UNIVERSIDADE FEDERAL DA PARAÍBA

CENTRO DE CIÊNCIAS E TECNOLOGIA

COORDENAÇÃO DE PÓS-GRADUAÇÃO EM INFORMÁTICA

Dissertação de Mestrado

ESPECIFICAÇÃO DE COMPONENTES PARA UM AMBIENTE DE SIMULAÇÃO DE REDES TCP/IP

Juliana Camboim Lopes de Andrade Lula

Maria Izabel Cavalcanti Cabral
Orientadora

Campina Grande, Paraíba, Brasil
Julho de 2001

UNIVERSIDADE FEDERAL DA PARAÍBA

CENTRO DE CIÊNCIAS E TECNOLOGIA

COORDENAÇÃO DE PÓS-GRADUAÇÃO EM INFORMÁTICA

ESPECIFICAÇÃO DE COMPONENTES PARA UM AMBIENTE DE SIMULAÇÃO DE REDES TCP/IP

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Informática da Universidade Federal da Paraíba – Campus II, como parte dos requisitos necessários para obtenção do grau de Mestre em Informática.

Área de Concentração: Redes de Computadores

JULIANA CAMBOIM LOPES DE ANDRADE LULA

Maria Izabel Cavalcanti Cabral
Orientadora

Campina Grande, Paraíba, Brasil
Julho de 2001

LULA, Juliana Camboim Lopes de Andrade

L955E

Especificação de Componentes para um Ambiente de Simulação de Redes TCP/IP.

Dissertação (Mestrado) - Universidade Federal da Paraíba, Centro de Ciências e Tecnologia, Coordenação de Pós-Graduação em Informática, Campina Grande, Pb, Julho de 2001.

97 p. Il.

Orientador: Maria Izabel Cavalcanti Cabral

Palavras Chaves:

1. Simulação Digital
2. Componentes de Software
3. Redes TCP/IP

CDU - 519.876.5

"Gosto que haja dificuldades em
minha vida, pois sem obstáculos
nao haveria nem esforço nem
luta, e a vida seria insípida"
Khalil Gibran

Agradecimentos

À minha orientadora, profa. Maria Izabel C. Cabral, que sempre me transmitiu força, incentivo e entusiasmo sem os quais este trabalho não teria sido realizado.

De forma muito especial, aos meus pais queridos, sem os quais nada em minha vida até hoje seria possível. E aos meus irmãozinhos Genaro, Aurora e Joãozinho, que sempre estiveram ao meu lado nos momentos mais difíceis.

À minha família pelo apoio, ela que de alguma forma esteve presente durante este trabalho. Principalmente aos meus avós, tia Socorro e Jhésus que de todos foram os que estiveram mais próximos.

Aos meus amigos do mestrado pelos momentos agradáveis e pelas trocas de conhecimento.

De forma carinhosa, à Karina, Claudia, Tarig, Renato, Ladjane, Kyller, Gustavo, Lília, Juliana, Glauco e todos os outros cuja amizade me é muito cara.

Aos professores do Mestrado, que foram incentivadores além de transmissores de conhecimento.

Aos meus amigos de graduação que, mesmo de longe, sempre me acompanharam e me proporcionaram momentos de alegria.

Aos meus amigos do prédio, principalmente Emmanuel e Hélio, pelas farras e pela companhia.

Á Marcão pela sua grandiosa contribuição.

Aos funcionários do DSC e da COPIN pela paciência e colaboração.

À Inês, Romildo e Jô pelos deliciosos lanches e momentos de descontração.

Enfim, obrigada a todos que de alguma forma me ajudaram e me inspiraram.

Resumo

Devido à complexidade inerente aos sistemas de Redes de Computadores e ao fato de que algumas tecnologias ainda não estão exploradas ou mesmo implementadas totalmente, ferramentas de simulação digital apresentam-se como importantes opções na análise desses sistemas. Nesta Dissertação de Mestrado foram especificados componentes de software que facilitam a construção de ferramentas de simulação voltadas para a modelagem e análise de desempenho de redes de computadores com a tecnologia TCP/IP. A utilização de uma abordagem de desenvolvimento orientado a componentes permite que novas aplicações possam ser construídas visualmente a partir de um conjunto de componentes interligados, usando um editor gráfico. Os componentes aqui especificados exibem as funcionalidades essenciais para uma simulação orientada a eventos, tais como escalonamento de eventos, geração de valores aleatórios, controle do relógio e coleta de dados para o cálculo de medidas de desempenho. Nessa especificação foi utilizada a Linguagem de Modelagem Unificada (UML) enfocando, principalmente, as fases de análise e projeto.

Abstract

Due to both the complexity held down by the systems of Computer Networks and the fact that some technologies have not yet being explored or even completely implemented, the digital simulation tools are presented as important options in the analysis of these systems. In this Master Dissertation some software components were specified to facilitate the construction of the simulation tools that are applied in both modeling and performance evaluation of computers networks with TCP/IP technology. The utilization of a component-oriented software development approach allows new applications to be constructed virtually by means of a set of components linked with each other, using a graphic editor. The components specified in this work present the essential functionality to event-oriented simulations like event scheduler, clock control, random value generators and performance measures collect. In this specification, it was utilized the Unified Modeling Language (UML), on focusing principally the analysis and project phases.

SUMÁRIO

LISTA DE FIGURAS

Capítulo 1

1 Introdução

A indústria de computadores, apesar de ainda jovem em relação às outras indústrias (exemplo: automóveis), teve um crescimento surpreendentemente rápido nas últimas décadas [Soares, 95]. Os avanços tecnológicos sempre crescentes na microeletrônica e nas comunicações, a necessidade do compartilhamento de recursos (hardware e software) e a tendência crescente de descentralização geográfica e funcional do uso de computadores ocasionaram o surgimento das Redes de Computadores. As Redes de Computadores se caracterizam por serem um conjunto de computadores autômatos (cada processador tem poder de decisão) e interconectados (capazes de trocar informações) [Tanembaum, 94].

As Redes de Computadores são classificadas em redes locais (LANs), metropolitanas (MANs) e geograficamente distribuídas (WANs), de acordo com sua escalabilidade [Soares, 95]. As WANs são redes que se estendem por uma área geográfica bastante abrangente, como uma cidade ou um continente. Suas subredes são compostas de dois componentes: as linhas de transmissão (também chamadas de circuitos ou canais) e os elementos de comutação, que conectam duas ou mais linhas de transmissão.

As Redes de Computadores são exemplos de sistemas discretos. Sistemas discretos caracterizam-se pelo fato do processo de mudanças de estados ocorrer em pontos discretos do tempo. Tais sistemas podem ser modelados usando o paradigma de redes de filas [Kleinrock, 75].

Para solucionar modelos de redes de filas podem-se usar técnicas exatas, substanciadas pela Teoria das Filas [Kleinrock, 75], ou técnicas aproximadas. As técnicas aproximadas geralmente utilizam métodos numéricos para solucionar esses modelos. A simulação digital é a técnica aproximada mais usada, onde um programa de computador simula o funcionamento do sistema [Soares, 90].

Devido à complexidade inerente aos sistemas de Redes de Computadores e ao fato de que algumas tecnologias ainda não estão exploradas ou mesmo implementadas totalmente, a técnica de Simulação Digital apresenta-se como uma importante opção na análise desses sistemas.

A técnica da simulação envolve um sistema e um modelo desse sistema. Os sistemas são, geralmente, estudados visando a análise de seu desempenho. Do ponto de vista prático, a Simulação Digital constitui-se no projeto e construção de modelos computadorizados de sistemas reais ou propostos, com o propósito de realizar experimentos numéricos que visam uma melhor compreensão do comportamento desses sistemas em um dado conjunto de condições específicas [Kelton, 98].

Ferramentas de simulação permitem modelar e avaliar o desempenho de sistemas discretos complexos com um grande número de variáveis e operações. Sendo assim, a utilização de ferramentas de simulação digital tem merecido grande atenção por parte da comunidade científica que se volta para o desenvolvimento de tais sistemas.

Nos últimos anos, os objetivos das pesquisas de modelagem e avaliação de desempenho de sistemas de redes de computadores do Grupo de Redes de Computadores da UFPB se ampliaram no sentido de desenvolver ferramentas inteligentes para a solução de modelos de redes de computadores. Neste contexto, o foco das pesquisas tem sido a construção de novos ambientes de simulação (SAVAD [Souto, 93], SIMILE [Dias, 92]) e de bibliotecas de classes (ATMLib [Almeida, 99]) que ajudam no desenvolvimento de simuladores de alto nível.

O desenvolvimento de software vem se tornando cada vez mais complexo, dispendioso e lento, exigindo novas abordagens de programação [Krajnc, 97]. Uma das mais visadas abordagens, atualmente, é a utilização de componentes para o desenvolvimento de aplicações de forma rápida e de boa qualidade, a um menor custo. As aplicações são construídas conectando um ou mais componentes de forma visual, sem necessitar escrever

código fonte, apenas reutilizando código. Um estudo sobre componentes se encontra nesta Dissertação.

A experiência comprova que durante o processo de desenvolvimento normal de qualquer produto de software, as fases de análise (descrição do problema) e de projeto (descrição da solução) são as mais importantes e consomem a maior parte do tempo [Landin & Niklasson, 98]. Para reduzir efetivamente o tempo gasto durante o desenvolvimento de uma aplicação, é preciso fornecer não só reutilização de código, mas também a fase de **análise** e de **projeto**, de modo a reutilizar o conhecimento empregado e as principais decisões tomadas durante estas etapas [Freire, 00].

1.1 Objetivos

1.1.1 Objetivos Gerais

O objetivo desse trabalho é apresentar uma especificação de componentes que permita a construção facilitada de ferramentas de simulação de alto nível, através da reusabilidade de software. Os componentes devem fornecer o funcionamento básico de uma simulação orientada a eventos, tais como controle do relógio simulado, geradores de valores aleatórios e escalonamento de eventos, como veremos em detalhes nos próximos capítulos.

Esse ambiente de simulação deve fornecer uma interação facilitada com os elementos básicos de uma rede TCP/IP de longa distância, tais como *hosts*, roteadores e enlaces, especificados em [Wagner, 00], e deve prover um meio de coletar medidas de desempenho relevantes a tais redes e estatísticas sobre essas medidas.

1.1.2 Objetivos Específicos

- ✓ Estudar a técnica de Simulação Digital.
- ✓ Estudar as ferramentas de software que apresentem facilidades para a especificação de componentes de software;
- ✓ Levantar requisitos funcionais e não funcionais inerentes ao processo de simulação orientada a eventos;

- ✓ Propor uma especificação de componentes de software, que simule o funcionamento básico de um ambiente de simulação, de acordo com os requisitos levantados, utilizando a linguagem de modelagem UML [Rumbaugh et al., 99].
- ✓ Validar os componentes especificados de acordo com os requisitos levantados anteriormente.

1.2 Relevância

O interesse dos desenvolvedores de software pela utilização de componentes pré-definidos vem crescendo bastante nos últimos tempos, pois eles viabilizam a reutilização de componentes executáveis.

A reutilização de software vem sendo um fator primordial, nesses últimos anos, no desenvolvimento de software visando menor preço e eficiência. Existem várias formas para reutilização de software, entre eles *frameworks* [Landin & Niklasson, 98] e componentes [Szyperski, 99]. Componentes são objetos inteiros já instanciados que podem ser conectados a uma aplicação qualquer. O uso de componentes facilita o desenvolvimento de sistemas pois as aplicações são construídas visualmente através da junção dos componentes já fabricados, com a ajuda de alguma ferramenta gráfica.

Simuladores como o Arena [Kelton, 98] e o BONeS Designer [Alta Group, 96] usam o conceito de componentes, mas a utilização dos mesmos se restringe ao ambiente do simulador em questão. Entretanto, iniciativas isoladas de desenvolvimento de novos simuladores têm gerado um grande *overhead* no desenvolvimento e, consequentemente, na pesquisa como um todo [Almeida, 99]. A utilização da tecnologia de componentes facilita a construção de novos simuladores, específicos ou não, através da junção visual desses componentes.

Esta Dissertação contribui para facilitar o desenvolvimento de simuladores de redes de computadores, tornando-o mais eficiente e econômico, através da reutilização de componentes essenciais na construção de um simulador. Os conceitos envolvidos nesse trabalho se baseiam na utilização de técnicas de orientação a objetos e de componentes. Toda a especificação fornecida servirá de base para implementações futuras.

Devido aos componentes propostos serem considerados essenciais a uma simulação, estes podem ser reutilizados na construção de qualquer ferramenta de simulação orientada a eventos.

É importante ressaltar a multidisciplinaridade desta proposta, uma vez que a mesma engloba várias áreas, tais como modelagem e simulação, engenharia de software e redes de computadores.

1.3 Organização da Dissertação

Esta dissertação está organizada em 6 capítulos, da seguinte forma:

No capítulo 2 são introduzidos os principais conceitos de Simulação Digital e a sua aplicabilidade em estudos de avaliação de desempenho de sistemas discretos. Estes conceitos são necessários ao entendimento e o desenvolvimento desta Dissertação.

No capítulo 3 é mostrada uma introdução ao conceito de componentes, suas principais características e seu desenvolvimento de acordo com a arquitetura *JavaBeans*.

No capítulo 4 é apresentada a especificação dos componentes que fornecem o funcionamento básico de um simulador orientado a eventos. Foram levantados seus requisitos e identificadas suas características no sentido de atender aos requisitos levantados. Essa especificação foi feita utilizando a Linguagem de Modelagem Unificada (UML), seguindo um processo de desenvolvimento apresentado em [Larman, 98], enfocando a fase de Planejamento e Elaboração.

No capítulo 5 é apresentado o projeto mais detalhado dos componentes do sistema, fornecendo aspectos de implementação que serão úteis para o desenvolvedor das ferramentas de simulação. Neste capítulo, a validação da especificação é feita verificando se os requisitos levantados foram atendidos.

No capítulo 6 são apresentadas as considerações finais sobre o trabalho desenvolvido nesta Dissertação, assim como sugestões de continuidade da mesma.

Capítulo 2

2 Simulação Digital

Neste capítulo são introduzidos os principais conceitos de Simulação Digital e a sua aplicabilidade em estudos de avaliação de desempenho de sistemas discretos, necessários ao entendimento e ao desenvolvimento desta Dissertação.

A seção 2.1 apresenta uma introdução à técnica de Simulação Digital. Na seção 2.2 são apresentadas definições básicas sobre modelagem e a classificação dos modelos. Na seção 2.3 é descrito um processo de simulação. Na seção 2.4 é apresentada a classificação de simuladores. Na seção 2.5 são apresentados conceitos sobre simulação discreta orientado a eventos, assim como os elementos que a compõem e uma metodologia orientada a objetos para a descrição dos modelos.

2.1 Introdução

Simulação é o processo de construir um modelo de um sistema real ou imaginário e realizar experimentos com o propósito de entender o comportamento do sistema e avaliar estratégias para a sua operação [Smith, 00]. Um modelo é uma abstração de um sistema e pode ser construído para qualquer tipo de sistema, como, por exemplo, redes de computadores, circuitos integrados e sistemas de manufatura.

Os modelos podem ser solucionados analiticamente ou utilizando técnicas aproximadas. As técnicas analíticas se baseiam geralmente na Teoria das Filas [Kleinrock,

75]. Porém, apesar da solução analítica ser mais econômica e eficiente, ela pode se tornar inviável para sistemas mais complexos.

Para solucionar modelos em geral, uma alternativa de solução bastante utilizada, dentre as técnicas aproximadas, é a Simulação Digital [Kleinrock, 75]. O uso da Simulação Digital baseia-se na idéia de que uma abordagem experimental pode ser útil no suporte à tomada de decisões. Do ponto de vista prático, a Simulação Digital constitui-se no projeto e construção de modelos computadorizados de sistemas reais ou propostos, com o propósito de realizar, sobre eles, experimentos numéricos que visam uma melhor compreensão do seu comportamento em um dado conjunto de condições específicas [Kelton, 98].

A simulação do modelo em um computador resulta na obtenção das medidas de desempenho de interesse em função dos parâmetros de entrada e do conjunto de valores fornecidos. Os resultados obtidos numa simulação irão depender da qualidade do modelo construído e dos dados a serem aplicados ao modelo. Bons modelos e dados coerentes levam a resultados satisfatórios, e vice-versa [Kelton, 98].

2.1.1 Histórico

Um dos pioneiros do conceito de simulação foi John von Neumann. Em 1940, ele concebeu a idéia de executar múltiplas repetições de um modelo, obtendo dados estatísticos, e derivando comportamentos do sistema real baseado nesse modelo. Isso foi conhecido como o método de Monte Carlo por causa do uso da geração de variáveis randômicas para representar os comportamentos estatisticamente [Smith, 00].

O uso da simulação, como a conhecemos nos dias de hoje, começou na década de 70 quando ela se tornou uma ferramenta de escolha para muitas empresas, mais notadamente, nas indústrias automotivas e de ferro, como forma de evitar desastres e apontar falhas.

Com o surgimento dos computadores pessoais, a simulação invadiu também o mundo dos negócios, mas ainda não era muito utilizada e raramente era usada em empresas de pequeno porte. Com os computadores mais rápidos, mais recursos para animações, facilidades de uso e de integração com outros pacotes de software, contribuíram para a simulação se tornar uma ferramenta padrão em uma maior variedade de empresas, sendo agora empregada tanto na fase de projeto quanto para realizar mudanças em sistemas [Kelton, 98].

A maior causa de impedimento para que a simulação se torne uma ferramenta aceita e utilizada universalmente é o tempo gasto e a falta de habilidade no desenvolvimento

de modelos [Kelton, 98]. No intuito de resolver esse impasse, os simuladores estão se tornando cada vez mais específicos, utilizando terminologias específicas de vários ambientes de trabalho diferentes. Hoje já existe no mercado esse tipo de produto em áreas de aplicação como comunicação, semicondutores, central de chamadas e reengenharia de empresas.

2.2 Modelagem e Simulação

Para que um sistema posse ser estudado utilizando a Simulação Digital, faz-se necessário que seja construído um modelo. Modelar é uma arte, não uma ciência. Modelar consiste em ter um conhecimento profundo do sistema e saber extrair o essencial, sem incluir detalhes desnecessários [Soares, 90]. A definição do nível de detalhamento do modelo deve ser baseado no propósito para o qual ele foi construído. Já que o propósito do estudo vai determinar a natureza da informação que é coletada, não existe um único modelo para um dado sistema [Geoffrey, 78].

A construção de modelos pode ser útil para tomadas de decisão durante o projeto de um sistema, pois pode resultar na descoberta, e, consequentemente, na correção antecipada de falhas no projeto [Almeida, 99].

2.2.1 Classificação de modelos

Existem várias formas de classificar modelos de simulação.

Um modelo pode ser **estático** ou **dinâmico**, dependendo da importância do fator tempo no modelo. Um modelo estático é aquele no qual as mudanças de estado não envolvem tempo [Almeida, 99]. A maioria dos sistemas em operação ou de interesse de estudos é dinâmica [Soares, 90]. Exemplos são sistemas de redes de computadores e sistemas de manufatura, onde o tempo para a mudança de estados é fator bastante relevante.

Os modelos também podem ser classificados como **determinístico** ou **estocástico**, dependendo do tipo de entradas no modelo. Um modelo determinístico é aquele no qual uma entrada válida do sistema leva a exatamente uma mesma saída [Almeida, 99]. São modelos que não contêm variáveis aleatórias.

Modelos com entradas randômicas são considerados modelos estocásticos, ou seja, uma entrada pode levar a mais de uma saída. Um exemplo de um modelo estocástico é um sistema de agência bancária, onde a chegada dos clientes e o tempo de serviço nos caixas são

aleatórios. Modelos estocásticos são mais complexos devido aos eventos aleatórios, e, portanto, são os que merecem maior atenção.

Um modelo também pode ser **contínuo** ou **discreto**. Num modelo contínuo, as mudanças de estado do sistema ocorrem de forma contínua no tempo [Soares, 90]. Num modelo discreto, as mudanças de estado ocorrem em instantes específicos do tempo referidos como *tempo simulado* [Soares, 90].

2.3 Processo de Simulação

Em um estudo de simulação deve-se definir adequadamente a metodologia a ser utilizada na sua realização. Uma metodologia para um processo de simulação pode ser encontrada em [Soares, 90]. Apesar desse processo ser apresentado em diversos estágios, ele é iterativo (tem várias iterações no tempo) e não seqüencial. Esses estágios são mostrados a seguir:

Formulação do problema: consiste na definição clara do problema a resolver e dos objetivos da análise, importante para a futura análise de desempenho.

Construção do modelo: consiste na descrição estática e dinâmica do modelo. A descrição estática consiste em definir os elementos do sistema e suas características. A descrição dinâmica consiste em definir o modo como esses elementos interagem causando mudanças no estado do sistema no decorrer do tempo. Na construção de modelos, uma das tarefas mais difíceis é a decisão sobre quais elementos do sistema devem ser incluídos no modelo [Soares, 90] e qual o relacionamento entre eles. Para a construção de qualquer modelo, é necessário que se escolha uma ferramenta de simulação adequada.

Determinação dos dados de entrada e saída: a fase de formulação do problema gera requisitos dos dados de entrada. Esses dados podem ser hipotéticos ou baseados em alguma análise preliminar. A sensibilidade dos resultados da simulação pode ser avaliada pela realização de uma série de repetições da simulação, variando os dados de entrada.

Verificação: consiste em definir se o modelo executa conforme esperado. O processo de verificação consiste em isolar e corrigir erros não intencionais no modelo e sua complexidade depende do tamanho do modelo. A verificação geralmente é realizada por uma análise através de cálculos manuais.

Validação: consiste em definir se o modelo é uma representação razoável do sistema real [Horn, 71], quanto ao comportamento e quanto aos resultados obtidos. A validação pode ser realizada usando o teste de *razoabilidade* [Soares, 90] ou através da comparação com outros modelos do sistema já validados, por exemplo, modelos analíticos.

Execução do Modelo: consiste na simulação do modelo usando a ferramenta de simulação escolhida.

Análise e Apresentação dos Resultados: consiste na interpretação das saídas e da apresentação dos resultados às pessoas interessadas na área em questão. Segundo [Kelton, 98], a apresentação dos resultados deve:

- ✓ Ser breve e simples para um entendimento fácil;
- ✓ enfocar a apresentação dos resultados significativos alcançados;
- ✓ direcionar a linguagem de apresentação à audiência, lembrando-se que muitos deles não participaram do projeto, e
- ✓ apresentar razões para os resultados expostos, senão a apresentação perderá o seu embasamento.

Nenhum projeto de simulação pode ser considerado completo até que os seus resultados sejam utilizados [Soares, 90].

2.4 Classificação de Simuladores

Um simulador pode ser classificado quanto aos modelos que utiliza (*discretos* ou *contínuos*) ou quanto à forma em que sua execução é realizada (*orientado a eventos* ou *a processos*) [Almeida, 99].

Em um simulador orientado a processos, sua execução é realizada através da seqüência de processos, onde cada processo manipula um conjunto de eventos do mesmo tipo. A desvantagem é que os eventos não são tratados na mesma seqüência do sistema real, além de acrescentar complexidade ao controle dos instantes de ocorrência dos eventos e distribuí-la entre as entidades do modelo, uma vez que não há entidades dedicadas ao controle da simulação [Almeida, 99].

Num simulador orientado a eventos, sua execução é realizada através da seqüência dos eventos que ocorrem no sistema. Cada evento é associado a tarefas que são acionadas nas

entidades envolvidas e a instantes de tempo em que os mesmo ocorrem. Em outras palavras, neste tipo de simulador, um sistema é modelado pela definição das possíveis mudanças de estado que ocorrem no instante de cada evento e sua execução é produzida pela execução lógica associada a cada evento em uma seqüência ordenada do tempo [Soares, 90].

Uma das vantagens encontradas para essa abordagem é a possibilidade de conhecer o estado de qualquer entidade do sistema em qualquer instante do tempo. Por outro lado, esta possibilidade torna-se bastante complexa quando o sistema sendo modelado envolve muitas entidades e muitos tipos de eventos, cada um com várias ocorrências [Almeida, 99].

2.5 Simulação Discreta Orientada a Eventos

Uma simulação discreta apresenta eventos, entidades e estados. Os eventos são perturbações instantâneas que mudam o estado do sistema. Entidades são os objetos dinâmicos representados na simulação, descritos por seus atributos e para os quais os eventos ocorrem. O estado de um sistema é definido em termos de valores numéricos dados às variáveis e aos atributos das entidades, em um tempo específico.

O comportamento dinâmico da simulação é obtido pelo processamento seqüencial dos eventos e pela coleta de valores nos tempos de evento [Soares, 90], com o objetivo de conhecer algo sobre o comportamento e desempenho do sistema.

2.5.1.1 Eventos

Eventos podem ser a chegada de um cliente em um banco ou a transmissão de uma mensagem em uma rede de computadores. Como o evento é uma perturbação instantânea, o tempo não avança durante a execução de um evento. Para cada evento, existem um tempo simulado e uma lógica associados a ele. Os eventos são itens chaves pois especificam a lógica que controla as mudanças de estados do modelo que ocorrem em tempos específicos do tempo.

2.5.1.2 Escalonadores de Eventos

Os eventos são geralmente armazenados e gerenciados pelo uso de listas ou filas, mais freqüentemente chamadas de *Calendário de Eventos*. Essas listas identificam quais eventos estão prontos para serem processados, quais estão à espera do avanço do tempo, ou

quais precisam ser escalonados conforme condições específicas. Os eventos são geralmente armazenados em ordem cronológica.

2.5.1.3 Relógio

O tempo da simulação é controlado por um *relógio*. Diferentemente do tempo real, o tempo simulado não flui continuamente, ele sempre avança para o instante de ocorrência do evento iminente. O tempo simulado é descrito através de uma unidade de tempo de serviço (*uts*), que representa uma certa quantidade de tempo real [Almeida, 99], como por exemplo, dois segundos, 10 minutos ou 20 dias. Dessa forma, o relógio é adiantado em valores múltiplos daquele definido para a *uts*, que vai depender do problema a ser resolvido. Isto permite que a simulação de uma determinada situação, que levaria dias para ocorrer no sistema, possa ser efetuada em segundos ou minutos. No começo de cada simulação, o relógio é geralmente configurado para zero e a lista de eventos é inicializada.

