que incluem: um editor de código-fonte, um compilador/interpretador e um depurador, empacotados como uma aplicação única. Outras funcionalidades também estão presentes em vários desses ambientes, como ferramentas de construção automática – como no caso de GUI – , sistemas de controle de versões e inspetor de objetos. Esse tipo de ambiente pode ser específico para uma linguagem (como Visual Basic) ou independente de linguagem (como Eclipse). (VAUGHAN-NICHOLS, 2003) (GEER, 2005)

A idéia de IDE consiste em um mesmo programa onde todo o desenvolvimento é feito, em contraponto a ambientes distintos que se comunicam através de linhas de comando. Essa abstração defende que com essa unidade o tempo de aprendizado de uma determinada linguagem é reduzido, aumentando a produtividade do desenvolvedor. Esse tipo de programa provê um grande número de funcionalidades que incluem criação, modificação, compilação, implantação e depuração de software. Com essa integração, é possível, por exemplo, que um código seja compilado enquanto está sendo escrito, provendo um feedback instantâneo para erros de sintaxe.

Algumas IDE são capazes de proporcionar a seus usuários a confecção de novas aplicações a partir da movimentação de blocos de construção programáveis ou nodos com códigos-fonte para criação de *flowchart* ou diagramas estruturais capazes de serem compilados ou interpretados. Muitos dos diagramas são criados a partir de construções conhecidas como as definidas em UML.

Martin (1991) cita algumas características que formam o núcleo da metodologia RAD, que incluem: prototipação, desenvolvimento interativo, comunicação com o cliente, times e ferramentas. Com exceção da última característica, as outras são encontradas em várias das metodologias chamadas hoje em dia de “ágeis”, como XP de Beck e Fowler (2000). Essa preocupação com ferramentas fez com que vários ambientes fossem desenvolvidos, sendo apresentados nesse trabalho através de três grupos representativos.

3.3.3 Estudo de Ferramentas

No escopo desse trabalho, algumas ferramentas e ambientes foram analisados para fins de comparação e inspiração para o ambiente proposto. O termo IDE é usado em contraste a ferramentas de linha de comando como vi e make, mesmo que alguns programadores mais entusiasmados considerem o ambiente UNIX como um ambiente de desenvolvimento. As três primeiras seções mostram ambientes de desenvolvimento de linguagens de programação em geral, enquanto que as três seguintes apresentam soluções na área de HDL.

3.3.3.1 Ambiente Textual

Este tipo de ambiente é em geral o mais simples, e por isso, muitas vezes utilizado por iniciantes na tecnologia, tendo como alguns de seus exemplos o DevC++ (BLODSHED, 2000) e o JCreator (XINOX, 2000). Uma exceção a essa regra é a ferramenta IntelliJ (JETBRAINS, 2001), que possui uma interface mais arrojada, disponibilizando maiores recursos, principalmente nas áreas de gerência de projetos e *refactoring* de código.

3.3.3.2 Ambiente Gráfico

Os primeiros expoentes de sucesso deste tipo de IDE RAD são o Delphi (BORLAND; 1995) e o Visual Basic (MICROSOFT, 1991), surgidos nos anos 90. Em 1996, o surgimento do Windows 95 fez com que 70% de todos os programas do tipo desktop fossem construídos a partir de Visual Basic (VB). (ZUBECK, 1997)

Este tipo de IDE é organizada ao redor de objetos de tela que possuem o seu próprio universo de objetos com funcionalidades próprias. Vendedores autônomos criam seus componentes e os distribuem através de pacotes. No caso de VB, por exemplo, em arquivos com extensões .vbx ou .ocx. (ZUBECK, 1997) Sua arquitetura é baseada em um conjunto de APIs conectadas com editores, compiladores e depuradores.

Algumas delas são otimizadas para determinadas tecnologias, enquanto que outras como o Netbeans[2] possibilitam a interconexão com outras ferramentas que se acoplem a sua base. Ele também pode ser usado para desenvolvimento em linguagens como C++, HTML ou XML. (VAUGHAN-NICHOLS, 2003)

Esse tipo de ambiente é composto por um núcleo de desenvolvimento onde todas as suas funcionalidades estão disponíveis através do mesmo programa desde sua instalação. São compostos por áreas de programação (textual e gráfica) e pelas ferramentas de compilação, documentação e depuração.

3.3.3.3 Ambiente com arquitetura de plug-in

IBM encabeçou o projeto de um ambiente multi-companhia e de código aberto, chamado Eclipse. Sua idéia era usar essa ferramenta para o desenvolvimento de qualquer tipo de linguagem, como Java, C++, Cobol ou HTML. (VAUGHAN-NICHOLS, 2003) Para isso, comprou junto à empresa Object Technology International a tecnologia básica que está por trás do ambiente. A IBM desenvolveu a plataforma Eclipse de acordo com os padrões de especificação para interoperabilidade publicados pela OMG. Atualmente a gerência da plataforma está a cargo da Eclipse Foundation, o que não impede que desenvolvedores que não sejam membros possam construir suas próprias soluções baseadas nessa plataforma. A plataforma Eclipse é livre, enquanto seus *plug-ins* podem ser tanto gratuitos quanto pagos. Para comparação, IDE proprietárias como JBulider, JDeveloper ou IntelliJ podem custar mais do que US$ 3500. (GEER, 2005)

---

[2] O Netbeans é originário de um projeto chamado Xelfi, criado na República Tcheca na metade dos anos 90. Seu criador, Roman Stanek fundou uma companhia que chamou de Netbeans com o intuito de vender essa tecnologia, sendo adquirida pela Sun em 1999. Atualmente, o desenvolvimento voltou para o grupo checo.

A arquitetura da plataforma Eclipse, como ilustrada na figura 3.4, foi construída com um núcleo central que dá amplo suporte a inúmeros *plug-ins* que têm como objetivo estender suas capacidades, tanto em nível de aplicações quanto de linguagens de programação. As ferramentas acopladas à plataforma operam normalmente sobre os arquivos do ambiente. Estes arquivos podem vir de um repositório de time, que está associado a trabalho colaborativo, sendo posicionados no espaço individual do usuário. Os *plug-ins* disponíveis são descobertos a cada novo recomeço da plataforma, quando são lidos seus arquivos de manifesto e é então criado um registro em memória desses *plug-ins*.

No universo Eclipse, o termo *Workbench* é usado para tratar a interface gráfica como um todo e é criado com o uso da *Standard Widget Toolkit* (SWT), uma biblioteca de propósitos geral para construção de componentes visuais. Sua maior contribuição é uma melhora de desempenho em comparação às bibliotecas padrão da plataforma Java (AWT/Swing). Enquanto plataformas como o Netbeans podem rodar de maneira nativa em qualquer plataforma que disponibilize a máquina virtual Java, o Eclipse está preso às limitações impostas pela SWT.

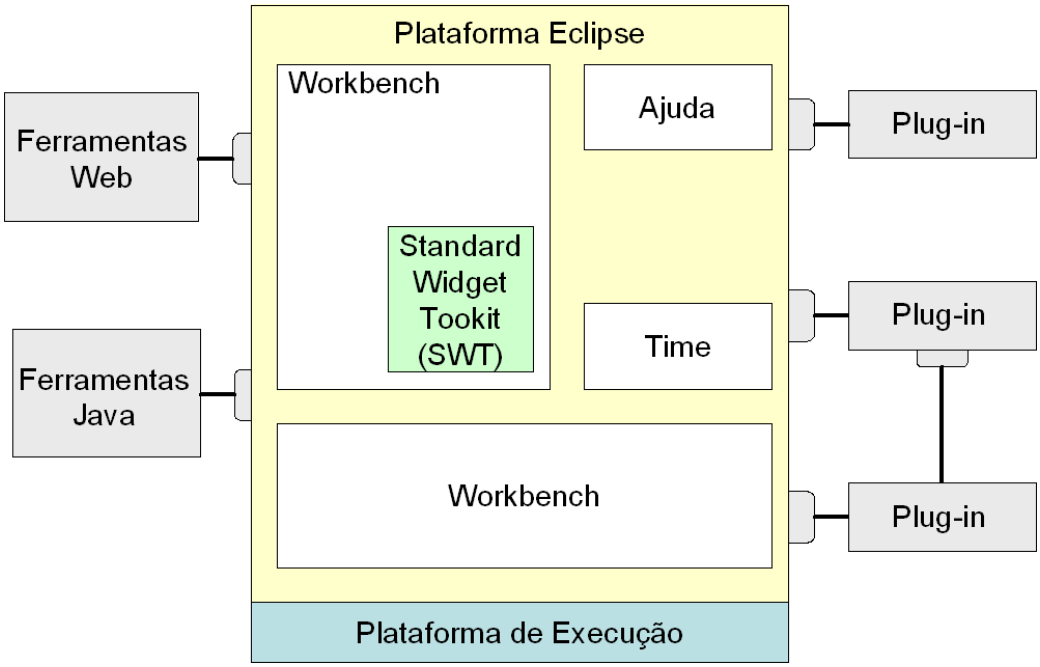


Figura 3.4: Arquitetura Eclipse

3.3.3.4 Visual HDL

A ferramenta da Summit (SUMMIT, 1997) é uma solução que gerencia projetos de hardware em diferentes aspectos, sendo uma plataforma que possui ferramentas para projeto, integração e verificação. Ela transforma código Verilog ou VHDL (informação puramente textual) em informação gráfica. O desenvolvimento se baseia em importar para a ferramenta uma descrição existente, e depois transformá-la explicitamente em uma das opções gráficas possíveis: diagrama de blocos, diagrama de estados, máquinas de estados ou fluxogramas. A transformação então é vista em uma nova janela, como pode ser visto na figura 3.5.

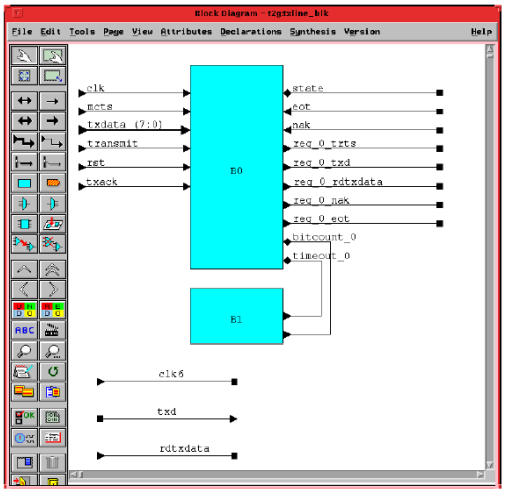


Figura 3.5: Screenshot Visual HDL/Summit

Visual HDL funciona bem na gerência e compartilhamento de projetos visuais, entretanto deixa de ser eficiente quando o objetivo é disponibilizar o projeto como um todo. Para tanto, soluções como SOS (CLIOSOFT, 2003) foram criadas para tentar minimizar esse aspecto.

A Summit lançou em 2005 uma nova versão da plataforma, chamada de Visual Elite, que melhorava o gerenciamento dos projetos, tanto em suas formas textuais quanto gráficas, assim como ampliava o nível de abstração com a introdução da possibilidade de uso de SystemC. Com isso, é capaz de gerenciar projetos através da integração de SystemC com HDL. Por ter um ambiente integrado com diferentes ferramentas, era capaz de tarefas mais sofisticadas, como fazer anotações no projeto gráfico a partir de dados coletados por simulações através de *waveforms*.

3.3.3.5 HDS/Mentor

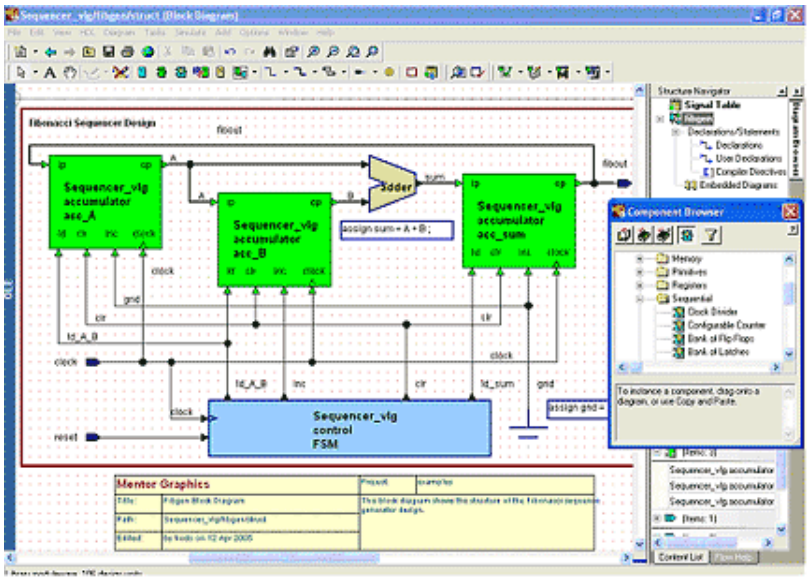


Figura 3.6: Screenshot HDS/Mentor

A Mentor possui um vasto *framework* de ferramentas que tentam cobrir todo o fluxo de projeto de CI. Na etapa que trata de HDL, ela possui uma solução que também possui uma abordagem visual, chamada HDS. Esse software trabalha com componentes gráficos que são adicionados e manipulados em uma área de trabalho, sendo possível sua edição através desta área ou de outras áreas de sua GUI como um inspetor de componentes. Uma amostra pode ser vista na figura 3.6. Cada projeto gera um código VHDL que é então disponibilizado automaticamente em futuros usos da ferramenta.

3.3.3.6 Silicon Integration Initiative

A *Silicon Integration Initiative* (Si2) é uma organização composta por indústrias da área de semicondutores e ferramentas de EDA. Antigamente conhecida como *CAD Framework Initiative* (CFI) é o órgão por trás de iniciativas como OpenCores e OpenAccess.

A OpenCores (OPENCORES.ORG, 1999) é uma organização que tem como objetivo o desenvolvimento de hardware, seguindo os mesmos padrões dos projetos de software livre. É aberto a participantes de diferentes níveis de conhecimento (estudantes, profissionais) assim como a empresas. O foco é em blocos de desenvolvimento (cores) e a plataforma web é a base para a troca desses blocos entre os projetistas, como em Source Forge (OSTG, 2001).

A documentação dos projetos é feita através da própria estrutura hierárquica de diretórios que o sistema disponibiliza, com a possibilidade ainda, da criação de fóruns para a troca de experiências e esclarecimento de dúvidas. Esta estrutura de projetos o disponibiliza abertamente (reuso caixa-branca) através dos diferentes arquivos que compõe o projeto.

Open Access é uma proposta de interoperabilidade entre dados e ferramentas de um projeto de CI, onde a comunicação entre as ferramentas é feita através de uma API implementada em C++. A proposta surgiu no final de 1999 e tem como objetivo diminuir o impacto de integração entre as diferentes ferramentas, juntamente com seus formatos de arquivo e bases de dados, através da construção de fluxos de projetos com a incorporação de diferentes programas. A comunidade envolvida no desenvolvimento do Open Access inclui empresas como Cadence, Synopsys, IBM, HP, Motorola e Sun. (SI2, 2005)

A abordagem do Open Access tenta reduzir a tradução de dados entre as diferentes etapas do fluxo de desenvolvimento. Isso envolve a prevenção de perda de dados nas transferências entre ferramentas, assim como ela também tenta gerenciar de forma inequívoca os conflitos de representação de dados. Esse processo acaba centralizando o acesso aos dados, assegurando a completude da informação. A arquitetura Open Access acaba sendo, então, fortemente baseado em banco de dados. (SI2, 2005)

.

