Especificação de um *Framework* baseado em Componentes de Software Reutilizáveis para Aplicações de Gerência de Falhas em Redes de Computadores

Raissa Dantas Freire

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

Jacques Philippe Sauvé
(orientador)

Campina Grande, Paraíba, Brasil

FREIRE, Raissa Dantas

F866E

Especificação de um *Framework* baseado em Componentes de Software Reutilizáveis para Aplicações de Gerência de Falhas em Redes de Computadores

Dissertação de Mestrado, Universidade Federal da Paraíba, Centro de Ciências e Tecnologia, Coordenação de Pós-Graduação em Informática, Campina Grande, Paraíba, Abril de 2000.

133p. Il.

Orientador: Jacques Philippe Sauvé

1. Redes de Computadores
2. Gerência de Redes
3. *Frameworks* e Componentes de Software Reutilizáveis

CDU - 621.391

Resumo

Neste trabalho, especificamos um *framework* baseado em componentes de software reutilizáveis com o objetivo de facilitar o desenvolvimento de aplicações de gerência de falhas em redes de computadores. Diferentemente de outras propostas, buscamos fornecer um nível de abstração mais adequado ao desenvolvimento deste tipo de aplicações, de forma que o programador possa concentrar seus esforços na implementação da funcionalidade básica de sua aplicação, tendo em vista as abstrações fornecidas. Consideramos, em especial, a identificação dos recursos e facilidades que devem estar disponíveis e apresentamos, adicionalmente, uma proposta de implementação particular.

Abstract

This dissertation presents a component-based framework. The objective is to ease the development of fault management applications for computer networks. Differently from other approaches, the proposed solution provides an appropriate level of abstraction for the development of this type of application and enables the application developer to concentrate on the desired solutions, rather than on low-level details. We identify and characterize the necessary abstractions and present a particular implementation approach.

Ser mestre não significa apenas obter resultados;

mas significa, sobretudo, trilhar os caminhos que levam até eles

com conhecimento técnico e senso crítico.

Agradecimentos

Gostaria de agradecer primeiramente a Deus, pela possibilidade de realizar este trabalho. Nos momentos mais difíceis, senti-me amparada no seu amor.

A minha mãe, Evani, pelo apoio e compreensão. Se cheguei até aqui, grande parte da “culpa” é dela. Seus exemplos de força e coragem sempre me nortearão.

Ao meu orientador, Prof. Jacques, pela confiança e paciência. Obrigada pela seriedade na orientação, pela qualidade das discussões e pelo empenho.

Ao Prof. Pedro Sérgio Nicolletti, pelo inestimável apoio.

Ao meu noivo, Vinicius, pelo fundamental incentivo, pelo inestimável apoio, pela paciência e pelo carinho.

Aos colegas de mestrado, em especial a Érica, Lívia e Andréa.

Aos professores e funcionários.

Obrigada por estarem por perto e me fazerem lembrar que eu jamais estive sozinha.

Conteúdo

Índice de Figuras

Capítulo 1

Introdução

Nos últimos anos, as redes de computadores têm experimentado um crescimento vertiginoso. A integração de múltiplas tecnologias e o crescente número de usuários que passaram a utilizar estas redes para diversas finalidades, não só estenderam a sua aplicabilidade, como também as tornaram mais complexas. Ao mesmo tempo, passou-se a exigir um nível de qualidade cada vez melhor para o serviço fornecido [Ezhilchelvan, 1991].

Sistemas grandes, complexos e heterogêneos e que lidam com grandes volumes de informação precisam ser supervisionados e controlados a fim de que as necessidades dos usuários possam ser atendidas a contento. Num momento em que as redes se tornam cada vez mais importantes para as empresas, deixando de ser infra-estrutura de comunicação dispensável para se tornarem ferramentas cruciais ao desenvolvimento empresarial, é necessário fazer com que a rede opere eficientemente, livre de falhas, e, com isso, garantir que as necessidades dos seus usuários sejam adequadamente supridas. Sendo assim, é importante automatizar e melhorar cada vez mais o processo de gerência de redes de computadores a fim de manter a rede funcionando bem e de acordo com as suas especificações, durante a maior parte do tempo.

Com efeito, há muito que as tradicionais técnicas de gerenciamento *ad hoc*, tais como *ping* (teste de conectividade entre dois dispositivos) e *traceroute* (análise da rota estabelecida entre dois dispositivos) não são escaláveis e o fato é que um verdadeiro “arsenal” de ferramentas e mecanismos de gerência tem sido desenvolvido para realizar tarefas em cada uma das cinco áreas funcionais da gerência de redes [ISO/IEC 7498:1984], quais sejam: configuração, desempenho, falhas, segurança e contabilidade. Tudo isto define um ramo mercadológico particular, voltado para a gerência de redes, e uma ampla área de pesquisa.

Muitas aplicações e algumas plataformas de gerência estão disponíveis e podem ser utilizadas para diversas finalidades, que vão desde a configuração de equipamentos da rede até uma análise de desempenho ou a contabilização de gastos de recursos. Por outro lado, para desenvolver aplicações de gerência específicas, que atendam a requisitos particulares de um determinado contexto, outras APIs (*Application Programming Interfaces*) e linguagens podem ser utilizadas - é o caso, por exemplo, do JMX (*Java Management Extensions*) [Sun Microsystems, 1999], o qual define uma API para o desenvolvimento de aplicações de gerência em Java, e da OVSNMP [Hewlett Packard, 1998a], uma API de gerência fornecida pela plataforma de gerência *OpenView* da *Hewlett Packard*.

O fato é que o desenvolvimento de aplicações especializadas não tem sido uma tarefa fácil. Com o nível de abstração fornecido pelas APIs e linguagens disponíveis, o programador (gerentes e operadores da rede) precisa estar envolvido com aspectos de programação que não estão necessariamente associados ao domínio do problema modelado. Tipicamente, ele precisa ter um bom entendimento sobre estruturas de dados complexas ou sobre detalhes de comunicação associados a um protocolo de gerência particular e não conta com as abstrações necessárias para que possa se concentrar na solução desejada.

Sendo assim, uma solução que permita ao programador de aplicações de gerência se concentrar na obtenção da funcionalidade básica de sua aplicação, deve fornecer abstrações de mais alto nível que escondam do programador determinados aspectos de programação com os quais ele realmente não deve se preocupar, e que representem, conseqüentemente, mecanismos mais elaborados para a realização de tarefas de gerência. Trata-se, por exemplo, de prover uma coleta automática de informação de gerência na rede para fornecer mecanismos mais apropriados para a identificação e diagnóstico de falhas, no caso específico da gerência de falhas, ou mecanismos mais apropriados para a implementação de políticas de segurança, no caso específico da gerência de segurança. Com isto, espera-se tornar o desenvolvimento de aplicações de gerência de redes uma tarefa mais simples e mais rápida.

1.1. Objetivos da Dissertação

Este trabalho tem como objetivo principal especificar uma solução que facilite o desenvolvimento de aplicações de gerência de falhas em redes de computadores, ao elevar o

nível de abstração fornecido. Em particular, a solução consiste de um *framework*[1] baseado em componentes de software reutilizáveis, ou simplesmente um *framework* CO (*Component-Oriented*).

Adicionalmente, uma proposta de implementação é fornecida.

1.2. Escopo e Relevância

Dentro dos objetivos propostos, este trabalho apresenta como as aplicações de gerência podem ser desenvolvidas com base em diversas APIs e linguagens atualmente disponíveis. Este estudo serve de base para caracterizar um nível de abstração mais adequado ao desenvolvimento de aplicações de gerência, o qual não é fornecido pelas linguagens e APIs estudadas mas que deve ser fornecido pela solução proposta a fim de atingir seu objetivo principal, isto é, facilitar o desenvolvimento de aplicações de gerência de redes.

Por entender que a gerência de redes é uma área muito abrangente, que envolve a realização de muitas tarefas em cada uma de suas subáreas funcionais, julgamos necessário delimitar melhor o domínio do problema a fim de obter melhores resultados. Não parece factível ter um único *framework* que permita configurar a segurança da rede e, ao mesmo tempo, identificar e corrigir as possíveis falhas que tenham ocorrido. Um *framework* atende às necessidades de um conjunto bem definido de aplicações, inseridas num domínio de problema bem particular, e a sua funcionalidade e aplicabilidade dependem da captura do comportamento genérico compartilhado pelas diferentes aplicações que o utilizam [Landin & Niklasson, 1998]. Assim, quanto mais bem delimitado o domínio do problema, mais bem caracterizadas as aplicações que dele fazem parte e melhor pode ser a identificação deste comportamento comum que deve ser embutido no *framework*. Escolhemos, então, a gerência

[1]Não há uma boa tradução para o termo *framework* quando se fala em desenvolvimento de software. Segundo dicionários da língua inglesa, *framework* significa “estrutura”, palavra bastante genérica que não exprime o significado particular que gostaríamos de destacar. Sendo assim, preferimos não traduzi-lo, mas optamos por usar uma definição semanticamente mais completa: um *framework* é um conjunto de classes (no sentido de orientação a objetos) coesas que colaboram entre si para compor um projeto reutilizável para uma classe específica de software [Gamma *et al*., 1994]. No nosso caso, este projeto é descrito em termos de componentes de software reutilizáveis. Um estudo sobre *frameworks* pode ser encontrado em [Landin & Niklasson, 1998] e uma visão geral sobre o assunto é fornecida no apêndice A.

de falhas, por ser esta uma das áreas mais importantes da gerência de redes. Identificamos os requisitos funcionais da solução tendo em vista as aplicações de gerência de falhas.

Para atender aos requisitos levantados, especificamos um *framework* CO conforme já foi antecipado. Os conceitos envolvidos se baseiam, por excelência, na utilização de técnicas de orientação a objetos e toda a especificação fornecida serve de base para implementações futuras.

A importância do trabalho está em reunir conceitos pertinentes à gerência de falhas e propor mecanismos para facilitar o desenvolvimento deste tipo de aplicação. Assim, ele serve como ponto de partida para a implementação de soluções de gerência que busquem, efetivamente, atender às necessidades do programador de aplicações de gerência de falhas.

1.3. Estrutura da Dissertação

No capítulo 2, apresentamos alguns conceitos relacionados à gerência de redes e, em particular, à gerência de falhas. O leitor bem familiarizado com esta área de conhecimento pode iniciar sua leitura a partir do capítulo seguinte.

No capítulo 3, apresentamos diversas APIs e linguagens voltadas especificamente para a gerência de redes e exemplificamos como as aplicações podem ser desenvolvidas com base no nível de abstração fornecido. Comparamos as soluções apresentadas e concluímos, então, que tal nível de abstração não se mostra adequado para o desenvolvimento de aplicações específicas de gerência de falhas.

No capítulo 4, especificamos um *framework* baseado em componentes de software reutilizáveis para a gerência de falhas. Levantamos seus requisitos e identificamos suas características no sentido de atender aos requisitos levantados.

No capítulo 5, fornecemos uma proposta de implementação particular. Discutimos alguns aspectos de implementação importantes segundo a utilização de padrões de projeto [Gamma *et al*., 1994] e de características da linguagem Java.

No capítulo 6, apresentamos as conclusões e trabalhos futuros.

Capítulo 2

A Gerência de Redes de Computadores

Gerenciar uma rede de computadores consiste em supervisionar e controlar o seu funcionamento para que ela satisfaça as necessidades dos seus usuários [Sloman, 1994]. Uma gerência eficaz pode implicar em significativo acréscimo na produtividade da rede através da melhor utilização dos recursos disponíveis. Conseqüentemente, uma melhoria na gerência da rede pode trazer como resultados a melhoria na qualidade dos serviços, do ponto de vista dos usuários, e a redução nos custos de operação e manutenção da rede [Meira & Lages, 1998]. No caso das atuais redes, que tornam-se cada vez maiores e heterogêneas, esta é uma tarefa indispensável.

Neste capítulo fornecemos uma visão geral sobre a gerência de redes ao apresentar um conjunto de conceitos básicos sobre o assunto. A seção 2.1 descreve as atividades de gerência, agrupadas em áreas funcionais distintas. A seção 2.2 apresenta um modelo genérico para soluções de gerência e exemplifica uma implementação deste modelo através do padrão de gerência Internet (não pretendemos apresentar a descrição detalhada deste padrão de gerência, mas apenas o suficiente para o bom entendimento e fácil leitura dos capítulos seguintes). A seção 2.3 apresenta e estabelece conceitos relacionados especialmente à gerência de falhas, foco do nosso trabalho.

2.1. Áreas Funcionais da Gerência de Redes

A gerência de redes envolve a realização de um grande número de atividades, agrupadas em cinco áreas funcionais distintas segundo o modelo *OSI (Open System Interconnection*), definido pela ISO (*International Standards Organization*) [ISO/IEC 7498:1984], a saber:

Gerência de configuração: responsável pela manutenção e monitoração da estrutura física e lógica da rede. Trata da inicialização, alteração e coleta de informação de configuração presente na rede através da troca de informação com os elementos gerenciados e da atuação sobre esses elementos. Também inclui a adição e remoção de elementos gerenciados.

Gerência de desempenho: provê a avaliação permanente da utilização dos recursos da rede com base nos dados coletados e estatísticas de desempenho. Inclui planejamento de capacidade, onde se faz uma avaliação prévia da necessidade de recursos para evitar problemas futuros causados pela sobrecarga dos recursos disponíveis.

Gerência de falhas: inclui identificação, diagnóstico e correção de falhas.

Gerência de segurança: tem como objetivo controlar o acesso aos recursos da rede através do uso de técnicas de autenticação e políticas de autorização.

Gerência de contabilidade: visa identificar o consumo de recursos da rede, provendo, inclusive, a habilidade para cobrar pela utilização de tais recursos.

Esta divisão das atividades de gerência em áreas funcionais distintas facilita a modularização de projeto e a implementação de soluções de gerência, mas vale salientar que para se ter um gerenciamento eficaz, é necessária a integração entre as diversas áreas. Na verdade, uma falha pode causar redução no desempenho da rede; um erro de configuração pode causar prejuízo para a segurança; uma violação de segurança pode comprometer a contabilização dos recursos utilizados; e em qualquer uma destas situações, ações corretivas ou ajustes sobre a configuração do sistema podem ser necessários. Gerenciar uma rede, portanto, consiste em tratar aspectos em cada uma das cinco áreas, levando-se em conta a dependência que possa existir entre elas.

Plataformas de gerência, como *HP OpenView* da *Hewlett Packard*; *NetView*, da IBM; e *SunNetManager*, da *Sun Microsystems*, fornecem uma série de facilidades para realização destas tarefas e são, muitas vezes, utilizadas. Além disso, outras aplicações, inclusive baseadas em *web* [Sauvé, 1999], são desenvolvidas a fim de atender a necessidades de gerência específicas.

2.2. Um Modelo de Gerência

O desenvolvimento de aplicações de gerência se baseia numa arquitetura genérica que possui quatro componentes, a saber:

Estação de gerência: hospedeiro onde aplicações de gerência são executadas para monitorar e controlar os elementos gerenciados. O software composto por estas aplicações é normalmente conhecido como **gerente**. Em geral, há apenas uma estação de gerência por rede gerenciada.

Elemento gerenciado: qualquer dispositivo da rede que possa ser gerenciado, tais como hospedeiros, roteadores, *switches*, interfaces de rede, etc.; além do software instalado no sistema. O software presente nestes dispositivos, responsável por processar e responder os comandos de gerência enviados pela estação de gerência, chama-se **agente**.

Protocolo de gerência: usado pela estação de gerência e demais elementos gerenciados para efetuar a troca de informação de gerência. Define operações de monitoração (READ) e operações de controle (WRITE).

Informação de gerência: descreve o estado da rede.

A figura 2.1 mostra uma rede onde há uma estação de gerência (gerente) e vários elementos gerenciados ou agentes (roteadores, estações de trabalho, *hubs*, *switches*, impressora e computadores). A troca de informação de gerência é feita através da rede de acordo com determinado protocolo de gerência.

Para implementar a arquitetura descrita pela figura 2.1 e com o objetivo de obter uma gerência verdadeiramente integrada, diferentes padrões surgiram ao longo do tempo. A atenção está voltada para a estrutura, o conteúdo e a manipulação da informação de gerência. Imagine, por exemplo, se a sintaxe e a semântica dos dados obtidos de cada elemento gerenciado dependesse do fabricante de cada dispositivo. Considerando a heterogeneidade de tecnologias e a quantidade de fabricantes das atuais redes, obter uma visão geral do estado da rede com base na informação colhida seria uma tarefa terrivelmente complexa. Então, para permitir uma melhor integração entre as diferentes tecnologias no que diz respeito à gerência de redes, vários padrões foram propostos de modo a especificar a estrutura e o conteúdo da

informação de gerência que deve ser mantida por dispositivos de rede específicos, independentemente do fabricante.

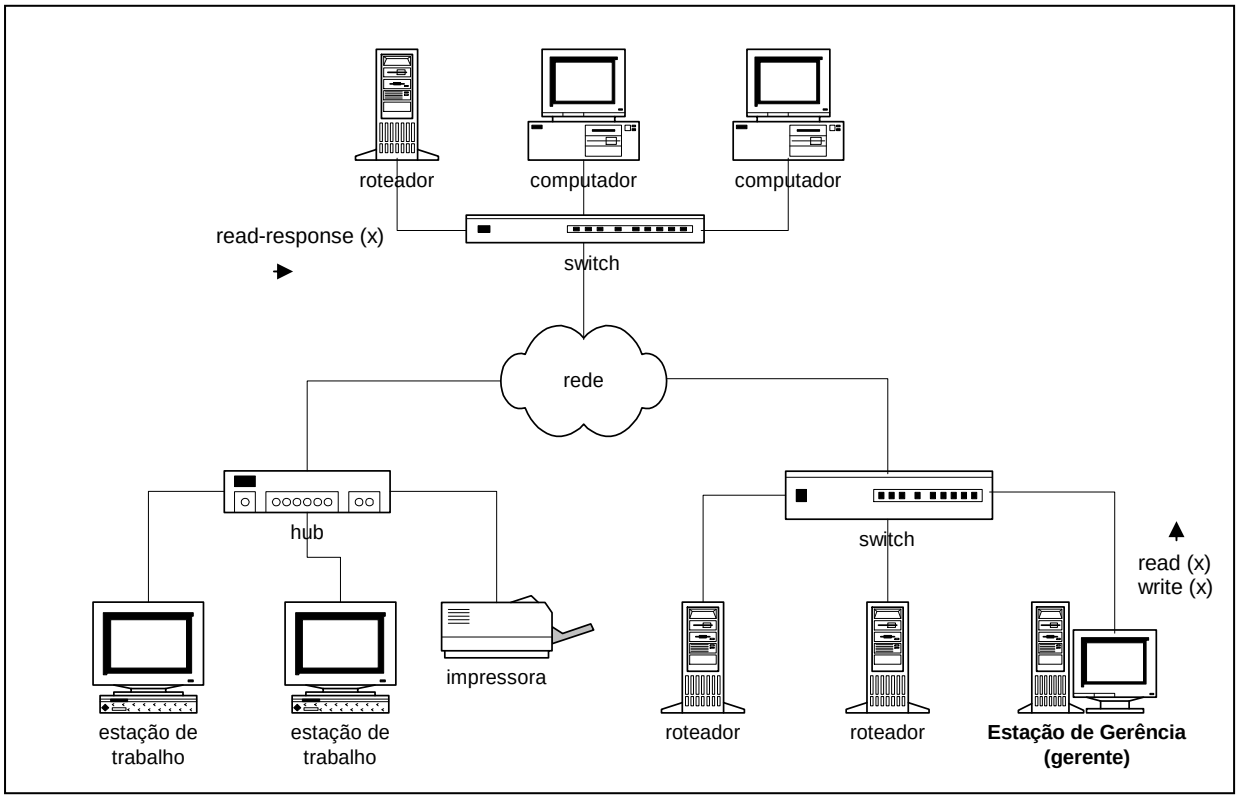


Figura 2.1. Visão física da rede gerenciada

Como exemplos de padrões de gerência, podemos citar o padrão OSI, definido pela ISO [ISO/IEC 7498:1984]; o padrão DMI (*Desktop Management Interface*) [DMTF, 1998], liderado pela *Microsoft*; o padrão TMN (*Telecommunications Management Network*) [Raman & Raman, 1999], desenvolvido pela ITU-T (*International Telecommunications Union*) para o gerenciamento de redes de telecomunicações; e o padrão Internet [Case *et al*., 1990]. Dentre todos, o padrão Internet é o mais utilizado e, sem a pretensão de esgotar o assunto, mas com a intenção de fornecer uma visão geral, ele será apresentado com mais detalhes a seguir.

2.2.1. O Padrão de Gerência Internet

O principal objetivo deste padrão de gerência é reduzir, ao mínimo, o impacto causado pelas atividades de gerência nos elementos gerenciados. Se a atividade de gerência não compromete o desempenho dos dispositivos da rede, então todos os dispositivos devem ser gerenciáveis; caso contrário, os fabricantes de roteadores, *switches*, modems e dispositivos

em geral, podem optar por não implementar esta funcionalidade. Portanto, o processamento de gerência realizado nos agentes é estritamente limitado. Em contrapartida, as estações de gerência são máquinas dedicadas a esta tarefa, realizando funções de gerência bem mais especializadas.

Segundo este padrão, agentes são vistos como coleções de variáveis escalares, armazenadas numa base de dados virtual, e todas as funções de gerência se resumem à monitoração e controle dos valores destas variáveis. A informação de gerência mantida por cada agente (variáveis escalares) é definida através do par **SMI/MIB** (*Structure of Management Information/Management Information Base*).

A SMI [Rose & McCloghrie, 1990b] especifica regras para nomear cada variável, uma sintaxe para descrever sua estrutura e regras para codificá-la. Em particular, variáveis são descritas através de um subconjunto da notação ASN.1 (*Abstract Syntax Notation One*) [ISO/IEC 8824:1987] e codificadas através de um conjunto básico de regras chamado BER (*Basic Enconding Rules*) [ISO/IEC 8825:1987]. Em outras palavras, a SMI define um "esquema" para a base de dados mantida por cada elemento gerenciado da rede [Rose, 1990]. Esta base de dados tem um nome bem determinado e se chama MIB (*Management Information Base*).

Um módulo MIB [Rose & McCloghrie, 1990a] define uma coleção de variáveis relacionadas. A MIB padrão atual ou MIB-II [Rose & McCloghrie, 1991], por exemplo, contém 190 variáveis que descrevem, em conjunto, a informação de gerência que deve ser mantida por cada elemento gerenciado que suporte a pilha de protocolos TCP/IP. Fabricantes podem definir seus próprios módulos MIB, agrupando a informação associada a uma tecnologia específica. Ao implementar uma MIB, um agente está definindo qual informação de gerência ele contém.

Para tornar mais claras as idéias contidas nos parágrafos anteriores, considere a variável abaixo, definida para a MIB-II. Esta variável indica a quantidade de tempo em centésimos de segundos desde a última vez em que o elemento gerenciado foi reinicializado.

```
sysUpTime
SYNTAX TimeTicks
ACCESS read-only
STATUS mandatory
```

```
::= {system 3 }
```

A sintaxe da definição acima (não confundir com o campo SYNTAX) é especificada pela SMI, a qual requer que cada variável tenha um nome (sysUpTime) e um tipo (SYNTAX). Neste caso, a sintaxe indica que esta variável possui um valor escalar do tipo TimeTicks. Tal valor pode ser apenas lido pela estação de gerência, como indicado pelo campo ACCESS, e deve estar presente em qualquer implementação da MIB-II, como indicado pelo campo STATUS. A especificação desta variável e de outras 189 constitui a MIB-II.

Para efetuar a troca de informação de gerência, agentes e gerentes se comunicam através do protocolo **SNMP** (*Simple Network Management Protocol*) [Case *et al.*, 1990], o qual define cinco primitivas básicas de comunicação: **get**, **get-next**, **set**, **get-response** e **trap**. Estas operações estão ilustradas na figura abaixo.

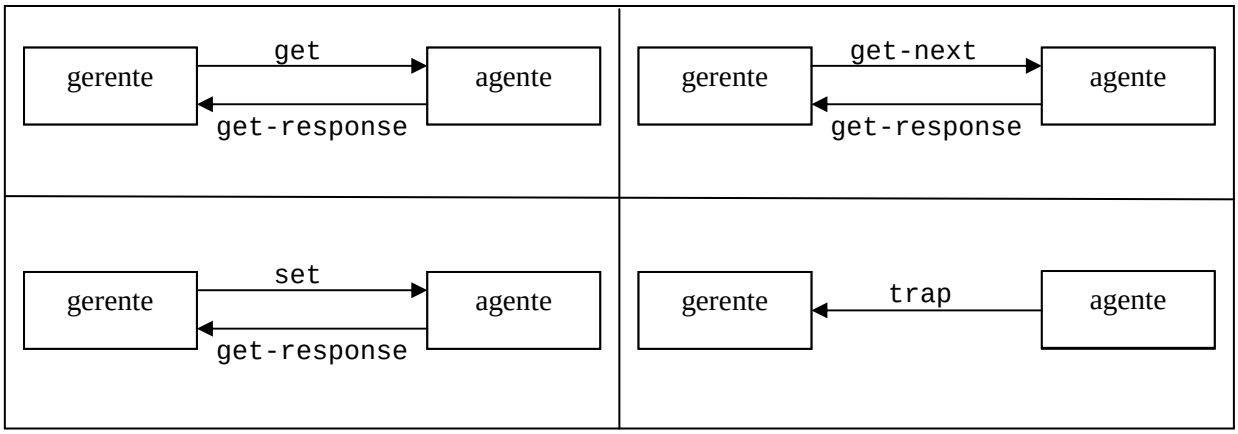


Figura 2.2. Primitivas de comunicação SNMP

As operações set e get permitem que o gerente possa, respectivamente, modificar e obter o valor de uma variável MIB presente num agente. A operação get-next permite recuperar a variável seguinte ao identificador de variável que lhe é passado, de forma que é possível "percorrer" uma MIB e recuperar, iterativa e seqüencialmente, a informação mantida pelo agente a partir das respostas obtidas com sucessivas requisições get-next (útil para, por exemplo, recuperar valores de tabelas com tamanho desconhecido). A primitiva get-response contém a resposta a uma requisição SNMP ou, simplesmente, o resultado da operação no caso de uma requisição do tipo set.