2.5.1.4 Entidades

As entidades do sistema podem ser os servidores ou os clientes. Um cliente é uma entidade que se move através do modelo e é caracterizado por um conjunto de atributos. Um cliente pode ser modelado diferentemente de acordo com o sistema a ser analisado. Por exemplo, em uma rede de computadores, os clientes podem ser os pacotes que trafegam pela rede, e seus atributos poderiam ser o tempo de criação, tamanho e a prioridade. Um cliente entra no sistema em busca de um serviço e, eventualmente, parte após recebê-lo ou desiste do mesmo.

O serviço procurado pelo cliente é oferecido pelos servidores. Servidores são entidades responsáveis por atender os cliente em suas demandas de serviço [Giozza et al., 86]. Quando um cliente chega em busca de um serviço e o servidor está disponível, o cliente o ocupa durante um determinado tempo aleatório e o disponibiliza quando esse tempo termina. Caso o servidor esteja ocupado quando o cliente chega, este último é encaminhado para uma fila (caso o servidor o tenha). A ordem de atendimento dos clientes nas filas depende da disciplina de escalonamento que o servidor utiliza.

As disciplinas de escalonamento das filas mais utilizadas em sistemas de redes de filas são [Almeida, 99]:

- √ FCFS (*First Come, First Served*): o primeiro que chega à fila é o primeiro a ser atendido;
- √ LCFS (*Last Come, First Served*): o último que chega à fila é o primeiro a ser atendido;
- √ RR (*Round-and-Robin*): cada um dos clientes recebe uma parcela de serviço até que se complete sua demanda total. No caso de haver mais de uma fila, cada cliente que ocupava o primeiro lugar de sua fila recebe uma parcela de serviço até que se atinja o total desejado e parta do servidor;
- √ SPT (*Shortest Processing Time First*): o cliente que demandar menor quantidade de processamento do servidor é o que será atendido;
- √ HPR (*Higher Priority First*): o cliente, ou a fila quando houver mais de uma, que tiver maior prioridade é o que será atendido.

2.5.2 Medidas de Desempenho

As medidas de desempenho servem para expressar o comportamento de um sistema de forma quantitativa e qualitativa mediante um determinado conjunto de entradas [Almeida, 99]. Essas medidas podem variar de um sistema para outro e de acordo com o objetivo do estudo.

No entanto, algumas dessas medidas são comuns a vários sistemas, como por exemplo [Giozza et al., 86.], tempo médio de espera em fila e tempo médio de atendimento em um dado servidor.

Existem tipos de estatísticas que são coletadas durante uma simulação:

- contadores (número de clientes no sistema);
- medidas sumárias (valores extremos, valores médios);
- utilização (porcentagem de tempo de utilização de uma entidade);
- ocupação (fração de tempo que um número de entidades está ocupado);
- distribuições de variáveis importantes (tamanho em fila, tempo de espera);
- tempo de transição (tempo que um cliente sai de uma entidade do sistema para a entrada de outra entidade).

2.5.3 Geração de Números e Variáveis Aleatórias

Muitos dos sistemas que são estudados utilizando a técnica da simulação têm características de aleatoriedade [Giozza et al., 86]. A razão dessa aleatoriedade ocorre basicamente em função da limitação de recursos do sistema e da ocorrência aleatória dos eventos (*tempo de interchegada de clientes* e o *do tempo de serviço* são variáveis aleatórias do sistemas). Para simular essa aleatoriedade, o simulador deve produzir valores aleatórios para cada variável aleatória do modelo. Em diversas situações, faz-se necessário que essas variáveis aleatórias sejam caracterizadas através de funções de distribuição de probabilidade apropriadas [Giozza et al., 86].

Os valores para as variáveis aleatórias podem ser gerados utilizando dois processos consecutivos: primeiramente, a geração de números aleatórios, e, depois, a geração de valores aleatórios.

A geração de números aleatórios consiste em gerar um conjunto de números equiprováveis (mesma probabilidade de ocorrência). Esses números são gerados a partir de um número inicial, chamado de semente. Esses números são chamados de números pseudo-aleatórios.

Após a geração dos números aleatórios, faz-se necessário convertê-los em um valor aleatório de acordo com uma distribuição desejada. Para isso, existem vários métodos, entre eles o método da *transformação inversa* e o método da *aproximação retangular* [Soares, 90].

2.5.4 Simulação de modelos

Na simulação de modelos pode-se utilizar linguagens de programação de propósito geral ou específico, ou ambientes de simulação de alto nível.

Inicialmente, para simular modelos, foram utilizadas linguagens de programação estruturada de propósito geral, tais como FORTRAN, Pascal ou C. Elas permitem uma grande flexibilidade na construção dos modelos (em termos de tipos de modelos e manipulações possíveis) [Kelton, 98]. Porém o trabalho na simulação do modelo se torna mais complexo e mais dispendioso pois os desenvolvedores terão que lidar com aspectos básicos de simulação (relógio, gerador de números aleatórios, etc.), podendo dar margem a um variado número de erros.

Posteriormente, foram desenvolvidas linguagens estruturadas voltadas especificamente para a simulação, tais como GPSS, SIMScript e SLAM [Kelton, 98] e foram largamente utilizadas.

Com o advento da abordagem orientada a objetos, surgiram linguagens voltadas para os mais diversos tipos de sistemas: gerenciamento de banco de dados, editores de textos e sistemas operacionais [Almeida, 99]. Muitas dessas linguagens, no entanto, foram desenvolvidas a partir de outras já existentes, tais como o C++ (derivado do C), o Java (derivado do C/C++) ou o Delphi (derivado do Pascal).

Assim como algumas linguagens de programação de propósito geral foram estendidas, surgiram também as linguagens de programação orientadas a objetos voltadas para a construção de modelos de simulação (*Object-Oriented Simulation* - OOS) [Roberts et al., 98]. Tais linguagens oferecem facilidades adicionais, entre elas, maior simplicidade pois são utilizadas entidades que são naturais ao sistema físico. O SimJava [McNab, 99] e o Sim++ [C++Sim, 99] são exemplos de bibliotecas de classes, referentes às linguagens Java e C++, respectivamente, que oferecem um conjunto de rotinas que facilitam a construção dos modelos.

Ao longo do tempo, surgiram os simuladores de alto nível, tais como o BONeS [Alta Group, 96] e o Arena [Kelton, 98], que oferecem facilidades na construção dos modelos por propiciarem a representação gráfica dos mesmos, e por fazerem uso de interfaces gráficas amigáveis, tais como menus, caixas de diálogos, etc. Porém, muitos simuladores são restritos a domínios específicos (como manufatura ou comunicação), e, geralmente, não permitem grande flexibilidade no desenvolvimento dos modelos.

O escopo desta Dissertação refere-se à especificação de componentes para a construção de ferramentas de simulação específicas voltadas para sistemas de redes de computadores. No entanto, esses componentes podem também ser reutilizados na construção de ferramentas de simulação de propósito geral.

Capítulo 3

3 Componentes de Software

Este capítulo mostra uma introdução ao conceito de componentes, suas principais características e seu desenvolvimento de acordo com a arquitetura *JavaBeans*.

Ele é dividido em quatro seções. A seção 3.1 apresenta uma introdução a componentes. Na seção 3.2 são mostrados os principais problemas da solução orientada a objetos para o desenvolvimento de sistemas mais complexos. A seção 3.3 descreve o que são os componentes e suas principais características, assim como seu impacto no processo de desenvolvimento de software. A seção 3.4 é voltada para o desenvolvimento de componentes. Ainda nesta seção é mostrada a arquitetura *JavaBeans*.

3.1 Introdução

Os desenvolvedores de software estão constantemente em busca de novas formas de construir aplicações em menos tempo com um menor custo sem afetar muito sua qualidade e eficiência [Englander, 97]. A tecnologia de componentes é reconhecida como um caminho freqüentemente apontado para atender esses requisitos.

A reutilização através de componentes surgiu somente recentemente na área da Engenharia de Software. Mas há muito tempo que a idéia de maximizar a reutilização de artefatos prontos é utilizada em outras áreas. Sistemas eletrônicos e produtos manufaturados, por exemplo, são construídos com base em componentes pré-fabricados que podem ser

prontamente interconectados. Um conjunto bem escolhido de componentes pode fabricar produtos finais de forma mais rápida e mais confiável.

Mas será que a tecnologia de orientação a objetos não é suficiente para desenvolver software rápido e de boa qualidade?

3.2 Solução Orientada a Objetos

Desenvolver software reutilizável não é uma tarefa fácil. Com o advento da tecnologia orientada a objetos, pensou-se a princípio ser essa a melhor solução para o reuso. Entretanto, muitos projetos utilizando esta tecnologia têm fracassado ultimamente: os benefícios da linguagem adotada dependem do uso correto da tecnologia. Alguns dos problemas levantados são [Krajnc, 97]:

- Orientação a objetos (sozinha) não produz software reutilizável automaticamente;
- Orientação a objetos (sozinha) não tem boa escalabilidade;
- Orientação a objetos (sozinha) não provê boa encapsulação (esconder informação).

Em sistemas complexos o mal uso da tecnologia orientada a objetos pode levar a uma armadilha chamada geralmente de *Hyperspaghetti Objects*, representada na figura 3.1. Esse problema ocorre quando o sistema começar a crescer demasiadamente, sem disciplina.

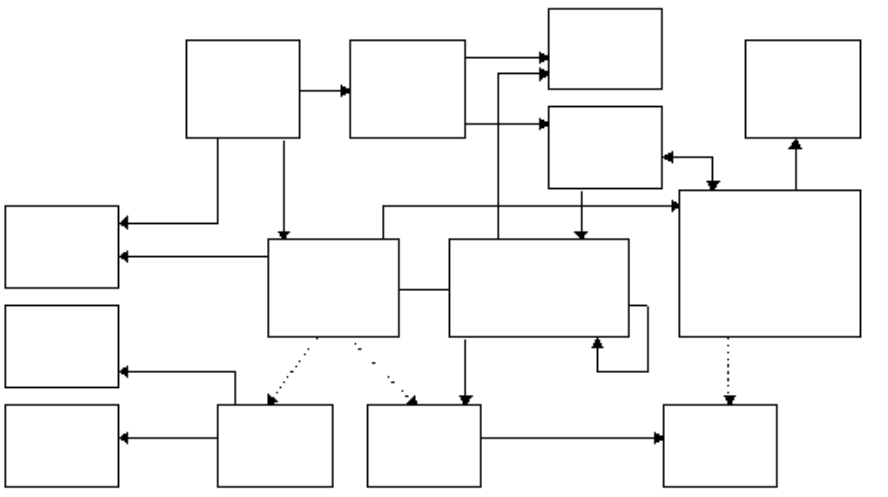

Figura 3-1: Hyperspaghetti Objects

Um dos problemas encontrados é que qualquer objeto pode referenciar qualquer outro, dificultando o reuso de um objeto isolado. Outro problema é a dificuldade de manutenção (adição de novas funcionalidades e remoção de *bugs*), causando instabilidade no sistema.

Uma solução adotada para resolver esses problemas é agrupar alguns objetos em **componentes** e isolá-los usando **interfaces**. Componentes de software representam um importante passo no sentido de sistematizar a produção de software, ao prover reusabilidade num alto nível de abstração e, inclusive, através da utilização apropriada de técnicas de orientação a objetos [Freire, 00].

3.3 O que são Componentes ?

Existem muitas definições para componente. Uma definição geral de componente foi dada por D'Souza [D´Souza & Wills, 99] : "componente é um pacote coerente de artefatos de software que pode ser desenvolvido independentemente e entregue como unidade e que pode ser composto, sem mudança, com outros componentes para construir algo maior"

Usando essa definição, um componente pode incluir código executável, código fonte, projetos (designs), especificações, testes, documentação, etc. Em termos de implementação, segundo D'Souza, um componente "é um pacote coerente de implementação de software que:

- pode ser desenvolvido independentemente e entregue como unidade;
- tem interfaces explícitas e bem definidas para os serviços que oferece;
- tem interfaces explícitas e bem definidas para os serviços que requer; e
- pode ser composto com outros componentes, talvez após a customização de algumas propriedades, mas sem modificar os componentes em si."

De um ponto de vista prático, componentes são objetos inteiros já instanciados que podem ser conectados a uma aplicação qualquer. Comparado às bibliotecas de classes e outros artefatos de software, os componentes se diferenciam por:

- ✓ Permitirem a construção de aplicações por montagem de pedaços já existentes. A aplicação é construída graficamente com a ajuda de uma ferramenta visual.

- ✓ Explicitarem suas interfaces. Para ser conectados a uma aplicação, um componente deve ter suas interfaces padronizadas dos serviços que requer e dos serviços que oferece.
- ✓ Serem unidades de empacotamento, de entrega, implantação e carga. Um componente engloba especificação, implementação, imagens, etc. Ele é vendido e implantado integralmente e é carregado por inteiro numa aplicação final.

3.3.1 Interfaces

As interfaces de um componente definem os pontos de acesso ao componente. Esses pontos permitem aos clientes, ou a outros componentes, acessar os serviços oferecidos pelo componente. Um componente pode conter diversas interfaces, sendo cada interface correspondente a diferentes serviços de acordo com as necessidades do cliente. A especificação da interface é um contrato feito entre os clientes e os desenvolvedores do componente. Esse contrato pode ser estabelecido através da definição das pré- e pós-condições por ambas as partes.

Tecnicamente, uma interface é um conjunto de operações que são chamadas pelos clientes [Szyperski, 99]. A semântica de cada operação é especificada e serve tanto para a pessoa que implementa a interface quanto para o cliente que usa a interface.

Assim como as interfaces explicitam os serviços que um componente oferece, deve haver também interfaces bem definidas para as necessidades que um componente requer.

3.3.2 Componentes e Objetos

Um componente normalmente consiste de um ou mais objetos. Mas um componente não necessariamente contém somente objetos, e pode até não conter nenhum. Se o componente obedece às suas interfaces, ele pode ser implementado usando qualquer paradigma de programação, seja ele funcional, utlizando a liguagem *assembly*, ou qualquer outro paradigma [Szyperski, 99]. Os objetos são somente usados com linguagens orientadas a objetos.

Componentes são unidades de implantação e objetos são unidades de instanciação [Szyperski, 99]. O componente deve ser independente de plataforma e de outros componentes. Um componente, portanto, encapsula suas características, e nunca pode ser implantado

parcialmente. O objeto também não pode ser instanciado parcialmente. Cada objeto tem sua identidade única que é suficiente para identificá-lo durante toda sua vida.

3.3.3 Processo de Desenvolvimento

Na definição sobre componentes feita na Conferência Européia sobre Programação Orientada a Objetos (ECOOP – *European Conference on Object-Oriented Programming*) em 1996, um componente “está sujeito à composição por terceiros” [Szyperski, 99]. Portanto, percebe-se que o uso de componentes traz mudanças no processo de desenvolvimento de um software, com o surgimento de novas fases.

Além das etapas básicas de qualquer processo de desenvolvimento (levantamento de requisitos, análise do domínio do problema, projeto da solução, implementação e testes), aparecem o "*assembly time*", para construção das aplicações, já que a composição é feita em etapa separada da construção dos componentes, e o "*deployment time*", para configurar os componentes no ambiente final, que consiste em modificações nos seus atributos. A etapa de composição (*assembly time*) passa a ser a fase principal na construção de aplicações, em vez da implementação. Novas ferramentas visuais se tornam necessárias para essas novas etapas. A composição das aplicações é feita utilizando ferramentas gráficas que permitem a configuração e conexão dos componentes necessários à aplicação de forma visual.

Além do surgimento de novas etapas, surgem também novos papéis no processo de desenvolvimento:

- ✓ *Coordenador*: quem decide quais componentes deve-se comprar ou desenvolver.
- ✓ *Desenvolvedor*: quem desenvolve os componentes.
- ✓ *Montador*: quem monta as aplicações baseadas em componentes
- ✓ *Instalador*: quem instala e configura os componentes em ambientes finais.

Software baseado em componentes é extensível por definição, isto é, novos componentes podem ser acrescentados a qualquer momento. Se o componente obedece às suas interfaces, ele pode ser implementado em qualquer linguagem e rodar em qualquer plataforma.

3.4 Desenvolvimento de Componentes

É necessário um suporte adequado nas linguagens de programação para dar apoio ao conceito de componentes. Um dos aspectos mais importantes é o encapsulamento do componente inteiro. Mecanismos de segurança também são necessários para que componentes não possam afetar a integridade de outros (inclui integridade de memória). Finalmente, é necessário um mecanismo para definir interfaces.

3.4.1 Principais Padrões

As principais abordagens atuais que suportam bem o desenvolvimento de componentes são: o padrão CORBA (*Common Object Request Broker Architecture*) [Siegel, 96], desenvolvido pelo *Object Management Group* (OMG); o padrão COM (*Component Object Model*) [Rofail & Shohoud, 99], desenvolvido pela Microsoft; e padrão JavaBeans [Sun Microsystems, 99], desenvolvido pela Sun Java.

Todas essas abordagens suportam mecanismos de *late binding*, encapsulação, polimorfismo dinâmico e herança de interface [Szyperski, 99].

3.4.2 O Modelo de Componente

Os componentes precisam funcionar de acordo com um conjunto de regras e diretrizes. Eles precisam exibir comportamento e características esperados para participar de uma estrutura de componente e para interagir com o ambiente e outros componentes.

O modelo de componentes é composto de uma arquitetura e uma API (*Application Programming Interface*) [Englander, 97]. Juntos, esses elementos provêm uma estrutura na qual os componentes podem ser combinados para criar uma aplicação. Os componentes são providos de características necessárias para trabalhar em um ambiente, e exibir um comportamento que os identificam.

Os elementos do modelo de componentes são detalhados a seguir.

3.4.2.1 Descoberta e Registro

A descoberta de classes e interfaces é o mecanismo usado para utilizar um componente em tempo de execução e determinar as interfaces que suporta. O modelo de

componente deve também prover um processo de registro para que este possa ser conhecido, assim como suas interfaces.

O componente pode ser descoberto em tempo de execução. Composição dinâmica permite desenvolver aplicações e componentes independentemente, a dependência se limitando ao contrato feito por ambas as partes (interfaces do componente). Descoberta dinâmica também permite aos desenvolvedores atualizar componentes sem ter que reconstruir as aplicações.

O processo de descoberta pode também ser usado em tempo de composição (*assembly time*) em um ambiente. Importante para programação com ambientes visuais.

3.4.2.2 Criação e Escalonamento de Eventos

Um evento é algo importante que acontece em determinado ponto do tempo. Um evento pode acontecer devido a uma ação, como o clique do mouse, ou por outras maneiras. Componentes vão enviar notificações para outros objetos interessados quando um evento acontecer.

3.4.2.3 Persistência

Geralmente, todos os componentes têm estados. Portanto, os componentes devem ser capazes de participar de um mecanismo de persistência padrão à aplicação.

3.4.2.4 Representação Visual

O ambiente do componente permite que componentes individuais escolham os aspectos de sua representação visual. A maioria dessas características serão propriedades do componente. O *layout* é um aspecto importante da representação visual. Ele se relaciona com a disposição dos componentes na tela, como eles se relacionam com outros componentes, e o comportamento que eles exibem quando eles interagem com os outros.

3.4.2.5 Suporte de Programação Visual

Programação visual é uma parte chave do modelo de componente. Componentes são representados por caixas de ferramentas ou *palettes*. O usuário pode “arrastar” o *bean* para uma área central da ferramenta de composição e configurá-lo através de um editor de propriedades. Não é necessário escrever nenhuma linha de código.

Na figura 3.2 é mostrado um exemplo de uma ferramenta visual (o *BeanBox*). Essa ferramenta é usada na construção de *JavaBeans* (descritos na seção seguinte).

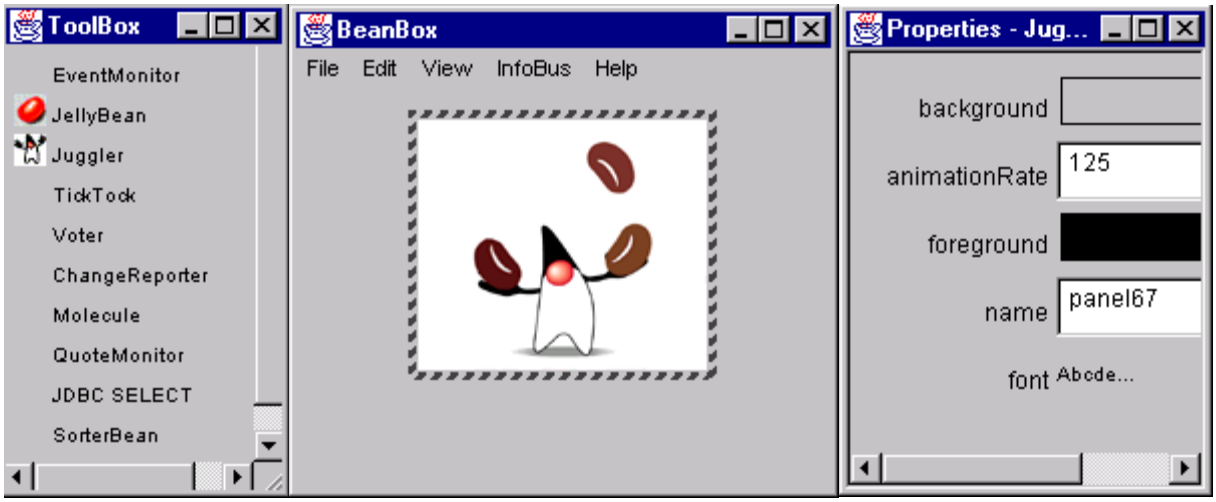

Figura 3-2: Editor Gráfico de Componentes (*BeanBox*)

O *BeanBox* permite testar a funcionalidade de *beans* (componentes Java), incluindo propriedades, eventos e serialização.

3.4.3 A arquitetura *JavaBeans*

JavaBeans é uma arquitetura que é utilizada para construir componentes em Java [Englander, 97]. Essa arquitetura suporta características de reusabilidade de software e orientação a objetos. Uma das mais importantes características de *JavaBeans* é que ela não modifica a linguagem Java. Em Java, *bean* significa componente. A API utilizada para a construção de componentes de software em Java chama-se `java.beans`.

Os principais aspectos de um modelo de um *bean* são [Szyperski, 99]:

- ✓ *Eventos*: os *beans* podem ser fontes ou consumidores potenciais de eventos. É através da troca de eventos que os *beans* se comunicam uns com os outros.A ferramenta de composição conecta consumidores a fontes.
- ✓ *Métodos*: serviços que o cliente pode utilizar.
- ✓ *Propriedades*: As propriedades podem ser modificadas em tempo de composição e representam os atributos de estado e de comportamento através dos quais *beans* podem ser configurados. As mudanças nas propriedades podem disparar eventos.

- ✓ *Introspecção*: um *bean* pode ser inspecionado pela ferramenta de composição para encontrar suas propriedades, eventos e métodos suportados.
- ✓ *Configuração*: usando a ferramenta de composição, um *bean* pode ser configurado modificando suas propriedades.
- ✓ *Persistência*: os *beans* configurados e conectados necessitam ser salvos para futuro carregamento em tempo de execução da aplicação.

3.4.3.1 Eventos

Os eventos são objetos criados por uma fonte de eventos (*event source*) e propagado para todos os consumidores de eventos (*event listeners*) cadastrados. Os consumidores se registram em fontes de eventos para receber notificações de eventos. Para se registrar em uma fonte, os consumidores usam métodos padronizados, cujas assinaturas são:

```
public void add<tipo>Listener (<tipo> Listener)
public void remove<tipo>Listener (<tipo> Listener)
```

O pacote `java.util` provê o suporte básico para o modelo de eventos dos *beans*. A comunicação baseada em eventos geralmente é *multicast*.

3.4.3.2 Propriedades

O *bean* pode definir um número abitrário de propriedades. Uma propriedade é um atributo ou característica do *bean* que pode afetar sua aparência ou comportamento. Propriedades são referenciadas por seu nome e pode ter qualquer tipo, incluindo tipos primitivos, tal como *int*, e tipos de classes ou interfaces, tal como `java.awt.Color`. Propriedades são geralmente parte do estado persistente de um objeto.

As propriedades de um *bean* são acessadas através de métodos padronizados para ler e escrever. As propriedades podem ser para ler/escrever, somente para ler ou somente para escrever. Os métodos usados para acessar as propriedades seguem o padrão de projeto para propriedades [Englander, 97]. Suas assinaturas são:

```
public void get<NomePropriedade> (<tipo> valor);
public void set<NomePropriedade> (<tipo> valor);
```

3.4.3.3 Intropescção

Quando se usa uma ferramenta de desenvolvimento visual (exemplo *BeanBox*), ela deve expor as propriedades, métodos e eventos do *bean*, permitindo manipular sua aparência e comportamento. Para descobrir essas priopriedades e eventos, a ferramenta usa um mecanismo de introspecção, realizado através da API de reflexão Java, `java.lang.reflect`, que permite descobrir características de um *bean* via padrões de projeto.

Para que este mecanismo de introspecção funcione, programadores de *beans* devem seguir convenções de nomeação [Englander, 97]. Essa padronização é uma combinação de regras para a formação de assinatura de métodos, seu tipo de retorno e seu nome. Por exemplo, para as propriedades, deve existir os métodos de acesso *get* e *set*, descritos nesta seção.

Algumas vezes o mecanismo de reflexão força a utilização de padrões de projeto que não são necessários ou, então, deseja-se expor informações de *beans* que não podem ser representados pelos padrões de projeto. Nestes casos, o *bean* deve fornecer, explicitamente, informações sobre suas propriedades, métodos e eventos, implementando a interface `java.beans.BeanInfo`. Essa interface especifica um conjunto de métodos que podem ser usados para recuperar vários elementos de informação sobre o *bean*. Se essa interface for implementada, a ferramenta de composição vai usá-la para descobrir os métodos, propriedades e eventos do *bean*.

