UNIVERSIDADE DE BRASÍLIA - UnB
FACULDADE DE TECNOLOGIA - FT
DEPARTAMENTO DE ENGENHARIA ELÉTRICA - ENE

IMPLEMENTAÇÃO DE UMA INTERFACE NEUTRA CLIENTE-SERVIDOR DEDICADA À SIMULAÇÃO E DOS MÓDULOS DE SISTEMAS MÓVEIS E CIRCUITOS ELÉTRICOS

ALEX HELDER CORDEIRO DE OLIVEIRA
HERMÓGENES BATISTA CORREIA
RENATO FARIA IIDA

ORIENTADOR: PAULO HENRIQUE PORTELA DE CARVALHO

PROJETO FINAL DE GRADUAÇÃO EM
ENGENHARIA DE REDES DE COMUNICAÇÃO

BRASÍLIA / DF
SETEMBRO/2002

AGRADECIMENTOS

Agradecemos a nossos pais e amigos por muito nos ajudar neste trabalho, ao nosso orientador Paulo Henrique Portela de Carvalho, por toda a paciência que teve durante este tempo, por expandir os limites de nossas capacidades, além de ter acreditado em nós, confiando-nos tamanha e nos ter confiado tamanha tarefa. Aos funcionários do Departamento e a todos aqueles que, de alguma forma, contribuíram para o desenvolvimento deste trabalho. Não podemos esquecer a ajuda do Marçal Chaiben e toda sua família pelo apoio completo no desenvolvimento desse projeto. E, especialmente, a Deus, que a cada dia nos fortalece, a fim de aprendermos sempre com nossas provações e regozijarmos com as vitórias. Agradecemos a ti, Senhor, sempre.

DEDICATÓRIA

Dedico esse trabalho aos meus pais, Anna e Itiro, pois eles sempre me apoiaram principalmente agora para terminar essa etapa da minha vida. Espero sempre superar as expectativas deles. Não posso esquecer de agradecer a paciência da Giza nessa etapa final.

Obrigado a todos.

Renato Faria Iida

Consagro este trabalho primeiramente à minha família, pelo apoio e carinho que nunca me faltaram, as palavras de consolo, de advertência e de ensinamento que me fizeram crescer, e que sempre estiveram presentes ao meu lado, seja nas vitórias ou nas derrotas. Dedico este trabalho também aos meus verdadeiros amigos, que me dão a força para seguir em frente.

Hermógenes Batista Correia

Dedico este projeto aos meus pais e a meu irmão, que me levaram a seguir este caminho e me deram o apoio necessário para prosseguir. Muito obrigado.

Alex Helder Cordeiro de Oliveira

RESUMO

O objetivo deste trabalho é a modelagem, estudo e implementação de uma interface cliente-servidor neutra de simulação, e dos módulos de simulação de circuitos e sistemas celulares. Este trabalho se dividiu em duas etapas distintas. A primeira consistiu da modelagem da interface e do estudo das teorias referentes a este processo e a segunda se baseia na implementação deste sistema junto com o estudo de ferramentas adequadas à execução desta. A implementação, por sua vez, se dividiu nas criações do servidor, do banco de dados, da interface principal, do módulo de simulação de circuitos elétricos e do módulo de simulação de sistemas celulares.

Java foi escolhida como linguagem de desenvolvimento por ser uma linguagem orientada a objeto e multiplataforma, sendo possível a execução do programa implementado em qualquer sistema que suporte a máquina virtual Java (JVM). Para a modelagem da interface e do ambiente cliente-servidor, foi utilizada a linguagem de modelagem UML. Foram definidos sete objetos fundamentais para a execução deste trabalho: projeto, sistema, ambiente, comportamento, arquitetura, componente e subprojeto. Foram especificadas três entidades para o ambiente: o cliente, o servidor de simulação e o banco de dados. Para a conexão entre o servidor de simulação e o banco de dados foi escolhido o driver JDBC 100% puro, por ter apresentado a melhor performance dentre os drivers estudados. Para a interface entre o servidor de simulação e o cliente, foi usado o FTP para o envio de dados da simulação de circuitos e os Métodos Remotos Java (RMI-IIOP) para o envio de dados da simulação de sistemas celulares.

O módulo de simulação de circuitos consiste do acréscimo de algumas classes para a adaptação deste módulo de forma a não prejudicar a modularidade da interface. A simulação de um circuito se faz em duas etapas. Na primeira, é feita uma conversão das classes da interface neutra para a linguagem própria do algoritmo de simulação utilizada. Em seguida, estes dados são enviados ao simulador por meio de ferramentas de FTP.

No módulo de simulação de sistemas celulares, os cálculos são feitos a partir de um "mapa" composto por uma matriz bidimensional onde os índices correspondem às posições das coordenadas de latitude e longitude e os valores correspondem às altitudes destes locais. Com estes dados e uma lista das ERB é feito o cálculo da potência das ERB para cada ponto do mapa utilizando o modelo de Lee. O resultado dos cálculos é então quantizado em uma escala de cores para uma melhor visualização do resultado da simulação.

ABSTRACT

The objective of this work is the modeling, study and implementation of a neutral interface client-server of simulation, and the modules of simulation of circuits and cellular systems. This work is divided in two distinct stages: The first one is consisted of the modeling of the interface and the study of the referring theories to this process, and the second is based on the implementation of this system with the study of adequate tools to the execution of this. The implementation, in turn, was divided on the creation of the server, the database, the main interface, the module of simulation of electrical circuits and the module of simulation of cellular systems.

Java was chosen as the development language for being an object oriented language and a multiplatform, being possible the execution of the program implemented in any system that has supported the "Java Virtual Machine". For the modeling of the interface and the client-server environment, the UML language was used for this purpose. It had been defined seven basic objects for the execution of this work: project, system, environment, behavior, architecture, component and subprojetct.

Three entities for the environment had been specified: the customer, the simulation server and the database. For the connection between the simulation server and the database, it was chosen the driver JDBC 100% pure, for having presented the best performance among the drivers studied. For the interface between the simulation server and the client, it was used the FTP to send the simulation of circuits and Remote Methods Invocation (RMI-IIOP) to send the simulation of cellular systems.

The module of simulation of circuits consists of the addition of some classes for the adaptation of this module to not harm the modularity of the interface. The simulation of a circuit split in two stages: in the first one, a conversion of the classes of the neutral interface for the proper language of simulation algorithm used. After that, these data are sent to the simulation server by means of FTP tools.

In the module of simulation of cellular systems, the calculations are made from a "map" of a bidimensional array where the indices correspond to the positions of the latitude and longitude coordinates and the values correspond to the altitudes of these points. With these data and a list of the ERBs, it is made the calculation of the power of the ERB for each point of the map using Lee model. The result of the calculations then is quantized in a scale of colors for a better visualization of the result of the simulation.

ÍNDICE

GLOSSÁRIO

JVM(*Java Virtual Machine*) Máquina Virtual Java

API(*Application Programming Interface)*- Interface de programação de aplicativos

GUIs (*graphical user interface*) - Interface gráfica do usuário,

ODBC(Open Data Base Connectivity) – Conectividade aberta a Banco de Dados

AWT (*Abstract Window Toolkit)* – Kit de ferramentas de janelas abstratas

CORBA(Common Object Request Broker Architecture) - Arquitetura do Mediador de Requisições de Objetos Comuns

IDL(Interface Definition Language) – Linguagem de definição de interface

ORB(Object Request Broker)- Mediador de Requisições de Objetos

FTP(File Transfer Protocol) – Arquitetura do Mediador de Requisições de Objetos Comuns

UML(Unified Modeling Language)- Linguagem unificada de modelagem

ÍNDICE DE FIGURAS

ÍNDICE DE TABELAS

1 INTRODUÇÃO

Programas de simulação possuem uma grande aceitação no mercado devido à facilidade que criam para os engenheiros e projetistas, ao simular matematicamente o provável comportamento do projeto.[1] As simulações procuradas são diversas, desde tráfego de redes de computadores às características elétricas de diodos e transistores. Dentre os diversos tipos de simulação, duas que tem grande destaque são as simulações de circuitos elétricos e de sistemas celulares.

Tal qual a demanda por simuladores aumentou, assim também aconteceu com sua quantidade e complexidade. Estes softwares estão requerendo uma grande capacidade de processamento. Esta necessidade de processamento unido ao crescente desenvolvimento das redes de computadores, que estão se tornando sempre mais rápidas e confiáveis, vem a tornar viável a utilização de sistemas de processamento centralizado, isto é, vários computadores utilizando um computador de grande porte para executar estas operações. [1]

Partindo deste cenário, percebeu-se a importância de projetar um ambiente cliente-servidor de simulação neutro, onde possam ser implementados variados tipos de simuladores dentro de uma interface única, sendo que cada ambiente de simulação é um módulo desta interface.A solicitação de simulação gerada pelo cliente deve ser passada a um servidor juntamente com os dados necessários à realização desta simulação, para que este servidor faça a interface com o simulador e o banco de dados para gerar os resultados e os envie para o cliente para apresentá-los ao usuário.

Foi definido como objetivo, então, a modelagem e implementação deste sistema, juntamente com o desenvolvimento dos módulos de simulação de sistemas celulares e circuitos elétricos. Estes módulos devem ser integrados a este sistema para validar esta estrutura neutra. A tarefa foi realizada por um grupo de quatro alunos, sendo a fase de modelagem do sistema realizada em conjunto por todos os integrantes do grupo, e a implementação divida em quatro partes distintas que foram divididas entre os componentes.

A implementação consistiu da criação dos seguintes itens:

- O Servidor, que gerencia todo o processo de simulação;
- O banco de dados, para armazenar importantes dados sobre os usuários e projetos criados;
- A interface principal do programa, que se mantém a mesma, independente da simulação desejada;

- O módulo de simulação de circuitos elétricos;
- O módulo de simulação de sistemas celulares.

A etapa do banco de dados já foi previamente concluída em outro projeto final de graduação, e podem ser encontradas mais referências sobre ela em [1]. As outras partes desta implementação estão descritas neste relatório.

Este trabalho vem a detalhar toda a modelagem, estudo e implementação do programa desejado, que foi denominado ProgSim (Programa de Simulação). A organização deste relatório foi feita em sete capítulos, sendo o primeiro esta introdução e o último a conclusão.

No Capítulo 2, tem-se a descrição da modelagem do sistema, juntamente com as ferramentas e conceitos estudados para criação e entendimento da modelagem, além da implementação da interface principal, que foi realizada por todo o grupo. O Capítulo 3 detalha a estrutura cliente-servidor adotada, junto com todas as ferramentas utilizadas e estudadas para a criação desta estrutura. O capítulo 4 vem a especificar a estruturação e criação do módulo de simulação de circuitos. No capítulo 5 será apresentada a modelagem e os processos de criação e implementação do módulo de simulação de sistemas celulares. O capítulo 6 apresenta o estudo e criação do servidor simulação e suas ferramentas básicas.

2 FILOSOFIA DOS OBJETOS E DA INTERFACE DO PROGSIM

2.1 *Introdução*

Para compreender os capítulos posteriores desse trabalho, faz-se necessário uma base teórica apresentada nas primeiras partes desse capitulo. Com base nesse conhecimento, serão apresentados os objetos básicos que irão formar um projeto de simulação, além de explicar a parte da interface ProgSim, que será o suporte para qualquer módulo de simulação.

2.2 *Programação Orientada a Objeto [2]*

Programação orientada a objetos (POO) é uma metodologia de programação adequada ao desenvolvimento de sistemas de grande porte, provendo modularidade e reusabilidade. A POO introduz uma abordagem na qual o programador visualiza seu programa em execução como uma coleção de objetos cooperantes que se comunicam através de mensagens. Cada um dos objetos é instância de uma classe e todas as classes formam uma hierarquia de classes unidas via relacionamento de herança. Existem alguns aspectos importantes na definição de POO:

- Usa *objetos*, e não funções ou procedimentos como seu bloco lógico fundamental de construção de programas.
- Objetos comunicam-se através de *mensagens*.
- Cada objeto é instância de uma *classe*.
- Classes estão relacionadas às outras via mecanismos de *herança*.

Programação orientada a objetos dá ênfase à estrutura de dados, adicionando funcionalidade ou capacidade de processamento a estas estruturas. Em linguagens tradicionais, a importância maior é atribuída a processos e sua implementação em subprogramas. Em linguagens orientadas a objetos, ao invés de passar dados a procedimentos, requisita-se que objetos realizem operações neles próprios.

Alguns aspectos são fundamentais na definição de programação orientada a objetos:

Objetos

Na visão de uma linguagem imperativa tradicional (estruturada), os *objetos* aparecem como uma única entidade autônoma que combina a representação da informação (estruturas

de dados) e sua manipulação (procedimentos), uma vez que possuem capacidade de processamento e armazenam um estado local. Pode-se dizer que um objeto é composto de:

- Propriedades: são as informações, estruturas de dados que representam o estado interno do objeto. Em geral, não são acessíveis aos demais objetos.
- Comportamento: conjunto de operações, chamados de métodos, que agem sobre as propriedades. Os métodos são ativados (disparados) quando o objeto recebe uma mensagem solicitando sua execução. Embora não seja obrigatório, em geral uma mensagem recebe o mesmo nome do método que ela dispara. O conjunto de mensagens que um objeto está apto a receber está definido na sua interface.
- Identidade: é uma propriedade que diferencia um objeto de outro; ou seja, seu nome.

Enquanto que os conceitos de dados e procedimentos são freqüentemente tratados separadamente nas linguagens de programação tradicionais, em POO eles são reunidos em uma única entidade: o objeto. A Figura 2.1 apresenta uma visualização para um objeto.

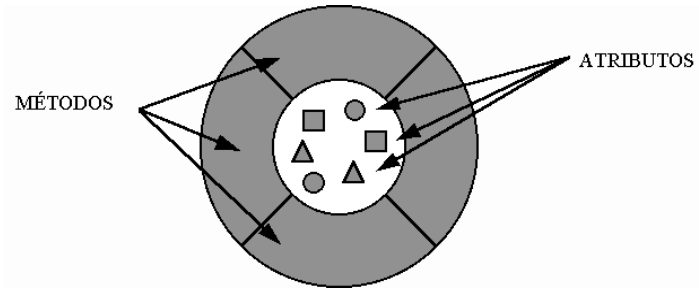

Figura 2.1 - Representação para um objeto

No mundo real não é difícil a identificação de objetos (em termos de sistemas, objetos são todas as entidades que podem ser modeladas, não apenas os nossos conhecidos objetos inanimados).

Uma vez que objetos utilizam o princípio da abstração de dados, o encapsulamento de informação proporciona dois benefícios principais para o desenvolvimento de sistemas:

- Modularidade: o código fonte para um objeto pode ser escrito e mantido independentemente da código fonte de outros objetos. Além disso, um objeto pode ser facilmente migrado para outros sistemas.
- Ocultamento de informação: um objeto tem uma interface pública que os outros objetos podem utilizar para estabelecer comunicação com ele. Mas, o objeto mantém informações e métodos privados que podem ser alterados a qualquer hora sem afetar os outros objetos que dependem dele. Ou seja, não é necessário saber como o objeto é implementado para poder utilizá-lo.

Mensagens

Um objeto sozinho não é muito útil e geralmente ele aparece como um componente de um grande programa que contém muitos outros objetos. Através da interação destes objetos pode-se obter uma grande funcionalidade e comportamentos mais complexos.

Objetos de software interagem e comunicam-se com os outros através de mensagens. Quando o objeto A deseja que o objeto B execute um de seus métodos, o objeto A envia uma mensagem ao objeto B. Algumas vezes o objeto receptor precisa de mais informação para que ele saiba exatamente o que deve fazer; esta informação é transmitida juntamente com a mensagem através de *parâmetros.*

Uma mensagem é formada por três componentes básicos:

- O objeto a quem a mensagem é endereçada (receptor)
- O nome do método que se deseja executar
- Os parâmetros (se existirem) necessários ao método

Classe

"É a definição dos atributos e funções de um tipo de objeto. Cada objeto individual é então criado com base no que está definido na classe. Por exemplo, *homo sapiens* é uma classe de mamífero; cada ser humano individual é um objeto dessa classe."

Objetos de estrutura e comportamento idênticos são descritos como pertencendo a uma classe, de tal forma que a descrição de suas propriedades pode ser feita de uma só vez, de forma concisa, independente do número de objetos idênticos em termos de estrutura e comportamento que possam existir em uma aplicação. A noção de um objeto é equivalente ao conceito de uma variável em programação convencional, pois especifica uma área de armazenamento, enquanto que a classe é vista como um tipo abstrato de dados, uma vez que representa a definição de um tipo.

Cada objeto criado a partir de uma classe é denominado de *instância* dessa classe. Uma classe provê toda a informação necessária para construir e utilizar objetos de um tipo, cada instância pertence a uma classe e uma classe pode possuir múltiplas instâncias. Devido ao fato de todas as instâncias de uma classe compartilharem as mesmas operações, qualquer diferença de respostas a mensagens aceitas por elas é determinada pelos valores das variáveis de instância.

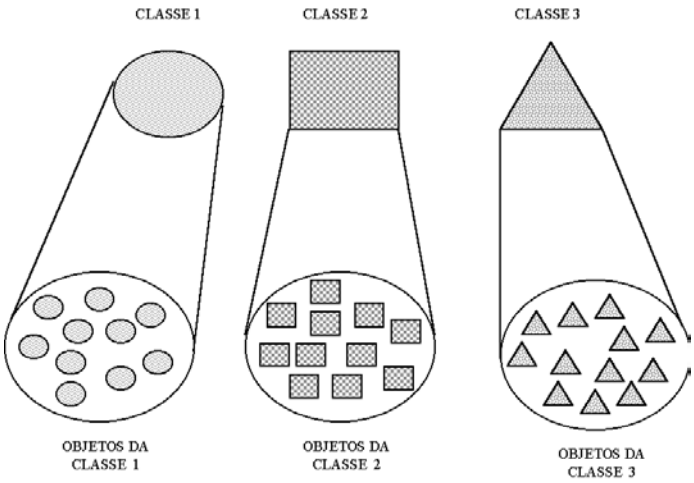

Figura 2.2 - Relacionamento entre classes e objetos

A Figura 2.2 ilustra o relacionamento entre classes e objetos. Cada objeto instanciado a partir de uma classe possui as propriedades e comportamento definidos na classe, da mesma maneira que uma variável incorpora as características do seu tipo. A existência de classes proporciona um ganho em reusabilidade, pois o código das operações e a especificação da estrutura de um número potencialmente infinito de objetos estão definidos em um único local, a classe. Cada vez que um novo objeto é instanciado ou que uma mensagem é enviada, a definição da classe é reutilizada. Caso não existissem classes, para cada novo objeto criado, seria preciso uma definição completa do objeto.

O maior benefício proporcionado pela utilização das classes é a reusabilidade de código, uma vez que todos os objetos instanciados a partir dela incorporam as suas propriedades e seu comportamento.

Metaclasses

Uma metaclasse é uma classe de classes. Pode-se julgar conveniente que, em uma linguagem ou ambiente, classes também possam ser manipuladas como objetos. Por exemplo, uma classe pode conter variáveis contendo informações úteis, como:

- o número de objetos que tenham sido instanciados da classe até certo instante;
- um valor médio de determinada propriedade, calculado sobre os valores específicos desta propriedade nas instâncias (por exemplo, média de idade de empregados).

Métodos

Um método implementa algum aspecto do comportamento do objeto. Comportamento é a forma como um objeto age e reage, em termos das suas trocas de estado e troca de mensagens.

Um método é uma função ou procedimento que é definido na classe e tipicamente pode acessar o estado interno de um objeto da classe para realizar alguma operação. Pode ser pensado como sendo um procedimento cujo primeiro parâmetro é o objeto no qual deve trabalhar. Este objeto é chamado receptor. Abaixo é apresentada uma notação possível para o envio de uma mensagem (invocação do método).

Construtores são usados para criar e inicializar objetos novos. Tipicamente, a inicialização é baseada em valores passados como parâmetros para o construtor. Destrutores são usados para destruir objetos. Quando um destrutor é invocado, as ações definidas pelo usuário são executadas, e então a área de memória alocada para o objeto é liberada. Em algumas linguagens, como C++, o construtor é chamado automaticamente quando um objeto é declarado. Em outras, como Object Pascal, é necessário chamar explicitamente o construtor antes de poder utilizá-lo.

Um exemplo de utilização de construtores e destrutores seria gerenciar a quantidade de objetos de uma determinada classe que já foram criados até o momento. No construtor pode-se colocar código para incrementar uma variável e no destrutor o código para decrementá-la.

Herança

O conceito de herança é fundamental na técnica de orientação a objetos. A herança permite criar um novo tipo de objeto - uma nova classe - a partir de outra já existente.

A nova classe mantém os atributos e a funcionalidade da classe da qual deriva; por isso, dizemos que ela "herda" as características daquela classe. Ao mesmo tempo, ela pode receber atributos e funções especiais não encontrados na classe original.

Uma das vantagens da herança é a facilidade de localizar erros de programação. Por exemplo, caso um objeto derivado de outro apresente um erro de funcionamento; se o objeto original funcionava corretamente, é claro que o erro está na parte do código que implementa as novas características do objeto derivado. A herança permite, também, reaproveitar o código escrito anteriormente, adaptando-o às novas necessidades.

Isso é muito importante porque os custos de desenvolvimento de software são muitos elevados. A mão-de-obra altamente especializada é cara; o processo é demorado e sujeito a ocorrências inesperadas.

Polimorfismo

Polimorfismo refere-se à capacidade de dois ou mais objetos responderem à mesma mensagem, cada um a seu próprio modo. A utilização da herança torna-se fácil com o polimorfismo. Desde que não seja necessário escrever um método com nome diferente para responder a cada mensagem, o código é mais fácil de entender.

Outra forma simples de polimorfismo permite a existência de vários métodos com o mesmo nome, definidos na mesma classe, que se diferenciam pelo tipo ou número de parâmetros suportados. Isto é conhecido como *polimorfismo paramétrico*, ou *sobrecarga de operadores* ("overloading"). Neste caso, uma mensagem poderia ser enviada a um objeto com parâmetros de tipos diferentes (uma vez inteiro, outra real, por exemplo), ou com número variável de parâmetros. O nome da mensagem seria o mesmo, porém o método invocado seria escolhido de acordo com os parâmetros enviados.

Alguns benefícios proporcionados pelo polimorfismo:

- Legibilidade do código: a utilização do mesmo nome de método para vários objetos torna o código de mais fácil leitura e assimilação, facilitando muito a expansão e manutenção dos sistemas.
- Código de menor tamanho: o código mais claro torna-se também mais enxuto e elegante. Pode-se resolver os mesmos problemas da programação convencional com um código de tamanho reduzido.

Vantagens da POO

A POO tem alcançado tanta popularidade, devido às vantagens que ela traz. A reusabilidade de código é, sem dúvida, reconhecida como a maior vantagem da utilização de POO, pois permite que programas sejam escritos mais rapidamente. Todas as empresas sofrem de deficiência em seus sistemas informatizados para obter maior agilidade e prestar melhores serviços a seus clientes. Um levantamento feito na AT&T, a gigante das telecomunicações nos EUA, identificou uma deficiência da ordem de bilhões de linhas de código. Uma vez que a demanda está sempre aumentando, procura-se maneiras de desenvolver sistemas mais rapidamente, o que está gerando uma série de novas metodologias

e técnicas de construção de sistemas (por exemplo, ferramentas CASE). A POO, através da reusabilidade de código, traz uma contribuição imensa nesta área, possibilitando o desenvolvimento de novos sistemas utilizando-se muito código já existente. A maior contribuição para reusabilidade de código é apresentada pela herança.

Escalabilidade pode ser vista como a capacidade de uma aplicação crescer facilmente sem aumentar demasiadamente a sua complexidade ou comprometer o seu desempenho. A POO é adequada ao desenvolvimento de grandes sistemas uma vez que pode-se construir e ampliar um sistema agrupando objetos e fazendo-os trocar mensagens entre si. Esta visão de sistema é uniforme, seja para pequenos ou grandes sistemas (logicamente, deve-se guardar as devidas proporções).

O encapsulamento proporciona ocultamento e proteção da informação. Acessos a objetos somente podem ser realizados através das mensagens que ele está habilitado a receber. Nenhum objeto pode manipular diretamente o estado interno de outro objeto. De modo que, se houver necessidade de alterar as propriedades de um objeto ou a implementação de algum método, os outros objetos não sofrerão nenhum impacto, desde que a interface permaneça idêntica. Isto diminui em grande parte os esforços despendidos em manutenção. Além disso, para utilizar um objeto, o programador não necessita conhecer a fundo a sua implementação.

O polimorfismo torna o programa mais enxuto, claro e fácil de compreender. Sem polimorfismo, seriam necessárias listas enormes de métodos com nomes diferentes mas comportamento similar. Na programação, a escolha de um entre os vários métodos seria realizada por estruturas de múltipla escolha (*case*) muito grandes. Em termos de manutenção, isto significa que o programa será mais facilmente entendido e alterado.

A herança também torna a manutenção mais fácil. Se uma aplicação precisa de alguma funcionalidade adicional, não é necessário alterar o código atual. Simplesmente cria-se uma nova geração de uma classe, herdando o comportamento antigo, adicionando-se novo comportamento ou redefinindo-se o comportamento antigo.

Desvantagens da POO

Apesar de das inúmeras vantagens, a POO tem também algumas desvantagens. A apropriação é apresentada tanto como uma vantagem como uma desvantagem, porque a POO nem sempre soluciona os problemas elegantemente. Enquanto que a mente humana parece classificar objetos em categorias (classes) e agrupar essas classes em relacionamentos de herança, o que ela realmente faz não é tão simples. Em vez disso, objetos com características

mais ou menos similares, e não precisamente definidas, são reunidos em uma classificação. A POO requer definições precisas de classes; definições flexíveis e imprecisas não são suportadas. Na mente humana, essas classificações podem mudar com o tempo. Os critérios para classificar objetos podem mudar significativamente. A apropriação utilizada na POO torna-a muito rígida para trabalhar com situações dinâmicas e imprecisas.

Além disso, algumas vezes não é possível decompor problemas do mundo real em uma hierarquia de classes. Negócios e pessoas têm freqüentemente regras de operações sobre objetos que desafiam uma hierarquia limpa e uma decomposição orientada a objetos. O paradigma de objetos não trata bem de problemas que requerem limites nebulosos e regras dinâmicas para a classificação de objetos.

Isto leva ao próximo problema com POO: fragilidade. Desde que uma hierarquia orientada a objetos requer definições precisas, se os relacionamentos fundamentais entre as classes-chave mudam, o projeto original orientada a objetos é perdido. Torna-se necessário reanalisar os relacionamentos entre os objetos principais e reprojetar uma nova hierarquia de classes. Se existir uma falha fundamental na hierarquia de classes, o problema não é facilmente consertado.

2.3 *Linguagem Java*

2.3.1 Java [3]

Java é uma linguagem orientada a objeto recente. Ela foi idealizada no ano de 1991 e a primeira versão pública foi lançada em março de 1995. Isso tornou possível analisar os defeitos das outras linguagens anteriores e assimilar as vantagens das mesmas.Por causa da grande quantidade de programadores C e C++, todos os operadores lógicos, aritméticos foram mantidos da mesma forma em Java. Mas as declarações de baixo nível, como ponteiros, foram abandonadas por causar vários problemas de desenvolvimento. Outra causa de dificuldades no desenvolvimento em C era o gerenciamento manual da memória com os comandos `malloc`, e `free`. Em Java esse problema foi resolvido com a figura do *garbage collection* que desaloca a memória de objeto quando ele não é mais necessário.O modelo de gerenciamento de memória do Java é baseado em referências aos objetos.Quando não existe mais nenhuma referência a um objeto no gerenciador de memória, automaticamente o *garbage collection* desaloca a memória do objeto.

2.3.2 Plataforma Java [4]

No cenário atual, existem várias plataformas de desenvolvimento entre elas estão Windows, Unix e Macintosh. Um software desenvolvido para um desses sistemas operacionais (SO) exige adaptações e recompilação do código fonte para trocar para outro SO. Isso é um grave problema dentro do cenário heterogêneo que é a Internet, que essas várias plataformas formam uma rede mundial. A plataforma Java desenvolveu uma forma de resolver esse problema. Basicamente essa plataforma interpreta os códigos binários feitos para uma máquina virtual que é executada pelos sistemas operacionais citados anteriormente. Isso cria a plataforma neutra, um executável que roda em qualquer sistema operacional sem a necessidade de qualquer modificação no código. Esse novo ambiente pode ser dividido entre duas partes principais.

Máquina Virtual Java (JVM) – Uma máquina virtual que é emulada nos processadores atuais ou implementada em um hardware específico.

Interface de programação de aplicação Java (Java API)- Interface padrão para o desenvolvimento de aplicações para esta linguagem.

A Figura 2.3 mostra como essas camadas se organizam e alguns aplicativos que podem ser feitos em Java.