As operações set, get e get-next são operações através das quais o gerente requisita e o agente responde com a informação solicitada. Um erro também pode ser

retornado no caso em que a operação não seja bem sucedida. A operação `trap`, por sua vez, permite que o próprio agente envie informação ao gerente em resposta a um evento ocorrido (como o cruzamento de um limiar previamente estabelecido), sem que tenha havido requisição prévia.

O SNMP é um protocolo muito utilizado e atualmente há três versões disponíveis. Sua primeira versão, SNMPv1 [Case *et al*., 1990], foi descrita nesta seção e ainda é bastante utilizada. A versão seguinte, SNMPv2 [Case *et al*., 1993], trouxe alguns melhoramentos para o protocolo, dentre as quais podemos citar a definição de duas novas operações: `get-bulk` e `inform`. A operação `get-bulk` permite recuperar grandes quantidades de informação de uma só vez, de forma que o que anteriormente seria feito com várias requisições `get-next`, por exemplo, possa ser feito com uma única requisição `get-bulk`; a operação `inform` facilita a comunicação entre estações de gerência distintas. O SNMPv3 [Case *et al.*, 1998], por sua vez, preocupou-se especialmente com segurança e, de fato, muito se tem feito no sentido de incluir e melhorar aspectos de segurança no SNMP para garantir a autenticidade e a integridade das operações de gerência realizadas através deste protocolo. A princípio, segurança era muito fracamente fornecida.

Enfim, o padrão de gerência Internet, assim como outros padrões de gerência, fornece um mecanismo particular para especificar e manipular a informação de gerência presente na rede. Aplicações de gerência são desenvolvidas com base nestes padrões.

2.3. A Gerência de Falhas

Como vimos na seção 2.1, muitas são as tarefas de gerência a serem realizadas. Medidas de gerência eficazes em cada uma das cinco áreas - configuração, desempenho, falhas, segurança e contabilidade - são importantes para manter a rede funcionando sob controle.

Entretanto, uma das áreas mais importantes da gerência de redes é a gerência de falhas. Na presença de falhas, todo o funcionamento da rede pode ser comprometido e um grande prejuízo pode ser contabilizado. Em termos práticos, a parada de uma rede por um curto espaço de tempo pode custar milhares de dólares. Empresas dependem cada vez mais das redes de computadores que utilizam e se mostram cada vez mais dispostas a investir em mecanismos que forneçam um alto grau de confiabilidade e disponibilidade à rede. Então

quanto mais eficiente o gerenciamento de falhas, mais eficiente será o gerenciamento da rede como um todo.

> **Definição 1**: No contexto da gerência de redes, uma **falha**, ou uma **falta**, é definida como uma causa de um mau funcionamento. Falhas são responsáveis por dificultar ou impedir o funcionamento normal de um sistema. Neste trabalho, os termos falha e falta têm o mesmo significado.

A gerência de falhas consiste em detectar alguma anormalidade no comportamento padrão do sistema a fim de que ações corretivas possam ser apropriadamente aplicadas [Oates, 1995]. Para tanto, as tarefas básicas descritas na figura abaixo devem ser realizadas.

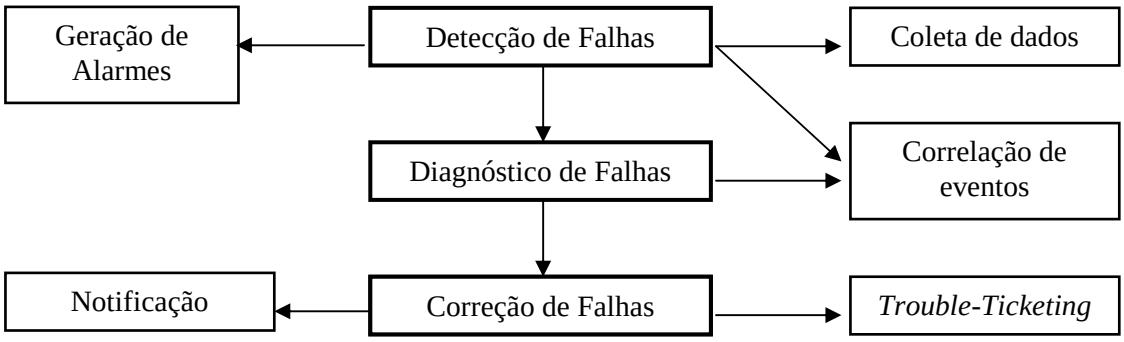


Figura 2.3. Tarefas básicas da gerência de falhas

A primeira de todas as tarefas consiste em identificar um desvio no comportamento do sistema, detectando que uma falha ocorreu. Para isto, é necessário fazer a coleta de dados nos elementos gerenciados a fim de monitorar indicadores da presença de falhas na rede (taxas de erro, atrasos de transmissão, etc.). A detecção de falhas também envolve a correlação de eventos ocorridos na rede, a qual será descrita mais adiante, e a geração de alarmes.

> **Definição 2**: Um **evento** é um momento interessante de atividade na rede (o cruzamento de um limiar, por exemplo, indicando que a taxa de erros ultrapassou determinado valor). Neste contexto, um evento descreve uma falha.

> **Definição 3**: Um **alarme** é um evento que será levado ao conhecimento do operador da rede através de algum mecanismo de notificação.

A tarefa de diagnóstico consiste em identificar a causa e a localização de uma falha. Um diagnóstico preciso é indispensável para que se possa resolver o problema eficazmente e, assim, "curar" os possíveis efeitos causados. O diagnóstico de falhas também envolve a correlação de eventos para fornecer melhores resultados.

Por fim, a tarefa de correção consiste em formular ações corretivas que possam restabelecer o estado da rede para um estado normal, livre de falhas. Ações podem ser sugeridas de forma automática e outras aplicações, normalmente conhecidas como aplicações de *trouble-ticketing* - ou aplicações de acompanhamento de ocorrências - podem ser utilizadas para controlar o processo de resolução de problemas.

> **Definição 4**: Um ***ticket*** é um registro que descreve o tempo de vida de um problema.

Sistemas de acompanhamento de ocorrências podem ajudar *durante* o processo de correção de falhas. Enquanto o problema não é efetivamente resolvido, as ações realizadas são registradas num *ticket* (que pode, inclusive, trafegar através da rede), permitindo sincronismo e uma maior integração entre as pessoas envolvidas na resolução de determinado problema [Madruga & Tarouco, 1994]. O *ticket* é "aberto" tão logo o problema seja identificado e só é "fechado" quando o problema é resolvido, guardando todo o histórico do tempo de vida do problema. Uma descrição do problema, as ações corretivas aplicadas, as pessoas e empresas envolvidas e o tempo de resolução do problema são algumas das informações mantidas.

Sistemas de acompanhamento de ocorrências, portanto, provêem uma base de dados com informação sobre o processo de resolução dos eventuais problemas identificados na rede. Todos os esforços envidados no sentido de resolver determinado problema são registrados para que esta informação possa ser útil futuramente, nos casos em que o mesmo problema volte a ocorrer ou problemas similares sejam detectados.

A correção de falhas também se baseia na notificação sobre as falhas ocorridas, a qual pode ser feita visualmente (através de uma interface gráfica que indica o que está funcionando com que estado), via mensagem eletrônica, *pager*, etc. Isto também envolve a realização de registros de ocorrências (*logging*).

2.3.1. A Correlação de Eventos

O principal requisito para se fazer uma gerência de falhas de forma integrada é o fornecimento da informação sobre o funcionamento da rede em um centro de gerência (estação de gerência), em tempo real [Meira, 1997]. As anormalidades que ocorrem durante a operação da rede provocam a geração de alarmes, indicando o estado da rede.

Com o crescimento das redes, entretanto, a quantidade de alarmes gerados pode ser muito grande - em decorrência da quantidade de eventos ocorridos - e isto fatalmente tornará o processamento individual de cada um destes alarmes inviável e humanamente impossível. Além disso, muitos dos alarmes recebidos não contêm informação original e diversos fatores contribuem para esta situação [Houck *et al.*, 1995]:

> um dispositivo pode indicar várias ocorrências de um mesmo evento em decorrência de uma única falha;
>
> a falha pode ser intrinsecamente intermitente, o que implica na ocorrência de um evento a cada nova ocorrência da falha;
>
> a falha de um componente pode resultar na ocorrência de um evento a cada vez que se invoca o serviço prestado por esse componente;
>
> uma única falha pode ser detectada por múltiplos componentes da rede, cada um deles constatando a ocorrência de um evento;
>
> a falha de um dado componente pode afetar diversos outros componentes, causando a propagação da falha.

Para contornar estes problemas, levando informação verdadeiramente útil ao operador da rede, aplicações de gerência de falhas fazem uso da correlação de eventos.

> **Definição**: A **correlação de eventos** consiste na interpretação conceitual de múltiplos eventos, levando à atribuição de um novo significado aos eventos originais [Jakobson & Weissman, 1993]. A correlação de eventos geralmente tem como objetivo reduzir a quantidade de alarmes transferidos aos operadores do sistema de gerência de rede, ao aumentar o conteúdo semântico dos eventos resultantes.

A possibilidade de reduzir a quantidade de informação gerada, respeitando a propriedade de dependência que existe entre os eventos ocorridos, faz da correlação de eventos um importante conceito no sentido de agilizar o diagnóstico de falhas. Suponha, por exemplo, que ocorra uma partição na rede, devido a uma falha num dos roteadores ou qualquer outro equipamento de interconexão. Muitos outros dispositivos ficarão inacessíveis a partir da estação de gerência e a propagação desta falha poderá levar à ocorrência de inúmeros eventos derivados de um só: “roteador inoperante”. Conhecendo a topologia da rede e sabendo quais os dispositivos inacessíveis, a aplicação poderia gerar um único alarme, em torno do qual os eventos ocorridos nestes dispositivos estariam associados à falha no roteador. Isto não só reduziria a quantidade de alarmes gerados como facilitaria o diagnóstico do problema.

Há diversos tipos de correlação, dentre os quais podemos citar [Meira, 1997]:

Contagem: consiste em gerar um novo evento cada vez que o número de ocorrências de um determinado tipo de evento ultrapassar um limiar previamente estabelecido.

Compressão: consiste em detectar múltiplas ocorrências de um mesmo evento, num dado intervalo de tempo, substituindo os eventos correspondentes por um único evento, possivelmente indicando quantas vezes o evento ocorreu durante o intervalo de tempo considerado.

Supressão: consiste em detectar se um mesmo evento ocorreu determinado número de vezes (k) dentro de um intervalo de tempo, substituindo os k eventos ocorridos por um único evento tão logo k seja atingido.

Filtragem: consiste em selecionar determinado evento com base em parâmetros previamente configurados. A origem do evento, por exemplo, pode servir como parâmetro de filtragem.

Relacionamento Temporal: a correlação depende da ordem ou do tempo em que os eventos ocorrem.

Há, ainda, muitos métodos e algoritmos. Algumas dessas abordagens são probabilistas, outras utilizam paradigmas tradicionais da Inteligência Artificial. Uma descrição mais detalhada para abordagens como correlação baseada em regras, correlação por

codificação ou correlação distribuída, bem como uma comparação entre as abordagens existentes podem ser encontradas em [Meira, 1997]. A escolha do tipo de correlação a ser realizada e da abordagem a ser adotada vai depender do objetivo da correlação, o qual pode ser simplesmente a redução de informação enviada ao gerente ou algo mais elaborado, como a localização e o diagnóstico de falhas.

A correlação de eventos pode ser aplicada a qualquer uma das cinco áreas funcionais de gerência, mas a maioria das aplicações que a utilizam, encontradas na literatura, têm a funcionalidade específica de gerência de falhas. Na verdade, em qualquer área onde o volume de informações envolvido é muito grande, a correlação de eventos pode ser útil.

2.3.2. Arquitetura Geral de uma Aplicação de Gerência de Falhas

Aplicações de gerência de falhas distintas têm suas particularidades, mas a fim de prover sua funcionalidade básica, todas elas apresentam o comportamento descrito pela figura 2.4 [Gibson *et al*., 1996].

De acordo com a figura, de um lado estão os agentes ou elementos gerenciados; do outro, está a aplicação (gerente). Para realizar a sua tarefa de gerência, a aplicação depende da informação colhida nos agentes e a obtenção desta informação é feita de duas formas:

> **sincronamente**: a aplicação, periodicamente, requisita informação ao agente e este responde com a informação solicitada; e
>
> **assincronamente**: os próprios agentes enviam informação para aplicação em resposta à ocorrência de um evento, sem requisição prévia.

A informação recebida passa por filtros, geralmente baseados em limiares definidos de acordo com políticas de gerência adequadas. Indicadores de falhas como taxas de erros, são analisados a fim de que a ocorrência de eventos na rede possa ser identificada. Eventos podem ser correlacionados (informações sobre a topologia da rede podem ser úteis) e, finalmente, identificadas as falhas, alarmes devem ser gerados. Os alarmes podem, ainda, ser priorizados e notificados de várias formas (via correio eletrônico, via interface gráfica, etc.). A `prioridade` de um alarme indica a gravidade da falha por ele notificada. Adicionalmente, o registro de ocorrências pode ser feito (geração de *logs*).

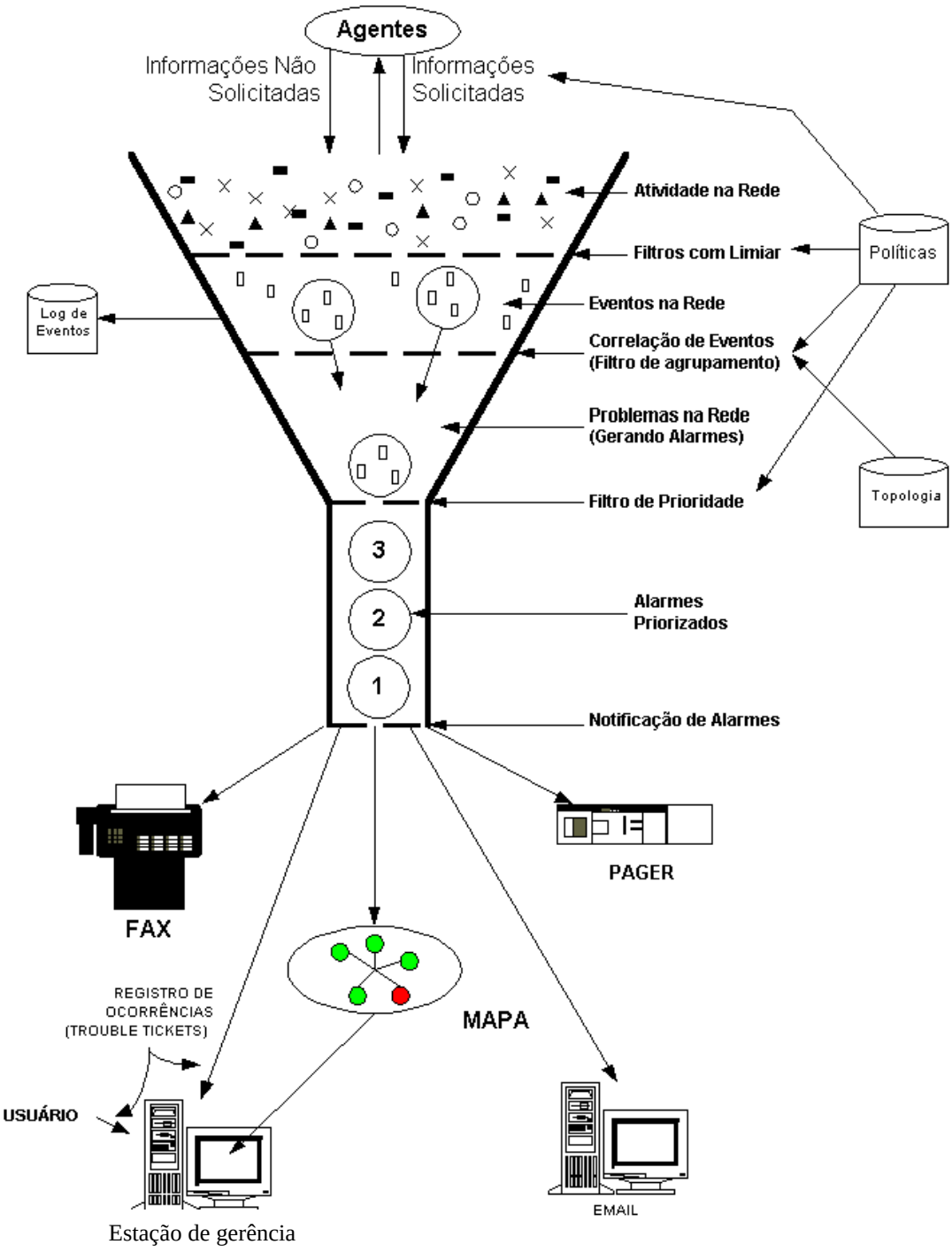


Figura 2.4. Arquitetura geral de uma aplicação de gerência de falhas

Assim, a figura 2.4 descreve o comportamento genérico de uma aplicação de gerência de falhas. Quanto mais eficientes os mecanismos empregados para identificação, diagnóstico e correção de falhas, mais eficiente o gerenciamento da rede. A caracterização deste comportamento será importante para a identificação dos requisitos de uma solução que

visa facilitar o desenvolvimento de aplicações de gerência de falhas. Tal solução será descrita no capítulo 4.

Capítulo 3

APIs e Linguagens para a Implementação de Aplicações de Gerência de Redes

Com a evolução da gerência de redes, uma grande variedade de ferramentas e aplicações de gerência tem sido desenvolvida. Em cada uma das cinco áreas funcionais descritas na seção 2.1, aplicações realizam as tarefas descritas de modo a atender às necessidades específicas de cada sistema segundo políticas de gerência adequadas. Para tanto, elas se utilizam dos protocolos de gerência disponíveis, tal como o SNMP no caso do padrão de gerência Internet. O desenvolvimento de aplicações de gerência, entretanto, não tem sido uma tarefa fácil.

Plataformas de gerência fornecem APIs (*Application Programming Interfaces*) para o desenvolvimento de aplicações de gerência. Tipicamente, uma linguagem como C ou C++ é utilizada para construir uma aplicação "sob medida" para um determinado contexto. Esta solução, entretanto, exige considerável experiência de programação, já que fazer gerência de redes através da linguagem C, por exemplo, requer um bom entendimento de estruturas de dados complexas até mesmo para a realização de tarefas simples.

Opcionalmente, outras APIs e linguagens mais especificamente voltadas para a gerência de redes - exemplos podem ser encontrados em [Lima, 1998], [Schönwälder & Langendörfer, 1995], [Leinen, 1999], [Mellquist, 1997], [Sun Microsystems, 1999], [Hewlett Packard, 1998a] e [Simões *et al.*, 1994] - podem ser utilizadas. Mesmo assim, como veremos adiante, programadores ainda precisam estar envolvidos com detalhes de programação que não estão necessariamente associados ao domínio do problema modelado, o que sugere que tais APIs e linguagens não fornecem um nível de abstração adequado ao desenvolvimento de aplicações de gerência específicas. De um modo geral, elas fornecem uma interface para realização de operações SNMP mas deixam de fornecer abstrações de mais alto nível que

atendam a necessidades de gerência específicas, como mecanismos mais elaborados para a identificação de falhas, no caso da gerência de falhas.

Neste capítulo, apresentamos alguns exemplos de como aplicações de gerência de redes podem ser desenvolvidas com base nas APIs e linguagens supracitadas. Nas seções 3.1, 3.2, 3.3 e 3.4, apresentamos trechos de código que exemplificam como programadores podem construir suas aplicações de acordo com as características que cada API ou linguagem possui. Na seção 3.5, comparamos as APIs e linguagens apresentadas e, ao caracterizarmos o nível de abstração adequado ao desenvolvimento de aplicações de gerência de falhas, concluímos que nenhuma das APIs ou linguagens vistas é suficientemente capaz de fornecê-lo.

3.1. A Linguagem de Comandos Tcl

A linguagem de comandos Tcl (*Tool Command Language*) tem uma API para o SNMP conforme descrito pela figura 3.1. O comando `snmp session` permite abrir uma *sessão* (estabelecer um contexto) através da qual requisições SNMP serão realizadas. Cada sessão pode ser configurada de acordo com um conjunto de opções, como o endereço do agente com o qual a sessão será estabelecida. Os comandos `configure` e `cget` podem ser utilizados para modificar e obter, respectivamente, a configuração corrente de uma sessão.

```
#seja s uma sessão snmp
snmp session [<options>]
$s configure [<options>]
$s cget <option>
$s get <vbl> [<callback>]
$s getnext <vbl> [<callback>]
$s getbulk <nr> <mr> <vbl> [<callback>]
$s set <vbl> [<callback>]
$s inform <trapOid> <vbl> [<callback>]
$s trap <trapOid> <vbl>
$s bind <notification> <script>
$s wait [<id>]
$s walk <var> <vbl> <body>
$s destroy
snmp wait
```

Figura 3.1. API SNMP/Tcl

Todas as operações SNMP (`get`, `getnext`, `getbulk`, `set`, `inform` e `trap`) podem ser realizadas no modo síncrono ou no modo assíncrono. No modo síncrono, a aplicação fica bloqueada, esperando pela resposta à requisição feita. Uma lista de variáveis de gerência ou um erro são retornados. No modo assíncrono, a aplicação não fica bloqueada mas especifica um parâmetro de retorno (*callback*) que será associado à requisição enviada. A resposta, então, será recebida através deste parâmetro. O resultado da operação e o estado da informação retornada são obtidos, no código, através de seqüências iniciadas com o símbolo %. Adicionalmente, a opção `wait` faz com que o programa (*script*) espere até que todas as requisições estabelecidas durante seu fluxo de execução sejam processadas. A figura 3.2 mostra um exemplo de código escrito com base nesta API de gerência.

```
proc obtemSysUpTime {endRede} {
        for {set i 1} {$i < 10} {incr i} {
        # abre uma sessão (s)
        set s [snmp session –address $endRede.$i]
        # faz a requisição e especifica o callback
        $s get sysUpTime.0{
            if {"%E" == "noError"}{
                 set d [lindex [lindex "%V" 0] 2]
                 puts "{%S cget –address] \t $d"
            }
            %S destroy
        }
    }
    snmp wait
}
```

Figura 3.2. *Script* para a obtenção de valores em agentes SNMP

No exemplo acima, deseja-se obter o valor de `sysUpTime` das dez primeiras máquinas da rede. Uma operação `get` é enviada no modo assíncrono, em cada sessão estabelecida, a fim de recuperar o valor de `sysUpTime` de cada uma das máquinas. Ao tratar cada resposta recebida, verifica-se, primeiramente, se ocorreu algum erro (usando a seqüência %E); caso não haja erro, o endereço do agente e o valor de `sysUpTime` são escritos na saída padrão, antes que a sessão (recuperada através da seqüência %S) seja finalizada.

Uma API de gerência bastante similar, LuaMan [Lima, 1998], foi desenvolvida na PUC-Rio (Pontifícia Universidade Católica, Rio de Janeiro - RJ) para a linguagem de extensão Lua [Ierusalimschy *et al.*, 1996]. LuaMan é uma biblioteca especificamente voltada para a gerência de redes que fornece basicamente os mesmos recursos fornecidos pela API SNMP do Tcl. A linguagem interpretada Perl também fornece uma API de gerência muito semelhante [Leinen, 1999], com as mesmas operações e características básicas. Exemplos de aplicações e sistemas desenvolvidos em Tcl, LuaMan e Perl podem ser encontrados, respectivamente, em [Schönwälder & Langendörfer, 1996], [Lima *et al*., 1998] e [Leinen, 1999].

3.2. A API de Gerência Java

A *Sun Microsystems* possui uma API de gerência denominada *Snmp Manager API* [Sun Microsystems, 1999], baseada na linguagem Java. Ela consiste de um conjunto de classes que simplificam o desenvolvimento de aplicações para gerenciar agentes SNMP[2].

Há quatro classes básicas: `SnmpPeer`, `SnmpParameters`, `SnmpSession` e `SnmpRequest`. As classes `SnmpPeer` e `SnmpParameters` descrevem o agente com o qual a aplicação (gerente) deseja se comunicar. A figura 3.3 mostra a instanciação e configuração de um agente SNMP.