3.4.3.4 Persistência

A maioria dos componentes mantém informação que definem sua aparência e comportamento. Essa informação é conhecida como o estado do objeto. Algumas dessas informações são representadas pelas propriedades do objeto. Quando uma aplicação é executada, os componentes devem automaticamente exibir um comportamente pré-estabelecido. Portanto, a informação de estado de todos os componentes precisa ser salva em algum meio de armazenamento persistente, visto que ele pode ser usado para recriar todo o estado da aplicação em tempo de execução.

A arquitetura JavaBeans usa o mecanismo de serialização do Java para a persistência. Tudo que o *bean* deve fazer é implementar a interface `java.io.Serializable`.

Componentes Java compilados são empacotados em arquivos do tipo JAR. Os arquivos JAR podem conter um número abritrário de arquivos e podem prover compressão baseada no formato ZIP. Esses arquivos podem ser usados para empacotar arquivos de classes relacionados, *beans* serializados, e outros recursos necessários ao *bean*. Para que um *bean* seja utilizado através de uma ferramenta gráfica, então, ele deve ser empacotado num arquivo do tipo JAR juntamente com todas as classes e arquivos que ele requer [Freire, 00].

A API *JavaBeans* faz parte do JDK1.1 e qualquer ferramenta compatível com ele suporta implicitamente os conceitos e características envolvidos.

Capítulo 4

4 Especificação dos Componentes

Neste capítulo é iniciada a especificação dos componentes propostos nesta dissertação, apresentando as fases de levantamento de requisitos e de análise do processo de desenvolvimento adotado. Estas fases dizem respeito à especificação em alto nível dos componentes para uma ferramenta de simulação orientada a eventos. As fases subsequentes são apresentadas no próximo capítulo.

Este capítulo é composto de três seções. A seção 4.1 introduz o processo de desenvolvimento adotado nesta Dissertação para a construção dos componentes. A seção 4.2 descreve o sistema, seus usuários alvos, assim como os requisitos levantados. Ainda nesta seção última é apresentado um diagrama de *use-cases*. A seção 4.3 descreve a fase de análise juntamente com seus diagramas de conceitos e de seqüencia.

4.1 Introdução

A reusabilidade de software é reconhecida como o caminho mais freqüentemente apontado para aumentar a produtividade no processo de desenvolvimento de sistemas. Como pudemos ver no capítulo anterior, a programação usando somente a orientação a objetos não oferece um nível de abstração adequado para o desenvolvimento facilitado de novas ferramentas de simulação, exigindo dos programadores uma grande experiência de programação.

Neste capítulo é descrita uma especificação dos componentes de software que fazem parte de qualquer ambiente de simulação, apresentando uma solução para o desenvolvimento rápido e com um mínimo de programação possível.

O processo de desenvolvimento escolhido neste trabalho foi baseado em [Larman, 98], e segundo ele, as grandes fases de qualquer processo de desenvolvimento são Planejamento e Elaboração (análise e projeto), Construção do Sistema (codificação e testes) e Implantação (colocar em produção, treinar usuários, etc). Este trabalho se deterá somente nessa primeira fase, que inclui as etapas de levantamento de requisitos, análise do domínio do problema e projeto da solução. Para tal, foi utilizada a linguagem de modelagem UML (*Unified Modeling Language*) [Rumbaugh et al., 99].

A seguir são mostrados os requisitos básicos levantados para a construção de um simulador de redes TCP/IP, com base em componentes.

4.2 Requisitos do Sistema

Para propor uma solução viável para qualquer problema, é necessário que se defina bem esse problema. Em outras palavras, é necessário definir os requisitos para a solução. Os requisitos são uma descrição das necessidades do sistema. Eles especificam *que* sistema deve ser construído. O levantamento correto dos requisitos provém de uma boa interação entre o usuário e o desenvolvedor.

Os artefatos típicos a serem criados na fase de levantamento dos requisitos são:

- ✓ Breve descrição do sistema;
- ✓ Descrição dos usuários alvos;
- ✓ Descrição das metas do sistema;
- ✓ Descrição dos requisitos funcionais do sistema (o que o sistema deve fazer);
- ✓ Descrição dos requisitos não funcionais do sistema (atributos do sistema).

4.2.1 Descrição do sistema

O sistema proposto é a especificação de componentes de software que permita a construção facilitada de uma ferramenta de simulação de alto nível. Esse ambiente de simulação deve fornecer uma interação facilitada com os elementos básicos de uma rede

TCP/IP de longa distância, tais como *hosts*, roteadores e enlaces, especificados em [Wagner, 00], e deve prover um meio de coletar medidas de desempenho relevantes para tais redes e apresentar estatísticas sobre essas medidas.

Os componentes devem fornecer o funcionamento básico de uma simulação orientada a eventos, tais como adiantamento do relógio simulado e escalonamento de eventos, como visto no capítulo anterior.

A simulação realizada utilizando os componentes especificados neste trabalho se apresenta dividida em três partes: *inicialização*, *execução* e *finalização*. A etapa de inicialização constitui-se na construção e configuração do modelo e na configuração dos parâmetros de entrada da simulação (por exemplo, tempos de início e de finalização).

Depois da construção do modelo e inicialização da simulação, a execução da simulação é iniciada. Um algoritmo básico de execução de uma simulação pode ser visto na figura 4.1. Esse algoritmo consiste na repetição dos quatro passos descritos abaixo.

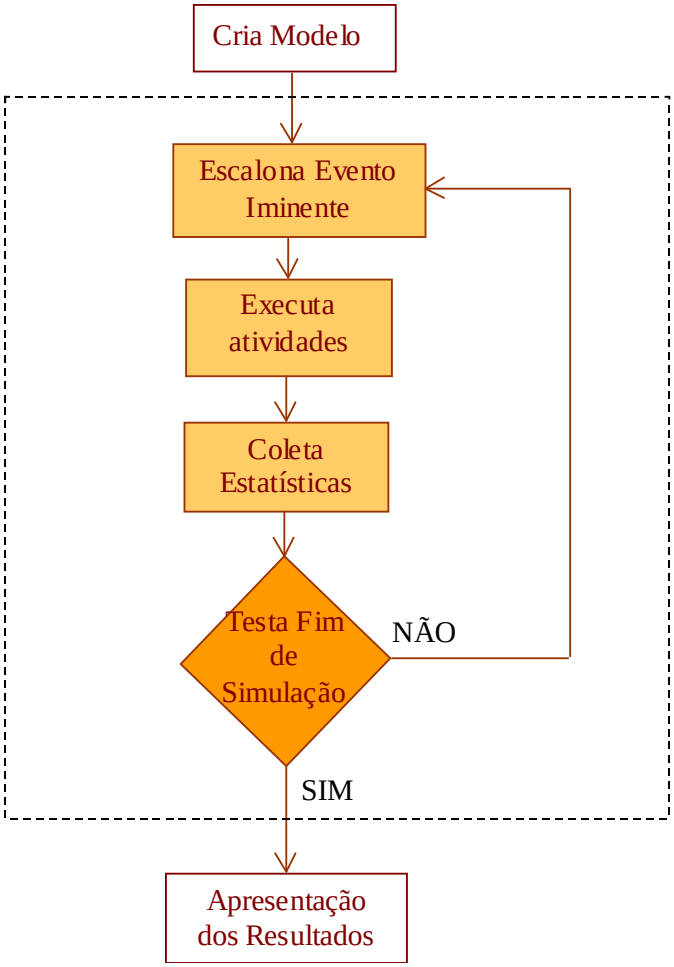

Figura 4-1: Algoritmo de Simulação

1. Escalonamento do evento iminente;
2. Execução das ações relacionadas ao evento, que dependem do tipo do evento e do estado do modelo no momento de sua ocorrência;
3. Coleta de dados para o cálculo de medidas de desempenho de interesse;
4. Teste se é fim de simulação. Caso não seja, o algoritmo é repetido.

A finalização constitui-se do cálculo das medidas de desempenho de interesse e da apresentação dos resultados.

Durante esse processo, eventos vão ser gerados, armazenados em uma lista de eventos, em ordem cronológica, e é sempre escalonado o evento que estiver no topo da lista. Após um evento ser escalonado, deve-se determinar o instante de ocorrência desse evento, atualizar o relógio simulado para esse instante e processar o evento.

Em uma simulação, deve-se identificar quais os eventos que podem ocorrer no sistema, avaliar seus efeitos no estado e atividades do sistema e permitir a sua ocorrência [Giozza et al., 86]. Os tipos de eventos identificados nesse sistema são:

Chegada de pacotes originados na fonte: chegada de pacotes nos *hosts* originados na fonte em tempos de geração obtidos de amostras de uma função de distribuição de probabilidades especificada pelo usuário. A geração desse tipo de evento segue a técnica de autogeração.

Chegada de pacotes: esse evento ocorre quando os pacotes são transmitidos através dos enlaces.

Término de serviço: fim do processamento dos pacotes nos *hosts* e roteadores. O tempo de serviço é uma amostra de uma Função de Distribuição de Probabilidade especificada pelo usuário.

Fim de Transmissão: fim da transmissão de um pacote em um enlace. A transmissão do pacote vai depender de seu tamanho e do tempo de transmissão de cada enlace.

Fim de simulação: fim da simulação de acordo com os parâmetros fornecidos pelo usuário.

Atualização da tabela de roteamento: atualização das tabelas de roteamento nos roteadores de acordo com os protocolos de roteamento escolhido.

Os estados mais importante encontrados, em função dos elementos de rede especificados em [Wagner, 00] são:

Roteador: o roteador pode estar livre (à espera de um pacote), ocupado (processando algum pacote), congestionado (quando ele tem as suas filas de entrada cheias) ou inativo (quando ele está fora do ar).

Host: o *host* pode estar livre (à espera de um pacote), ocupado (processando algum pacote), congestionado (quando ele tem todas as suas filas cheias) ou inativo (quando ele está fora do ar).

Enlace: o enlace pode estar livre (à espera de um pacote), ocupado (transmitindo algum pacote), inativo (quando o enlace estiver desativado) ou congestionado (com seu limite de capacidade alcançado, i.e., com sua fila de entrada no limite de capacidade).

Fonte: a fonte pode estar ativa, quando a fonte estiver gerando mensagens segundo uma Função de Distribuição de Probabilidade, ou inativa, quando ela está desativada.

Para melhor entendimento das relações entre os estados e os eventos do sistema, foi construído um diagrama de estado. O diagrama da figura 4.2 mostra os possíveis estados em que o roteador pode se encontrar durante uma simulação, de acordo com os eventos do sistema. Inicialmente, o roteador está **`livre`**, e somente mudando para ocupado (**`Processando Pacote`**) quando chega algum pacote em alguma de suas filas de entrada. Enquanto ele estiver ocupado, os próximos pacotes que chegam são armazenados em fila para serem processados futuramente. Quando a fila estiver cheia, pacotes serão descartados e o roteador passará para o estado **`congestionado`** só saindo desse estado quando a fila de entrada puder receber pacotes. Quando ocorrer o evento **`fim de serviço`** e a fila de entrada estiver vazia, o roteador volta a ficar **`livre`** novamente. A qualquer momento da simulação (em um tempo determinado probabilisticamente) o roteador pode cair (se tornar **`inativo`**). Quando ocorre o evento **`elemento disponível`**, ele passa ao estado **`livre`**, pois não conterá nenhum pacote em suas filas de entrada.

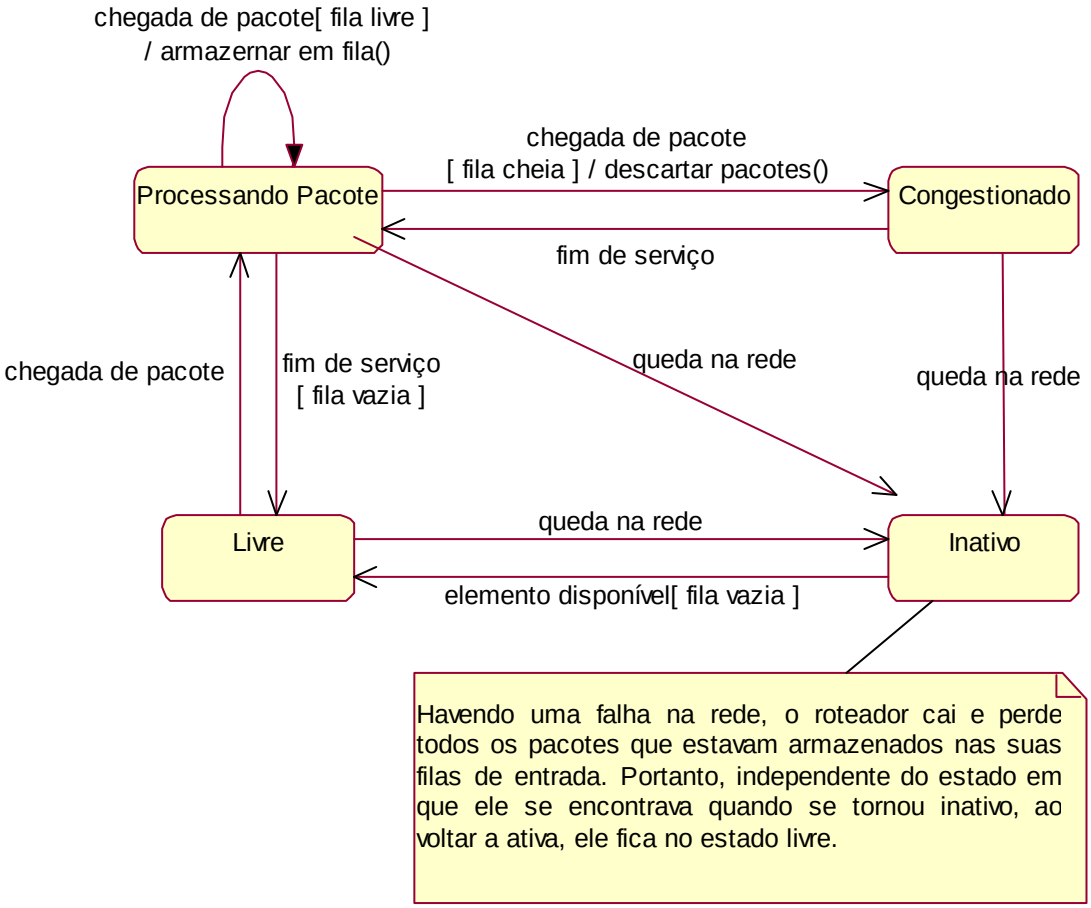

Figura 4-2: Diagrama de Estados do Roteador

Diagramas de estado similares podem ser construídos para os elementos *host* e enlace.

A utilização dos componentes especificados nesse trabalho devem permitir a implementação de qualquer tipo de simulação orientada a eventos. Entretanto, para tipos de simulações específicos pode ser necessário especificar novas entidades de interesse.

4.2.2 Descrição dos usuários alvos

O usuário principal deste trabalho é o desenvolvedor dos componentes aqui especificados para a construção facilitada de uma ferramenta de simulação orientada a eventos. Os componentes especificados, a princípio voltados para redes TCP/IP, poderão também ser utilizados para o desenvolvimento de qualquer ambiente de simulação orientado a eventos.

4.2.3 Descrição das metas do sistema

A meta básica desse trabalho é a especificação de componentes para facilitar o desenvolvimento de novos ambientes de simulação, através da reusabilidade de software. Para atingir essa meta, é necessário atingir metas mais específicas durante todo o processo de desenvolvimento, descritas a seguir:

- ✓ Definir o problema a ser tratado e qual o nível de abstração a ser adotado;
- ✓ Definir quais os componentes que farão parte do ambiente de simulação;
- ✓ Fazer a especificação desses componentes;
- ✓ Propor relacionamentos entre esses componentes para a construção de um ambiente de simulação.

4.2.4 Descrição dos requisitos

Os requisitos são uma descrição das necessidades ou desejos para um produto. Os requisitos do sistema podem ser classificados como Funcionais (o que o sistema deve fazer) e Não-Funcionais (atributos do sistema). As categorias de requisitos funcionais são [Larman, 98]:

Categoria de funcionalidade	Significado
Evidente	Usuário do sistema está ciente de que a função está sendo feita
Escondida	Embora a função seja feita, ela é invisível ao usuário.
Opcional	Funcionalidade opcional; sua adição não afeta outras funções ou o custo de desenvolvimento significativamente

Os Requisitos Funcionais levantados juntamente com sua classificação são:

F1 - O sistema deve permitir a construção e a simulação de modelos de redes TCP/IP utilizando os componentes especificados em [Wagner, 00] (Evidente).

F2 – O sistema deve prover um mecanismo de coleta de dados para o cálculo de medidas de desempenho relevantes (Evidente). Essas medidas são definidas pela ferramenta de simulação.

F3 – O sistema deve fornecer também um mecanismo para calcular medidas não definidas pela ferramenta de simulação, a pedido do usuário (Evidente).

F4 - A simulação é acionada por eventos. Um evento é uma "perturbação" instantânea que modifica o estado do sistema (Escondido). Os módulos do sistema devem responder à ocorrência de eventos. Os eventos devem ser escalonados (processados) segundo sua ordem cronológica de ocorrência.

F5 – a construção e a configuração do modelo são realizadas de forma visual (Opcional).

F6 – Os elementos de modelagem devem ser configuráveis pelo usuário (Evidente).

F7 - O usuário define os parâmetros iniciais da simulação: condição de término (por tempo ou por alguma condição especificada pelo usuário), tempo inicial e número de replicações. (Evidente).

F8 - Um mesmo modelo pode ser simulado mais de uma vez, permitindo a obtenção de amostras de medidas de desempenho. Portanto, pode-se calcular valores médios para as medidas coletadas, definindo intervalos de confiança para esses valores médios (Evidente).

F9 - O simulador deve verificar a consistência do modelo antes da simulação ou a pedido do usuário (Evidente).

F10 - A ferramenta de simulação deve gerar valores aleatórios conforme alguma função de distribuição de probabilidade conhecida (exponencial, uniforme, etc.), gerando automaticamente as sementes necessárias (Escondido).

F11 - O relógio é o mecanismo adotado que representa a evolução do tempo na simulação e representa um tempo "virtual" em que a simulação ocorre. Ele deve avançar de acordo com o tempo do evento sendo processado (Escondido).

F12 – O sistema deve ser capaz de possibilitar perdas de pacotes durante a simulação, devido aos *buffers* utilizados nas redes serem finitos (Evidente).

F13 – O sistema deve prover um mecanismo de acompanhamento passo a passo do comportamento do modelo durante a simulação (Evidente).

Os requisitos não funcionais, também chamados de atributos do sistema, especificam restrições impostas à solução, como aquelas relacionadas com a adoção de

padrões e a integração com outros sistemas: flexibilidade, facilidade de uso, portabilidade, etc [Freire, 00].

Os Requisitos Não-Funcionais encontrados são:

NF1 – A utilização de componentes permite o desenvolvimento rápido de novas ferramentas de simulação. A construção dessas ferramentas deve ser realizada visualmente através da composição dos componentes em um editor gráfico, exigindo o mínimo de programação.

NF2 – Os componentes fornecem abstrações de alto nível que são facilmente utilizáveis e estensíveis. Portanto, não deve ser necessário manipular estruturas de dados complexas nem utilizar características de uma linguagem particular que exijam considerável experiência do programador no momento de utilizar ou estender os componentes disponíveis.

NF3 – As ferramentas de simulação construídas a partir dos componentes especificados devem permitir a simulação de qualquer tipo de modelo discreto.

NF4 – Os componentes devem ser transportáveis para os principais ambientes operacionais em uso.

NF5 - Os documentos gerados na especificação dos componentes devem seguir uma linguagem padrão de modelagem (UML), permitindo sua reutilização futuramente, e facilitando a sua compreensão.

Analisando os requisitos não-funcionais levantados, a solução para esse sistema deve permitir velocidade (NF1) e flexibilidade (NF2) no desenvolvimento de novos simuladores. Portanto, para atender tais requisitos, torna-se necessário prover uma forma de reutilização. Componentes de software representam um importante passo no sentido de sistematizar a produção de software, pois provê reusabilidade num alto nível de abstração [Freire, 00].

Com a utilização de componentes, as aplicações são desenvolvidas com a composição de pedaços existentes. A composição das aplicações é feita utilizando ferramentas gráficas que permitem a configuração e conexão dos componentes necessários à aplicação de forma visual, segundo o requisito NF1.

O resultado do levantamento dos requisitos funcionais do sistema pode ser visualizado graficamente através de um diagrama de use-cases, descrito a seguir.

4.2.5 Diagrama de *Use-Cases*

Os *use-cases* são usados para descrever os processos do domínio de problema. Eles são uma excelente forma de explorar e documentar os requisitos funcionais, sendo usado como documento básico de referência durante todo o processo de desenvolvimento [Larman, 98]. Eles também podem servir como base para elaborar os testes funcionais do sistema final.

O diagrama de *use-cases* descreve graficamente as relações entre os *use-cases* (acontecimentos que ocorrem no sistema) e os usuários do sistema (atores). A figura 4.3 representa o diagrama de *use-cases* do sistema proposto neste trabalho.

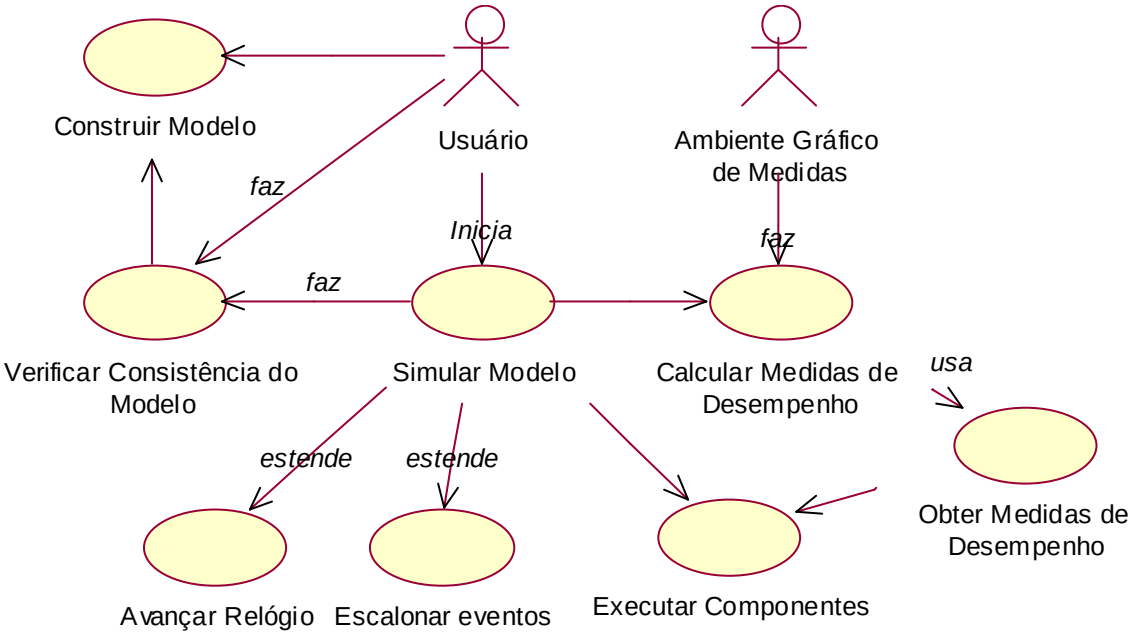

Figura 4-3: Diagrama de *Use-Cases*

A seguir são descritos os *use-cases* do diagrama da figura 4.3, sendo o *use-case* **`Executar Componentes`** descrito brevemente pois os elementos que fazem parte da rede TCP/IP são especificados com detalhes em [Wagner, 00].

`Simular Modelo`:

Ator: usuário

Descrição: é o processo responsável pela inicialização e controle da simulação e sua finalização. Na inicialização, os parâmetros da simulação são configurados. Esse *use-case* é estendido nos *use-cases* **`Avançar Relógio`**, **`Escalonar Eventos`**, **`Executar Componentes`** descritos a seguir:

`Avançar Relógio:`

Ator: não há interação direta com os atores

Descrição: responsável pela atualização do relógio simulado.

`Escalonar Eventos:`

Ator: não há interação direta com os atores

Descrição: responsável pelo escalonamento dos eventos ocorridos durante a simulação em ordem cronológica.

`Executar Componentes:`

Ator: não há interação direta com os atores

Descrição: de acordo com cada tipo de evento escalonado, um dos elementos da rede deve ser executado. A execução de um elemento da rede consiste em executar ações associadas a ele que podem gerar novos eventos.

`Obter Medidas de Desempenho:`

Ator: usuário

Descrição: responsável pela obtenção das medidas de desempenho "*default*" em cada elemento da rede, ou de uma medida determinada pelo usuário final.

`Calcular Medidas de Desempenho:`

Ator: ambiente gráfico de medidas

Descrição: responsável por calcular estatísticas sobre as medidas coletadas que podem ser visualizadas através de gráficos (histogramas, curvas de nível, etc.).

`Verificar Consistência do Modelo:`

Ator: usuário

Descrição: responsável por fazer a verificação do modelo antes de iniciar a simulação, a pedido do analista de modelagem ou pelo próprio simulador. Essa verificação deve informar ao usuário de erros caso sejam detectados.

`Construir Modelo:`

Ator: usuário

Descrição: responsável pela construção do modelo de rede para a simulação e sua configuração. Este *use-case* terá uma interação muito grande com a interface do usuário do simulador que não será especificada uma vez que não pertence ao escopo deste trabalho.

Foram levantados os requisitos básicos do sistema proposto, sendo necessário agora definir mais detalhadamente esse sistema e prover uma solução adequada, através das fases de análise e projeto.

A experiência comprova que durante o processo de desenvolvimento normal de qualquer produto de software, as fases de análise (descrição do problema) e de projeto (descrição da solução) são as mais importantes e consomem a maior parte do tempo [Landin & Niklasson, 98], pois são tomadas as principais decisões, e um erro cometido pode prejudicar a qualidade do sistema.