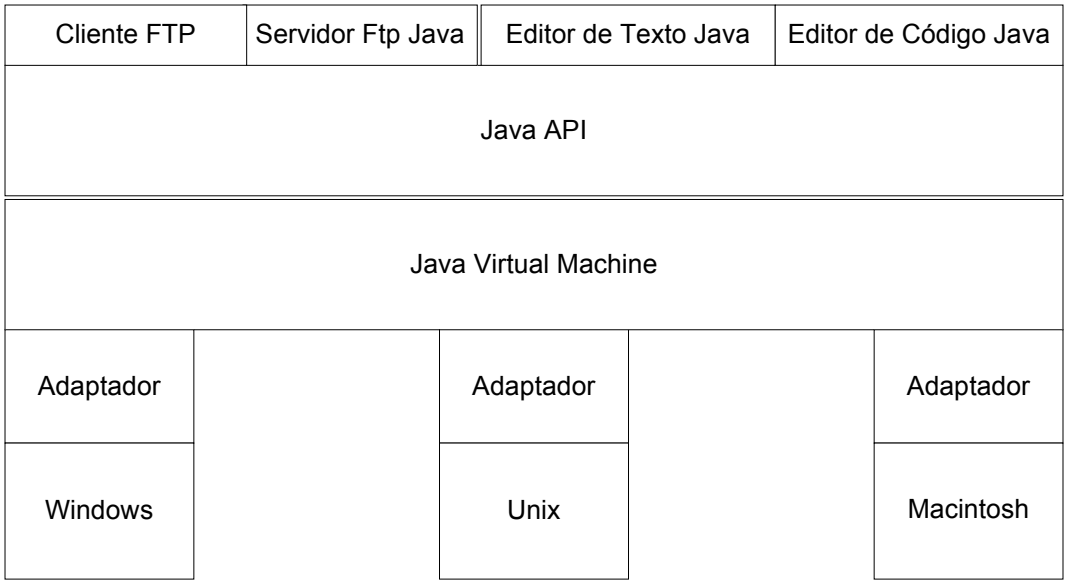

Figura 2.3 - Plataforma Java

Máquina Virtual Java

A máquina virtual Java é a chave para a independência dos binários dessa linguagem que tem a extensão class. As instruções são codificadas para os binários para uma plataforma virtual que poder ser implementada em qualquer sistema operacional. Os métodos dentro de

uma classe são indexados por nomes e isso torna possível, em tempo de execução, que um novo método possa ser inserido ou cancelado.

2.3.3 Java APIs

A interface de programação de aplicativos (API, *Application Programming Interface*) do Java apresenta inúmeras classes já implementadas para facilitar ao programador na construção de seu aplicativo, entre elas estão as JFCs. JFC corresponde à abreviatura de *Java Foundation Classes*, que abrange uma série de atributos para ajudar na construção de interfaces gráficas, as GUIs (*graphical user interfaces*). O JFC foi lançado em 1997 na conferência de desenvolvedores de *JavaOne* e contém os seguintes pacotes:

- Componentes *Swing*: Inclui todos os componentes visuais desde botões a janelas.
- Suporte a *Look and Feel*: Oferece a qualquer programa que utilize componentes Swing a escolha de sua aparência.
- API acessíveis: Permite opções de acessibilidade para leitores viva-voz e dispositivos de impressão em Braile para fornecer informações da interface do usuário.
- Java 2D (Java 2 somente): Permite a desenvolvedores a incorporarem facilmente gráficos 2D de alta qualidade, textos e imagens em aplicações.
- Suporte a *Drag and Drop* (Java 2 somente): Fornece a operação de "arrastar e soltar" em aplicações Java e entre estas e outras aplicações nativas.

Para esta monografia serão abordados os componentes *Swing* e a operação de *Drag and Drop*, que foram aplicadas na interface ProgSim. Antes, porém, serão abordados temas relativos à manipulação de eventos e tratamento de exceções, que são se importância fundamental em programas com GUI. E para finalizar a abordagem relativa à linguagem Java, apresenta-se o conceito de fluxo de dados e arquivos.

2.3.3.1 Manipulação de eventos

Qualquer sistema operacional que suporte interfaces gráficas de usuário precisa constantemente monitorar o ambiente buscando por eventos tais como teclas pressionadas ou cliques do mouse para então informar esses eventos aos programas que estão em execução. Cada programa então decide o que fazer em resposta a esses eventos. A linguagem Java adota uma metodologia em termos de recursos e, conseqüentemente, na complexidade resultante. Dentro dos limites dos eventos que o pacote AWT (*Abstract Window Toolkit* – kit de ferramentas de janelas abstratas) conhece, pode-se controlar totalmente como os eventos são

transmitidos desde as origens de evento (como botões e barras de rolagem) até os ouvintes de eventos.

Pode-se designar qualquer objeto para ser um ouvinte de evento – na prática, escolhe-se um objeto que possa efetuar convenientemente a resposta desejada ao evento. As origens dos eventos têm métodos que permitem registrar ouvintes de eventos neles. Quando um evento ocorre na origem, esta envia uma notificação do mesmo para todos os objetos ouvintes que foram registrados para esse evento. Como era de se esperar de uma linguagem orientada a objeto como a Java, a informação sobre o evento é encapsulada em um objeto *evento*. Em Java, todos os objetos *evento*, no fim, derivam da classe `Java.util.EventObject`. Segue alguns exemplos de classes ouvintes e suas respectivas ações que resultam num evento:

- *ActionListener*: Usuário clica num botão, pressiona a tecla *Return* enquanto está digitando em um campo de texto, ou seleciona algum item de menu.
- *WindowListener*: Usuário fecha uma janela principal.
- *MouseListener*: Usuário pressiona um botão de mouse enquanto o cursor está sobre um componente.
- *MouseMotionListener*: Usuário move o mouse sobre um componente.
- *ComponentListener*: Componente se torna visível.
- *FocusListener*: Componente recebe foco do teclado.
- *ListSelectionListener*: Tabela ou lista alteram seu conteúdo.

Com exceção do *ComponentListener* e do *ListSelectionListener*, todas estas classes foram utilizadas no ProgSim.

2.3.3.2 Tratamento de Exceções

Examinam-se agora os mecanismos de que a linguagem Java dispõe para lidar com dados incorretos e programas com erros. Quando o programa encontra erros durante sua execução, o ideal seria notificar o usuário do erro e permitir ao usuário salvar todo trabalho e sair do programa de forma adequada. Para tanto, a linguagem Java usa uma forma de captura de erros chamada, apropriadamente, de tratamento de exceções.

A reação tradicional a um erro num método é retornar um código de erro especial, que o método chamador possa analisar. Infelizmente, nem sempre é possível retornar um código de erro. Pode ocorrer de não haver uma maneira óbvia de distinguir entre dados válidos e inválidos. Em vez disso, a linguagem Java permite que todo método tenha uma forma de saída alternativa caso seja incapaz de finalizar sua tarefa normalmente. Nessa situação, o método

não retorna nenhum valor. Em vez disso, ele *lança* um objeto que encapsula a informação do erro. Além disso, a linguagem Java não ativa o código que chamou o método; em vez disso, o mecanismo de tratamento de exceções começa sua busca por um manipulador de exceção que possa lidar com essa condição de erro particular. Em Java, um objeto exceção é sempre instância de uma classe derivada de *Throwable*.

2.3.4 Componentes Swing

Nos subitens seguintes são apresentadas as ferramentas mais importantes que são necessárias para se elaborar interfaces de usuário gráficas com mais recursos. Antes, analisar-se-á a arquitetura que forma o Swing. Seu modelo segue o padrão de projeto *Modelo-Visualização-Controlador* (*model – view – controller*). Este padrão, como muitos outros modelos de projeto, se baseia num dos princípios do projeto orientado a objeto: não construir um objeto responsável por tarefas em demasia. Em vez disso, associa-se o estilo visual do componente com um objeto e armazene seu conteúdo em outro objeto. O padrão de objeto MVC (modelo-visualização-controlador) basicamente implementa três classes separadas:

- Model: o modelo, que armazena o conteúdo e não tem interface de usuário
- View: a visualização, que exibe o conteúdo armazenado no modelo
- Controller: o controlador que processa a entrada de dados do usuário

Uma das vantagens do padrão MVC é que o modelo pode ter várias visualizações, cada uma mostrando aspectos diferentes do conteúdo. Assim, projetistas do Swing podem implementar estilos de aparência e funcionalidade intercambiáveis.

Gerenciador de Layout

Gerenciamento de layout é o processo de determinar o tamanho e a posição dos componentes. Por definição, cada contêiner tem um *gerenciador de layout* – um objeto que execute a gerência de *layout* para os componentes dentro do contêiner.

A plataforma de Java fornece cinco gerentes geralmente usados da disposição: `BorderLayout`, `BoxLayout`, `FlowLayout`, `GridBagLayout` e `GridLayout`. Estes são projetados para visualizar múltiplos componentes. Um sexto gerenciador, *CardLayout*, é usado para combinação com outros gerenciadores.

Sempre que se utiliza o método `add()` para adicionar um componente em um contêiner, deve-se notificar o gerenciador. Para alguns gerenciadores, como `BorderLayout`, é necessário especificar a posição relativa do componente no contêiner,

usando um argumento adicional para o método `add()`. Ocasionalmente, um gerenciador tal como `GridBagLayout` requer elaborados procedimentos de configuração. Muitos gerenciadores, entretanto, simplesmente alocam os componentes baseados na ordem em que foram adicionados.

Painéis

A classe `JPanel` implementa painéis de uso geral usado para agrupar outros elementos leves. Por definição, painéis não imprimem nada na tela, exceto seu "pano de fundo". Entretanto, pode-se facilmente adicionar bordas a eles ou personalizar seu desenho.

Rótulos

Com a classe `JLabel`, pode-se apresentar textos e imagens não-editáveis. Sua aplicação, quase imprescindível em qualquer GUIs, tem como exemplo apresentar e informar ao usuários sobre componentes vizinhos.

File Chooser

A classe `JFileChooser` provê uma interface gráfica para navegar em um sistema de arquivos. Nessa interface é possível escolher um arquivo ou um diretório de uma lista ou entrando diretamente o nome do arquivo ou do diretório. Existem duas formas de utilizar essas classes: a primeira é acrescentar uma instância a um painel ou usar a JFileChooser API para mostrar uma caixa de diálogo que contem a interface gráfica. Alem disso, esse objeto se limita a selecionar o arquivo, a ação associada ao arquivo ou diretório.

Hastable

Tabela especial onde os objetos armazenados dentro dela são indexados por outros objetos. Mas os objetos usados como chaves devem ter dois pré-requisitos: eles devem implementar os métodos `hashCode()` e `equals()`. O primeiro método cria um número que é o *hash* do objeto e o `equals()` verifica se o objeto é igual a outro.

Vector

Tabela de objetos em que a quantidade de elementos é alocada dinamicamente e os elementos são acessados por uma chave numérica. Tal chave deve possuir valor igual ou

menor que o número de espaços reservados na memória para objetos que nesta tabela serão inseridos.

Árvores

Através da classe `JTree`, pode-se visualizar dados hierárquicos. Um objeto de `JTree` na verdade não contem dados; Ele simplesmente fornece uma visualização para estes dados. Como qualquer componente Swing não-trivial, a árvore acessa um modelo de dados que guarda todas as informações necessárias. Eis um exemplo de árvore:

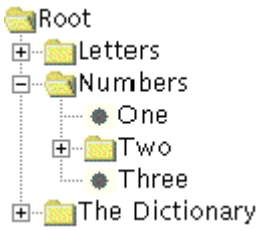

Figura 2.4 - Exemplo de uma árvore

Como a figura anterior mostra, `JTree` exibe seus dados verticalmente. Cada linha apresentada pela árvore contém exatamente um item de dados, que é chamado de *nodo*. Cada árvore tem um nodo *raiz* de onde todos os outros nodos descendem. Um nodo pode ter filhos ou não. Nodos que não possuem nodos filhos são chamados de *folhas*. Nodos podem ter qualquer número de filhos. Tipicamente, o usuário pode abrir ou fechar nodos (fazer seus filhos visíveis ou invisíveis) clicando neles.

A biblioteca *Swing* fornece um modelo de árvore padrão, o `DefaultMutableTreeModel` que implementa a interface `TreeModel`. Para criar um modelo de árvore padrão, deve-se fornecer um nodo raiz. Preenche-a com objetos de qualquer classe que implemente a interface `TreeNode`. A classe de nodo concreta que o Swing fornece, a saber, é o `DefaultMutableTreeNode`. Essa classe implementa a interface `MutableTreeNode`, uma subinterface de `TreeNode`. A figura a seguir apresenta o relacionamento das classes de árvore:

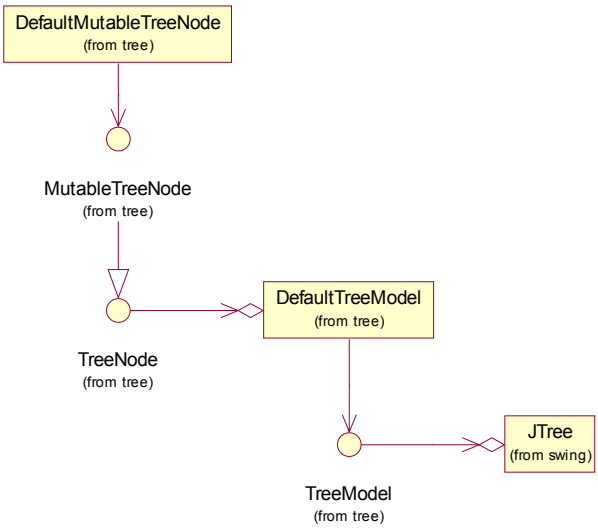

Figura 2.5 - Classes de árvore

Um nodo de árvore mutante padrão contém um objeto, o objeto usuário. A árvore representa o objeto usuário de todos os nodos. A não ser que seja especificado um exibidor, a árvore apresenta simplesmente a string que é o resultado do método `toString()`. Pode-se especificar o objeto usuário na construtora, ou pode-se configurá-lo posteriormente, com o método `setUserObject(Object userObject)`. Para estabelecer os relacionamentos progenitor/filho entre os nodos, a partir do nodo raiz, usa-se o método `add(MutableTreeNode newChild)` para inserir filhos.

2.3.5 Drag and Drop

Drag and Drop (DnD) é uma operação de gesto direto encontrado em muitos sistemas do tipo *Graphical User Interface* (GUI) que fornece um mecanismo para transferir informação entre duas entidades associadas logicamente com os elementos da apresentação na GUI. Normalmente conduzido por um gesto de um usuário, utilizando um dispositivo de entrada apropriado, DnD fornece tanto um mecanismo para permitir um retorno contínuo a respeito do resultado possível de qualquer transferência de dados feita pelo usuário durante a navegação sobre os elementos da apresentação na GUI, quanto as facilidades de fornecer qualquer negociação e transferência de dados.

Uma operação típica de DnD pode ser decomposta nos seguintes estados, não estando obrigatoriamente nesta seqüência. Os termos em itálico correspondem a objetos instanciados a partir destas classes ou interfaces implementadas nesta operação:

- Um `DragSource` é criado e associado com algum elemento da apresentação (componente) na GUI, para iniciar uma operação de DnD de algum dado em potencial do tipo `Transferable`.
- Um ou os mais `DropTargets` são criados ou chamados, associado com os elementos de apresentação na GUI (componentes), potencialmente capaz de consumir dados do tipo `Transferable`.
- Um `DragGestureRecognizer` é obtido de `DragSource` e associado a um componente a fim de localizar e identificar qualquer gesto de arraste iniciado pelo usuário sobre o componente.
- Um usuário faz um gesto de arraste sobre o componente, que o `DragGestureRecognizer` registrado detecta, e notifica seu `DragGestureListener`.

Nota: Embora seja consistentemente referido ao estímulo para uma operação de DnD como sendo um gesto praticado por um usuário, isto não impossibilita uma operação de DnD ser programada para agir automaticamente, dada uma implementação apropriada de um `DragSource`.

- O `DragGestureListener` faz com que *o* `DragSource` inicie a operação DnD em nome do usuário, talvez animando o cursor da GUI e/ou renderizando uma imagem do item que é a fonte da operação.
- Enquanto o usuário navega sobre os componentes da GUI com o `DropTarget` associado, o `DragSource` recebe notificações a fim fornecer o *feedback* de *Drag Over*, e *o* `DropTarget(s)` recebe notificações a fim fornecer o *feedback* de *Drag Under* baseados nas operações suportadas e nos tipos de dados envolvidos.

O gesto por si só move um cursor lógico através da hierarquia do GUI, cruzando a geometria de seus componentes, possivelmente tendo por resultado um cursor lógico de "arraste" quando entra, cruza, e subseqüentemente sai dos componentes e dos `DropTargets` associados. A determinação dos efeitos do *feedback*, e o sucesso ou a falha final de transferência de dados, deverá ocorrer como segue:

- Pela operação de transferência selecionada pelo usuário, e suportada por ambos `DragSource` e `DropTarget`: Copiar, mover ou criar atalho.
- Pela interseção do conjunto de tipos de dados fornecido pelo `DragSource` e pelo conjunto dos tipos de dados compreensíveis pelo `DropTarget`.

- Quando o usuário terminar a operação de arraste, normalmente tendo por resultado uma operação bem sucedida, o `DragSource` e o `DropTarget` recebem as notificações que incluem, e resultem no tipo de negociação e transferência, a informação associada com o `DragSource` através de um objeto `Transferable`.

2.4 *UML*

2.4.1 Introdução

A implementação de um aplicativo em sistemas orientados a objetos se torna mais eficiente se for feito uma modelagem, isto é, uma representação visual de uma especificação, um projeto ou um sistema por meio de um ou mais diagramas. O modelo deve apontar o essencial de alguns aspectos do que está sendo feito sem dar detalhes desnecessários. Seu objetivo é permitir às pessoas envolvidas no desenvolvimento pensar sobre e discutir problemas e soluções sem desviar-se.

Desenvolver um modelo para softwares complexos antes de iniciar a implementação ou renovação do projeto é tão essencial quanto ter uma planta para uma grande construção. Bons modelos são essenciais para comunicação entre equipes de projeto e para garantir solidez arquitetural. São construídos modelos de sistemas complexos porque é difícil compreender o projeto como sistema inteiro. Assim como a complexidade dos sistemas aumenta, assim aumenta a importância das boas técnicas de modelagem. Uma linguagem de modelagem deve conter:

Elementos do Modelo – conceitos e semântica de modelagem fundamentais;

Notação – representação visual dos elementos do modelo;

Guidelines – idioma de uso em trocas.

Em frente à crescente complexidade dos sistemas, visualização e modelagem através de uma linguagem padrão e rigorosa se torna essencial. A UML é uma resposta bem definida e amplamente aceita para esta necessidade. É a linguagem de modelagem visual com chance para construir sistemas orientados a objeto.

A UML (*Unified Modeling Language*) é a sucessora dos métodos de projeto e análise orientada a objetos mais importantes dos anos 80 e 90. Apesar de ter nascido de uma junção de métodos, ela é uma linguagem de modelagem, isto é, a notação, principalmente gráfica, utilizada para expressar projetos. Esta linguagem tem sido adotada como padrão pelo OMG

(*Object Management Group*) e a familiaridade com ele parece ter se tornado uma habilidade fundamental para engenheiros de software.

Este padrão, entre outras coisas, visa a modelagem de sistemas e não apenas o conceito de Orientação à Objeto, o estabelecimento de uma união fazendo com que alguns métodos conceituais sejam também executáveis e uma linguagem de modelagem usável tanto pelo homem quanto pela máquina.

2.4.2 Histórico [9]

Linguagens de Modelagem Orientada a Objeto começaram a surgir entre meados dos anos de 1970 e 1980 por meio de vários metodistas com diferentes abordagens para a análise e projetos orientados a objetos. O número de linguagens chegou a mais de 50 em 94, entretanto nenhuma delas satisfazia completamente as necessidades dos usuários. Dentre todas estas técnicas, três tiveram grande destaque: Booch'93, OOSE (*Object Oriented Software Engineering*) e OMT-2 (*Object Modeling Technique*). Cada um destes foi um método completo e amplamente reconhecido por suas qualidades. Em termos simples, OOSE foi uma abordagem orientada a Use-Case que provia excelente suporte a engenharia de negócios e análises de requisitos. OMT-2 foi especialmente expressiva por análise de sistemas de informação de dados. Booch'93 foi particularmente expressiva para fase de desenvolvimento e construção de projetos e popular para aplicações de engenharia. À medida que estes métodos foram evoluindo, eles começaram a incorporar as técnicas uns dos outros.

O desenvolvimento do UML começou em outubro de 94, quando Grady Booch e Jim Bumbaugh do Rational Software Corporation começaram seu trabalho na unificação dos métodos de Booch e OMT. Uma versão "rascunho" 0.8 do método unificado, como foi chamado, foi lançada em outubro de 95. No final de 1995, Ivar Jacobson e sua companhia, Objectory, se uniram à Rational e seu esforço de unificação, fundindo o OOSE a este método unificado. Estes metodistas foram motivados a criar uma nova linguagem de modelagem unificada por três razões. Primeiro, estes métodos já foram evoluídos uns em direção aos outros voluntariamente, fazendo mais sentido continuar esta evolução juntos que separados. Segundo, pela unificação da semântica e notação, podendo trazer estabilidade ao mercado de OO, permitindo que projetos se estabeleçam em uma linguagem modelo madura e permitindo aos produtores de ferramentas focar em outras características úteis. Terceiro, eles esperavam que suas colaborações rendessem melhorias em seus três métodos originais, ajudando-os a aprender lições e tratar problemas com os quais lidava de modo satisfatório.

Quando começaram a unificar os métodos, eles estabeleceram quatro objetivos para focar seus esforços. Permitir a modelagem de sistemas e não somente software usando conceitos de orientação à objetos. Estabelecer um acoplamento levando alguns métodos conceituais a serem também executáveis. Tratar a distribuição da herança em camadas em sistemas complexos. Criar uma linguagem de modelagem que possa ser utilizada tanto por humanos quanto por máquinas.

Os esforços de Booch, Rumbaugh e Jacobson resultaram na distribuição UML 0.9 e 0.91 documentadas em junho e outubro de 1996. Durante 96, tornou-se claro que algumas organizações viam UML como estratégica a seus negócios. Uma RFP (*Request for Proposal*) enviada pela OMG (*Object Management Group*) serviu de catalizador para que estas organizações se unissem em resposta à RFP e a Rational estabeleceu o consórcio parceiros UML com algumas das organizações dispostas a dedicar recursos para trabalhar visando uma definição forte do UML.

Como resultado deste esforço, o UML estabilizou-se como não-proprietário e aberto a todos, tratando das necessidades dos usuários e da comunidade científica, como estabelecido pela experiência com os métodos fundamentais nos quais se baseia. A especificação UML v.1.1 foi adicionada à lista do OMG Adopted Technologies em novembro de 1997. Desde então, o OMG tem assumido responsabilidade sobre o fornecimento e desenvolvimento do padrão UML.

2.4.3 Objetivos da Modelagem com UML

Em programas complexos, a modelagem bem estruturada constitui uma base estável para o desenvolvimento de um software otimizado. A modelagem faz uma simplificação da realidade para o sistema possibilitado assim, a visualização e o controle da arquitetura do sistema. Através da modelagem, há uma comunicação da estrutura desejada com o comportamento do sistema. Ela facilita o controle de riscos.

Usando a UML, pode-se ver o sistema como ele é ou da forma como quer que seja, possibilitando um melhor entendimento do sistema que está sendo construído. Tem-se também a especificação da estrutura e do comportamento do sistema, um “template” que pode servir de guia para a construção do sistema além de uma documentação das decisões tomadas.

A UML suporta especificações independentes de linguagem de programação ou processos de desenvolvimento, isto é, pode suportar todas as linguagens de programação razoáveis e vários métodos e processos de modelos de criação. Pode também suportar

múltiplas linguagens de programação e métodos de desenvolvimento sem grandes dificuldades.

2.4.4 Fases de Desenvolvimento de Sistemas

O desenvolvimento de sistemas em UML se inicia pela Análise dos Requisitos, onde se verificam as necessidades dos usuários expressas através de Use-Cases. Em seguida, temos a Análise, na qual se obtém as primeiras abstrações (classes e objetos), os mecanismos de domínio do problema e a colaboração entre as classes. A seguir passamos para o Projeto (Design) onde se faz a expansão da análise em soluções técnicas, criação de novas classes, interface com periféricos, Banco de Dados, etc e detalhamento da fase de programação.

Após estas fases iniciais, vem a fase da Programação, onde ocorre a conversão da fase de projetos para o código da linguagem de programação escolhida. É recomendável que se evite a mentalização do código nas três fases anteriores. Por fim, a fase de Testes, que pode ser dividida em três: o teste de unidade é geralmente realizado pelo programador e visa testar as classes e objetos gerados; o teste de integração verifica a cooperação das classes de acordo com o modelo; e o teste de aceitação que analisa a funcionalidade do sistema como o proposto nas primeiras especificações.

Em alguns casos, é desejado que se faça manutenção no sistema após sua implantação. Nestes casos, da fase de testes, volta-se ao início, refazendo todas as fases até fazer novos testes com o programa após a manutenção.

2.4.5 Conceitos e Elementos [9][10]

Alguns elementos fundamentais da UML são elementos próprios de Orientação a Objeto. Dentre estes elementos, podemos citar:

Classe: Coleção de objetos que podem ser descritos com os mesmos atributos e as mesmas operações, sua representação pode ser vista na Figura 2.6;

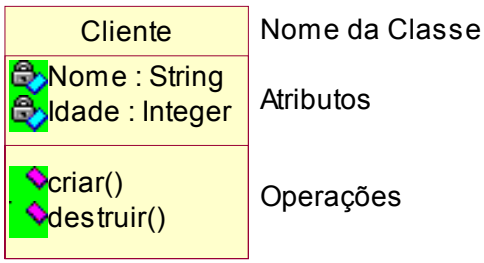

Figura 2.6 - Exemplo de Classe

Operação: Lógica contida em uma classe para designar-lhe um comportamento.

Objeto: Ocorrência específica (instância) de uma classe, similar a uma entidade/tabela no modelo relacional somente até o ponto onde representa uma coleção de dados relacionados com um tema em comum;

Mensagem: Uma solicitação entre objetos para invocar certa operação;

Encapsulamento: Combinação de atributos e operações em uma classe. É o processo de combinar tipos de dados, dados e funções relacionadas em um único bloco de organização e só permitir o acesso a eles através de métodos determinados. O encapsulamento tem as vantagens de proteger os dados e simplificar o uso de objetos;

Estado: Situação de um objeto em um dado instante de tempo;

Pacotes: Organização de elementos em grupos. A Figura 2.7 é a forma de representação gráfica de um pacote;

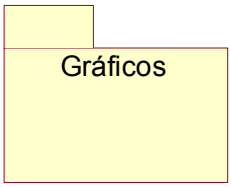

Figura 2.7 - Exemplo de Pacote

Subclasse: Característica particular de uma classe; e

Superclasse: Classe pai (possui pelo menos uma subclasse).

A Figura 2.8 representa graficamente a relação entre a superclasse e a subclasse.

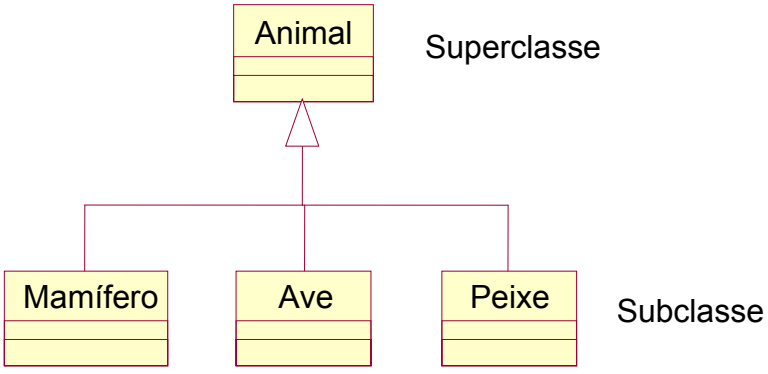

Figura 2.8 - Representação gráfica de Superclasse, Subclasse e herança

2.4.6 Relacionamentos [10]

Assim como os elementos, os relacionamentos também vêm da filosofia de OO. As relações fornecem um caminho para a comunicação entre os objetivos. É importante lembrar dos seguintes relacionamentos:

Associações: Ligam as classes/objetos entre si criando relações lógicas entre as entidas;

Associação Unária: Relacionamento de uma classe para consigo mesma. Este tipo de relação é representada da forma mostrada na Figura 2.9;

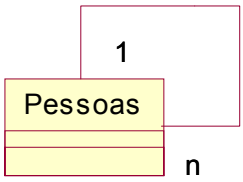

Figura 2.9 – Exemplo de Relação Unária

Associações binárias: Relacionamento de duas classes. Sua representação pode ser vista na Figura 2.10;

Figura 2.10 – Exemplo de Relação Binária

Associações n-ária: Associação entre 3 ou mais classes. Sua visualização está representada na Figura 2.11;

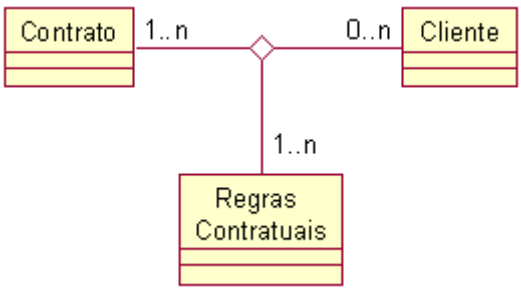

Figura 2.11 – Exemplo de Associação n-ária

Multiplicidade: define como muitos objetos participam numa relação;

Agregação: Indica um relacionamento que mostra uma relação de todo/parte;

Figura 2.12 - Exemplo de Agregação

Herança: é uma relação entre uma superclasse e suas sub-classes. Atributos comuns, operações, relações e/ou, são mostradas no nível aplicável mais alto da hierarquia. Sua representação gráfica pode ser vista na Figura 2.8.

Existem dois tipos de herança:

Generalização: atributos e operações comuns compartilhados por classes; e

Especialização: atributos e operações diferentes de uma subclasse, acrescentando ou substituindo características herdadas da classe pai.

Herança Múltipla: múltiplas superclasses para uma mesma subclasse.