4 PROCESSO MACANUDO

O ciclo de desenvolvimento baseado em componentes proposto pelo processo Macanudo segue as idéias citadas nos capítulos 2 e 3, e pode ser acompanhado pela figura 4.1. A criação do processo Macanudo passou pela definição de alguns fatores, que serão elencados nas seções que compõe esse capítulo, onde podemos destacar a definição de um modelo de componentes e de um ambiente de desenvolvimento, que também leva o nome Macanudo.

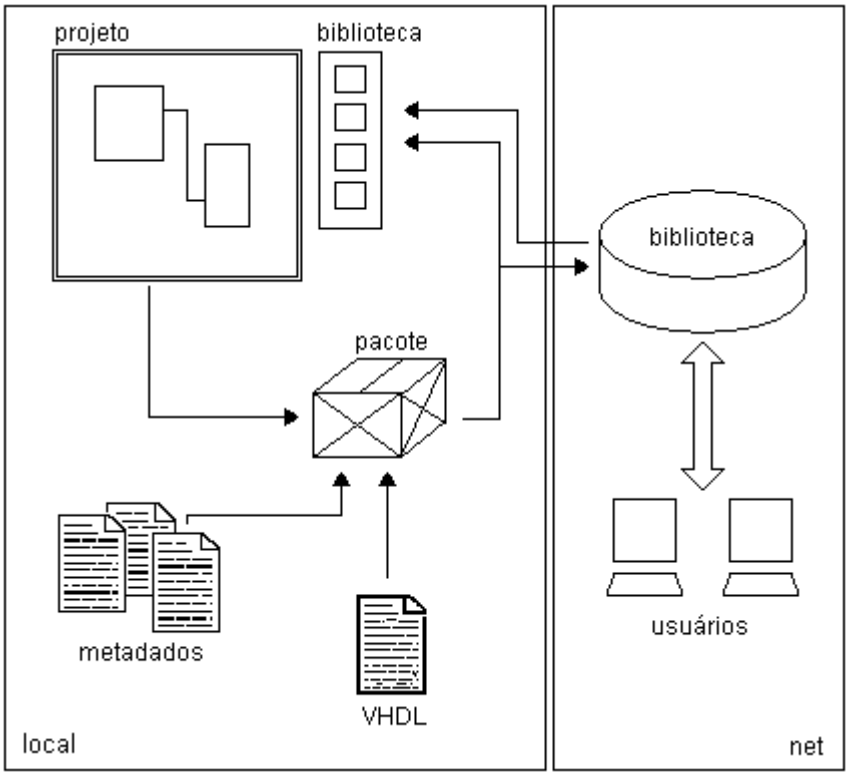


Figura 4.1: Ciclo de Desenvolvimento Macanudo

Os projetos são construídos através de uma IDE que disponibiliza os componentes da biblioteca local através de uma interface gráfica. Esse ambiente funciona como base para a instanciação, edição e composição dos componentes. Ao finalizar a construção de um novo componente, cria-se o pacote JAR, onde se localizam todas as informações

visuais de projeto, os códigos-fonte dos componentes e todo o metadado que o projetista escolher disponibilizar. Com a fase de empacotamento completa, esse componente pode então ser reutilizado para popular a biblioteca local, assim como pode ser disponibilizado em um repositório *online*, onde poderá ser usado, testado e reutilizado por terceiros. O escopo do protótipo apresentado neste capítulo está localizado no universo local deste ciclo de desenvolvimento. A parte distribuída teve algumas funcionalidades implementadas e pode ser vista no apêndice A.

A primeira definição a ser feita é sobre a natureza de cada componente. Um componente é uma entidade capaz de executar um conjunto de operações, funcionando como uma caixa-preta ligando seus aspectos composicionais aos computacionais. Um componente pode representar uma operação simples como adição ou um sistema complexo de comunicação envolvendo procedimentos de *handshaking*. A correspondência dos componentes se dará, então, a um projeto produzido por uma entidade externa ou por um sub-sistema resultante de sessões anteriores de projeto.

4.1 Modelo de Componentes Macanudo

O modelo de componentes genérico define alguns padrões. O modelo apresentado nesse trabalho, chamado Macanudo, foi criado preocupando-se com os aspectos de interfaces, nomenclatura, metadados, interoperabilidade, adaptação e evolução, composição e empacotamento e implantação. Uma breve descrição desses elementos básicos pode ser visto na tabela 2.5. O modelo deve vislumbrar a implementação possuindo também padrões especializados para a descrição de funcionalidades especificas a um certo domínio, necessárias para aplicações dentro de um determinado escopo.

4.1.1 Nomenclatura

Chamamos de colisão de nomes (*name clashes*) quando dois componentes distintos possuem o mesmo nome. Esse tipo de episódio deve ser evitado dentro do projeto. Para tanto, é necessário um esquema de padronização de nomes dentro do modelo de componentes. Existem duas grandes abordagens sendo utilizadas atualmente. A primeira trabalha com identificadores únicos e a segunda com uma hierarquia de nomes.

- **Identificadores Únicos:** Identificadores únicos são gerados por ferramentas dedicadas que usam uma combinação de dados específicos que garantem a unicidade de cada identificador gerado.
- **Hierarquia de Nomes:** Hierarquia de nomes são garantias de nomes únicos através de um modelo *top-down*, vinculado a uma autoridade global de registro de nomes.

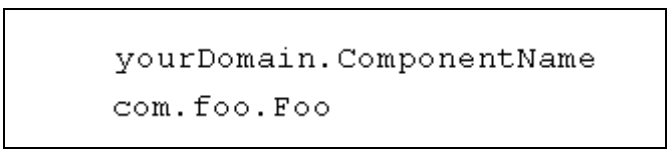


Figura 4.2: Nomenclatura Componente Macanudo

Este modelo utiliza a segunda abordagem para nomear seus componentes. Ela é inspirada na hierarquia sugerida para os componentes JavaBeans (SUN, 1997), como mostra a figura 4.2. Todo o projeto está diretamente ligado a uma instituição que possui um endereço de domínio na Internet. Usa-se essa primeira informação como identificador, sendo seguida da informação de projeto e então o nome do componente.

4.1.2 Metadados

Segundo Hillmann (2003), os metadados tem origem nos bibliotecários que criavam lista de itens sobre os livros que catalogavam. O radical grego "meta" era usado para demonstra idéias de "com, próximo ou depois", sendo usado depois com significado de "transcendental, além do natural". Em nosso universo, metadado acaba sendo a informação da informação. Um registro dessa natureza é um conjunto de atributos e elementos necessários para descrever o recurso em questão.

Metadado é toda informação sobre os componentes, suas interfaces e suas relações que disponibilizam informação para ferramentas de composição e invocação remota. O modelo de componentes deve especificar como os meta dados são descritos e como podem ser encontrados, deixando a cargo da implementação o provimento de serviços para a obtenção dessa informação. Nesse modelo, o XML que identifica o componente possui o papel de metadado, servindo de link para os diferentes tipos de informação anexada ao pacote do componente.

O uso de XML para descrição de componentes de hardware mostra-se interessante devido a alguns fatores, como citados em (GRASSENS, RANGANATHAN E FECHSER; 2005):

- XML é um padrão que vem se firmando na indústria, pela facilidade de compartilhamento de informação entre diferentes organismos. É facilmente visualizado por ferramentas proprietárias ou por aplicações web. A leitura de arquivos XML é eficiente e possui suporte a diferentes linguagens e infra-estruturas de software, sendo independente de plataforma.
- XML é uma linguagem que suporta hierarquia. Muitos CI são projetados através de hierarquia de componentes.

Uma descrição de hardware é formada por dados internos do componente – como largura de barramento –, arquivos de especificação – VHDL –, arquivos de uso do ambiente – figuras – e por toda a informação adicional que o fabricante do componente achar importante. Essa informação – metadados – pode incluir desde documentação de projeto, a manuais de instalação ou mesmo exemplos de uso. A descrição de componente para uso no modelo e na ferramenta pode ser vista a partir de uma XMLSchema, representada na figura 4.3.

A descrição XML do componente acaba sendo dividida em quatro grupos. No primeiro temos o atributo de nome, seguindo o padrão definido na nomenclatura do modelo. O segundo trata da interface, ou seja, suas portas. Os terceiro e quarto tratam dos arquivos que foram empacotados junto com o componente, trazendo toda aquela informação extra sobre documentação e elementos do ambiente.

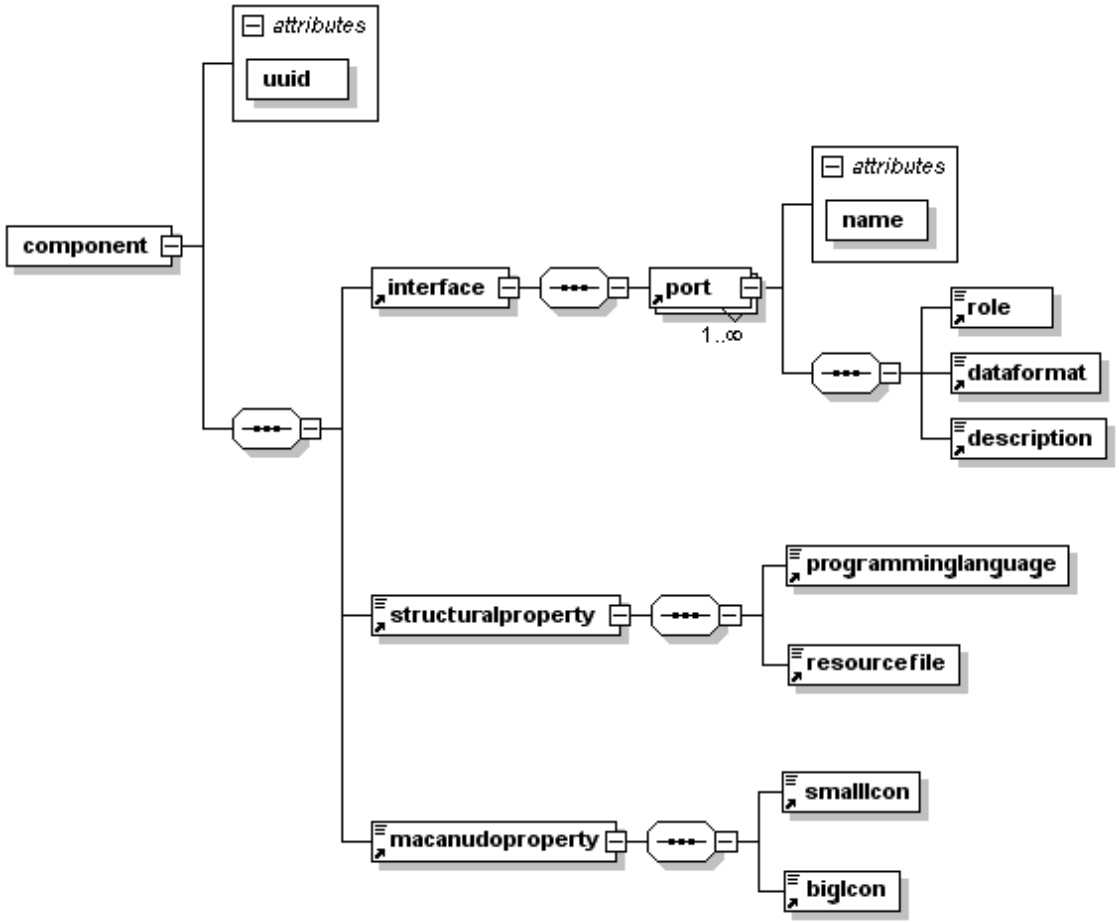


Figura 4.3: XMLSchema do Componente

Cada porta possui um nome que a identifica dentro da interface. Ela ainda possui mais três atributos, definidos a partir de sugestões de padronização da VSIA[3], que demonstrou ser um dos padrões mais aceitos. A representação gráfica dos papéis não foi implementada nesta primeira versão do Macanudo, mas já aparece presente na definição de sua estrutura.

- **role:** Indica o papel que aquela porta possui. Pode possuir um dos seguintes valores: IN, OUT, IN/OUT, INITIATOR, RESPONDER, PRODUCER e CONSUMER.
- **dataFormat:** Informa o formato do dado daquela porta.
- **description:** Espaço destinado a uma descrição textual sobre o funcionamento e objetivos daquela porta.

As propriedades estruturais (“structuralproperties”) do XML contêm a linguagem de descrição daquele componente e os arquivos de recurso (“resourcefiles”) empacotados

[3] A VSIA (*Virtual Socket Interface Alliance*) é um grupo que trabalha na área de reuso de projeto, adotando padrões de interfaces de ferramentas e projetos, e em boas práticas de documentação. (VSIA, 2005)

juntos. Os tipos possíveis são de liberdade de utilização do usuário em tempo de criação de seu pacote. Entretanto, alguns tipos (“types”) são indicados para utlização:

- **“source code”:** Qualquer arquivo fonte para o funcionamento do componente.
- **“macanudo source”:** Arquivo de descrição de componente criado dentro do ambiente.
- **“online reference”:** Uma descrição do componente, ou documentação, ou ainda exemplo, disponível na web.
- **“manual”:** Documentação incluída no pacote.

A última seção do XML mostra as propriedades usadas pelo ambiente para representar um determinado componente. Em um primeiro momento, estão apenas representadas as figuras disponíveis na barra de ferramentas, em dois tamanhos distintos: pequeno (“smallIcon”) e grande(“bigIcon”).

4.1.3 Empacotamento e Implantação

Um modelo de componentes deve ser capaz de descrever como componentes são empacotados de modo a poderem ser implantados independentemente. Um componente é implantado, ou seja, instalado e configurado, em uma infra-estrutura de componentes. Um pacote, em geral, consiste em uma ou mais descrições e um conjunto de arquivos, onde as descrições apresentam as características do pacote com auxílio desses vários arquivos.O componente deve ser empacotado com toda a informação que seu produtor acredite não existir na infra-estrutura. Isso inclui, por exemplo, código-fonte, dados de configuração e outros componentes. O padrão de implantação especifica o formato dos pacotes, assim como o processo de implantação, incluindo uma descrição e o registro do componente. Uma descrição de implantação provê informação sobre o conteúdo do pacote, assim como a informação necessária para o processo de implantação. Essa descrição é analisada pela infra-estrutura alvo e usada para a correta instalação e configuração.

O componente encapsula a informação necessária e os algoritmos para a realização das tarefas. A maneira com que é efetuada a implantação é determinada pelo modelo de empacotamento e envolve três etapas:

- Instalação do componente e preparação para o uso.
- Configuração do componente, dentro da infra-estrutura, e eventualmente do sistema operacional, com o objetivo de disponibilizá-lo.
- Iniciar o componente para uso.