A partir da próxima seção essas fases serão detalhadas, através de diagramas que servem para definir melhor o sistema. Também será mostrada a solução adotada para o desenvolvimento desse sistema.

4.3 Fase de Análise

Na fase de análise é gerado um documento de alto nível para entender o domínio do problema, mostrando uma solução possível para atender aos requisitos levantados na etapa anterior. Nesta fase, são considerados somente os aspetos do ponto de vista do usuário (domínio do problema), não entrando em detalhes de implementação. Investigação e análise são freqüentemente caracterizadas por focalizarem questões do tipo ***qual*** – quais são os processos, os conceitos, os eventos e as operações [Larman, 98].

Durante essa fase, alguns diagramas UML podem ser usados para facilitar a compreensão do problema, sendo um modelo conceitual o principal artefato gerado. O modelo conceitual ilustra os conceitos importantes do domínio do problema, suas associações e atributos. Uma técnica utilizada para se gerar um modelo conceitual é isolar os substantivos das descrições textuais dos *use-cases*, utilizar regras para selecionar os conceitos relevantes [Rumbaugh et al., 99a], e adicionar associações e atributos a esses conceitos.

4.3.1 Modelo Conceitual

O modelo conceitual gerado a partir de conceitos encontrados é mostrado na figura 4.4.

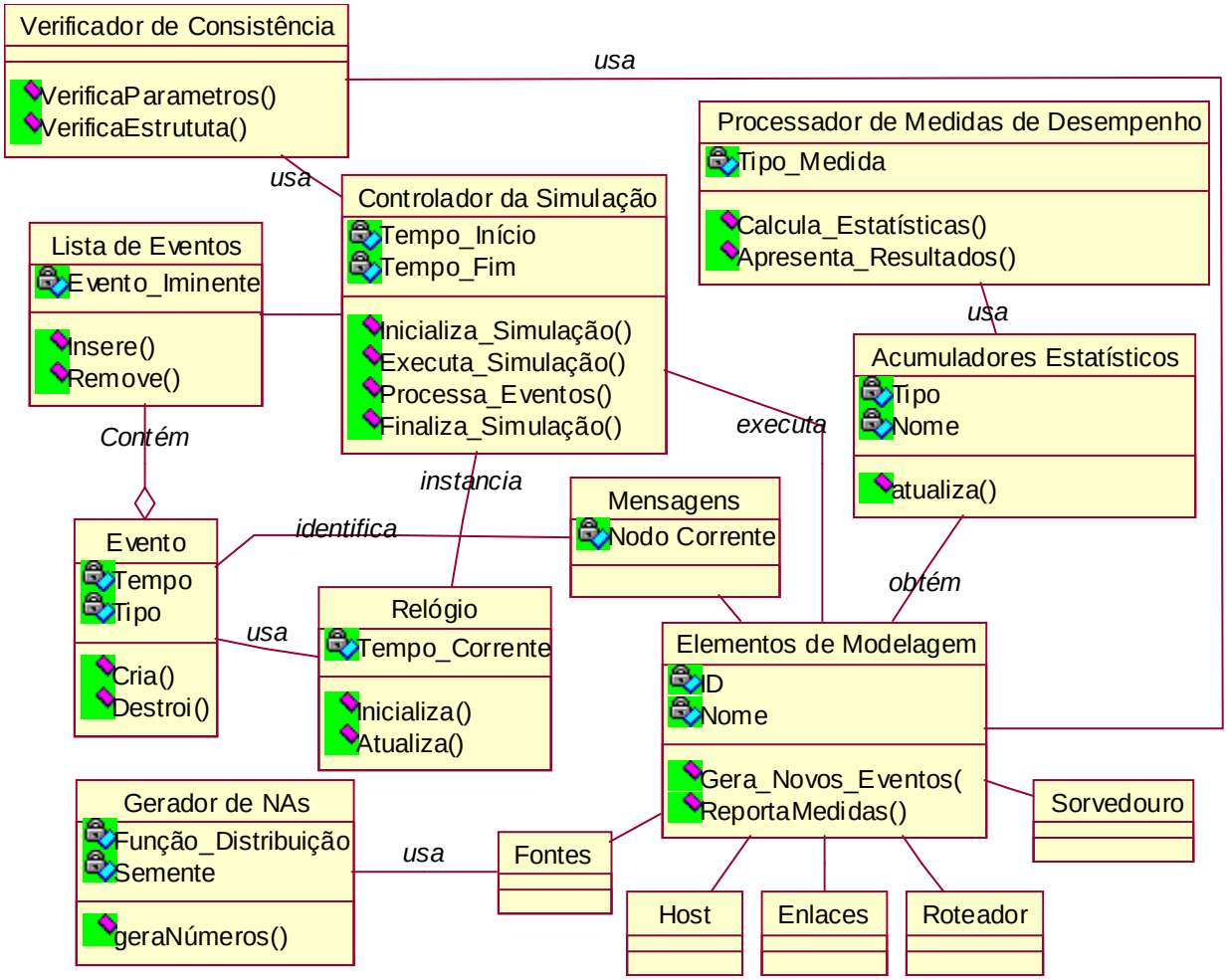

Figura 4-4: Diagrama de Conceitos

Os conceitos **`Fonte`**, **`Host`**, **`Enlaces`**, **`Roteador`**, **`Sorvedouro`** e **`Mensagens`** foram especificados em [Wagner, 00], sendo somente acrescentado o atributo *Nodo_Corrente* ao conceito **`Mensagens`**. Os pacotes que transitam nas redes TCP/IP são especializações do componente **`Mensagens`**. Os outros conceitos são encapsulados no conceito **`Elementos de Modelagem`**.

Os conceitos envolvidos nesse diagrama dizem respeito aos candidatos a componentes de um ambiente de simulação genérico.

O **`Controlador da Simulação`** é o responsável pelo controle da simulação de uma forma geral. Ele é responsável pelo avanço do relógio, escalonamento dos eventos e execução das ações associadas a cada evento. Ele também é responsável pelo início e término da simulação. Para tal, ele possui os atributos *início* e *fim da simulação*, e suas

responsabilidades são *inicializar*, *executar* e *finalizar* uma simulação, de acordo com seus atributos.

O **Relógio** indica o tempo simulado durante a simulação. Ele vai ser atualizado de acordo com o tempo do evento iminente (atualizando *Tempo_Corrente*). A cada início de simulação, ele é zerado e atualizado para o tempo de inicialização especificado para a simulação (*inicializar* e *atualizar*).

A **Lista de Eventos** é responsável pelo armazenamento dos eventos criados durante toda a simulação em ordem cronológica. Ele tem como atributo o *evento_iminente* e ele usa uma técnica intrínseca de inserção e remoção de eventos (*inserir* e *remover*).

Cada **Evento** pertence a um determinado tipo e tem um tempo de ocorrência associado a ele (atributos *tipo* e *tempo*). Cada evento também tem associado uma instância do conceito **Mensagem**. O evento pode ser criado e destruído (*cria* e *destrói*).

Os **Acumuladores Estatísticos** são variáveis que representam as medidas que serão necessárias para avaliar o desempenho do sistema. Esses acumuladores serão obtidos durante toda a simulação, e alguns serão invisíveis aos usuários. Eles deverão especificar o *tipo* (que identifica qual a medida correspondente) e um *nome*. De acordo com cada tipo, ele terá uma função associada a ele.

O **Processador de Medidas de Desempenho** obtém dados dos **Acumuladores Estatísticos** para calcular algumas medidas de desempenho de interesse (*calcular_estatísticas*).

Alguns elementos de rede necessitarão de amostras de números aleatórios para obterem valores aleatórios. O **Gerador de VAs** é responsável pela geração de valores aleatórios de acordo com alguma função de distribuição de probabilidade (*Função_Distribuição*), conforme uma semente especificada (*semente*).

Antes de cada simulação, é realizada uma verificação de consistência do (**Verificador de Consistência**). Ele não permite nenhum tipo de inconsistência na rede entre seus elementos (*VerificaEstrutura*). Deve também verificar se todos os dados de entrada foram especificados e se são válidos (*VerificaParametros*).

Somente o modelo conceitual apresentado pode não ser suficiente para que o sistema seja realmente compreendido. Com esse intuito, alguns diagramas complementares

podem ser criados para dar suporte ao entendimento do sistema, como o diagrama de seqüência e o diagrama de estados.

4.3.2 Diagrama de Seqüência

O diagrama de seqüência mostra o comportamento dinâmico do sistema. Os diagramas de seqüência do sistema são construídos a partir dos *use-cases*. Um diagrama de seqüência mostra, para um cenário particular de um *use-case*, os eventos gerados pelos atores, sua ordem, além de eventos envolvendo sistemas externos . Esses eventos geram operações sobre o sistema. Vale salientar que esse diagrama continua sendo do ponto de vista do usuário, mostrando o *que* o sistema faz, e não *como* ele faz. É interessante que se gere diagramas de seqüência apenas para os *use-cases* mais importantes, e dentro destes, aqueles que representam as situações mais críticas [Larman, 98].

Um diagrama de seqüência foi construído para o sistema especificado nesta dissertação. Esse diagrama está representado na figura 4.5 e diz respeito ao *use-case* **`Simular o Modelo`**, o principal *use-case* do modelo.

Após construir e configurar o modelo, o usuário inicializa a simulação. O **`Simular Modelo`**, antes de efetivamente começar a simular, executa a verificação de consistência do modelo. Estando o modelo devidamente verificado, a simulação é inicializada com o escalonamento do primeiro evento. O relógio é atualizado para o tempo do evento escalonado, e as ações associadas ao evento são executadas. Essas ações geralmente dizem respeito à execução de algum elemento de rede.

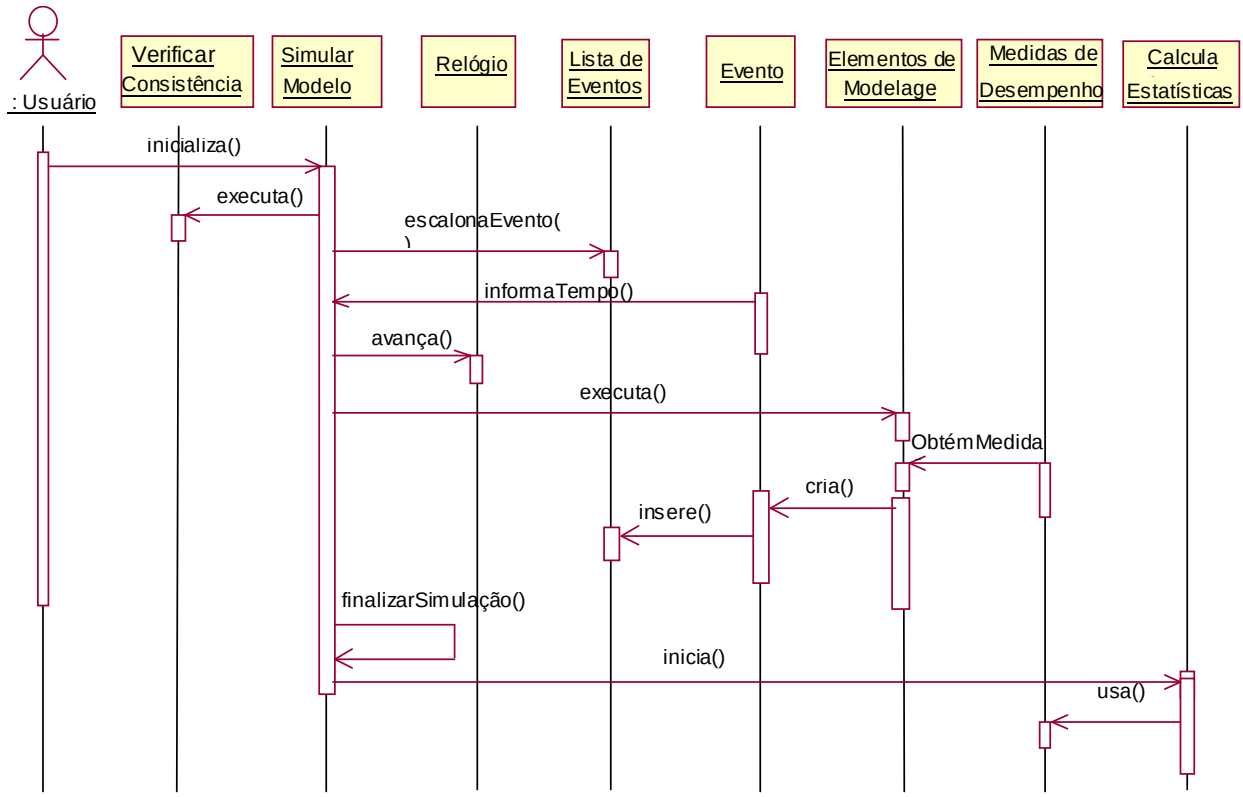

Figura 4-5: Diagrama de Sequência do *Use-Case* Simular Modelo

Quando um elemento de rede é executado, são obtidas algumas medidas de desempenho de interesse e eventualmente são criados novos eventos que são inseridos na lista de eventos. Esse mesmo procedimento (a partir do escalonamento do evento) é feito para cada evento da lista de eventos.

Após um determinado tempo ou uma quantidade de eventos terem sido processados, a simulação é finalizada. Então, é iniciada a fase de cálculo das estatísticas que permite a análise de desempenho da rede utilizando as medidas obtidas durante a simulação.

Capítulo 5

5 Fase de Projeto

Neste capítulo é apresentada a fase de projeto do processo de desenvolvimento adotado. Esta fase diz respeito à especificação em baixo nível dos componentes para a construção de uma ferramenta de simulação orientada a eventos.

A seção 5.1 apresenta uma introdução à fase de projeto. Na seção 5.2 é apresentado o projeto arquitetural em três camadas. Na seção 5.3 é descrita a fase de projeto detalhado. Nesta seção são construídos os diagramas de colaboração. Na seção 5.4 são mostrados os componentes especificados em termos de seus métodos, atributos e interfaces. Uma tabela resumindo todos os componentes levantados é mostrada ao final desta seção. Na seção 5.5 é mostrada uma validação para a especificação realizada de acordo com os requisitos levantados no capítulo anterior.

5.1 Introdução

A fase de projeto é uma extensão da fase de análise, visando a implementação do sistema em um computador. Na fase de análise, ênfase é feita em cima da questão "*o quê?*". A fase de projeto concentra-se na questão "*como?*". Os resultados obtidos nesta fase visam atender às necessidades do programador, não se voltando, portanto, para o entendimento do usuário.

O projeto de um sistema é dividido em duas partes: projeto arquitetural (projeto de alto nível) e projeto detalhado (projeto de baixo nível). Durante a fase de projeto são

identificados os componentes que fazem parte do sistema. Este capítulo tem como referência [Larman, 98].

5.2 Projeto Arquitetural

Durante o projeto arquitetural são tomadas decisões estratégicas, tais como modularização do projeto em subsistemas, determinação de oportunidades para o reuso de software, atendimento a requisitos de desempenho, custo, uso de padrões, etc. Nessa fase os subsistemas encontrados são apresentados em alto nível.

O projeto arquitetural geralmente é apresentado numa arquitetura de 3 camadas:

- ✓ *Camada de Apresentação*: define a interface com o usuário;
- ✓ *Camada de Aplicação*: define a lógica da aplicação, isto é, define as tarefas e regras que governam o processo;
- ✓ *Camada de Dados*: consiste nos mecanismos de armazenamento persistente.

Uma das vantagens do uso de uma arquitetura em 3 camadas é a capacidade de isolar a lógica da aplicação em uma camada intermediária, provendo um mecanismo de comunicação com as outras duas. Esse tipo de arquitetura permite que a aplicação seja representada em componentes separados, fornecendo um alto grau de reutilização [Larman, 98].

Outra vantagem do uso de uma arquitetura em 3 camadas é a distribuição das camadas em diferentes nós físicos de processamento, melhorando o desempenho e aumentando a coordenação e o compartilhamento de informações em um sistema cliente-servidor.

Essa arquitetura possibilita também a alocação de desenvolvedores com especialização de conhecimentos para a construção de camadas específicas, permitindo o desenvolvimento simultâneo das camadas. Por exemplo, existir uma equipe de desenvolvedores exclusivamente para a camada de apresentação.

É possível acrescentar camadas adicionais e decompor ainda mais as existentes (**arquitetura em múltiplas camadas**). Em um projeto orientado a objetos, a camada da lógica da aplicação é decomposta nas seguintes subcamadas:

- ✓ *Objetos do Domínio*: classes que representam conceitos do domínio do problema, tal como uma venda.
- ✓ *Serviços*: objetos de serviço, tais como interação com o banco de dados, segurança, etc.

Para representar graficamente os subsistemas encontrados e suas dependências entre camadas, a linguagem UML provê o mecanismo de *pacotes*. Um *pacote* em UML é um conjunto de elementos do modelo de qualquer tipo, tais como classes. Neste trabalho, um pacote define um componente de software.

O projeto arquitetural para o projeto é mostrado na figura 5.1:

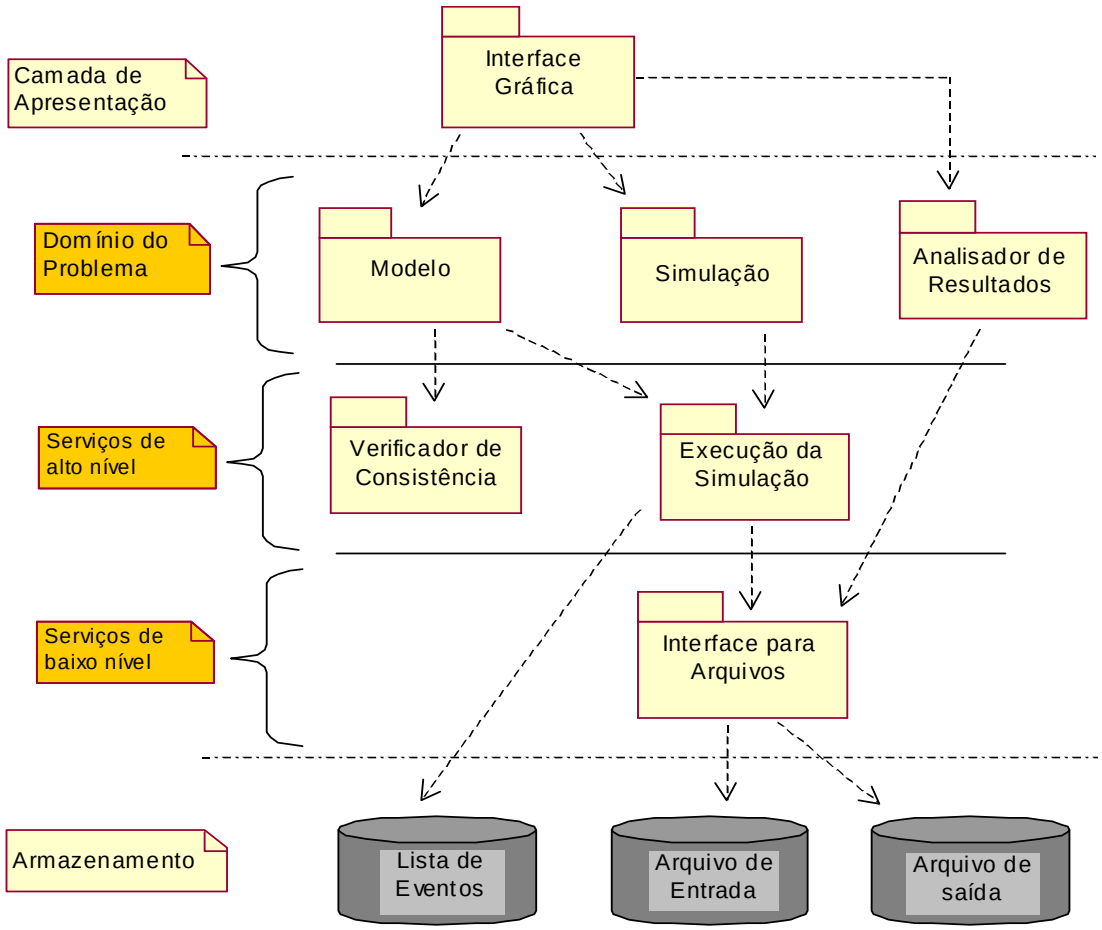

Figura 5-1: Projeto Arquitetural em Camadas

As camadas de apresentação e de armazenamento não estão contidas dentro do escopo desta dissertação. Portanto, o trabalho está focado na camada da lógica da aplicação. Cada camada mostrada na figura 5.1 é detalhada a seguir.

5.2.1 Camada de Apresentação

Apesar da camada de apresentação não estar contida no escopo desse trabalho, algumas diretrizes podem ser traçadas. A camada de apresentação define a interface com o usuário da aplicação, tais como janelas, *applets*, etc. Elas são responsáveis pelas entradas e saídas, mas não mantêm dados nem fornecem, diretamente, funcionalidades da aplicação.

Tipicamente, nenhum pacote das camadas de domínio e de serviço deve ter visibilidade direta com os pacotes da camada de apresentação, segundo o padrão de projeto *Model-View Separation* [Larman, 98]. O uso desse padrão permite o desenvolvimento e a execução da camada de aplicação independente da interface do usuário, provendo maior reusabilidade dos componentes da aplicação.

Mas algumas aplicações de monitoramento para produzir uma exibição em tempo real de uma atualização em andamento, tais como aplicações para simulação de redes de computadores, necessitam de algum mecanismo de comunicação indireta com a interface. Uma boa alternativa de comunicação é o uso do padrão *Observer* [Gamma et al., 95]. A utilização deste padrão permite que um objeto notifica os objetos cadastrados nele quando seu estado mudar.

5.2.2 Camada de Aplicação

Para que os componentes da camada da aplicação sejam reutilizados em novas aplicações ou independente de interfaces, eles devem encapsular somente a informação e o comportamento relacionados com a lógica da aplicação.

A camada da lógica de aplicação da figura 5.1 foi dividida em 3 subcamadas:

- ✓ *Domínio do problema*: nesta camada estão inseridos os componentes que são relacionados ao analista de modelagem durante a construção de uma simulação e apresentação dos resultados.
- ✓ *Serviços de Alto Nível*: nesta camada estão inseridos os componentes relativos à execução da simulação, tais como relógio, escalonador de eventos, etc.
- ✓ *Serviços de Baixo Nível*: nesta camada estão inseridos os componentes que dão suporte à comunicação com os arquivos de armazenamento de dados.

A seguir são descritos os componentes que fazem parte de cada subcamada citada acima.

Domínio do Problema:

`Componente Modelo`: responsável pela instanciação e configuração dos elementos de modelagem que farão parte do modelo. Ele vai conter todos os elementos que fazem parte do modelo.

`Componente Simulação`: responsável pela configuração dos parâmetros iniciais da simulação e sua inicialização.

`Analisador de Resultados`: responsável pelo cálculo e apresentação das estatísticas sobre as medidas de desempenho de interesse coletadas durante a execução da simulação.

Serviços de Alto Nível:

`Componente Verificador de Consistência`: responsável pela verificação de erros no modelo.

`Execução da Simulação`: responsável pela execução da simulação em si. Para que a execução da simulação seja realizada, alguns componentes de suporte são necessários. Esses componentes são mostrados na figura 5.2:

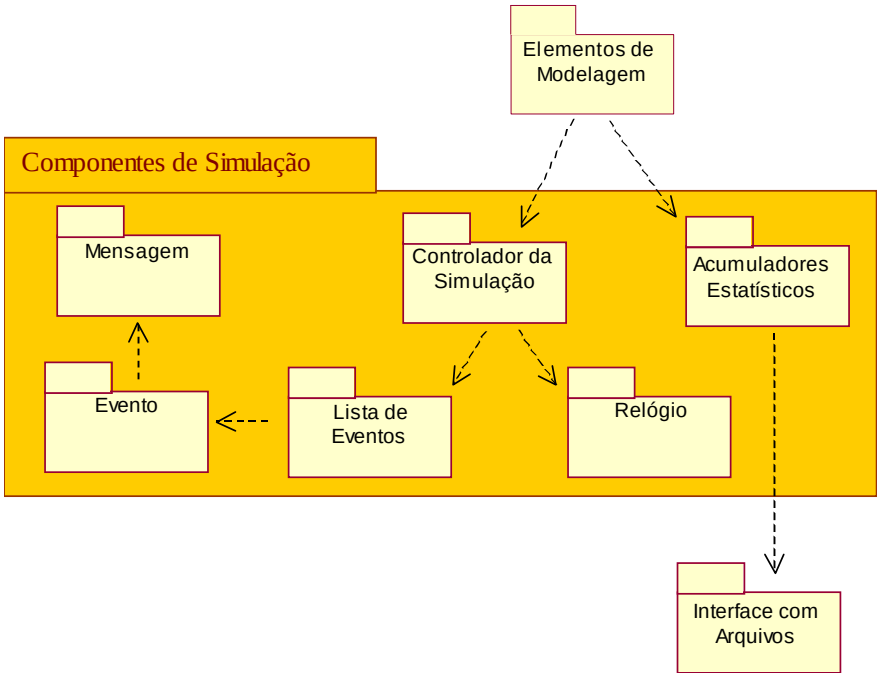

Figura 5-2: Componentes do Pacote da Execução da Simulação

`Componente Controlador da Simulação`: responsável pelo controle da simulação. Ele ativa o algoritmo de simulação.

Componente Acumuladores Estatísticos: responsáveis por armazenar os dados levantados durante a simulação. Representam os dados coletados e permitem o cálculo das medidas de desempenho de interesse.

Componente Relógio: armazena o tempo corrente da simulação.

Componente Lista de Eventos: armazena, em ordem cronológica, os eventos criados durante a simulação.

Componente Evento: corresponde a um evento da simulação, como **chegada de pacotes**.

Componente Mensagem: representa as entidades que circulam entre os elementos de modelagem do modelo.

Serviço de Baixo Nível:

Interface com Arquivos: responsável pela comunicação entre a aplicação e os arquivos de entrada e de saída da simulação.

5.2.3 Camada de Armazenamento

Uma forma de persistência dos dados da aplicação deve ser escolhida durante a fase de projeto. As formas básicas de persistência são:

- ✓ Dados na memória
- ✓ Dados em arquivos
- ✓ Dados sob controle de um Banco de Dados

Alguns fatores podem influenciar a escolha do método de armazenamento de dados, tais como custos, persistência e quantidade de dados, desempenho, etc.