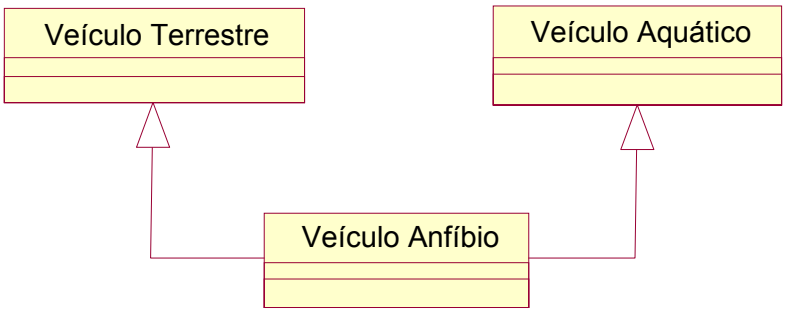

Figura 2.13 - Exemplo de Herança Múltipla

Polimorfismo: Vários comportamentos que uma mesma operação podem assumir;

Sobreposição: Sobrepões o método definido na classe pai;

Sobrecarga: Usa-se o mesmo nome e mais alguma característica para se fazer a mesma coisa.

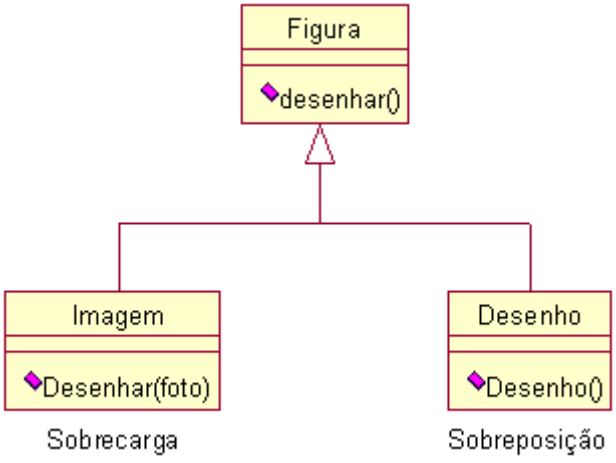

Figura 2.14 - Exemplo de Sobrecarga e Sobreposição

2.4.7 Componentes da UML

O UML é formado por quatro componentes básicos: visões, modelos de elementos, mecanismos gerais e diagramas.

2.4.7.1 Visões

As visões mostram diferentes aspectos do sistema que está sendo modelado, não sendo um gráfico, mas uma abstração com uma série de diagramas. Elas proporcionam uma ligação entre a linguagem de modelagem e os processos de desenvolvimento. Existem cinco diferentes visões: visão de Use-Case, visão de lógica, visão de componentes, visão de concorrência e visão organizacional.

A visão de Use-Case é muito utilizada na fase de análise, para verificar as funcionalidades do sistema desempenhadas por agentes externos. Ela é a base do desenvolvimento de outras visões no sistema.

A visão lógica descreve como devem ser implementadas as funcionalidades do sistema. Esta visão fornece uma descrição da estrutura estática do sistema (descrita nos diagramas de classes e objetos), isto é, seus relacionamentos, classe, objetos, etc; e demonstra as colaborações dinâmicas (expresso pelos diagramas de estado, seqüência, colaboração e atividade), de como são enviadas mensagens de uns objetos aos outros. Normalmente, esta visão é feita por analistas e desenvolvedores com a finalidade de observar o sistema internamente.

A visão de componentes consiste nos componentes de diagramas, dando uma descrição da implementação dos módulos e suas dependências.

A visão de concorrência tem por finalidade uma melhor utilização do ambiente onde o sistema se encontrará, isto é, um aproveitamento maior das execuções paralelas e gerenciamento assíncrono. Esta visão divide o sistema em processos e processadores.

A visão organizacional descreve a organização física do sistema e como os elementos do sistema se relacionam entre si. Esta visão é representada no diagrama de execução.

2.4.7.2 Modelos de elementos

Os modelos de elementos são a representação de definições comuns em orientação à objetos, tais como classes, associações, relacionamentos, etc. Estas definições estão mais bem detalhadas nos itens 2.2, 2.4.4 e 2.4.5.

2.4.7.3 Mecanismos Gerais

São extensões para processos, usuários ou organização em específico e consistem de comentários suplementares, informações ou semântica sobre os objetos do modelo.

2.4.7.4 Diagramas [10]

Diagramas são gráficos que descrevem o conteúdo de uma visão. A UML possui nove tipos de diagramas usados em combinação para representar todas as visões do sistema.

Os diagramas de Use-Case identificam como o sistema se comporta em várias situações que podem ocorrer durante sua operação, descrevem o sistema, seu ambiente e a relação entre ambos. Estes diagramas fornecem a base da comunicação entre clientes e desenvolvedores, mas não possuem nenhuma relação direta com a implementação do sistema, sendo uma forma mais ilustrativa. A **Erro! A origem da referência não foi encontrada.** ilustra um exemplo de diagrama de Use-Case.

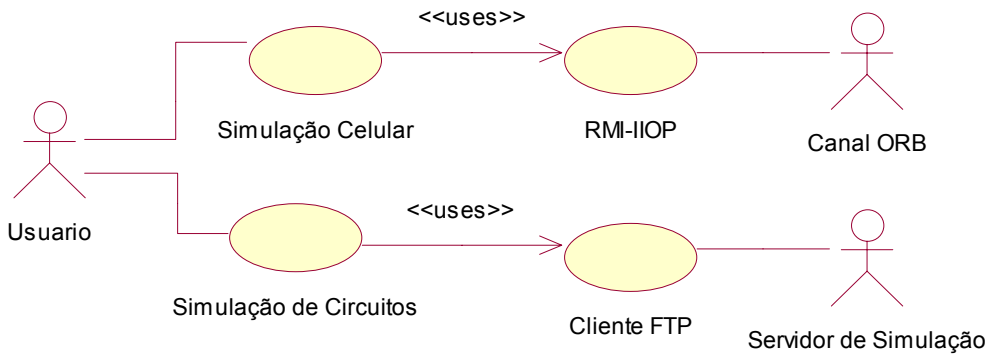

Figura 2.15 - Exemplo de Diagrama de Use-Case

Os diagramas de classe são estruturas estáticas das classes que foram modeladas demonstrando os seus relacionamentos e o tipo de relacionamento. É possível criar vários diagramas de classe em um único sistema e também é possível que uma única classe participe de vários diagramas de classe. Pode ser visualizado um exemplo de diagrama de classe na Figura 2.16.

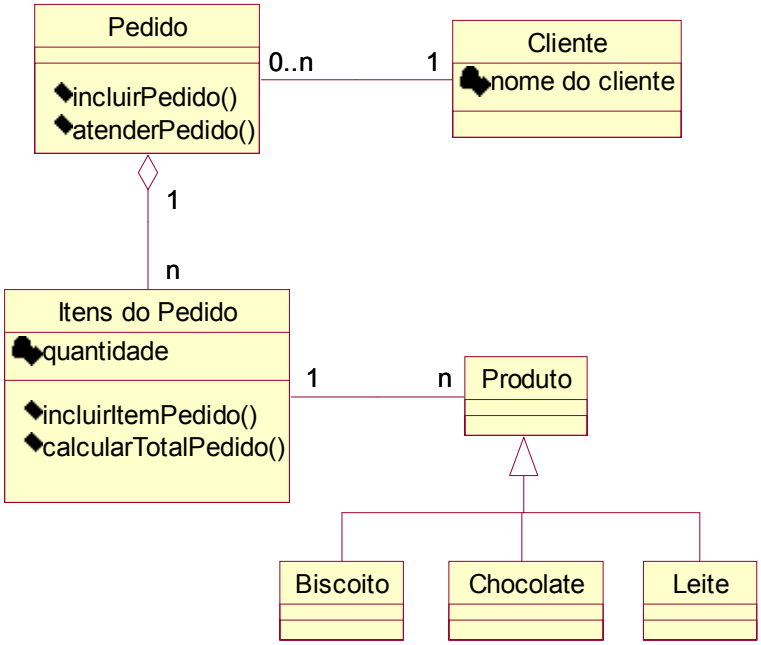

Figura 2.16 - Exemplo de Diagrama de Classe

O diagrama de objetos é uma variação do diagrama de classes e usa praticamente a mesma notação do diagrama de classes. A sua visualização se faz pela substituição das classes no diagrama de classes pelas suas instâncias. Ele mostra os objetos que foram instanciados nas classes e a colaboração dinâmica entre estes objetos. Sua principal função é a exemplificação dos diagramas de classes para sistemas complexos.

O diagrama de estados é praticamente uma descrição dos diagramas de classe que mostra todos os estados possíveis em que um objeto de uma determinada classe pode estar e os eventos possíveis de realizar mudanças. Não é necessário para todas as classes, somente para as que sofrem modificações nos diferentes estados. Ele apresenta o ciclo de vida dos objetos, sistemas e subsistemas, apresentando o ponto de início e os pontos de finalização. Um exemplo do diagrama de estados está representado na Figura 2.17.

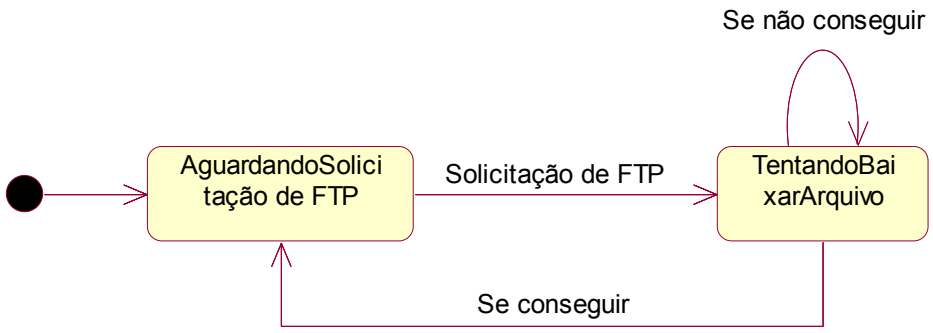

Figura 2.17 – Exemplo de Diagrama de Estados

O diagrama de seqüência mostra a colaboração dinâmica entre os vários objetos, demonstrando a seqüência de mensagens enviadas pelos objetos e os objetos que podem ser criados ou destruídos por estas mensagens. Este diagrama está representado na Figura 2.18.

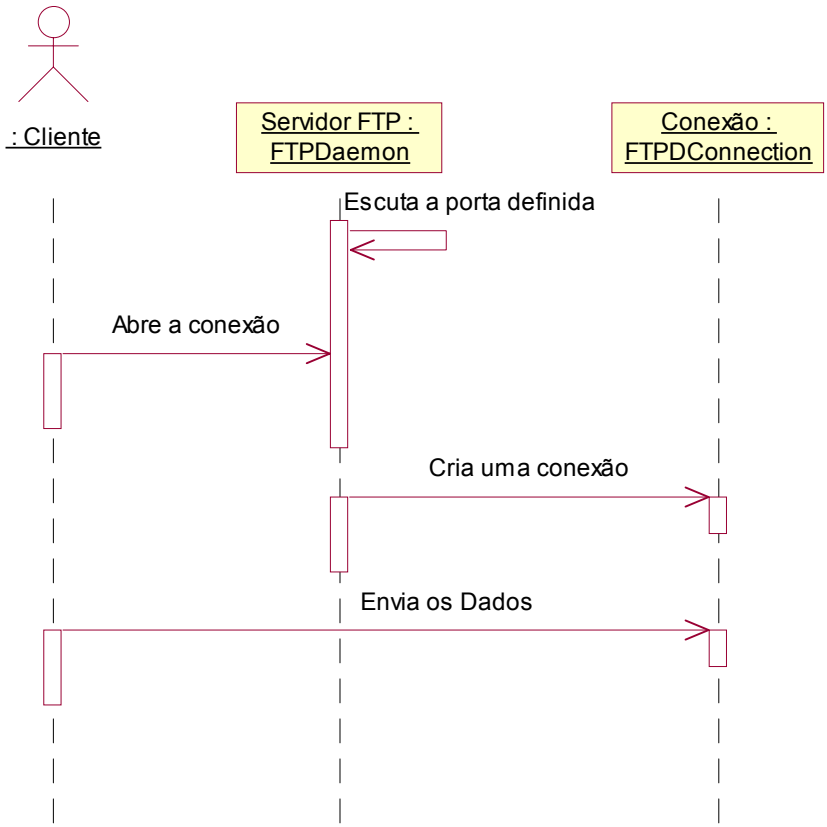

Figura 2.18 - Exemplo de Diagrama de Seqüência

O diagrama de colaboração é semelhante ao diagrama de seqüência, mostrando a colaboração entre os objetos e como são seus relacionamentos. Ele pode ser desenhado como o diagrama de objetos. A Figura 2.19 representa visualmente o diagrama de colaboração.

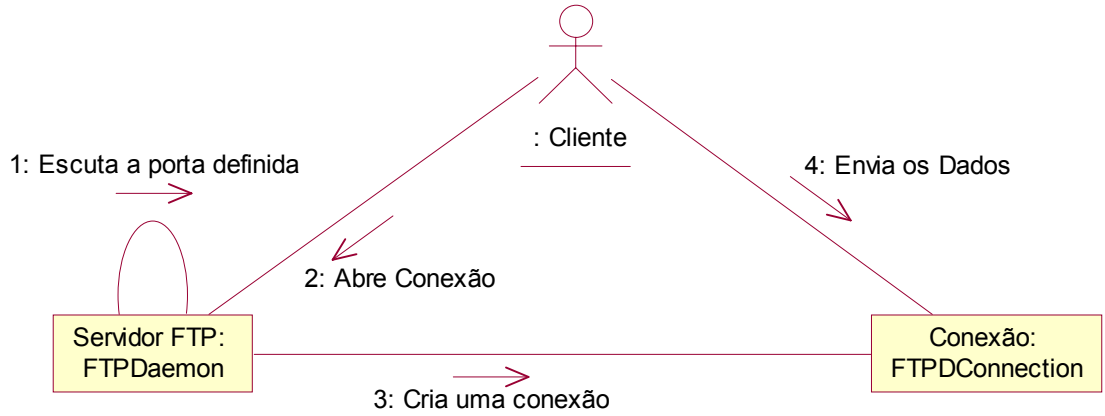

Figura 2.19 - Exemplo de Diagrama de Colaboração

O diagrama de atividade é uma variação do diagrama de estados com foco nas ações e seus resultados específicos. Ele mostra como uma instância pode ser executada em termos de ações e objetos. Este diagrama pode ser visto na Figura 2.20.

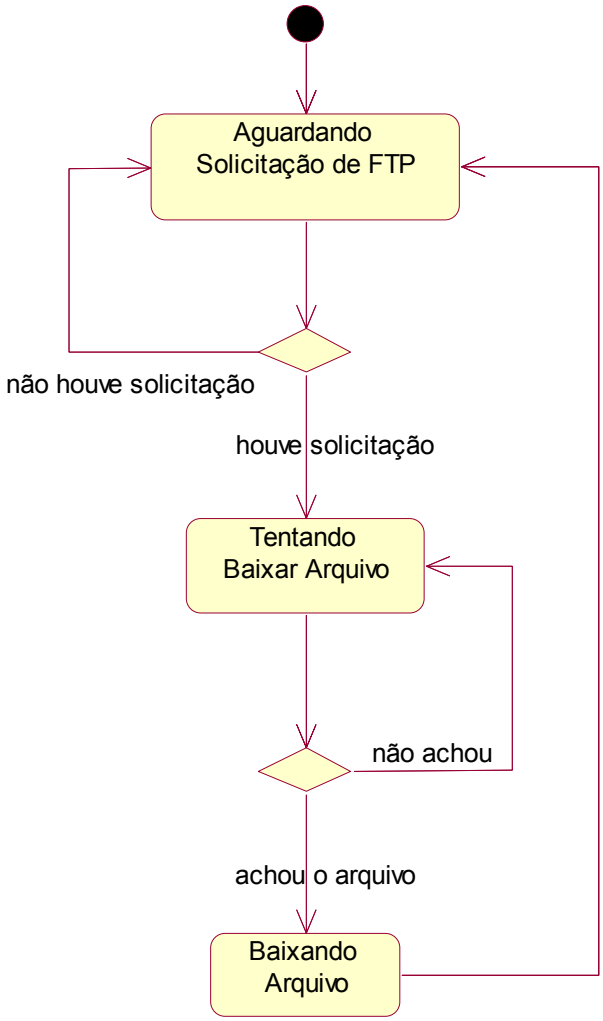

Figura 2.20 - Exemplo de Diagrama de Atividades

Diagramas de componentes mostra uma visão do sistema por um lado funcional, identificando a relação entre os seus componentes e a organização dos seus métodos em execução. Descreve os componentes de software e suas dependências entre si, mostrando a estrutura do código gerado. Pode-se ver este diagrama na Figura 2.21.

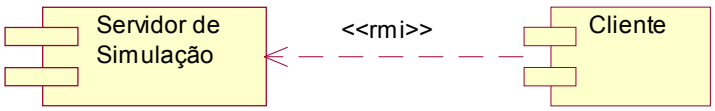

Figura 2.21 - Exemplo de Diagrama de Componentes

O diagrama de execução mostra a arquitetura física de hardware e software do sistema. Ele é composto por componentes como no diagrama de componentes. A Figura 2.22 apresenta a representação visual do diagrama de execução.

Figura 2.22 - Exemplo de Diagrama de Execução

2.4.8 Blocos de Construção

O vocabulário UML engloba três tipos de blocos de construção: as Coisas, consideradas as primeiras classes e objetos do sistema; os relacionamentos, que são a união destas coisas; e os diagramas que agrupam coisas com certa afinidade em um conjunto.

Things

As *things* são divididas em quatro tipos: as estruturadas, as comportamentais, as de agrupamento e as de anotações.

As *things* estruturadas constituem a parte estática do modelo em si, representando elementos conceituais ou físicos:

- Classes: representação dos objetos e seus atributos. Implementa uma ou mais interfaces;
- Interfaces: coleção de operações que especificam um serviço de uma classe ou componente;
- Colaboração: define a interação dos elementos do modelo e seu comportamento cooperativo;
- Use-Case: descrição de uma seqüência de ações realizadas pelo sistema;
- Classe ativa: classe cujos objetos possuem um ou mais processos e assim podem iniciar uma atividade de controle;
- Componente: parte do sistema capaz de prover a realização de um conjunto de interfaces.

As *things* de comportamento são a parte dinâmica dos modelos UML. São de dois tipos: de interação e máquina de estados. A interação é um conjunto de mensagens trocadas entre objetos com um propósito. A máquina de estados é o que um objeto passa durante sua vida, em resposta a eventos.

As *things* de agrupamentos são a parte organizacional dos modelos UML. São os pacotes, mecanismo para organização dos elementos em grupo. É puramente conceitual. Só existe em tempo de desenvolvimento.

As *things* de anotação são a parte explanatória dos modelos UML. São comentários ou notas, isto é, mecanismos de anotação em um diagrama com finalidade puramente explicativa de elementos do sistema.

Os relacionamentos se dividem em quatro tipos: dependência, associação, generalização e realização. A dependência é um relacionamento semântico entre duas coisas onde a mudança em uma afeta a outra. A associação descreve uma ligação entre objetos. A agregação é um tipo especial de associação. A generalização é um relacionamento onde podem ocorrer substituições específicas entre os objetos conectados com compartilhamento das estruturas. A realização é um relacionamento entre duas entidades onde o cliente suporta pelo menos todas as operações do fornecedor, mas sem necessidade de suportar sua estrutura de dados (atributos e associações).

2.4.9 Considerações finais a respeito de UML

A UML é uma linguagem de modelagem com regras e vocabulário focado na representação física e conceitual de um sistema. Ela facilita a comunicação e cria modelos precisos sem ambigüidades e completos. Os modelos podem ser conectados a uma série de linguagens de programação. Ela provê mecanismos de documentação da arquitetura do sistema e de todos os seus detalhes.

O sucesso de utilização da UML está relacionado com o método de desenvolvimento utilizado que descreve o que fazer, como fazer, quando fazer e porque deve ser feito.

A UML pode ser usada em sistemas de informação, sistemas financeiros, telecomunicações, serviços WEB distribuídos, etc. A UML, em sua forma atual, é esperada que seja a base para muitas ferramentas, incluindo para modelagem visual, simulação e desenvolvimento de ambientes. Como interessantes ferramentas de integração são desenvolvidas, padrões de implementação baseadas no UML vão se tornar amplamente disponibilizadas.

2.5 *Modelagem dos objetos de simulação*

Visto que o objetivo é buscar uma interface de simulação neutra de forma a possibilitar qualquer tipo de simulação, foi necessário criar toda uma filosofia de objetos a serem utilizados para viabilizar esta interface. O ponto de partida foi desenvolver uma estrutura genérica onde qualquer *componente* de qualquer sistema pudesse ser descrito. Tomou-se então, como princípio para este projeto que *todo componente será descrito como um objeto com parâmetros, comportamento e/ou arquitetura específicos que o caracterizará.*

A *arquitetura* do *componente*, se houver, definirá toda a estrutura física deste e como seus subcomponentes estão conectados e organizados. O *comportamento* do *componente* caracterizará a reação deste sob as condições em que ele se encontra, dentro de um universo de situações. Podem-se citar exemplos que definem o *comportamento* como gráficos, tabelas e até mesmo funções que quantizam a reação do *componente* para uma dada excitação e/ou configuração. Um conceito mais avançado seria a partir da *arquitetura* obter o *comportamento* e vice-versa, quando viável.

Caracterizado o *componente*, o passo seguinte é de idealizar como este será interagirá em situações que simulem a realidade. Optou-se então que as simulações serão feitas em *sistemas* onde estes terão, além de uma *arquitetura* e um *comportamento* próprio com as mesmas definições já descritas anteriormente, um *ambiente* onde serão incluídos todos os fatores externos que influenciarão o *comportamento* do *componente*. Para exemplos de *sistemas* para implementação pode-se citar sistema celular, circuitos, enlace de microondas, dentre outros. O *ambiente* que caracterizará estes sistemas poderá conter fatores como topologia e morfologia de terreno, condições de temperatura e pressão, clima, etc.

Após discussão sobre as possibilidades viáveis, foi decidido que toda e qualquer simulação deve ser realizada a partir de um *projeto*. Sendo assim, o objeto *projeto* se torna a raiz para todos os outros objetos a serem criados.

Consideramos, também, que pode ser desejado por um usuário da interface simular uma parte do *projeto* sem realizar a simulação do todo. Para esta finalidade, foi pensado um novo objeto, o *subprojeto*.

A relação entre os objetos descritos desde o início da seção está representada pela Figura 2.23.

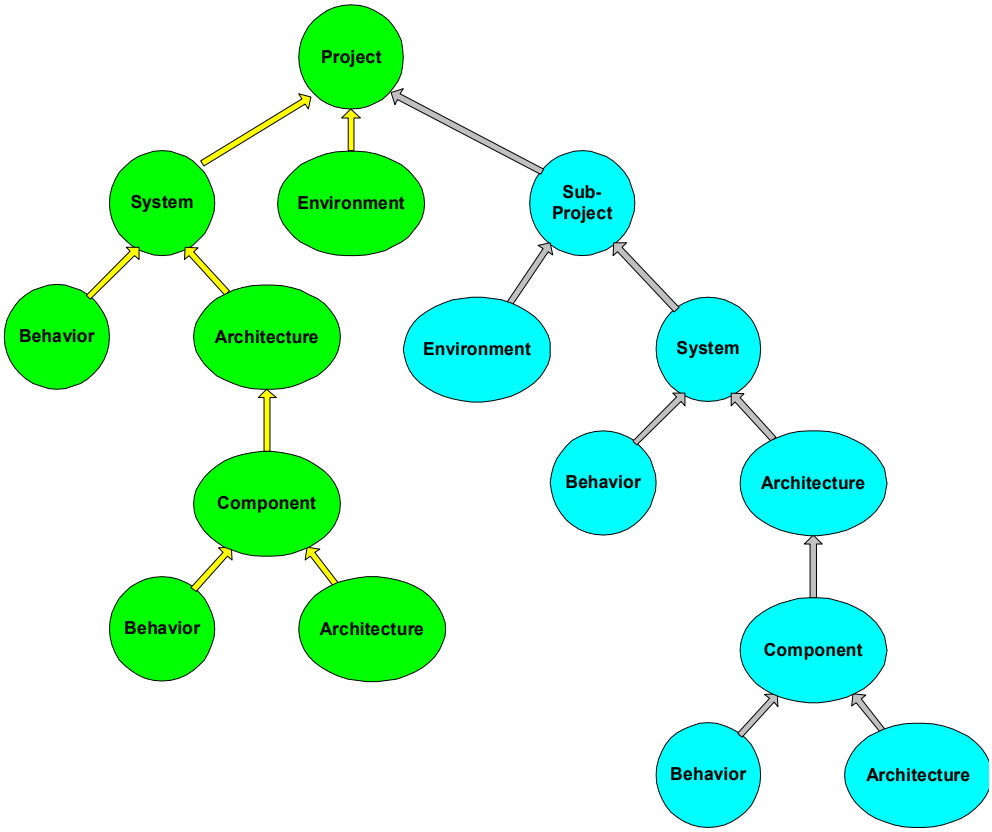

Figura 2.23 - Relação entre objetos

Definido os objetos, abre-se oportunidade para criar uma interface neutra que lidará com qualquer tipo de componente e o sistema onde ele se encontra.

2.6 *Implementação da interface ProgSim*

Levantado todo o conceito acerca deste projeto, iniciou-se a segunda parte que consiste em implementar uma interface neutra que contenha todo estes conceitos já pré-estabelecidos.

Definiu-se que a melhor forma de organizar os dados, provenientes de sistemas, componentes, arquiteturas, comportamentos, ambientes e subprojetos, seria sob um formato de árvore, onde ficariam preservadas, inclusive em sua visualização, todo o modelo hierárquico descrito na Figura 2.23 da seção anterior. Cada nodo de cada árvore provê referência aos elementos aludidos por este nodo.

Este formato de árvore, que doravante irá se chamar *árvore de projeto*, por conveniência, também foi aproveitado para a visualização e organização dos dados da biblioteca, que irá se chamar *árvore de biblioteca*. Foi designada uma biblioteca para o ProgSim visando armazenar outros projetos desenvolvidos pelo próprio usuário e também

projetos públicos de outros usuários. Para esta versão da interface do ProgSim resolveu-se separar as bibliotecas de cada sistema, forçando assim o usuário a utilizar elementos pertinentes àquele sistema no qual o projeto está envolvido. E, por fim, para completar a interface neutra, foi adicionada a esta um painel por onde será visualizado qualquer conteúdo indicado pela *árvore de projeto*, denominado como *painel principal*.

As classes principais que foram criadas e desenvolvidas para a interface ProgSim são descritas logo a seguir:

MainFrameClass: Classe principal da interface, onde são implementados todos os métodos, eventos e atributos referentes aos elementos pertencentes à interface, tais como menus, botões, rótulos, painéis, etc.

ProjectClass: Na interface ProgSim, existe o objeto instanciado da classe `ProjectClass`, único para cada sessão ou projeto em manipulação, que contém todos elementos constituintes do sistema a ser simulado. Este objeto iniciará e armazenará a *árvore de projeto*, que contém todos os objetos e eventos utilizados para a simulação. Quando um novo projeto é criado, são adicionados na árvore de projeto um, e apenas um objeto instanciado de cada classe `SystemClass` e `EnvironmentClass`, que representam o sistema e o ambiente do projeto, respectivamente.

SystemClass: Define os parâmetros do sistema a ser simulado.

ArchitectureClass: Um objeto de `ArchitectureClass` estabelece a arquitetura do elemento que contém tal objeto. Este objeto tratará da manipulação, essencialmente visual, da arquitetura dos elementos constitutivos do sistema. `ArchitectureClass` recebe atributos do ambiente descrito em `EnvironmentClass`.

BehaviorClass: Um objeto da classe `BehaviorClass` estabelece o comportamento do elemento (seja instanciado de `SystemClass` ou de `ComponentClass`) que contém tal objeto.

Todas estas classes, com exceção de `MainFrameClass` são derivadas a partir da classe `JPanel`. Foi estabelecido este critério a fim de facilitar, na implementação, a visualização em painéis do conteúdo dos objetos provenientes destas classes ao usuário.

Definiu-se também que a interface ProgSim deve apresentar uma política de segurança através de uma interface responsável pela criação, autenticação e permissão de usuário. Esta interface consiste de um conjunto de janelas, que fazem parte da interface do usuário. Estas janelas estão conectadas ao banco de dados MySQL, que contém informações referentes à todos os usuários cadastrados e seus projetos, por meio do *driver* JDBC descrito nos itens

3.3.1.2 e 3.3.1.3. A interface de autenticação se apresenta na inicialização do PROGSIM para permitir a utilização do programa apenas para usuários cadastrados. Algumas informações do usuário são armazenadas localmente, em variáveis globais do PROGSIM, para que o programa possa reutilizá-las de modo a identificar o usuário em situações tais como solicitação de simulação e criação de novos projetos de forma transparente ao usuário, isto é, para que não seja necessário que o este digite seu *login* e senha a todo instante. O programa grava suas informações a fim de serem verificadas as permissões no banco de dados sem intervenção do usuário.

Além da autenticação, existem também interfaces de criação de usuário, de modificação nos dados cadastrados e de permissões para o projeto desenvolvido. A criação do novo usuário se faz através de uma ferramenta do PROGSIM, sendo necessário que o usuário já esteja logado no sistema para utilizar esta ferramenta. As modificações dos dados se limitam a alterar informações do usuário que está acessando a interface, sendo apresentado a ele suas informações para que possa verificar quais dados estão desatualizados.