Eventualmente os componentes podem estar disponíveis apenas em forma binária. Esse tipo de empacotamento pode ocorrer para proteger a propriedade intelectual, diminuir custos de implantação e/ou reduzir dependências de contexto. Integração e implantação de componentes devem ser fases independentes do ciclo de desenvolvimento, devendo não existir necessidades de recopilação de aplicação, ou mesmo reinicio. (CRNKOVIC et. al., 2002)

Cada componente Macanudo é empacotado usando-se um arquivo do tipo JAR com a representação interna mostrada na figura 4.4. Na raiz do pacote encontra-se o arquivo

XML descritor do componente. Cada um dos diretórios então subdivide a informação do pacote. Em <*images*> ficam todas as informações visuais que acompanham o componente. Desde ícones para a IDE até material visual de documentação. Em <*source*> localizam-se os códigos-fonte daquele componente. Em <*docs*> encontram-se todos aqueles materiais de suporte para o componente, como manuais e exemplos de utilização. O diretório <*meta-inf*> é usado para controle do pacote jar pela plataforma Java.

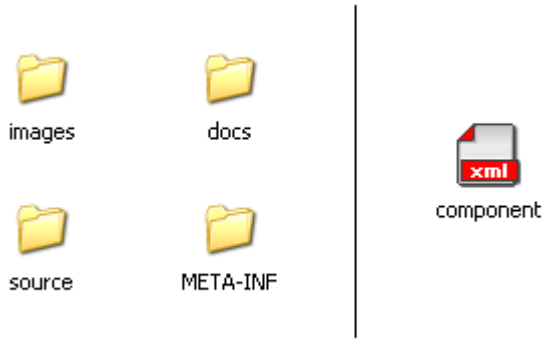


Figura 4.4: Estrutura do Pacote do Componente

4.1.4 Composição

A composição somente é possível se diferentes componentes de variadas fontes possam ser conectados e sejam aptos a trocar informação e compartilhar controle, através de canais de comunicação definidos. Um componente é uma unidade de composição e deve ser especificado de modo a ser possível sua composição com outros componentes, assim como sua integração em sistemas de forma previsível. (CRNKOVIC et. al., 2002) A composição, ou montagem, de componentes é a combinação de dois ou mais componentes de software permitindo o surgimento de um novo comportamento de componente. Um padrão de composição de componentes permite a criação de estruturas maiores a partir da conexão de componentes já existentes e das inserções e substituições de componentes em uma estrutura existente, como pode ser vislumbrado na figura 4.5. Essa estrutura também é chamada de *framework*. A infra-estrutura possibilita não somente o reuso de componentes, mas também de um projeto inteiro.

As duas formas básicas de interações entre componentes são a de cliente/servidor e a de publicação/cadastramento (*publish/subscribe*). Componentes podem agir como clientes, requisitando informação, através de dados ou invocação de métodos, de outros componentes. Um componente pode se registrar junto a um outro componente ou a um serviço dedicado e receber notificações de eventos predefinidos. O modelo de componentes deve definir como produzir interfaces que suportem esse tipo de interação. Metadados das interfaces do componente acabam sendo necessárias para as linguagens e ferramentas de composição.

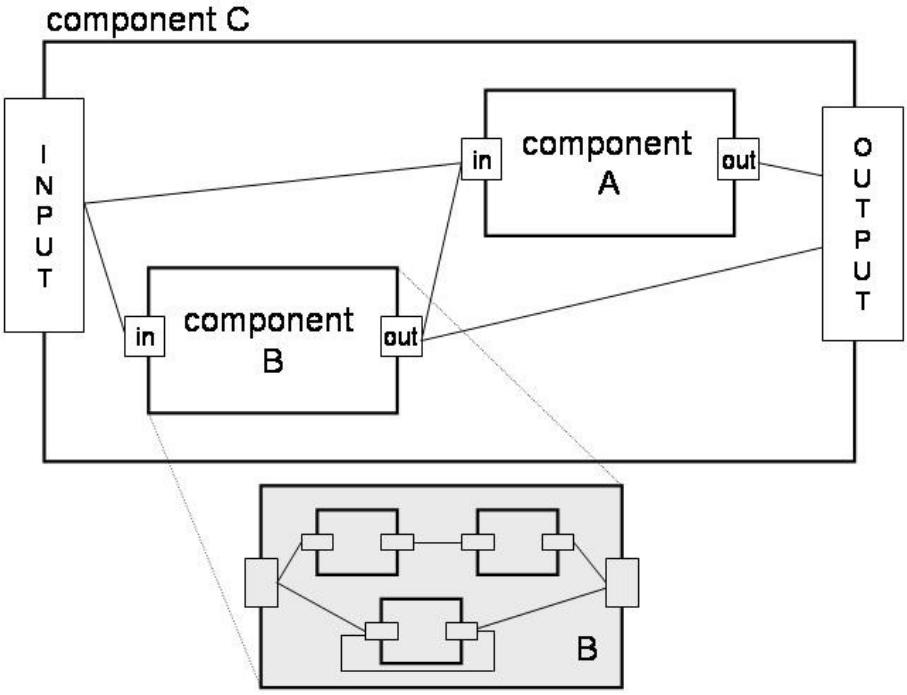


Figura 4.5: Composição de Componentes

Os componentes podem ser conectados usando linguagens de programação, de script, assim como por programação visual ou ferramentas de composição. Composição através de programação visual aumenta o nível de abstração, entretanto necessitam de um esforço extra de projeto para uma representação e edição gráfica eficiente. No modelo Macanudo, optou-se por representar um componente e suas interfaces como a figura 4.6. A interface representada por **`a`** mostra uma disponibilidade, enquanto que a de **`b`** representa uma necessidade daquele componente. Quando as duas se encontram e definem uma ligação, é apresentada pela opção em **`c`**.

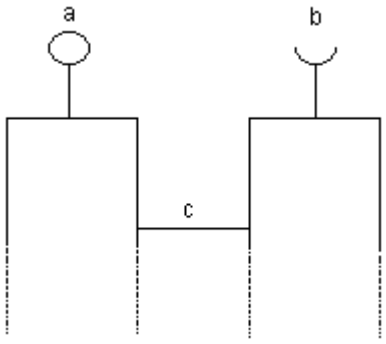


Figura 4.6: Estilos de Interface de Composição

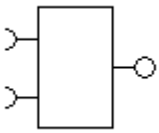
A desvantagem de linguagens e ferramentas de composição é que o código gerado tem de ser escrito ou graficamente especificado na concepção. O máximo do reuso é

alcançado quando a infra-estrutura de componentes é projetada para um domínio específico onde a interação entre as instâncias de componentes é predefinida. A composição acaba se tornando uma tarefa de inserção e substituição, regida pelos padrões de interação e composição definidos no modelo. Adicionalmente, a composição deve permitir a especificação de propriedades particulares que devem ser preenchidas seguindo predefinições de estilo e de ambiente. Por outro lado, Morch (2004) sublinha a importância da fase de integração. Para ele, é uma etapa que aparece como uma poderosa camada intermediária entre a personalização e a programação.

4.1.5 Gramática Visual

Uma linguagem visual é composta por objetos de comunicação, representações textuais e gráficas, que possuem significado em determinados sistemas de interpretação. A definição de formalismos para uma linguagem visual é feita a partir de gramáticas visuais. Uma gramática visual é o conjunto de disposições espaciais de elementos de modo a criar bem-formadas expressões visuais de uma linguagem. Um editor gráfico de propósito geral é construído de maneira a dar bastante liberdade ao usuário. O Macanudo, por outro lado, é bastante restritivo nas formas de comunicação visual. Em sua primeira versão, apenas cinco elementos visuais distintos estão disponíveis para a criação de projetos, como pode ser visto na tabela 4.1.

Tabela 4.1: Elementos Visuais

Elemento Visual	Nome
	"Pino" ou "Porta" do Componente
	Componente
	Interface Requerida
	Interface Disponibilizada
	Conexão Estabelecida

Para futuras implementações, espera-se dar mais liberdade para a representação dos diferentes componentes no projeto. Sabe-se que para isso, entretanto, será necessária a realização de uma definição formal da gramática visual.

4.2 Elaboração do Protótipo

Como todo software com aspirações um pouco mais ambiciosas, o projeto Macanudo foi elaborado seguindo um roteiro de análise, projeto e implementação. O

ambiente é parte integrante de um projeto maior chamado Cave (INDRUSIAK; REIS, 1997).

O Projeto Cave baseia-se no paradigma de integração de ferramentas de apoio ao projeto de circuitos integrados, rodando sobre uma plataforma distribuída com uma interface comum. Sua proposta tem por objetivo a distribuição de recursos de CAD usando o ambiente de rede como infra-estrutura básica, através de um modelo para distribuição de recursos de projeto e ferramentas. Assim como, através da interface ativa do ambiente, chamada de *Tool Laucher*, 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 normalmente sujeitos a erros e caros do ponto de vista computacional.

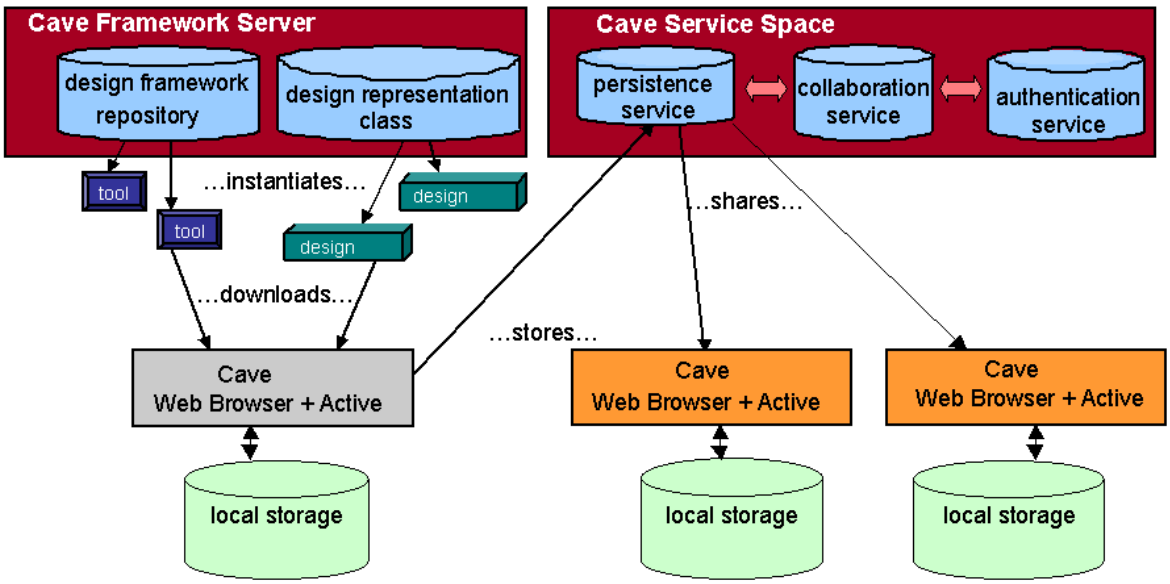


Figura 4.7: Estrutura do Cave

A arquitetura do ambiente Cave, vista na figura 4.7, pode ser analogamente comparada a estrutura do projeto Eclipse (seção 3.3.3.3), onde cada uma das ferramentas introduzidas seria um *plug-in*, assim como serviços de persistência ou de projeto colaborativo (SAWICKI; INDRUSIAK; REIS, 2002). A ferramenta Macanudo, no entanto, pode ser comparada estruturalmente a uma IDE como a vista na seção 3.3.3.2. Dentre todos os elementos que compõe a arquitetura, existem três elementos que devem ser melhor entendidos para uma maior compreensão da interação entre o Macanudo e o CAVE.

O Cave Framework Server é um framework de softwares reutilizáveis que possui um repositório de classes para criação de ferramentas de projeto, e de classes para modelagem das representações dos dados de projeto, que também serão chamadas de primitivas de projeto. O Service Space atualmente disponibiliza serviços de persistência de dados, autenticação, integração com ferramentas externas e colaboração. O Service Space é um dos elementos mais importantes no framework, pois através dele são compartilhados os dados de projeto, permitindo a colaboração entre projetistas O

usuário pode selecionar o servidor ao qual deseja se conectar, entrando com seu username e senha. Após a autenticação do usuário, é apresentada uma janela chamada de Tool Launcher. O Tool Launcher possui uma lista das ferramentas disponíveis no servidor selecionado, podendo esta lista ser personalizada para cada usuário. Ao selecionar uma ferramenta da lista, uma conexão *socket* é aberta e uma instância da ferramenta é criada na máquina cliente, possibilitando o seu uso pelo usuário. Durante a interação do usuário com as ferramentas, os objetos de projeto são instanciados pelo usuário. Ao término de uma sessão de projeto, estes objetos são armazenados em um servidor de persistência, chamado de Cave Service Space.

4.2.1 Planejamento

A elaboração de um programa em nível acadêmico normalmente fica a cargo de uma equipe pequena que se auto-alimenta, sem a busca por informações de usuários-finais distintos dos próprios desenvolvedores. Para tentar modificar um pouco essa situação, esse trabalho se propôs a expandir o universo de requisitos do sistema, ouvindo pessoas de fora do projeto para depois serem construídos diagramas UML para a execução do protótipo.A definição do escopo do projeto é claramente visualizada quando é usado um Digrama de Casos de Uso UML (OMG, 1997) como o da.figura 4.8

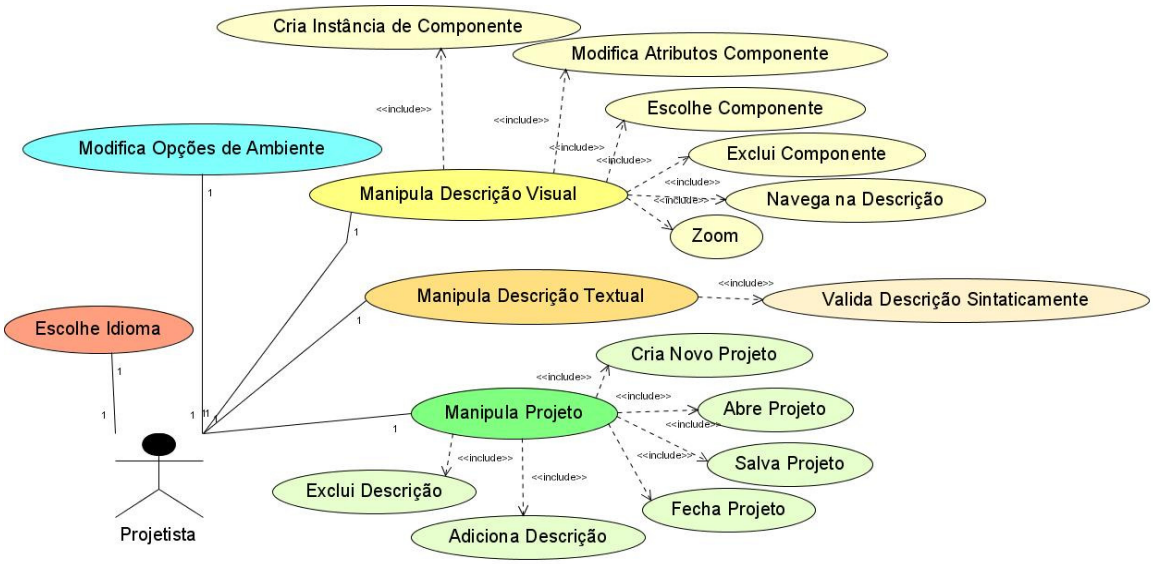


Figura 4.8: Diagrama de Casos de Uso Essenciais

Para um melhor entendimento, anexa-se ao diagrama uma descrição textual e tabular sobre os casos de uso. Escolhe-se para esse trabalho, entretanto, uma visão mais sucinta, mostrando apenas os casos de uso preliminares, apresentadas da tabela 4.2 à tabela 4.6. Cada uma dessas tabelas servirá como introdução a explicação da implementação de cada um destes casos de uso, dando um panorama mais abstrato das funcionalidades projetadas e implementadas para o ambiente.