A classe `SnmpSession` representa o contexto para as conexões abertas entre o gerente e um ou mais agentes. Ela implementa os métodos através dos quais os diferentes tipos de requisições SNMP podem ser realizados. São eles: `snmpGet`, `snmpGetNext`, `snmpSet` e `snmpGetBulk`. Adicionalmente, estão disponíveis as operações `snmpGetPoll`, `snmpGetNextPoll` e `snmpWalkUntil`. Com `snmpGetPoll` e `snmpGetNextPoll` uma monitoração periódica pode ser realizada no agente, de acordo com um parâmetro da operação que especifica um intervalo de tempo. A operação é automaticamente realizada segundo esta periodicidade, retornando uma lista de variáveis conforme requisitado. Com `snmpWalkUntil`, a MIB é percorrida até que determinada condição seja satisfeita e o conjunto de variáveis lido é retornado. Cada operação SNMP realizada durante a sessão é representada por um objeto do tipo `SnmpRequest`.

---

[2] Uma outra API de gerência, escrita em C++ e similar ao *SNMP Manager API*, pode ser encontrada em [Mellquist, 1997].

```
String nomeHost = String ("anjinho.dsc.ufpb.br");
int porta = 8085;

//Cria um objeto SnmpPeer para representar o agente remoto.
SnmpPeer roteadorA = new SnmpPeer (nomeHost, porta);

//Cria parâmetros para associar com o agente remoto. Neste caso,
//as comunidades de leitura e escrita estão sendo especificadas[3].
SnmpParameters params = new SnmpParameters ("public", "private");

//Configura o agente remoto de acordo com os parâmetros
//especificados.
RoteadorA.setSnmpParam (params);
```

Figura 3.3. Instanciando e configurando um agente SNMP

Na figura 3.4, a aplicação instancia e configura uma sessão SNMP (`sessão`), através da qual requisições SNMP serão feitas (`get`, por exemplo). Um agente *default* (`roteadorA`) pode ser especificado, de modo que requisições feitas durante a sessão, sem especificar um agente destino, sejam enviadas para este agente. Cada sessão pode ter seu comportamento configurado de acordo com um conjunto de opções. Através do método `setPduFixedOnError`, por exemplo, caso uma resposta à requisição contenha um erro devido à obtenção do valor de uma variável, a variável com problema é removida da lista de variáveis requisitadas (`lista`) e a requisição é automaticamente reenviada ao agente. A aplicação usa a sessão criada e faz uma requisição (`snmpGet`) no modo síncrono para obter o valor da variável `sysUpTime`. Ela também estabelece um limite de tempo, 10 segundos, até o qual a resposta é esperada.

```
//Instancia sessão com o nome dado.
SnmpSession sessao = new SnmpSession("Sessão atual");

//Usa o agente (roteadorA) como agente default.
sessao.setDefaultPeer(roteadorA);
```

[3] Comunidades são utilizadas para prover segurança e há, tipicamente, dois tipos de comunidades, uma para cada permissão de acesso: leitura e escrita. Para que o gerente tenha acesso à informação presente no agente, ele deve conhecer o nome da comunidade conforme especificado pelo agente. Tal parâmetro é utilizado para autenticação.

```
//Especifica as opções de sessão desejadas.
sessao.snmpOptions.setPduFixedOnError(false);

//Constrói a lista de variáveis a ser recuperada.
SnmpVarbindList lista = new SnmpVarbindList("lista de variáveis");

//Prepara uma lista de variáveis para obtenção do valor da variável
//sysUpTime.
lista.addVariable("sysUpTime.0");

// Cria uma requisição no modo assíncrono (parâmetro de callback =
// null) para obtenção do valor desejado.
SnmpRequest request = sessao.snmpGet(null, lista);

// Envia a requisição e espera por, no máximo, 10 seg.
boolean fim = request.waitForCompletion(10000);
```

Figura 3.4. Instanciando uma requisição no modo síncrono

Assim como no Tcl, operações podem ser realizadas no modo síncrono ou assíncrono. O modo de operação é determinado pelos parâmetros passados às requisições realizadas (no exemplo anterior, se tivéssemos passado um objeto específico em vez de `null` no momento em que a requisição é feita, a aplicação não ficaria esperando pela resposta, mas esta última seria enviada para o objeto especificado, através de um *callback*). O *SNMP Manager API* também fornece recursos para a manipulação de eventos assíncronos (*traps*) e permite implementar segurança ou utilizar criptografia durante a realização das operações.

3.3. A API SNMP do *OpenView*

Conforme dito anteriormente, plataformas de gerência também fornecem APIs para o desenvolvimento de aplicações de gerência. É o caso da OVSNMP (*OpenView SNMP*) [Hewlett Packard, 1998a], a API SNMP de gerência do *OpenView*, plataforma de gerência da *Hewlett Packard*. A API OVSNMP está disponível em C e C++ e fornece suporte ao SNMPv1 e SNMPv2 a aplicações construídas para o *OpenView*. Um exemplo de aplicação desenvolvida com base nesta API pode ser encontrado em [Gosselin, 1999].

Assim como nos exemplos vistos anteriormente, a OVSNMP também se baseia no conceito de **sessões configuráveis** através das quais requisições podem ser feitas. Mensagens

SNMP são enviadas através da sessão estabelecida com o agente remoto. As estruturas de dados básicas que podem ser utilizadas estão listadas na tabela abaixo.

Nome da Estrutura de Dados	Descrição
`OvsnmpSession`	Obtida através de uma chamada a `OvsnmpOpen()`, esta estrutura identifica uma sessão SNMP particular. Usada como um parâmetro para muitas outras funções. Criada por `OvsnmpOpen`() e liberada pela função `OvsnmpFree`().
`OvsnmpPdu`	Contém uma mensagem SNMP. Inclui informação sobre o tipo de mensagem (`get`, `getNext`, `getBulk`, `set`, `trap` ou `inform`), assim como sobre os dados nela contidos. Criada por `OvsnmpCreatePdu`() e liberada por `OvsnmpFreePdu`().
`OvsnmpVarBind`	Elementos de uma lista de variáveis. Cada elemento da lista contém o identificador de uma variável MIB e o seu valor. Faz parte da estrutura `OvsnmpPdu`. Criada por `OvsnmpAddVarBind()` e liberada por `OvsnmpFreePdu()`.

Tabela 3.1. Estruturas de dados básicas da API OVSNMP

Com base nisto, é possível ter o trecho de código apresentado na figura 3.5 (todos os exemplos fornecidos estão escritos em C). Uma sessão SNMP é estabelecida com o agente identificado pelo nome `anjinho.dsc.ufpb.br` e uma mensagem (`pdu`) é criada com uma lista de variáveis vazia.

```
//Abre sessão SNMP com o agente anjinho.dsc.ufpb.br.
sessao = OvsnmpOpen ("anjinho.dsc.ufpb.br", ...);

//Cria a mensagem SNMP a ser enviada.
pdu = OvsnmpCreatePdu (...);

//Adiciona uma lista de variáveis vazia ao pdu.
OvsnmpAddNullVarBind(...);
```

Figura 3.5. Abertura de uma sessão e preparação de uma mensagem SNMP

Requisições podem ser feitas no modo síncrono ou no modo assíncrono. A função `OVsnmpBlockingSend`, por exemplo, é síncrona, enquanto a função `OVsnmpSend` é assíncrona. No segundo caso, uma função de retorno (*callback*) deve ser especificada a fim de tratar a resposta recebida (um apontador para a função de retorno é especificado no momento de abertura da sessão). Confira na figura abaixo.

```
//Abre sessão SNMP com o agente remoto e especifica uma função de
//retorno.
sessao = OvsnmpOpen ("150.165.75.21", ..., applCb, ...);

//Cria a mensagem SNMP a ser enviada.
pdu = OvsnmpCreatePdu (..., GET_REQ_MSG, ...);
OvsnmpAddVarBind(pdu, "sysUpTime.0");

// Envia uma requisição assíncrona.
req = OvsnmpSend (session, pdu);
```

Figura 3.6. Envio de uma requisição no modo assíncrono

De acordo com a figura 3.6, uma sessão SNMP é estabelecida com o agente cujo endereço IP é 150.165.75.21. Uma mensagem (`pdu`) é criada para a obtenção (`GET_REQ_MSG`) do valor de `sysUpTime`. A requisição (`req`) é feita através da função `OvnsmpSend`[4] e a resposta é recebida através do parâmetro `applCb`.

Para tratar eventos assíncronos SNMP, ou simplesmente *traps*, as aplicações podem utilizar o **subsistema de eventos** do *OpenView*. Este subsistema repassa os eventos recebidos para as aplicações interessadas, entre os quais estão os *traps* SNMP. Como parte deste subsistema, há módulos que oferecem serviços de correlação de eventos (ECS - *Event Correlation Services engine*) e, para receberem os eventos de interesse, aplicações devem se cadastrar no subsistema e especificar uma das três fontes de eventos possíveis, a saber:

RAW - a aplicação recebe todos os eventos recebidos pelo subsistema de eventos, exceto aqueles oriundos do ECS.

CORR - a aplicação recebe os eventos oriundos de um módulo ECS específico.

ALL - a aplicação recebe todos os eventos oriundos do subsistema de eventos e do ECS.

[4] Também é possível obter a informação de gerência armazenada no banco de dados do próprio *OpenView*, em vez de buscar informação diretamente no agente. Neste caso, uma outra API, a OVW API (*OpenView Windows API*) [Hewlett Packard, 1998b], pode ser utilizada. Trata-se de uma API de mais alto nível através da qual é possível, entre outras coisas, manipular *objetos* do banco de dados do *OpenView*. Objetos mantidos no banco de dados podem ser lidos e escritos e novos objetos podem ser criados. A informação mantida é coletada automaticamente e qualquer aplicação pode ter acesso a esta informação.

Ao utilizar a função `OVsnmpEventOpen`, as aplicações podem estabelecer uma sessão de comunicação direta com o subsistema de eventos para enviar e receber eventos SNMP. Filtros podem ser aplicados a fim de reduzir o número de eventos recebidos.

Um outro aspecto importante a considerar com relação a esta API, é **a gerência de memória**. Aqui, as estruturas de dados utilizadas para a realização de operações SNMP não só precisam ser alocadas, como ocorre com as outras APIs apresentadas, mas também precisam ser explicitamente liberadas. Linguagens como C ou C++ não possuem *garbage collection* - mecanismo através do qual o espaço de memória previamente alocado e em desuso é automaticamente liberado – e, neste caso, toda a liberação de memória deve ser feita explicitamente pela própria aplicação. De acordo com a tabela 3.1, estruturas de dados são criadas e liberadas através de funções específicas (`OVsnmpOpen`, `OVsnmpFree`, etc.).

Além disso, outras estruturas precisam ser **alocadas dinamicamente**. Considere a estrutura `OVsnmpPdu`, por exemplo, descrita pela figura 3.7. Ela possui alguns campos que são específicos de determinado tipo de mensagem (`trap`, por exemplo). Neste caso, estes campos devem ser alocados dinamicamente, sendo de responsabilidade do programador realizar esta tarefa através de funções da API, como `OVsnmpMalloc`, `OvsnmpCalloc`, `OVsnmpRealloc` e `OVsnmpFree`. A manipulação das estruturas de dados disponíveis, portanto, torna-se bem mais complexa.

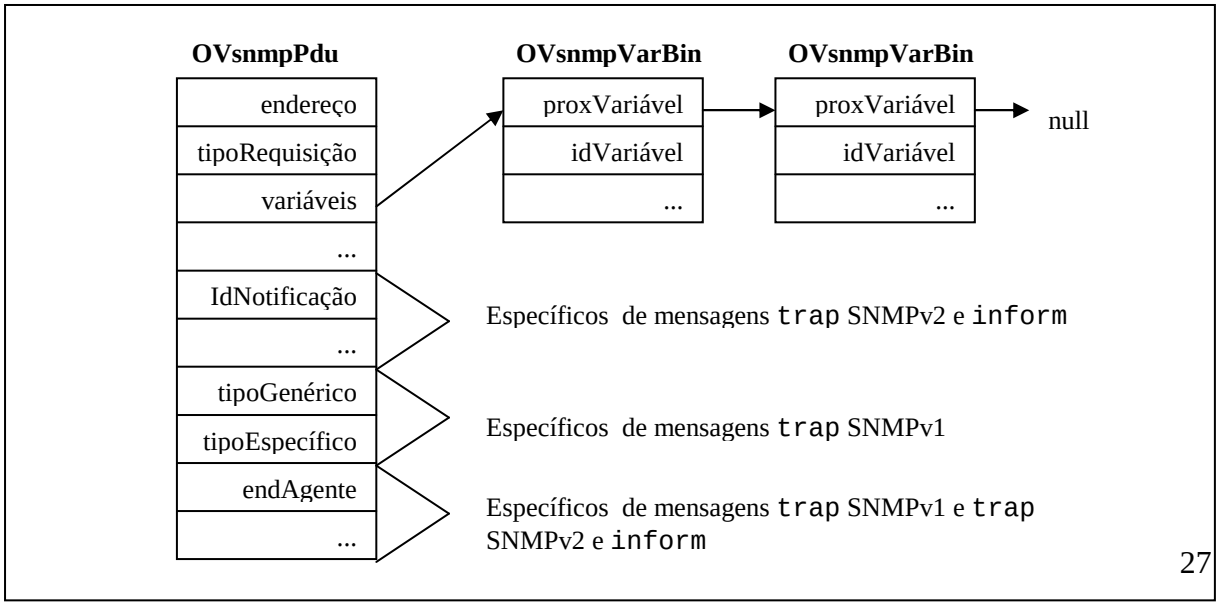


Figura 3.7. A estrutura de dados `OVsnmpPdu`

Programadores ainda podem utilizar outros subsistemas que compõem o *OpenView* como aquele que facilita o registro de ocorrências (*logging*) e outros voltados para a correlação de eventos. O *OpenView*, portanto, fornece meios mais eficientes para identificar, diagnosticar e tratar (registrar) as falhas ocorridas. Na maioria das vezes, basta utilizar algumas dezenas entre as centenas de funções que compõem as APIs disponíveis.

3.4. A Notação SNMA

A SNMA (*Specification of Network Management Applications*) [Simões *et al.*, 1994] é uma notação de alto nível para a especificação de aplicações de gerência de redes. Diferentemente das outras soluções apresentadas, a SNMA é uma **linguagem** que busca prover as abstrações necessárias para que o programador possa se concentrar na solução desejada enquanto não precisa se preocupar com detalhes ligados a protocolos de comunicação.

A notação SNMA tem duas características básicas:

> Ela provê **meios de alto nível para representar a informação de gerência**. Tal representação esconde o processo usado para obter esta informação e detalhes relacionados (operações do protocolo de comunicação, por exemplo).
>
> Ela suporta meios para definir e manipular **abstrações de alto nível**, como **eventos** e **ações**.

Para representar a informação de gerência, a SNMA define e utiliza uma sintaxe própria, conforme descrito pela figura abaixo.

```
(ifInOctets - ifInOctets[1]) / @pollInterval
```

Figura 3.8. Definição de um objeto de alto nível

Na expressão acima, um *objeto de alto nível* foi definido com base na taxa de octetos recebida por uma determinada interface de um dos agentes da rede. Se considerarmos que um mesmo agente pode ter várias interfaces e que operações SNMA se baseiam em **conjuntos de**

agentes, a expressão acima significa “a taxa de octetos recebida por cada interface de cada agente presente num conjunto de agentes específico”. A variável `ifInOctets[1]` representa o valor de `ifInOctets` obtido na penúltima monitoração feita; `ifInOctets` representa o valor de `ifInOctets` obtido na monitoração corrente; e `@PollInterval` significa o intervalo de tempo entre duas monitorações consecutivas. O programador, portanto, especifica a expressão acima mas não precisa se preocupar com as operações de gerência realizadas para que tal informação seja obtida.

Para atender a necessidades de gerência específicas, a SNMA também define uma sintaxe própria, através da qual busca fornecer o nível de abstração desejado. Para atender a necessidades da **gerência de falhas**, por exemplo, a SNMA fornece os *construtores* descritos na tabela abaixo.

Construtor	Função
`POLLED-EVENT`	Descreve um evento síncrono.
`TRAP-EVENT`	Descreve um evento assíncrono.
`PROCEDURE`	Descreve um conjunto de ações SNMA a serem executadas em resposta à ocorrência de uma falha (por exemplo: SNMP-SET, LOG, TICKET, EXECUTE).
`AGENT-SET`	Especifica um conjunto de agentes sobre o qual as operações SNMA serão executadas.
`TRAP`	Define um filtro para a geração de eventos assíncronos.
`CONDITION`	Descreve uma situação de falha na rede.
`STATE`	Permite modelar a funcionalidade da aplicação como uma máquina de estados finita.

Tabela 3.2. Construtores SNMA

Então, suponha que se deseje monitorar dois servidores, S_1 e S_2. Para isto, é necessário especificar o conjunto formado por estes dois servidores como a seguir:

```
AGENT-SET  servidores {S1, S2}
```

Uma falha terá ocorrido se a taxa de erros de uma das interfaces, de um dos servidores, for maior que 40% do tráfego que passa através dela. É possível descrever este cenário da seguinte maneira:

```
CONDITION InterfaceDefeituosa {ifOutErrors - ifOutErrors[1] >
(ifOutPkts - ifOutPkts[1]) * 0.4}
```

Figura 3.9. Definição de uma condição de falha

Condições como esta podem ser utilizadas para definir eventos síncronos através do construtor `POLLED-EVENT` (cada construtor tem um conjunto de campos que o caracteriza). A condição é avaliada de acordo com um intervalo de tempo, `POLL-PERIOD`, e, uma vez satisfeita, uma ação (`PROCEDURE`) a ser executada em resposta à ocorrência do evento pode ser especificada. Na figura abaixo, o evento síncrono `ServidorForaDoAr` foi definido.

```
       POLLED-EVENT ServidorForaDoAr {
            CONDITION InterfaceDefeituosa
            POLL-PERIOD 15         #15 minutos
            ACTION corrigeFalha
            AGENT-SET servidores
       }
       PROCEDURE corrigeFalha {
            LOG {"Falha no servidor", @hit_location}
            SNMP-SET desligaInterface {ifAdminStatus = DOWN}
            EXECUTE {habilitaServidorRedundante}
       }
```

Figura 3.10. Definição de um evento síncrono

De acordo com a figura acima, o evento ocorrerá se condição `Interface Defeituosa`, avaliada a cada 15 minutos, for satisfeita. O procedimento `corrigeFalha`, gera um registro que identifica onde ocorreu a falha (`@hit_location`), altera o estado da interface para `DOWN` e executa um programa externo, `habilitaServidorRedundante`, o qual deve substituir o servidor com problemas por um outro servidor.

Analogamente, é possível definir um evento assíncrono, conforme descrito pela figura 3.11. O evento `ServidorOk` ocorrerá se a interface de um dos `Servidores` voltar ao seu estado de operação normal (`TRAP-KIND InterfaceOk`). Neste caso, executa-se um outro programa externo, `desabilitaServidorRedundante`, que deve fazer com que o servidor gerador do *trap* `InterfaceOk` volte à operação normal, desabilitando o servidor redundante que estava em seu lugar.

```
TRAP InterfaceOk {
       ORIGIN Servidores
       TYPE linkUp
    }
```

```
TRAP-EVENT ServidorOk {
  TRAP-KIND InterfaceOk
  ACTION desabilitaServidorRedundante
}
```

Figura 3.11. Definição de um evento assíncrono

Para modelar a funcionalidade da aplicação, devem-se definir *estados de execução*, através do construtor `STATE`. Um estado é definido como uma coleção de eventos. Quando uma aplicação está em determinado estado de execução, os componentes de monitoração de todos os eventos que constituem aquele estado estão ativos [Simões *et al.*, 1994]. Se a falha correspondente a qualquer um dos eventos ativos ocorrer, a ação associada àquela falha é executada e a aplicação passa para um novo estado. Veja o exemplo a seguir.

```
STATE OperaçãoNormal {
  ON-EVENT ServidorForaDoAr NEW-STATE EsperandoCorreção
}
STATE EsperandoCorreção {
  ON-EVENT ServidorOk NEW-STATE OperaçãoNormal
  ON-EVENT ServidorForaDoAr NEW-STATE EsperandoCorreção
}
```

Figura 3.12. Definição de estados de execução

De acordo com a figura acima, a aplicação monitora a ocorrência de uma falha (estado `OperaçãoNormal`, evento `ServidorForaDoAr`). Uma vez que ela tenha ocorrido, a aplicação deve habilitar um outro servidor que substituirá o servidor com problemas (`ACTION corrigeFalha`, evento `ServidorForaDoAr`). Neste momento, a aplicação vai para o estado `EsperandoCorreção`, onde deve esperar até que a falha seja efetivamente corrigida. Quando, então, o servidor principal voltar ao seu estado de operação normal (estado `EsperandoCorreção`, evento `ServidorOk`), a aplicação vai para o estado `OperaçãoNormal`. Estas atividades correspondem à figura abaixo.

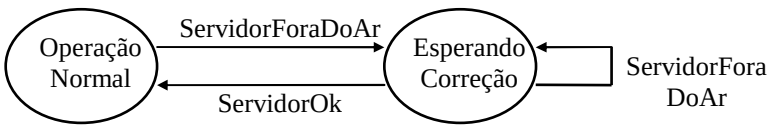


Figura 3.13. Diagrama de estados

Há ainda variáveis especiais que podem ser utilizadas, como é o caso de `@hit_location`, que indica o endereço do agente onde a última falha foi detectada; `@current_state`, que indica o nome do estado corrente; e `@SNMP_error`, que indica o código do último erro SNMP detectado. Estas variáveis podem ser muito úteis e o programador pode utilizá-las para definir objetos de alto nível e ações conforme foi exemplificado nas figuras 3.8 e 3.10.

3.5. Uma Análise das Soluções Apresentadas

Do exposto nas seções anteriores, verificamos que todas as APIs fornecem os recursos básicos para realização de coleta de informação na rede, tarefa indispensável para que a gerência de redes possa ser feita. O estabelecimento explícito de sessões SNMP e a chamada de primitivas de comunicação de baixo nível são aspectos que estão diretamente associados a esta tarefa. Entretanto, embora as APIs disponíveis forneçam um nível de abstração mais alto para a realização de operações SNMP, elas não fornecem as abstrações necessárias para que o programador possa se concentrar na solução desejada (gerência de falhas, gerência de configuração, etc.).

Em geral, para realizar a coleta de dados, programadores ainda precisam conhecer e manipular estruturas de dados complexas, tratar erros de comunicação decorrentes da operação do protocolo de gerência utilizado, tornar explícita a realização de operações de gerência, enfim, tratar de aspectos de baixo nível que não estão diretamente associados à funcionalidade específica de sua aplicação. Por mais importante, necessária e complexa que seja, a coleta de dados não é gerência de redes mas apenas um pré-requisito para essa gerência e, sendo assim, **programadores (gerentes e operadores de redes) não gostariam nem deveriam gastar muito esforço aqui**, mas gostariam de contar com outros recursos e mecanismos mais diretamente voltados a atender a necessidades de gerência específicas, como o **tratamento de falhas** ou a **definição de políticas de segurança**. Nesse sentido, as APIs disponíveis não fornecem uma “interface mais inteligente” capaz de prover maior reusabilidade e, conseqüentemente, maior produtividade.

A SNMA, por sua vez, representa algum ganho com relação às demais APIs apresentadas e citadas. Para obter informação de gerência, por exemplo, o programador não

precisa se preocupar com detalhes do protocolo de gerência utilizado. Além disso, a SNMA foi a única que se preocupou, em seu projeto, em identificar e prover abstrações de mais alto nível ao considerar as necessidades específicas de subáreas da gerência de redes. Entretanto, ela não se mostra uma notação suficientemente **flexível e reutilizável**.

Primeiramente, a SNMA é uma linguagem totalmente nova e ao definir uma sintaxe própria, novos interpretadores e compiladores tiveram que ser implementados. Para simplificar esta tarefa, apenas os seguintes "comandos essenciais de fluxo de controle" podem ser utilizados: `IF-THEN-ELSE`, `GOTO` e `RETURN`; e o programador deve se satisfazer com isto. Não parece conveniente, portanto, especificar uma linguagem totalmente nova, mas deve-se utilizar uma das linguagens hospedeiras disponíveis, **adicionando-lhe** as facilidades necessárias.

As abstrações (construtores) disponíveis também não podem ser estendidas ou modificadas para acomodar necessidades específicas de uma aplicação particular. Os construtores possuem um conjunto de campos invariável e o máximo que o programador pode fazer é omitir um ou mais campos opcionais conforme necessário. Não há a possibilidade de acrescentar campos, caracterizando melhor uma situação específica. Um construtor `POLLED-EVENT`, por exemplo, jamais poderia ter um campo que descrevesse a prioridade do evento, permitindo que filtros pudessem ser definidos com base nesta característica.

Também não há como se ter acesso aos campos dos construtores para modificar o comportamento da aplicação em tempo de execução. Por exemplo, não há como alterar o intervalo de monitoração (`POLL-PERIOD`) de um evento específico para permitir que, diante de determinada situação, tal evento possa ser mais rapidamente detectado a partir de uma monitoração mais freqüente. O máximo que se pode fazer é controlar o fluxo de execução (mudança de estados) com base na ocorrência de eventos pré-configurados.

Além disso, nenhuma API disponível, exceto a OVSNMP, fornece mecanismos mais elaborados para o **tratamento das falhas ocorridas**. Não há facilidades para fazer **correlação de eventos** (nenhuma informação sobre a topologia da rede gerenciada está disponível, por exemplo) nem **supervisão dos alarmes gerados** (definição de filtros de prioridade e mecanismos de notificação). Também não há facilidades para fazer **gerenciamento de *tickets***. O máximo que a SNMA permite é definir uma ação do tipo TICKET, através da qual podemos especificar um arquivo onde os *tickets* previamente configurados serão registrados.