O ajuste das permissões do projeto são feitos apenas com o projeto que esteja aberto no momento. Esta interface permite que seja feita uma busca por usuário no banco de dados, e grava no banco o nome do projeto que está sendo desenvolvido, o nome do usuário selecionado e um número inteiro que representa o tipo de permissão de acesso que será dada ao usuário.

Existe ainda outra forma de gerenciar estas informações, que é a interface Web do PROGSIM, já desenvolvida anteriormente. Esta interface funciona em um Servidor Apache, e faz sua conexão com o banco MySQL por meio de JavaScript, PHP e SQL.

Após a inicialização do sistema e autenticação do usuário, é iniciado e carregado então a interface ProgSim através da classe `MainFrameClass`. A construtora desta classe contém a iniciação de todos os componentes visuais e agregados pertinentes à interface, como painéis com seus gerenciadores de *layout*, menus e botões com seus *listeners* e a performance de funções e parâmetros próprios de `MainFrameClass`. Os *listeners* referentes às árvores, funções de DnD e alguns botões serão destacados mais adiante, quando apropriado. Enfim, depois de executada a construtora de `MainFrameClass`, chega-se na tela principal da interface ProgSim. A figura a seguir apresenta o design básico da interface.

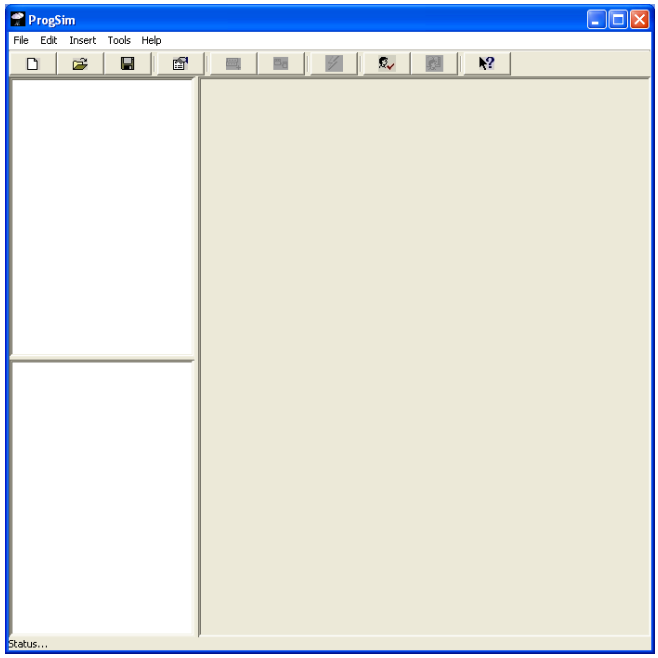

Figura 2.24 - Tela principal da interface ProgSim

Estabeleceu-se de início um design simples e harmonioso para a interface que consiste em: o primeiro quadro, localizado no setor superior esquerdo, contém a *árvore de projeto*; o segundo quadro, localizado no setor inferior esquerdo, contém a *árvore de biblioteca*; o terceiro quadro, localizado no setor central, contém o *painel principal*.

Nesta tela, apresentam-se as opções de criar um novo projeto, abrir um já existente ou alterar informações de usuário. Ao selecionar a opção de "criar um novo projeto" na interface ProgSim, seja pelo menu ou pelo botão, é chamado o método `New_actionPerformed()` em `MainframeClass`. Este método cria uma tela uma caixa de diálogo para selecionar o sistema a ser trabalhado (sistema celular, comunicação via satélite, laço de microondas, diagrama de circuito, dentre outros). A figura a seguir exibe tal tela.

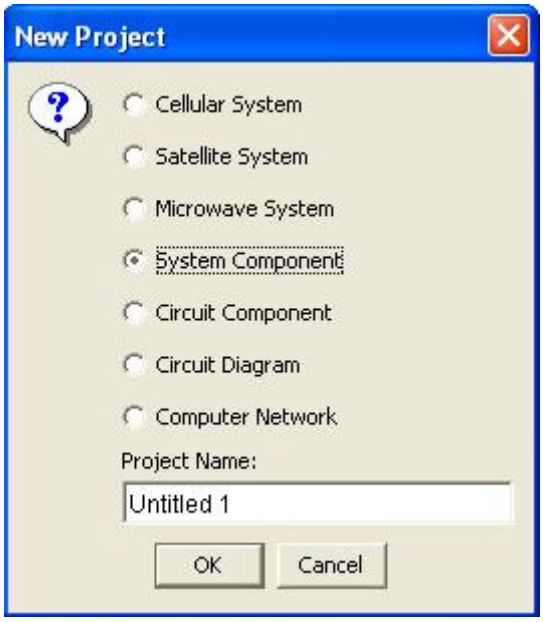

Figura 2.25 - Caixa de seleção para o tipo de projeto a ser criado

A partir da seleção do tipo de sistema, é instanciado um novo projeto a partir de `ProjectClass`, onde sua construtora é dada por `ProjectClass(String actionCommand, String name, String login, String senha)`. O primeiro parâmetro refere-se ao sistema escolhido; o segundo refere-se ao nome do projeto (que será o tipo de componente, se o sistema escolhido for do tipo circuito ou componente de sistema); o terceiro e o quarto referem-se ao *login* e à senha do usuário, respectivamente. Nesta construtora, é basicamente iniciada a árvore de projeto, instanciando para cada nodo um objeto cuja classe é referenciada pelo nodo. A seguir são descritas as classes utilizadas por cada nodo, de acordo com o sistema implementados e em seguida, a árvore de projeto já preenchida.

	Sistema Celular	Componente de Sistema	Circuito
- Raiz	`ProjectClass`	`ProjectClass`	`ProjectClass`
-System	`SystemClass`	`SystemClass`	`SystemClass`
-Architecture	`CellularSystemArchitecture`	`ArchitectureClass`	`CircuitSystemArchitecture`
-Behavior	`CellularSystemBehavior`	`BehaviorClass`	`CircuitSystemBehavior`
-Environment	`CellularEnvironmentClass`	`EnvironmentClass`	`CircuitSystemEnvironment`
-Subprojects	`JPanel`	`JPanel`	`JPanel`

Tabela 2.1 - Relação de objetos por nodo na árvore de projetos

A razão pela qual optou-se por criar novas classes derivadas a partir das classes padrões `ArchitectureClass`, `BehaviorClass` e `EnvironmentClass` para o

sistema celular e para o circuito se deve ao fato que cada um destes sistemas possuem um modo específico de visualização em sua arquitetura e também configurações próprias de comportamento e ambiente, fugindo da filosofia genérica proposta para o componente. Portanto, optou-se que para qualquer novo módulo (sistema) a ser desenvolvido para a interface, faz-se necessário derivar e implementar as classes descritas no começo deste parágrafo.

No processo de iniciação de um novo projeto, é carregada a biblioteca específica para o sistema escolhido definido na construtora da classe `LibraryClass` (esta classe contém apenas a implementação da árvore de biblioteca. Para o armazenamento de cada biblioteca optou-se escrever o objeto `libraryTreeRoot` da classe `LibraryClass` em um arquivo, o que resulta que cada biblioteca de cada sistema terá um arquivo de extensão PSL (ProgSim Library). Portanto, a construtora `LibraryClass(String systemSelected)`, a partir de seu valor de entrada, carrega o nodo raiz com o mesmo nome do sistema selecionado (ou cria um novo, caso a biblioteca esteja vazia).

A seguir será descrito como projetos de componentes são iniciados e caracterizados na interface de ProgSim. A descrição para a implementação de circuito e de sistema celular para a validação da interface é descrita nos capítulos 4 e 5, respectivamente.

Depois da seleção do tipo de projeto a ser criado, no caso específico da escolha ser um componente (de sistema ou de circuito), na construtora de `ProjectClass` é instanciado um novo objeto a partir de `ComponentClass`, que é inserido no nodo raiz. Na construtora desta classe são iniciados todos os eventos, botões, campos de texto e painéis pertinentes à criação do componente. Os atributos principais da classe `ComponentClass` são duas *hashtables*, um vetor e dois painéis:

`parametersTable`: *hashtable* que armazena os parâmetros do componente;

`relatedStringToPanel`: *hashtable* utilizada para relacionar parâmetros importados de subcomponentes a painéis que apresentam tais parâmetros, um para cada subcomponente;

`circuitNodeSet`: vetor que armazena os nós do componente;

`componentDesign`: painel que contem o formato (ícone, nós e design) do componente a ser apresentado quando este for para a biblioteca e desta para um novo projeto;

`componentParametersPanel`: painel onde são visualizados os parâmetros do componente.

Feito os procedimentos para o objeto da classe `ComponentClass`, o mesmo é feito para a classe `SystemClass`. Nesta primeira versão do ProgSim não foi necessário implementar esta classe, o que será feito quando for desenvolvido um módulo completo para um dado sistema.

Em seguida é instanciado e armazenado no nodo relativo, um objeto da classe `ArchitectureClass`. O atributo mais importante desta classe é o objeto `netList` instanciado da classe `NetListClass`, que conterá todas as ligações deste componente com outros.

O mesmo acontece com os objetos das classes `BehaviorClass` e `EnvironmentClass`. Apenas as classes de sistema derivadas de `BehaviorClass` e de `EnvironmentClass` estão implementadas a fim de descrever o comportamento e o ambiente de tais sistemas, pois não foi desenvolvido um módulo de simulação de componente apenas.

O último passo na construtora de `ProjectClass` é instanciar um painel em branco de `JPanel` para o nodo de subprojetos. Isto se faz necessário apenas para preencher este nodo com um objeto de visualização para manter a integridade do programa. Depois, quando o usuário quiser criar um novo subprojeto, este painel em branco será descartado e um novo objeto de `ProjectClass` ocupará seu lugar.

Depois de iniciar por completo o projeto, no final do método `New_actionPerformed()`, são habilitados todos os botões e menus aplicáveis àquele tipo de projeto, descrito em `MainFrameClass`. A figura a seguir apresenta a interface após a criação de um novo projeto de componente.

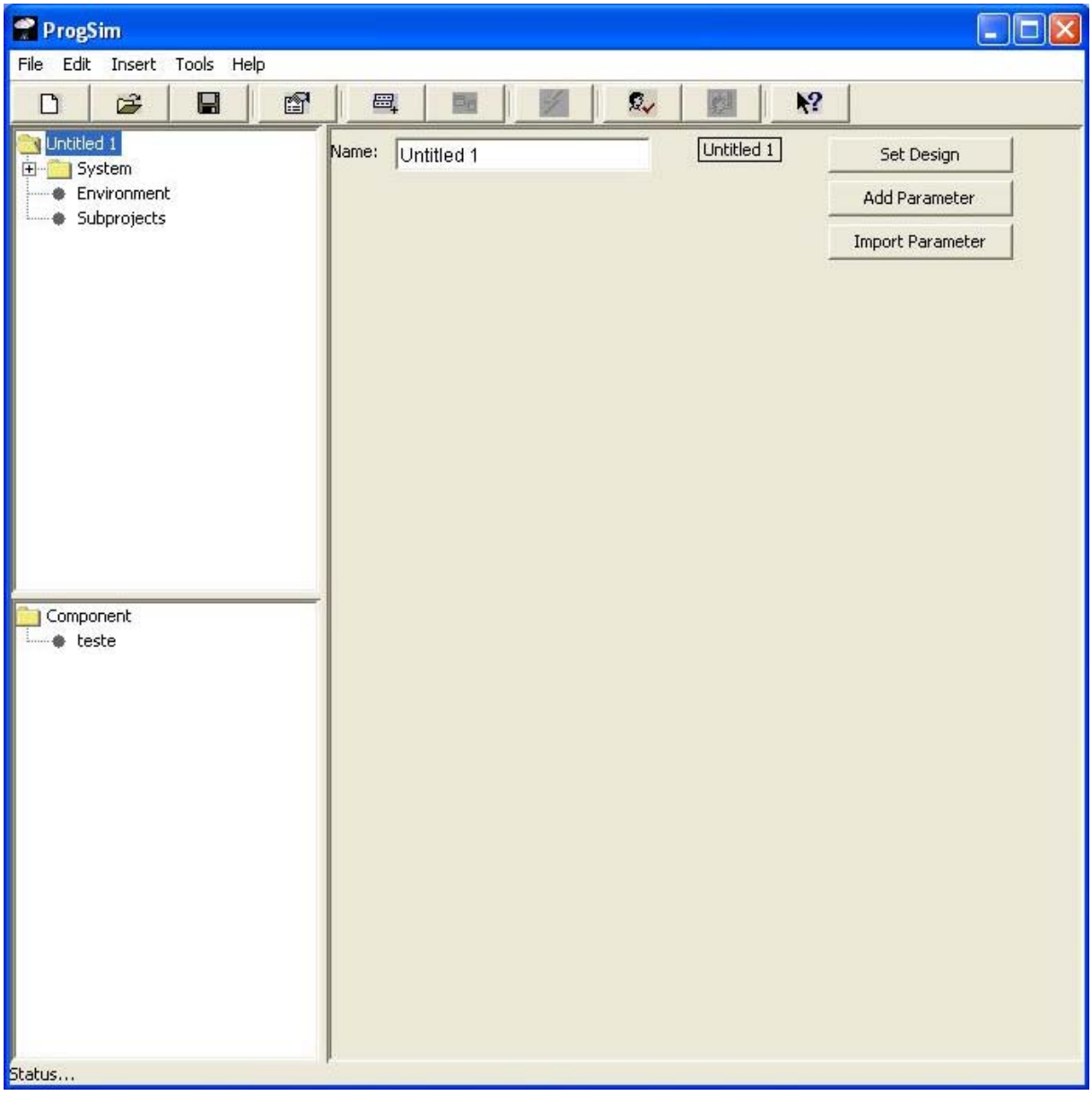

Figura 2.26 - Interface ProgSim, após criar um novo projeto de componente

O nome do projeto, que será definido como o tipo de componente ao ser inserido na biblioteca, pode ser editado através da caixa de texto apresentado no canto superior do painel principal da interface. Ao alterar o nome, automaticamente é alterado na árvore de projeto e no design do componente.

Para alterar o design, clica-se no botão *Set Design*, que abrirá uma nova caixa de diálogo, descrita na Figura 2.27. Nela são apresentadas as seguintes opções:

Incluir/excluir nós do componente (nesta versão limitada a 4 nós);

Selecionar um ícone para o componente;

Configurar para que apareça/desapareça a borda e no nome na visualização do componente.

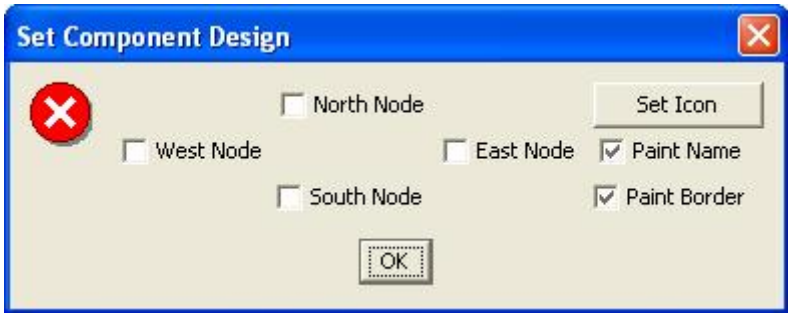

Figura 2.27 - Configuração do design do componente

Para a criação de nós, é chamado o método `createCircuitNode(int constraints)` em `ComponentClass`. O parâmetro de entrada deste método é um número inteiro constante que representa a posição do nó. Neste método é adicionado ao objeto `componentDesign` e ao vetor de nós `circuitNodeSet` um novo objeto instanciado da classe `CircuitNodeClass`. Nos objetos desta classe, que é derivada de `JLabel`, é armazenada uma "string" contendo o nome do nó e a parte visual deste. O método `removeCircuitNode(int constraints)` faz o processo inverso, removendo este nó do objeto `componentDesign` e do vetor de nós `circuitNodeSet`. Chama-se o método `setComponentIcon()` para selecionar um ícone para o componente. Este método instancia um objeto de `JFileChooser` para navegar no sistema de arquivos no intuito de localizar uma figura do tipo GIF ou JPEG a ser usado pelo componente. Por fim, na caixa de diálogo para a configuração do design do componente existe as opções de fazer visível/invisível a borda e o nome do componente.

Para adicionar novos parâmetros ao componente, utiliza-se o botão *Add Parameter*. Este botão chama o método `createParameterDialog()`, que apresenta ao usuário uma tela para preencher os dados do parâmetro. Esta tela é apresentada na figura a seguir:

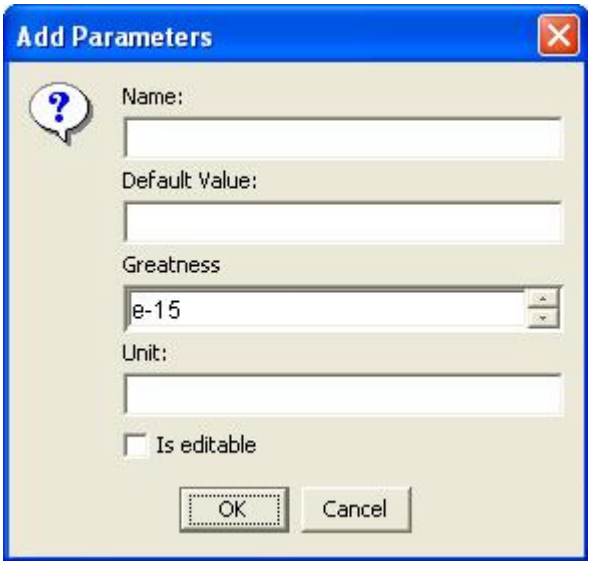

Figura 2.28 - Adicionar Parâmetro

Os dados do parâmetro consistem em: nome, valor padrão, grandeza e unidade. Nesta tela também é dada ao usuário a opção do valor deste parâmetro ser fixo ou variável. Terminada a entrada de dados, através do botão "OK", todos os valores são armazenados em `ComponentParameterClass`, classe criada apenas para tal finalidade (esta clase implementa a interface `Cloneable`). Em seguida, o método `createParameterDialog()` invoca o método `addParameter(String panelName, ComponentParameterClass parameter)`, que adiciona o parâmetro recém-criado tanto na *hashtable* parametersTable para armazenamento quanto no painel `componentParametarsPanel` para visualização.

Caso a configuração do design e especialmente dos parâmetros não seja suficiente para caracterizar o componente a ser desenvolvido, a interface ProgSim oferece a alternativa de adicionar subcomponentes já criados na arquitetura deste projeto. Tais subcomponentes se encontram na árvore de biblioteca e também já passaram pelo processo de projeto. O procedimento de adição de componentes à árvore de biblioteca será descrito posteriormente.

Determinou-se que uma forma adequada de inserir componente provindo da biblioteca seria através da técnica de *Drag and Drop* (DnD). Portanto, estabeleceu-se que a árvore de biblioteca seria a origem de arraste e o painel principal, enquanto a arquitetura do sistema estiver selecionada, consistiria no destino de arraste. Na inicialização da interface ProgSim são estabelecido os listeners de gesto de arraste `DragGestureListener` para ser reconhecido na árvore de biblioteca e o `DropTargetListener` para efetuar as operações em caso de uma operação DnD bem sucedida. Quando o nodo da arquitetura do sistema é

selecionado, então o `DropTarget`, já instanciado, é apontado para o painel que representa tal arquitetura.

Assim que a interface ProgSim detecta um gesto de arraste executado pelo usuário, esta invoca o método `componentDragGestureRecognized(DragGestureEvent event)` que se encontra em `MainFrameClass`. Este método obtém o nodo selecionado imediatamente antes da operação DnD e o encapsula em uma instância de `TreeNodeSelectionClass`. Esta classe, que implementa `Transferable`, molda o nodo definido pela classe `DefaultMutableTreeNode` a uma forma viável de transferência de dados. Finalmente chama-se o método `startDrag()` a partir de *`DragSource`* para alertar aos listeners envolvidos na operação de DnD de que um gesto de arraste foi efetuado.

A operação DnD terá sucesso apenas se o destino do arraste for o painel da arquitetura do sistema, caso contrário, toda a operação será descartada. Em caso de sucesso, os listeners envolvidos evocam o método `componentDrop(Object dataObject, DropTargetDropEvent event, DefaultMutableTreeNode node)`. Neste método é criada uma caixa de diálogo, exibido na figura 2.3.6, onde se dá ao usuário as alternativas de confirmação de adição ou mesmo um posicionamento mais preciso para o componente a ser adicionado.

Figura 2.29 - Confirmação de Coordenadas

Confirmada a inclusão do componente pelo usuário, o método `componentDrop()` então adiciona-o na árvore de projeto e no painel da arquitetura do sistema. A Figura 2.30 apresenta o resultado da operação de adição de componente provindo da biblioteca:

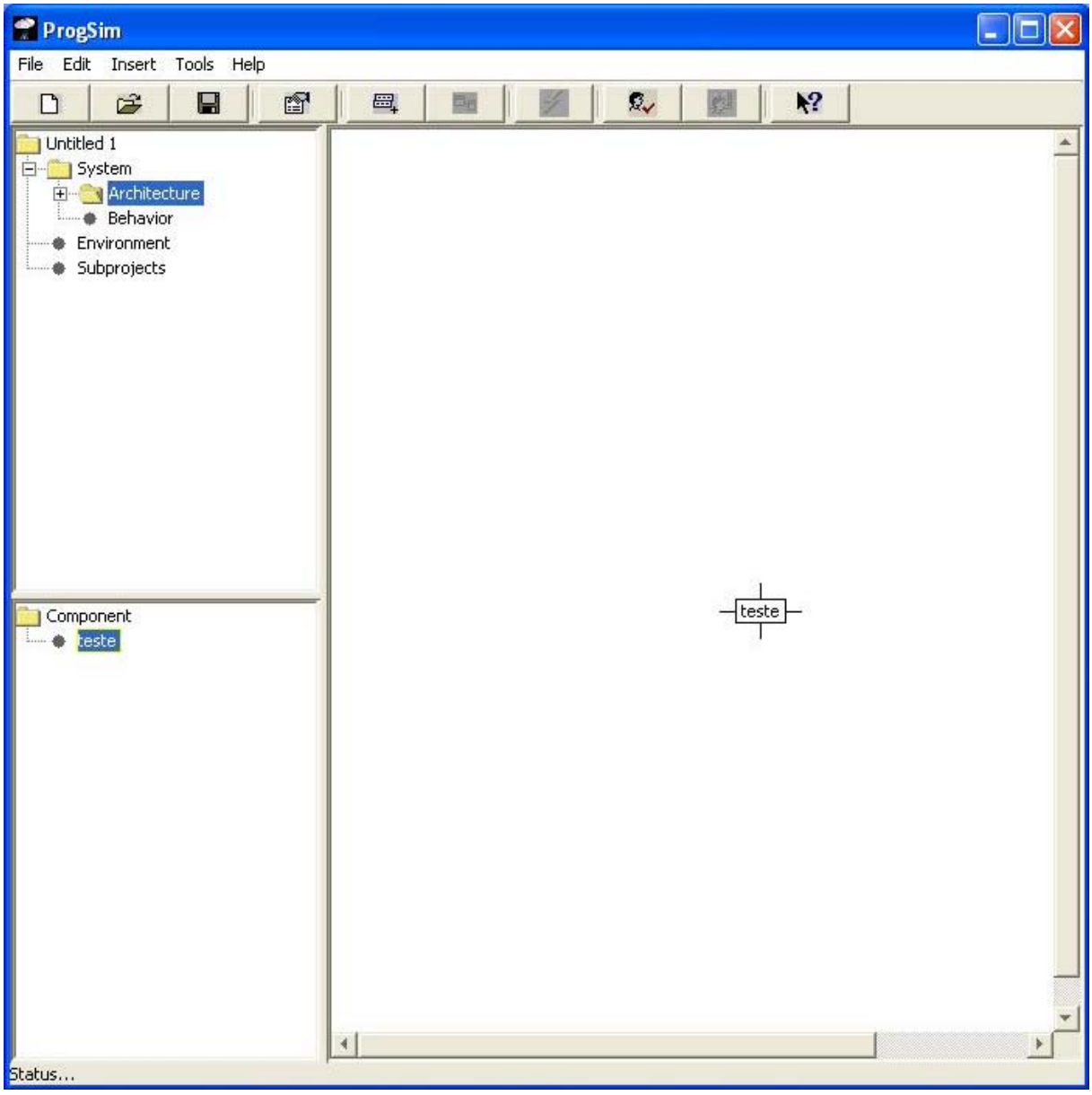

Figura 2.30 - Interface ProgSim após inserção de subcomponente

O usuário pode vir a ter a necessidade de alterar a posição de subcomponentes na arquitetura de seu projeto. Visando facilitar esta tarefa, implementou-se também uma operação de arrastar e soltar para alterar o posicionamento do subcomponente, mas sem utilizar a filosofia DnD, visto que esta operação ocorre dentro do mesmo objeto (painel). Então foram implementados listenters em uma classe aninhada à `ArchitectureClass`, a `DragListerner`. Apesar do nome desta classe aninhada, esta nada tem a ver com a operação DnD. `DragListener` capta as ações do mouse ocorridas dentro do painel de `ArchitectuireClass`, e dividem-se em três etapas expostas a seguir em ordem de execução:

Botão do mouse pressionado. Ao captar esta ação, é verificado se aonde foi clicado existe um componente passível de movimentação. Caso positivo, é armazenado tal componente, bem como sua presente localização;

Com o botão pressionado, o mouse é arrastado. Verifica-se se existe algum componente armazenado durante a fase anterior. Caso positivo, é alterada a posição do componente para o lugar apontado pelo mouse;

Botão do mouse liberado. Verifica-se se existe algum componente armazenado durante a primeira fase. Caso positivo, é alterada a posição do componente para o lugar apontado pelo mouse e finalmente é liberado o componente da memória.

Se existirem vários subcomponentes no projeto, a interface ProgSim provê a oportunidade para o usuário ligar, caso existam, os nós destes subcomponentes com outros. Para tanto, foi aproveitada a implementação de `DragListener` descrito anteriormente. Seguem-se então as três etapas para o estabelecimento de conexão entre os nós, processo similar ao arraste de um subcomponente:

Botão do mouse pressionado. Ao captar esta ação, é verificado se aonde foi clicado existe um nó de componente passível de conexão. Caso positivo, é armazenado tal nó, bem como sua presente localização;

Com o botão pressionado, o mouse é arrastado. Verifica-se se existe algum nó armazenado durante a fase anterior. Caso positivo, é desenhado um fio entre este nó e a posição apontada pelo mouse invocando o método `paintNetList(Graphics g)` contido no objeto `netList` da classe `ArchitectureClass`;

Botão do mouse liberado. Verifica-se se existe algum nó armazenado durante a primeira fase e se na posição aonde o mouse foi liberado existe algum nó passível de receber alguma conexão. Caso positivo, é invocado o método `addNodeConnection(CircuitNodeClass sourceNode, JPanel sourceComponent, CircuitNodeClass targetNode, JPanel targetComponent)` de `netList`, onde serão inseridas todas as informações relativas a esta nova conexão. E finalmente é liberado o nó da memória.

Uma outra facilidade da interface ProgSim está aplicada ao nodo *Subprojects* na árvore de projeto, porém, para esta versão, só será aplicável no manuseio de projetos para sistemas celulares. Esta facilidade consiste em selecionar um subcomponente e copiá-lo para a área *Subprojects* na árvore de projeto. A interface ProgSim instancia um novo objeto de

`ProjectClass` para este subcomponente. Assim, este terá alguns privilégios, como executar a simulação específica para tal subcomponente, como seria para um projeto padrão.

A interface ProgSim também oferece a opção de importar parâmetros de subcomponentes que já foram adicionados na arquitetura do projeto. Tal ação é disponível através do botão *Import Parameter*, já apresentado na Figura 2.26. Ao clicá-lo, este invoca o método `importParameters()`, encontrado na classe `ProjectClass`, que apresenta uma caixa de diálogo, onde são apresentados os subcomponentes com seus respectivos parâmetros. O usuário então seleciona quais parâmetros que serão acrescentados no componente em construção. Ao terminar a seleção, utiliza-se do método `clone()` com a finalidade de parâmetro a ser importado ser o mesmo parâmetro do subcomponente, ou seja, as duas variáveis apontarem para o mesmo endereço de memória.

Resta agora adicioná-lo à árvore de biblioteca. Para executar tal intento, utiliza-se o botão *`Add to Library`* localizado no menu de botões da interface ProgSim. Ao clicá-lo, abre-se uma caixa de diálogo questionando o usuário se este deseja salvar o projeto em manuseio. Se for selecionada a opção de salvar o projeto, o método `saveAsProject()` localizado na classe `ProjectClass` se encarrega de executar tal feito. Este método cria um `JFileChooser` para o usuário indicar o caminho aonde o projeto será salvo e em seguida encapsula o objeto `projectRoot` da classe `ProjectClass` em um arquivo já especificado pelo usuário. Tal objeto contém a raiz e todos os nodos, com seus respectivos objetos, da árvore de projeto, ou seja, toda a informação relativo ao projeto está contido em um único objeto, o `projectRoot`. Por conseguinte, para abrir um projeto faz-se apenas o processo inverso, lendo o objeto do arquivo e inserindo em `ProjectClass` e na árvore de projeto. Depois de salvo (ou não) o projeto, inicia-se o processo de transferência do projeto para a biblioteca.