Tabela 4.2: Caso de Uso Manipula Descrição Visual

Caso de Uso:	Manipula Descrição Visual
Atores:	Projetista
Tipo:	Preliminar
Descrição:	O projetista manipula graficamente o projeto corrente, escolhendo os componentes através de uma barra de ferramentas e instanciando-os na área de trabalho. As características de cada componente podem ser visualizadas e manipuladas através de um inspetor de objetos.

Tabela 4.3: Caso de Uso Manipula Descrição Textual

Caso de Uso:	Manipula Descrição Textual
Atores:	Projetista
Tipo:	Preliminar
Descrição:	O projetista manipula textualmente o projeto corrente.

Tabela 4.4: Caso de Uso Modifica Opções de Ambiente

Caso de Uso:	Modifica Opções de Ambiente
Atores:	Projetista
Tipo:	Preliminar
Descrição:	O projetista modifica opções de acessibilidade do ambiente, como cores e fontes.

Tabela 4.5: Caso de Uso Manipula Informação Armazenada

Caso de Uso:	Manipula Informação Armazenada
Atores:	Projetista
Tipo:	Preliminar
Descrição:	O projetista manipula as informações de projeto, podendo escolher entre as opções de abrir um existente, salvar o corrente ou criar um novo.

Tabela 4.6: Caso de Uso Escolhe Língua

Caso de Uso:	Escolhe Língua
Atores:	Projetista
Tipo:	Preliminar
Descrição:	Um projetista ao iniciar o uso do programa escolhe a língua em que o programa será apresentado.

4.2.2 Execução

A classe principal leva o nome do projeto, e funciona como centralizador, gerenciando a informação entre os objetos que fazem parte do programa. O modelo de domínio da figura 4.9 mostra os principais componentes do ambiente, representados por algumas das classes que o compõem.

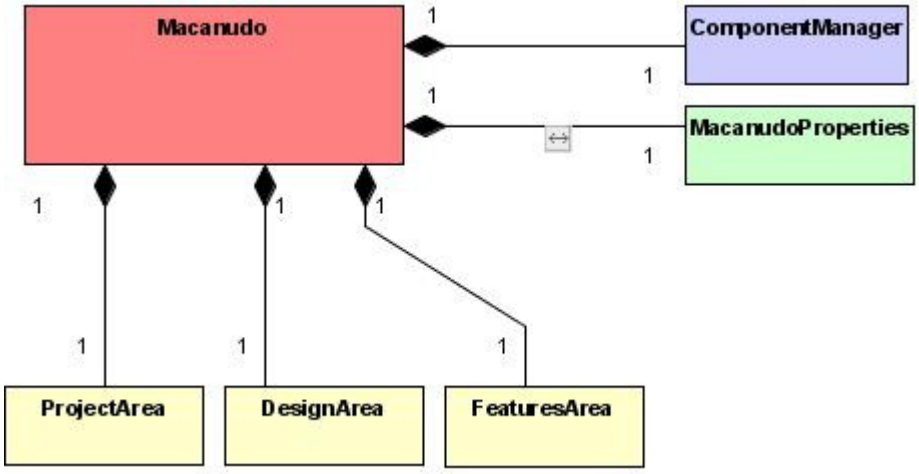


Figura 4.9: Modelo de Domínio do Ambiente Macanudo Desktop

O componente MacanudoProperties é responsável por gerenciar as propriedades do ambiente, o que inclui desde as opções de GUI como cores e fontes, até os dados para gerência de línguas. Essas duas características são melhor explicadas na seção 4.2.2.1.

A gerência dos componentes é feita através do ComponentManager. É ele que controla a disponibilidade dos componentes e como são acessados pelo ambiente. A cada novo projeto gráfico, a linguagem a usar é escolhida, possibilitando assim que o ambiente disponibilize o gerador de código específico, assim como seleciona apenas os componentes que possuem descrições naquela determinada HDL escolhida.

A interface gráfica do ambiente Macanudo é composta por três áreas distintas, como pode ser vista na figura 4.10. Esta separação lógica leva em conta os diferentes universos representados em cada uma delas. Na seção nomeada por **a**, temos a área de funcionalidades do sistema, acessíveis por meio de um menu e barra de ferramentas, chamado FeaturesArea. Em **b**, temos a classe ProjectArea, o espaço destinado às informações sobre projeto e sobre o componente selecionado. Por fim, a maior área da interface gráfica, representada por **c**, é destinada à descrição de hardware, e por isso chamada de DesignArea. Esta área possui uma barra de ferramentas interna, como em outros ambientes integrados do mercado (GENTLEWARE; 2003). Essa separação se deve a um melhor entendimento das funcionalidades disponibilizadas para cada uma das representações possíveis. Como as descrições podem ser manipuladas de maneira gráfica ou textual, foi preferida uma abordagem de representação seccionada para uma maior compreensão pelo usuário-final das opções disponibilizadas para seu uso.

As funcionalidades disponíveis no ambiente estão organizadas pelo menu e pelas barras de ferramentas. Cada uma delas está associada a uma ação, que por sua vez, está associada a uma classe, subclasse de MacanudoAbstractAction. O menu é dividido em

quatro opções: Arquivo, Editar, Componente e Ajuda. Em <Arquivo> estão as funcionalidades de gerenciamento (abrir, salvar, criar novo) de projeto e de descrições (visuais e textuais). Em <Editar> estão as opções de edição das descrições (recortar, copiar, colar) assim como as funcionalidades de visualização dos componentes. Em <Componente> estão agrupadas as operações sobre os componentes, envolvendo adição e gerencia. Em <Ajuda> encontram-se uma breve descrição do ambiente, assim como pequenos exemplos de uso.

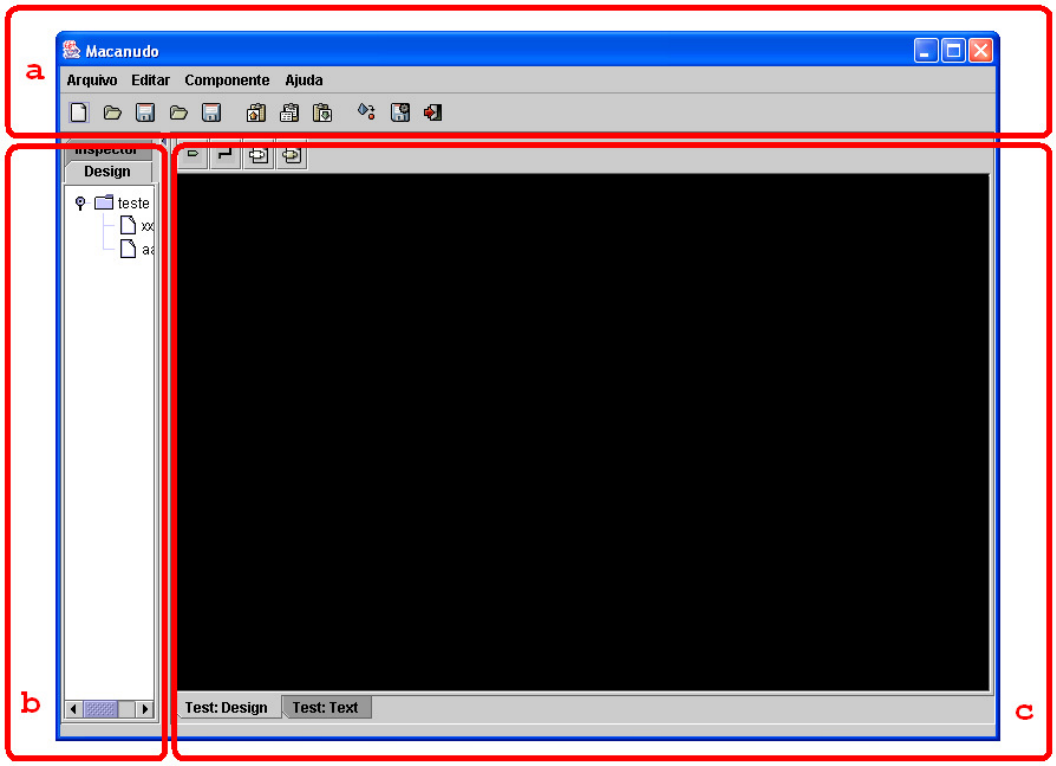


Figura 4.10: GUI Macanudo

Na área de projetos destinada à manipulação gráfica, o desenho dos elementos visuais é feito através de três camadas distintas. Assim, os objetos sofrem uma distinção semântica que os separa em grupos: interfaces externas, componentes e conexões. O objetivo dessa separação é possibilitar a visualização de cada um desses universos de maneira distinta ou de combinações entre eles, com a escolha a cargo do usuário.

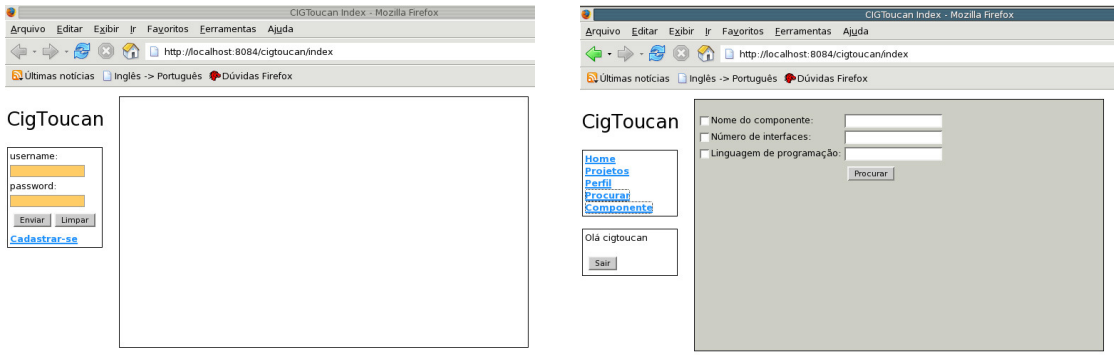
A escolha por uma interface com menor grau de opções simultâneas de escolha de funcionalidades foi feita segundo as noções de Reis e Cartwright (2004), quando falam de IDEs educativas. Para os autores, existem requisitos fundamentais:

- A interface de programação deve ser simples e intuitiva. Uma IDE profissional tende a falhar nesse requisito por ser projetada para prover todas as funcionalidades de auxílio ao desenvolvedor em uma mesma tela.
- Uma IDE pedagógica deve ser leve a ponto de rodar em plataformas menos capazes ou mais antigas. Estudantes em cursos introdutórios normalmente não possuem acesso a computadores com configurações avançadas de hardware e software.

Todo projeto efetuado no ambiente Macanudo é um componente em potencial. O objetivo é sempre mover de níveis mais abstratos para soluções mais específicas, criando um processo de adição de detalhes, em busca de incorporar implementações que solucionem os desafios e problemas propostos. Com isso, algumas separações em nível de implementação devem ser planejadas e executadas para diferenciar os artefatos produzidos. Um projeto é independente de linguagem de programação, pois os componentes que a ele pertencem podem ser fabricados usando diferentes linguagens. Ou seja, mesmo que se possa escolher um determinado HDL para o componente em execução, a noção de projeto continua sendo genérica e trabalha em outro nível. Um arquivo XML guarda a lista de arquivos que compõe aquele componente, assim como gerenciam os *flags* de salvamento. A ele, se usa o termo em inglês *Project*.

Para cada uma das manipulações gráficas de projeto criou-se uma classe específica chamada Design. Ela nada mais é que uma descrição de objetos gráficos, com localização espacial de elementos: portas (interfaces externas), componentes e conexões. Os elementos visuais são subclasses de MacanudoVisualObject, e possuem referência ao nome de um componente real. Essa referência através de uma String cria um compromisso para o gerente de componentes (ComponentManager).

O ambiente de desenvolvimento integrado apresentado nesse trabalho está focado na plataforma *desktop*. Uma versão preliminar para o universo *web* foi elaborada como trabalho de conclusão de curso. (KOPLIN; SERENI; BOLZAN, 2005). O protótipo criado tem a finalidade de facilitar a inserção, atualização, remoção e busca de componentes de hardware. Isto é alcançado através da criação de uma biblioteca de componentes, onde o usuário pode fazer diferentes tipos de busca. A arquitetura do sistema foi feita com base a suportar as idéias de times de projeto, ou seja, existem todas as funcionalidades de documentação e disponibilização de componentes e projetos, assim como administração dos participantes da equipe de desenvolvimento. As telas de início do sistema e de busca por componente podem ser vistas, respectivamente, pela figura 4.11.



(a) (b)

Figura 4.11 Telas Projeto CigToucan

4.2.2.1 Personalização do Ambiente e Internacionalização

Todos os programas estão imersos em ambientes que possuem atributos de sistema, como nome ou endereço da máquina, sistema operacional e usuário. A aplicação pode,

adicionalmente, possuir suas próprias configurações que serão carregadas ao inicializar a aplicação, e deverão ser persistidas quando acabar sua execução. Essas propriedades, muitas vezes chamadas de preferências, podem estar relacionadas com a interface gráfica (tamanho da janela, conjunto de cores) ou mesmo com endereços para aquisição de recursos necessários para o funcionamento do sistema (lista de componentes, arquivos de línguas).

```
    Title = Macanudo
    GenerateVHDLCodeCommand = Gera Código VHDL
    GenerateVHDLCodeDescription = Gera Código VHDL da Descrição
Visual
    GenerateVHDLCodeMnemonic = G
    SaveCommand = Salvar
    SaveDescription = Salva a \u00C1rea de Trabalho em Arquivo
    SaveMnemonic = S
    FileException = Não existe arquivo com esse nome.
    ComponentException = Não existe componente com esse nome.
    Creation = Criação
    Edition = Edição
```

Figura 4.12: Extrato do Arquivo de Internacionalização

A plataforma Java disponibiliza uma classe chama Properties que gerencia os atributos em conjuntos de pares no estilo “chave-valor”. Essa estrutura está ligada ao fato da classe implementar o conceito de *hashtable*. A chave é usada para a identificação do valor que se deseja recuperar e, portanto, deve ser única no documento. Essa estrutura é a mesma usada nos arquivos de internacionalização, como pode ser visto no trecho apresentado na figura 4.12.

O ambiente geográfico, juntamente com a linguagem, são fatores que influenciam a percepção e a interação das pessoas e dos eventos que nos envolvem. A comunicação efetiva entre diferentes pessoas passa pela consideração que possuem sobre a cultura, a linguagem e o ambiente de cada uma delas. Analogamente, um sistema de software deve ser capaz, através de sua GUI, de se comunicar efetivamente com seus usuários. Para isso, deve ser capaz de regionalizar informações de ambiente e linguagem. Uma das maneiras mais eficientes para isso é através de uma metodologia chamada Internacionalização.