Para os dados de entrada e de saída desse sistema foi escolhido arquivos como forma de armazenamento. Os dados utilizados e gerados pela simulação são basicamente simples, não necessitando de mecanismos mais complexos como o uso de bancos de dados. Uma das vantagens do uso de arquivos neste projeto é a persistência de dados, necessário para acesso futuro dos resultados da simulação para análise. Outra vantagem é o baixo custo de aquisição pois eles são inerentemente suportados pelo sistema operacional. O acesso ao arquivo não afeta o desempenho da simulação, pois os dados deverão ser lidos e gravados seqüencialmente com alguns acessos randômicos.

Porém, a lista de eventos será armazenada em memória pois, como ela deverá ser muito atualizada durante toda a simulação, seu acesso deve ser bastante rápido.

Após essa visão macroscópica do sistema, os componentes devem ser detalhados num projeto de baixo nível.

5.3 Projeto Detalhado

Enquanto que a fase de projeto arquitetural consiste numa visão “macro” do sistema, a fase de projeto detalhado visa definir a arquitetura interna da solução lógica, identificando os componentes individuais que compõem o sistema. Durante esta fase, são criados diagramas de interação e diagramas de classes de projeto. Os diagramas de interação consistem nos diagramas de colaboração e nos diagramas de sequência. Por causa de sua capacidade de expressar mais detalhes sobre colaboração entre os objetos e de expressar exceções, são utilizados somente os diagramas de colaboração durante o projeto detalhado.

Os diagramas de colaboração mostram como os componentes devem se comunicar de maneira a atender os requisitos especificados, além de ajudar a atribuir responsabilidades (métodos) a cada componente. Sendo a atribuição de responsabilidades uma das tarefas mais difíceis desta fase, o uso de padrões de projeto são extremamente úteis.

Como foi dito anteriormente, uma simulação consiste em três fases: inicialização do modelo, sua execução e finalização. Para cada etapa, foi desenvolvido alguns diagramas de colaboração.

5.3.1 Fase de inicialização

Alguns aspectos nesta fase não serão detalhados pois fogem ao escopo deste trabalho, por involver a interface gráfica do ambiente de simulação. A inicialização (construção e configuração do modelo) é feita com a ajuda de uma interface gráfica (o componente **`Simulador`** provê a comunicação entre a interface gráfica e o simulador). A partir do componente **`Simulador`** são criados os elementos de modelagem que são inseridos no **`Modelo`**, segundo pode ser visto na figura 5.3. Os parâmetros de entrada (atributos) de cada elemento de modelagem é referenciado genericamente pela variável *dados_entrada*.

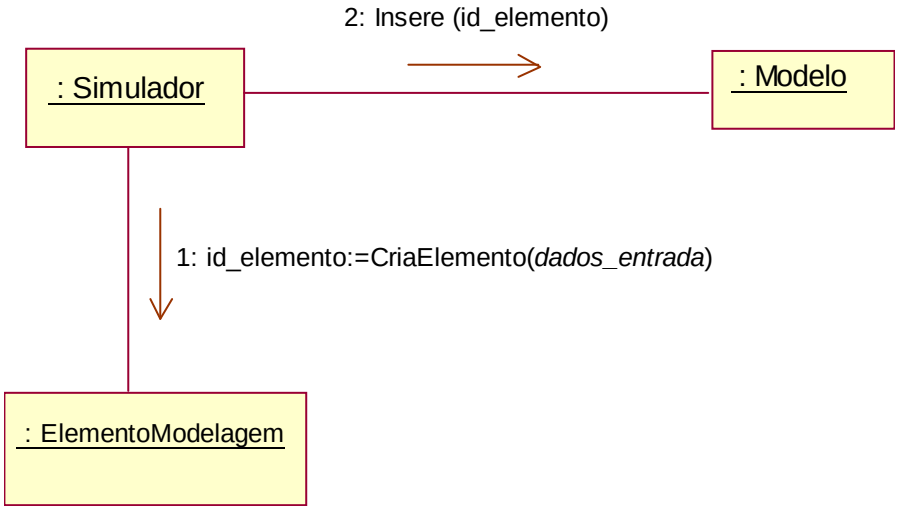

Figura 5-3: Diagrama de Colaboração para Construção do Modelo

É através da criação desses elementos que é definida a estrutura do modelo que representa a rede de comunicação a ser simulada. A validação dos parâmetros de entrada de cada elemento de modelagem é feita automaticamente quando eles estão sendo criados, de acordo com os seus parâmetros de entrada.

Antes da execução do modelo, deve-se inicializar a simulação através da definição de seus parâmetros. Essa incialização está representada no diagrama de colaboração da figura 5.4.

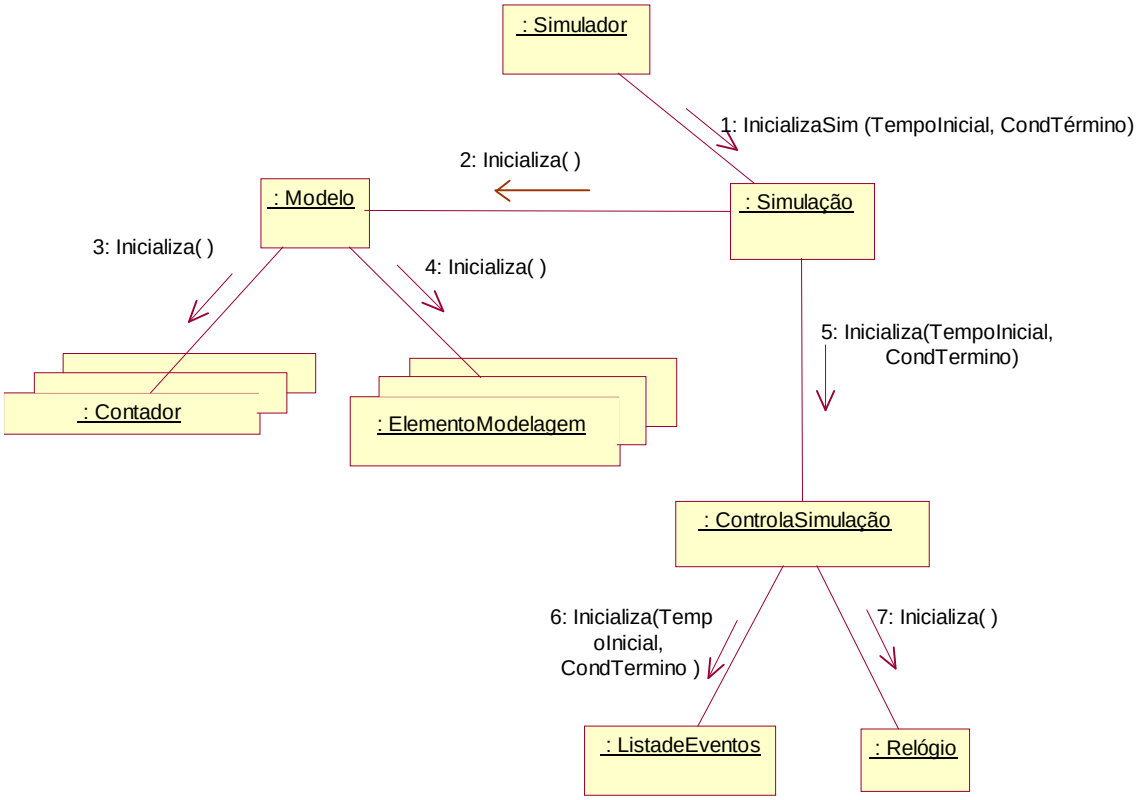

Figura 5-4: Diagrama de Colaboração para Inicialização da Simulação

Na incialização da simulação, o tempo inicial (*TempoInicial*) e a condição de término (*CondTermino*) são definidos pelo usuário. A inicialização consiste em chamar o método *InicializaSim()* do componente **`Simulação`**, que chama o método *Inicializa()* do componente **`ControlaSimulação`**. Este último inicializa o **`Relógio`** com tempo igual a zero e a **`ListaEventos`** com os primeiros eventos a serem escalonados de acordo com os parâmetros. A inicialização também consiste em inicializar as instâncias de **`ElementoModelagem`** (zerar acumuladores estatísticos, gerar novas sementes, etc.) e de **`Contador`**, contidos no **`Modelo`**. A cada nova replicação, a simulação é reinicializada.

Após a inicialização da simulação, deve-se iniciar a execução da simulação.

5.3.2 Fase de execução

Esta fase consiste na execução do algoritmo de simulação através do processamento dos eventos e da coleta de dados necessários para o cálculo de medidas de desempenho. O usuário inicia a execução da simulação através da ativação de algum comando do **`Simulador`** (figura 5.5). Neste momento, a simulação e o modelo já foram incializados.

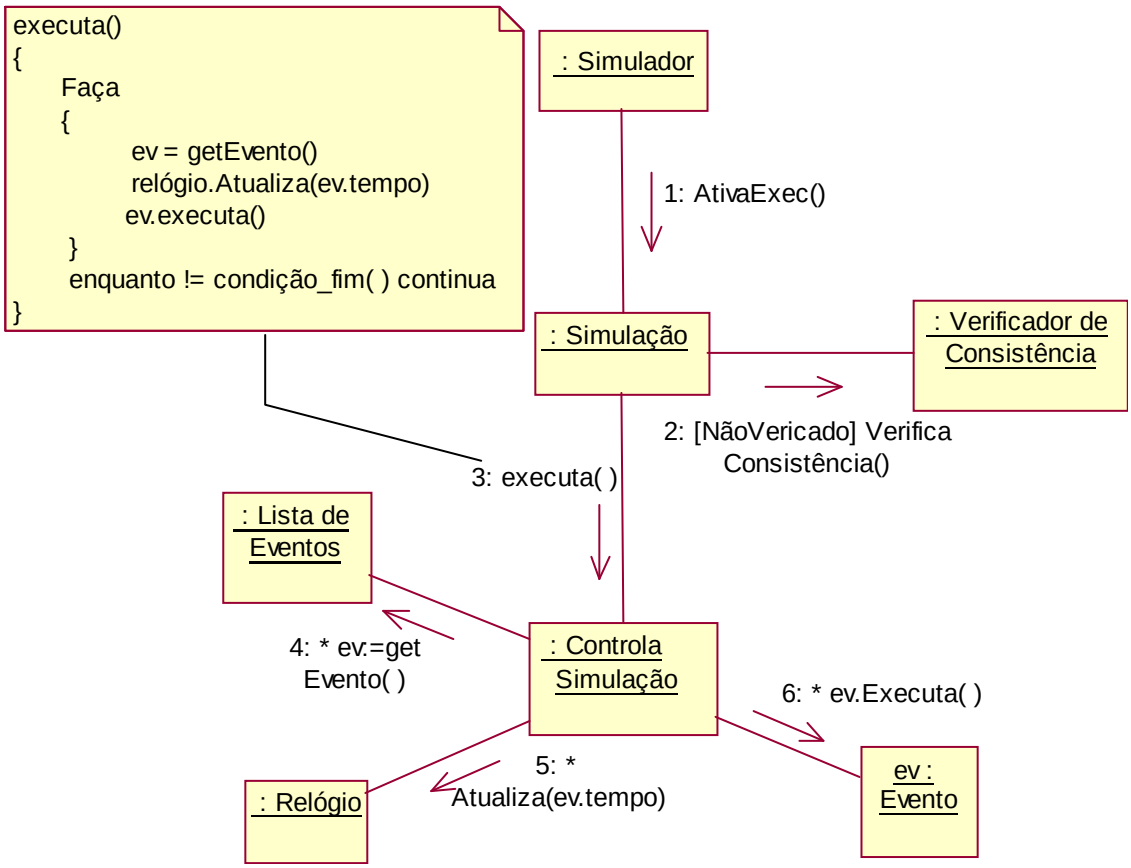

Figura 5-5: Diagrama de Colaboração para Execução da Simulação

Antes da execução em si, é feita a verificação de integridade do modelo. A verificação é sempre feita antes da execução da simulação para evitar que esta seja efetuada com informações inconsistentes. Ela consiste em verificar se existe:

- ✓ elementos referenciados no modelo mas não instanciados;
- ✓ elementos encadeados inadequadamente na rede;
- ✓ elementos instanciados mas não usados no modelo;
- ✓ redes sem fonte.

Essa verificação é feita pelo componente **`VerificadorConsistência`**. Para isso, ele se utiliza do componente **`Modelo`**, onde estão instanciados todos os elementos de modelagem da rede assim como a sua estrutura.

O algoritmo de simulação é iniciado pela execução do componente **`ControlaSimulação`**. Este componente chama o método *getEvento()* da **`ListaEventos`** que retorna o evento do topo da lista (*ev*). A partir do evento retornado ele chama o método *atualiza()* do componente **`Relógio`** passando como parâmetro o tempo do evento (*ev.tempo*).

Depois, ele chama o método *executa()* do componente **`Evento`**. Cada evento implementa o método *executa()* de acordo com seu tipo.

Para os principais eventos do simulador (*chegar pacote, fim de serviço*) foi criado um diagrama de colaboração. Estes diagramas ajudam a entender a execução de cada tipo de evento.

Primeiramente vamos analisar o evento **`ChegaPacote`**, mostrado na figura 5.6. Este diagrama mostra a chegada de pacotes em alguma estação de serviço (roteadores e *hosts*).

Quando o método *executa()* do evento **`ChegaPacote`** é executado, ele chama o método *RecebePacote(pacote)* da **`EstaçãoServiço`** associado ao nó corrente em que se encontra o pacote.

Se a **`EstaçãoServiço`** estiver livre, ela executa seu algoritmo (roteamento, caso seja um roteador e diagramação, caso seja um *host*) . Caso a estação esteja ocupada, o pacote é inserido na fila de entrada (**`Entrada`**). Se a fila estiver cheia, um algoritmo de descarte é acionado (*DescartaPacote()*).

Após a execução da **`EstaçãoServiço`**, um novo evento F**`imServiço`** é criado equivalente ao fim da execução. O tempo do evento é calculado de acordo com a distribuição de probabilidade da **`EstaçãoServiço`**, com a ajuda do **`GeradorVAs`**, e retornado pelo método *RecebePacote()*.

A inserção de um evento na **`ListaEventos`** utiliza o padrão de projeto *Observer*. O evento é criado pelo componente **`GeradorEvento`** que o dispara. O componente **`ListaEventos`** obedece a uma interface chamada **`EventoListener`** caracterizando-o como sendo um “consumidor” de eventos (mais detalhes na próxima seção). Este componente é cadastrado no **`GeradorEvento`**, capta o evento gerado e o insere na lista executando o método *InsereEvento(evento).*

Durante a execução do pacote, os acumuladores estatísticos são atualizados.

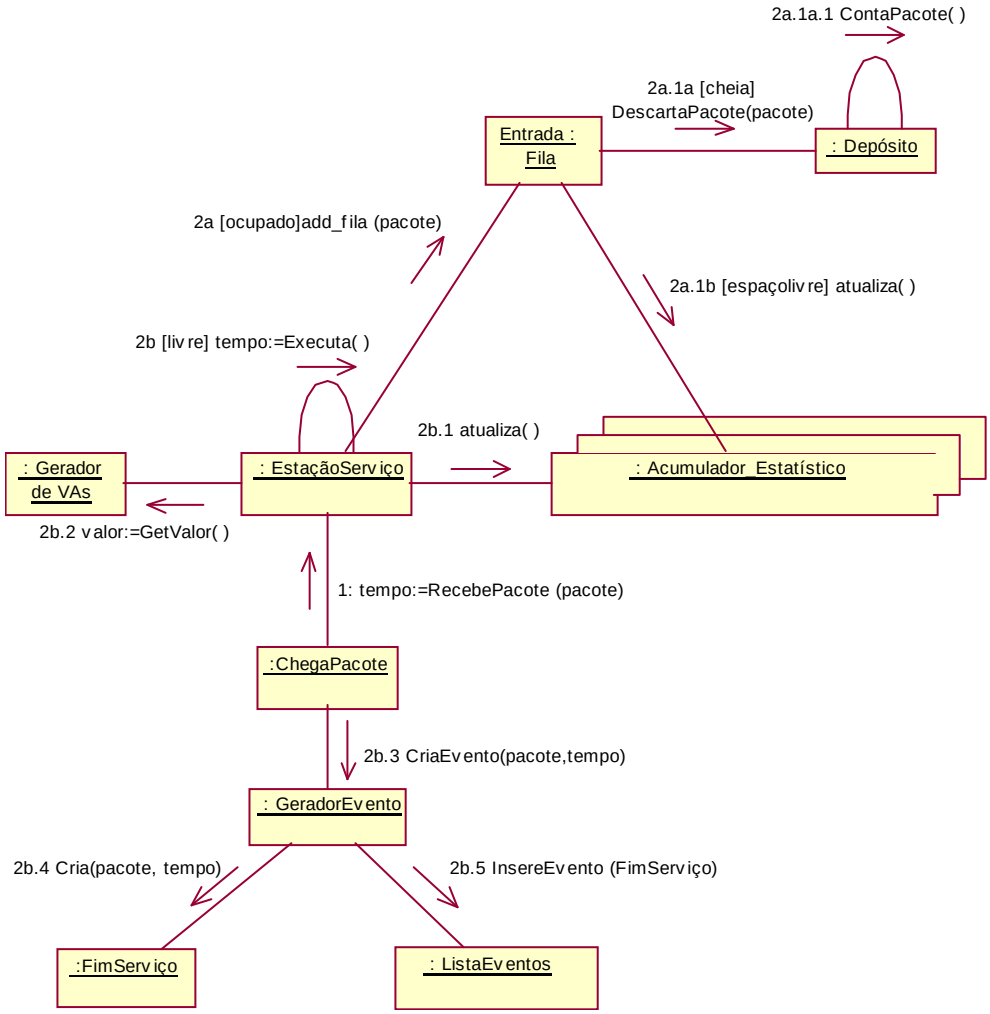

Figura 5-6: Diagrama de Colaboração para o Evento Chegar Pacote

Quando uma mensagem (pacote) chega em um *host*, uma próxima mensagem é criada automaticamente pela **`Fonte`**, como mostrado na figura 5.7. A **`Fonte`** obtém um valor aleatório do **`GeradorVAs`** para calcular a o intervalo de tempo para a próxima mensagem. O evento é inserido na **`ListaEventos`**.

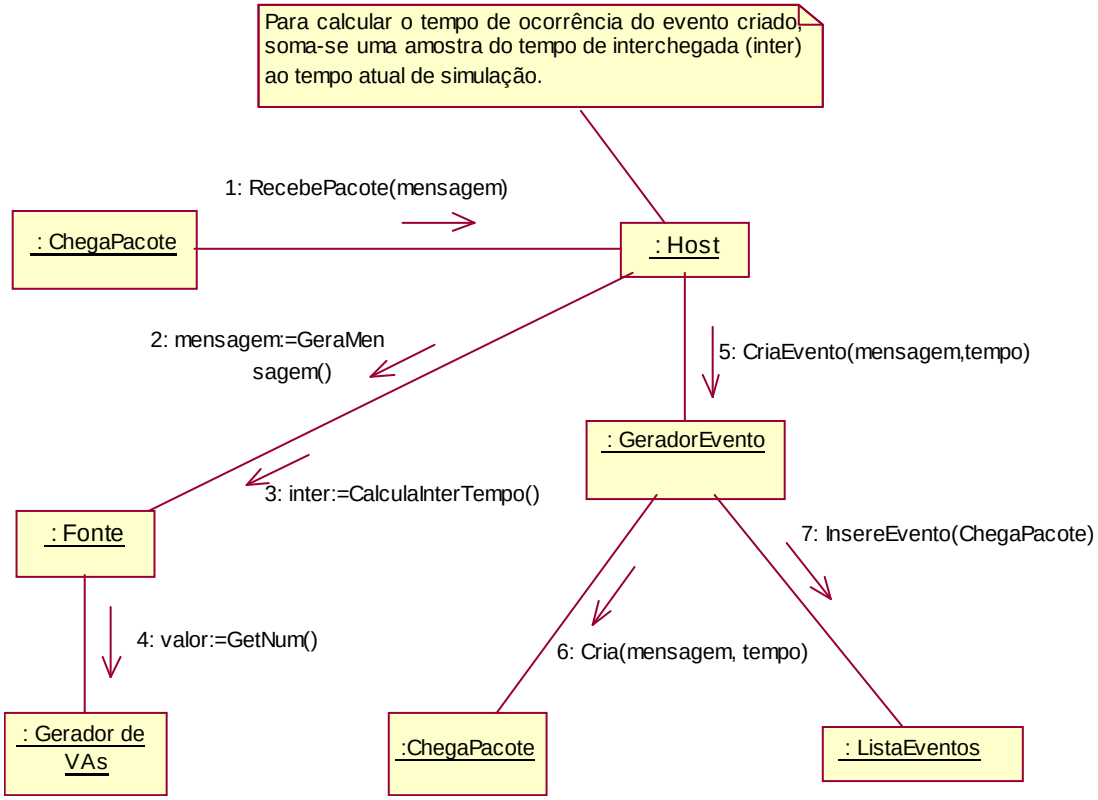

Figura 5-7: Diagrama de Colaboração para o Evento *ChegaPacote* no *Host*

Caso o pacote esteja chegando no sorvedouro, somente são atualizados os acumuladores estatísticos, como mostrado na figura 5.8. Não é criado nenhum evento no sorvedouro.

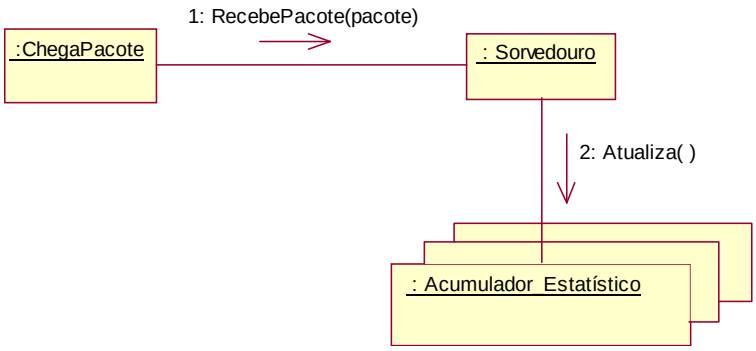

Figura 5-8: Diagrama de Colaboração para o Evento *ChegaPacote* no Sorvedouro

A seguir, vamos analisar o evento **`FimServiço`**, mostrado na figura 5.9. Quando o método *executa()* desse tipo de evento é acionado, ele chama o método *FinalizaServiço(pacote)* da **`EstaçãoServiço`**. Este método tenta transmitir o pacote pelo **`enlace`** correspondente. Se o **`enlace`** estiver livre, um evento **`FimTrasmissão`** é criado. O atributo *PróximoNó* do pacote é atualizado para a próxima **`EstaçãoServiço`** ou **`Sorvedouro`** e o evento **`FimTrasmissão`** é inserido na **`ListaEventos`**. O tempo de

ocorrência do evento é calculado observando o tempo de transmissão do enlace e o tamanho do pacote sendo transmitido.

Se o **enlace** estiver ocupado, o pacote é inserido na **fila de saída** correspondente. Caso a fila estiver cheia, o algoritmo de descarte é acionado. Geralmente, o algoritmo de roteamento escolhe o melhor caminho, procurando evitar congestionamento.

Após a criação do evento **FimTrasmissão**, é verificado se existe pacote na fila de entrada através do método *ExecutaProxPacote()*. Caso tenha, o pacote do topo da fila é removido, executado e esse método retorna o pacote. A seguir, é criado um evento **FimServiço** que é inserido na **ListaEventos**, com tempo retornado pelo método *getTempo()* da **EstaçãoServiço**.

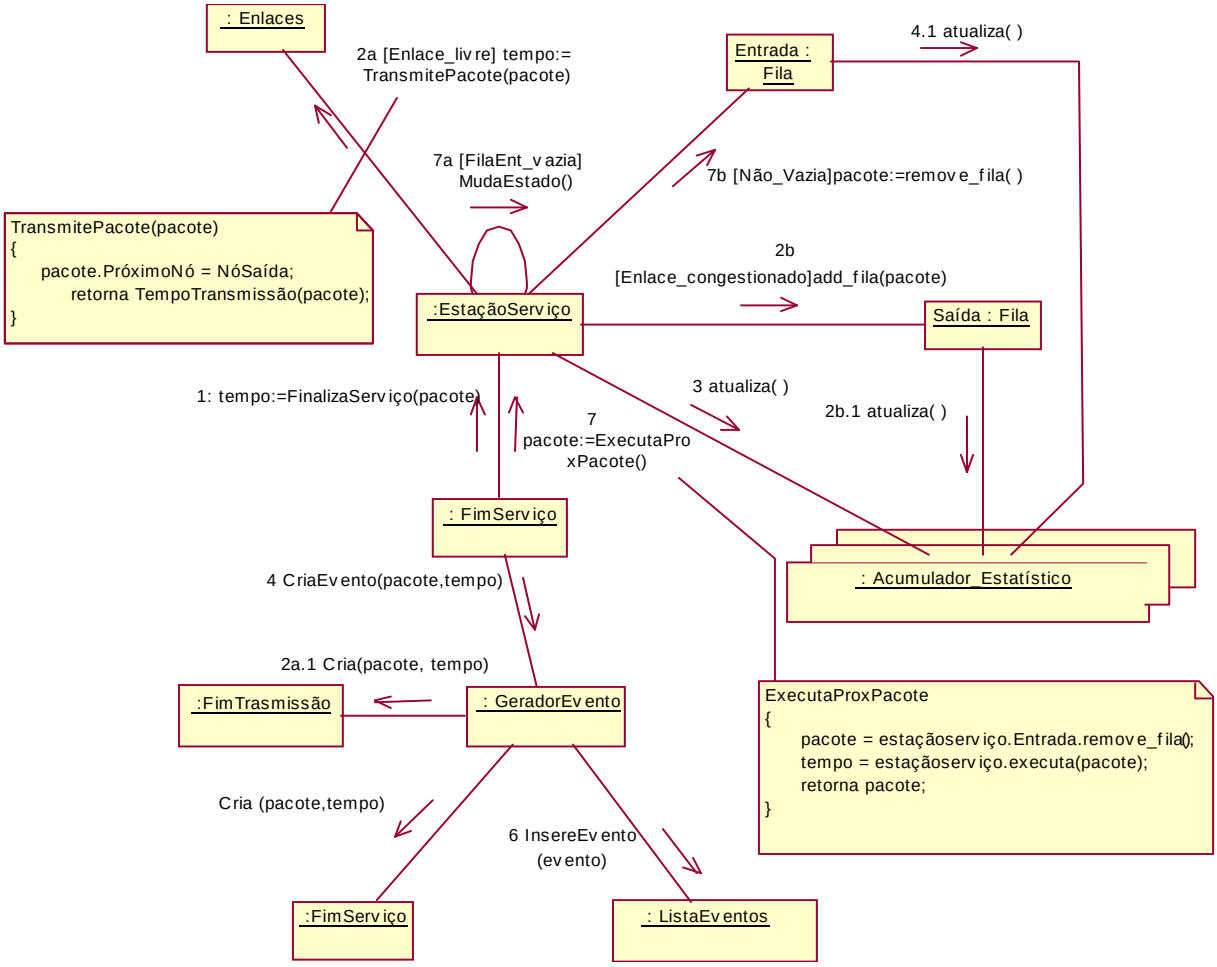

Figura 5-9: Diagrama de Colaboração para o Evento *FimServiço*

Durante esse processo, são atualizados os acumuladores estatísticos.