Ao clicar no botão *Add to Library*, são criados novos botões na parte reservada para a árvore de biblioteca para que o usuário possa escolher em qual biblioteca o componente será inserido e também criar novas pastas para que se possa manter um mínimo de organização na biblioteca por parte do usuário. Selecionada a pasta e a biblioteca para onde o componente será transferido, o usuário deve então clicar novamente no botão *Add to Library*. Ao executar tal ação, é invocado o método `setToLibrary()` que é descrito na classe `ProjectClass`, que invoca o método com o mesmo nome na classe `ComponentClass`. Estes métodos transformam o projeto completo em um componente utilizável por outros projetos, executando as seguintes tarefas:

Elimina todos os botões, caixas de texto e painéis referentes à criação do componente;

Elimina o nodo de sistema, de ambiente e de subprojetos;

Adiciona-se o nodo do componente na árvore de biblioteca.

2.7 *Conclusão*

Com os conhecimentos apresentados nesse capítulo é possível entender o projeto de uma aplicação orientada a objeto, além de possibilitar o desenvolvimento de módulo de simulação que utiliza a interface neutra para uma simulação local e de possibilitar o entendimento de como integrar essa interface com um elemento remoto que exerce o papel de simulador de simulação que vai ser apresentado no capítulo 3.

3 ESTRUTURA CLIENTE-SERVIDOR

3.1 *Introdução*

Para resolver o problema do grande processamento exigido pelo programa de simulação, a equipe de desenvolvimento desse projeto definiu que a melhor solução é utilizar uma estrutura cliente servidor para distribuir o processamento. Na topologia do ProgSim o papel do cliente é desempenhado pela interface neutra com o usuário e o servidor é dividido em duas entidades, o servidor de simulação e o banco de dados. Esse capítulo é destinado à mostrar quais formas os dados podem ser trocados remotamente entre as entidades.

3.2 *Entidades da estrutura cliente-servidor*

Esse tópico comentará o papel de cada entidade na estrutura cliente-servidor. A função de cada elemento será explicada mais profundamente nos capítulos posteriores:

Cliente: A parte responsável de gerar a interface gráfica para o usuário final, formatar e enviar os projetos gerados por ele ao servidor e mostrar os resultados obtidos da simulação de uma forma amigável.

Servidor de Simulação: Esse servidor é o responsável em receber os dados dos clientes, processar e simular e enviar os resultados ao cliente. Alem de registrar as atividades de simulação e autenticar o usuário final no banco de dados.

Banco de Dados: É responsável por armazenar as informações dos usuários de seus projetos e suas simulações.

A topologia que eles se apresentam é mostrado na Figura 3.1

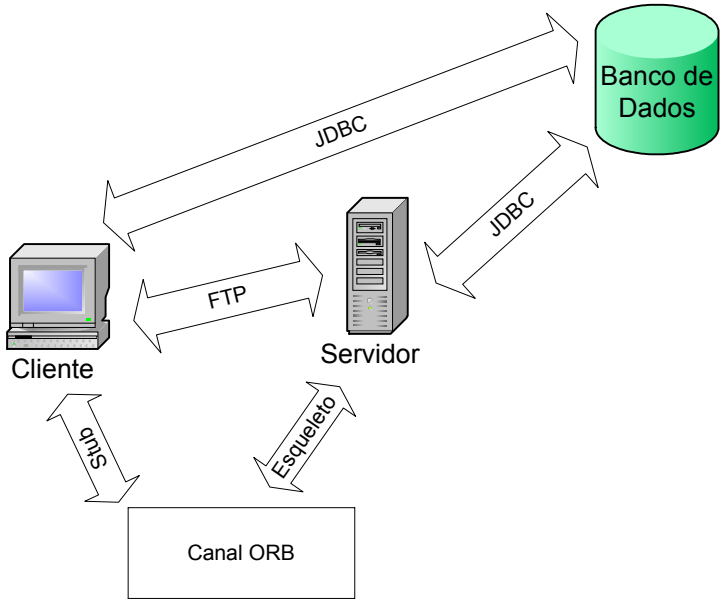

Figura 3.1 - Topologia Cliente-Servidor

3.3 *Interconexão*

Existem várias formas das entidades da estrutura cliente-servidor mostradas no item 3.2 trocarem informações. Nesse item serão apresentadas diversas formas de implementação e as vantagens e desvantagens associadas a cada uma delas. Sendo divididas em dois sub-grupos. O primeiro é a comunicação com o banco e o outro é entre o cliente e o servidor de simulação.

3.3.1 Comunicação com o Banco de dados

3.3.1.1 ODBC[11]

Para a comunicação de um programa com o banco de dados é utilizada a linguagem SQL. E. Para enviar os comandos remotamente utiliza-se o ODBC, que é uma API (interface de programação de aplicação). Isso significa que as funções contidas nesse driver abstraem a forma proprietária com que o banco se comunica com o cliente. Isso permite que uma aplicação não precise modificar seu código se o banco mudar, pois as chamadas de funções ODBC são independentes do tipo de banco. Mas, cada banco tem seu driver ODBC associado. Além disso, esse driver é dependente do sistema operacional, pois é escrito na linguagem C. As principais funções exercidas por um driver ODBC são:

Estabelecer a conexão com uma base de dados ou uma fonte de dados tabular;

Enviar comandos SQL;

Processar os resultados.

A Figura 3.2 mostra um diagrama de blocos mostrando como é organizada uma estrutura de comunicação remota como o banco.

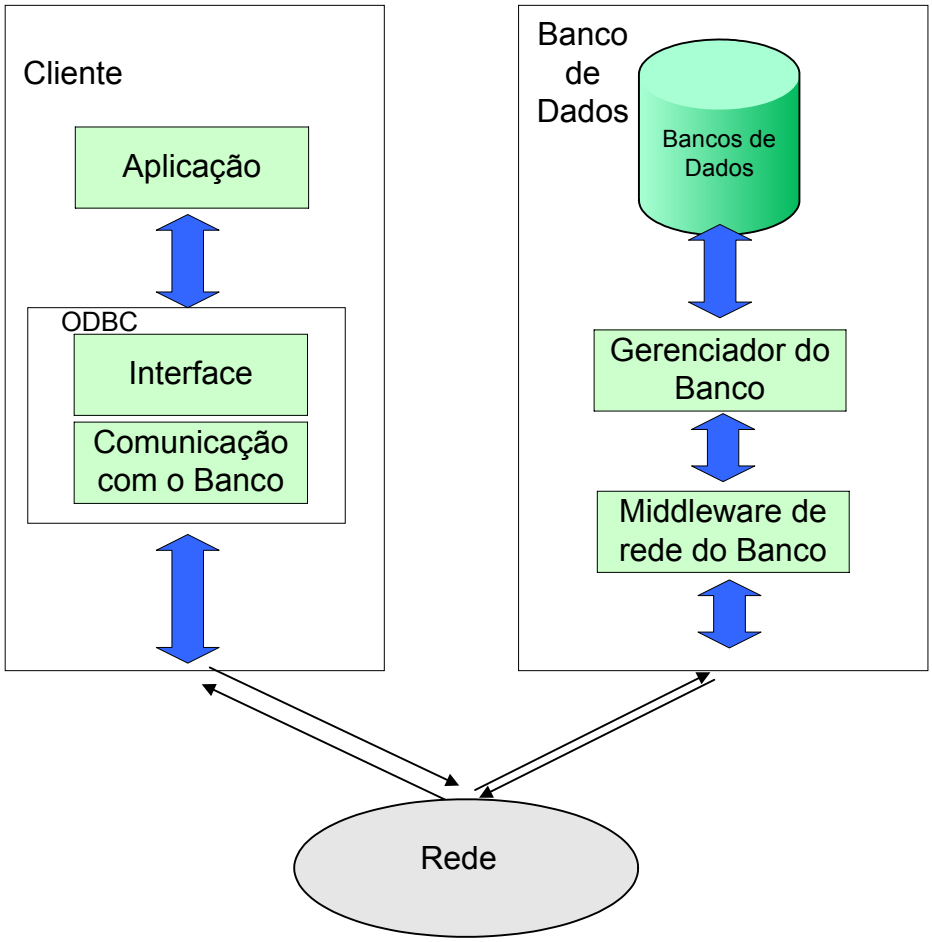

Figura 3.2 – Comunicação ODBC

3.3.1.2 JDBC[12][13]

A plataforma Java, apresentada anteriormente no item Figura 3.3 é uma plataforma neutra. Assim sendo, não se poderia utilizar a API ODBC para comunicação com os bancos de dados, pois essa implementação é dependente do sistema operacional. A solução foi criar os "drivers" JDBC que tem a mesma função. Existem quatro formas de implementar os "drivers" JDBC.

1- Ponte JDBC-ODBC

O "driver" tipo 1 foi o primeiro "driver" JBDC desenvolvido.Ele apenas atua com uma ponte para um driver nativo OBDC. Esse tipo é o menos recomendado, pois é muito lento e

não é plataforma neutra. Assim, a estação cliente tem que ter um driver ODBC configurado e funcionando para que a ponte possa usá-lo.

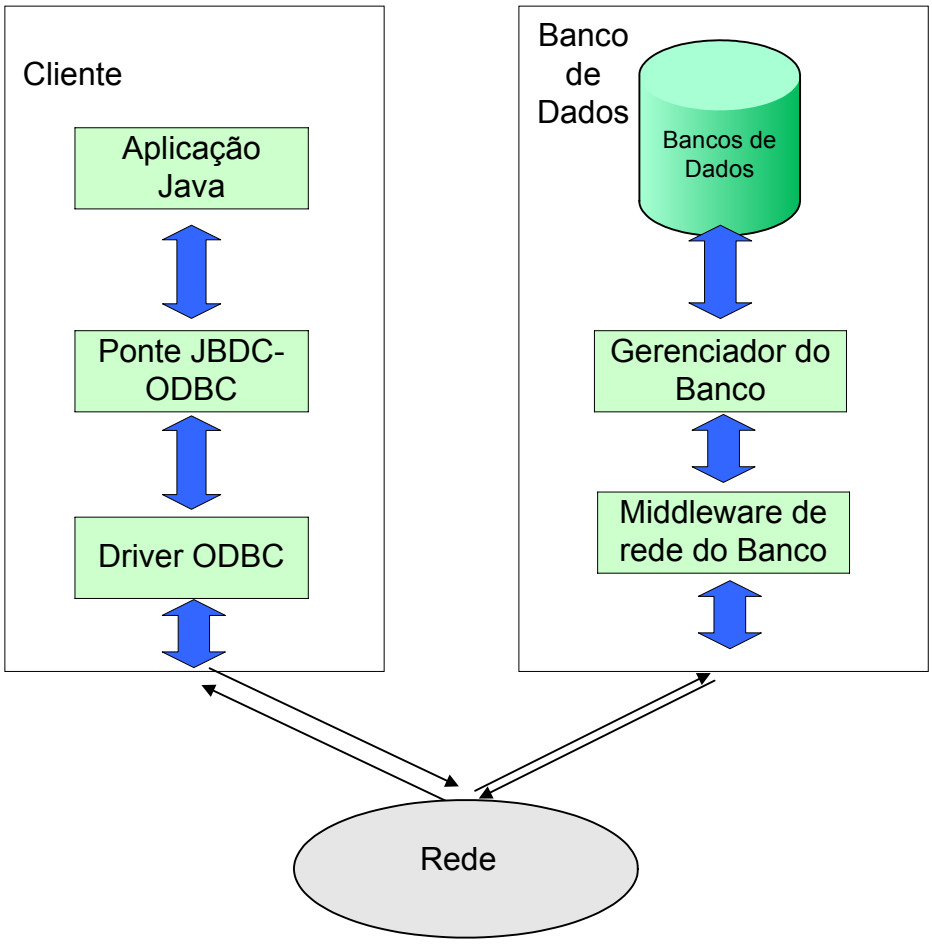

Figura 3.3 - Esquemático do driver tipo 1

2-JNMI(Invocação Java de métodos nativos)

Muito parecido com o tipo 1, pois o driver continua nativo do sistema operacional, mas as chamadas de função são escritas em java e isso dispensa o uso de um driver OBDC. Apresenta uma performance melhor que o tipo 1 mas não é plataforma neutra.

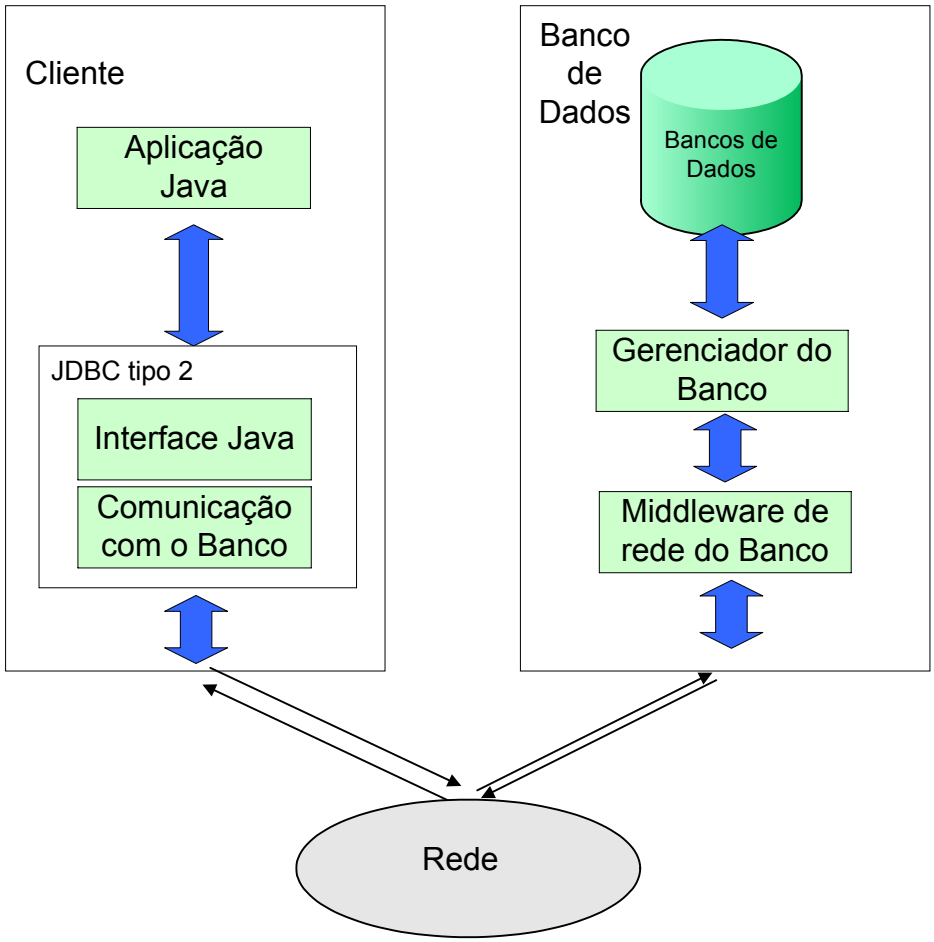

Figura 3.4 - Esquemático do driver tipo 2

3 - JDBC-NET Driver

Esta categoria consiste de um driver proprietário que acessa um *middleware* do mesmo proprietário que, através de um protocolo de rede padrão (como por exemplo HTTP) ou um protocolo proprietário, conecta-se a um servidor de acesso a banco de dados. Este servidor traduz protocolo com o cliente para um nativo do sistema de banco de dados, possivelmente utilizando ODBC. O driver cliente é plataforma neutra e tem uma performance melhor que os tipos 1 e 2, mas o servidor é dependente do sistema operacional e cria um *overhead* para acessar o banco de dados, criando mais uma camada para acessar o banco.

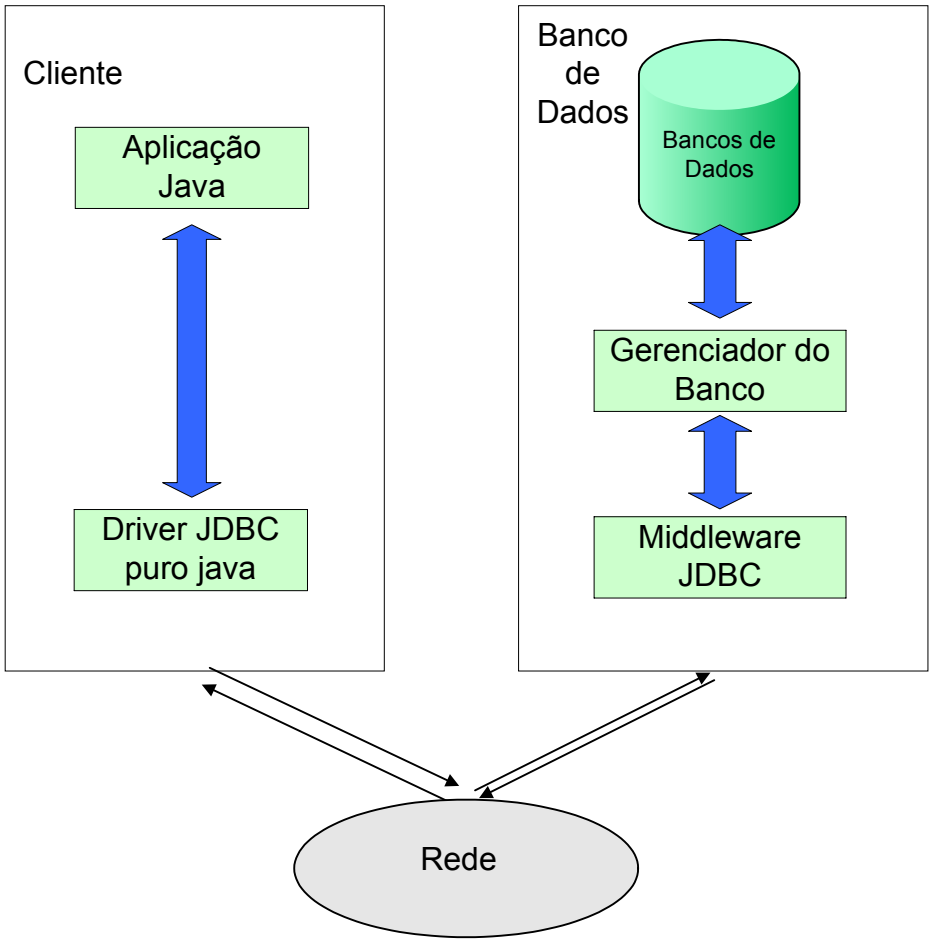

Figura 3.5 - Esquemático do driver tipo 3

4- JDBC 100% puro

Driver JBDC totalmente Java. As requisições do driver são feitas diretamente ao banco e de uma forma neutra independente do sistema operacional. A grande desvantagem desse driver é ser específico de cada banco e não poder ser otimizado para uma plataforma específica, mesmo assim apresenta uma performance melhor que os tipos 1 e 2.

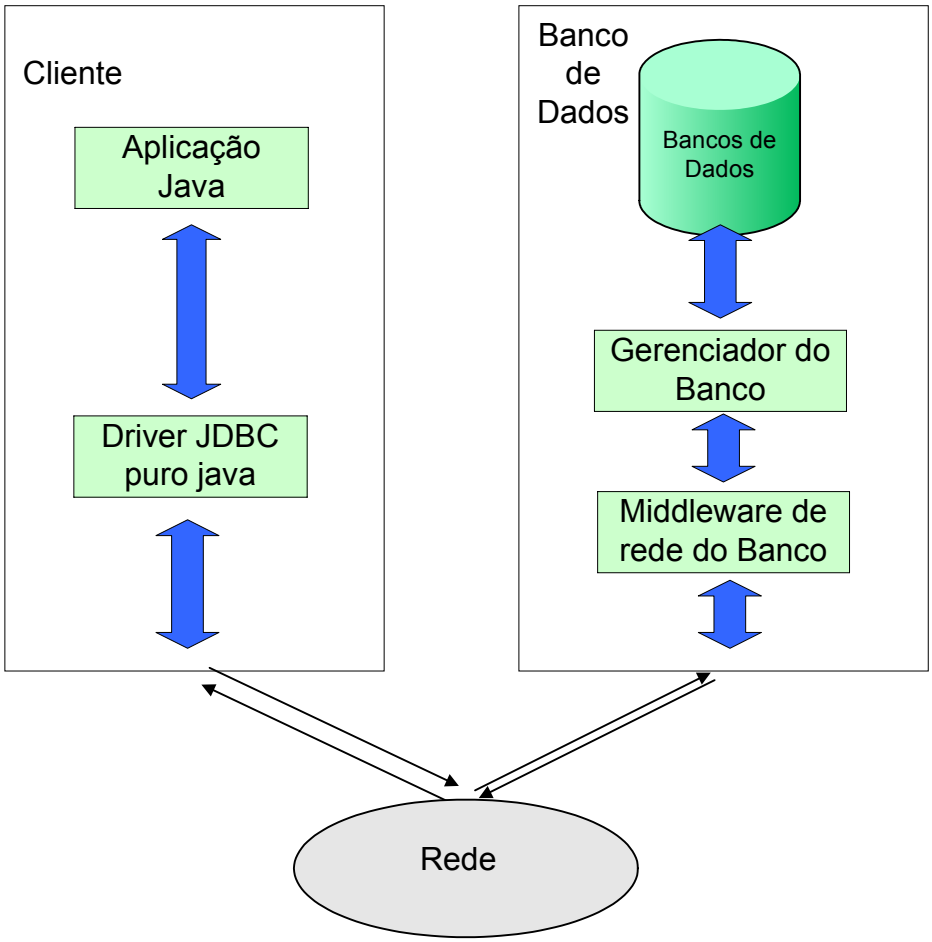

Figura 3.6 - Esquemático do driver tipo 4

3.3.1.3 O uso do JDBC no ProgSim

O banco de dados usado no ProgSim é o MySQL. O driver é um tipo 4. Essa foi a solução adotada por ser um driver código aberto, podendo ser usado livremente pela equipe de desenvolvimento do projeto. O programa cliente conecta-se ao banco de dados de uma forma direta e uma indireta.

Forma direta – usar o driver diretamente para obter dados desejados no banco.

Forma indireta – conecta-se ao servidor de simulação e o servidor que obtém os dados do banco e devolve os resultados esperados ao cliente.

3.3.1.4 JDO – Uma nova forma de acessar um banco

Apesar do servidor e do cliente serem orientados a objeto, o banco é estruturado e isso cria necessidade de representar os objetos em uma forma tabular. Ressalta-se que existem

estudos para tornar os bancos de dados orientados a objetos e substituir a forma de comunicação entre o banco e os clientes. A forma de comunicação nesse novo cenário é JDO.

3.3.2 Comunicação entre o cliente o servidor

Como foi dito anteriormente, umas das propostas do projeto ProgSim é utilizar uma estrutura cliente-servidor para diminuir o tempo de processamento das simulações. Essa proposta visa reduzir os gastos das empresas com a compra de computadores muito caros. Por tanto, a análise da melhor solução para distribuição desse processamento entre os elementos é um ponto determinante no projeto que, sem esse estudo, pode prejudicar ou até mesmo inviabilizar o projeto. Cada subitem a seguir vai mostrar uma solução possível para a implementação da estrutura cliente-servidor, as características, as vantagens e as desvantagens.

3.3.2.1 Socket

O conceito de socket de rede foi criado junto a ARPAnet no ano de 1982 e lançado com o sistema operacional 4.1BSD e 4.2BSD. O conceito de socket pode ser definido como pontos de entrada e saída de uma comunicação de dados dentro de uma rede. Quatro elementos definem um socket conectado que são o endereço local, a porta local, o endereço remoto e a porta remota. Outro conceito importante é o de porta. Cada endereço de rede pode ter vários sockets e o número da porta é usado para diferenciá-los dentro desse endereço.

Vantagens do Socket

Solução mais comum e básica para trocar informações entre cliente-servidor na Internet. Portanto uma solução bastante fundamentada e testada. A maioria dos sistemas operacionais tem a capacidade de manipular socket como o Windows e os sistemas tipo Unix. Além disso, por causa de um padrão definido pelas RFCs, o socket de um desenvolvedor deve se comunicar com qualquer socket de outro desenvolvedor.

Desvantagens do Socket

Como os objetos são complexos, é muito difícil formatá-los para que possam ser transmitidos pelo socket e isso exigiria muitas linhas de códigos. Se houvesse mudanças, exigiria muitas mudanças e testes para comprovar a integridade dos novos dados enviados pelo socket. Isso torna a evolução do produto complicada.

3.3.2.2 FTP

Protocolo definido na rfc 959, usado para transferência de arquivos entre computadores dentro da rede ip. A estrutura básica do ftp é mostrada abaixo.

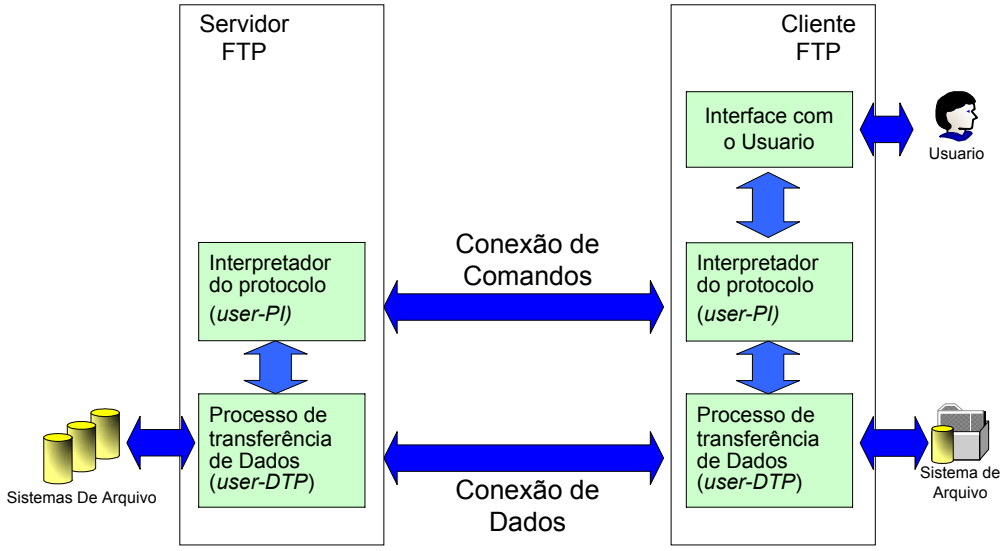

Figura 3.7 - Legenda de Blocos de uma Conexão FTP

Existem duas conexões entre o cliente e o servidor, uma para os comandos entre o cliente e o servidor e o outra para a transferência de dados entre os computadores. No protocolo de comandos é o telnet. No canal de dados, existem dois modos de trafegar os dados:

ASCII – O remetente converte os dados em uma representação no padrão 8-bit NVT-ASCII e o receptor converte desse formato para o formato interno.

BINARY – Envia os dados em formato original sem conversão

O cliente FTP é dividido em três partes:

Interface com o usuário - Forma que o usuário interage com o cliente FTP. Pode ser implementada como uma interface gráfica (GUI) ou de texto.

user-PI – Interpretador do protocolo que inicializa a conexão de controle e controla a conexão de dados do cliente

user-DTP – Processo que escuta a porta da conexão de dados com o servidor

O servidor FTP é dividido em duas partes:

Server-PI – Interpretador que espera uma nova conexão do Pi de um usuário. Estabelece a conexão de controle e controla a conexão de dados do servidor

Server-DTP – Processo de transferência de arquivo do servidor. Estabelece os parâmetros de transferência e de armazenamento dos arquivos.

Vantagens do FTP

Protocolo bastante difundido. Foram criadas várias implementações prontas para vários sistemas e linguagens. Suporta transferência de grandes quantidades de informação em formato de arquivos, além do que, por ser definido em uma rfc, o cliente ftp de um desenvolvedor deve se comunicar com qualquer servidor ftp de outro desenvolvedor.

Desvantagens do FTP

Muito limitado, pois é feito para suportar arquivos e isso obriga os dados a serem escritos em um arquivo e depois enviados para o outro lado. Isso pode gerar lentidão no processamento.

3.3.2.3 CORBA

CORBA é a sigla para Arquitetura do Mediador de Requisições de Objetos Comuns (*Common Object Request Broker Architecture*), um padrão aberto feito pela OMG para trabalhar com objetos remotos através das redes. Para transmissão dos dados é utilizado o protocolo IIOP. Esse padrão é útil para integração de sistemas heterogêneos, pois é independente da linguagem de programação, processador ou mesmo do sistema operacional. A base da interoperabilidade e neutralidade do CORBA está baseada em três conceitos. O primeiro é que as interfaces usadas pelos programas são escritas em uma linguagem neutra denominada IDL, o segundo é que a implementação é separada da interface e o terceiro conceito é que o canal para comunicar a interface com a implementação denominado de Mediador de Requisições de Objetos (ORB) é padronizado.

IDL

Essa linguagem é usada para descrever as interfaces dos objetos que são chamados pelos clientes e implementados pelos servidores. A interface se limita a definir o nome do método, a entrada e saída do mesmo. A linguagem de descrição de interface é denominada IDL Para implementar os métodos, o código deve ser mapeado para alguma linguagem de programação. Atualmente, existe norma para as linguagens: C, C++, Java, Smalltalk, COBOL, Ada, Lisp, PL/1, Python, e IDLscript. A grande vantagem é que uma interface implementada em C pode ser lida por um cliente em COBOL, pois a interface é a mesma. Genericamente, nem o servidor nem o cliente precisam saber em que linguagem está escrita o outro lado da comunicação.

Implementação separada da interface

Essa separação permite que a implementação fique em um lugar e possa ser chamada em vários lugares pelas interfaces locais. Isso é a essência da interoperabilidade do sistema. Além disso, como o cliente só tem acesso à interface, o código da implementação fica protegido.Essa divisão é implementada por duas entidades (STUB e Esqueletos):

Stub – Código que o cliente usa para instanciar os objetos remotos e fazer a comunicação com o canal ORB.

Esqueleto - Código do servidor implementa uma interface definida no stub e faz a comunicação com o canal ORB.