Internacionalização é o processo de projetar *software* para ser adaptado a várias linguagens e regiões de maneira fácil, pouco custosa e sem mudanças arquiteturais no sistema. Algumas vezes o termo é abreviado para i18n, da palavra inglesa *internationalization*, que possui 18 caracteres entre o i inicial e o n final. Localização (*Localization*) é a prática de adicionar componentes específicos a determinadas regiões, como texto traduzido e comportamentos de formatação de dados. A plataforma Java já possui muitas das funcionalidades de internacionalização integradas aos pacotes disponibilizados pela Sun, facilitando esse tipo de desenvolvimento. Um programa internacionalizado possui as seguintes características:

- O mesmo código executável funciona em todas as regiões do mundo com a adição de dados de local;
- Elementos textuais como mensagens de status e elementos de GUI não são armazenados no programa, mas em arquivos separados que podem ser consultados dinamicamente;
- A adição de novas linguagens não necessita de nova compilação;
- Dados como datas e moedas, que são dependentes de cultura, aparecem em formatos conhecidos e usuais para os usuários-finais.

Internacionalizar um *software* não é uma tarefa fácil para desenvolvedores que possuem uma arquitetura que não tenha sido planejada para isso desde o seu início. Para auxiliar esse trabalho de manutenção, já existem programas (LINGOPORT, 2005) e metodologias (GREEN, 2000). É importante que o papel de internacionalização dentro de uma equipe de desenvolvimento seja dado a uma pessoa com conhecimento da cultura e da língua em questão, além do conhecimento técnico.

A implementação da internacionalização usa o conceito de local (*locale).* Um identificador de local é um conjunto de parâmetros que define as variáveis da região. Nessa implementação, consiste no par de identificadores de língua e região, seguindo a codificação ISO 639 e 3166, respectivamente. Atualmente as versões que acompanham o ambiente são em português do Brasil e inglês norte-americano.

4.2.2.2 Leitura dos Arquivos XML

O projeto Commons Digester da Apache (APACHE, 2002) foi criado pela necessidade que outro de seus projetos, o Jakarta (APACHE, 1999), tinha de inicializar objetos Java através de arquivos de configuração escritos em XML. Muitas abordagens para leitura de XML existem, sendo baseadas em SAX ou DOM, com grandes códigos sendo gerados para a correta manipulação dessa informação. Digester é basicamente um mapeamento de XML para Java, que permite a criação de objetos a partir do reconhecimento de padrões existentes dentro do arquivo, definidos pelo programador.

Cada arquivo XML que compõe o ambiente Macanudo (componentes e projetos, por exemplo) é lido através de uma classe específica que implementa a abordagem Digester. Essa separação é de fundamental importância do ponto de vista de manutenção do ambiente, pois o uso de XML tende a ser dinâmico, com grandes possibilidades de extensão e adaptação. A criação de classes distintas, para tratar esse processamento, possibilita uma independência do sistema para a construção dos objetos persistidos.

```
<structuralproperty>
   <programmingLanguage>VHDL</programmingLanguage>
   <resourceFile type = "source code">test_architecture.vhd</resourceFile>
   <resourceFile type = "source code">test_configuration.vhd</resourceFile>
   <resourceFile type = "online reference">test_help.html</resourceFile>
   <resourceFile type = "manual">test_manual.pdf</resourceFile>
</structuralproperty>
```

Figura 4.13: Extrato Exemplo do XML de um Componente

Cada entrada do arquivo pode gerar um objeto Java ou um atributo de uma determinada classe. A escolha é feita através da informação armazenada. No caso do exemplo da figura 4.13, temos dois tipos de informação: *resourceFile* e *programmingLanguage*. Enquanto que a primeira cria um objeto do tipo File, tendo como atributos o nome de arquivo o *string* da *tag* e a propriedade de tipo (*type*) encontrada também dentro da *tag*, a segunda apenas identifica a linguagem fonte, sendo atributo do componente que está sendo lido.

A população do ambiente pelos componentes que compõe o repositório particular do projetista é feita, igualmente, através da leitura do XML de cada componente. Um objeto ComponentManager é responsável pela gerencia desde a leitura de cada um dos componentes até a utilização dos mesmos em diferentes projetos. Cada componente está localmente acessível para o projetista e sua leitura é efetuada no momento da inicialização do ambiente. O ComponentManager, através de um ComponentDigester lê as informações contidas no descritor XML do pacote JAR e cria os objetos em memória. Assim, a cada nova interação do usuário, o gerente de objetos é capaz de disponibilizar ou não um determinado componente, seguindo as restrições delimitadas pelo projetista, assim como os eventuais múltiplos comportamentos existentes.

4.2.2.3 Manipulação de Descrição Visual

A informação gráfica manipulada no ambiente é feita através de uma área projetada especialmente para isso, que diferencia as funcionalidades de inserção e edição através de diferentes objetos *listeners*. O processo de inserção disponibiliza ao usuário a escolha do tipo de informação a adicionar: conexão, interface externa ou componente. No caso dessa última escolha, o processo é composto por duas etapas adicionais. Após escolher a opção de inserção de componente, uma janela é mostrada ao projetista. Nela se encontram todos os componentes que o gerente de componentes disponibiliza para aquela determinada descrição visual, agrupada através de uma hierarquia de seções que tem como objetivo facilitar a visualização e identificação dos componentes. Depois do componente escolhido, se ele possuir mais de uma implementação possível, é informado ao usuário a opção de escolha entre as existentes.

Esse conceito de separar o projeto computacional do composicional é baseado na idéia apresentada no trabalho de Rowson e Sangiovanni-Vincentelli (1997). Essa abordagem recebeu o nome de Projeto Baseado em Interface (*Interface-Based Design*) e pode ser visualizado através de esquema na figura 4.14. Os aspectos composicionais ficam a cargo da interface, enquanto que os computacionais são os comportamentos possíveis. Esse paradigma baseia-se em três elementos: formalização, abstração e decomposição.

- Formalização: consiste na captura do projeto e sua especificação de forma a não deixar ambigüidades.
- Abstração: elimina detalhes que não possuem importância quando se lida com níveis mais altos de abstração ou quando é avaliado se um projeto satisfaz uma propriedade em particular.
- Decomposição: consiste em quebrar o projeto de um determinado nível de abstração em componentes capazes de serem projetados e verificados independentemente.

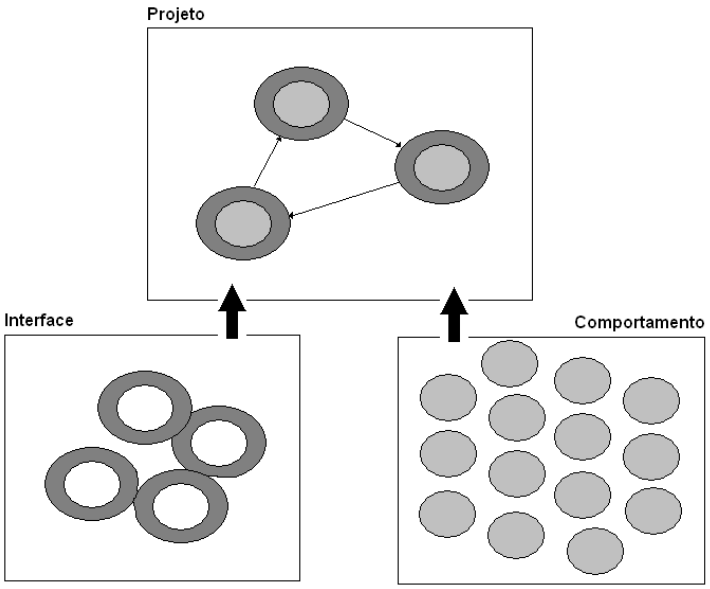


Figura 4.14: Separação entre Comportamento e Interface

Esta abordagem se enquadra bem ao nosso processo e podemos visualizá-la com o auxílio da figura 4.15 (LEE, 2000). Se considerarmos um componente como uma "*entity*", teremos a "*architecture*" como sua interface e a "*configuration*" como seu comportamento. Com isso, surge no projeto a possibilidade de dividir uma arquitetura em diferentes configurações, fazendo com que um mesmo componente possa ser construído com uma mesma interface para diferentes aspectos comportamentais. O ambiente, então, reflete essa possibilidade na hora de criação do componente, onde sua montagem ocorre em momentos distintos, manipulando-se tanto as funcionalidades composicionais como seus diferentes comportamentos.

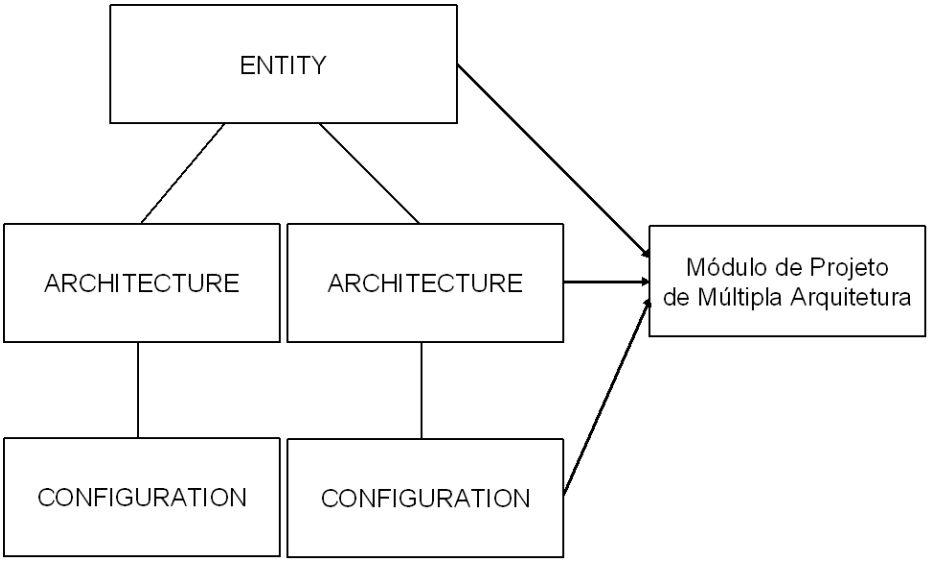


Figura 4.15: Estrutura de um Projeto VHDL

Eventualmente um componente necessário para uma determinada descrição visual pode não existir na biblioteca do projetista, sendo necessária sua fabricação. Para isso, existe a opção de adição de novo componente. Esta opção permite ao usuário a criação automática de uma nova descrição com interface definida na sua inicialização, ou seja, uma nova descrição com quantidade de portas pré-definida.

A montagem do pacote é feita através de uma GUI do tipo *wizard,* onde cada uma das etapas é apresentada em abas distintas de uma mesma janela, para guiar o projetista através das possibilidades de informações a agregar no componente. O produto final gerado é o arquivo jar com o xml correspondente, podendo então ser atualizado tanto na biblioteca local quanto em um ambiente distribuído.

4.2.2.4 Integração Cave

A informação gráfica manipulada no programa é feita através de uma área projetada baseada no modelo proposto, assim como também para ser integrada também no ambiente Cave. Pensando nisso, cada descrição além de possuir os dados do componente no universo da área de projeto (coordenadas espaciais e tamanho, por exemplo) também agrega a informação de autoria. Ou seja, cada componente inserido no universo é identificado como sendo de "autoria" do usuário que o instanciou no projeto.

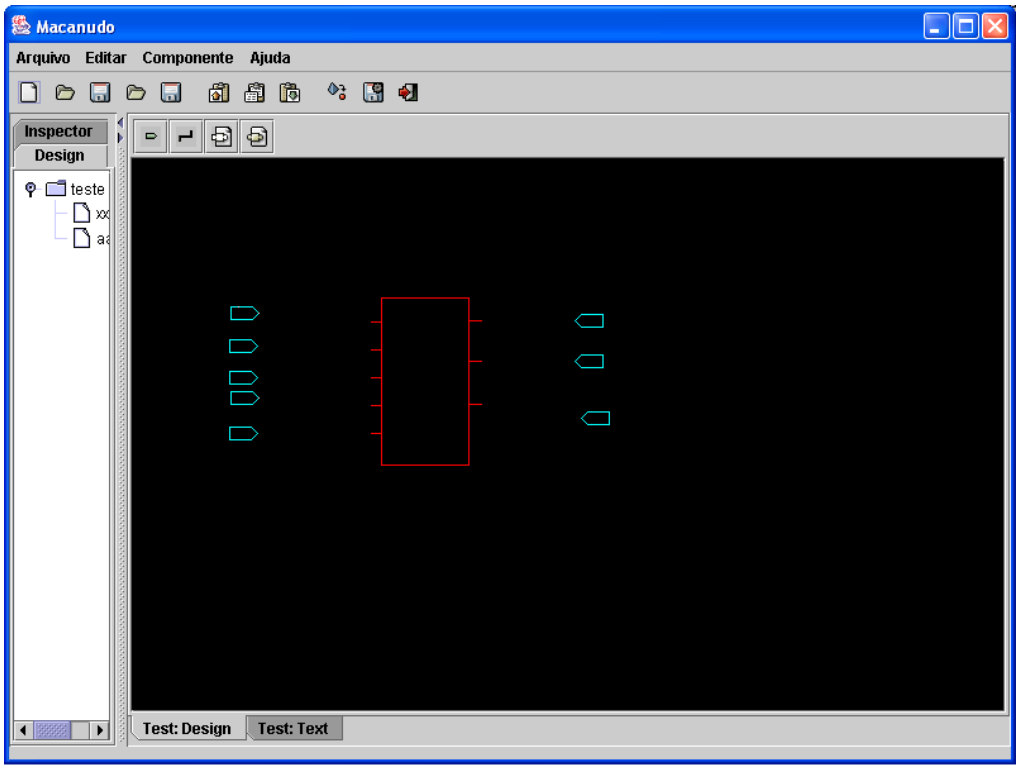


Figura 4.16: Visualização de Elementos por Autoria

Essa idéia vem da infra-estrutura disponibilizada pelo sistema de colaboração do Cave (SAWICKI; INDRUSIAK; REIS, 2002). O ambiente então, é capaz de diferenciar a autoria visualmente, possibilitando aos projetistas uma visão sobre a autoria da colaboração de determinados projetos. A figura 4.16 mostra o exemplo de dois usuários que compartilharam o mesmo projeto. O primeiro inseriu os pinos da interface, enquanto que o segundo definiu o componente que finaliza o projeto.

4.2.3 Exemplo de Construção Passo a Passo de um Componente Através de Componentes Existentes

A construção de um novo componente pode ser acompanhada nessa seção através de uma seqüência de passos que pretende demonstrar o ciclo de processo apresentado pela figura 4.1. Com o programa aberto (figura 4.10), escolhe-se a opção inserir componente existente, quarto elemento da barra de ferramentas da figura 4.17. Essa GUI é composta ainda pelas opções de inserção de novo pino, criação de conexão e adição de novo componente, respectivamente.

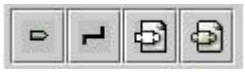


Figura 4.17: Barra de Ferramentas Interna

Esta opção disponibiliza para o projetista, a escolha de um componente para inserção no projeto e é apresentada de maneira organizada como pode ser vista na figura 4.18. Cada botão da barra representa um componente distinto disponível. As barras são agrupadas em seções dentro da biblioteca. Cada seção é representada por uma das abas da interface. Cada componente selecionado apresenta alguns dos metadados lidos pelo ambiente a partir do pacote, auxiliando ao projetista uma melhor escolha. Quando o projetista tem sua decisão tomada, seleciona a opção de OK da GUI e então seu componente é inserido em sua área de trabalho corrente, como visto na figura 4.19.