Não há como manipular os vários *tickets* possivelmente abertos ou gerenciá-los de uma forma mais eficiente.

Outrossim, nenhuma API permite a interoperação de protocolos de gerência diferentes. Todas elas, pelo menos em suas versões atuais, implementam exclusivamente o SNMP e, assim, não é possível ter uma mesma aplicação que possa monitorar um agente SNMP e um agente DMI, por exemplo. Seria útil ter uma **API independente de protocolo** que permitisse alcançar este objetivo. Na tabela 3.3, comparamos as APIs e a notação SNMA apresentadas, listando suas principais características.

O nível de abstração desejado seria descrito pela seqüência de respostas “sim”. Nenhuma API, entre todas as estudadas, portanto, é capaz de fornecê-lo. Contudo, elas podem representar o ponto de partida para que soluções com melhor nível de abstração sejam implementadas.

APIs e Linguagens / **Características**	Tcl, LuaMan, SNMP-Perl	SNMP Manager API, SNMP++	SNMA	OVSNMP
Detalhes do protocolo de comunicação para a coleta de informação de gerência são escondidos do programador	Não	Não	Sim	Não
Fornece abstrações de alto nível para atender a necessidades de gerência específicas	Não	Não	Sim	Não
Fornece abstrações reutilizáveis e facilmente extensíveis	Não	Não	Não	Não
Mecanismos mais elaborados para o tratamento das falhas ocorridas	Não	Não	Não	Sim
Independência do protocolo de comunicação	Não	Não	Não	Não

Tabela 3.3. Algumas APIs de gerência e suas características

Embora uma lista completa de requisitos para uma solução ideal seja apresentada apenas no próximo capítulo, o nosso trabalho propõe uma solução para a gerência de falhas com uma finalidade básica: **fornecer um nível de abstração mais adequado ao desenvolvimento de aplicações de gerência de falhas**. Ao prover os mecanismos básicos e necessários para atender às necessidades deste tipo de aplicação e ao dispensar o programador de detalhes que não estão diretamente associados ao domínio do problema, o programador pode, finalmente, concentrar-se na obtenção da funcionalidade básica de sua aplicação. No capítulo seguinte, a especificação da solução proposta é descrita em detalhes.

Capítulo 4

Um *Framework* para a Gerência de Falhas

Como vimos no capítulo anterior, as APIs atualmente utilizadas para a implementação de aplicações de gerência não fornecem o nível de abstração adequado, de modo a facilitar, efetivamente, o trabalho do programador. De um modo geral, programadores precisam ter uma boa experiência de programação para atingir os seus objetivos no que diz respeito ao desenvolvimento de aplicações de gerência de redes.

A pergunta que surge, então, é: como prover este nível de abstração? Em nosso caso particular, a pergunta adequada seria: o que devemos prover para satisfazer as necessidades do **programador** de aplicações de gerência de falhas, o nosso **usuário**? Neste capítulo, descrevemos a especificação de uma solução que busca fornecer um nível de abstração mais adequado ao desenvolvimento de aplicações de gerência de falhas. Trata-se de um *framework* baseado em componentes de software reutilizáveis para a gerência de falhas.

O desenvolvimento de um *framework* se baseia nas etapas básicas de qualquer **processo de desenvolvimento** de um produto de software [Rumbaugh *et al*., 1999b]: levantamento de requisitos, análise do domínio do problema, projeto da solução, implementação e testes. Contudo, vale salientar que algumas metodologias especialmente voltadas para o desenvolvimento de *frameworks* [Landin & Niklasson, 1998; Johnson & Roberts, 1998] destacam um importante aspecto: as três primeiras etapas do processo de desenvolvimento se baseiam em exemplos de aplicações que mais tarde poderão ser desenvolvidas com base no *framework* resultante, ou seja, os requisitos e funcionalidade do *framework* são identificados a partir das características e requisitos comuns aos diversos exemplos de aplicações considerados. As três primeiras etapas foram realizadas para especificação do *framework* proposto neste trabalho e a linguagem de modelagem UML *(Unified Modeling Language*) [Rumbaugh *et al*., 1999a] foi utilizada.

Na seção 4.1, apresentamos o levantamento de requisitos realizado a partir do estudo e da análise de algumas aplicações de gerência de falhas. Os requisitos comuns aos exemplos considerados se transformaram em requisitos do *framework* e, além disso, são identificados outros requisitos que caracterizam o nível de abstração a ser fornecido. A análise do domínio do problema já foi sucintamente apresentada na seção 2.3.2, quando caracterizamos o comportamento genérico de uma aplicação de gerência de falhas.

O projeto do *framework* é apresentado ao longo das seções seguintes. Na seção 4.2, apresentamos um projeto arquitetural que define as arquiteturas interna e externa do *framework* e, com base nesta arquitetura, caracterizamos **dois perfis de usuários** que utilizarão o *framework* em momentos distintos: o **programador de novos componentes** e o **projetista de aplicações**. Na seção 4.3, descrevemos os componentes básicos do *framework*, detalhando a arquitetura interna definida na seção anterior. Na seção 4.4, são fornecidos alguns exemplos de aplicações construídas com base na especificação apresentada.

As características do *framework*, portanto, são apresentadas gradativamente ao longo do capítulo, à medida em que vamos atendendo aos requisitos levantados. No final da seção 4.3, é fornecido um resumo que reúne as características gerais da solução proposta e que destaca como os requisitos listados foram atendidos.

4.1. Requisitos da Solução

Uma vez que seja necessário propor uma solução para qualquer problema, a primeira coisa a se fazer é entender bem o problema. Só para citar o filósofo John Dewey, “um problema bem formulado já está, em parte, resolvido”. Assim, deve-se, antes de mais nada, identificar os requisitos da solução. Mais especificamente falando, deve-se identificar os requisitos que, uma vez atendidos, caracterizarão a solução que fornece o nível de abstração adequado para o desenvolvimento de aplicações de gerência de falhas.

A partir do estudo de alguns exemplos de aplicações de gerência de falhas e com base no exposto na seção 3.5, onde foi caracterizado o nível de abstração desejado, seguem abaixo os **requisitos funcionais** e os **requisitos não-funcionais**[5] a serem atendidos.

[5]Requisitos funcionais são aqueles requisitos que devem ser atendidos para prover a funcionalidade da solução. Requisitos não-funcionais especificam outras restrições impostas à solução, como aquelas relacionadas com a adoção de padrões e integração com outros sistemas, políticas externas, custos, desempenho, portabilidade, etc.

Requisitos funcionais

F1. Deve ser possível fazer monitoração síncrona e monitoração assíncrona.

A tarefa de monitoração consiste em obter, efetivamente, a informação de gerência mantida pelos agentes da rede (eventos são definidos com base nesta informação). Para isto, é necessário realizar dois tipos de monitoração: síncrona e assíncrona. No primeiro caso, a aplicação (gerente) deve buscar, periodicamente, a informação nos agentes gerenciados - tal periodicidade, ou freqüência, deve ser facilmente configurável. No segundo caso, é necessário preparar-se para receber, a qualquer momento, a informação enviada pelos agentes (*traps* SNMP, por exemplo).

F2. Deve ser possível descrever situações de falha na rede.

O primeiro objetivo de uma aplicação de gerência de falhas é identificar as falhas ocorridas na rede. Para isto, é preciso, antes de tudo, definir os **eventos** que as descrevem, pois é a partir da verificação da ocorrência destes eventos que torna-se possível detectar, automaticamente, as falhas ocorridas.

A definição de eventos se baseia na análise da informação de gerência colhida nos agentes da rede - tal informação, obtida através de monitoração síncrona ou assíncrona, deve estar disponível. Dizemos que um evento ocorreu quando a condição que o define for satisfeita. Por exemplo, o evento “rota inacessível” ocorrerá se uma alta taxa de perda de pacotes for observada na interface do roteador que dá acesso a esta rota. **Limiares** são comumente utilizados para definir estas condições, refletindo as políticas de gerência a serem aplicadas para atender necessidades específicas.

F3. Usando a informação de gerência disponível, deve ser possível definir eventos que descrevam situações de falha semanticamente mais ricas.

Muitas vezes, a detecção de uma falha (definição de um evento) é trivial no sentido em que ela resulta da análise de uma única variável de gerência em torno de um único limiar ou valor de referência. Por exemplo, o fato de que o estado de operação de determinada interface foi modificado (variável MIB

`ifOperStatus = DOWN`), indicando que ela está inoperante, pode representar uma falha.

Entretanto, outras vezes, uma falha pode resultar da análise de diferentes variáveis, inclusive variáveis mantidas por diferentes agentes. Considere os exemplos abaixo.

1. Suponha que seja necessário monitorar um determinada interface `i` do roteador `R`. A falha "interface `i` do roteador `R` está com defeitos" poderia resultar da verificação de que o número de pacotes descartados (variável MIB `ifOutDiscards`) ultrapassou determinado valor v_1 **e** o número de erros (variável MIB `ifOutErrors`) apresentados por tal interface ultrapassou um outro valor v_2.

2. Suponha agora, que seja necessário identificar um congestionamento em `R` e que `R` dá acesso a dois servidores multimídia **distintos**, S_1 e S_2. A falha "congestionamento em `R`", impossibilitando a comunicação de S_1 e S_2 com seus clientes remotos, poderia resultar da verificação de que a soma do tráfego gerado por S_1 e S_2 ultrapassou determinado valor.

Sendo assim, deve ser possível definir eventos que descrevam estas situações, ao analisar várias variáveis de um ou mais agentes em torno de um ou vários limiares. Para isto, é preciso ter o acesso devido à informação de gerência necessária.

F4. Deve ser possível fazer correlação de eventos.

Uma vez que os eventos tenham ocorrido, deve ser possível correlacioná-los, observando a relação de dependência que possa existir entre eles. Isto permitirá reduzir a quantidade de informação apresentada ao operador da rede ao mesmo tempo em que **acrescentará valor semântico aos eventos gerados**. Por exemplo, saber que uma determinada interface de um roteador foi "desligada" (variável MIB `ifAdminStatus = DOWN`) pode não representar uma falha; mas saber que esta mesma interface foi "desligada" três vezes no intervalo de uma hora pode representar uma situação de interesse. Neste caso, não só se reduz o número de eventos de 3 para 1, como também se acrescenta significado ao

evento resultante. Analogamente, descobrir que um conjunto de agentes tornou-se inacessível pode não ter qualquer relevância diante do fato de que o roteador através do qual se tem acesso a eles também está inoperante. Neste caso, os primeiros eventos gerados devem levar à conclusão de problemas no roteador, permitindo resolver o problema de forma mais eficaz.

A solução, portanto, deve fornecer meios para que a correlação possa ser feita, pois a partir do momento em que é possível considerar apenas a informação de verdadeira relevância, adicionando-lhe valor, torna-se possível fazer diagnósticos mais precisos e, conseqüentemente, um gerenciamento de falhas mais eficiente. Entre outras coisas, a solução deve fornecer informação sobre a **topologia** da rede para que se possa tirar conclusões sobre a influência que alguns eventos exercem sobre outros, como no caso do último exemplo descrito.

F5. Deve ser possível automatizar tarefas em resposta aos eventos ocorridos.

É preciso tratar as falhas ocorridas na rede, sejam elas resultantes de uma correlação ou não. Neste sentido, o maior grau de automação possível deve ser fornecido, idealmente dispensando a intervenção humana. Mecanismos para supervisão e notificação de alarmes e facilidades para a geração de registros de ocorrências devem ser fornecidos.

F6. Deve ser possível realizar diferentes ações em resposta à ocorrência de um único evento.

Algumas vezes, a ocorrência de um único evento pode *disparar* diversas ações. Por exemplo: ao detectar que o tráfego que passa por uma das interfaces de um roteador ultrapassou determinado valor, pode ser necessário (1) obter informação sobre esta interface mais freqüentemente, modificando o processo de monitoração; (2) gerar um alarme, notificando o operador da rede sobre a ocorrência do evento; e (3) registrar a ocorrência numa base de dados, atualizando o histórico sobre os eventos ocorridos. A solução deve permitir que todas estas ações sejam realizadas em resposta à ocorrência do mesmo evento.

F7. Deve ser fornecido um controle máximo sobre o comportamento da aplicação através da parametrização e/ou configuração dos elementos que implementam sua funcionalidade.

Ao construir qualquer aplicação ou sistema, **parâmetros** devem ser devidamente ajustados a fim de caracterizar um comportamento específico - a periodicidade de monitoração, por exemplo, deve ser um parâmetro ajustável, de modo que se possa determinar com que freqüência a aplicação deve buscar a informação de gerência nos agentes.

A solução deve permitir atuar convenientemente sobre tais parâmetros para que se possa, inclusive, modificar o comportamento da aplicação em tempo de execução. Conforme exemplificado na seção 3.5, é possível que seja necessário modificar o processo de monitoração haja vista a ocorrência de um determinado evento (ao detectar que determinado evento ocorreu, deveria ser possível mudar a freqüência de monitoração). A solução deve permitir que isto seja feito.

F8. Deve ser possível fazer *trouble-ticketing*.

Aplicações de *trouble-ticketing* são muito utilizadas no contexto da gerência de falhas. Uma vez que um problema não possa ser imediatamente resolvido (a correção de um problema pode envolver fabricantes ou pessoas mais capacitadas, o que, em geral, leva tempo), o operador da rede deve poder "abrir" um *ticket*, mantendo um histórico de resolução do problema (origem, data de ocorrência, data de resolução, ações realizadas, responsável, etc.), ou a própria solução de gerência deve poder abri-lo automaticamente, dispensando a intervenção manual do operador. A solução, portanto, deve fornecer meios através dos quais seja possível definir e gerenciar *tickets*, bem como deve fornecer a informação e mecanismos necessários para que *tickets* possam ser automaticamente abertos.

F9. Deve ser possível reaplicar uma determinada configuração do ambiente gerenciado sempre que necessário.

Para que a gerência de falhas possa ser realizada, é necessário, antes de tudo, descrever o ambiente gerenciado. Em outras palavras, é necessário definir quais

agentes da rede devem ser gerenciados e qual informação de gerência deve ser observada em cada agente. Muitas vezes, a mesma configuração se repete em aplicações distintas, por exemplo: várias aplicações poderiam estar interessadas em monitorar um roteador com uma quantidade “x” de interfaces, sendo necessário obter a taxa de tráfego que passa através de cada uma delas para identificar problemas de congestionamento. A solução deve fornecer, portanto, mecanismos que permitam reaplicar uma determinada configuração sempre que necessário.

F10. Não é necessário permitir a modificação das variáveis de gerência mantidas pelos agentes da rede.

No protocolo SNMP, o comando `set` permite modificar o valor de algumas variáveis MIB. Entretanto, além de causar potenciais problemas de segurança - em seu projeto inicial, o SNMP não se preocupou devidamente com segurança - é aceitável dispensar a sua utilização para o propósito da gerência de falhas.

Das 190 variáveis presentes na MIB-II padrão, apenas 30 podem ser modificadas (campo `ACCESS = read-write`). Destas 30, 20 descrevem informação sobre roteamento (grupo IP da MIB-II) e servem basicamente para indicar as rotas e as métricas a serem utilizadas. A configuração de rotas, por sua vez, depende normalmente de outros protocolos específicos, de forma que alterações feitas nestas variáveis não surtirão os efeitos desejados. De fato, a configuração de dispositivos é raramente feita através de protocolos de gerência de redes, e, sendo assim, parece aceitável dispensar o comando `set` em nosso contexto específico, haja vista que, para identificar e diagnosticar falhas, é necessário e suficiente **recuperar** (ler) os valores das variáveis de interesse.

Requisitos não-funcionais

NF1. A solução deve ser independente de protocolo de gerência.

A solução deve ser suficientemente flexível a ponto de acomodar qualquer modelo de gerência potencialmente suportado. Além disso, ela deve permitir a interoperação entre os diferentes modelos para que seja possível construir

aplicações capazes de gerenciar, simultaneamente, diferentes agentes independentemente do modelo de gerência que eles suportam.

NF2. A solução deve fornecer, por *default*, suporte ao SNMP em todas as suas versões: SNMPv1, SNMPv2C e SNMPv3.

Embora seja independente de protocolo de gerência, a solução deve fornecer o suporte *default* ao SNMP.

NF3. Os detalhes de baixo nível diretamente relacionados com o protocolo de gerência utilizado devem ser transparentes para o programador.

Neste sentido, enumeramos os seguintes tópicos:

NF3.1. Deve ser transparente para o programador de aplicações o uso das primitivas de comunicação de baixo nível como `get` ou `get-next`, bem como o estabelecimento de sessões de gerência. O programador está apenas interessado em utilizar a informação de gerência e não em **como** recuperá-la.

NF3.2. Devem ser fornecidos mecanismos automáticos de recuperação de erros decorrentes da utilização do protocolo de gerência utilizado. No protocolo SNMP, um erro do tipo `tooBig`, por exemplo, o qual indica que uma resposta a uma requisição SNMP é muito grande para ser enviada num único pacote de informação, nada tem a ver com a funcionalidade implementada pela aplicação e, conseqüentemente, deve ser tratado de maneira tal que o programador nem tome conhecimento de sua ocorrência - se a resposta é realmente maior que o tamanho máximo do pacote de informação que pode ser enviado, então que a requisição seja, automaticamente, refeita (*quebrada* em sub-requisições), de maneira a contornar o problema.[6] Além disso, quando uma auto-recuperação não puder ser feita, a aplicação deve ser devidamente notificada.

[6]Algumas APIs já fornecem opções para recuperação automática de erros SNMP. É o caso do método `setPduFixedOnError` da API de gerência Java vista na seção 3.2.

NF3.3. Deve ser possível definir expressões de alto nível, calculadas a partir das variáveis de gerência mantidas nos agentes. Por exemplo, para definir o evento "congestionamento no roteador R" descrito anteriormente no requisito funcional de número 3, seria necessária a seguinte expressão :

$$\texttt{TaxaDeTráfegoEmS}_1 + \texttt{TaxaDeTráfegoEmS}_2 > \texttt{Limiar}$$

(os valores acima utilizados são obtidos a partir de variáveis MIB adequadas, por exemplo). A solução deve permitir definir e utilizar expressões deste tipo em qualquer lugar em que a análise da informação de gerência colhida nos agentes se fizer necessária.

NF3.4. A gerência de memória deve ser feita automaticamente. A alocação e a liberação das estruturas de dados necessárias para a realização das operações de gerência não devem ser de responsabilidade do programador. Obviamente que, ao configurar sua aplicação, ele terá que especificar o que deve ser instanciado mas ele não terá que se preocupar em **como** isto será feito.

NF4. Deve haver abstrações de alto nível facilmente utilizáveis e extensíveis que atendam aos requisitos funcionais anteriormente listados, facilitando a implementação da funcionalidade da aplicação.

Não deve ser necessário, portanto, manipular estruturas de dados complexas nem utilizar características de uma linguagem particular que exijam considerável experiência do programador no momento de utilizar ou estender as abstrações disponíveis. Desta forma, escrever uma aplicação pequena, que monitore algumas situações de falha específicas em dezenas de agentes, não deve custar mais de 1 hora de trabalho.

NF5. Deve ser possível monitorar centenas de agentes em poucos minutos.

Trata-se aqui de uma questão de desempenho. A aplicação não deve sofrer os efeitos de uma monitoração lenta em decorrência da quantidade de agentes que devem ser monitorados e da quantidade de informação que deve ser obtida.

NF6. A solução deve exigir o mínimo de programação.

Idealmente, deve ser possível construir a aplicação visualmente.

NF7. A solução deve fornecer portabilidade .

A aplicação de gerência resultante deve ser facilmente transportável para os principais ambientes operacionais em uso (Windows NT e UNIX).

Estes são, portanto, os requisitos básicos que devem ser atendidos de modo a satisfazer as necessidades do programador de aplicações de gerência de falhas. Certamente, uma solução final exige uma lista de requisitos mais completa que considere, por exemplo, outros aspectos relacionados a desempenho ou segurança. Neste trabalho, entretanto, concentramo-nos nos requisitos listados na certeza de que o atendimento a tais requisitos já caracterizam bem a solução a ser fornecida. A partir da próxima seção, é apresentada uma solução que atende a tais requisitos de forma conveniente.

4.2. O Projeto Arquitetural

Para atender aos requisitos listados na seção anterior, especificamos um *framework* baseado em componentes de software reutilizáveis. Nesta seção, consideramos que o leitor tem um conhecimento básico sobre os principais conceitos envolvidos, ou seja, *frameworks* e componentes de software. Se este não for o caso, recomendamos a leitura prévia dos apêndices A e B, onde é fornecida uma visão geral sobre tais conceitos na ordem em que foram citados. Este conhecimento prévio é essencial para o bom entendimento e fácil leitura do que se segue. Feito isto, vejamos como os conceitos envolvidos podem ser utilizados para atender aos requisitos levantados.

4.2.1. *Framework* e Componentes

A fim de encontrar uma solução adequada, reconsidere os requisitos listados. Primeiramente, de acordo com os requisitos NF4 e NF6, a solução deve permitir o desenvolvimento rápido de aplicações de gerência de falhas (em questão de horas) ao exigir o mínimo de programação e ao fornecer abstrações facilmente utilizáveis. Para isto, um **alto grau de reusabilidade** deve ser fornecido, permitindo ao programador concentrar seus esforços na obtenção da funcionalidade específica de sua aplicação. Seria possível, portanto, fornecer uma biblioteca de classes mas a reutilização de código obtida através da utilização de bibliotecas de classes em geral, pode não ser suficiente.

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, 1998]. É durante estas etapas que as principais decisões são tomadas (arquitetura da solução e escolha de tecnologias, por exemplo) e os erros cometidos podem custar muito caro - um problema mal entendido não pode ser bem resolvido e uma solução mal pensada jamais resolverá o problema em questão; em outras palavras, jamais atenderá às necessidades do usuário. Sendo assim, é possível concluir que para reduzir efetivamente o tempo gasto durante o desenvolvimento de uma aplicação, é preciso fornecer não só reutilização de código, mas também de **análise** e de **projeto**, de modo a reutilizar o conhecimento empregado e as principais decisões tomadas durante estas etapas. Em outras palavras, deve-se fornecer um ***framework*** para a gerência de falhas.

Frameworks são soluções quase prontas que **maximizam** a reusabilidade fornecida ao permitir reutilização de análise, projeto, implementação e testes. Eles capturam o que há de comum entre várias aplicações de um domínio de problema bem delimitado e podem ter seu comportamento **modificado** e/ou **estendido** de modo a acomodar características específicas das aplicações que o utilizam [Landin & Niklasson, 1998]. Assim, um *framework* determina a arquitetura das aplicações construídas com base nele ao capturar as decisões de projeto que são comuns ao domínio de problema no qual a aplicação está inserida, permitindo ao programador se concentrar nas particularidades de sua aplicação. Deve-se fornecer um *framework* que capture o comportamento genérico apresentado por aplicações de gerência de falhas e que permita reutilizar a arquitetura resultante para gerenciar falhas na rede.

Conforme explicado no apêndice A, *frameworks* são geralmente implementados através da **orientação a objetos** (OO). Desta forma, pode-se definir um *framework* como um conjunto de classes concretas e abstratas que cooperam entre si, fornecendo um projeto reutilizável para uma classe específica de software [Gamma *et al*., 1994]. Para construir uma aplicação com base num *framework* OO, deve-se estender as classes abstratas fornecidas, introduzindo as especificidades da aplicação, e o resto fica a cargo do *framework* que, de fato, tem o controle sobre a funcionalidade da aplicação desenvolvida.

Entretanto, de modo a permitir a **construção visual da aplicação** (ainda conforme requisito NF6), facilitando ainda mais a construção de aplicações de gerência de falhas, deve-

se ter não um *framework* OO, mas um *framework* **CO** (*Component-Oriented*), isto é, um *framework* baseado em **componentes de software reutilizáveis** [D'Souza & Wills, 1999]. Ao utilizar componentes, é possível construir a aplicação através de um **ambiente gráfico**, ao **configurar** e **interconectar** os componentes disponíveis.

Num *framework* CO, analogamente ao que ocorre com objetos no caso de um *framework* OO, cada componente implementa uma parte bem definida da funcionalidade geral do *framework* e, da cooperação entre os componentes utilizados, obtém-se a funcionalidade da aplicação. Para permitir tratar características específicas de cada aplicação, entretanto, em vez de fornecer simplesmente classes abstratas que podem ser devidamente estendidas (o que ocorre no caso de um *framework* OO), um *framework* CO fornece componentes que podem ser **visualmente** configurados. Além disso, ele pode fornecer **componentes semiprontos** a partir dos quais novos componentes podem ser criados. Cada novo componente criado passa a fazer parte do *framework* e, uma vez disponível, pode ser também configurado através de um ambiente gráfico, permitindo, finalmente, a construção visual da aplicação. A figura 4.1. mostra este processo.