Canal ORB

O canal ORB é responsável em direcionar uma requisição do cliente para o objeto e encaminhar a resposta para o destino. No lado do cliente, o canal fornece as definições das interfaces. No lado do servidor, o canal é responsável por ativar os objetos requisitados e desativar os objetos que não estão sendo usado para poupar os recursos do servidor.

IIOP

O protocolo IIOP é uma implementação em TCP/IP do protocolo GIOP. Esse protocolo destina traduzir as mensagens entre os elementos da estrutura CORBA para a camada 4 e inferiores.

Vantagens do CORBA

Sendo uma solução muito flexível e com poucas linhas de código, é possível enviar os objetos entre o cliente e o servidor. Aceita muito bem as evoluções de novas funcionalidades do ProgSim, pois exige apenas a criação de uma nova interface, e toda a estrutura de interconexão é aproveitada. Facilmente os algoritmos de simulação poderiam trocar de linguagem de programação que não exigiria nenhuma mudança do código.

Desvantagens do CORBA

Por exigir que interfaces sejam escritas em IDL, isso demandaria da equipe de desenvolvedores o aprendizado dessa linguagem de descrição. Conseqüentemente, atrasaria o desenvolvimento do projeto.

3.3.2.4 RMI

Rmi (Métodos remotos Java) é uma implementação orientada a objeto em Java do mecanismo RPC (chamada remota de processo). O processo é separar em uma estrutura cliente servidor uma chamada de função e o instanciamento de objetos. De forma similar ao

CORBA, existem os arquivos stub e esqueleto. O protocolo usado para fazer essas transferências é o *Java Remote Method Protocol* (JRMP). Esse protocolo funciona somente entre cliente e servidores escritos em Java. Para aumentar a compatibilidade com outras linguagens, o protocolo para transferência foi trocado pelo IIOP do padrão CORBA. Esse novo padrão foi denominado de RMI-IIOP.

Vantagens do RMI

Implementação pura em Java, ou seja, é plataforma neutra, suporta a transferência de qualquer objeto serializável.

Desvantagens de RMI

Limita a linguagem de desenvolvimento a Java, pois a comunicação é feita entre JVM. A troca do JRMP para IIOP tenta solucionar esse problema.

3.3.2.5 RMI-IIOP

Solução desenvolvida pela Sun para adaptar a tecnologia chamada Interface Remota de Métodos (RMI) com o padrão CORBA. Isso foi possível porque o canal utilizado é o mesmo do CORBA. Sendo assim, as aplicações desenvolvidas em RMI-IIOP comunicam com outros agentes desenvolvidos no padrão CORBA.

Vantagens do RMI-IIOP

A diferença entre o RMI-IIOP e CORBA é que as interfaces são escritas em Java. Isso elimina a principal desvantagem do CORBA e mantém as vantagens apresentadas.

Desvantagens do RMI-IIOP

Para desenvolver implementações em linguagens diferentes do Java é necessário mapear as interfaces feitas em Java para IDL e depois re-mapear para a linguagem desejada como, por exemplo, C. Isso pode necessitar alguns ajustes no código para corrigir erros de mapeamento.

3.4 *Comunicação entre o modulo de circuitos e o servidor de simulação*

O protocolo FTP foi utilizado devido às necessidades do módulo externo de simulação que foi desenvolvido em MATLAB. Então, a única forma de trocar dados seria pela criação de arquivos que seriam lidos e analisados conforme serão analisados nos Capítulos 4 e 6 com mais detalhes. Então a solução natural foi o FTP. Os passos simplificados que o são feitos

para simular são mostrados abaixo. Nos próximos capítulos, esse processo vai ser descrito com mais detalhes.

- O cliente abre uma conexão ftp com o *deamon* ftp do servidor de simulação;
- O cliente escreve a *netlist* em um arquivo temporário que é enviado para o servidor;
- A rotina no MATLAB executa a simulação usando a *netlist;*
- O cliente obtém do servidor os arquivos resultantes da simulação;
- O cliente mostra o resultado ao usuário.

3.5 *Comunicação entre o modulo celular e o servidor de simulação*

O modo escolhido para transferir os dados nesse módulo foi o RMI-IIOP. Esse método é o de mais rápido desenvolvimento entre as outras soluções. Considera-se também que uma nova rotina de simulação de sistema celular feita em qualquer outra linguagem poderia substituir a implementação feita pela equipe do Projeto. Os passos simplificados para simular são mostrados abaixo. Nos próximos capítulos, esse processo vai ser descrito com mais detalhes.

- O orb é inicializado;
- O servidor de simulação registra o serviço de simulação;
- O cliente chama a função de simulação que se conecta ao orb;
- O cliente envia os dados necessários para o canal ORB;
- O servidor coleta os dados e calcula a simulação;
- O servidor envia os resultados para o canal ORB;
- O cliente coleta os resultado e mostra os resultados ao cliente.

3.6 *Conclusão*

Foi mostrado que existem várias formas de implementar a troca de dados. Para cada situação existe a solução mais eficiente; não existe uma solução geral. Os itens 3.4 e 3.5 são exemplos de duas situações que necessitam de soluções diferentes, pois a resposta de um caso não se adequa a outra. Essa análise da melhor resposta para um caso é primordial, pois a forma como os dados serão trocados na estrutura cliente-servidor define a eficiência de todo o cenário.

4 MÓDULO DE SIMULAÇÃO DE CIRCUITOS

4.1 *Introdução*

O módulo de simulação de circuitos consiste de um acréscimo de classes ao programa permitindo ao usuário utilizá-lo para montar algum circuito, e solicitar a simulação, que será realizada por um algoritmo implementado no servidor. Tem-se, neste módulo, a mesma filosofia de objetos utilizada na base do programa descrito no Capítulo 2; entretanto, foram feitas algumas adaptações para facilitar o uso desta em diagramas de circuitos.

Para a simulação, o algoritmo utilizado para validar o programa ProgSim é o que vem sendo desenvolvido no Departamento de Engenharia Elétrica da UnB com relação à simulação por harmônicos. A forma de entrada de dados deste simulador não se baseia na mesma organização de classes utilizadas no ProgSim, portanto é necessário fazer uma "tradução" da linguagem utilizada no ProgSim para ser utilizada no simulador.

Esta interface não está ligada permanentemente a este simulador, ela pode ser utilizada com outros simuladores. Para isso temos duas opções: o outro simulador utilizar a mesma linguagem de entrada de dados e a mesma forma de apresentação de resultados, ou produzir um novo módulo que faça a tradução da linguagem do ProgSim para a linguagem de entrada de dados deste novo simulador e um novo método de coleta de resultados.

A simulação não se limita apenas à tradução de uma sintaxe à outra e a execução do algoritmo, mas envolve também o envio e recepção dos dados de simulação. O processo de troca de informações se faz por meio dos mecanismos de transmissão de dados descritos no Capítulo 3.

4.2 *Interface do Usuário*

4.2.1 Ambientes de Simulação de Circuitos

A interface é composta de três formas diferentes de visualização. A escolha desta interface se faz no Ambiente do projeto, através do painel Apresentado na Figura 4.1. Esta possibilidade tem por objetivo dar a opção ao usuário de trabalhar no ambiente que considerar mais simples, ou familiar.

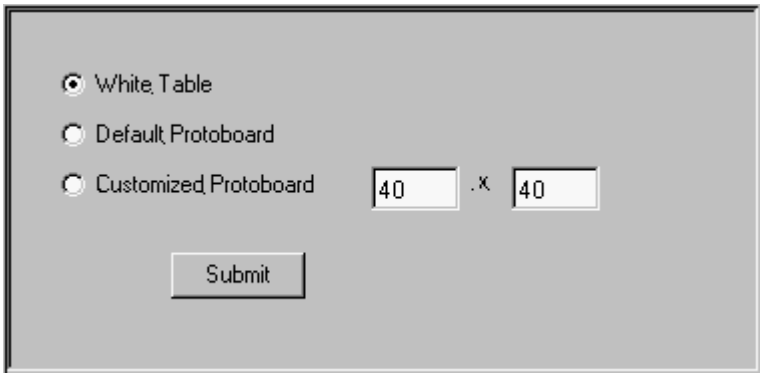

Figura 4.1 - Painel de escolha do ambiente de circuitos

O ambiente *White Table* deve ser utilizado por quem já tem familiaridade com os programas comerciais de simulação de circuito, tais como EWB ou CircuitMaker. Nele, não são utilizados valores para indicar a posição onde estão sendo inseridos os elementos. Os componentes inseridos neste ambiente são inseridos em posições arredondando os valores das coordenadas onde o usuário o está adicionando. Este arredondamento facilita a conexão dos terminais dos componentes para que o usuário consiga fechar o circuito para simulação.

O ambiente *Default Protoboard* se assemelha a um *protoboard* de laboratório, onde tem-se os pontos numa mesma coluna de cinco linhas e sem divisória, com mesmo potencial elétrico e nos pontos entre divisórias que possuem apenas duas linhas entre si, o mesmo potencial está na linha, não na coluna. Este ambiente é dedicado àquele usuário que está habituado a utilizar *protoboards*, ou que pretende analisar a disposição de um determinado circuito que deve ser montado em laboratório. Neste ambiente, a inserção dos componentes se faz pelo arraste e confirmação dos pontos onde estarão os terminais, isto é, o usuário confirma a linha e a coluna dos pontos onde devem estar os terminais do componente. A Figura 4.2 ilustra este ambiente.

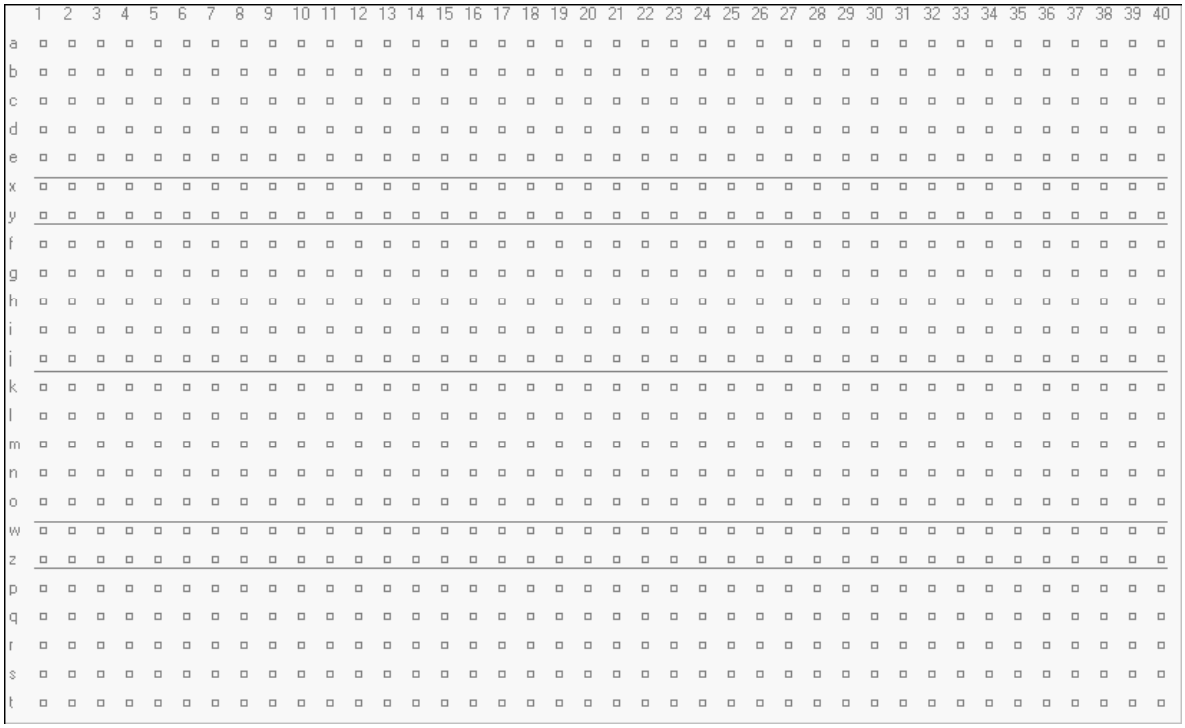

Figura 4.2 - Ambiente *Default Protoboard*

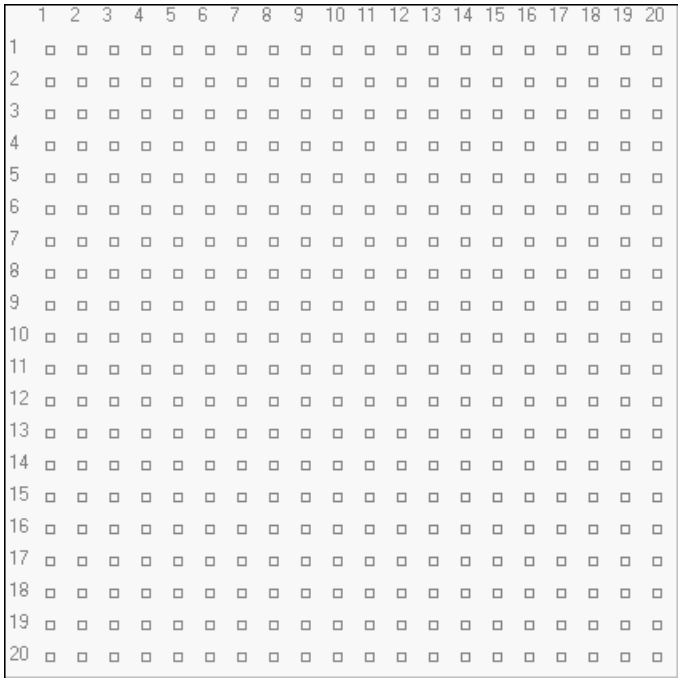

Figura 4.3 - Ambiente *Customized Protoboard*

O ambiente *Customized Protoboard* se caracteriza por seu tamanho variável. A dimensão do *protoboard* é definida pelo usuário no painel do ambiente (ver Figura 4.1). A tensão neste ambiente é específica ao ponto, não havendo simulação de um sistema de barramentos como é feito no ambiente *Default Protoboard*.

4.2.2 Criação de Componentes

Os componentes são criados praticamente da mesma forma que qualquer outro no ProgSim; sua única diferença está no fato de que, transparente ao usuário, são criados cinco parâmetros fundamentais em circuitos: o nome do elemento, a linha e coluna de um terminal e a linha e coluna do outro terminal. A diferenciação entre estes parâmetros e os outros está no fato de todo elemento de circuito obrigatoriamente deve possuir todos estes campos para poder ser inserido na arquitetura do projeto sem erro, e ser simulado com sucesso. Isto porque estes campos definem onde este componente está conectado e seu nome para identificação. Todos os outros parâmetros são inseridos separadamente pela ferramenta de acréscimo de parâmetros. Caso o componente tenha mais de dois nós, os nós adicionais devem ser inseridos separadamente como os outros parâmetros do componente.

4.2.3 Inserção de Componentes em um Projeto

A inserção de um componente se faz pelo seu arraste da janela de biblioteca para o painel da arquitetura do projeto. Feito isso, abre-se uma janela como a representada na Figura 4.4, onde o usuário deve inserir um nome ao elemento inserido e confirmar ou alterar os pontos onde devem estar os terminais deste componente, confirmando também a posição do componente. Caso haja outro parâmetro neste componente, haverá outras janelas para coletar seus dados.

Para facilitar a visualização da conexão dos componentes, quando estes são inseridos em um ambiente de simulação de circuitos, os pontos onde se conectam seus terminais são preenchidos de forma a dar destaque nestes pontos. Quando este processo ocorre no ambiente *White Table*, são criados pequenos pontos escuros nas extremidades dos componentes de forma a facilitar a visualização dos terminas do componente. As representações visuais dos componentes na interface de circuitos podem ser vistos nas Figura 4.5 e Figura 4.6.

Além dos componentes de circuito comuns, existe também um outro elemento que participa desta interface de maneira um pouco diferente. Este elemento é o fio (*Wire*) e tem por finalidade fechar o circuito ligando um ponto a outro do *protoboard* com o mesmo valor de tensão.

O usuário pode adicionar um fio no projeto pelo simples arraste do cursor do mouse no painel da arquitetura. Isto cria um fio que se estende do ponto onde foi pressionado o mouse até o ponto onde arraste foi concluído. Este processo também abre uma janela de confirmação possibilitando a alteração dos pontos onde se encontram os terminais do fio.

Após criado o fio, para que não fique atrapalhando a visualização dos pontos do *protoboard*, o fio passa por entre as linhas ou colunas do *protoboard*. Assim como os outros componente, os fios também causam o preenchimento dos pontos onde estão seus terminais, como pode ser visto na Figura 4.7.

Cada objeto inserido na janela tem tamanho fixo. Para completar sua extensão de um ponto a outro ponto qualquer, utiliza-se fios na sua extremidade esquerda de forma a conectá-lo ao outro ponto desejado, sendo que a posição do objeto fica a cargo do terminal da direita. O resultado visual desta junção está representada na Figura 4.8.

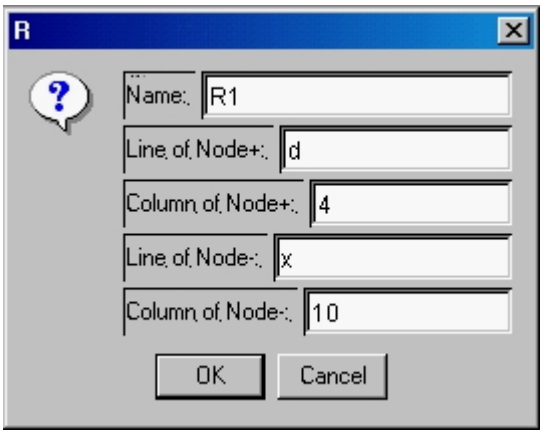

Figura 4.4 - Janela de confirmação de dados dos elementos de circuitos

Figura 4.5 - Componente em um ambiente "Protoboard"

Figura 4.6 - Componente no ambiente "White Table"

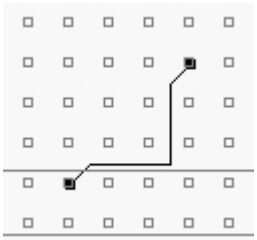

Figura 4.7 - Fio (*Wire*)

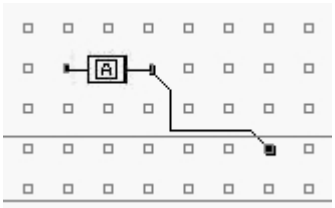

Figura 4.8 - Componente completado por fio

4.3 *Adaptações na Estrutura de Classes*

ComponentClass

Houve uma única alteração significativa feita sobre o `ComponentClass`. Consiste do acréscimo de um novo método, o `setParameters()`, que substitui o método `confirmCoordinates()`, que é responsável pela confirmação do local onde será inserido o componente. A necessidade desta mudança está no fato de que a inserção dos componentes está amarrada aos pontos do *protoboard*, além da necessidade de colher dados como nome ou valor.

Este método cria uma primeira janela estilo formulário como a representada na Figura 4.4, apresentando os campos que devem constar obrigatoriamente no componente. O campo nome já vem preenchido por um nome padrão do programa, e os outros campos, referentes aos nós em que o componente está conectado já vem com a posição em que o objeto foi solto na tela para o primeiro nó (*Node* +) e uma posição defasada em duas unidades em relação à coluna da primeira posição para o segundo nó (*Node* –).

Após a confirmação do usuário a respeito dos dados, este método verifica a *hashtable* que contém os dados dos componentes para verificar se existem outros parâmetros que não sejam estes já citados acima. Enquanto houver dados não apresentados ao usuário, o método cria janelas com o nome do parâmetro, seu valor, grandeza e unidade. Após a apresentação de todos os parâmetros, o `setParameter()` fixa o elemento na posição confirmada na primeira janela.

Para o caso da utilização do ambiente *White Table*, o método simplifica a primeira janela à confirmação apenas do nome do componente, visto que o usuário não tem nenhuma referência de onde o componente está sendo inserido. Neste caso, o objeto é fixado no ponto referente ao local onde o mouse foi solto.

CircuitEnvironmentClass

A `CircuitEnvironmentClass` contém um painel que é apresentado para o usuário poder escolher seu ambiente de simulação de circuitos, e três variáveis, sendo todas *strings*. Uma guarda qual o tipo de ambiente – *White Table*, *Default Protoboard* ou *Customized Protoboard* – as outras guardam a quantidade de linhas e colunas que são desejadas ao sistema caso seja escolhido o *Customized Protoboard*.

CircuitSystemArchitecture

Nesta classe, foi implementado um *listener* que marca onde o mouse foi pressionado e onde ele foi solto, e utiliza estes dados para criar um novo fio. É também nesta classe que está

a função que desenha o *protoboard*, se houver. Nesta classe também está armazenado o `circuitList`, que é a classe onde os componentes são inseridos para facilitar a sua busca na ocasião de simulação.

CircuitSystemBehavior

A classe responsável pelo comportamento de circuitos consiste de um painel onde são colocadas as figuras resultantes das simulações. Como pode haver mais de uma figura, foi criado um vetor de *labels*, e, quando a simulação ocorre, as figuras são passadas a esta classe como uma *hashtable*. Para cada elemento desta *hashtable*, sua figura é colocada no *label*, e acrescentada no Painel, formando assim o painel do comportamento do circuito simulado.

WireClass

A classe responsável pelos fios é composta basicamente pelos pontos que indicam os nós de onde ele sai e onde ele chega, e do método que cria a janela onde podem ser confirmados os pontos nos quais estão sendo conectados.

CircuitList

Contém dois vetores extremamente importantes para circuitos, pois um destes contém todos os componentes do projeto, e o outro contém todos os fios utilizados. Além disso, é responsável por desenhar os componentes de circuito e fios na tela, preencher os pontos do *protoboard* onde estão conectados os componentes e fios ou desenhar os nós para o caso de o usuário estar usando o ambiente *White Table* e fazer o acréscimo de um fio ao componente, caso o segundo nó não esteja na posição sugerida.

O desenho dos fios se faz em quatro passos. O primeiro faz uma diagonal saindo do primeiro nó em direção ao segundo, mas parando exatamente no meio entre os quatro pontos mais próximos. A partir daí, ele segue por entre as linhas até logo antes da coluna onde deve se conectar. O terceiro passo é a criação da linha que vai deste último ponto até o meio entre os quatro pontos mais próximos de seu destino, e faz seu último passo pela diagonal que liga este ponto ao ponto de destino.

A ordem dos passos dois e três pode vir trocada, mas o algoritmo de desenho do fio segue esta lógica de funcionamento. Este método tem se mostrado eficiente em mostrar os fios de forma que não impeça a visualização dos nós do *protoboard*.

RunCircuitSimulation e CircuitSim

Estas duas classes são responsáveis pela simulação. A primeira classe realiza a tradução dos objetos do ProgSim para a linguagem do simulador, e a chamada das funções da `CircuitSim` para enviar os dados e receber os resultados da simulação. Esta classe também

realiza o envio dos resultados para o `CircuitSystemBehavior`. A outra classe é responsável pela conexão com o servidor, onde estará sendo feita a simulação. Estas duas classes estarão sendo discutidas com maiores detalhes no tópico 4.4, que trata diretamente da simulação.

MainFrameClass, ProjectClass e demais classes.

A classe `ProjectClass` contém a função responsável pela criação dos componentes de circuito, sendo esta função uma pequena adaptação da responsável pela criação dos outros componentes do ProgSim. Além desta função, existem pequenas adaptações naturais visto que é responsável pelo gerenciamento de todo o projeto que o usuário estiver realizando. Estas alterações têm por finalidade escolher uma ou outra função, dependendo do tipo de sistema que está sendo criado.

Na classe responsável pela interface principal, a única adaptação feita foi a verificação do ambiente antes de finalizar o arraste de um objeto para determinar se deve executar a função `confirmCoordinates()` ou `setParameters()` da classe `ComponetClass`. Todas as outras classes do programa não tiveram nenhuma modificação para adaptação à interface de simulação de circuitos.

4.4 *Simulação*

A simulação é composta por duas fases: a tradução dos objetos em linguagem do simulador e envio e recebimento de dados de simulação.

4.4.1 Tradução dos Objetos para Linguagem do Simulador

A tradução dos objetos para a linguagem de simulação se faz pelo método `RunSimulation()` da classe `RunCircuitSimulation`. Seu algoritmo se faz por um laço que, para cada componente dentro do `circuitList` da arquitetura, escreve uma linha contendo o tipo de objeto, o nome do objeto, o primeiro nó, o segundo nó, e todos os outros parâmetros que existirem neste componente.

O tipo de objeto é obtido por meio do atributo `componentName` da classe `ComponentClass`. Quando o tipo obtido é A ou V, isto indica que seja um Amperímetro ou um Voltímetro e seu nome, que é obtido em seguida é acrescentado a um vetor. O nome é obtido do parâmetro `name`, próprio de componentes de circuitos. Os nós seguem um algoritmo um pouco mais complexo, pois cada nó para a linguagem do simulador é

representado apenas por um inteiro, enquanto na interface é representado por duas variáveis do tipo *string*.

Foi criado um vetor bidimensional para armazenar os valores das colunas e linhas do nó. A cada componente verifica-se o primeiro nó, e faz uma busca dos valores do nó no vetor. Esta busca, para o *Default Protoboard* considera elementos na mesma coluna dentro de um bloco com mesmo potencial, exceto nas linhas `x`, y, w e z que tem mesmo potencial nas linhas e não nas colunas. Para os outros ambientes, o potencial só é o mesmo dentro do mesmo ponto.

Após a realização da busca, caso tenha sido encontrado o nó, ou um de mesmo potencial, dentro do vetor, utiliza-se o índice que endereça este nó dentro deste vetor como o inteiro que representa o nó para o simulador. Caso não tenha sido encontrado este nó dentro do vetor, é feita uma verificação no vetor de fios do `circuitList`. Caso haja algum fio conectado neste ponto, faz-se uma verificação da outra ponta do fio para saber se este nó está presente no vetor de nós. Caso esteja, utiliza-se o índice para representar o nó. Se não for encontrado o nó no vetor, é criado um novo registro neste vetor, com a linha e coluna do nó não encontrado. Utiliza-se o índice deste novo registro para identificar o nó para o simulador.

Após a identificação do primeiro nó, repetem-se os passos referente à esta identificação com o segundo nó. Em seguida, procuram-se outros parâmetros no *hashtable* que contém os parâmetros do componente. Enquanto forem encontrados novos parâmetros no componente, acrescenta-se seus valores na linha.

Após a criação desta *netlist*, a função `Simula()` da classe `CircuitSim` é chamada, passando esta *netlist* e o vetor que contém os nomes dos voltímetros e amperímetros, juntamente com os *logins* e *passwords* do usuário que está utilizando o simulador. A importância dos voltímetros e amperímetros se faz pelo fato deles serem os objetos que indicam onde devem ser medidos os resultados da simulação. O retorno desta função é uma *hashtable* que contém o local onde foram armazenados localmente os resultados da simulação.

Após a simulação, envia-se esta *hashtable* para a classe `CircuitSystemBehavior` por meio da função `loadBehavior()` desta mesma classe. Esta mesma função já acrescenta as figuras no painel do comportamento, sendo possível verificar o resultado da simulação através da seleção o item *Behavior* na árvore de projetos na parte superior da lateral esquerda. O resultado se faz através de uma ou mais figuras semelhantes à da Figura 4.11.

Foi implementado o circuito da Figura 4.9, isto é, um circuito RC em série com uma fonte de tensão e um voltímetro conectado aos terminais do resistor. O resultado visual da montagem do circuito em um ambiente do tipo *protoboard* é semelhante ao observado na Figura 4.10. Os valores utilizados neste exemplo foram: resistência = 50Ω; capacitância = 1nF; fonte de tensão em banda base de sinal senoidal com sinal dc de 1V; amplitude de 3V, freqüência de 1kHz sem fase.

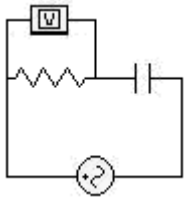

Figura 4.9 – Circuito a ser simulado

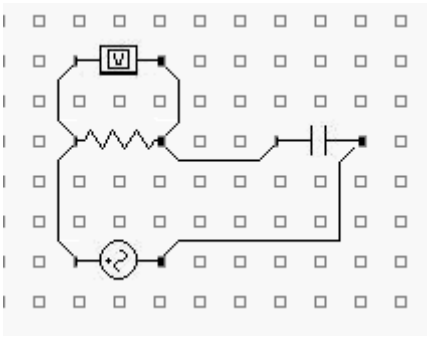

Figura 4.10 – Representação gráfica do circuito em um ambiente tipo *protoboard*.

O resultado obtido para esta simulação está representado na Figura 4.11. Como a fonte de tensão é uma fonte senoidal de 1kHz de freqüência, o espectro de freqüência (primeiro gráfico da figura) só apresentou esta componente. A existência do capacitor neste circuito resultou nas mudanças de fase verificadas no segundo item do comportamento. O último gráfico representa o sinal que está sendo aplicado nos terminais do voltímetro. Toda simulação de circuitos realizada terá como resultado uma ou mais figuras divididas nestes três gráficos distintos: espectro de freqüência, espectro de fase e tesão pelo tempo.