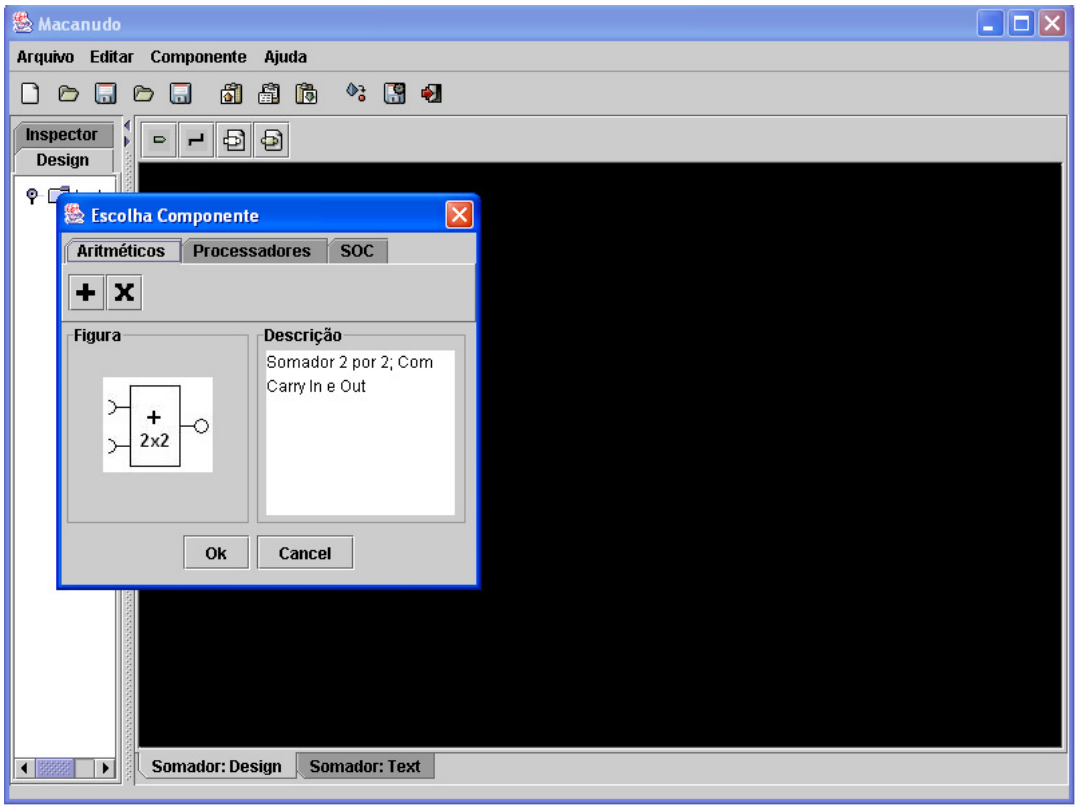


Figura 4.18: Escolha de Componente

Um componente é definido pelo seu comportamento, assim como pela sua interface. Para isso, são adicionados pinos ao projeto, através do botão mais a esquerda da figura 4.17. Com pinos e/ou componentes inseridos, já é possível que sejam criadas conexões entre os elementos, como em figura 4.20.

Depois que todos os pinos e componentes estiverem devidamente conectados, como na figura 4.21, o projeto pode gerar um código VHDL, assim como também está pronto para ser empacotado e virar um novo componente. A geração de VHDL é feita através de uma ação específica, representada pela classe GenerateVHDLCodeAction. Esta forma de projetar o ambiente deixa margem a desenvolvimento do programa, em adições de novas linguagens ou mesmo melhora na geração de código, por usar a arquitetura de ações do Java, que diminui enormemente o acoplamento entre a GUI e a geração de código-fonte HDL. O código gerado pela ação é mostrado através de uma nova aba na área de trabalho, como pode ser vista na figura 4.22. As classes Java da área textual são reusadas do projeto jEdit (JEDIT, 1999) e do próprio CAVE.

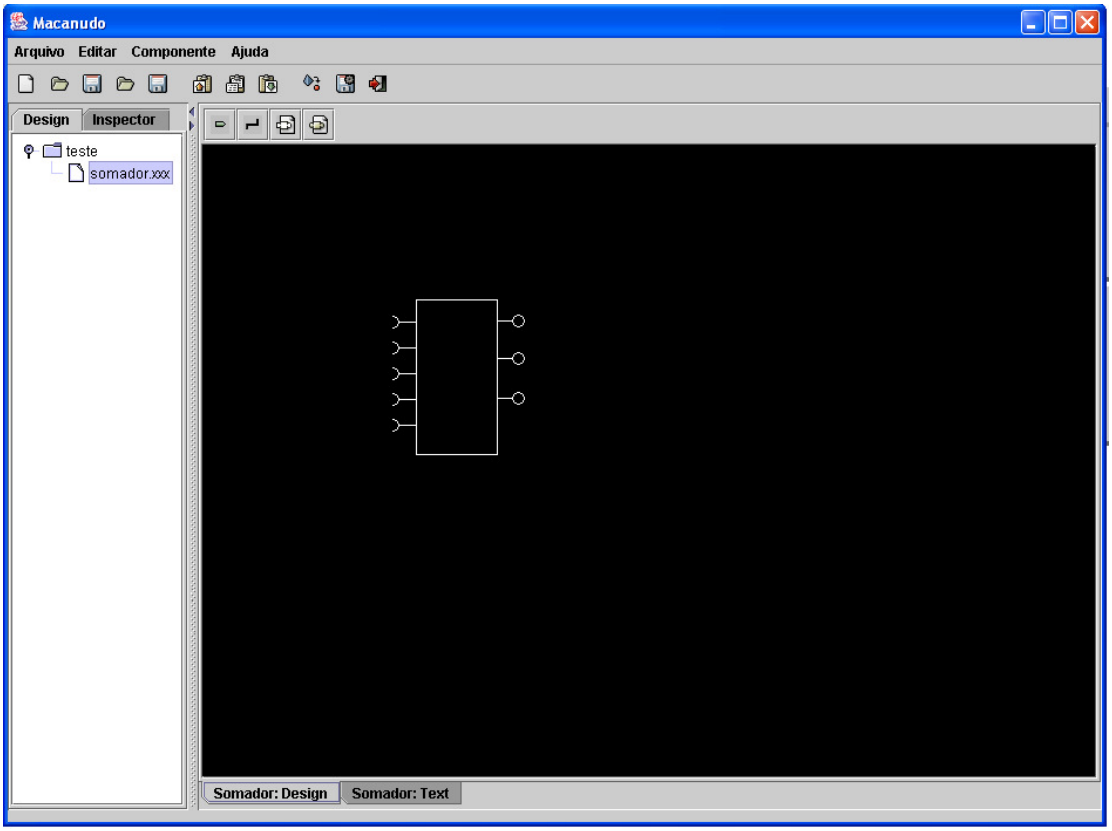


Figura 4.19: Área de Trabalho com Componente Existente

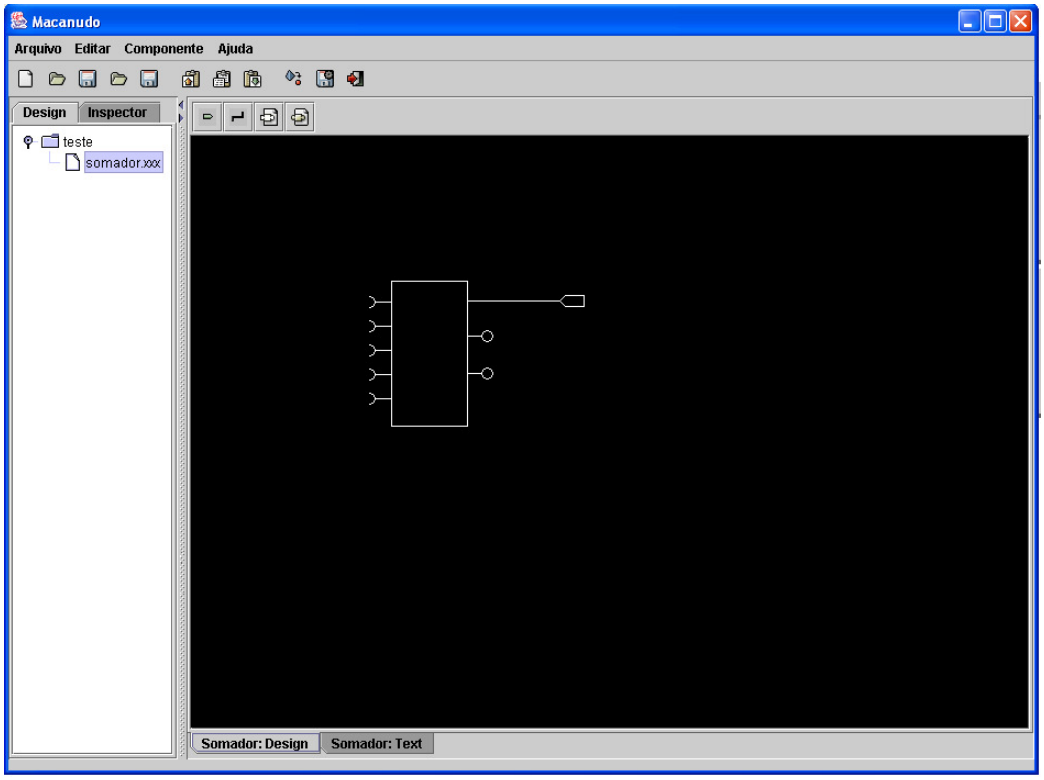


Figura 4.20: Inserção de Pino e Conexão

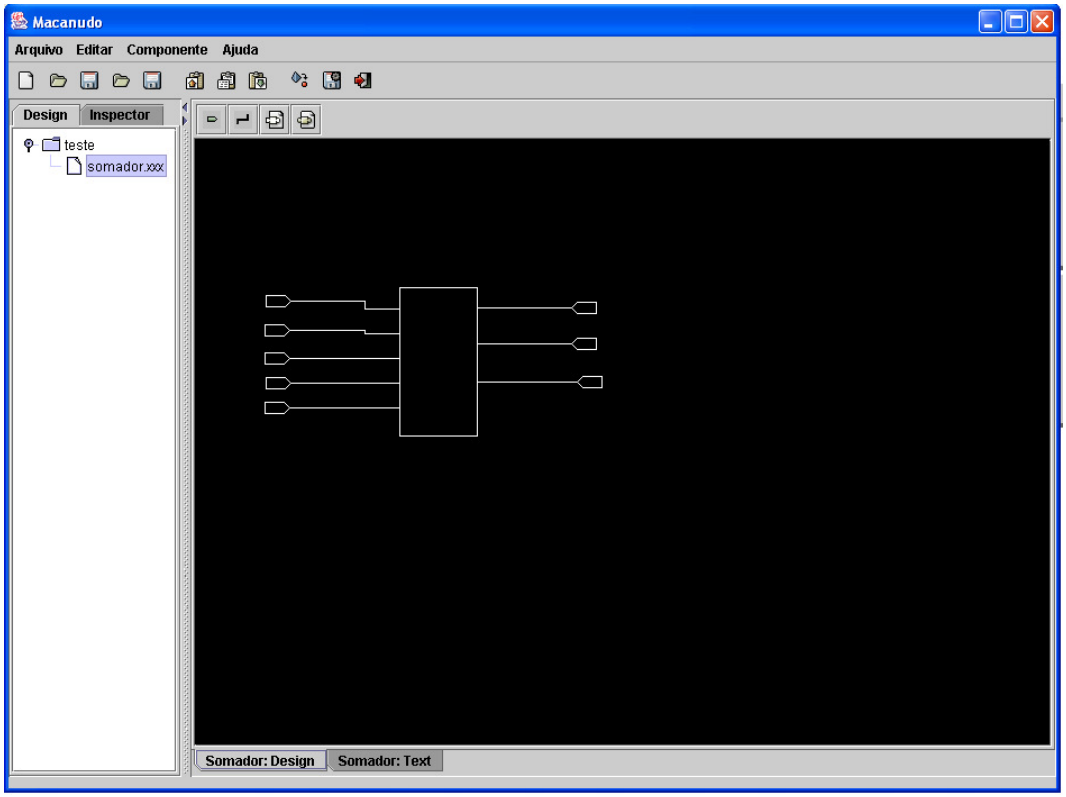


Figura 4.21: Projeto Pronto para Ser Empacotado

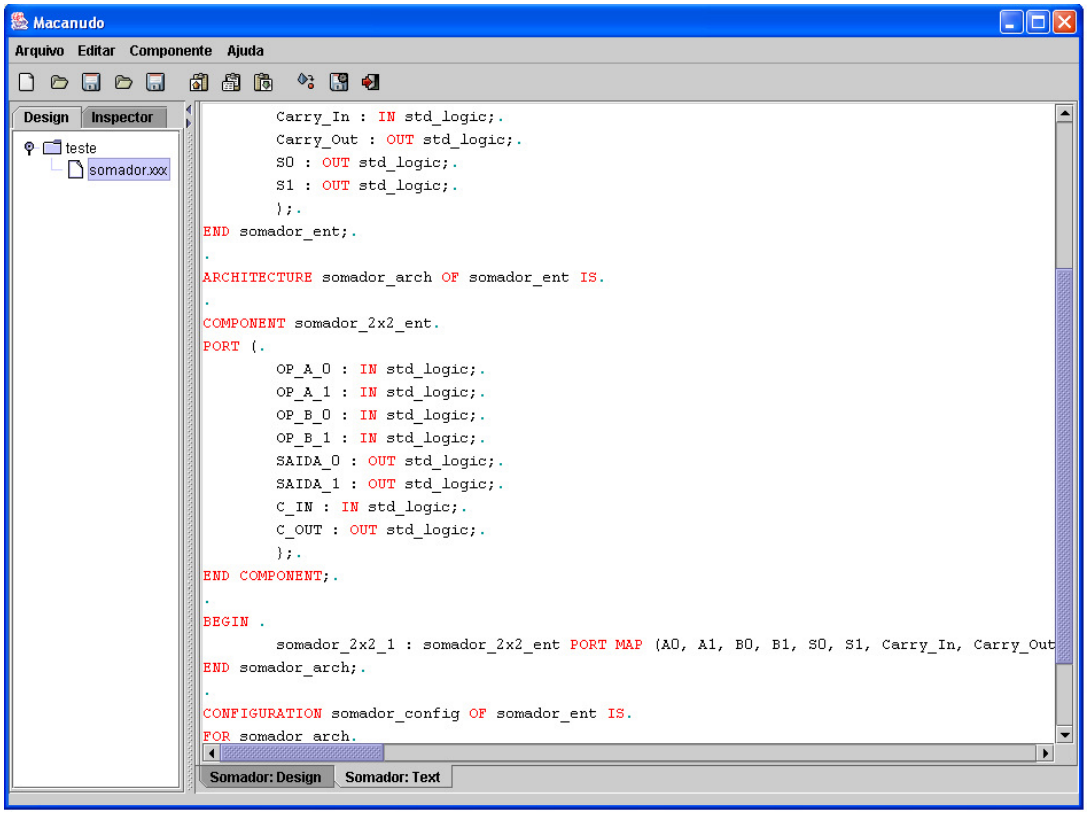


Figura 4.22: Geração VHDL

O empacotamento do projeto para a construção do componente é dado através do ComponentPackager, uma ferramenta que pode ser acessada tanto externamente quanto internamente. Essa opção foi realizada por haver dois tipos de usuários que queiram usá-la: aqueles que já possuem um código VHDL e querem compor um componente, e por aqueles que estão usando o ambiente Macanudo e querem criar seu pacote. Para esse segundo grupo, o ComponentPackager traz a facilidade de se comunicar com o ambiente e popular automaticamente alguns dos campos de sua GUI, como as interfaces. Esses dados são recolhidos diretamente da descrição do projeto que está sendo construído. A interface do ComponentPackager foi organizada de modo a auxiliar o projetista a preencher corretamente todos os dados para a construção do XML de cada componente, como pode ser visto na figura 4.23 e na figura 4.24.Com o pacote gerado, fica a cargo do projetista popular seu repositório pessoal ou disponibilizá-lo para outros usuários.