Quando um evento **FimTransmissão** é escalonado, é chamado o método *TransmiteProxPacote()* do componente **enlace** correspondente. Esse método é responsável por retirar o pacote da **fila de saída**, caso ela contenha pacotes. O pacote retirado é transmitido gerando, consequentemente, um novo evento **FimTransmissão** que é inserido na **ListaEventos**, como pode ser visto na figura 5.10.

Caso a fila esteja vazia, nenhum evento é gerado.

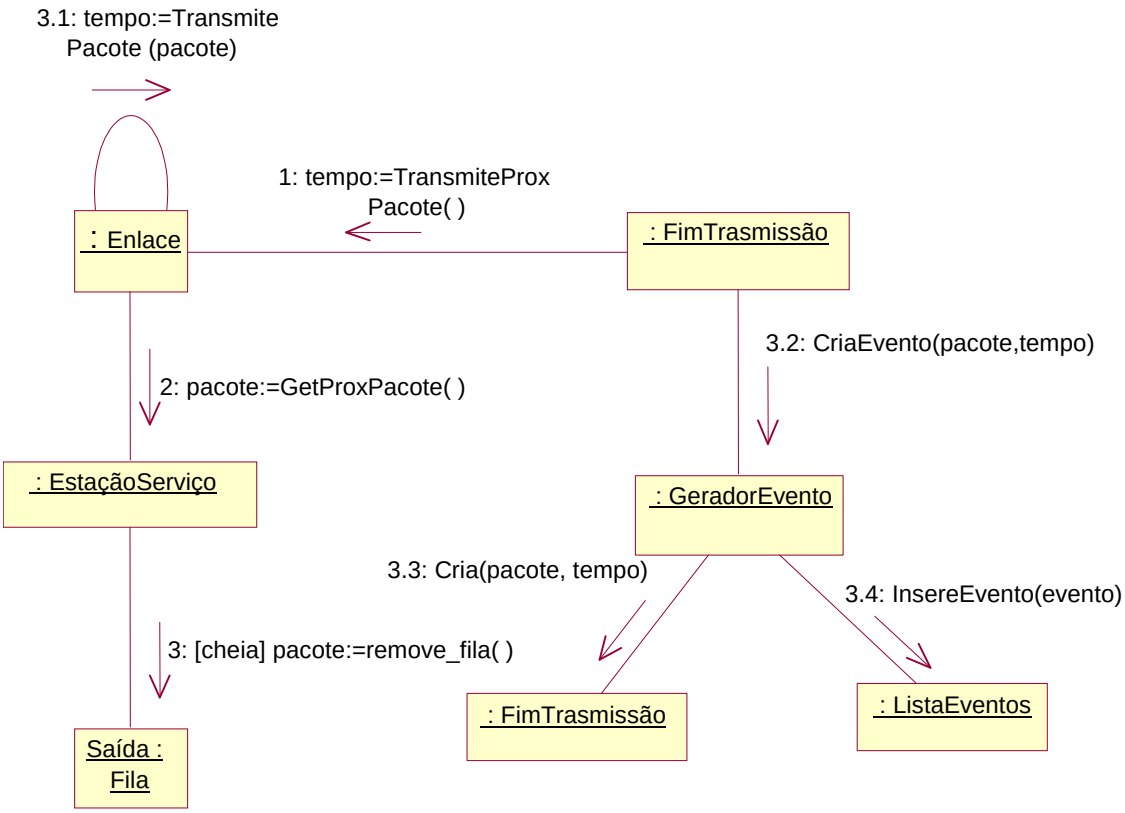

Figura 5-10: Diagrama de Colaboração do Evento *FimTransmissão*

Após a execução, passa-se à fase de finalização da simulação.

5.3.3 Fase de Término de Simulação

Ao final da simulação, o evento fim de simulação é escalonado. Com a execução desse evento, os acumuladores estatísticos são atualizados e é verificado se necessita uma nova replicação (verifica o atributo *NumReplic*), como mostrado na figura 5.11. O **ProcessadorMedidasDesempenho** é acionado para armazenar as estatísticas (médias, máximos, mínimos) sobre as medidas coletadas. As medidas de interesse são armazenadas em arquivo através do componente **InterfaceArquivos**.

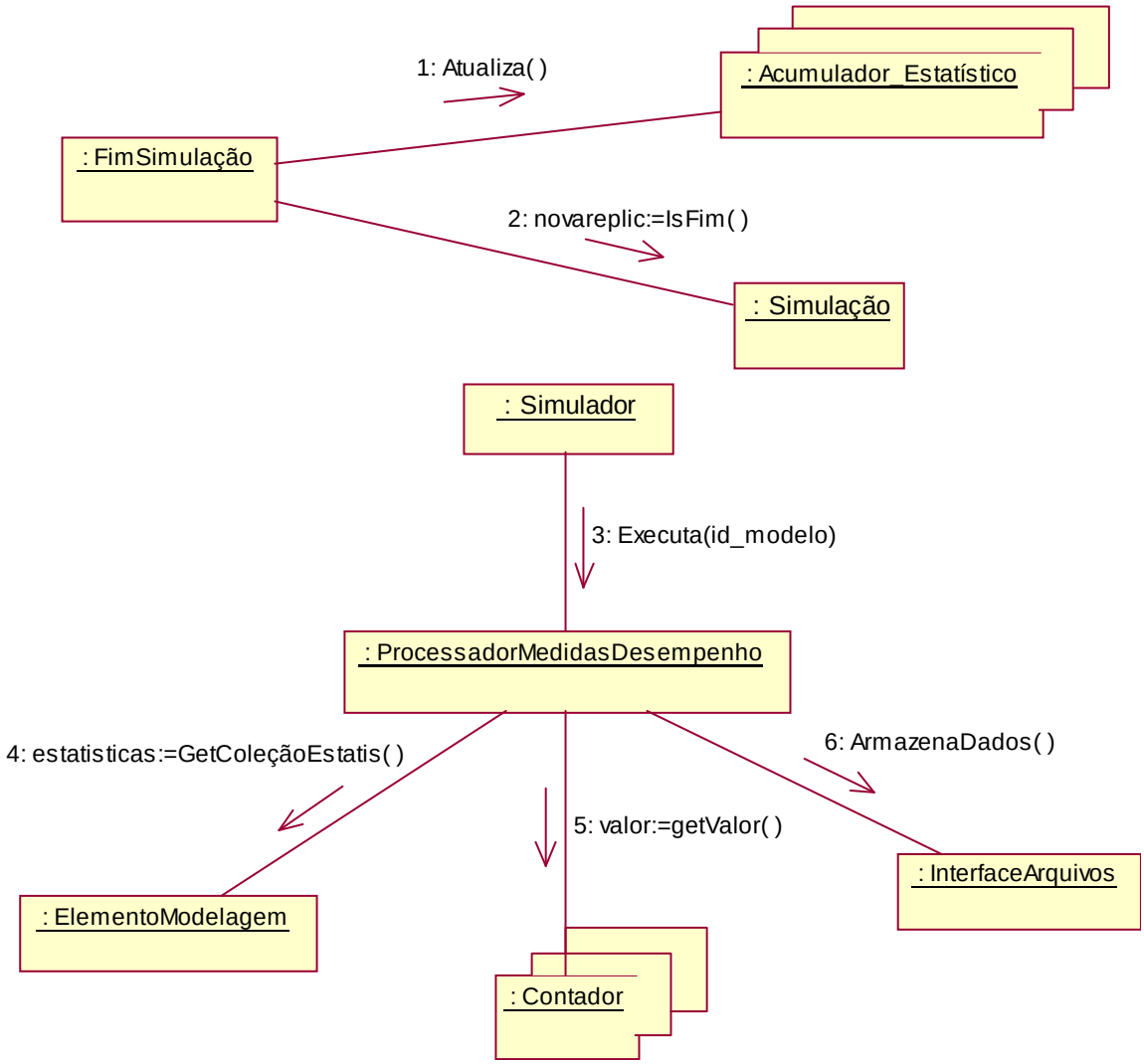

Figura 5-11: Diagrama de Colaboração para o Fim de Simulação

Após a execução da simulação, passa-se a fase de apresentação dos resultados. A apresentação de resultados graficamente poderia ser mais uma responsabilidade do componente **`ProcessadorMedidasDesempenho`**, que poderá ser acrescentada futuramente. Esta fase não será explorada neste projeto, pois está fora do escopo deste. Sendo os resultados armazenados em arquivos de texto, ele poderá ser utilizado por qualquer editor de texto e seus dados podem ser utilizados facilmente para a geração de gráficos por qualquer editor de gráfico.

5.4 Componentes do Simulador

Neste tópico, são descritos os componentes especificados individualmente em termos de seus métodos, atributos e interfaces. Cada componente é visto como uma classe, dentro dos conceitos da linguagem UML.

Como foi dito anteriormente, os elementos de modelagem representam os *hosts*, roteadores, fontes, filas, enlaces e sorvedouros, especificados em [Wagner, 00]. Porém, para o cálculo das medidas de desempenho relevantes ao sistema, necessita-se fazer o levantamento de alguns dados durante a simulação. Para tal, foram criados outros elementos de modelagem, tais como contadores (**`Contador`**) e variáveis estatísticas (**`Acumulador_Estatístico`**). Seus valores podem ser analisados após a simulação para a avaliação de desempenho do modelo.

Os acumuladores estatísticos são variáveis determinadas pelo usuário ou medidas de desempenho "*default*" dos elementos de modelagem (*host*, filas, roteadores, etc.). Exemplos de medidas de desempenho "*default*" são o *NroMesgFila* (Número de Mensagens em Fila), no componente **`Fila`**, e a *Utilização*, no componente **`Roteador`**. Para cada medida, ao final da simulação, são armazenados sua média, o seu número máximo e mínimo e o seu valor final.

Cada medida de desempenho (ou variável determinada pelo usuário) implementa a interface **`Acumulador_Estatístico`**, cujos métodos são mostrados na figura 5.12. Nesta figura são apresentadas somente duas das medidas de desempenho "*default*" dos elementos de modelagem.

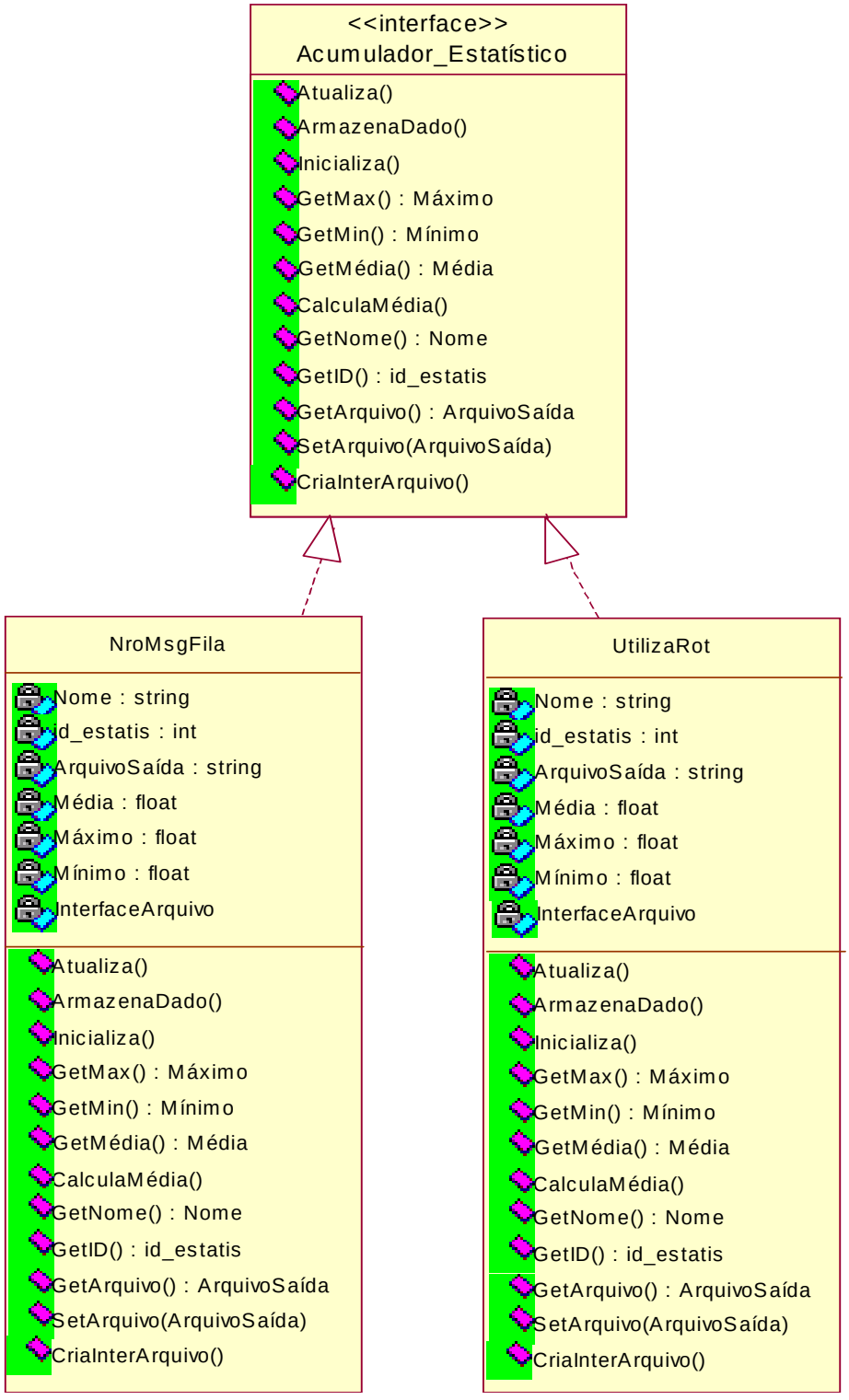

Figura 5-12: Interface Acumulador_Estatístico

São detalhados abaixo os atributos e métodos dos componentes:

Nome – o nome do componente que o identifica.

ArquivoSaída – o nome do arquivo de saída onde serão armazenadas todas as abservações feitas sobre o dado coletado. Este arquivo permite uma melhor avaliação do dado

estatístico. Caso ele esteja vazio, as observações não serão armazenadas e o usuário somente terá conhecimento dos valores extremos e a média.

Média, Máximo e Mínimo – números médio, máximo e mínimo, respectivamente.

InterfaceArquivo – instância do componente **`InterfaceArquivos`** que é responsável pelo interfaciamento com os arquivos usados pelo simulador (abrir, fechar, armazenar, etc.).

Atualiza( ) - método que atualiza o dado a cada observação. Cada atributo ou variável implementa este método de acordo com seu tipo. Quando é uma variável especificada pelo usuário, ele tem que entrar com uma expressão matemática determinada por ele.

ArmazenaDado( ) – armazena o dado no arquivo de saída especificado pelo atributo *ArquivoSaida*.

Inicializa( ) – incializa todos atributos do componente. Ele é utilizado em cada inicialização da simulação.

CalculaMédia( ) – calcula a média.

GetMédia( ) – retorna a média.

GetMax( ) – retorna o número máximo observado.

GetMin( ) – retorna o número mínimo observado.

GetNome( ) – retorna o nome do acumulador.

GetID( ) – retorna o identificador do acumulador.

GetArquivo( ) – retorna o nome do arquivo de saída onde serão armazenados os dados observados durante a simulação.

CriaInterArquivo( ) – cria uma instância do componente **`InterfaceArquivos`**.

Outros tipos de medidas que podem ser coletadas dizem respeitos às mensagens que trafegam pela rede, tais como tempo de permanência no sistema e tempo em fila. Estes são atributos das mensagens e são calculadas por métodos intrínsicos.

Certos dados podem ser necessários, tais como número de mensagens criadas. Esses dados são armazenados em contadores (componente **`Contador`**). Este componente pode ser visto na figura 5.13. Seus métodos e atributos são descritos a seguir. Esse componente implementa a interface **`IContador`**.

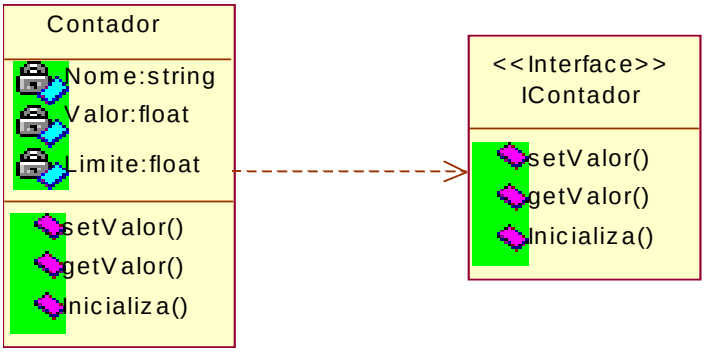

Figura 5-13: Componente Contador

Nome – nome que o identifica.

Valor – valor que é atualizado a cada observação.

Limite – limite que o *Valor* pode assumir

SetValor( ) – atualiza o atributo *Valor*.

GetValor( ) – retorna o atributo *Valor*.

Inicializa( ) – zera o atributo *Valor* a cada inicialização da simulação.

Esses elementos de modelagem são instanciados no componente **`Modelo`**, mostrado na figura 5.14. Esse componente tem como interface **`IModelo`**. Esta interface não é mostrada na figura porém contém os mesmo métodos do componente. Os demais componentes que não apresentarem suas interfaces nas figuras correspondentes assumirão que as mesmas contêm os mesmos métodos mostrados para o componente.

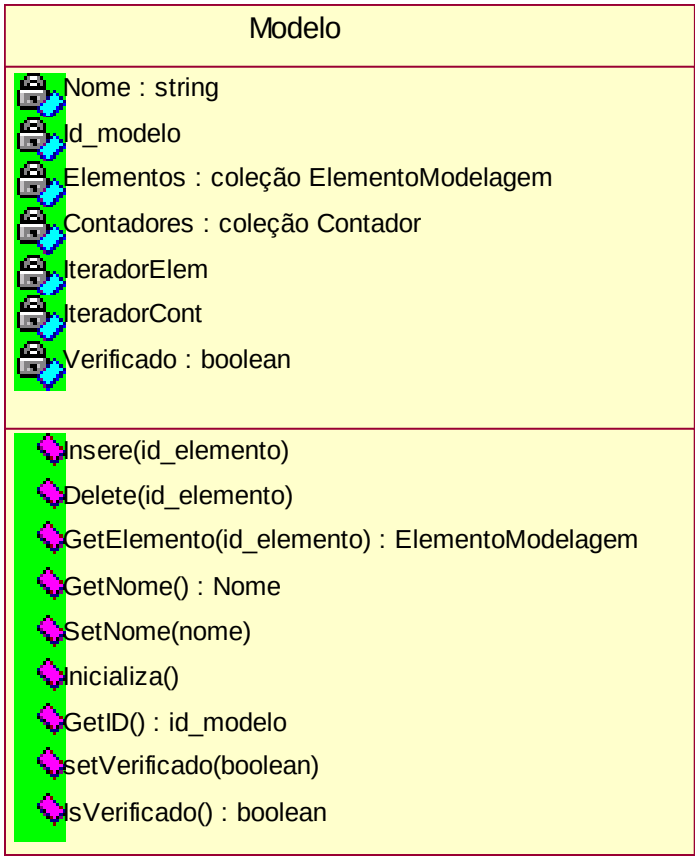

Figura 5-14: Componente Modelo

Seus métodos e atributos são descritos abaixo:

Id_elemento – identificador do modelo.

Nome – nome do componente.

ElementoModelagem – coleção de elementos de modelagem que fazem parte do modelo da rede. Esses elementos são os *host*, fontes, filas, sorvedouros e roteadores, especificados em [Wagner, 00], e os componentes para levantamento de medidas de desempenho especificadas pelo usuário.

Contador – coleção de contadores que fazem parte do modelo.

IteradorElem e *IteradorCont* –variáveis responsáveis pela varredura das coleções de elementos de modelagem e contadores, respectivamente (Padrão *Iterator*). O padrão *Iterator* provê uma forma de acessar sequencialmente os elementos de um objeto agregado (ex: coleção) sem expor sua representação interna [Gamma et all, 95].

Verificado – atributo booleano que indica se o modelo já foi verificado.

GetID( ) – retorna o atributo *id_modelo*.

Insere( ) – insere um novo elemento de modelagem na coleção. O parâmetro *id_elemento* identifica qual elemento está sendo inserido.

GetElemento( ) – retorna um elemento de modelagem da coleção de acordo com o parâmetro *id_elemento*.

Delete( ) – elimina o elemento de modelagem da coleção especificado por *id_elemento*.

Inicializa( ) – inicializa os elementos de modelagem e os contadores inseridos nas coleções. Esse método chama o método *inicializa( )* de cada elemento e contador.

SetVerificado( ) – modifica o atributo *Verificado* para TRUE ou FALSE.

IsVerificado( ) – retorna o atributo *Verificado*.

Os mesmos métodos de criação, inserção e remoção dos elementos de modelagem também existem para a coleção de contadores. Esses métodos utilizam o *iterador* de cada coleção para executarem suas tarefas.

Cada elemento de modelagem implementa a interface **`IElementoModelagem`**, mostrada na figura 5.15. Os atributos do componente **`host`** também estão contidos nos demais componentes, cuja descrição pode ser vista abaixo:

id_elemento – identificador do componente

Nome – nome do componente que o identifica

Estatísticas – coleção de **`Acumulador_Estatístico`**. Cada elemento de modelagem tem associado um conjunto de acumuladores estatísticos para armazenarem as medidas relevantes.

Iterador - variável responsável pela varredura da coleção de acumuladores estatísticos.

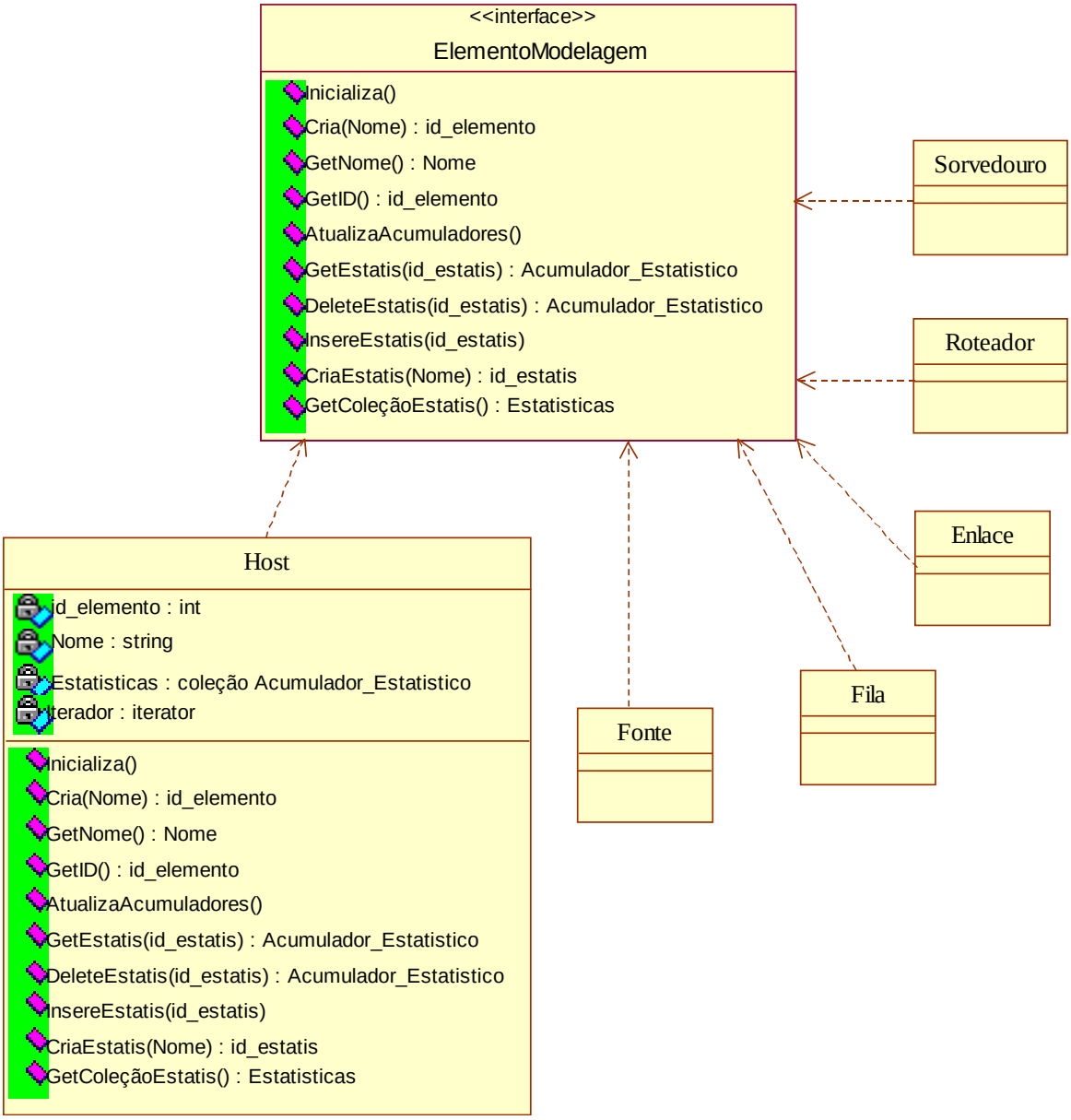

Figura 5-15: Interface ElementoModelagem

Os métodos são descritos a seguir:

Inicializa( ) – inicializa o elemento de modelagem, i.e., inicializa todos os acumuladores inseridos na coleção e seus atributos.