Na figura 4.1, tem-se um *framework* CO que possui seus próprios componentes (CP1, CP2, CP3) e que fornece componentes semiprontos (CS1, CS2) a partir dos quais o programador pode construir novos componentes. À medida em que uma nova aplicação é construída, necessidades particulares podem ser identificadas, por exemplo, a necessidade de verificar a ocorrência de uma determinada falha. Neste momento, para fazer com que o *framework* atenda às novas necessidades identificadas, o programador pode acrescentar o código necessário (P1, P2) ao *framework*, construindo novos componentes (CN1, CN2) a partir dos componentes semiprontos fornecidos. Uma vez que os componentes necessários à aplicação estejam disponíveis, o programador pode simplesmente instanciá-los através de um ambiente gráfico, configurando suas propriedades e interconectando-os (i_1, i_2, ...) conforme necessário.

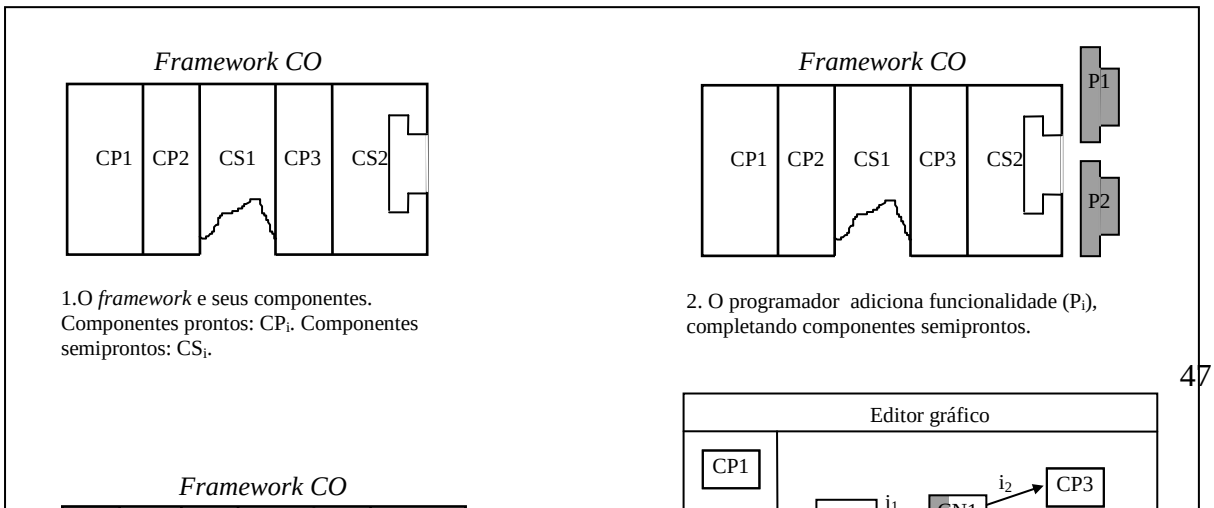


Figura 4.1. Construindo uma aplicação com base num *framework* CO

No caso em que se utilize uma linguagem OO para implementação dos componentes, a construção de novos componentes resulta da extensão de classes ou da redefinição de métodos que definem os componentes semiprontos fornecidos pelo *framework*, como veremos em detalhes na seção 4.3. Entretanto, o *framework* ainda pode fornecer outras classes e interfaces - no caso de se utilizar a linguagem Java, por exemplo - para permitir a construção de componentes novos, resultantes da extensão destas classes ou da implementação das interfaces fornecidas.

Veja a figura 4.2. Nela, o *framework* é visto como um conjunto não só de componentes prontos e semiprontos (CP_i e CS_i, respectivamente), mas como um conjunto que possui, além de componentes, outras classes (Cl_i) e interfaces (I_i) que podem ser diretamente utilizadas pelo programador. Cada componente, por sua vez, é definido em termos de suas próprias classes e interfaces, e componentes semiprontos permitem ter seu comportamento modificado através de uma de suas classes - por exemplo, extensão da classe cl_2 do componente CS_1.

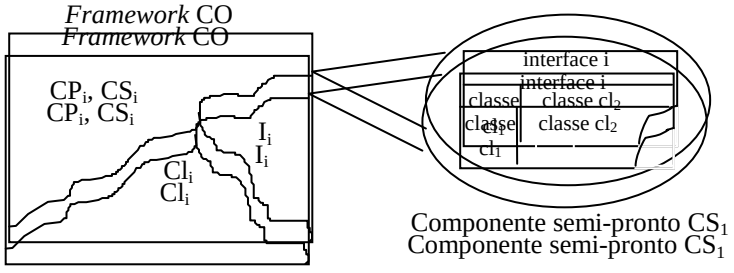


Figura 4.2. Componentes, classes e interfaces do *framework* CO

Assim, para adicionar funcionalidade ao *framework*, o programador pode utilizar componentes semiprontos que agrupam suas próprias classes e interfaces, bem como outras classes e interfaces fornecidas pelo *framework*. Cada componente semipronto pode ser visto como um *subframework* que permite modificar não só o seu comportamento interno, mas o comportamento do *framework* como um todo.

Com base na arquitetura apresentada, é possível identificar **dois** tipos de usuários que utilizarão o *framework* em momentos diferentes:

há o **programador de novos componentes** - aquele que constrói componentes ao identificar novas necessidades durante o desenvolvimento de determinada aplicação; e

há o **projetista de aplicações** - aquele que constrói a aplicação visualmente, instanciando, configurando e interconectando os componentes disponíveis.

O primeiro tipo de usuário, ou programador de novos componentes, certamente deve ter um maior conhecimento em termos de gerência de redes e deve ter um conhecimento básico sobre a linguagem de programação utilizada para implementar e estender os componentes semiprontos. No caso em que seja necessário implementar outras interfaces e/ou estender outras classes especificadas pelo *framework*, uma experiência maior é necessária. Além disso, o programador de novos componentes tem o compromisso de construir componentes genéricos e reutilizáveis que podem ser utilizados toda vez que a mesma necessidade seja identificada. Cada componente criado - seja ele através de um componente semipronto, seja ele através de outras classes e interfaces - pode ser utilizado pelas diversas aplicações que devem atender a uma necessidade comum.

O projetista de aplicações, por sua vez, pode utilizar o *framework* num nível de abstração mais alto. Ele deve "compor" a aplicação final ao reutilizar os componentes disponíveis, manipulando estes componentes graficamente.

4.2.2. Fontes e Consumidores de Eventos

Definida a **arquitetura externa** da solução, é preciso definir a **sua arquitetura interna**, isto é, deve-se identificar os componentes do *framework* que devem estar disponíveis para que o programador possa construir sua aplicação e deve-se definir uma forma de interconectá-los. Em outras palavras, deve-se definir como o comportamento genérico de uma

aplicação de gerência de falhas, representado pela figura 2.4 e sujeito aos requisitos F1 a F8, será capturado pelo *framework*.

De um modo geral, aplicações de gerência de falhas realizam monitoração nos agentes a fim de obter a informação de gerência de interesse, analisam esta informação a fim de identificar falhas na rede e, uma vez verificada a ocorrência de uma falha, as ações cabíveis devem ser realizadas de modo a resolver o problema. Com base nisto, foram identificados os **componentes básicos** do *framework*, responsáveis pela realização de cada uma destas etapas (detalharemos estes componentes na próxima seção), e a fim de **embutir** este **comportamento** no *framework*, um modelo de execução configurável, baseado em **fontes e consumidores de eventos**, foi adotado. Tal modelo, também conhecido como o padrão de projeto *Observer* [Gamma *et al.*, 1994], define como interconectar os componentes utilizados durante a construção da aplicação.

De acordo com este modelo de execução, todo componente é um gerador potencial de eventos. Um evento neste contexto, é qualquer resultado decorrente da ação de um componente, não representando apenas falhas. Se o componente A está interessado no evento gerado por outro componente B, A deve se cadastrar em B, declarando seu interesse em receber tal evento. B, por sua vez, compromete-se a enviar os eventos gerados para A e todos os demais componentes cadastrados. O acoplamento entre fontes e consumidores é mínimo, de modo que eles podem ser reutilizados separadamente. Um consumidor pode se cadastrar ou cancelar seu cadastro junto a uma determinada fonte dinamicamente, sem alterar a fonte ou qualquer outro consumidor já cadastrado. A figura 4.3 descreve a situação em que A é consumidor dos eventos gerados por B. Adicionalmente, B também deve enviar os eventos gerados para C.

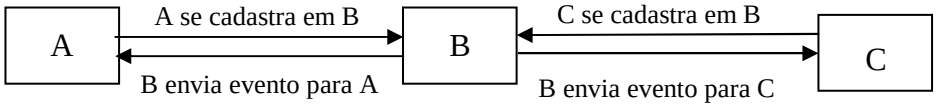


Figura 4.3. Fonte (B) e consumidores de eventos (A e C)

Um mesmo componente ainda pode ser, ao mesmo tempo, fonte e consumidor de eventos (componente-processador), de modo que, qualquer fluxo de execução resultante da associação entre os componentes do *framework* pode ser genericamente descrito pela figura abaixo. Nesta figura, há um **componente-fonte** (F), um **componente-consumidor** (C) e um

P se cadastra em F C se cadastra em P

componente-processador (P). Este último componente é consumidor de eventos de F e fonte de eventos para C simultaneamente.

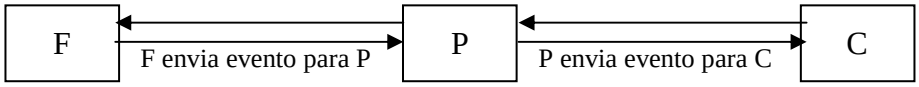


Figura 4.4. Modelo geral de execução

Segue um exemplo prático da utilização deste modelo. Suponha que haja um componente responsável por fazer monitoração síncrona (`Monitor M`), outro responsável por verificar a ocorrência da falha específica "interface do roteador R com defeitos" (`GeradorDeEventos` G_E) e outro responsável por gerar um alarme quando a falha ocorrer (`GeradorDeAlarmes` G_A). Para que o operador da rede seja notificado sobre a ocorrência da falha descrita, o `GeradorDeEventos` G_E deve se cadastrar no `Monitor M`, declarando o seu interesse em receber a informação de gerência decorrente da monitoração – a falha será identificada a partir da análise da informação recebida. Além disso, o `GeradorDeAlarmes` G_A deve se cadastrar no G_E, declarando o seu interesse em ser notificado sempre que a falha ocorrer, para, assim, poder notificar o operador da rede. O fluxo de execução descrito pela figura 4.5 reflete esta situação.

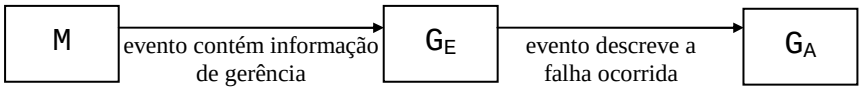


Figura 4.5. Fluxo de eventos `Monitor-GeradorDeEventos-GeradorDeAlarmes`

Além disso, é possível definir fluxos arbitrários de execução de acordo com as necessidades a serem atendidas por cada aplicação. Conforme descrito na seção 4.1, deve ser possível realizar diferentes ações em resposta à ocorrência de um único evento (requisito **F6**) ou modificar o processo de monitoração diante da ocorrência de um evento. No primeiro caso (situação A, figura 4.6), os diferentes componentes (A_1, A_2, A_3) que realizam cada uma das ações necessárias estão cadastrados no mesmo `GeradorDeEventos` (G_E) e o G_E, então, repassa o evento gerado para todos eles (requisito **F6** satisfeito). No segundo caso (situação B, figura 4.6), o `Monitor M` está cadastrado no `GeradorDeEventos` G_E e G_E está cadastrado

em M. M recebe o evento cuja geração resultou da análise da informação de gerência que ele próprio forneceu e, então, pode modificar seu comportamento de forma conveniente.

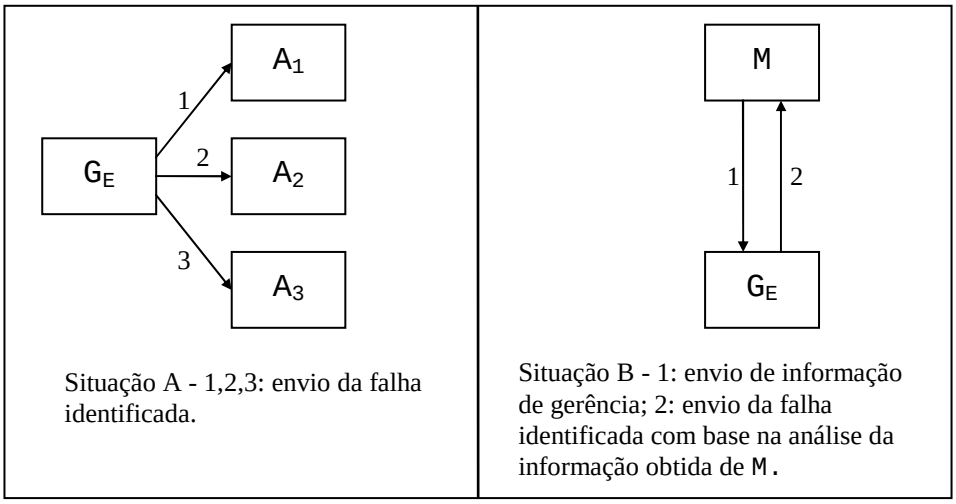


Figura 4.6. Exemplos de fluxos de eventos

Na verdade, qualquer grafo pode ser construído para representar as interações estabelecidas entre os componentes envolvidos, desde que sejam obedecidas algumas regras de interconexão de componentes, como será descrito na seção a seguir. Ao interconectar os componentes de forma adequada, é possível definir um comportamento específico, **reutilizando** a arquitetura imposta pelo *framework.*

Enfim, ao combinar *framework*, componentes e um modelo de execução baseado em fontes e consumidores de eventos, atende-se (arquiteturalmente falando) aos requisitos destacados. Na seção seguinte, detalhamos os componentes básicos do *framework*, destacando o atendimento a outros requisitos não mencionados nesta seção.

4.3. Os Componentes Básicos do *Framework*

Como vimos, a solução consiste de um *framework* CO que fornece componentes através dos quais aplicações de gerência de falhas podem ser desenvolvidas. Para desenvolver cada aplicação, o programador pode (1) construir novos componentes, (2) configurar visualmente os componentes disponíveis e (3) interconectar os componentes necessários através de um modelo de execução simples e flexível para que, da cooperação entre os

componentes interconectados, obtenha-se a funcionalidade de cada aplicação. Resta saber, então:

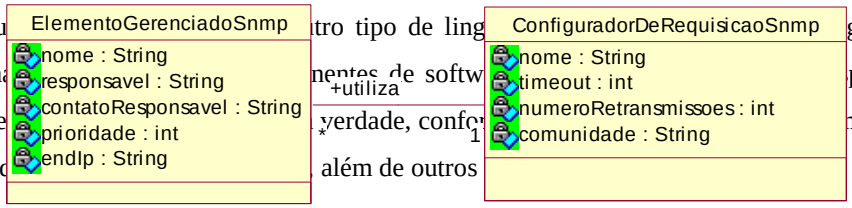
1. Quais são os componentes que o *framework* deve fornecer?
2. Quais componentes podem ser simplesmente configurados e quais podem ser "completados" de forma a originar novos componentes conforme necessário?
3. Quais classes e interfaces adicionais devem estar disponíveis?
4. Como os componentes instanciados podem ser interconectados, isto é, quem produz e quem consome eventos?

Mais uma vez, tendo em vista os requisitos listados na seção 4.1, é possível identificar os componentes, classes e interfaces a serem fornecidos pelo *framework*, bem como as relações que podem ser estabelecidas entre os componentes instanciados. Tudo isto será descrito a partir de agora[7].

4.3.1. Descrevendo a Rede Gerenciada

Antes de qualquer coisa, o *framework* deve permitir configurar o ambiente a ser gerenciado. Em outras palavras, para construir qualquer aplicação de gerência, é necessário descrever quais entidades da rede (roteadores, *switches*, estações de trabalho, etc.) devem ser gerenciadas e qual a informação de interesse. Para isto, três componentes foram especificados: `ElementoGerenciadoSnmp`, `InfoGerenciaSnmp` e `GrupoInfoGerencia`[8].

A figura abaixo mostra o componente `ElementoGerenciadoSnmp`. Através de suas propriedades é possível configurar cada uma das entidades da rede com as quais a



[7] Neste capítulo, utilizamos uma classe UML para representar um componente por falta de um símbolo mais apropriado. A lingu tro tipo de ling gramação não fornecem suporte na nentes de softw põem de uma simbologia mais ade verdade, confo nponente pode ser formado não só d além de outros

[8] Os componentes `ElementoGerenciadoSnmp`, `InfoGerenciaSnmp` e `ReceptorDeTrapsSnmp` cuidam de características específicas do SNMP. Mais a seguir, para satisfazer o requisito NF1, veremos como a independência de protocolo de gerência pode ser conseguida.

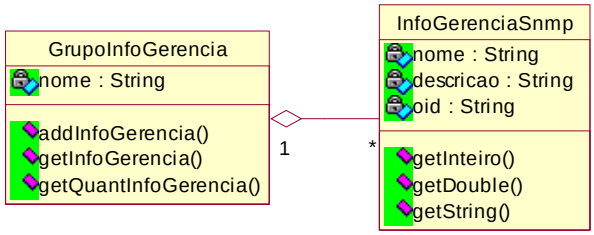
aplicação deve estabelecer algum tipo de comunicação[9]. Em todas as figuras apresentadas nesta seção, cada propriedade tem dois métodos de acesso que permitem obter (`get`) e atualizar (`set`) o seu valor.

Figura 4.7. `ElementoGerenciadoSnmp`

Cada `ElementoGerenciadoSnmp` é identificado através de um `nome`, o qual, conseqüentemente, deve ser único para cada `ElementoGerenciadoSnmp` instanciado pela aplicação. Além disso, ele possui um endereço IP (`endIp`) que permite localizar o agente SNMP representado. Este tipo de componente fornece informação de gerência e é através dele que a aplicação obtém a informação necessária para identificar falhas na rede.

Os parâmetros de configuração de cada requisição de gerência a ser realizada por determinado `ElementoGerenciadoSnmp` – como o *timeout* da requisição ou o número de tentativas que devem ser feitas até que a requisição execute adequadamente – são especificados através do componente `ConfiguradorDeRequisicaoSnmp`. Este componente deve ser associado a um `ElementoGerenciadoSnmp` para que a configuração feita através dele seja utilizada. Se nenhum configurador for instanciado e associado ao `ElementoGerenciadoSnmp`, uma configuração *default* é utilizada.

Para determinar qual informação deve ser fornecida pelo `ElementoGerenciadoSnmp`, os componentes `InfoGerenciaSnmp` e `GrupoInfoGerencia` foram especificados. Vide figura 4.8.



[9]No capítulo 5, fornecemos uma descrição mais detalhada de cada componente. Nesta seção, destacamos apenas os aspectos de maior relevância do ponto de vista do usuário que vai "compor" a aplicação visualmente.

Figura 4.8. `GrupoInfoGerencia` e `InfoGerenciaSnmp`

De acordo com a figura acima, um `GrupoInfoGerencia` é um agregado de componentes do tipo `InfoGerenciaSnmp`. Cada `InfoGerenciaSnmp`, por sua vez, representa uma variável MIB mantida pelo agente monitorado e possui três propriedades principais:

1. um `nome` que o identifica univocamente dentro do grupo ao qual pertence (“`TaxaDeErros`”, por exemplo);
2. um `oid` que corresponde ao identificador específico da variável MIB representada (“`ifInErrors`”, por exemplo);
3. um valor que deverá ser retornado pelo `ElementoGerenciadoSnmp` e que pode ser acessado através dos métodos `getInteiro`, `getDouble` e `getString`.

Operações de gerência se baseiam nestes grupos, de maneira que ao invés de fornecer uma única variável MIB (`InfoGerenciaSnmp`), um `ElementoGerenciadoSnmp` pode fornecer um grupo (`GrupoInfoGerencia`) que reúne várias destas variáveis.

Mas por que agrupar componentes `InfoGerenciaSnmp` em um componente `GrupoInfoGerencia`? Segundo o requisito **NF5**, deve ser possível monitorar centenas de entidades gerenciadas em poucos minutos. Para conseguir isto, uma das coisas que pode ser feita é reduzir o número de requisições realizadas através da rede. De fato, de acordo com [Tanenbaum, 1998], para melhorar a eficiência de aplicações distribuídas é preciso reduzir o número de requisições que devem ser feitas, sendo preferível recuperar muita informação em uma única requisição a recuperar pouca informação em várias requisições. Desta forma, operações de gerência se baseiam em grupos que reúnem várias unidades de informação para permitir recuperar, de uma única vez, o máximo de informação de gerência possível. Há, portanto, componentes do tipo `GrupoInfoGerencia` que agrupam componentes do tipo `InfoGerenciaSnmp`. Para especificar qual informação de gerência um determinado `ElementoGerenciadoSnmp` deve fornecer, o programador instancia vários componentes `InfoGerenciaSnmp` e os agrupa num `GrupoInfoGerencia`, de maneira que o `ElementoGerenciadoSnmp` possa fornecer o grupo requisitado. As APIs vistas no capítulo 3 também empregam a mesma idéia.

4.3.2. Monitorando a Rede

Com os três componentes vistos até agora, é possível identificar as entidades SNMP da rede que devem ser gerenciadas (`ElementoGerenciadoSnmp`) e qual informação de gerência (`GrupoInfoGerencia`) deve ser fornecida por cada uma delas. Para controlar a freqüência com que tais entidades devem fornecer a informação requisitada, um outro componente foi especificado. Trata-se do `Monitor`, mostrado na figura 4.9.

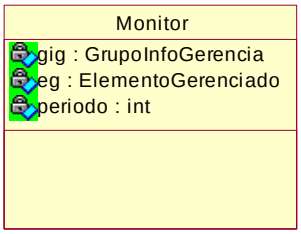


Figura 4.9. `Monitor`

O `Monitor` é o componente responsável pela realização de **monitoração síncrona** (requisito **F1**). Um `Monitor` pede ao `ElementoGerenciado` ao qual ele está associado (`ElementoGerenciado` é uma interface implementada por um `ElementoGerenciadoSnmp` e será descrita mais adiante, na seção 4.3.5) para que forneça a informação descrita por um `GrupoInfoGerencia` de acordo com uma periodicidade configurada pelo programador da aplicação (`periodo`). Em outras palavras, se um `Monitor M` está associado a um `ElementoGerenciado eg`, se ele está interessado em obter o `GrupoInfoGerencia gig` deste `eg` e se possui um `periodo` de 300 segundos, significa que `M` pedirá `gig` a `eg` a cada 5 minutos. O `ElementoGerenciado` cuida de todos os detalhes para que a informação requisitada possa ser efetivamente recuperada através da rede e retornada para o `Monitor`, de maneira que o programador **não** precise se preocupar com a abertura de sessões ou o envio de requisições (`get`) SNMP, por exemplo. Tudo é feito automaticamente pelo *framework* em atendimento ao requisito **NF3**, sendo transparentes para o programador os detalhes de mais baixo nível diretamente relacionados ao protocolo de gerência utilizado. Para coletar informação de gerência nas entidades gerenciadas, o programador só precisa instanciar e configurar os componentes necessários.

Uma instância do componente `Monitor` pode ser associada a uma única entidade gerenciada (`ElementoGerenciado`), de modo que um `Monitor` obtém informação de

gerência (`GrupoInfoGerencia`) de uma única entidade da rede. Por outro lado, para fazer com que uma mesma entidade forneça diferentes grupos de informação de gerência com diferentes freqüências, pode-se associar vários monitores à mesma entidade gerenciada. A figura 4.10 descreve ambas as situações.

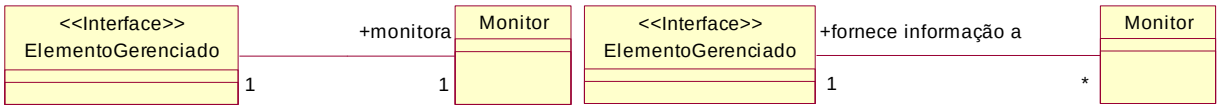


Figura 4.10. Relações `Monitor-ElementoGerenciado` e `ElementoGerenciado-Monitor`

Então, se cada instância de um `ElementoGerenciado` é controlada por seu(s) próprio(s) monitor(es), as diferentes instâncias de um `ElementoGerenciado` que, em geral, existem numa mesma aplicação, podem fornecer informação de gerência independentemente umas das outras e, inclusive, de forma **concorrente**. Em decorrência disto, é possível monitorar várias entidades da rede paralelamente, melhorando o desempenho da aplicação e reforçando-se o atendimento ao requisito **NF5**.

Para fazer **monitoração assíncrona**, ainda conforme requisito **F1**, o *framework* fornece um outro componente, chamado `ReceptorDeTrapsSnmp`. Este componente recebe os *traps* SNMP gerados pelos agentes da rede gerenciada, sendo suficiente para tanto, instanciá-lo. Nenhuma configuração precisa ser feita.

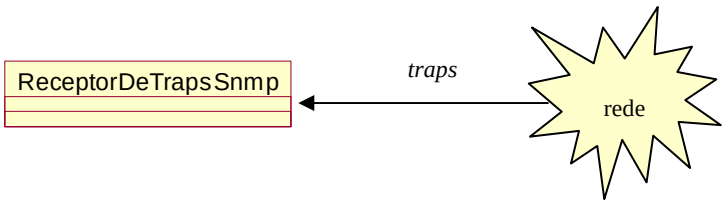


Figura 4.11. `ReceptorDeTrapsSnmp`

Até agora, portanto, há duas fontes de eventos, responsáveis pela monitoração da rede: `Monitor` e `ReceptorDeTrapsSnmp`. Os eventos gerados pelo `Monitor` contêm a informação de gerência buscada nos elementos gerenciados da rede de acordo com o período de monitoração determinado. Os eventos gerados pelo `ReceptorDeTrapsSnmp` contêm os *traps* gerados na rede de forma assíncrona. Os eventos gerados são utilizados para identificação de falhas, como será mostrado a seguir.