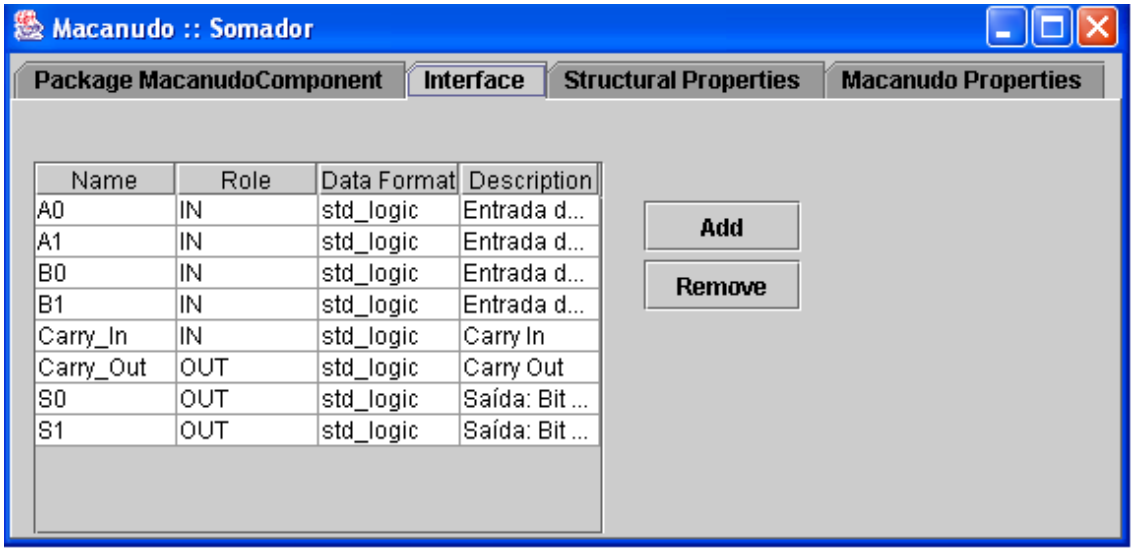


Figura 4.23: Empacotador

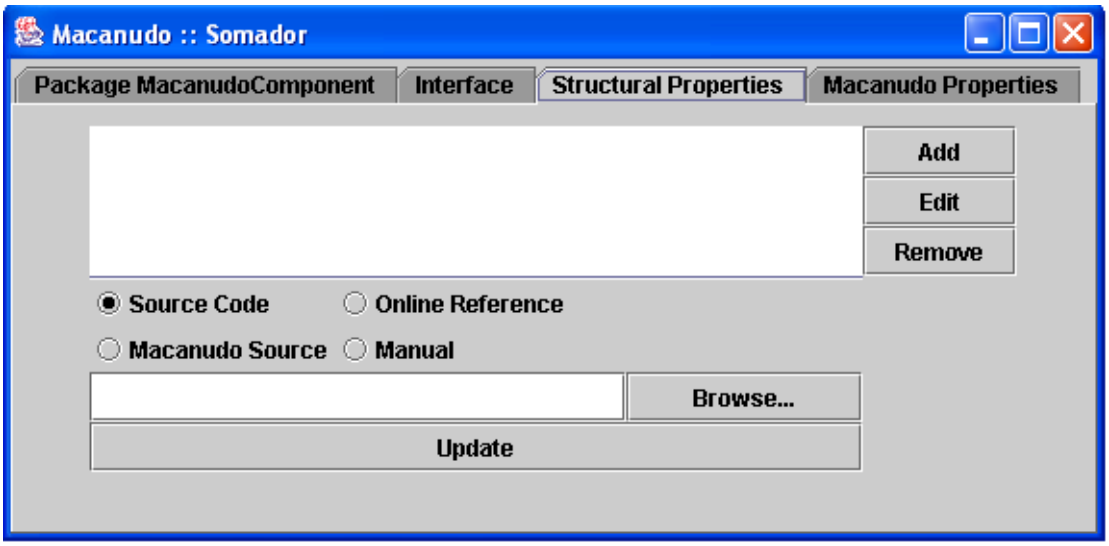


Figura 4.24: Empacotador (Propriedades Estruturais)

.

5 CONCLUSÃO

O reuso de projetos é ponto-chave para trazer os esforços de lidar com a complexidade de projetos a níveis gerenciáveis. Para isso, políticas orientadas a reuso devem estar presentes em todas as etapas do processo. Isso é política fundamental, como já dizia Zubeck (1997) ao mostrar que desenvolvedores de software que usam RAD não são acostumados a criar seus códigos-fonte para reuso. Ao afirmar que várias das técnicas de reuso são aprendidas pelos projetistas no meio acadêmico, mas não são utilizadas no mercado, Keating e Bricaud (2002) potencializam o problema apresentado por McFarland (1993) quando este afirma que os engenheiros não se consideram programadores. Para eles, é fundamental que além de uma nova ferramenta, venha atrelada a ela uma metodologia que dê o embasamento necessário para aumentar o desempenho nos processos de projeto, verificação e depuração ao reduzir o custo das iterações de codificação e teste, concentrando esforços na fase de desenvolvimento e daí então na fase de integração.

Reusar componentes complexos de software e hardware já utilizados e testados em projetos anteriores pode reduzir drasticamente o tempo de projeto. (WAGNER et. al., 2004) No entanto, é importante enfatizar-se o fato de ferramentas de projeto não serem suficientes. Junto delas devem existir novas metodologias que lidem com os problemas que os ambientes de desenvolvimento não são capazes de lidar. (SANGIOVANNI-VINCENTELLI, MCGEER, SALDANHA, 1996) Uma das maiores contribuições do Macanudo esta na habilidade de escrita de código reusável através de um processo documentado e de ferramentas que dão suporte a essa abordagem. O desenvolvimento de código reutilizável através de projetos e componentes acelera o processo diminuindo custos de escrita e manutenção de projetos.

Outro fator diferenciado no ambiente é usar o conceito de projetos ao invés de ir diretamente ao código-fonte, textual ou gráfico. Com isso, funcionalidades de criação e gerência de projetos acabam disponibilizando uma melhor navegação, organização e edição em formatos mais acessíveis aos desejos e necessidades dos projetistas. Esse tipo de separação gera flexibilidades individuais, que através de organizações faz com que haja um maior controle sobre projetos, desde sua forma de desenvolvimento a sua forma de compartilhamento entre diferentes setores ou até mesmo com outras empresas.

Quanto maior o projeto, maiores e mais detalhadas acabam sendo as revisões. Além disso, mais componentes acabam participando dos projetos. Dependendo das condições do projeto, seus membros podem estar a uma mesa ou a um continente de distância. Documentar o componente com auxílio de meta-dados usando uma linguagem

descritiva como XML é uma vantagem tanto para compartilhamento como reuso. Por essa razão, esse foi o formato definido na proposta desse modelo. O reuso de projetos é uma realidade em todos os níveis de projeto de CI. Hierarquizar a composição através de novos projetos e componentes é aumentar a compreensão do projeto pela utilização das possibilidades de visualização de diagramas e navegação dentro das hierarquias. A isso adicionamos o que disse Jung et. al. (2000) quando falam que a composição ainda é vista como um mecanismo mais flexível do que herança, porque pode ser mudada dinamicamente.

Sempre foi preocupação desse trabalho que o produto gerado fosse parte integrante de um fluxo de desenvolvimento automatizado, como o Fucas. Por isso, o programa é um protótipo operacional cuja saída é uma descrição de componentes em RTL, contendo os detalhes necessários para a seqüência automática de síntese. Ainda, a construção das classes do ambiente foi feita para ser integrada ao projeto CAVE. Todas elas tendo sido construídas seguindo algumas diretivas quanto à necessidade (no caso de classes que implementam QuadTree) e à documentação. Nessa última, pode-se afirmar que houve um ganho significativo da documentação gerada automaticamente através da ferramenta Javadoc, em comparação àquela existente antes do Macanudo. Com isso, aumenta-se a vida útil das classes elaboradas neste trabalho. A versão atual conta com a implementação na ordem de grandeza de centenas de classes, distribuídas em diferentes pacotes.

As ferramentas mostradas na seção 3.3.3 (página 51) trouxeram uma amostra do universo que serviu como inspiração e agora é utilizada como comparação. Quando pensamos na disponibilização de componentes, temos como exemplo o OpenCores. Ele funciona como um repositório *online*, onde projetistas de todo mundo podem acessar e baixar os projetos e a documentação disponibilizados. Entretanto, nada mais é do que um *front-end* para um sistema de controle de versões. O modelo Macanudo melhora essa visão ao construir pacotes para distribuição, organizando melhor a implantação do código gerado, tanto no próprio ambiente quanto em outros projetos. Testes preliminares de integração foram feitos com o auxílio da ferramenta CigToucan, onde foi constatado uma maior agilidade na navegação entre os componentes. O OpenAccess centraliza a informação em uma mesma base de dados, entretanto essa informação é populada por diferentes ferramentas em diferentes níveis de um fluxo de projeto. No Macanudo, a informação sobre componente é específica ao nível visual que o modelo se propõe e ao HDL gerado por ele.

A análise das GUI entre o Macanudo e as ferramentas da Mentor e da Summit demonstram a preocupação que esse trabalho teve na construção de uma ferramenta que pudesse ser usada primeiramente em meio acadêmico. Um dos objetivos principais era fazer com que usuários novatos pudessem usá-la sem o temor de não saber qual das funcionalidades usar dentro das inúmeras a seu alcance. Essa abordagem fica clara ao colocarmos lado-a-lado as diferentes ferramentas. Mesmo que o Macanudo possua menos opções disponíveis, o leiaute foi projetado de modo a aumentar a velocidade do usuário em encontrar a opção desejada. Essa filosofia será mantida para os trabalhos futuros onde novas ferramentas, como simulação e trabalho colaborativo, serão integradas ao ambiente.

A geração de descrições de projeto diferentes de componentes auxilia, principalmente, para projetos envolvendo diferentes projetistas ou até mesmo equipes. Com isso, a modularização e divisão de tarefas e esforços fica facilitada pela opção

entre essas duas possibilidades, sem a necessidade de uma ferramenta externa, como é o caso da solução da Summit.

5.1 Trabalhos Futuros

O ambiente Macanudo tem várias expansões a serem feitas. Algumas delas são bastante importantes para facilitar a seqüência de desenvolvimento de IC. Para isso, está na agenda a integração do Macanudo em um fluxo de desenvolvimento existente, assim como a ferramentas de depuração e compilação para o código gerado.

Dando continuidade a exploração de projetos de hardware, os próximos trabalhos devem abordar outras abordagens em HDL que não foram implementas por decisão de escopo deste trabalho, como ampliar as linguagens e até mesmo integrar com outros modelos de componentes similares, como o proposto por Beck et. al (2003). Ainda dentro dos projetos surgidos dentro da mesma instituição, existe a possibilidade de estudar a adequação das classes Java para serem utilizadas por abordagens gráficas baseadas em UML. (BRISOLARA et. al., 2005)

Ferramentas que agilizem o processo de captura de informação também são esperadas para futuras versões. Estudo preliminar, para aquisição de informação através de máquinas de estados, foi começado, mas não teve uma continuação adequada. Ainda é interessante uma ferramenta de geração de novos componentes através do próprio uso de HDL. Por exemplo, é possível gerar um multiplicador de 16 bits através de uma enumeração em VHDL.

REFERÊNCIAS

ALLEN, P. CBD Survey: The State of the Practice. **The Cutter Edge**, 2002.

ALLEN, P.; FROST, S. Planning Team Roles for CBD. In: HEINEMAN, G. COUNCILL, W. (Ed.). **Component-Based Software Engineering - Putting the Pieces Together.** Addison-Wesley, 2001. p.113-129

APACHE SOFTWARE FOUNDATION. **Commons Digester.** 2002. Disponível em: <http://jakarta.apache.org/commons/digester/>. Acesso em: nov. 2005.

APACHE SOFTWARE FOUNDATION. **The Jakarta Project.** 1999. Disponível em: <http://jakarta.apache.org/>. Acesso em: nov. 2005.

ASHENDEN, P. J.; WILSEY, P. A.; MARTIN, D. E. SUAVE: Extending VHDL to Improve Data Modeling Support. **IEEE Design & Test**, Los Alamitos, v.15, n.2, p.34-44. 1998.

BARNA, C.; ROSENSTIEL, W. Object-oriented reuse methodology for VHDL. In DESIGN, AUTOMATION AND TEST IN EUROPE CONFERENCE, DATE, 1999. **Proceedings...** New York: ACM Press, 1999.

BECK, A.C.S.; MATTOS, J.C.B.; WAGNER, F.R.; CARRO, L. CACO-PS: a general purpose cycle-accurate configurable power simulator. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, 16., 2003, São Paulo. **Proceedings...** Los Alamitos: IEEE Computer Society, 2003. p. 349 - 354

BECK, K.; FOWLER, M. **Planning Extreme Programming**. [S.l.]: Addison-Wesley, 2000

BELLOWS, P.; HUTCHINGS, B. JHDL - An HDL for Reconfigurable Systems. In: IEEE SYMPOSIUM ON FPGAS FOR CUSTOM COMPUTING MACHINES (FCCM). **Proceedings...** 1998. IEEE Computer Society, Washington, Estados Unidos.

BENNATAM, E. M. **Software Project Management:** A Practitioner's Approach**.** McGraw-Hill, 1992.

BENZAKKI, J.; DJAFRI, B. Object oriented extensions to VHDL, the LaMI proposal. In: IFIP TC10 WG10.5 INTERNATIONAL CONFERENCE ON HARDWARE DESCRIPTION LANGUAGES AND THEIR APPLICATIONS : SPECIFICATION,

MODELLING, VERIFICATION AND SYNTHESIS OF MICROELECTRONIC SYSTEMS. **Proceedings...** Toledo, Espanha. 1997. p. 334-347.

BLODSHED SOFTWARE. **Dev C++.** 2000. Disponível em: <http://www.bloodshed.net/index.html>. Acesso em: nov. 2005.