Cria( ) – método que cria um novo elemento de modelagem. Cada tipo de elemento implementa esse método.

GetNome( ) – retorna o atributo *Nome*.

GetID( ) – retorna o atributo *id_elemento*.

AtualizaAcumuladores( ) – atualiza os acumuladores estatísticos inseridos na coleção. Este método é chamado cada vez que o elemento de modelagem é executado.

GetEstatis( ) – retorna um **`Acumulador_Estatístico`** da coleção *Estatísticas* de acordo com o parâmetro *id_elemento*.

DeleteEstatis( ) – deleta um **`Acumulador_Estatístico`** da coleção *Estatísticas* de acordo com o parâmetro *id_elemento*. Este método retorna o acumulador apagado.

InsereEstatis( ) – insere um Acumulador_Estatístico na coleção *Estatísticas*. Este acumulador é identificado pelo parâmetro *id_elemento*.

CriaEstatis( ) – cria um novo **`Acumulador_Estatístico`**. O identificador desse acumulador é retornado (*id_elemento*).

GetColeçãoEstatis( ) – retorna a coleção *Estatisticas*.

Os elementos *host* e roteador também estendem a classe **`EstaçãoServiço`**, que implementa a interface **`IEstaçãoServiço`** (figura 5.16). Esta classe pode ser considerada uma classe de adaptação (padrão de projeto *Adapter* [Gamma et al., 95]) que facilita a implementação da interface **`IEstaçãoServiço`**. A utilização de uma classe adaptadora permite ao programador, ao invés de implementar uma interface por inteiro, simplesmente estender a classe adaptadora, redefinindo apenas os métodos necessários. Essa classe abstrata implementa os métodos para tratar os eventos recebidos deixando a cargo de suas subclasses implementarem o método *Executa()*.

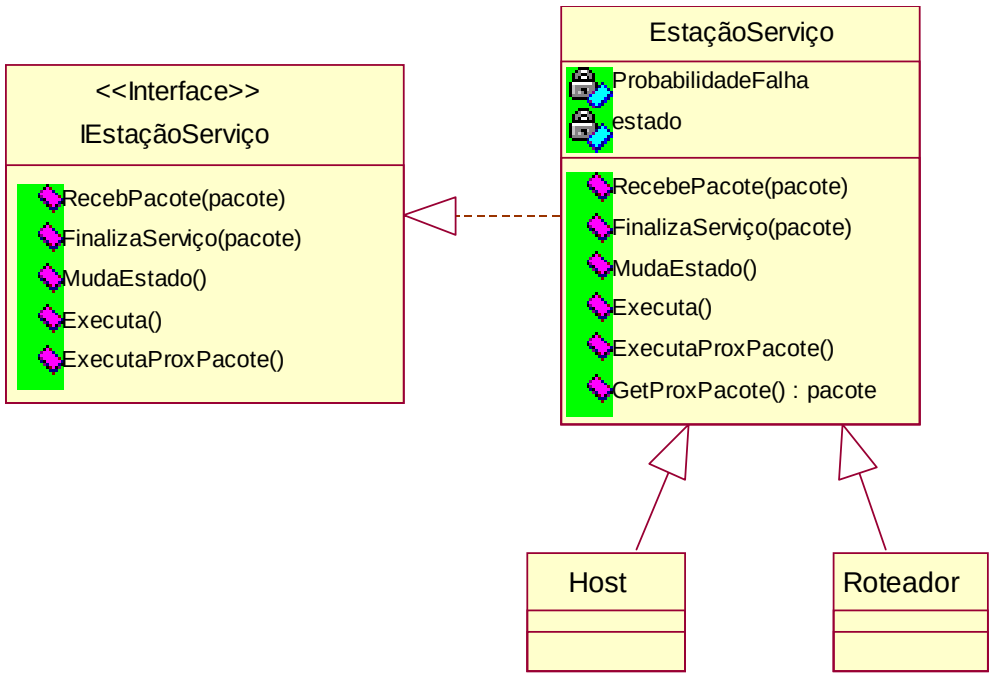

Figura 5-16: Classe EstaçãoServiço

Os métodos e atributos são descritos a seguir:

ProbabilidadeFalha – probabilidade de falha da estação de serviço. Deve ser configurada pelo usuário.

RecebePacote( ) – este método é chamado pelo evento **`ChegaPacote`**. Ele chama o método *Executa()*, caso seu estado esteja **livre**. Caso contrário, ele adiciona o pacote na fila de entrada. Ao final, ele chama o método *AtualizaAcumuladores()*.

FinalizaServiço( ) – método chamado pelo evento **`FimServiço`**.

MudaEstado( ) – método responsável para mudar o estado da **`EstaçãoServiço`**. Ocasionalmente, a estação de serviço pode se tornar desativada (ocorre uma queda na rede), de acordo com a probabilidade de falha da estação de serviço (*ProbabilidadeFalha*).

Executa( ) – é um método abstrato que tem que ser redefinido por suas subclasses.

As fontes necessitam de geradores de números aleatórios para gerarem as mensagens segundo alguma distribuição de probabilidade. As estações de serviço também necessitam desse componente para gerarem seus tempos de serviço. Esses números são gerados por geradores de valores aleatórios, que implementam a interface **`GeradorVAs`**, mostrada na figura 5.17:

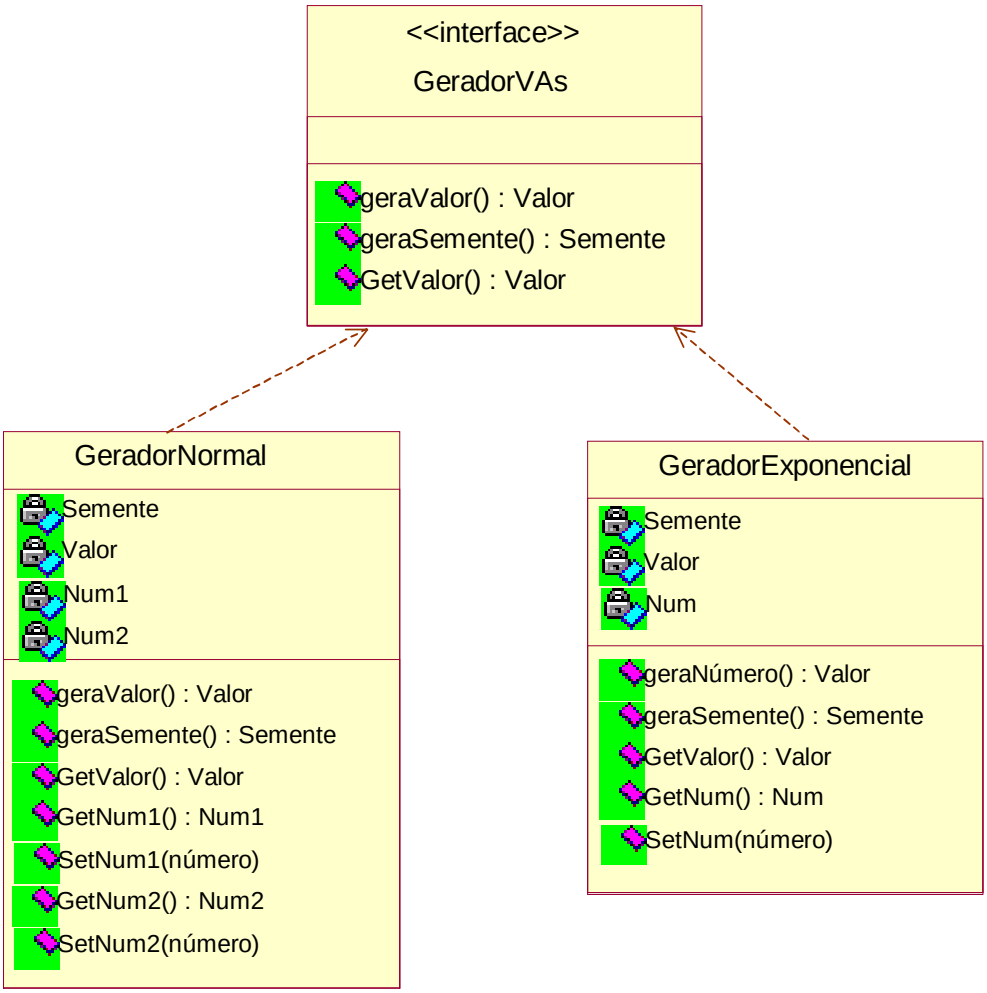

Figura 5-17: Interface GeradorVAs

Seus métodos são descritos a seguir:

geraNúmero( ) – este método é implementado em cada gerador de acordo com sua funcção de distribiução de probabilidade. Ele gera um número aleatório segundo sua função de distribuição.

geraSemente( ) – gera uma semente.

GetNum( ) – retorna o atributo *Num*.

Todos os componentes que implementam essa interface têm dois atributos comuns: *Semente* e *Valor*. A *Semente* representa a semente necessária geração de valores aleatórios de acordo com uma função de distribuição de probabilidade. O *Valor* representa o valor que será gerado pela função de distribuição. Os outros atributos dependem do tipo de função de distribuição de probabilidade. A figura 5.17 somente mostra dois geradores, por efeito de simplificação, mas o simulador deve permitir gerar valores para as principais funções de distribuição de probabilidade.

Os eventos manipulados pelo controlador da simulação estende a classe **`Evento`** que tem como interface **`IEvento`**, como mostrado na figura 5.18. Permite que novos eventos sejam incluídos naturalmente ao simulador, somente implementando o método *Executa()*.

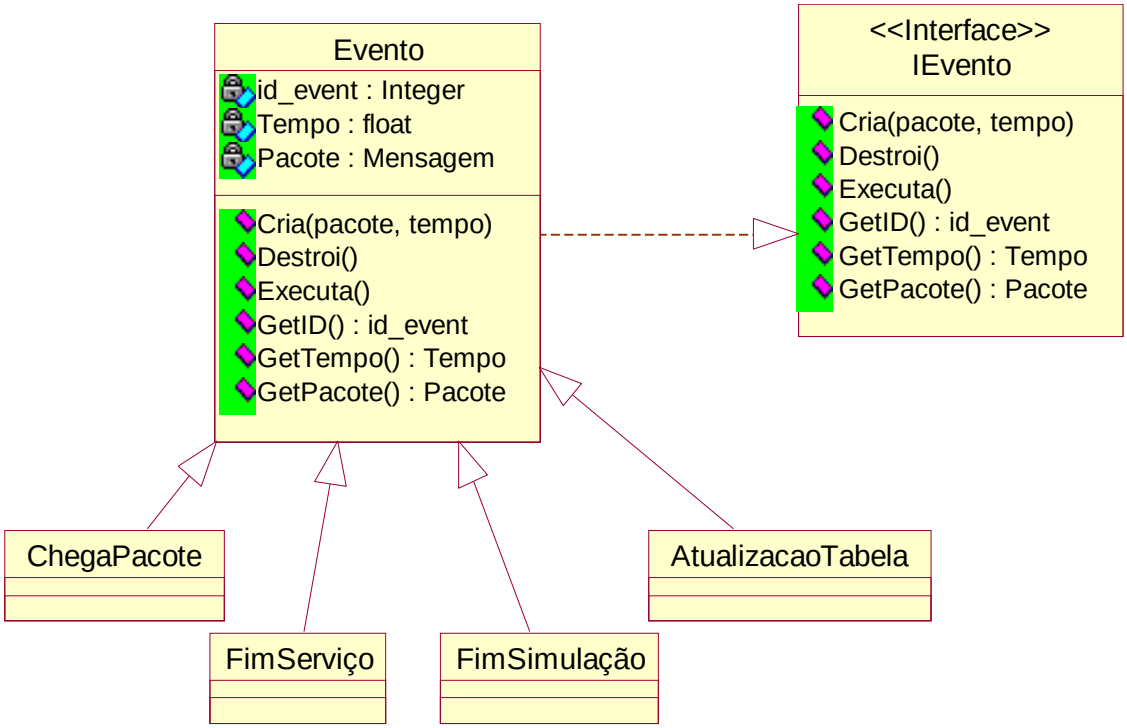

Figura 5-18: Classe Evento

Os atributos e métodos da classe **`Evento`** são descritos a seguir:

id_event – identificador do evento.

Tempo – tempo de ocorrência do evento. Esse tempo é levado em conta na hora de ser inserido na lista de eventos.

Pacote – cada evento está associado a um pacote. Segundo pode ser visto em [Marcao, 00], o componente **`Pacote`** é derivado do componente **`Mensagem`**.

Cria( ) – cria um novo evento. Equivale ao construtor do componente.

Destroi( ) – deleta um evento. Equivale ao destrutor do componente.

Executa( ) – é um método abstrato, implementado diferentemente para cada tipo de evento. Para o evento **`ChegaPacote`**, por exemplo, este método chama o método *RecebePacote()* da **`EstaçãoServiço`** correspondente ao elemento corrente do *Pacote* (como pode ser visto nos diagramas de colaboração da seção anterior).

Os eventos são inseridos numa lista de eventos (**ListaEventos**). O componente **ListaEventos** implementa a interface **EventoListener**.

A interface **EventoListener** contém somente um método (*ProcessaEvento()*) responsável por captar e tratar dos eventos gerados pelo **GeradorEvento**. Todos os componentes interessados nos eventos gerados devem implementar essa interface (figura 5.19). No simulador, somente o componente **ListaEventos** é interessado nos eventos gerados.

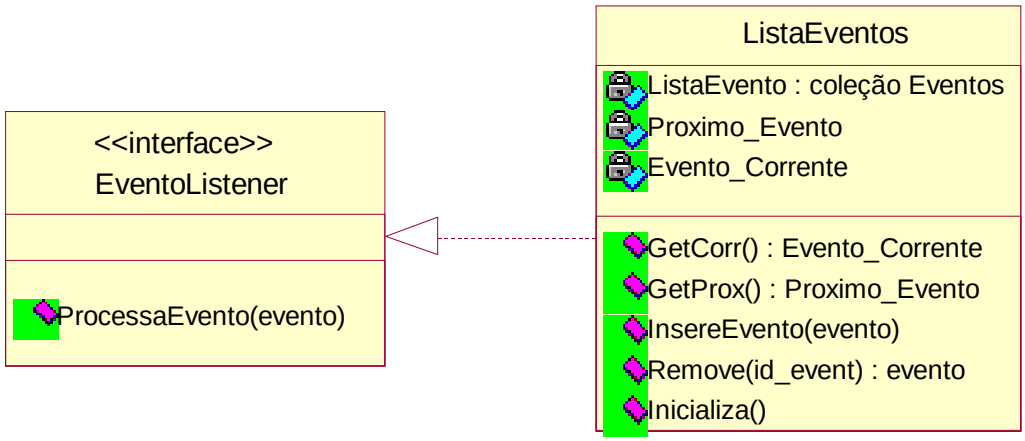

Figura 5-19: Componente ListaEventos

Os atributos e métodos da classe **ListaEventos** são descritos abaixo:

Proximo_Evento – corresponde ao próximo evento a ser escalonado.

ListaEvento – coleção de Evento a serem escalonados pelo simulador.

GetCorr( ) – retorna o Evento corrente.

GetProx( ) – retorna o *Proximo_Evento*.

Inicializa( ) – inicializa a lista de eventos, isto é, insere os eventos iniciais e retira os que poderiam estar presentes em alguma simulação anterior, antes do início da simulação.

InsereEvento( ) – insere um Evento, passado como parâmetro, na coleção *ListaEvento*. A inserção do evento na coleção é feita de acordo com o tempo de cada **Evento**.

Remove( ) – remove um **Evento** da coleção *ListaEvento*.

A inserção de eventos na **ListaEventos** é feita usando o padrão *Observer*. Os eventos são gerados pelos componentes que implementam a interface **GeradorEvento**. Esses componentes disparam os eventos gerados que são captados pelos *listeners* cadastrados neles (**ListaEventos**). A interface **GeradorEvento** é mostrada na figura 5.20. Para cada

tipo de evento, existe um componente *gerador*. Aqui é utilizado o padrão de projeto *Factory Method* [Gamma et al., 95]. A responsabiblidade de criação de cada tipo de evento é delegada às subclasses de **`GeradorEvento`**, através do método *CriaEvento()*.

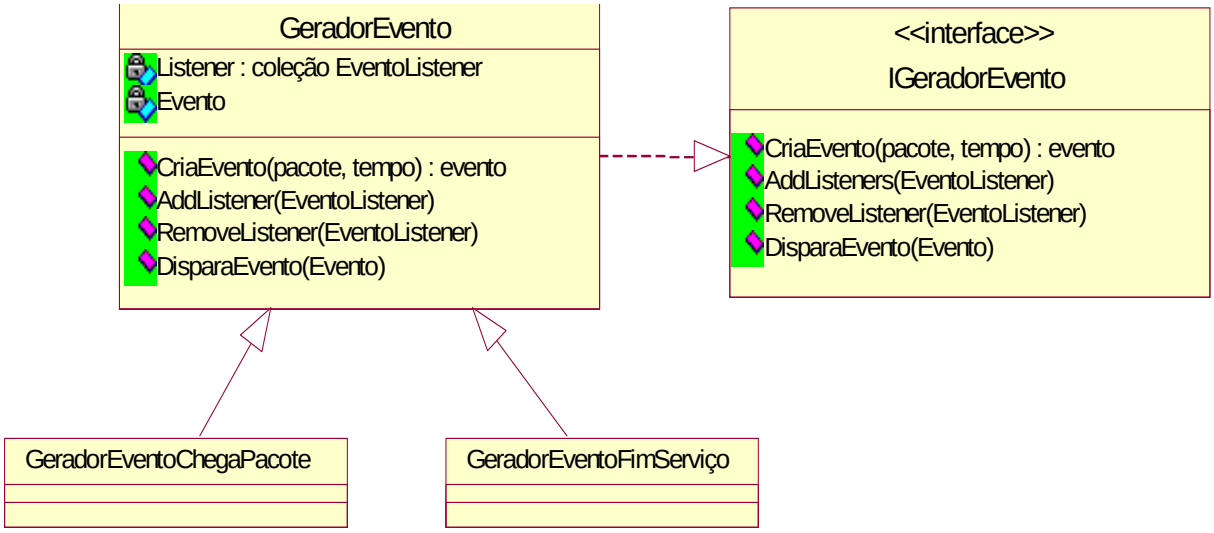

Figura 5-20: Classe GeradorEvento

Seus métodos e atributos são descritos a seguir:

Listeners – coleção de "consumidores" cadastrados no componente. No caso, o único interessado nos eventos gerados é o **`EventoListerner`**.

Evento – evento gerado.

CriaEvento( ) – cria um novo evento.

AddListener( ) – cadastra um novo *listener* no **`GeradorEvento`**.

RemoveListener( ) – remove um *listener* cadastrado.

DisparaEvento( ) – notifica os *listeners* cadastrados sobre o evento gerado.

O componente responsável por escalonar e executar eventos da lista de eventos é o **`ControlaSimulação`**. Ele executa o algoritmo de simulação, e está mostrado na figura 5.21.

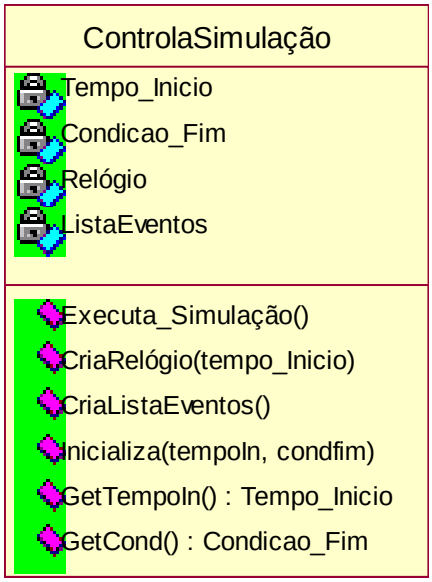

Figura 5-21: Componente ControlaSimulação

Seus atributos e métodos são descritos a seguir:

Tempo_Inicio – indica o tempo inicial da simulação, fornecido pelo usuário.

Condição_Fim – indica a condição de término da simulação (por tempo, por número de pacotes, etc.)

Relógio – Instância do componente **`Relógio`**.

ListaEventos – Instância do componente **`ListaEventos`**.

Executa_Simulação( ) – método que executa o algoritmo de simulação.

CriaRelógio( ) - cria um componente **`Relógio`**. Só vai existir um único relógio na simulação.

CriaListaEventos( ) – cria um componente **`ListaEventos`**. Só vai existir uma única lista de eventos na simulação.

Inicializa( ) – inicializa o relógio e a lista de eventos.

GetTempoIn( ) – retorna o tempo inicial (*Tempo_Inicio*).

GetCond( ) – retorna a condição de término da simulação (*Condição_Fim*).

O componente responsável pelo avanço do tempo simulado é o **`Relógio`** (figura 5.22), cujos atributos e métodos são descritos abaixo.

Figura 5-22: Componente Relógio

Tempo_Corrente – indica o tempo corrente da simulação.

Inicializa( ) – inicializa o *Tempo_Corrente* com o valor zero.

Atualiza( ) – atualiza o *Tempo_Corrente* com o valor passado como parâmetro.

GetTempoCorr( ) – retorna o atributo *Tempo_Corrente*.

Cada modelo é associado a uma simulação que deve ser configurada pelo usuário e que deve acionar o componente **`ControlaSimulação`** para o início da execução da simulação. O componente **`Simulação`** é mostrado na figura 5.23.

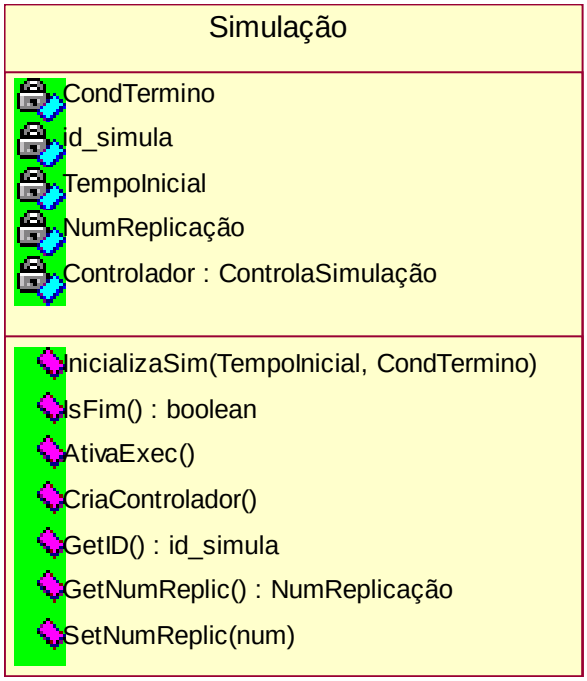

Figura 5-23: Componente Simulação

Seus métodos e atributos são descritos a seguir:

CondTermino – indica a condição de término definido pelo usuário.

TempoInicial – indica o tempo inicial da simulação, definido pelo usuário.

NumReplicação – indica o número de replicações para a simulação, definido pelo usuário.

id_simula – identificador da simulação.

Controlador – instância do componente **`Controlador`**.

ModeloSim – instância do componente **`Modelo`**.

InicializaSim( ) – inicia uma nova simulação, de acordo com o diagrama de colaboração mostrado na seção anterior.

IsFim( ) – chamado ao final da simulação. Verifica se há uma nova replicação da simulação. Caso o número de replicação (*NumReplicação*) seja maior que 1, retorna FALSE e a simulação é reiniciada e reativada. Senão, retorna TRUE. Antes de nova execução, o componente **`ProcessadorMedidaDesempenho`** é ativado para armazenar os dados coletados durante a simulação.

AtivaExec( ) – ativa a execução da simulação, que consiste em chamar o método *Executa_Simulação()* do componente **`ControlaSimulação`**.

CriaControlador( ) – cria uma instância do componente **`ControlaSimulação`**. Esse método é chamado na inicialização da simulação.

GetNumReplic( ) – retorna o número de replicações (*NumReplicação*) de simulações do modelo.

SetNumReplic( ) – modifica o número de replicações (*NumReplicação*) de simulações do modelo.

GetID( ) – retorna o atributo *id_simula*.

Ao final da simulação, o componente **`ProcessadorMedidasDesempenho`** é acionado pelo componente **`Simulador`**. Ele é mostrado na figura 5.24.

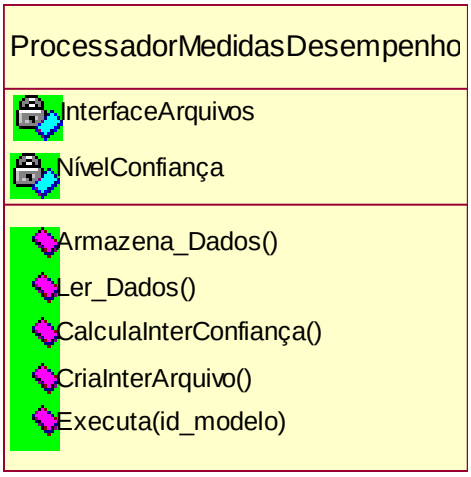

Figura 5-24: Componente ProcessadorMedidasDesempenho

Seus atributos e métodos são descritos abaixo:

InterfaceArquivo – instância do componente InterfaceArquivos. Faz o interfaceamento entre o simulador e os arquivos utilizados.