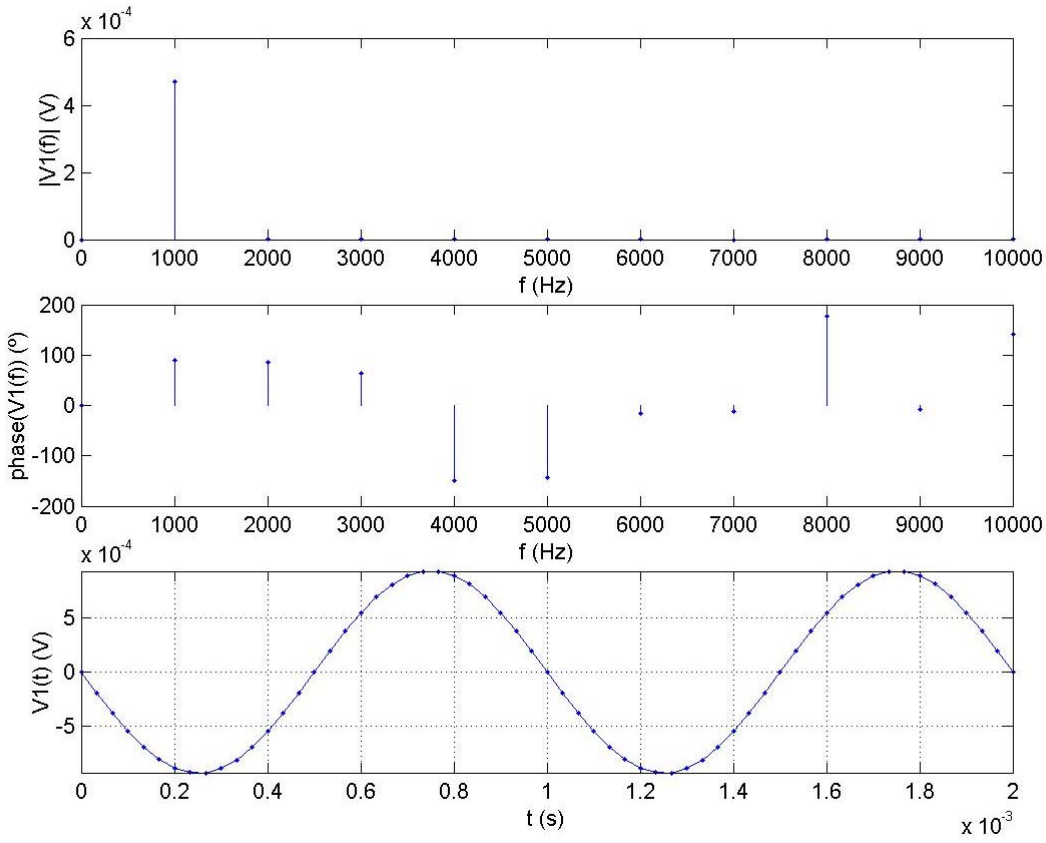

Figura 4.11 - Exemplo de resultado de simulações de circuitos

4.4.2 Envio e Recepção dos Dados de Simulação

A classe `CircuitSim` realiza a conexão com o servidor de simulação por meio de FTP. É feito um ftp que gera o arquivo de dados da *netlist* no diretório do servidor onde o simulador de circuitos está escutando. Ao encontrar este arquivo, o simulador realiza os cálculos e gera os resultados sob a forma de figuras utilizando como referência os amperímetros e voltímetros.

Após a escrita do arquivo da *netlist*, o `CircuitSim` passa a realizar uma busca pelas figuras em uma pasta específica onde o simulador gera as figuras. Esta busca é feita utilizando os nomes das dos amperímetros e voltímetros, passados por meio do vetor recebido junto com a *netlist*. Quando se percebe que as figuras foram geradas, utiliza-se o FTP para baixá-las à um diretório especifico, e o caminho e nome para encontrá-las são armazenados em uma *hashtable* que é devolvido à classe `RunCircuitSimulation`. Um diagrama de seqüência desses evento é mostrado na Figura 4.12.

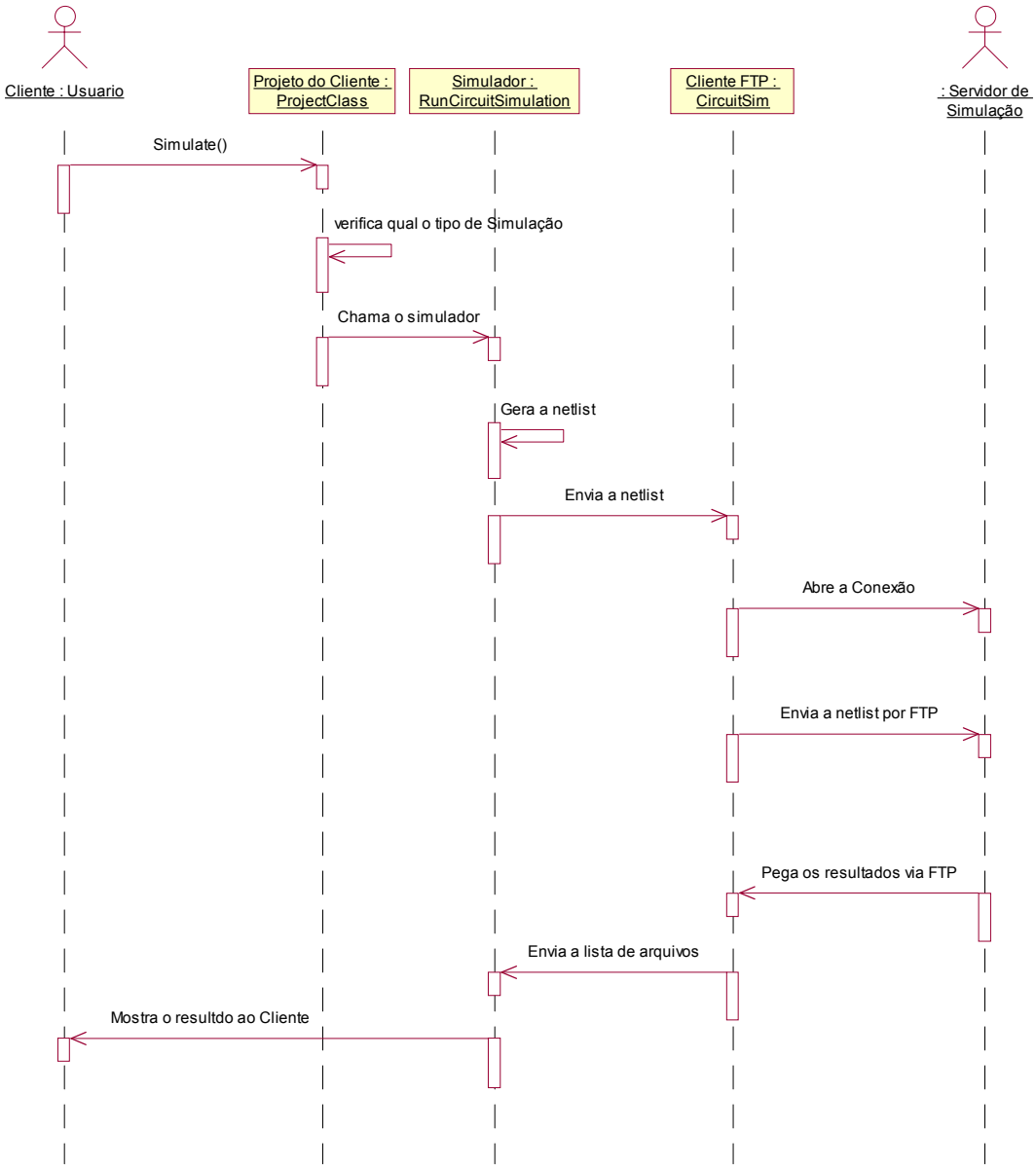

Figura 4.12 - Diagrama de Seqüência da Simulação de Circuitos

4.5 *Conclusão*

A interface de simulação ProgSim recebeu de maneira eficiente o acréscimo do módulo de simulação de circuitos. Muitas das funções acrescentadas têm por caráter apenas facilitar a utilização da interface para propósito de simulação de circuitos, como é o caso da criação de componentes de circuitos separada da criação dos outros componentes, visto que através da outra ferramenta pode-se gerar um componente de circuitos com sucesso comprovado apesar de ser um pouco mais trabalhoso.

O acréscimo do simulador de circuitos só representou o acréscimo de algumas classes e adaptação de duas das classes principais da interface ProgSim, comprovando a modularidade da filosofia orientada a objeto utilizada no programa.

Parte das classes acrescentadas, como por exemplo a `CircuitSystemArchitecture`, são herdeiras das classes principais do projeto. No caso deste exemplo, é herdeira da *`ArchitectureClass`*. Isto é importante para manter a filosofia dos objetos que permitiram ao programa a modularidade e flexibilidade para criação de novos ambientes de simulação, além de facilitar a incorporação deste módulo ao programa.

5 MÓDULO DE SIMULAÇÃO DE SISTEMAS CELULARES

5.1 *Introdução*

O conceito de sistema celular é concebido como um dos principais atalhos para resolver o problema de congestionamento espectral e capacidade de tráfego para o usuário. Oferece uma alta capacidade em uma limitada alocação de espectro sem mudanças significativas de tecnologia. O conceito de sistema celular parte do princípio de, em vez de utilizar-se um único transmissor de alta potência, fazer-se uso de muitos transmissores de baixa potência. Cada Estação Rádio Base (ERB) é responsável por fornecer cobertura para apenas uma pequena porção de área.

Assim como a demanda por serviço aumenta, assim podem acrescer o número de ERBs (com a diminuição correspondente na potência transmitida para evitar maiores interferências). Porém, a instalação e expansão do sistema celular são considerados dispendiosos, exigindo dos projetistas o máximo de eficiência com o mínimo de custo.

Ferramentas de simulação eficientes são de grande valia para este cenário, pois através delas é possível predizer a melhor configuração do sistema sem despesas desnecessárias. Assim, por exemplo, evita-se que sejam feitas instalações físicas para testes de melhor configuração do sistema.

A ferramenta ProgSim permite que sejam desenvolvidos módulos de simulação para qualquer sistema. Para validar tal ferramenta nesta tese de graduação, foram desenvolvidos um módulo de simulação de circuitos elétricos, já visto no capítulo anterior, e um outro módulo de simulação de sistema celular, que será detalhado neste capítulo. A seguir, será apresentada uma breve introdução sobre sistemas celulares e, em seguida, se analisará a modelagem e a implementação o módulo de simulação de sistema celular para a interface ProgSim.

5.2 *Introdução ao Sistema Celular*

5.2.1 Uma visão geral do sistema celular

Um sistema celular é formado por três partes básicas: telefones móveis (celulares), estações rádio base (ERB) e o SMC (*mobile switching center*). Os telefones celulares se comunicam via radio-transmissão com a ERB mais próxima que, por sua vez, converte esses

sinais de rádio para serem transferidos para uma SMC que contacta outro telefone celular ou um telefone fixo através da rede pública. Após essa operação a MST redireciona a chamada para outro telefone móvel ou fixo. A Figura 5.1 representa esse sistema.

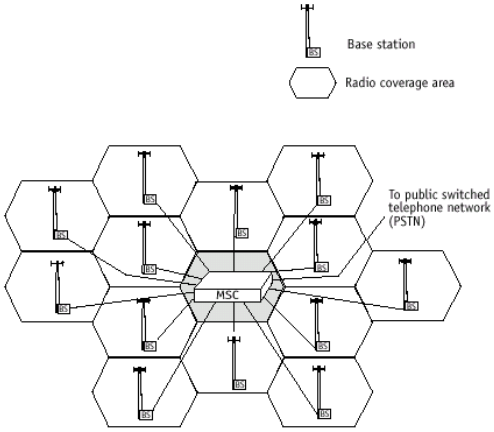

Figura 5.1 - Sistema Celular

As diversas SMCs são as responsáveis por interligar as pequenas áreas de cobertura, mais conhecidas por células, formando um sistema amplo de telefonia. À medida que um telefone celular se movimenta de uma célula para outra, sua freqüência de operação muda para a freqüência presente na nova área. Esse processo é conhecido como *hand-off,* mostrado na Figura 5.2.

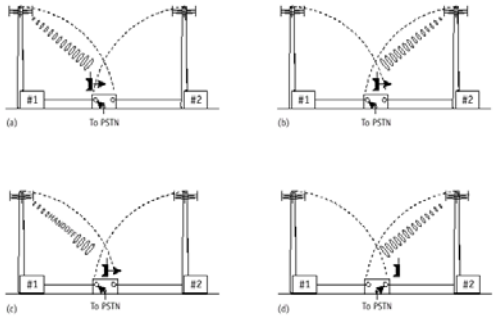

Figura 5.2 – Hand-off

(a) A estação radio base #1 detecta uma queda de potência do celular.

(b) A estação #2 detecta um novo sinal de potência.

(c) A estação ERB #1 envia uma mensagem de hand-off.

(d) O celular se adapta a nova freqüência da ERB #2 que inicia a transmissão.

5.2.2 Objetivos de um sistema celular ideal

Servir os usuários com serviços padrões de voz e serviços especiais de comunicação (PCS) como, por exemplo; qualidade otimizada, sistemas de mensagens via Pager ou integrado à Internet, serviços de armazenamento de dados, utilização econômica de energia.

Utilizar vários usuários com limitados canais de rádio, com multiplexação de canais.

Ter uma eficiência espectral razoável, utilizando mecanismos de reutilização de freqüência e técnicas de múltiplo acesso.

Compatibilidade internacional.

Possuir sistemas de transmissão mais apropriados para detecção e correção de falhas, garantindo qualidade de serviços.

5.2.3 Tipos de tecnologia de transmissão

Os sistemas atuais de comunicação móvel utilizam tanto sistemas analógicos de transmissão com modulação direta, quantos sistemas digitais de comunicação. Os tipos mais utilizados no mundo são, AMPS, NAMPS, TDMA (IS-136), GSM e CDMA (IS-95).

5.3 *Módulo de simulação para o projeto de sistema celular*

Apresenta-se nesta seção o segundo módulo implementado para a interface ProgSim com o objetivo de validar este projeto de graduação: o módulo de simulação de sistema celular. Para esta versão do ProgSim, este módulo basicamente executa simulações envolvendo Estações Rádio Base (ERB), apresentando os conceitos de mapa de potência irradiada isotrópica equivalente (EIRP), razão sinal-ruído e *best server*.

Para criar um novo projeto de sistema celular, depois do usuário iniciar o programa ProgSim e se autenticar, este deve selecionar, pelo menu ou pelo botão, a opção *New*. Surgirá uma caixa de diálogo para a seleção do tipo de sistema a ser trabalhado, onde deverá ser escolhida a opção *Cellular System*, como mostra a Figura 5.3.

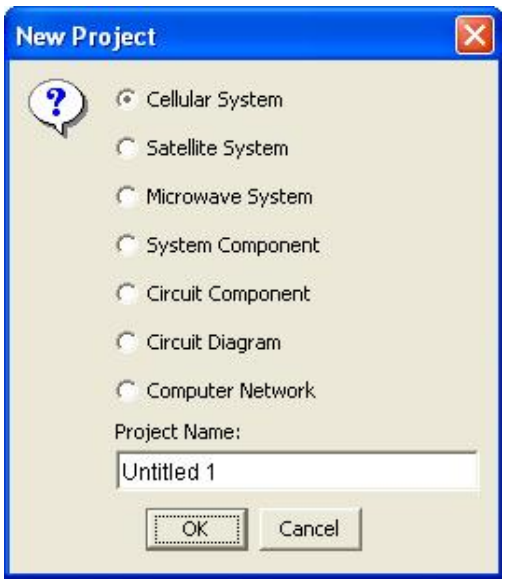

Figura 5.3 - Selecionar o projeto de sistema celular

Depois da seleção do tipo de projeto a ser criado, a construtora de `ProjectClass` se incumbe de popular a árvore de projeto com os objetos específicos, cada qual relacionado com o seu nodo. Para o nodo *System* é instanciado do `SystemClass` um objeto. Para esta versão do ProgSim, este objeto não contém informações relevantes para o projeto de sistema celular.

Em seguida é instanciado e armazenado no nodo relativo, um objeto de `CellularSystemArchitecture`, uma subclasse de `ArchitectureClass`. Tal objeto possui cinco atributos indispensáveis à simulação do sistema, sendo que dois deles são descritos pela classe `LocationClass`. Os objetos desta classe armazenam coordenadas geográficas apresentadas em latitude e longitude. Os atributos de `CellularSystemArchitecture` são:

`mapSize`: armazena o tamanho do mapa, medindo a distância entre o ponto de latitude 0°0’0’’N e longitude 0°0’0’’E até o ponto armazenado por este objeto.

`upperLeftLocation`: `LocationClass` que armazena a coordenada geográfica do ponto situado no canto superior esquerdo do mapa.

`mapPath`: URL que contém o caminho para o arquivo que possui a imagem do mapa.

`altituteMap`: matriz bidimensional onde os índices são relacionados às coordenadas em pixels, e cada valor representa a altura em uma determinada área do mapa. Este objeto é preenchido apenas depois de um mapa ser carregado.

`powerMap`: matriz tridimensional onde os índices representam a ERB e as coordenadas em *pixels*, e cada valor corresponde à potência recebida pela ERB em dada coordenada, ambos indicados pelos índices da matriz.

O mesmo acontece com os objetos de `CellularSystemBehavior`, uma subclasse de `BehaviorClass`. Para esta versão do ProgSim, o comportamento de um projeto de sistema celular conterá apenas o histórico de simulações deste projeto. Para tal, definiu-se apenas três atributos para a `CellularSystemBehavior`:

`simulationList`: uma lista, que será visualizada pelo usuário, contendo a data e a hora das simulações realizadas deste projeto.

`simulationDataTable`: *hashtable* que armazena a matriz tridimensional contendo a EIRP que corresponde ao resultado da simulação. O objeto-chave que referenciará cada elemento nesta *hashtable* será um objeto do tipo *Date* que contém a data e a hora da simulação.

`simulationLocationTable`: *hashtable* que armazena um vetor que contém as coordenadas de todas as ERBs no momento da simulação. O objeto-chave desta *hashtable* é o mesmo do item anterior.

Por fim, é instanciado e armazenado no nodo relativo um objeto de `CellularEnvironmentClass`, uma subclasse de `EnvironmentClass`. Para esta versão do ProgSim, no painel apresentado por este objeto oferece-se a opção definir as dimensões do mapa geográfico do sistema celular ou carregar um mapa já pré-definido, com imagem e informações como dimensões e mapa de altura. O principal atributo desta classe é o objeto `map` definido pela classe `MapClass`. Este objeto contém quatro atributos significativos:

`altituteMapPath`: URL que contém o caminho para o arquivo que possui a imagem do mapa de altura.

`altituteMap`: matriz bidimensional onde os índices são relacionados às coordenadas em *pixels*, e cada valor representa a altura em uma determinada área do mapa. .

`upperLeftLocation` e `lowerRightLocation`: objetos de `LocationClass` que armazenam as coordenadas geográficas dos pontos situados no canto superior esquerdo e no canto inferior direito do mapa, respectivamente.

Depois de iniciar por completo o projeto de sistema celular, o primeiro passo para configurar o projeto para a simulação é definir o mapa. Isto pode ser realizado selecionando o

nodo *Environment* na árvore de projeto. A interface ProgSim apresentará uma tela tal qual está descrita na Figura 5.4.

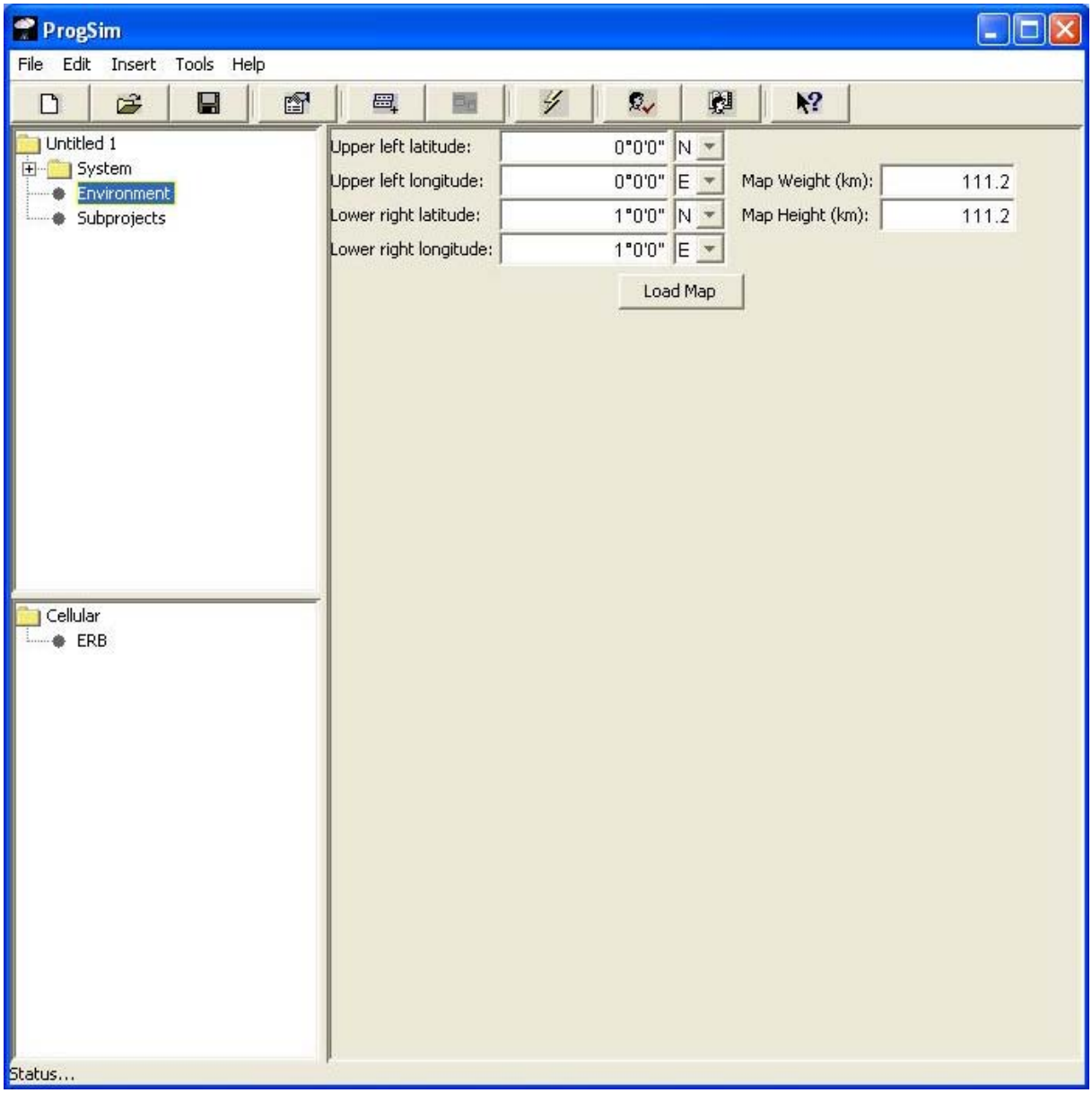

Figura 5.4 - Configuração de mapa

Nesta tela pode-se configurar as dimensões do mapa. Se feito apenas isto, na simulação será considerado que o mapa utilizado será plano, de altura constante e igual a zero. Para carregar um mapa com seu respectivo mapa de altura, utiliza-se o botão *Load Map*, que instancia um objeto de `JFileChooser` para navegar no sistema de arquivos no intuito de localizar um arquivo de extensão MAP que contém o caminho para o mapa representativo e o mapa de altura, ambos encapsulados em um objeto de `MapClass`. Este arquivo foi gerado a partir de `MapCreatorApplication` e `MapCreator`. Estas duas classes são

independentes do ProgSim e suas funções basicamente se constituem em traduzir em números os arquivos de imagem contendo o mapa de imagem relacionada a uma dada região. Um exemplo de uma imagem contendo o mapa de alturas segue abaixo, na Figura 5.5, onde percebe-se com uma certa dificuldade, admite-se, que representa o mapa de alturas de parte da região do Distrito Federal.

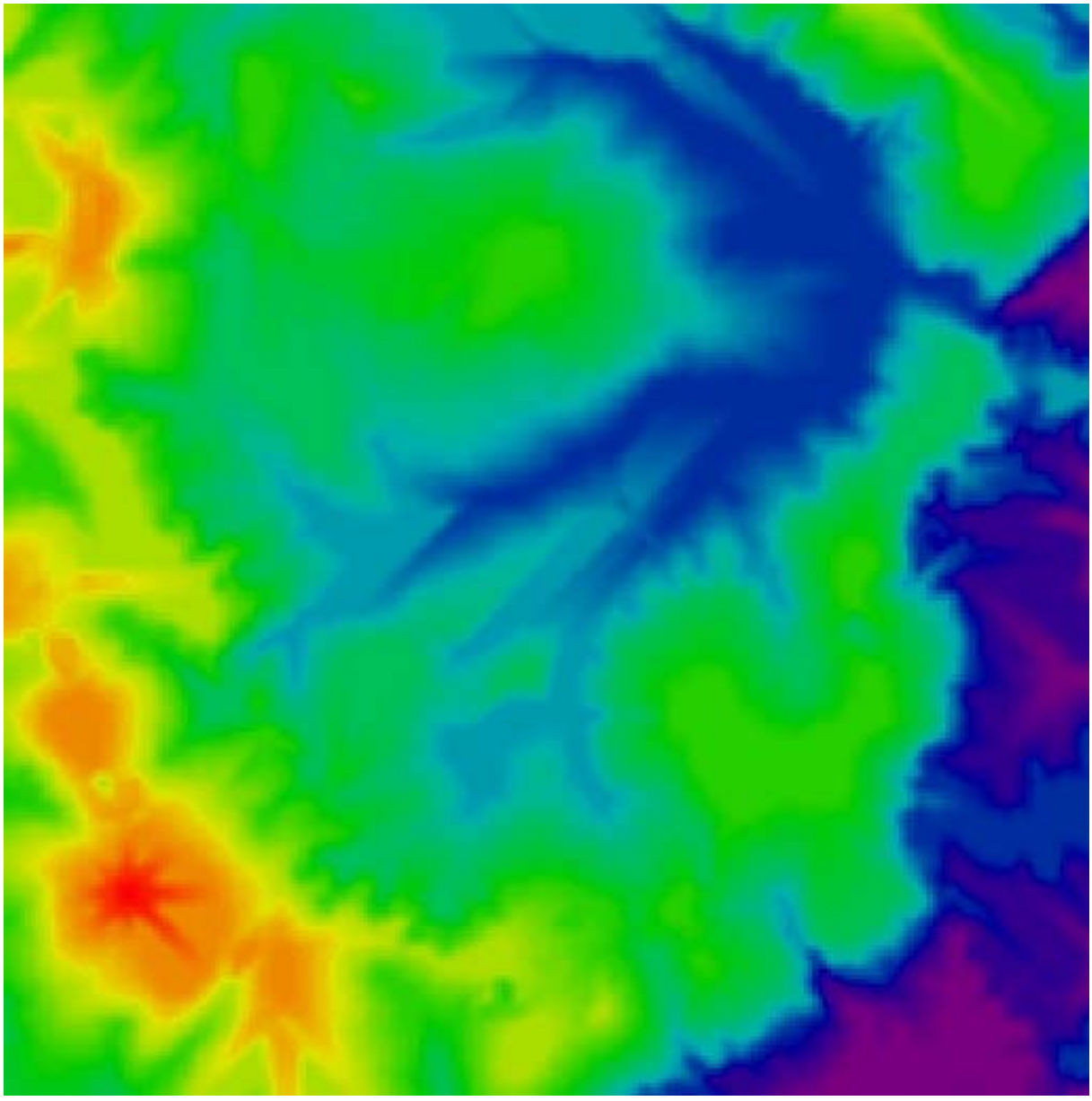

Figura 5.5 - Imagem contendo o mapa de altura de Brasília

O algoritmo utilizado para transformar esta imagem para uma matriz bidimensional encontra-se no escopo do método `setImage(URL altitudeMapPath, MediaTracker tracker)` que se encontra em `MapClass`. Este algoritmo utiliza a seguinte relação para traduzir a imagem em números:

Intervalo de Cor (formato RGB)	Intervalo de Altura (em metros)
120, 0, 110 a 10, 46, 110	900 a 950
10, 46, 110 a 10, 175, 157	950 a 1000
10, 175, 157 a 10, 175, 10	1000 a 1067
10, 175, 10 a 38, 210, 150	1067 a 1130
38, 210, 150 a 152, 205, 10	1130 a 1150
152, 205, 10 a 230, 210, 10	1150 a 1200
230, 210, 10 a 220, 138, 10	1200 a 1218
220, 138, 10 a 230, 138, 10	1218 a 1250
230, 138, 10 a 255, 0, 0	1250 a 1300

Tabela 5.1 - Conversão de cor para número

O método `setImage()` busca uma relação gradual para minimizar a discrepância entre pontos vizinhos. Assim, pontos da imagem cujas cores não se encontram em nenhum intervalo descrito na Tabela 5.1, este ponto busca o valor da altura de seu ponto vizinho. Portanto, e notório que o mapa da Figura 5.5 tem a finalidade mais ilustrativa do que repositório de informação precisa.

Carregado o mapa, ao selecionar o nodo de arquitetura do sistema, o painel principal apresenta um mapa ilustrativo de Brasília, referenciado pela URL no objeto `MapPath` de `CellularSystemArchitecture`. Este último carrega o mapa ilustrativo e o amplia três vezes, para melhor visualização do usuário. Para isso, faz-se uso de manipulação de imagens, processo descrito no Capítulo 2.

Ao passar o ponteiro do mouse sobre o mapa, são apresentadas informações de coordenada e altura referentes ao ponto determinado pelo mouse. Para tanto, a classe `CellularSystemArchitecture` sobrescreve o método `refreshData(Point pointerLocation)` de `ArchitectureClass`. Para o projeto de sistema celular, este método, a partir do valor de entrada que contém as coordenadas em *pixels* do mouse, encontra as coordenadas geográficas e a altura relacionada a este ponto. Após a simulação, também são apresentadas informações pertinentes a esta, como a potência recebida, a razão sinal-ruído e o *best server* referente a este ponto.