BORDLAND SOFTWARE CORPORATION. **Bordland Delphi**. 1995. Disponível em: <http://www.borland.com/us/products/delphi/index.html>. Acesso em: nov. 2005.

BRISOLARA, L. B.; BECKER, L. B.; CARRO, L.; WAGNER, F. R.; PEREIRA, C. E.; REIS, R. A. L. Comparing High-level Modeling Approaches for Embedded Systems Design. In: ASIA SOUTH PACIFIC DESIGN AUTOMATION CONFERENCE, 2005, China. **Proceedings...** Piscataway, NJ: IEEE, 2005. v. 2, p. 986-989.

BYU (Brigham Young University). **JHDL Home Page**. 1998. Disponível em: <http://www.jhdl.org/>. Acesso em: nov. 2005.

CHAND, M. **COM+ Tutorial from Scratch: Part I.** Disponível em: <http://www.mindcracker.com/mindcracker/c_cafe/com+/comp.asp>. Acesso em: mar. 2005.

CLIOSOFT. **Integration of SOS with Summit's Visual-HDL**. 2003. Disponível em: <http://www.cliosoft.com/products/visualhdl.html>. Acesso em nov. 2005.

CRNKOVIC, I. **Building Reliable Component-Based Software Systems.** Norwood, USA: Artech House. 2002

CRNKOVIC, I.; HNICH, B.; JONSSON, T.; KIZILTAN, Z. Specification, implementation, and deployment of components. **Communicatios of ACM**, New York, v. 45, n.10, 2002.

DOULOS LTD. **What is Verilog?** 2005. Disponível em: <http://www.doulos.com/knowhow/verilog_designers_guide/what_is_verilog>. Acesso em: nov. 2005.

EDDON, G.; EDDON, H. **Inside Distributed COM.** Redmond, Washington, EUA: Microsoft Press, 1998. 552 p.

FUGETTA, A. A classification of case technology. **Computer**, Los Alamitos, v.26, n.12, p.25-38, 1993.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. **Padrões de Projeto:** Soluções Reutilizáveis de Software Orientado a Objetos**.** Porto Alegre: Bookman. 2000. 364 p.

GEER, D. Eclipse Becomes the Dominant Java IDE. **Computer,** Los Alamitos, v.38, n.7, p 16-18, July 2005.

GENTLEWARE AG. **Poseidon for UML**. 2003. Disponível em: <http://gentleware.com/index.php?id=download>. Acesso em: nov. 2005.

GOKHALE, B.; KUMAR, A. D.; SAHUGUET, A. Reinventing the Wheel? CORBA vs. Web Services. In: INTERNATIONAL WORLD WIDE WEB CONFERENCE, WWW, 2002. **Proceedings...** [S.l.:s.n.] 2002.

GREEN, D. Internationalization: Checklist. In: HUML, A.; WALRATH, K.; CAMPIONE, M. **The Java Tutorial,** 3rd ed. Addison-Wesley, 2000.

HANSEN, C.; BRINGMANN , O.; ROSENSTIEL, W. A VHDL Component Model for Mixed Abstraction Level Simulation and Behavioral Synthesis. In: FORUM ON DESIGN LANGUAGES, FDL, 1999. **Proceedings...** [S.l.:s.n.]. 1999.

HILLMANN, D. **Using Dublin Core.** 2003. Disponível em: <http://dublincore.org/documents/usageguide/#whatismetadata>. Acesso em: mar. 2005.

HOLLAND, I. M. Specifying Reusable Components Using Contracts. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, ECOOP, 1992. **Proceedings...** [S.l.:s.n.] 1992. p 287-308.

HOWARD, A. Rapid Application Development: rough and dirty or value-for-money engineering? **Communications of the ACM**, New York, v. 45, n. 10. p. 27 - 29. October 2002

HUGHES, C. **Linux rapid application development**. Foster City, Ca: IDG Books Worldwide, 2000. 616 p.

HUIZING, M. Component Based Development. **Xootic Magazine,** [s.l]**..** 1999.

HUTCHINGS, B.; BELLOWS, P.; HAWKINS, J.; HEMMERT, S.; NELSON, B.; RYTTING, M. A CAD Suite for High-Performance FPGA Design. In: ANNUAL IEEE SYMPOSIUM ON FIELD-PROGRAMMABLE CUSTOM COMPUTING MACHINES, FCCM, 1999. **Proceedings...** [S.l.]: IEEE Computer Society, 1999.

INDRUSIAK, L. S.; REIS, R. A. L.; A WWW approach for EDA Tool Integration. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, SBCCI, 10., 1997, Gramado. **Proceedings...** Porto Alegre: CPGCC da UFRGS, 1997.

JACOME, M. F.; PEIXOTO, H. A Survey of Digital Design Reuse. **IEEE Design & Test**, Califórnia, v. 18, n. 3, p. 98-107, 2001.

JEDIT. **Home Page do Projeto jEdit**.1999. Disponível em <http://www.jedit.org/>. Acesso em: dez. 2005.

JETBRAINS. **IntelliJ IDEA.** 2001. Disponível em: <http://www.jetbrains.com/idea/>. Acesso em: nov. 2005.

JUNG, H.; KIM, D.; YANG, Y.; LEE, S. Design and Implementation of Object-oriented Framework-based RAD Tool (INTRAD). In: IEEE CONFERENCE ON SYSTEMS, MAN, AND CYBERNETICS, 2000. **Proceedings...** Piscataway: IEEE, 2000. p.2057 – 2061.

KEATING, M.;BRICAUD, P. **Reuse methodology manual:** for system-on-a-chip designs. 3rd ed. [S.l.]: Kluwer Academic Publishers, 2002. 291p.

KISSION, P.; DING, H.; JERRAYA, A. A.. VHDL based design methodology for hierarchy and component re-use. In EUROPEAN DESIGN AUTOMATION CONFERENCE, 1995, Brighton, Grã-Bretanha. **Proceedings…** Los Alamitos: IEEE Computer Society Press, 1995. p. 470-475.

KOPLIN, CLEBER; SERENI, GUIDO; BOLZAN, ILSON. **Plataforma Web para Suporte a Desenvolvimento de Componentes de Hardware**. 2005. Trabalho de Conclusão (Curso Técnico em Informática) - Escola José César de Mesquita, Porto Alegre.

LEE, W. F. **VHDL Code and Synthesis with Synopsys**. London: Academic Press, 2000.

LINGOPORT. **Globalizer Internationalization Development Environment**. 2005. Disponível em: <http://www.lingoport.com/public/index2.cfm?catID=4>. Acesso em: nov. 2005.

LOER, C.; ROSENBLUM, D. S. WREN - an environment for component-based development. In: EUROPEAN SOFTWARE ENGINEERING CONFERENCE, ACM SIGSOFT INTERNATIONAL SYMPOSIUM ON FOUNDATIONS OF SOFTWARE ENGINEERING, 9., 2001. **Proceedings...** New York: ACM Press. 2001. p 207-217.

MARTIN, J. **Rapid Application Development**. [S.l.]: Macmillan Publishing. 1991.

MCFARLAND, C.H.; HAGGARD, R.L. A discussion of the problems with current VHDL implementations. In: SOUTHEASTCON, 1993, **Proceedings...**, IEEE, Vol., Iss., 4-7 Apr 1993

MCILROY, M. D. Mass Produced Software Componentes. In: NAUR, P.; RANDELL, B. (Ed.). **Software Engineering**. [S.l.]: NATO Science Committee, 1968.

MEYER, B. On To Components. **Computer,** Los Alamitos, v. 32, n. 1, p 139-140. 1999.

MICROSOFT CORPORATION. **DCOM Technical Overview.** 2002. Disponível em: <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndcom/html/msdn_dcomtec.asp>. Acesso em: mar. 2005.

MICROSOFT CORPORATION. **Microsoft Visual Basic**. 1991. Disponível em: <http://msdn.microsoft.com/vbasic/>. Acesso em: mar. 2005.

MICROSOFT CORPORATION. **MSDN Library.** 2005. Disponível em: <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/com/html/d17dc0dd-3115-4830-8c6b-694a8d1accaa.asp>. Acesso em: mar. 2005.

MORCH, A. I. et al. Component-based technologies for end-user development. **Communications of the ACM**, New York, v. 47, n. 9. 2004

NIERSTRASZ, O.; DAMI, L. Component-Oriented Software Technology. In: NIERSTRASZ, O.; TSICHRITZIS, D. (Ed.). **Object-Oriented Software Composition**. [S.l.]: Prentice Hall, 1995. p.3-28

OBJECT MANAGEMENT GROUP. **Unified Modelling Language Specification**. 1997. Disponível em: <http://www.omg.org/technology/documents/modeling_spec_catalog.htm#UML>. Acesso em: nov. 2005.

OMG. **Corba Basics.** 2002. Disponível em: <http://www.omg.org/gettingstarted/corbafaq.htm>. Acesso em: mar. 2005.

OMG. **Corba Components, v3.0.** 2005. Disponível em: <http://www.omg.org/technology/documents/formal/components.htm>. Acesso em: mar. 2005.

OPENCORES.ORG. **Free Open Source IP Cores and Chip Design**. 1999. Disponível em: <www.opencores.org>. Acesso em: nov. de 2005.

OSTG. **SouceForge.Net**. 2001. Disponível em: <http://sourceforge.net/>. Acesso em: nov. 2005.

PATTISON, T. **Automating COM+ Administration.** 2000 Disponível em: <http://msdn.microsoft.com/library/default.asp?url=/msdnmag/issues/0900/instincts/toc.asp>. Acesso em: mar. 2005.

PREIS, V.; HENFTLING, R.; SCHÜTZ, M.; MÄRZ-RÖSSEL, S. A reuse scenario for the VHDL-based hardware design flow. In: CONFERENCE ON EUROPEAN DESIGN AUTOMATION, 1995, Brighton, Grã-Bretanha. **Proceedings...** Los Alamitos: IEEE Computer Society Press, 1995. p. 464-469.

PRESSMAN, R. S. **Engenharia de Software**. Rio de Janeiro: McGraw-Hill, 2002. 843p.

PRIDMORE, JEFF; BUCHANAN, GREG; CARACCIOLO, GERRY; WEDGWOOD, JANET. Model-Year Architectures for Rapid Prototyping. **Journal of VLSI Signal Processing Systems**, Boston, v. 15, n. 1-2, p. 83-96, 1997.

PUTZKE-RÖMING, W.; NEBEL, W. **Objective VHDL based Communication Design for Object-Oriented Hardware Design**. 2000. 16p. Final Report on the OFFIS research project OO-COM, Escherweg, Alemanha.

REILLY, J. P.; CARMEL, E. Point-Counterpoint: Does RAD Live Up to the Hype? **IEEE Software,** Los Alamitos, v. 12, n. 5, p. 24-26, September 1995.

REIS, C.; CARTWRIGHT, R. Taming a professional IDE for the classroom. In: SIGCSE TECHNICAL SYMPOSIUM ON COMPUTER SCIENCE EDUCATION, 2004. **Proceedings...** New York: ACM Press, 2004. p. 156-160.

ROWSON, J. A.; SANGIOVANNI-VINCENTELLI, A. L.. Interface-based design. In: ANNUAL CONFERENCE ON DESIGN AUTOMATION, DAC, 34., 1997. **Proceedings...** 1997. New York: ACM Press, 1997. p. 178-183.

SANGIOVANNI-VINCENTELLI, A. L.; MCGEER, P. C.; SALDANHA, A. Verification of electronic systems. In: ANNUAL CONFERENCE ON DESIGN AUTOMATION, DAC, 33., 1996. **Proceedings...** New York: ACM Press, 1996. p. 106-111.

SAWICKI, S.; INDRUSIAK, L. S.; REIS, R. A. L. Projeto Cooperativo no Ambiente Cave. In: WORKSHOP IBERCHIP, 8., 2002. **Proceedings...** Guadalajara, México: CINVESTAV, 2002.

SI2. **Silicon Integration Initiative Home Page**. Disponível em: <http://www.si2.org/>. Acesso em: dez. 2005.

SKAHILL, K. Optimizing VHDL designs for programmable logic. **Electronics Engineer**, Março 1998

SOMMERVILLE, I. **Engenharia de Software**. São Paulo: Addison Wesley, 2003. 592p.

SUMMIT DESIGN INC. **Text-to-Graphics Primer**. 1997. Disponível em: <http://wwwce.web.cern.ch/wwwce/dc/Visual/Visual_doc/T2g_vhdl.pdf>. Acesso em nov. 2005.

SUN MICROSYSTEMS. **JavaBeans Specification**. 1997. Disponível em: <http://java.sun.com/products/javabeans/docs/spec.html>. Acesso em: nov. 2005.

SZYPERSKI, C. **Component Software:** Beyond Object-Oriented Programming**.** 2nd ed. London: Addison-Wesley, 2002.

SZYPERSKI, C. Component Technology - What, Where and How? In: CONFERENCE ON SOFTWARE ENGINEERING, ICSE, 2003. **Proceedings...** [S.l., s.n.], 2003.

UDELL, J. Componentware. **Byte**, Peterborough, v. 19, n. 5, p. 46–56, May 1994.

VAUGHAN-NICHOLS, S. J. The Battle over the Universal Java IDE. **Computer,** Los Alamitos, v. 36, n. 4, p. 21-23, 2003.

VSI ALLIANCE. **About VSIA**. Disponível em: <http://www.vsia.org/aboutVSIA/index.htm>. Acesso em: nov. 2005.

WAGNER, F. R..; CESÁRIO, W. O.; CARRO, L.; JERRAYA, A. A. Strategies for the integration of hardware and software IP components in embedded systems-on-chip. **Integration: the VLSI Journal**, Amsterdan, v. 37, n. 4, p. 223-252. September. 2004.

WANG, N.; SCHMIDT, D. C.; O'RYAN, C. Overview of the CORBA Component Model. In: HEINEMAN, G.; COUNCILL, W. (Ed.). **Component-Based Software Engineering – Putting the Pieces Together**. [S.l.]: Addison-Wesley, 2001. p.303-318.

WEINREICH, R.; SAMETINGER, J. Component Models and Component Services: Concepts and Principles. In: HEINEMAN, G.; COUNCILL, W. (Ed.). **Component-Based Software Engineering – Putting the Pieces Together**. [S.l.]: Addison-Wesley, 2001. p.33-48.

WESTE, N. H. E.; ESHRAGHIAN, K. **Principles of CMOS VLSI Design**. [S.l.]: Addison-Wesley, 1993.

WILLIAMS, J. The Business Case for Components. In: HEINEMAN, G.; COUNCILL, W. (Ed.). **Component-Based Software Engineering – Putting the Pieces Together**. [S.l.]: Addison-Wesley, 2001. p.79-97

XINOX SOFTWARE. **JCreator.** 2000. Disponível em: <http://www.jcreator.com/>. Acesso em: nov. 2005.

ZEID, A.; MESSIHA, M.; YOUSSSEF, S. Applicabiliy of Component-Based Development in High-Performance Systems. In: INTERNATIONAL WORKSHOP ON COMPONENT-ORIENTED PROGRAMMING, WCOP, 9., 2004. **Proceedings...** [S.l.,s.n.], 2004.

ZUBECK, J. C. Implementing Reuse with RAD Tools' Native Objects. **Computer,** Los Alamitos, v. 30, n. 10, p. 60-65. October. 1997.