NívelConfiança – nível de confiança para o cálculo dos intervalos de confiança. Definido pelo usuário.

Armazena_Dados( ) – armazena os dados no arquivo. Ele chama o método de armazenamento do *InterfaceArquivo*. Os dados são obtidos dos acumuladores estatísticos dos elementos de modelagem, dos contadores e dos atributos dos pacotes.

Ler_Dados( ) – lê dados do arquivo.

CalculaInterConfiança( ) – calcula os intervalos de confiança dos dados armazenados para as replicações.

CriaInterArquivo( ) – cria uma instância do componente **`InterfaceArquivos`**.

Executa( ) – este método é chamado pelo **`Simulador`**, após a execução da simulação. Ele é responsável por chamar os métodos descritos acima.

O componente **`Simulador`** faz a interface entre o ambiente gráfico e os componentes da lógica da aplicação (ver projeto arquitetural). Através deste componente é construído o modelo a ser simulado. Ele é responsável pela ativação da execução da simulação como também por acionar o componente **`ProcessadorMedidasDesempenho`** e **`VerificadorConsistência`**. Ele é mostrado na figura 5.25.

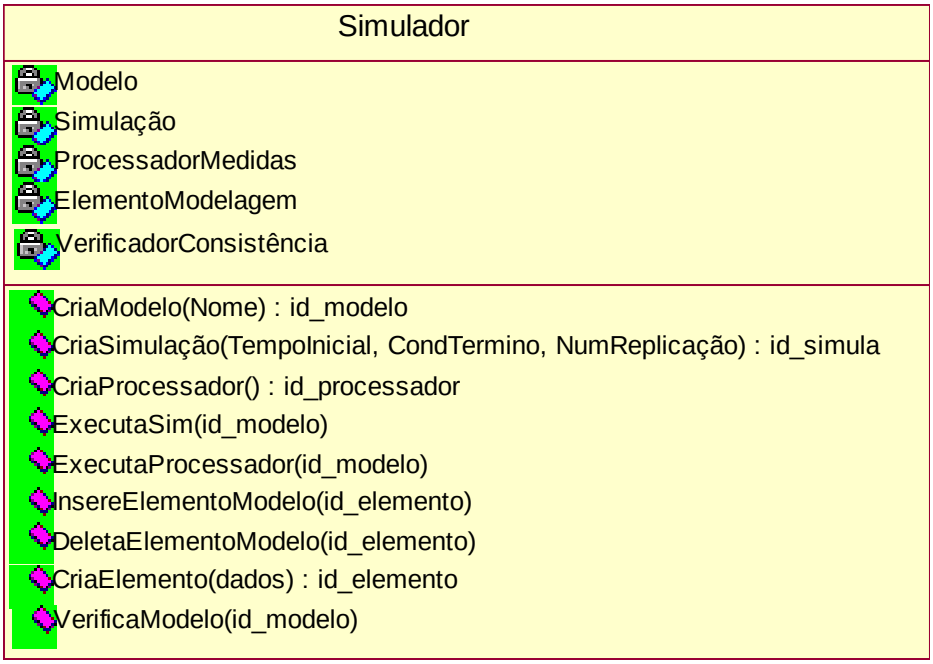

Figura 5-25: Componente Simulador

Seus atributos e métodos são descritos a seguir:

Modelo – instância do componente **`Modelo`**.

Simulação – instância do componente **`Simulação`**.

ProcessadorMedidas – instância do **`ProcessadorMedidasDesempenho`**.

ElementoModelagem –corresponde a um elemento de modelagem.

VerificadorConsistência - instância do componente **`VerificadorConsistência`**.

CriaModelo( ) – cria uma instância do componente **`Modelo`**. Esse método é acionado por um comando da interface gráfica do simulador (obs: caso o modelo já exista, ele pode ser "carregado" pelo sistema).

CriaSimulação( ) – cria uma nova instância do componente **`Simulação`**.

CriaProcessador( ) – cria uma nova instância do componente **`ProcessadorMedidasDesempenho`**.

ExecutaSim( ) – chama o método *AtivaExec()* do componente **`Simulação`**.

ExecutaProcessador( ) – chama o método *Executa()* do processador de medidas.

InsereElementoModelo( ) - insere um novo elemento de modelagem no modelo (chamando o método *insere()* do componente **`Modelo`**).

DeletaElementoModelo( ) – elimina um elemento de modelagem do modelo (chamando o método *Delete()* do componente **`Modelo`**).

CriaElemento( ) – cria uma nova instância de um elemento de modelagem.

VerificaModelo( ) – verifica a consistência do modelo antes da execução da simulação. Para isso ele chama os métodos de verificação do componente **`VerificaConsistência`**.

Os dados são armazenados em arquivos ao final da simulação. O componente responsável pela manipulação desses arquivos é mostrado na figura 5.26 e obedece a interface **`InterfaceArquivos`**.

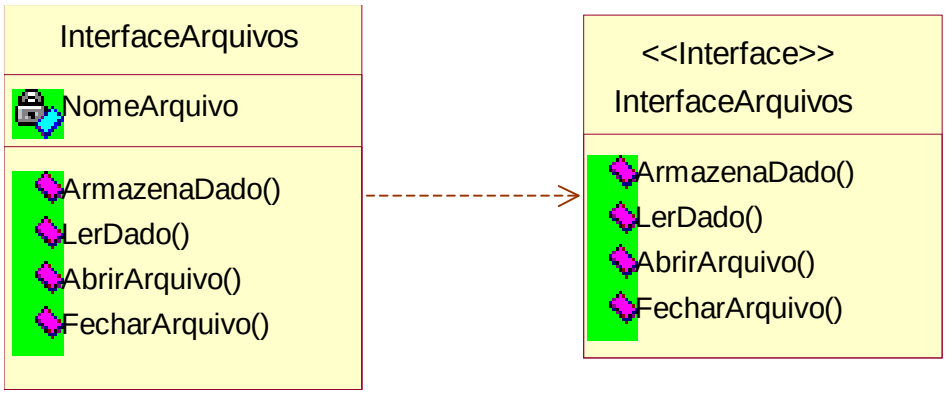

Figura 5-26: Componente InterfaceArquivos

Seus atributos e métodos são mostrados a seguir:

NomeArquivo – nome do arquivo onde serão armazenados os dados.

ArmazenaDado( ) – armazena os dados no arquivo.

LerDado( ) – ler os dados do arquivo.

AbrirArquivo( ) – abre ou cria um novo arquivo.

FecharArquivo( ) – fecha um arquivo.

O sistema deve permitir descarte de pacotes. Para realizar essa tarefa, foi especificado o componente **`Depósito`**, mostrado na figura 5.27.

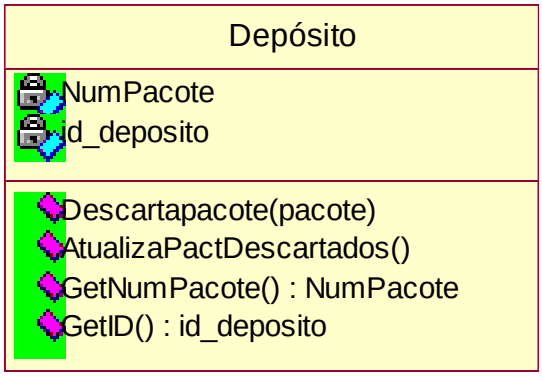

Figura 5-27: Componente Depósito

Seus métodos e atributos são descritos a seguir:

NumPacote – número de pacotes descartados.

Id_deposito – identificador do componente **`Depósito`**.

DescartaPacote( ) – deleta o pacote passado como parâmetro e chama o método *AtualizaPactDescartados()*.

AtualizaPactDescartados( ) – atualiza o número de pacotes descartados (*NumPacote*).

GetNumPacote( ) – retorna o atributo *NumPacote*.

GetID( ) – retorna o identificador do depósito.

Antes da execução da simulação o modelo precisa ser verificado. Essa verificação é realizada pelo componente VerificadorConsistência, mostrado na figura 5.28.

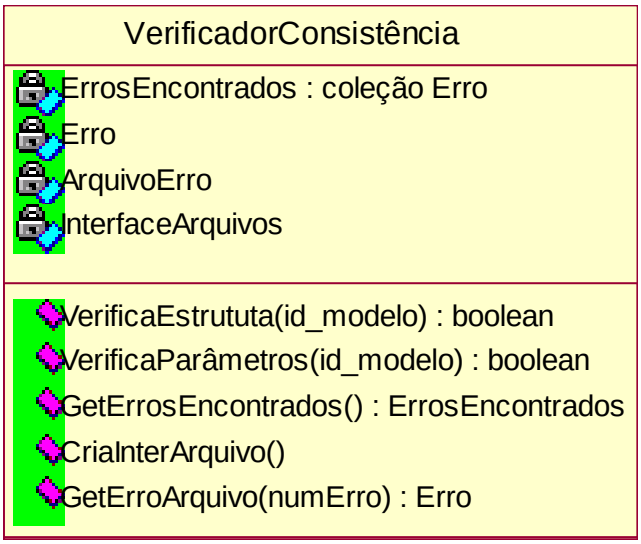

Figura 5-28: Componente VerificadorConsistência

Seus atributos e métodos são descritos a seguir:

ErrosEncontrados – coleção contendo as estruturas *Erro*.

Erro – é uma estrutura contendo o número do erro e a mensagem a ser transmitida ao usuário.

ArquivoErro – nome do arquivo contendo os possíveis erros. Os erros são encontrados a partir de seu número, e cada número é associado a uma mensagem.

InterfaceArquivos – instância do componente **`InterfaceArquivos`** para manipular o arquivo contendo os erros.

VerificaEstrutura( ) – verifica a consistência da estrutura do modelo. Essa verificação encontra o número do erro ocorrido, atualiza a estrutura *Erro* com o número e a mensagem a ser transmitida (obtida do *ArquivoErro*) e a insere na coleção *ErrosEncontrados*.

VerificaParâmetros( ) – verifica a consistência dos parâmetros da simulação e elementos de modelagem. Essa verificação encontra o número do erro ocorrido, atualiza a estrutura *Erro* com o número e a mensagem a ser transmitida (obtida do *ArquivoErro*) e a insere na coleção *ErrosEncontrados*.

GetErrosEncontrados( ) – retorna os erros encontrados durante a verificação do modelo (coleção *ErrosEncontrados*).

CriaInterArquivo( ) – cria uma instância do componente **`InterfaceArquivo`**.

GetErroArquivo( ) – retorna um erro do *ArquivoErro* de acordo com seu número.

Após a especificação dos componentes, eles devem ser implementados usando alguma linguagem de programação adequada. Uma vez que os componentes sejam implementados e estejam finalmente disponíveis para uso, a aplicação pode ser "composta" visualmente com a ajuda de algum editor gráfico. Este editor gráfico deve permitir a configuração dos componentes instanciados e a interconexão entre eles para criar uma aplicação.

A tabela 5.1 lista todos os componentes, interfaces e classes especificados acima.

Nome	Descrição
Acumulador_Estatístico	**Interface** responsável por armazenar dados coletados durante a simulação. Todas as medidas de desempenho devem implementar essa interface.

Contador	**Componente** utilizado para contagens de dados, tipo número de pacotes descartados, criados, etc.
Modelo	**Componente** que contém os elementos de modelagem e os contadores criados pelo usuário.
ElementoModelagem	**Interface**: todos os elementos de modelagem especificados em [Wagner, 00] devem implementar essa interface.
EstaçãoServiço	**Classe abstrata** que implementa a interface `IEstaçãoServiço`. Os componentes **`Host`** e **`Roteador`** estendem essa classe.
GeradorVAs	**Interface** responsável pela geração de valores aleatórios para os elementos de modelagem que utilizam funções de distribuição de probabilidade para executarem.
Evento	**Classe abstrata** que implementa a interface **`IEvento`**. Os componentes **`ChegaPacote`**, **`FimServiço`**, **`AtualizaPacote`** e **`FimSimulação`** estendem essa classe. Representa um evento gerado durante a simulação.
ListaEventos	**Componente** responsável por armazenar os eventos ocorridos durante a simulação. Ele implementa a interface **`EventoListener`**.
GeradorEvento	**Classe abstrata** responsável pela criação de um evento durante a simulação. Ela implementa a interface **`IGeradorEvento`**. Os componentes **`GeradorChegaPacote`**, **`GeradorFimServiço`**, **`GeradorFimSimulação`** e **`GeradorAtualTabela`** estendem essa classe.
ControlaSimulação	**Componente** responsável pela execução da simulação (escalonamento de eventos, avanço do relógio, etc.).
Relógio	**Componente** responsável pelo armazenamento do tempo simulado.
Simulação	**Componente** que representa uma simulação do modelo. Uma simulação pode ter várias replicações, mudando somente as sementes utilizadas nos processos estocásticos.
Simulador	**Componente** que representa a interface entre os componentes da lógica da aplicação e a interface gráfica do simulador. Ele contém uma referência ao modelo utilizado e sua simulação.

ProcessadorMedidaDesempenho	**Componente** responsável pelo armazenamento em arquivos dos acumuladores estatísticos (e seus cálculos) e os contadores.
VerificadorConsistência	**Componente** responsável por verificar a consistência do modelo.
Depósito	**Componente** que permite o descarte de pacotes durante a simulação. Ele armazena a quantidade de pacotes descartados.
InterfaceArquivos	**Componente** que representa a interface entre os componentes da aplicação e os arquivos utilizados pelo simulador.

Tabela 5-1: Lista de Componentes, Classes e Interfaces Especificados

Cada componente possui pelo menos uma classe, cujo nome é igual ao componente ao qual pertence e cuja responsabilidade é implementar sua funcionalidade básica. Para isto, esta classe pode estender outras classes e pode implementar algumas interfaces. Por exemplo, o componente **`GeradorChegaPacote`** é composto das classes **`GeradorChegaPacote`** e **`GeradorEvento`**, e tem como interface **`IGeradorEvento`**. A classe **`GeradorChegaPacote`**, por sua vez, é uma estensão da classe **`GeradorEvento`**.

5.5 Validação do Projeto

Existem várias formas de validação para uma especificação: formais e não-formais. Os métodos formais de validação de especificação, apesar de mais completos, são complexos e demandam muito tempo. Portanto, nesta Dissertação a validação é feita informalmente. Essa validação foi realizada verificando se os requisitos inicialmente levantados foram atendidos. A tabela abaixo mostra como os requisitos funcionais foram atendidos pela solução proposta.

Requisito	Componentes Relacionados
F1	Os componentes especificados por [Wagner, 00] devem implementar a interface **`ElementoModelagem`** para poderem ser utilizados.
F2	Os componentes **`Contador`** e **`Acumulador_Estatístico`** coletam os dados relevantes para o cálculo de medidas de desempenho. O componente **`ProcessadorMedidasDesempenho`** calcula os intervalos de confiança, caso necessário, e armazena as medidas em arquivos.
F3	O componente **`Acumulador_Estatístico`** pode ser configurado pelo usuário para a coleta de dados definidos por ele.
F4	O escalonamento dos eventos, representados pelos componentes estendidos da classe **`Evento`**, é solicitado pelo componente **`ControlaSimulação`**. Os eventos são armazenados pelo componente **`ListaEventos`**.
F6 e F7	Os parâmetros iniciais dos componentes durante a construção do modelo

	(componente **Modelo**) e inicialização da simulação (componente **Simulação**) são fornecidos pelo usuário através do componente **Simulador**.
F8	Durante a configuração da simulação, o usuário informa a quantidade de replicações da simulação. O **ProcessadorMedidasDesempenho** calcula os intervalos de confiança.
F9	A consistência do modelo é realizada pelo componente **VerificadorConsistência**.
F10	Os valores aleatórios são gerados pelos componentes que implementam a interface **GeradorVAs**.
F11	O tempo simulado é controlado pelo componente **Relógio**.
F12	O componente **Depósito** possibilita o descarte de pacotes durante a simulação.
F13	O componente **Acumulador_Estatístico** permite armazenar todos os dados observados por ele durante a execução da simulação em arquivo, para uma análise mais detalhada.

Tabela 5-2: Validação dos Requisitos Funcionais

Os requistos não funcionais que foram levantados durante a primeira fase da especificação são verificados de acordo com a tabela 5.3.

Requisito	Verificação
NF1	A construção de aplicações baseada em componentes permite o desenvolvimento rápido e de qualidade. Portanto, a utilização dos componentes aqui especificados permite o desenvolvimento mais rápido de ferramentas de simulação.
NF2	Este requisito pode ser verificado com a utilização das interfaces e classes abstratas, como por exemplo **ElementoModelagem**, **GeradorEventos** e **Evento**, a partir das quais novas funcionalidades podem ser criadas sem muito esforço do programador.
NF3	Os componentes aqui especificados são comuns a qualquer ambiente de simulação discreta orientada a eventos pois abrangem suas principais entidades, como, por exemplo, **Relógio**, **GeradorVAs** e **ListaEventos**.
NF4	Os componentes especificados podem ser implementados em qualquer linguagem de programação que suporte o conceito de componentes, como exemplo o *JavaBeans*. Portanto, eles são portáveis.
NF5	Toda a especificação é feita utilizando a UML, utilizando seus principais diagramas (*use-cases*, conceitos, colaboração, etc).

Tabela 5-3: Validação dos Requisitos Não-Funcionais

Capítulo 6

6 Conclusões

Neste trabalho foram especificados componentes de software que facilitam o desenvolvimento de uma ferramenta de simulação voltada para modelagem e análise de redes de computadores com a tecnologia TCP/IP. Estes componentes fornecem o funcionamento básico de uma simulação orientada a eventos, tais como controle do relógio simulado, geradores de valores aleatórios, escalonamento de eventos e coleta de dados para o cálculo de medidas de desempenho. Apesar do objetivo da especificação visar a simulação de redes TCP/IP, os componentes aqui especificados permitem a simulação de qualquer modelo que represente sistemas discretos. Essa especificação foi realizada utilizando a Linguagem de Modelagem Unificada (UML) seguindo o processo de desenvolvimento apresentado em [Larman, 98].

Foram levantados requisitos que visam oferecer as principais características inerentes a uma simulação orientada a eventos. Diante dos requisitos levantados, foram propostos componentes que permitem a reutilização de análise e de projeto no desenvolvimento de componentes voltados para a construção de ferramentas de simulação. Na fase de análise foram identificados e descritos os principais componentes que estão

diretamente relacionados ao domínio do problema e as relações entre eles. Na fase de projeto, foram tomadas as principais decisões de projeto associadas ao domínio do problema - ela define os componentes em termo de seus métodos, atributos e interfaces, além de conter alguns algoritmos abstratos.

Portanto, o desenvolvedor da ferramenta de simulação poderá utilizar essa especificação de forma natural para implementar os componentes em uma linguagem de programação desejada. Os componentes disponíveis podem ser graficamente manipulados através de uma ferramenta de composição visual de aplicações.

Nesta Dissertação, teve-se o cuidado de tratar aspectos relacionados a reusabilidade em todo as fases da especificação, elaborando uma documentação rica em figuras, diagramas, tabelas e narrativas explicativas, como sugere [D´Souza & Wills, 99].

Enfim, ao propor esses componentes, acredita-se ter contribuído efetivamente para o desenvolvimento rápido e eficiente de ferramentas de simulação. Os componentes especificados atenderam aos requisitos levantados, viabilizando os objetivos desta Dissertação.

6.1 Trabalhos Futuros

Uma continuadade natural desta Dissertação é a implementação e teste dos componentes aqui especificados e a construção de ferramentas de simulação utilizando estes componentes. Estas ferramentas podem ser construídas visualmente usando um editor gráfico. De forma a atender alguns requisitos inerentes aos componentes de software, alguns aspectos precisam ainda ser incluídos, tais como sua forma de persistência e sua representação visual.

Outro trabalho refere-se à especificação e implementação da interface gráfica da ferramenta de simulação. Esse ambiente gráfico facilitaria a construção do modelo e também a análise do comportamento dos elementos de modelagem durante e após a simulação, através de gráficos e outros elementos visuais.

Devido aos componentes propostos serem considerados essenciais a uma simulação, estes podem ser reutilizados na construção de qualquer ferramenta de simulação orientada a eventos, independentemente do tipo de modelo, seja este representando redes de computadores, um sistema de manufatura ou outro sistema discreto com contenção de recursos.

Referências Bibliográficas

[Almeida, 99] Almeida, Marcelo J. S. C., "*ATMLib- Uma biblioteca de classes para construção de Simuladores de Rede ATM*", UFPB,1999.

[Alta Group, 96] "*BONES Designer User's Guide*", Alta Group of Cadence Design System, Inc., 1996.

[C++Sim, 99] "*C++SIM User's Guide, Public Release 1.5*", Univeristy of Newcastle upon Tyne, UK.

[Conceição, 93] Conceição Filho, Haroldo Castro, "*SIM/SAVAD: Um Simulador de Modelos de Redes de Filas*", Dissertação de Mestrado, UFPB, 1993.

[Dias, 92] Dias, Maria Madalena, "*SIMILE – Um Simulador Reutilizável para Avaliação de Desempenho de Redes Locais*", UFPB, 1992.

[D´Souza & Wills, 99] D'Souza, D. F., Wills, A. C., "*Objects, Components, and Frameworks with UML - The Catalysis Approach*", Addison-Wesley, 1999.

[Englander, 97] Englander, Robert, "*Developing JavaBeans*", Ed O'Reilly, Junho 1997.

[Ford et al., 97] Ford, M.; Lew, H. K., Spanier, S., Stevenson, T., "*Internetworking Technologies Handbook*", Cisco Press, 1997, cp15.

[Freire, 00] Freire, Raissa Dantas, "*Especificação de um Framework baseado em Componentes de Software Reutilisáveis para Aplicações de Gerência de Falhas em Redes de Computadores*", Dissertação de Mestrado, UFPB, 2000.

[Gamma et al., 95] Gamma, E., Helm, R., Johnson, R., Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[Geoffrey, 78] Geoffrey, G., "*System Simulation*", 2a edição, New Jersey, Prentice-Hall, 1978.

[Giozza et al., 86] Giozza, W. Ferreira; Beltão, J. A. M.; Sauvé, J.P. e Araújo, J.F.M. de; "*Redes Locais de Computadores: Protocolos de Alto Nível e Avaliação de Desempenho*", McGraw-Hill/Embratel, São Paulo – SP, 1986.

[Horn, 71] Van Hor, R.L., "*Validation of Simulation Results*", Management Science, vol17, 1971.

[Kelton, 98] Kelton, W. D. et al., "*Simulation with ARENA*", McGraw-Hill, 1998.

[Kleinrock, 75] Kleinrock, Leonard, "*Queueing system – Volume I: Theory*", Los Angeles, John Wiley & Sons, 1975.

[Krajnc, 97] Krajnc, Marko, "*Why Component-Oriented Programming?*", http://www.odateam.com/cop/copwhy.html, Jan, 1997.

[Landin & Niklasson, 98] Landin, N., Niklasson, A., "*Development of Object-Oriented Frameworks*", Department of Communication Systems, Lund Institute of Technology, Sweden, 1998.

[Larman, 98] Larman, C., "*Applying UML and Patterns - An Introduction to Object-Oriented Analysis and Design*", Prentice-Hall, 1998.

[Wagner, 00] Wagner, Marcos V. S., "*Especificação de Componentes para Simulação de Redes TCP/IP*", Dissertação de Mestrado, UFPB, 2000.

[McNab, 96] McNab, R. and Howell, F.W.; "*Using Java for Discrete Event Simulation*", Setembro, 1996.

[McNab, 99] McNab, Ross, "*A Guide to the Simjava Package*"; http:// www.dcs.ed.ac.uk/home/hase/simjava, Dezembro, 1999.

[Micheli, 98] Michelli, Milena Pessoa, "*Modelagem e Análise de Desempenho do Método Relógio Adaptativo em Redes ATM*", Dissertação de Mestrado, UFPB, 1998.

[Roberts et al., 98] Roberts, Chell A. and Dessouky, Yasser M., "*An Overview of Object-Oriented Simulation*", SIMULATION, Number 70, pp. 359-368, June, 1998.

[Rofail & Shohoud, 1999] Rofail, A., Shohoud, Y., "*Mastering COM and COM+*", Fourth Edition, Sybex, 1999.

[Rumbaugh et al., 99] Rumbaugh, J., Booch, G., Jacobson, I., "*The Unified Modeling Language Reference Manual*", Addison-Wesley, 1999.

[Rumbaugh et al., 99a] Rumbaugh, J., Booch, G., Jacobson, I., "*The Unified Software Development Process*" Addison-Wesley, 1999.

[Siegel, 96] Siegel, J., "*Corba Fundamentals and Programming*", John Wiley & Sons, 1996.

[Smith, 00] Smith, Roger D., "*Simulation*", Encyclopedia of Computer Science, Groves Dictionaries, Julho 2000.

[Soares, 90] Soares, Luiz Fernando G., "*Modelagem e Simulação Discreta de Sistemas*", VII Escola de Computação – São Paulo, 1990.

[Soares, 95] Soares, Luiz Fernando; Lemos, Guido; Colcher, Sérgio. "*Redes de Computadores: das LANs, MANs e WANs às Redes ATM*". Editora Campus, 1995.

[Souto, 93] Souto, Francisco de Assis Coutinho, "*SAVAD – Sistema de Avaliação de Desempenho de Modelos de Redes de Fila*", Dissertação de Mestrado, UFPB, 1993.

[Sun Microsystems, 99] Sun Microsystems Inc., "*Java Management Extensions – SNMP Manager API*", August, 1999. Draft 2.0.

[Szyperski, 99] Szyperski, Clemens, "*Component Software Beyond Object-Oriented Programming*", Addison-Wesley, 1999.

[Takus & Profozich, 97] Takus, David A. e Profozich, David M., "*ARENA tutorial Software*", Pensylvania, System Modeling Corporation, 1997.

[Tanembaum, 94] Tanembaum, Andrew S., "*Redes de Computadores*", Rio de Janeiro, Ed. Campus, 1994.