4.3.3. Identificando Falhas na Rede

Uma vez que o `Monitor` e o `ReceptorDeTrapsSnmp` tenham colhido a informação de gerência de interesse presente nas entidades gerenciadas da rede, a aplicação pode finalmente começar sua tarefa de gerência de falhas propriamente dita, ou seja, ela passa a analisar a informação colhida a fim de identificar a ocorrência de falhas na rede. Isto pode ser feito através de dois componentes: `GeradorDeEventoFalhaLimiar` e `GeradorDeEventoFalhaTrap`.

O `GeradorDeEventoFalhaLimiar` gera eventos para indicar a ocorrência de falhas ao comparar a informação colhida pelo `Monitor` com limiares pré-configurados. O `GeradorDeEventoFalhaTrap` gera eventos ao avaliar o tipo dos *traps* recebidos do `ReceptorDeTrapsSnmp`. Então, `Monitor` e `ReceptorDeTrapsSnmp` repassam a informação colhida na rede para o `GeradorDeEventoFalhaLimiar` e o `GeradorDeEventoFalhaTrap`, respectivamente, e estes dois últimos geram **os eventos** (`EventoFalhaEvent`) **que descrevem as falhas ocorridas na rede** ao analisar a informação recebida. A figura 4.12 descreve esta situação. De acordo com a figura, um `GeradorDeEventoFalhaLimiar` pode estar associado não só a um, mas a vários monitores, de maneira que possa receber informação de diferentes entidades gerenciadas. O `GeradorDeEventoFalhaTrap`, por sua vez, recebe os *traps* gerados por qualquer entidade gerenciada da rede através do mesmo `ReceptorDeTrapsSnmp`.

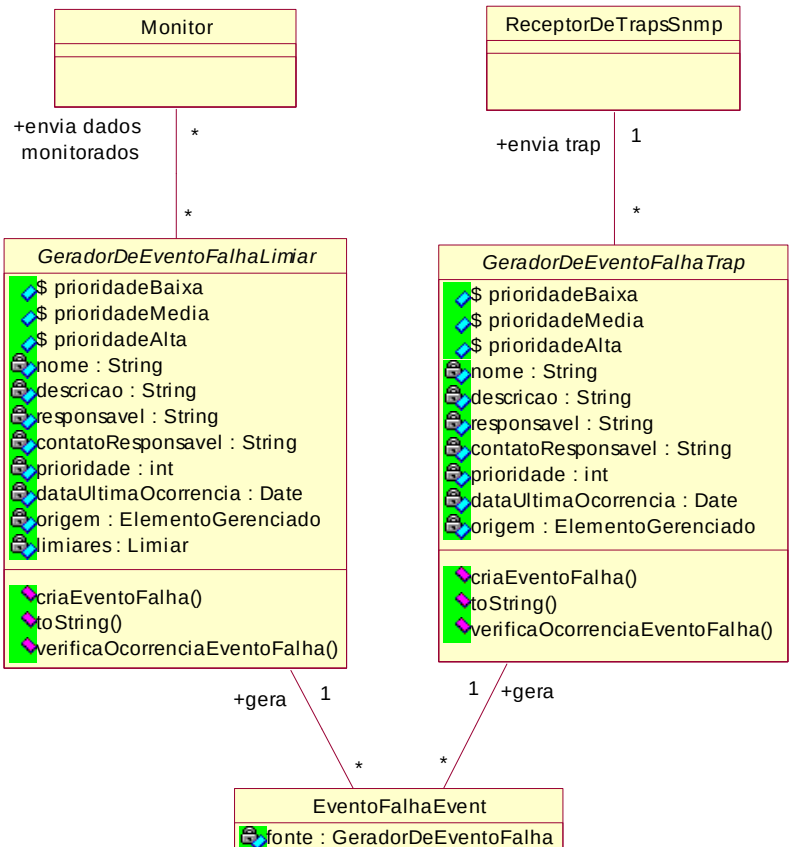


Figura 4.12. `GeradorDeEventoFalhaLimiar` e `GeradorDeEventoFalhaTrap`

Os geradores de eventos, `GeradorDeEventoFalhaLimiar` e `GeradorDeEventoFalhaTrap`, possuem várias propriedades que podem ser configuradas pelo programador e que servirão para caracterizar os eventos a serem gerados. São elas:

`nome` - um nome que identifica o tipo de falha descrita pelo evento gerado, por exemplo, `AltaTaxaDeErros`. Utilizado por um componente que esteja associado a vários destes geradores e deseje identificar qual o evento recebido num determinado instante.

`descricao` - uma descrição complementar sobre o tipo de falha descrito pelo evento gerado, por exemplo, "Alta taxa de erros devido a problemas na interface do equipamento". Pode ser útil no momento da geração de *logs* e até mesmo para a geração de alarmes, constituindo parte do conteúdo da notificação a ser enviada ao operador da rede.

`origem` - a entidade da rede onde a falha ocorreu.

`prioridade` - indica a gravidade da falha descrita pelo evento gerado e, conseqüentemente, a urgência com que o problema deve ser resolvido. Por *default*, pode assumir os seguintes valores: `prioridadeAlta`, `prioridadeMedia` e `prioridadeBaixa`.

`responsavel` - o nome da pessoa que deve ser especialmente notificada sobre a ocorrência da falha descrita pelo evento gerado.

`contatoResponsavel` - uma informação (*e-mail*, telefone, etc.) para contato com o `responsavel`.

O `GeradorDeEventoFalhaLimiar` ainda possui a propriedade `limiares` que representa os limiares a serem utilizados para verificação da ocorrência de falhas. Enfim, cada gerador configurado identifica um determinado tipo de falha.

Para identificar um tipo de falha específico, entretanto, não basta instanciar e configurar um gerador do tipo `GeradorDeEventoFalhaLimiar` ou `GeradorDeEventoFalhaTrap`. Antes disso, o programador deve definir o método `verificaOcorrenciaEventoFalha` para analisar a informação recebida do `Monitor`, no caso de um `GeradorDeEventoFalhaLimiar`, ou do `ReceptorDeTrapsSnmp`, no caso de um `GeradorDeEventoFalhaTrap`. Trata-se, portanto, dos dois primeiros **componentes semiprontos** fornecidos pelo *framework*.

Considere o exemplo a seguir. Suponha que seja necessário verificar se a taxa de erros de um roteador R ultrapassou determinado valor. Para isto, um novo componente pode ser construído a partir do `GeradorDeEventoFalhaLimiar`, ao definir o método `verificaOcorrenciaEventoFalha` conforme descrito pela figura 4.13 (os trechos de código apresentados a partir de agora estão escritos na linguagem Java).

```
//Criação do componente GeradorAltaTaxaDeErros.

public GeradorAltaTaxaDeErros extends GeradorDeEventoFalhaLimiar
                                          implements Serializable[10]{
  public boolean verificaOcorrenciaEventoFalha(ElementoGerenciado egs[]){
    //Retorna true se a TaxaDeErros de R (egs[0]) for maior que o
    //valor do limiar cujo nome é TaxaDeErros.
    return getInfoGerenciaDaOrigem("TaxaDeErros",  egs[0]).getDouble()
           > Double.parseDouble(getLimiares("TaxaDeErros").getValor())
  }
}
```

Figura 4.13. `GeradorAltaTaxaDeErros`

[10] Todo componente em Java (também chamado *JavaBean*) deve implementar a interface `java.io.Serializable`. Para maiores detalhes, consulte o apêndice B.

Construído o novo componente, pode-se instanciá-lo, configurá-lo e associá-lo ao `Monitor` que fornece a informação de gerência a ser utilizada para verificar a ocorrência da falha, isto é, o valor da taxa de erros de R. Toda vez que o `Monitor` fornecer tal informação, o `GeradorAltaTaxaDeErros` verifica se a falha em questão ocorreu.

Analogamente, podemos criar um outro componente, `GeradorLinkDown`, a partir do `GeradorDeEventoFalhaTrap`, para verificar se uma das interfaces de R está inoperante. Definimos o método `verificaOcorrenciaEventoFalha` como a seguir.

```
//Criação do componente GeradorLinkDown.
public GeradorLinkDown extends GeradorDeEventoFalhaTrap implements
                                                        Serializable{
  public boolean verificaOcorrenciaEventoFalha(TrapEvent trap){
       //Retorna true se o trap recebido é do tipo linkDown. O framework
       //verifica automaticamente se a origem do trap recebido é igual a
       //a R (propriedade origem do GeradorLinkDown).
            return trap.getTipoDeTrap() == trap.linkDown;
  }
}
```

Figura 4.14. `GeradorLinkDown`

Construído o novo componente, pode-se instanciá-lo, configurá-lo e associá-lo ao `ReceptorDeTrapsSnmp`, de maneira que o novo componente possa receber os *traps* gerados na rede e, então, verificar se o *trap* recebido foi gerado por R e se tem o tipo (`linkDown`) especificado.

É através do método `verificaOcorrenciaEventoFalha`, portanto, que o programador da aplicação descreve situações de falha na rede, em conformidade com o requisito **F2**. É também neste momento que ele precisa realmente programar, ou seja, escrever trechos de código, de modo a construir novos componentes para identificação de falhas conforme as necessidades da aplicação. O *framework* especifica os **parâmetros de entrada** e os **métodos** que podem ser utilizados para que o método `verificaOcorrenciaEvento Falha` possa ser definido.

Parâmetros de entrada

O programador tem acesso à informação de gerência enviada pelo `Monitor` ou pelo `ReceptorDeTrapsSnmp` através dos parâmetros de entrada do método `verificaOcorrenciaEventoFalha`.

No caso de um `GeradorDeEventoFalhaLimiar`, o método `verificaOcorrenciaEventoFalha(ElementoGerenciado egs[])` recebe a coleção de entidades gerenciadas (`egs`) de onde o `GeradorDeEventoFalhaLimiar` obtém informação, isto é, a coleção de entidades gerenciadas monitoradas pelos monitores com os quais o `GeradorDeEventoFalhaLimiar` está associado. A partir daí, o programador pode ter acesso à informação contida no `GrupoInfoGerencia` fornecido por cada uma delas. Para a situação em que o mesmo `GeradorDeEventoFalhaLimiar` esteja associado a vários monitores, portanto, o programador pode ter acesso à informação contida no `GrupoInfoGerencia` fornecido por qualquer uma das entidades monitoradas. Retorna "verdadeiro" (`true`) se a condição analisada for satisfeita, isto é, se a falha verificada ocorreu.

No caso de um `GeradorDeEventoFalhaTrap`, o método `verificaOcorrenciaEventoFalha(TrapEvent trap)` recebe o *trap* (`trap`) enviado pelo `ReceptorDeTrapsSnmp`. O programador pode, então, ter acesso à informação contida no `trap` para verificar a ocorrência de um evento, retornando "verdadeiro" (`true`) no caso em que a falha verificada tenha realmente ocorrido. Um `TrapEvent` possui um `tipo`, uma `origem`, uma data que indica o instante em que o `trap` ocorreu (`timeStamp`) e a informação de gerência (`gig`) enviada pela própria entidade geradora do `trap` (a `origem`). A figura 4.15 apresenta um `TrapEvent` com todas as suas propriedades e métodos de obtenção dos valores citados.

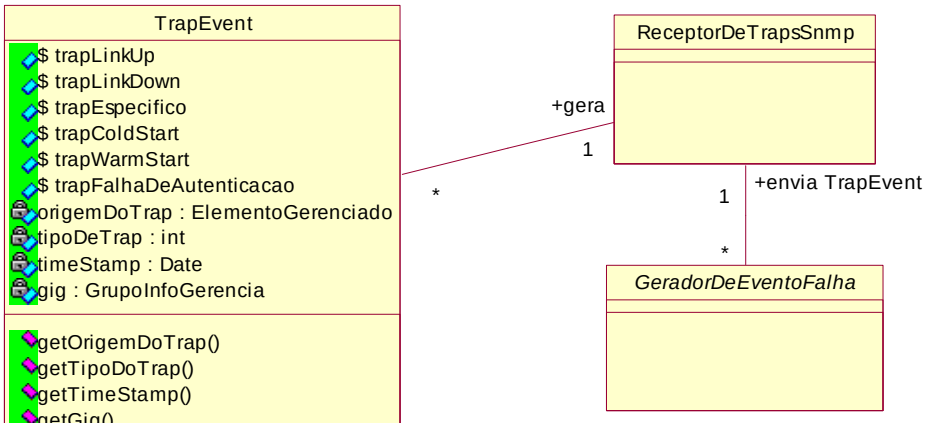


Figura 4.15. `TrapEvent`

Assim, para avaliar a ocorrência de falhas, o programador pode utilizar a informação (`GrupoInfoGerencia`) fornecida pelas entidades monitoradas da rede (`ElementoGerenciadoSnmp`) ou os *traps* (`TrapEvent`) enviados pelo `ReceptorDeTrapsSnmp`. Vejamos como é possível ter acesso a esta informação.

Métodos

Alguns métodos podem ser utilizados para manipular a informação de gerência disponível. No caso de um `GeradorDeEventoFalhaLimiar`, os métodos são os seguintes:

1. `GeradorDeEventoFalhaLimiar.getInfoGerenciaDaOrigem (String nomeIg, String nomeEg)` - retorna uma unidade de informação de gerência (`InfoGerencia`) contida no `GrupoInfoGerencia` fornecido por uma determinada entidade gerenciada (`ElementoGerenciado`). A unidade a ser retornada tem seu `nome` igual a `nomeIg` e a entidade gerenciada que fornece esta informação tem seu `nome` igual a `nomeEg`. É possível ter acesso a qualquer unidade de informação de um `GrupoInfoGerencia` fornecido por qualquer entidade gerenciada que faz parte de `egs`.
2. `InfoGerencia.getInteiro()`, `InfoGerencia.getDouble()` e `InfoGerencia.getString()` - retorna um valor `inteiro`, um valor `double` e um valor `string`, respectivamente, conforme o valor da unidade de informação de gerência.
3. `GeradorDeEventoFalhaLimiar.getLimiares(String nome)` - retorna o limiar que tem o nome especificado (`nome`). Um limiar é um outro componente do tipo `Limiar` que possui um `nome` e um `valor`.

4. `Limiar.getValor()` - retorna o valor do limiar.

No caso de um `GeradorDeEventoFalhaTrap`, os métodos utilizados seguem abaixo:

1. `TrapEvent.getTipoDoTrap()` - retorna o tipo do trap recebido. Pode ser um entre os seis seguintes valores:
   - `trapLinkUp` – indica que o estado de operação da interface de rede mudou de `down` para `up`, isto é, indica que a interface está em seu estado de operação normal.
   - `trapLinkDown` - indica que o estado de operação da interface de rede mudou de `up` para `down`, isto é, indica que a interface está inoperante.
   - `trapColdStart` - indica que o agente está se (re)inicializando e pode ser que a informação de gerência disponível seja alterada.
   - `trapWarmStart` - indica que o agente está se reinicializando mas a informação de gerência disponível não será alterada.
   - `trapFalhaDeAutenticacao` – indica que houve falha de autenticação de uma requisição de gerência, ou seja, uma requisição recebida não continha a informação de autenticação correta para que fosse devidamente atendida.
   - `trapEspecifico` – indica outro evento de interesse; pode conter, por exemplo, informação associada a uma tecnologia particular.
2. `TrapEvent.getOrigemDoTrap()` - retorna a entidade gerenciada (`ElementoGerenciado`) que gerou o *trap*.
3. `TrapEvent.getTimeStamp()` - retorna o instante de tempo em que o *trap* foi gerado.
4. `TrapEvent.getGig()` - retorna a informação de gerência (`Grupo InfoGerencia`) contida no *trap* recebido.

Observe, portanto, a flexibilidade que o *framework* fornece para que o programador possa definir suas condições de ocorrência de falhas, especialmente no caso de um

GeradorDeEventoFalhaLimiar. Primeiramente, o programador tem disponível todo o GrupoInfoGerencia de cada ElementoGerenciado monitorado pelos monitores com os quais o GeradorDeEventoFalhaLimiar está associado. Isto permite definir condições onde várias unidades de informação de gerência podem ser utilizadas, conforme mostra a figura 4.16.

```
//Criação do componente GeradorAltaPerdaDePacotes.
public GeradorAltaPerdaDePacotes extends GeradorDeEventoFalhaLimiar implements
                                                                    Serializable{
   public boolean verificaOcorrenciaEventoFalha(ElementoGerenciado egs[]){
        //Retorna true se a quantidade total de pacotes descartados por
        //egs[1] for maior que o valor do limiar TaxaDeErros.
     return getInfoGerenciaDaOrigem
        ("PacotesDescartadosIn",egs[0]).getDouble() +
        getInfoGerenciaDaOrigem
        ("PacotesDescartadosOut", egs[1]).getDouble()>
        Double.parseDouble(getLimiares ("TaxaDeErros").getValor())
   }
}
```

Figura 4.16. GeradorAltaPerdaDePacotes

Depois, se o GeradorDeEventoFalhaLimiar estiver realmente associado a vários monitores, o programador pode definir condições de ocorrência de falhas ao analisar a informação oriunda de diferentes entidades da rede. Por exemplo:

```
//Criação do componente GeradorAltaTaxaDeTrafego1.
public GeradorAltaTaxaDeTráfego1 extends GeradorDeEventoFalhaLimiar
                                              implements Serializable{
   public boolean verificaOcorrenciaEventoFalha(ElementoGerenciado egs[]){
        //Retorna true se a soma da quantidade de conexões TCP abertas em
        //egs[0] e egs[1] for maior que o limiar ConexoesAbertas.
     return (getInfoGerenciaDaOrigem("QuantConexoesAbertas",
         egs[0]).getInteiro() +
         getInfoGerenciaDaOrigem("QuantConexoesAbertas",
         egs[1]).getInteiro() >
         Double.parseDouble(getLimiares ("ConexoesAbertas").getValor())
   }
}
```

Figura 4.17. `GeradorAltaTaxaDeTrafego1`

E, finalmente, o programador pode configurar quantos limiares sejam necessários para o `GeradorDeEventoFalhaLimiar` em questão, sendo possível analisar várias unidades de informação de gerência em torno de um mesmo limiar (exemplos acima) ou analisar cada unidade de informação separadamente, utilizando vários limiares, como mostrado no exemplo da figura 4.18.

Desta maneira, é possível definir situações de falha semanticamente mais ricas, conforme o requisito **F3**, e assim elevar, também, a semântica dos eventos (`EventoFalhaEvent`) que podem ser gerados. Cada nova situação de falha a ser analisada pode resultar na construção de um novo componente e é assim que o programador **adiciona funcionalidade ao *framework***, determinando as falhas que devem ser identificadas. Ao ser possível definir expressões de alto nível para a análise da informação de gerência fornecida pela entidades da rede, atende-se, também, ao requisito **NF3.3**.

```
//Criação do componente GeradorAltaTaxaDeTrafego2.

public GeradorAltaTaxaDeTráfego2 extends GeradorDeEventoFalhaLimiar
                                              implements Serializable{
  public boolean verificaOcorrenciaEventoFalha(ElementoGerenciado egs[]){
    //Retorna true se a quantidade de conexões TCP abertas em egs[0]
    //for maior que o limiar ConexoesAbertas e a quantidade de pacotes
    //que chegam também a egs[0] for maior que o limiar TaxaDeTrafego.
    return(getInfoGerenciaDaOrigem("QuantConexoesAbertas",
         egs[0]).getInteiro() >
         Int.parseInt(getLimiares("ConexoesAbertas").getValor()) &&
         getInfoGerenciaDaOrigem("PacotesIn", egs[0]).getDouble() >
         Double.parseDouble(getLimiares ("TaxaDeTrafego").getValor())
  }
}
```

Figura 4.18. `GeradorAltaTaxaDeTrafego2`

Embora o programador possa se sentir à vontade para escrever qualquer condição de ocorrência de falha a ser verificada, muitas vezes esta condição se baseia num mecanismo de **histerese**. De acordo com este mecanismo, dois valores são utilizados: um limiar, cujo cruzamento indicará a ocorrência de uma falha, e um valor de reposicionamento (*rearm*), cujo cruzamento indicará a situação contrária, isto é, uma situação livre de falhas. A geração de eventos, ou identificação de falhas, baseia-se na análise de determinada informação de gerência em torno deste dois valores. O primeiro evento é gerado apenas quando o limiar é atingido pela primeira vez. Outro evento só é gerado quando o evento gerado anteriormente resultou do cruzamento do valor utilizado na direção oposta.

A figura 4.19 mostra este mecanismo em ação. O primeiro evento (e_1) resultou do cruzamento do limiar l; o segundo evento (e_2) resultou do cruzamento do valor de reposicionamento r; o terceiro evento (e_3) resultou novamente do cruzamento de l quando o valor de reposicionamento foi atingido pelo menos uma vez; e assim sucessivamente. e_1, e_2 e e_3 representam os eventos gerados. Nas situações A e B, são observados um limiar máximo e um limiar mínimo, respectivamente.

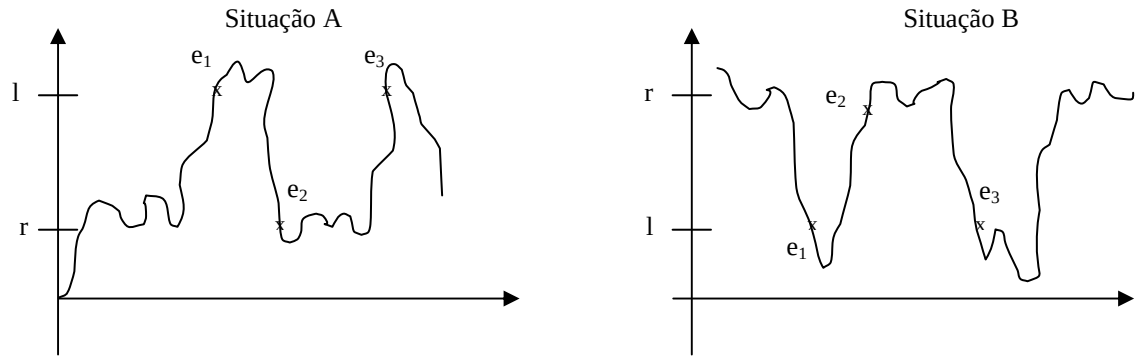


Figura 4.19. Mecanismo de histerese

Sendo assim, o *framework* fornece um componente pronto, muito útil, que implementa um mecanismo de histerese para identificação de falhas. Este componente se chama `GeradorDeEventoFalhaComHisterese` e pode ser visto na figura 4.20. Ele gera dois tipos de eventos diferentes, de acordo com o valor atingido.

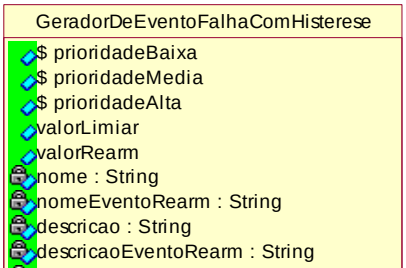


Figura 4.20. `GeradorDeEventoFalhaComHisterese`

Para utilizar um `GeradorDeEventoFalhaComHisterese`, deve-se configurar os valores do `limiar` e do `rearm`. Além disso, é preciso configurar o evento que será gerado no caso do `limiar` ser atingido (`nome`, `descricao`, `prioridade`), bem como o evento que será gerado no caso do valor de `rearm` ser atingido (`nomeEventoRearm`, `descricaoEventoRearm` e `prioridadeEventoRearm`). Finalmente, é preciso associar o `GeradorDeEventoFalhaComHisterese` ao `Monitor` que obtém a informação da gerência a ser analisada. O método `getValor Atingido` também está disponível, permitindo saber qual dos valores foi atingido (`limiar` ou `rearm`), ou seja, qual dos valores levou à geração do novo evento.

O *framework*, portanto, identifica falhas na rede através de componentes criados pelo próprio programador e através da utilização de componentes do tipo `GeradorDeEvento FalhaComHisterese`. Resta tratar as falhas identificadas a fim de fazer com que a rede volte ao seu estado de funcionamento normal o mais rápido possível. O *framework* fornece alguns componentes para tratamento de falhas mas também permite que o programador defina seus próprios componentes, como veremos a seguir.

4.3.4. Tratando as Falhas Identificadas

Conforme visto na seção anterior, geradores de eventos identificam falhas na rede ao gerarem eventos do tipo `EventoFalhaEvent`. Os eventos gerados descrevem as falhas ocorridas. Para que estas falhas possam ser tratadas, componentes devem se associar aos geradores de eventos adequados a fim de obter a informação necessária. Em outras palavras, componentes responsáveis pela geração de alarmes, atualização de registros de ocorrência

(*logs*) ou correlação de eventos devem se associar aos geradores de eventos previamente configurados, para que uma vez ocorrido determinado tipo de evento, eles possam tomar conhecimento do fato e realizar as ações cabíveis automaticamente. Assim sendo, é possível automatizar tarefas em resposta aos eventos ocorridos conforme requisito **F5**.