Terminada esta parte, o próximo passo consiste em adicionar as ERBs no mapa, pelo processo de DnD, descrito no Capítulo 2. A diferença para o projeto de sistema celular está na confirmação de coordenadas, agora sendo apresentadas em termos de latitude e longitude, como mostra a Figura 5.6.

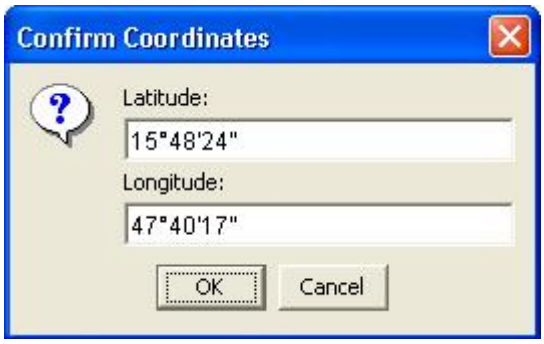

Figura 5.6 - Confirmação das coordenadas geográficas da ERB

Após a inserção de todas as ERBs necessárias, o projeto está apto a executar sua simulação através do botão *Run*. Ao clicá-lo, é invocado o método `simulate()` de `ProjectClass`. Este método verifica qual o tipo de sistema que será simulado, no caso sistema celular. Em seguida, encapsulam-se os objetos que contém o mapa de altura e os atributos essenciais das ERBs e invoca-se o método remoto `simulaERB()` a partir de `CellSim`, utilizando a operação de RMI. Tal método retorna uma matriz tridimensional contendo o resultado da simulação que consiste na potência recebida em cada ponto relativo a cada ERB. O diagrama de seqüência da Figura 2.3 mostra as mensagens durante o processo descrito.

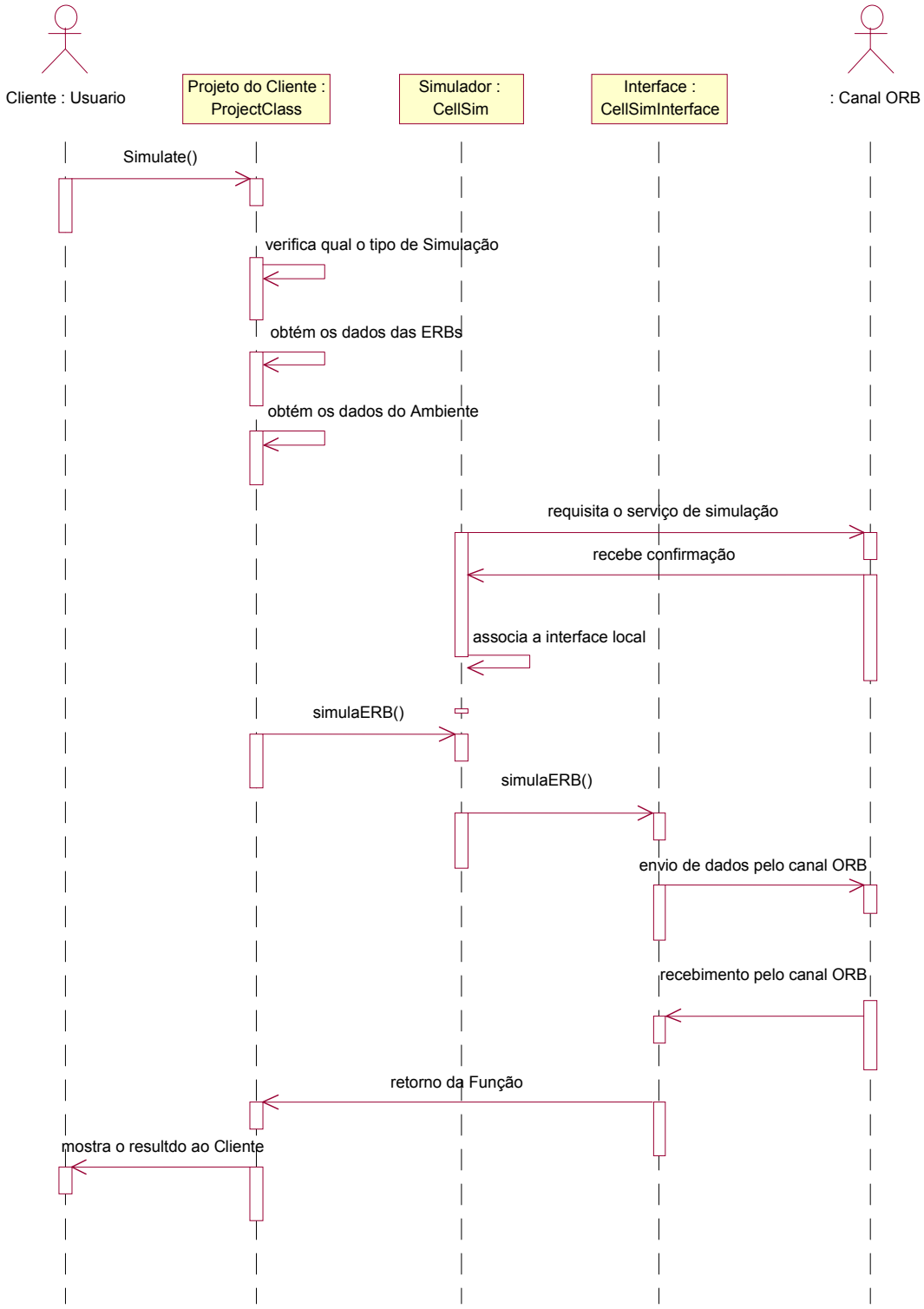

Figura 5.7 – Diagrama de Seqüência da Simulação de Sistemas Celulares

Finalmente, `CellularSystemArchitecture` recebe o resultado da simulação e o converte para uma imagem sobreposta à imagem do mapa com transparência de fator 68%. A conversão obedece a relação descrita na Tabela 5.2. Por fim, a Figura 5.7 apresenta o resultado final da simulação.

Intervalo de potência (em dBm)	Intervalo de cor (formato ARGB)
> -70	255, 0, 0, 170
-70 a -80	255, 0, 0, 170 a 255, 255, 0, 170
-80 a -90	255, 255, 0, 170 a 0, 255, 0, 170
-90 a -100	0, 255, 0, 170 a 0, 255, 255, 170
-100 a -110	0, 255, 255, 170 a 0, 0, 255, 170
-110 a -120	0, 0, 255, 170 a 0, 0, 255, 0

Tabela 5.2 - Conversão de número para cor da potência recebida

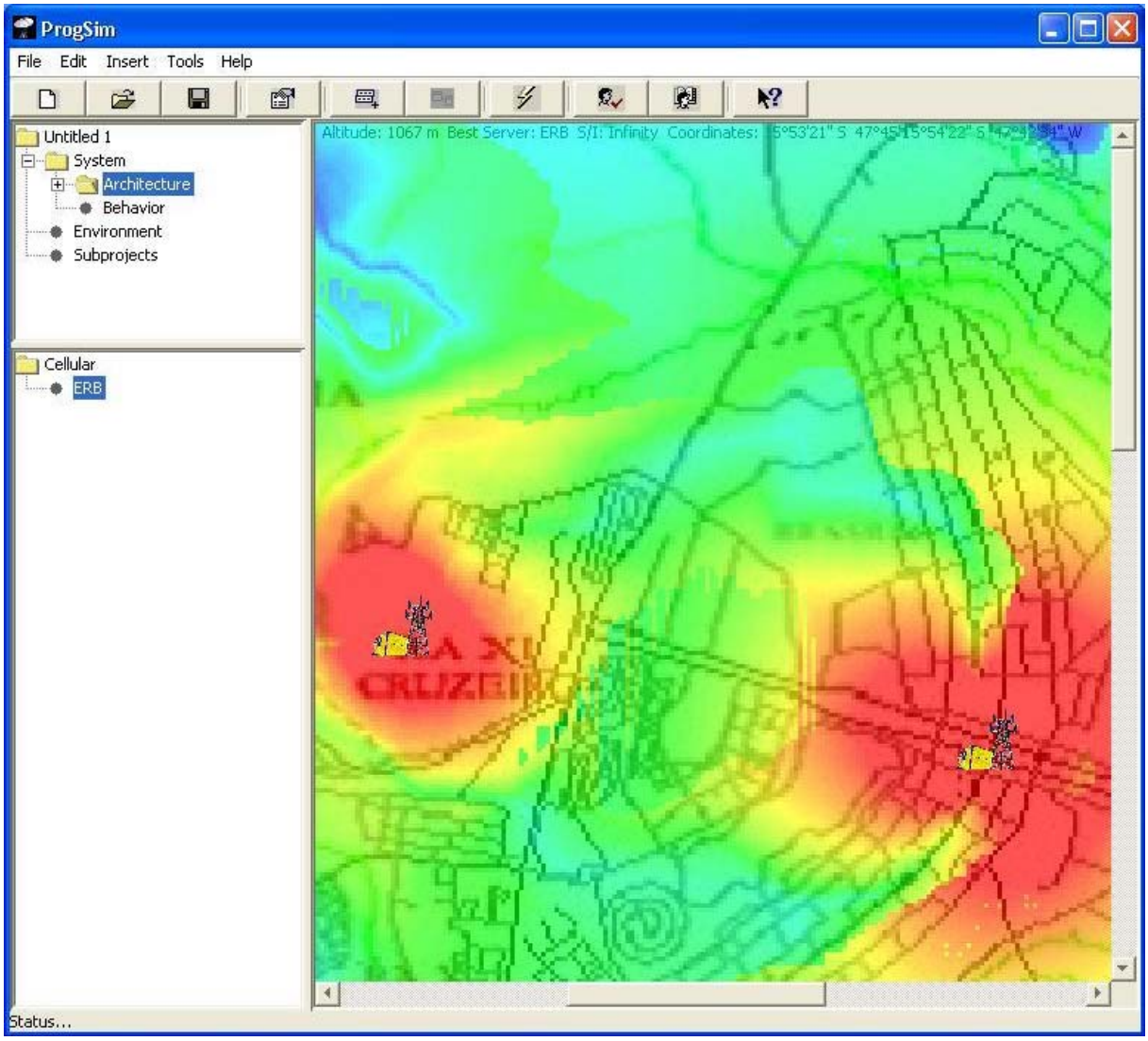

Figura 5.7 – Resultado da simulação

Para o módulo de simulação de sistema celular foi desenvolvido um adendo para inserir um mapa com o tráfego do sistema, mas por ter expirado o prazo, não foi possível colocar mais este adendo ao módulo em funcionamento.

5.4 *Conclusão*

Este capítulo apresentou apenas alguns conceitos pertinentes a sistemas celulares, que, na verdade, são de uma complexidade considerável. O módulo descrito neste capítulo limitou-se apenas a viabilizar a interface ProgSim, provando seus conceitos de neutralidade e modularidade.

Porém, verificou-se a importância concernente à simulação de sistema celular; de como é uma ferramenta essencial no projeto desse sistema. O próprio mercado sinaliza tal

importância, estabelecendo preços bem generosos a softwares que executam essas funções, indicando que tal ferramenta será sempre bem-vinda.

Portanto, a continuidade e o melhoramento do módulo de simulação de sistemas celulares, apresentado neste capítulo, tornam-se bem propícios tanto para o mercado quanto para os objetivos inerentes ao projeto ProgSim.

6 SERVIDOR DE SIMULAÇÃO (SERVERSIM)

6.1 *Introdução*

ServerSim é o responsável por receber os dados dos clientes remotamente, processar, enviar os resultados ao cliente, além de trocar informações com o banco de dados para verificar a permissão dos usuários simularem projetos. Foi mostrado nos capítulos 4 e 5 o que ocorre no lado cliente em cada simulação. Nesse capítulo será mostrado como as respectivas simulações ocorrem no lado do servidor.

6.2 *Objetos do ServerSim*

Este tópico apresenta um resumo das funções das principais classes do servidor.

6.2.1 CellSimImpl

Implementação da interface `CellSimInterface` do pacote protótipos. Essa classe é responsável por criar o `stub` e o `esqueleto`, onde as funções de ambos são detalhadas no item 3.3.2.3, que vai ser usado para negociar no canal ORB. Essa classe é a principal responsável pela simulação celular.

6.2.2 FTPDaemon

Classe responsável por escutar a porta FTP para criar uma conexão a cada nova requisição de um cliente FTP.

6.2.3 FTPConnection

Classe responsável pela conexão com o cliente. A cada nova conexão de um cliente a classe `FTPDaemon` cria uma nova *thread* da classe `FTPConnection`. Os comandos aceitos por esta classe são descrito na RFC959. Além disso, cada comando é definido como um método dentro desta classe.

6.2.4 *Lee*

Implementação em Java do modelo de simulação *Lee*. As fórmulas implementadas nesta classe são mostradas na equação 6.1.

$$L = L_M + \gamma \log(d) + 10n \log(\frac{f}{900}) - \delta$$

$$\delta = \delta_1 + \delta_2 + \delta_3 + \delta_4$$

$$\delta_1 = 20 \log(\frac{h_{erb}}{30{,}48}) dB$$

$$\delta_2 = \left\{ \begin{array}{l} 10 \log(\frac{h_{movel}}{3}) dB \Rightarrow h_{movel} \leq 3m \\ 20 \log(\frac{h_{movel}}{3}) dB \Rightarrow 3 \leq h_{movel} \leq 10m \end{array} \right\}$$

$$\delta_3 = G_{ERB} - 6dB$$

$$\delta_4 = G_{movel} dB$$

Variáveis

d - distancia do ponto calculado a erb em quilômetros.

f - freqüência central em megahetz.

h_{erb} - altura da erb em metros

h_{movel} - altura do móvel metros.

Constantes

Lm = 105 dB

γ = 43,3 dB

Equação 6.1 - Equação do Modelo de Propagação de Lee

Para o cálculo da distância, foi criado um método que retorna a distância em *pixel* do ponto que deseja medir a potência e onde se encontra a ERB. Depois este resultado é dividido pela proporção do mapa para achar a distância correspondente em kilômetros. A `h`$_{erb}$ é a altura equivalente entre a altura do mapa no ponto onde está sendo calculada a perda e a altura onde está a ERB e a sua altura em relação ao solo. Essa classe é usada em `CellSimImpl`.

6.2.5 MainFrame

Essa classe é a classe principal que inicia dois *deamons*. Um deles é *o* `FTPDaemon` e o outro é o `ServidorCell`.

6.2.6 ServidorCell

Essa classe é responsável por registrar o serviço de simulação celular no canal ORB para que os clientes encontrem uma implementação válida que se encontra na classe `CellSimImpl`.

6.3 *Simulação Celular no Servidor*

O diagrama de seqüência, mostrado na Figura 6.1, é um paralelo com o diagrama do cliente mostrado na Figura 5.7.

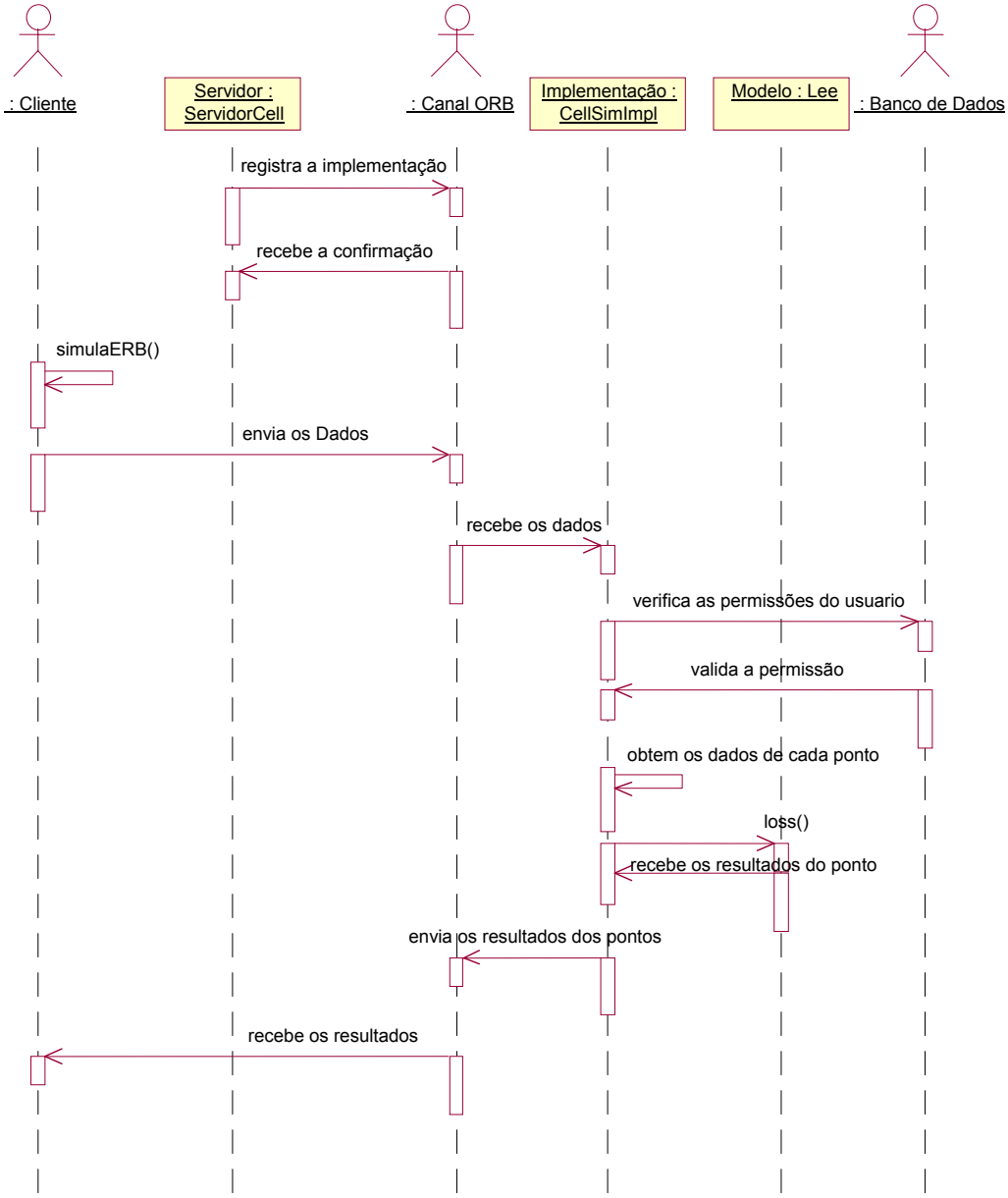

Figura 6.1 - Diagrama de Sequencia da Simulação Celular

Os principais pontos a serem observados neste diagrama de seqüência é que pode-se mudar com facilidade o código de simulação celular sem precisar mudar a interface com o canal ORB e o algoritmo do cliente. Além disso, para o cliente a chamada de simulação se porta como uma chamada local e isto torna o processo remoto transparente para o cliente. Outro ponto importante é que o acesso ao banco é feito pelo servidor. Isso isola o banco, evitando acessos indesejados do cliente.

6.4 *Simulação de Circuitos no Servidor*

O diagrama de seqüência mostrado na Figura 6.2 é um paralelo com o diagrama do cliente mostrado na Figura 4.12.

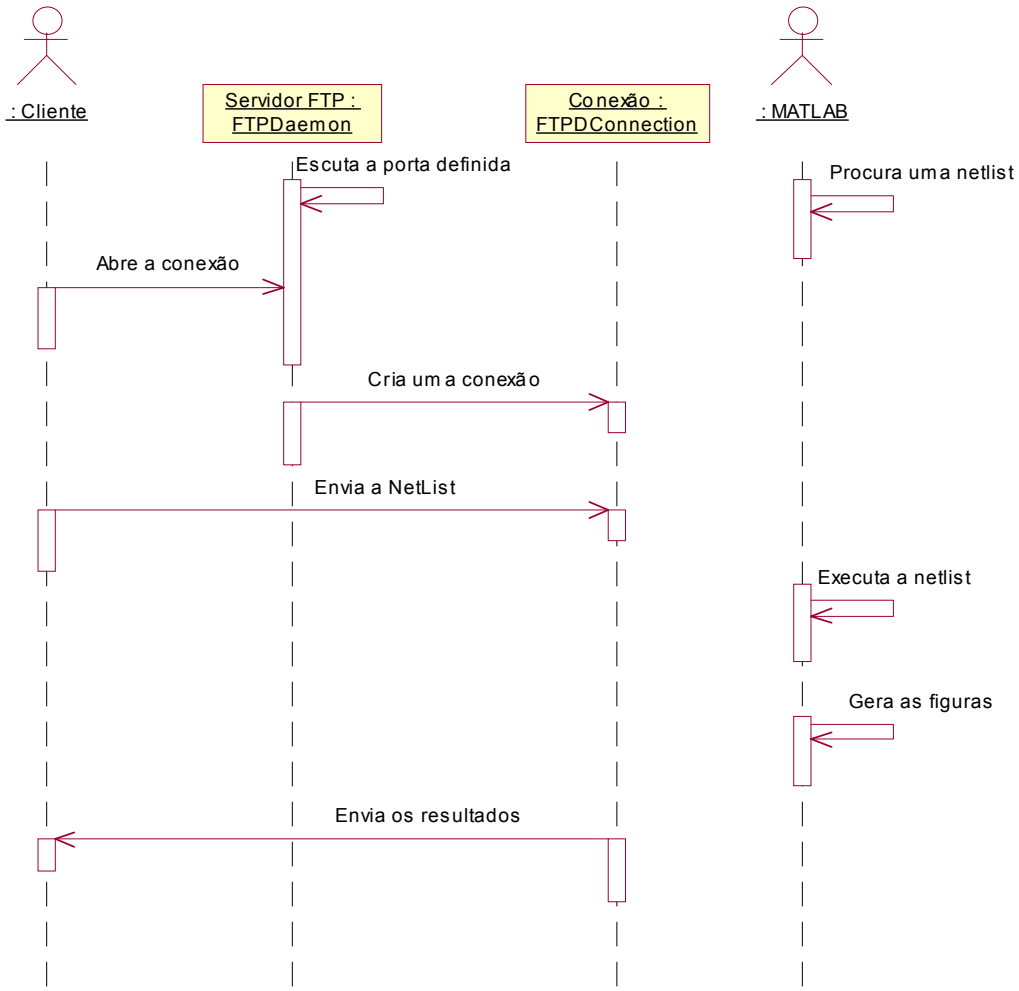

Figura 6.2 - Diagrama de Seqüência da Simulação de Circuitos

O diagrama de seqüência da simulação de circuitos é mais simples, pois a rotina no MATLAB é a principal responsável pela a simulação. As classes no ServSim são apenas uma interface para a troca necessária de arquivos entre o cliente e o MATLAB. Outro ponto é que a rotina do MATLAB detecta uma nova *netlist* automaticamente, não sendo necessária uma troca de mensagens entre o ServSim e MATLAB

6.5 *Conclusão*

Comparando os diagramas das Figura 6.1 e Figura 6.2 é possível perceber como a implementação da simulação é bem diferente em cada módulo. Porém, essas diferenças são transparentes para o usuário como foi mostrado nos capítulos 4 e 5. Isso demonstra que, para o acréscimo de novos módulos, a forma como o simulador for implementado é indiferente. Os módulos de simulação são feito em "threads" diferentes, deixando cada um independente do status do outro. Assim sendo, para acrescentar um novo módulo, é necessário somente criar uma nova "thread".

CONCLUSÃO

Foi realizada a modelagem de um ambiente de simulação neutro, visando a utilização deste sistema para diversos tipos de simulação. Este ambiente foi modelado segundo uma estrutura cliente-servidor, para possibilitar a centralização do processamento em uma máquina de grande porte. Esta centralização tem por objetivo tornar o sistema mais rápido para o usuário.

A implementação deste sistema se fez na linguagem Java, para que a interface do cliente seja suportada em diversos ambientes, sendo necessário para tanto, apenas que a maquina virtual Java esteja instalada na máquina.

Além da implementação da interface neutra, foram realizadas também a implementação dos módulos de simulação de circuitos elétricos e de sistemas celulares. Estes módulos foram elaborados de forma a minimizar as alterações no código da interface principal do programa, sendo utilizada muita herança e criação de algumas classes novas para auxiliar neste processo.

Os dois módulos implementados tiveram características muito distintas, das quais se destacam a posição dos elementos, os nós de conexão dos componentes, o envio e recebimento dos dados da simulação e a própria simulação. Apesar destas diferenças, o programa foi implementado de forma que isto seja transparente ao usuário. A estrutura cliente-servidor utilizada pode conciliar qualquer forma de transmissão de dados e isso foi mostrado com os módulos de simulação, onde foi usado FTP para trocar informação para a simulação de circuitos e RMI para o módulo de sistemas celulares. O servidor de simulação, assim como a estrutura cliente-servidor, realizou as duas formas de simulação distinta, sendo uma realizada por um algoritmo implementado em Java, e a outra por um simulador rodando em MATLAB. Isso demonstra que para a criação de um novo modulo ou em um dos módulos desenvolvidos, a forma que a simulação é calculada independe da interface.

Este ambiente ainda tem correções e ajustes a serem feitos. Dentre estes, podemos citar a implementação de algumas funcionalidades; a melhoria na apresentação de alguns dados e correção de *bugs*, além do acréscimo de novos módulos e evolução dos já existentes; com a melhoria dos módulos será possível utilizar o ProgSim nos laboratórios da Universidade de Brasília. Isso diminuiria o gasto da universidade com a compra de simuladores de empresas que têm um preço muito alto para a instituição.

Esta prevista a continuidade deste projeto por um grupo de alunos da graduação que planejam implementar o módulo de simulação de tráfego de dados em redes de computadores.

BIBLIOGRAFIA

[1] - CHAIBEN, Marçal M.: *Implementação da Interface de Processamento Distribuídos de Simulação de Sistemas Móveis e de Circuitos* – Universidade de Brasília, abril de 2002.

[2] - *Programação Orientada a Objeto.* Acessado em setembro de 2002 em http://www.unifieo.br/revista/rev1999/ProgOO.htm

[3] - GOSLING, James; MCGILTON, Henry: *The Java Language Environment, a White Paper*. Acessado em setembro de 2002 em http://java.sun.com/docs/white/langenv/

[4] - KRAMER, Douglas: *The Java Platform, a White Paper*. Acessado em setembro de 2002 em http://java.sun.com/docs/white/platform/javaplatformTOC.doc.html

[5] - HORSTMANN, Cay S. & CORNELL, Gary. *Core Java 2 Volume 1 – Fundamentos*. Makron Books, 2001.

[6] - HORSTMANN, Cay S. & CORNELL, Gary. *Core Java 2 Volume 2 – Recursos Avançados*. Makron Books, 2001.

[7] - *The Java Tutorial*. Acessado em setembro de 2002 em http://www.java.sun.br/docs/book/tutorial/

[8] - *Support Readness Documentes*. Acessado em setembro de 2002 em http://acess1.sun.com/RSDs/access1_srds.html

[9] - *OMG Unified Modeling Language Specification. Version 1.4*, September 2001. http://www.omg.org/technology/documents/formal/uml.htm

[10] - POOLEY, Rob e STEVENS, Perdita. *Using UML, Software Engineering with Objects and Components*. Harlow: Addison Wesley, 1999

[11] - GAY, Warren W.: *Linux Socket Programming: By Example*. Que. Abril de 2000.

[12] - Site oficial do CORBA feito pela OMG. Acessado em setembro de 2002 em http://www.corba.org

[13] - CORBA and Java(TM) technologies. Acessado em setembro de 2002 em http://java.sun.com/j2ee/corba/

[14] - BRITO, Leonardo da C.: *Métodos do Equilíbrio Harmônico – Envoltória Não-Linear para Simulação de Circuitos RF Multiexcitados por Sinais Modulados Analógicos e Digitais*. Universidade de Brasília. Março de 2001.

[15] - *UNIFLEX- Focus de Tecnologia: FLEX ODBC*. Acessado em setembro de 2002 em http://www.uniflex .com.br/flexodbc.htm

[16] - *CARIBOU LAKE: CLS JDBC Driver Types*. Acessado em setembro de 2002 em http://www.cariboulake.com/techinfo/jdbc_driver_types.html

[17] - *JDBC: Conectividade com o Banco de Dados*. Acessado em setembro de 2002 em http://www.inf.ufrgs.br/aulas/redesd/jdbc.pdf

[18] - *Informação de Java*. Acessado em setembro de 2002 em http://www.dcc.ufmg.br/~corelio/java.html

[19] - PETRINI Jr., Juracy; GARCIA, Flávio C.; BARCELLOS, Heitor A.: *JDBC*. Pontifícia Universidade Católica do Rio Grande do Sul. Porto Alegre. Acessado em setembro de 2002 em http://planeta.terra.com.br/informatica/arruda/Downloads/artigo09/

[20] - *Developer's Community for Java Data Objects*. Acessado em setembro de 2002 em http://www.jdocentral.com/index.html

[21] - *JDBC Data Access API*. Acessado em setembro de 2002 em http://java.sun.com/products/jdbc/related.html

[22] - *A Comparison Between Java Data Objects (JDO), Serialization and JDBC for Java Persistence*. Acessado em setembro de 2002 em http://www.jdocentral.com/pdf/DavidJordan_JDOversion_12Mar02.pdf

[23] - *Java Database Connectivity*. Acessado em setembro de 2002 em http://www.computerworld.com/softwaretopics/software/story/0,10801,43545,00.html

[24] - *RFC959: FTP: Overview*. Acessado em setembro de 2002 em http://www.w3.org/Protocols/rfc959/2_Overview.html

[25] - SAUNDERS, Simon R.: *Antennas and propagation for wireless communication systems*. John Wiley & Sons LTD. University of Surrey, Guildford, UK. Setembro de 2000.