Na figura 4.21, um componente responsável por fazer a correlação de eventos está associado a um gerador de eventos para que possa fazer a correlação com base na informação recebida. Qualquer componente responsável pelo tratamento de falhas pode se associar a qualquer tipo de gerador de eventos (`GeradorDeEventoFalhaLimiar`, `GeradorDeEventoFalhaTrap` ou `GeradorDeEventoFalhaComHisterese`) a fim de obter a informação de interesse.

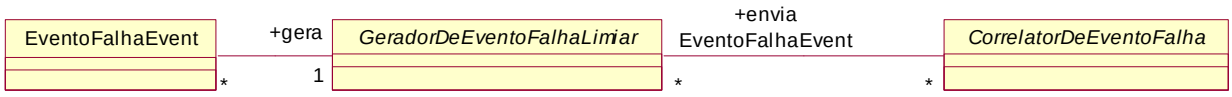


Figura 4.21. `CorrelatorDeEventoFalha`: um componente para tratamento de falhas

O *framework* fornece três componentes que podem ser diretamente utilizados para fazer **correlação de eventos**, em atendimento ao requisito **F4**: `CorrelatorDeEventoFalhaCompressao`, `CorrelatorDeEventoFalhaSupressao` e `CorrelatorDeEventoFalhaCMP`. Estes componentes são apresentados na figura 4.22. Eles implementam os algoritmos de correlação conhecidos, respectivamente, como compressão, supressão e controle de manutenção programada (CMP).

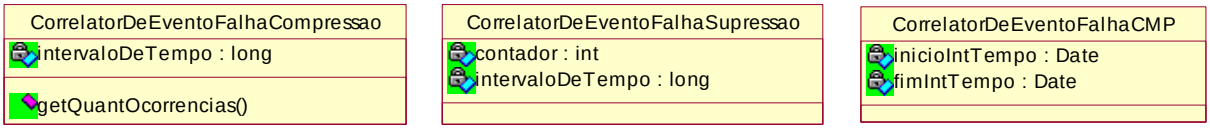


Figura 4.22. Componentes para a correlação de eventos

O algoritmo de compressão, conforme descrito na seção 2.3.1, consiste em detectar múltiplas ocorrências de um mesmo evento em um dado intervalo de tempo, substituindo os eventos ocorridos por um único evento, possivelmente indicando quantas vezes o evento ocorreu durante o período de observação. Para utilizar o componente `CorrelatorDeEventoFalhaCompressao` basta configurar este período de observação (`intervaloDeTempo`) e associá-lo ao gerador que verifica a ocorrência do evento a ser considerado. Na

figura abaixo, por exemplo, o `CorrelatorDeEventoFalhaCompressao` comprime os eventos (e_1, ..., e_n) recebidos de `GeradorLinkDown` durante 30 minutos (`intervaloDeTempo` = 30min), gerando um único evento (e_σ) ao final deste intervalo de tempo.

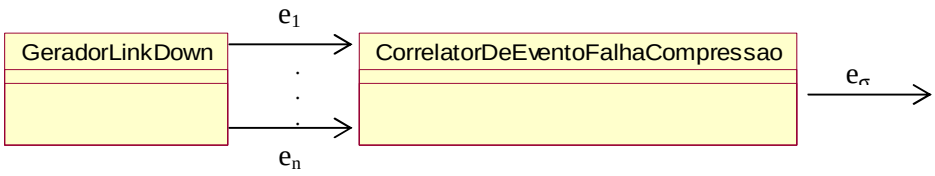


Figura 4.23. `CorrelatorDeEventoFalhaCompressao`

O algoritmo de supressão, também descrito na seção 2.3.1, consiste em detectar se um mesmo evento ocorreu determinado número de vezes (k) dentro de um intervalo de tempo, substituindo os k eventos ocorridos por um único evento tão logo k seja atingido. Sendo assim, para utilizar o `CorrelatorDeEventoFalhaSupressao`, é necessário configurar não só o período de observação (`intervaloDeTempo`), mas também o valor de k (`contador`), ou seja, o número máximo de vezes que o evento pode ocorrer antes que um novo evento seja gerado. Além disso, é preciso associá-lo ao gerador de eventos que verifica a ocorrência do evento a ser correlacionado. Na figura 4.24, por exemplo, o `CorrelatorDeEventoFalhaSupressao` possui um `contador` igual a 3 unidades e um `intervaloDeTempo` igual a 30 minutos. Ele gerará um novo evento (e_σ) tão logo receba três eventos (e_1, e_2, e_3) de `GeradorAltaTaxaDeErros` neste intervalo de tempo.

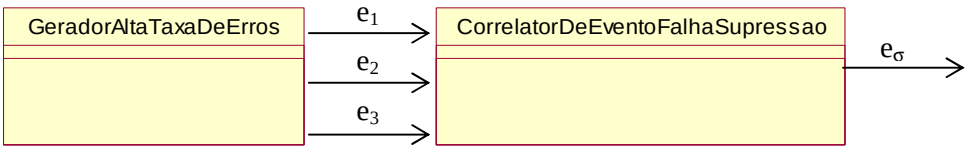


Figura 4.24. `CorrelatorDeEventoFalhaSupressao`

O algoritmo CMP, por sua vez, consiste em evitar a repetição de eventos decorrentes de atividades de manutenção realizadas em equipamentos da rede. De fato, equipamentos da rede que precisam de algum tipo de manutenção podem ficar temporariamente inacessíveis (serão desligados ou simplesmente desconectados da rede) e isto pode levar à ocorrência de múltiplos eventos do tipo "`EquipamentoForaDoAr`", o que pode não ser interessante. Deste modo, componentes do tipo `CorrelatorDeEventoFalhaCMP` podem ser

utilizados para que esta repetição de eventos seja evitada. Configura-se, então, a data em que as atividades de manutenção de um equipamento `E` terão início (`inicioIntTempo`) e a data prevista para o término de tais atividades (`fimIntTempo`). Além disso, associa-se o `CorrelatorDeEventoFalhaCMP` ao gerador que gera eventos do tipo "`EquipamentoForaDoAr`" e cuja origem do evento gerado seja igual a `E`. O componente gerará um único evento durante este intervalo de tempo, suprimindo os demais.

Confira na figura abaixo. e_1 foi recebido após `inicioIntTempo` e, conseqüentemente, foi repassado. Os demais eventos, e_2... e_n, foram recebidos após e_1 e antes de `fimIntTempo`, sendo desconsiderados.

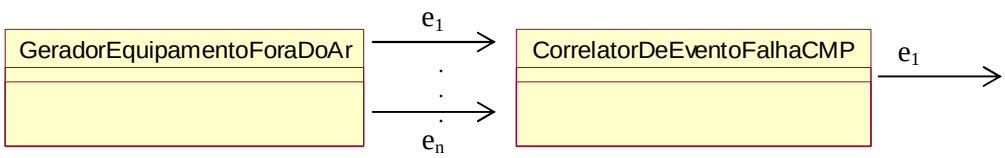


Figura 4.25. `CorrelatorDeEventoFalhaCMP`

O algoritmo implementado pelo componente `CorrelatorDeEventoFalhaCMP`, portanto, é uma variação do algoritmo de compressão e consiste em gerar um novo evento tão logo a primeira ocorrência de determinado evento seja detectada dentro do intervalo de tempo considerado. As demais ocorrências deste mesmo evento durante este período de observação são simplesmente ignoradas. Ele pode ser utilizado em diversas situações mas por ser especialmente útil para o caso em que citamos, este algoritmo é conhecido como correlação de controle de manutenção programada, ou simplesmente, CMP.

Do exposto acima, observe que componentes que fazem correlação de eventos são também geradores de eventos. Eles recebem os eventos gerados por geradores do tipo `GeradorDeEventoFalhaLimiar`, `GeradorDeEventoFalhaTrap` e `GeradorDeEventoFalhaComHisterese`, mas também geram seus próprios eventos como resultado da correlação realizada, identificando novos tipos de falhas - as figuras 4.23, 4.24 e 4.25 exemplificam este fato. Desta maneira, qualquer outro componente responsável por tratar as falhas identificadas na rede (um componente responsável por gerar alarmes, por exemplo) que possa ser associado a um `GeradorDeEventoFalhaLimiar`, `GeradorDeEventoFalhaTrap` ou `GeradorDeEventoFalhaComHisterese`, pode ser também associado a qualquer um dos componentes que fazem correlação de eventos. Na verdade, um

componente deste tipo é também um gerador de eventos que gera eventos do tipo `EventoFalhaEvent` e possui além das propriedades associadas ao algoritmo de correlação implementado, as propriedades que caracterizam o evento a ser gerado conforme descrito pela figura 4.26.

Figura 4.26. `CorrelatorDeEventoFalha`

Para utilizar outros tipos de correlação, o programador pode construir novos componentes a partir do `CorrelatorDeEventoFalha`, introduzindo seus próprios algoritmos de correlação através do método `correlaciona`. Então, assim como novos geradores de eventos podem ser construídos ao definir o método `verificaOcorrencia EventoFalha`, novos componentes para correlação de eventos podem ser construídos ao definir o método `correlaciona`. Este método recebe como parâmetro de entrada os eventos enviados pelo(s) gerador(es) de eventos com o(s) qual(is) o `CorrelatorDeEventoFalha` está associado e retorna "verdadeiro" (`true`) se o princípio de correlação utilizado for satisfeito. A figura abaixo exemplifica a criação de um novo componente para correlação de eventos.

```
//Criação de um novo correlator de eventos.
public CorrelatorTemporal extends CorrelatorDeEventoFalha implements
                                                    Serializable{
     public void correlaciona (EventoFalhaEvent ef){
          //Inclui algoritmo de correlação aqui...
```

```
    }
}
```

Figura 4.27. `CorrelatorTemporal`

Para tratar as falhas identificadas, o programador ainda pode construir novos componentes ao implementar uma interface fornecida pelo *framework*, chamada `EventoFalhaListener`[11]. Esta interface possui um único método, `processaEventoFalha`, através do qual o programador pode determinar o tratamento adequado para as falhas identificadas conforme suas necessidades. A figura 4.28, exemplifica a utilização da interface `EventoFalhaListener`.

```
//Criação de um gerador de alarmes.
public GeradorDeAlarmes implements EventoFalhaListener, Serializable{
     public void processaEventoFalha (EventoFalhaEvent ef){
          //Inclui tratamento de falhas aqui...
     }
}
```

Figura 4.28. `GeradorDeAlarmes`

Qualquer novo componente que implemente esta interface pode ser conectado a qualquer tipo de gerador de eventos (`GeradorDeEventoFalhaLimiar`, `GeradorDeEventoFalhaTrap` e `GeradorDeEventoFalhaComHisterese`) ou a um `CorrelatorDeEventoFalha`, a fim de que possa ser informado sobre a ocorrência das falhas a serem tratadas. Um componente responsável pela geração de alarmes, conforme descrito na figura 4.28, ou um componente responsável pelo registro de ocorrências são exemplos de componentes que poderiam implementá-la.

[11]Lembre-se que, além de componentes semiprontos, o *framework* também especifica interfaces e classes que podem ser utilizadas para construir novos componentes.

4.3.5. Fornecendo o Suporte a Vários Modelos de Gerência

Até agora, vimos como construir novos componentes a partir de componentes semiprontos, escrevendo métodos que o próprio *framework* determina. Para construir um novo gerador de eventos baseado em limiares (`GeradorDeEventoFalhaLimiar`), por exemplo, basta escrever o método `verificaOcorrenciaEventoFalha`; para construir um novo componente para correlação de eventos, basta escrever o método `correlaciona`.

Entretanto, também é possível utilizar interfaces adicionais fornecidas pelo *framework* conforme vimos na seção 4.2 e conforme já exemplificado na seção anterior quando a interface `EventoFalhaListener` foi apresentada. Nesta seção, vejamos como utilizar outras interfaces para fornecer o suporte a diferentes modelos de gerência.

Para construir uma determinada aplicação, é preciso, em algum momento, tratar características específicas de um ou de vários modelos de gerência. Por exemplo, no modelo Internet, deve-se especificar um endereço IP para cada entidade gerenciada (agente SNMP envolvido) e um identificador específico para cada unidade de informação de gerência (variável MIB) que deve ser obtida pela aplicação. Além disso, é preciso um `ReceptorDeTraps` que reconheça cada *trap* SNMP recebido e o mapeie para um modelo comum especificado pelo *framework* (*traps* do tipo `TrapEvent`). Cada modelo de gerência tem suas características próprias.

O *framework* fornece os componentes `ElementoGerenciadoSnmp`, `InfoGerenciaSnmp` e `ReceptorDeTrapsSnmp`, fornecendo suporte *default* ao SNMP conforme requisito **NF2**, mas também permite, a partir das interfaces `ElementoGerenciado` e `InfoGerencia` e do componente semipronto `ReceptorDeTraps`, que novos componentes possam ser construídos para descrever características particulares de outros modelos de gerência. A figura 4.29 mostra as interfaces e o componente semipronto citados.

Figura 4.29. `ElementoGerenciado`, `InfoGerencia` e `ReceptorDeTraps`

A interface `ElementoGerenciado` descreve qualquer tipo de entidade gerenciada, não possuindo características que estejam diretamente associadas a um modelo de gerência específico. É através do método `getGrupoInfoGerencia` que a informação de gerência requisitada é fornecida e é, aqui, portanto, que os detalhes do protocolo de gerência devem ser embutidos. O componente `ElementoGerenciadoSnmp`, por exemplo, implementa esta interface, acrescentando ao componente criado o endereço IP através do qual é possível localizar o agente SNMP representado, conforme visto na seção 4.3.1.

A interface `InfoGerencia` descreve uma unidade de informação de gerência genérica. Um `GrupoInfoGerencia` agrupa componentes deste tipo e não componentes `InfoGerenciaSnmp`, conforme dito anteriormente, de modo que um mesmo `GrupoInfoGerencia` pode conter diferentes unidades de informação de gerência referentes a diferentes modelos de gerência simultaneamente. O componente `InfoGerenciaSnmp` implementa a interface `InfoGerencia` e possui um identificador adicional (`oid`) que se refere ao identificador específico de cada variável MIB (`ifInErrors`, por exemplo).

Um `ReceptorDeTraps` possui dois métodos que devem ser redefinidos para tratar características específicas do modelo de gerência considerado. O método `getTrapDaRede` é responsável por tratar cada *trap* recebido e a partir daí gerar *traps* do tipo `TrapEvent`; o método `determinaTipoDeTrap` mapeia os tipos de *traps* conforme definidos pelo modelo de gerência considerado para um dos tipos de *traps* reconhecidos pelo *framework* (`TrapEvent.linkDown`, `TrapEvent.linkUp`, `TrapEvent.trapEspecifico` e outros). Então, um `ReceptorDeTrapsSnmp`, por exemplo, redefine estes dois métodos para tratar os *traps* SNMP recebidos e a partir da informação neles contida, gerar os eventos

que outros componentes do *framework* são capazes de reconhecer, isto é, *traps* do tipo `TrapEvent`.

Sendo assim, ao construir uma aplicação de gerência baseada em SNMP, instanciamos e configuramos os componentes `ElementoGerenciadoSnmp`, `InfoGerenciaSnmp` e `ReceptorDeTrapsSnmp`. Para tratar características de outros protocolos, outros componentes podem ser construídos e, uma vez obedecidas as regras estabelecidas pelas interfaces e componente apresentados na figura 4.29, mantém-se o *framework* independente de protocolo de gerência, em atendimento ao requisito **NF1**. A interoperação entre os diferentes modelos de gerência é garantida, sendo possível construir uma aplicação capaz de gerenciar, simultaneamente, diferentes entidades da rede, independentemente do modelo de gerência que elas suportam.

4.3.6. Modificando os Eventos Produzidos pelos Componentes do *Framework*

O *framework* ainda permite modificar os **eventos** trocados entre alguns componentes instanciados. Por exemplo, é possível modificar os eventos (`EventoFalhaEvent`) que podem ser gerados pelos geradores de eventos `GeradorDeEventoFalhaLimiar`, `GeradorDeEventoFalhaTrap` e `GeradorDeEventoFalhaComHisterese` e pelo `CorrelatorDeEventoFalha`. Para isto, pode-se acrescentar novas propriedades e métodos à classe `EventoFalhaEvent`, incluindo a informação adicional necessária e, assim, definindo uma nova classe para descrição dos eventos gerados por estes componentes. Deve-se, conseqüentemente, modificar o método `criaEventoFalha` do gerador de eventos correspondente para fazer com que o gerador passe a gerar o novo tipo de evento.

A figura 4.30 mostra um exemplo em que isto pode ser útil. Suponha que seja necessário gerar eventos que possuam outros níveis de prioridade além daqueles níveis suportados por *default* (`prioridadeAlta`, `prioridadeMedia` e `prioridade Baixa`). É possível definir e gerar este novo tipo de evento conforme descrito pela figura abaixo.

```
//Criação de um novo tipo de evento.
public EventoFalhaNovo extends EventoFalhaEvent{
     //Níveis de prioridade adicionais.
     public static final prioridadeMuitoAlta = 400;
     public static final prioridadeMuitoBaixa = 50;
```

```
     //Construtor.
     public EventoFalhaNovo(){super();}
}

//Criação do gerador que gera eventos do tipo EventoFalhaNovo.
public GeradorEventoFalhaNovo extends GeradorDeEventoFalhaLimiar
                                                  implements Serializable{
     //Redefine o método criaEventoFalha, de modo a gerar o novo tipo
     //de evento.
     public EventoFalhaEvent criaEventoFalha(){
         return new EventoFalhaNovo();
     }
     public boolean verificaOcorrenciaEventoFalha (ElementoGerenciado
                                                                egs[]){
          ...
     }
  }
```

Figura 4.30. Gerando novos tipos de eventos

O `GeradorEventoFalhaNovo` passa a gerar o novo tipo de evento (`EventoFalhaNovo`) conforme especificado pelo método `criaEventoFalha` e, assim, a aplicação pode manipular a informação adicional de forma conveniente.

Também é possível modificar o tipo de *trap* repassado por componentes do tipo `ReceptorDeTraps`. Para isto, deve-se estender a classe `TrapEvent` e deve-se fazer com que o `ReceptorDeTraps` repasse o novo tipo de *trap* especificado. `TrapEvent`, portanto, é uma classe que pode ser utilizada para fazer com que o *framework* possa reconhecer *traps* definidos pelo próprio programador.

4.3.7. Resumo

Conforme apresentado durante todo o capítulo 4, o *framework* fornece um conjunto básico de componentes a partir do qual aplicações de gerência de falhas podem ser desenvolvidas. De um modo geral, três etapas devem ser realizadas para construir cada aplicação:

1. construção de novos componentes para acomodar as novas necessidades identificadas durante a especificação da aplicação (esta etapa nem sempre é necessária desde que os componentes a serem utilizados já estejam disponíveis);

2. instanciação e configuração visual de componentes fornecidos pelo *framework* e de componentes criados pelo próprio programador; e
3. interconexão, também visual, dos componentes instanciados para que, do fluxo de eventos resultante, cada componente envolvido possa realizar sua tarefa apropriadamente.

Foram apresentados quais tipos de componentes podem ser construídos e quais tipos podem ser diretamente instanciados e configurados. A tabela 4.1 lista todos os componentes apresentados.

Componente	Descrição
`ElementoGerenciadoSnmp`	Utilizado para representar as entidades SNMP a serem gerenciadas. Implementa a interface `ElementoGerenciado` e fornece componentes do tipo `GrupoInfoGerencia`.
`InfoGerenciaSnmp`	Utilizado para representar variáveis MIB. Implementa a interface `InfoGerencia`.
`GrupoInfoGerencia`	Agrupa componentes do tipo `InfoGerencia` , representando grupos de informação de gerência utilizados pela aplicação.
`Monitor`	Responsável pela monitoração síncrona. Controla a freqüência com que um `ElementoGerenciado` deve fornecer determinada informação. Gera eventos que contém o resultado da monitoração realizada (`GrupoInfoGerencia`).
`ReceptorDeTrapsSnmp`	Responsável pela monitoração assíncrona. Recebe todos os *traps* SNMP gerados na rede, repassando *traps* do tipo `TrapEvent`. Construído a partir do componente `ReceptorDeTraps`.
`GeradorDeEventoFalhaLimiar`	Gera eventos (identifica falhas) do tipo `EventoFalhaEvent` com base em limiares pré-configurados. O programador deve "completar" este componente para determinar o tipo de falha a ser identificado.
`GeradorDeEventoFalhaTrap`	Gera eventos (identifica falhas) do tipo `EventoFalhaEvent` com base no tipo de *trap* recebido. O programador deve "completar" este componente para determinar o tipo de falha a ser identificado.
`GeradorDeEventoFalhaComHisterese`	Gera eventos (identifica falhas) do tipo

	`EventoFalhaEvent` com base num mecanismo de histerese.
`CorrelatorDeEventoFalha`	Faz correlação de eventos, identificando novos tipos de falhas ao gerar eventos do tipo `EventoFalhaEvent`. Implementa a interface `EventoFalhaListener`. Há três componentes que implementam diferentes tipos de correlação: `CorrelatorDeEventoFalhaCompressao`, `CorrelatorDeEventoFalhaSupressao` e `CorrelatorDeEventoFalhaCMP`.

Tabela 4.1. Os componentes do *framework*

Com base nestes tipos de componentes, segue abaixo uma lista de passos que podem ser seguidos para que se possa construir determinada aplicação.

Instanciar e configurar componentes do tipo `ElementoGerenciado`, para representar as entidades da rede (roteadores, switches, estações de trabalho, etc.) a serem gerenciadas.

Instanciar e configurar componentes do tipo `GrupoInfoGerencia`, para descrever os grupos de informação de gerência (`InfoGerencia`) a serem fornecidos por cada `ElementoGerenciado`.

Instanciar e configurar componentes do tipo `Monitor`, para controlar com que freqüência cada `ElementoGerenciado` deve fornecer informação de gerência (`GrupoInfoGerencia`)

Instanciar um componente do tipo `ReceptorDeTraps`, para receber os *traps* gerados na rede.

Instanciar e configurar geradores de eventos do tipo `GeradorDeEventoFalhaLimiar` e `GeradorDeEventoFalhaTrap`, para determinar quais tipos de falhas devem ser identificadas pelo *framework*. Neste caso, os componentes instanciados consistem dos novos componentes construídos pelo próprio programador durante a etapa 1 descrita no início desta seção.

Instanciar e configurar geradores de eventos do tipo `GeradorDeEventoFalhaComHisterese`, para identificar falhas com base no mecanismo de histerese prontamente disponível a partir da utilização deste componente.

Instanciar e configurar componentes para correlação de eventos (qualquer componente derivado de `CorrelatorDeEventoFalha`, inclusive componentes previamente criados pelo próprio programador durante a etapa 1).

Instanciar e configurar outros componentes do tipo `EventoFalhaListener`, responsáveis pelo tratamento de falhas e construídos durante a etapa 1.

Uma vez instanciados e configurados, os componentes devem ser **interconectados** de acordo com o **modelo fonte/consumidor**. Cada componente gera seus próprios eventos e determina quais tipos de componentes podem se cadastrar junto a ele para obterem a informação produzida. A saber:

Um `Monitor` gera um novo evento toda vez que novos dados são recebidos através da rede. Um `GeradorDeEventoFalhaLimiar` e um `GeradorDeEventoFalhaComHisterese` podem ser a ele interconectados para receberem esta informação, a qual será utilizada para verificar a ocorrência de falhas.

Um `ReceptorDeTraps` gera um novo evento toda vez que um *trap* SNMP for recebido. Um `GeradorDeEventoFalhaTrap` pode ser a ele interconectado de modo a verificar a ocorrência de falhas com base no tipo dos *traps* recebidos.

Geradores de eventos do tipo `GeradorDeEventoFalhaLimiar`, `GeradorDeEventoFalhaTrap` e `GeradorDeEventoFalhaComHisterese` geram um novo evento toda vez que a falha verificada por eles ocorrer. Um `CorrelatorDeEventoFalha` e componentes do tipo `EventoFalhaListener` podem ser interconectados a qualquer um dos componentes inicialmente citados a fim de correlacionar ou tratar os eventos deles recebidos.

Um `CorrelatorDeEventoFalha` gera um novo evento toda vez que a falha verificada por ele - falha esta resultante da correlação realizada - ocorrer, de modo que componentes do tipo `EventoFalhaListener` também podem ser a ele interconectados.

A figura 4.31 mostra as relações descritas. As setas não-hachuradas representam o envio de um evento para os componentes cadastrados.

De acordo com a figura 4.31, os componentes `Monitor` e `ReceptorDeTraps` são fontes de eventos e repassam a informação colhida na rede para outros componentes - o componente `Monitor` guarda uma referência para um componente do tipo `Elemento Gerenciado` ao qual pedirá um `GrupoInfoGerencia` com determinada freqüência. Os componentes `GeradorDeEventoFalhaLimiar`, `GeradorDeEventoFalhaTrap`, `GeradorDeEventoFalhaComHisterese` e `CorrelatorDeEventoFalha` são componentes-processadores que consomem os eventos recebidos e produzem novos eventos. Componentes do tipo `EventoFalhaListener` são consumidores de eventos que dão o tratamento adequado às falhas identificadas.

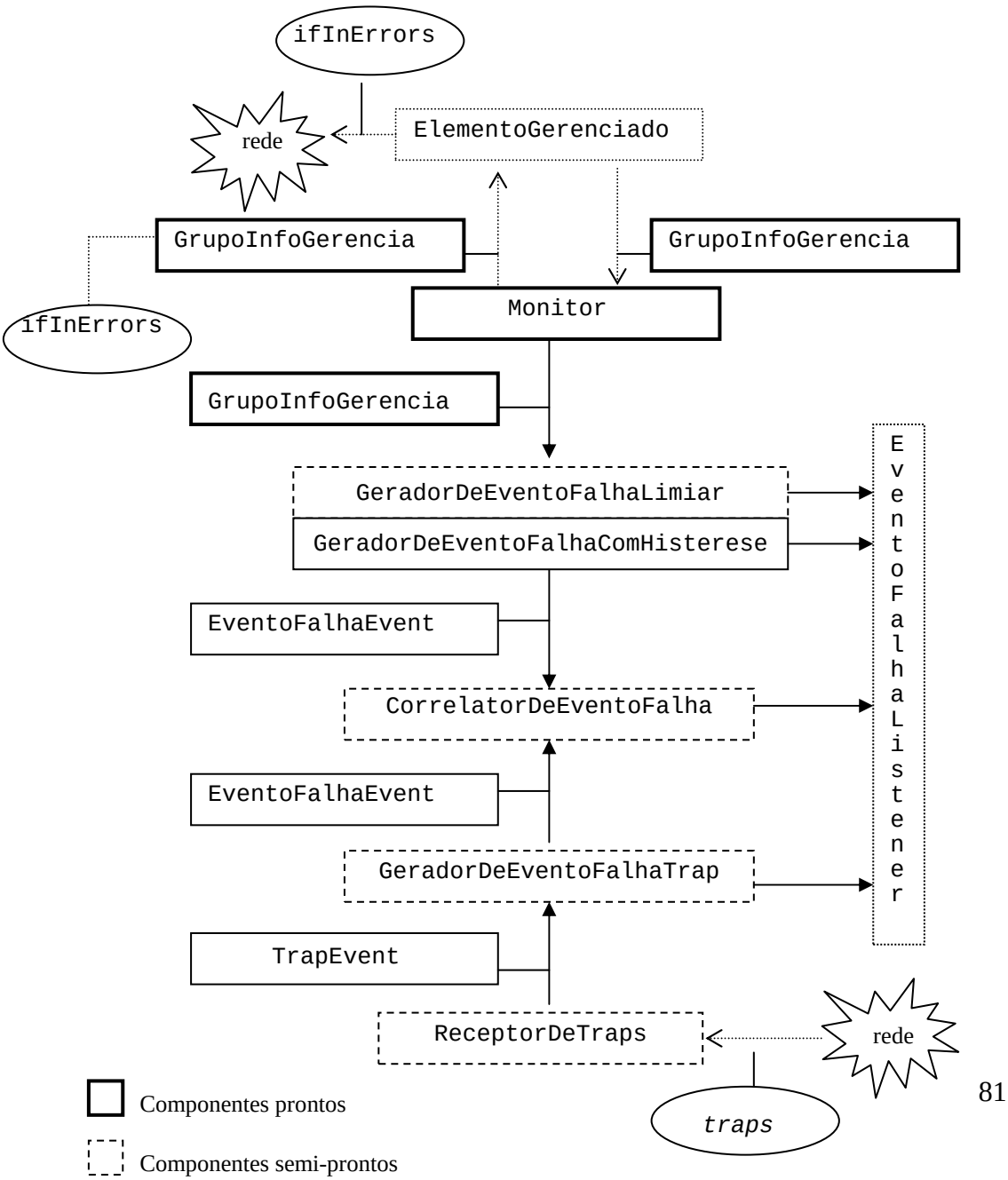


Figura 4.31. Interconexões possíveis entre os componentes do *framework*

Portanto, é assim que os componentes podem ser interconectados. Qualquer grafo pode ser estabelecido desde que sejam observados os tipos de componentes envolvidos. O programador estabelece as relações visualmente, de modo a configurar o fluxo de controle a ser executado pela sua aplicação. Finalmente, ao instanciar, configurar e interconectar os componentes conforme especificado, tem-se uma aplicação de gerência de falhas construída com base no *framework* proposto.

A tabela abaixo ainda resume como os requisitos inicialmente levantados foram atendidos pela solução proposta. Os requisitos F8 e F9 serão comentados no capítulo 6.

Requisito	**Característica da solução**
F1	Componentes `Monitor` e `ReceptorDeTrapsSnmp`, respectivamente.
F2	Componentes `GeradorDeEventoFalhaLimiar`, `GeradorDeEventoFalhaTrap` e `GeradorDeEventoFalhaComHisterese`.
F3	A possibilidade de interconectar um `GeradorDeEventoFalhaLimiar` a diferentes monitores para que ele tenha a informação necessária para avaliar a ocorrência da falha.
F4	Componente `CorrelatorDeEventoFalha` e variações.
F5, F6	Modelo de interconexão de componentes baseado em fontes e consumidores de eventos.
F7	Configuração visual dos componentes.
F10	O *framework* se baseia, apenas, na obtenção de informação de gerência.
NF1	Interfaces `ElementoGerenciado` e `InfoGerencia`.
NF2	Componentes `ElementoGerenciadoSnmp`, `InfoGerenciaSnmp` e `ReceptorDeTrapsSnmp`.
NF3 - 3.1, 3.2	*Framework* cuida dos detalhes associados a um protocolo de gerência específico através de seus componentes. O programador só precisa configurar os componentes devidamente.
NF3 – 3.3	Método `verificaOcorrenciaEventoFalha` dos componentes `GeradorDeEvento`

	`FalhaLimiar` e `GeradorDeEventoFalhaTrap`.
NF3 – 3.4	Instanciação gráfica dos componentes.
NF4	Utilização de componentes para implementar a funcionalidade da aplicação.
NF5	Componentes do tipo `GrupoInfoGerencia` e monitoração paralela das entidades gerenciadas da rede.
NF6	Combinação de *framework* e componentes.
NF7	Utilização da linguagem Java para especificação e posterior implementação dos componentes.

Tabela 4.2. Requisitos e características da solução proposta

4.4. Exemplos de Aplicações

Nesta seção, são apresentados três exemplos de aplicações construídas com base nos componentes do *framework*. São mostrados três diferentes cenários em que os componentes especificados são apropriadamente configurados e interconectados a fim de que se gerencie a rede conforme necessário. Em cada exemplo, são destacadas as três etapas a serem realizadas durante a construção de uma aplicação: construção de novos componentes, instanciação e configuração dos componentes necessários e interconexão dos componentes instanciados.

Exemplo 1

Deseja-se monitorar um roteador `R` a fim de se identificar se a taxa de erros de uma de suas interfaces ultrapassou determinado valor. Em caso positivo, um alarme deve ser disparado (uma indicação deve aparecer sobre o equipamento que o representa na interface gráfica).

1.1. Construção de novos componentes

1.1.1. Um `GeradorDeAlarmesHisterese` para gerar alarmes diante da ocorrência do evento que indica alta taxa de erros na interface do roteador `R`.

```
GeradorDeAlarmesHisterese implements EventoFalhaListener,
                                                Serializable{
    //Implementando a interface EventoFalhaListener.
    public void processaEventoFalha (EventoFalhaEvent ef){
         GeradorDeEventoFalha fonte = ef.getFonte();

         //Muda interface, indicando a ocorrência da falha.
         if (fonte.getValorAtingido() == fonte.valorLimiar)
```

```
                UI.instancia().mudaCor(fonte.getOrigem(),
                                fonte.get.prioridade());
        }
    }
```

1.2. Instanciação e configuração de componentes

1.2.1. Um `ElementoGerenciadoSnmp` para representar o roteador a ser gerenciado (`R`) .

`R` = {`nome`: rt-dsc.ufpb.br, `endIp`: 150.165.75.171,
`nomeResponsavel`: Administrador, `contatoResponsavel`: admin@dsc.ufpb.br, `prioridade`: prioridadeAlta}[12]

1.2.2. Um `GrupoInfoGerencia` (`grupoIG`) para agregar a informação a ser fornecida por `R`.

`grupoIG` = {`addInfoGerencia` (`InfoGerenciaSnmp` ("TaxaDeErros", "ifInErrors")}

1.2.3. Um `Monitor` (`monitor`) para controlar a freqüência de monitoração de `R.`

`monitor` = {`periodo`: 1800s}

1.2.4. Um `GeradorDeEventoFalhaComHisterese` (`gHisterese`) para verificar a taxa de erros de `R` a partir da informação obtida do `Monitor monitor.`

`gHisterese` = {`nome`: "TaxaDeErrosAlta", `origem`: R,
`prioridade`: prioridadeAlta, `nomeEventoRearm`: "TaxaDeErrosOk", `prioridadeEventoRearm`: prioridadeBaixa, `rearm`: x, `limiar`: y, `gig`: grupoIG}

1.2.5. Um `GeradorDeAlarmesHisterese` (`gAlarmes`) para notificar sobre uma alta taxa de erros em `R.`

1.3. Interconexão dos componentes

[12] Há outras propriedades que o leitor pode conferir em http://www.dsc.ufpb.br/~raissa/doc/frame.html.

1.3.1. Associamos o `ElementoGerenciadoSnmp R` e o `GrupoInfo Gerencia grupoIG` ao `Monitor monitor`, informando ao `monitor` qual informação de gerência ele deve obter e de onde.

1.3.2. Associamos o `GeradorDeEventoFalhaComHisterese gHisterese` ao `Monitor monitor` para que `gHisterese` possa obter o resultado da monitoração feita pelo `monitor`.

1.3.3. Associamos o `GeradorDeAlarmesHisterese gAlarmes` ao `GeradorDeEventoFalhaComHisterese gHisterese` para que toda vez que a falha verificada por `gHisterese` ocorrer, isto é, toda vez que a taxa de erros ultrapassar o limiar configurado, um alarme seja disparado.

O diagrama de seqüência da figura 4.32 mostra o fluxo de eventos resultante da associações descritas.

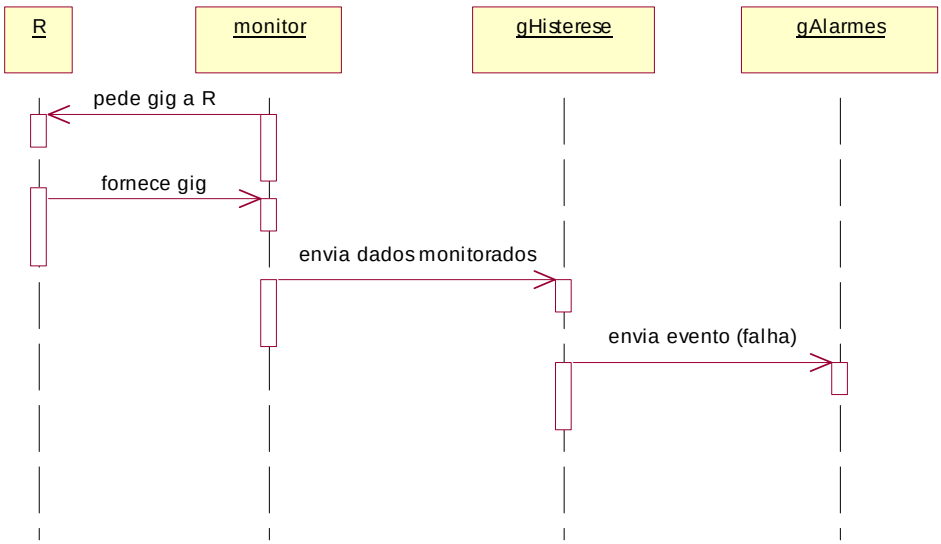


Figura 4.32. Exemplo 1

Exemplo 2

Deseja-se prever um congestionamento no roteador R da rede. Suponha que existam dois servidores, S_1 e S_2. S_1 tem, normalmente, muitas aplicações (tráfego em rajada) executando de forma concorrente, as quais acessam um servidor remoto; e S_2 é um servidor

multimídia que gera um alto tráfego na rede. O grande número de conexões abertas em S_1 aliado à grande largura de banda consumida por S_2, leva eventualmente a taxas inaceitáveis de perda de pacotes no roteador R. Deseja-se, portanto, monitorar S_1 e S_2 para que se possa identificar o congestionamento antes que ele ocorra e, assim, realizar uma gerência pró-ativa.

2.1.Construção de novos componentes

2.1.1.Um `GeradorTaxaDeTrafegoAlta` para avaliar o número de conexões abertas em S_1 e o tráfego gerado por S_2.

```
public void GeradorTaxaDeTrafegoAlta extends
            GeradorDeEventoFalhaLimiar implements Serializable{
    public boolean verificaOcorrenciaEventoFalha
                                (ElementoGerenciado egs[]){
     //Verifica se o número de conexões TCP abertas em S1 e o
     //tráfego gerado por S2 ultrapassam seus respectivos
     //limiares. Retorna true em caso positivo.
     return(getInfoGerenciaDaOrigem("QuantConexoesAbertas",
               egs[0]) > Int.parseInt (getLimiares
               ("QuantConexoesAbertas").getValor()) &&
               (getInfoGerenciaDaOrigem("TaxaDeTrafego",
               egs[1]) > Double.parseDouble (getLimiares
               ("TaxaDeTrafego").getValor())
    }
}
```

2.1.2.Um `GeradorDeAlarmes` para gerar alarmes diante da ocorrência do evento que indica um potencial congestionamento no roteador R.

```
GeradorDeAlarmes implements EventoFalhaListener, Serializable{
    //Implementando a interface EventoFalhaListener.
    public void processaEventoFalha (EventoFalhaEvent ef){
      GeradorDeEventoFalha fonte = ef.getFonte();
      //Envia uma mensagem, indicando a ocorrência da falha
      UI.instancia().enviaMensagem
                        (fonte.get.contatoResponsavel());
    }
}
```

2.2. Instanciação e configuração de componentes

2.2.1.Dois componentes `ElementoGerenciadoSnmp` para representar cada um dos servidores (S_1 e S_2).

S_1 = {nome: s1.ufpb.br, endIp: 150.165.75.172, nomeResponsavel: Administrador, contatoResponsavel: admin@dsc.ufpb.br, prioridade: prioridadeMedia}

S_2 = {nome: s2.ufpb.br, endIp: 150.165.75.173, nomeResponsavel: Administrador, contatoResponsavel: admin@dsc.ufpb.br, prioridade: prioridadeMedia}

2.2.2. Um GrupoInfoGerencia para agregar a informação ($grupoIG_1$) a ser fornecida por S_1.

$grupoIG_1$ = {addInfoGerencia (InfoGerenciaSnmp ("Quant ConexoesAbertas", "tcpCurrEstab")}

2.2.3. Um GrupoInfoGerencia para agregar a informação ($grupoIG_2$) a ser fornecida por S_2.

$grupoIG_2$ = {addInfoGerencia (InfoGerenciaSnmp ("TaxaDe Trafego", "tcpOutSegs")}

2.2.4. Dois componentes Monitor ($monitor_1$ e $monitor_2$) - um para controlar a freqüência de monitoração de S_1 e outro para controlar a freqüência de monitoração de S_2.

$monitor_1$ = {periodo: 300s}

$monitor_2$ = {periodo: 300s}

2.2.5. Um GeradorTaxaDeTrafegoAlta (gTrafegoAlto) para verificar o congestionamento em R a partir da informação obtida dos monitores $monitor_1$ e $monitor_2$.

gTrafegoAlto = {nome: "TaxaDeTrafegoAlta", origem: R, prioridade: prioridadeAlta, limiares {(“QuantConexoesAbertas, 15), (“TaxaDeTrafego”, 500K)}}

2.2.6. Um GeradorDeAlarmes (gAlarmes) para notificar o operador da rede sobre o congestionamento em R.

2.3. Interconexão dos componentes

2.3.1. Associamos o `ElementoGerenciadoSnmp` S_1 e o `GrupoInfo Gerencia grupoIG`$_1$ ao `Monitor monitor`$_1$, informando ao `monitor`$_1$ qual informação de gerência ele deve obter e de onde.

2.3.2. Associamos o `ElementoGerenciadoSnmp` S_2 e o `GrupoInfo Gerencia grupoIG`$_2$ ao `Monitor monitor`$_2$, informando ao `monitor`$_2$ qual informação de gerência ele deve obter e de onde.

2.3.3. Associamos o `GeradorTaxaDeTrafegoAlta gTrafegoAlto` aos monitores `monitor`$_1$ e `monitor`$_2$ para que `gTrafegoAlto` possa obter o resultado da monitoração feita por ambos.

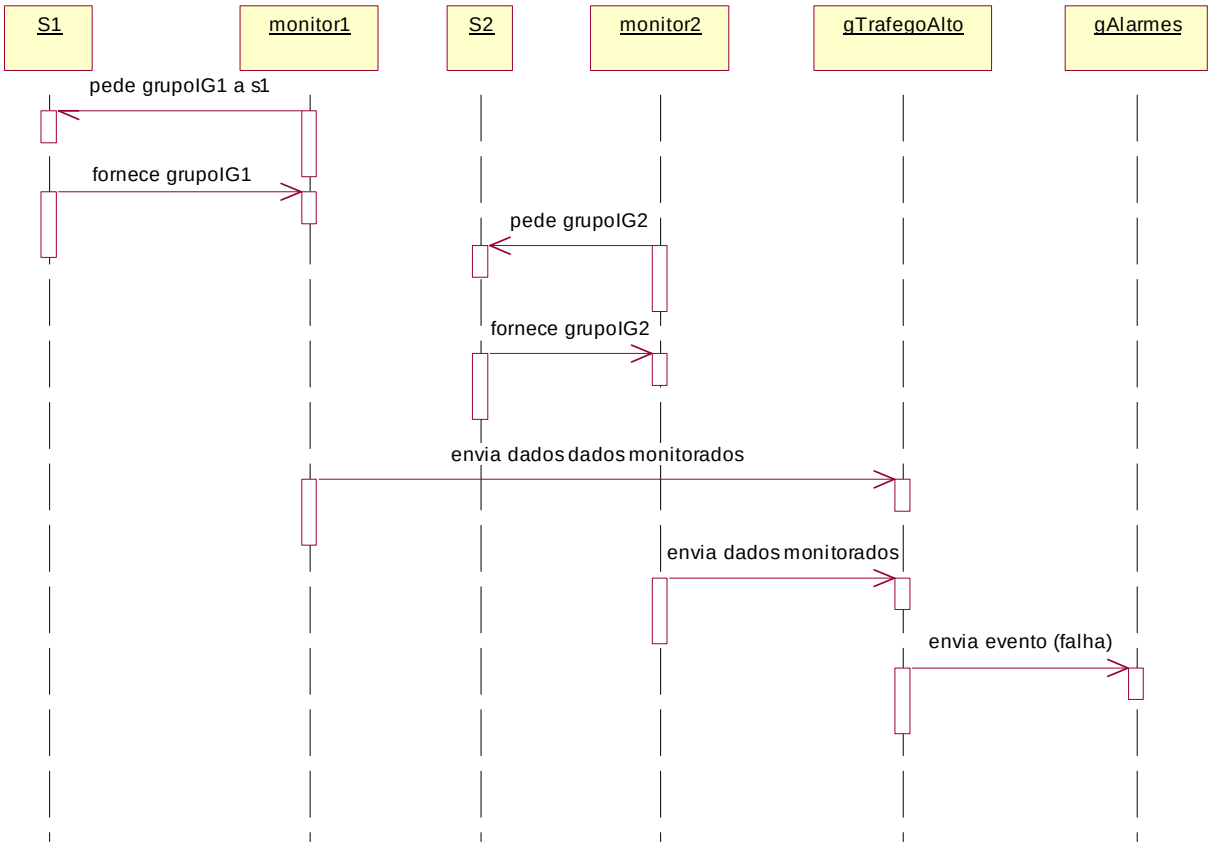


Figura 4.33. Exemplo 2

2.3.4. Associamos o `GeradorDeAlarmes gAlarmes` ao `GeradorTaxaDe TrafegoAlta gTrafegoAlto`, para que toda vez que a falha verificada por `gTrafegoAlto` ocorrer, ou seja, toda vez que a taxa de tráfego ultrapassar o limiar configurado, um alarme seja disparado.

O diagrama de seqüência da figura 4.33 mostra o fluxo de eventos resultante da associações descritas.

Exemplo 3

Deseja-se monitorar os roteadores de uma rede: R_1, R_2, R_3.

* O operador da rede deve ser notificado sempre que uma das interfaces de cada roteador estiver com problemas (`down`), mas o operador só deve saber se houve problema em R_2 caso este problema seja observado mais de uma vez na última meia hora.

* Quando a taxa de pacotes descartados por R_3 atingir determinado valor, deve-se monitorá-lo mais freqüentemente a fim de que se possa identificar um congestionamento o mais rápido possível. Tão logo a situação se normalize, deve-se voltar à freqüência de monitoração normal.

* Deve-se notificar o operador da rede quando houver um congestionamento em R_3.

* Deve-se registrar cada uma das ocorrências acima.

3.1.Construção de novos componentes

3.1.1.Um `GeradorLinkDown` para verificar se o *trap* recebido é do tipo `linkDown`.

```
GeradorLinkDown extends GeradorDeEventoFalhaTrap implements
                                                    Serializable{
    public boolean verificaOcorrenciaEventoFalha(TrapEvent trap){
          //Verifica o tipo de trap.
          return trap.getTipodeTrap()== trap.linkDown;
    }
}
```

3.1.2.Um `GeradorDeLogs` para registrar as ocorrências de falhas.

```
GeradorDeLogs implements EventoFalhaListener, Serializable{

    //Implementando a interface EventoFalhaListener.
    public void processaEventoFalha(EventoFalhaEvent ef){
          //Registra ocorrência da falha.
          UI.instancia().logEvento(ef);
```

```
    }
}
```

3.1.3.Um `GeradorDeAlarmesHisterese` para gerar alarmes diante da ocorrência de um congestionamento no roteador R.

```
GeradorDeAlarmesHisterese implements EventoFalhaListener,
                                                Serializable{
    //Implementando a interface EventoFalhaListener.
    public void processaEventoFalha (EventoFalhaEvent ef){
         GeradorDeEventoFalha fonte = ef.getFonte();
         //Envia uma mensagem, indicando a ocorrência da falha.
         if ( fonte.getValorAtingido() == fonte.valorLimiar)
         UI.instancia().enviaMensagem
                         (fonte.get.contatoResponsavel());
    }
}
```

3.1.4.Um `MonitorInteligente` que sabe se auto-reconfigurar com base no tipo de evento recebido.

```
MonitorInteligente extends Monitor implements
                                        EventoFalhaListener{
    //O período de monitoração quando não há indicação de
    //congestionamento. Freqüência mínima.
    public static final periodoNormal = 1800;

    //O período de monitoração quando há indicação de
    //congestionamento. Freqüência máxima.
    public static final periodoMinimo = 900;

    //Implementando a interface EventoFalhaListener.
    public void processaEventoFalha(EventoFalhaEvent ef){
          //Se não há indicação de congestionamento, utiliza
          //período de monitoração normal. Caso contrário,
          //utiliza o periodoMinimo, aumentando a freqüência.
          if (ef.get.Fonte().getNome()=="SituacaoNormal")
              setPeriodo(periodoNormal);
          else setPeriodo (peridoMinimo);
    }
}
```

3.2. Instanciação e configuração de componentes

3.2.1. Três componentes `ElementoGerenciadoSnmp` para representar cada um dos roteadores (R_1, R_2 e R_3).

R_1 = {`nome`: rt1-dsc.ufpb.br, `endIp`: 150.165.75.171, `nome Responsavel`: Administrador, `contatoResponsavel`: admin@dsc.ufpb.br, `prioridade`: prioridadeAlta}

R_2 = {`nome`: rt2-dsc.ufpb.br, `endIp`: 150.165.75.170, `nomeResponsavel`: Administrador, `contatoResponsavel`: admin@dsc.ufpb.br, `prioridade`: prioridadeMedia}

R_3 = {`nome`: rt3-dsc.ufpb.br, `endIp`: 150.165.75.169, `nomeResponsavel`: Administrador, `contatoResponsavel`: admin@dsc.ufpb.br, `prioridade`: prioridadeAlta}

3.2.2. Um `ReceptorDeTrapsSnmp` (`receptorDeTraps`) para receber os *traps* gerados na rede.

3.2.3. Três componentes `GeradorLinkDown` ($gLinkDown_1$, $gLinkDown_2$ e $gLinkDown_3$) - um por roteador.

$gLinkDown_1$ = {`nome`: "LinkDown", `origem`: R_1, `prioridade`: prioridadeAlta}

$gLinkDown_2$ = {`nome`: "LinkDown", `origem`: R_2, `prioridade`: prioridadeMedia}

$gLinkDown_3$ = {`nome`: "LinkDown", `origem`: R_3, `prioridade`: prioridadeAlta}

3.2.4. Um `GrupoInfoGerencia` para agregar a informação (`grupoIG`) a ser fornecida por R_3.

`grupoIG` = {`addInfoGerencia` (`InfoGerenciaSnmp` ("Pacotes DescartadosOut", "ipOutDiscards")}

3.2.5. Um `MonitorInteligente` (`monitor`) para controlar a freqüência de monitoração de R_3.

`monitor` = {`periodo`: 1800s}

3.2.6. Um componente `GeradorDeEventoFalhaComHisterese` (`gHisterese`$_1$) para verificar uma **indicação** de congestionamento em R_3 e a partir do qual o `MonitorInteligente monitor` controlará sua freqüência de monitoração.

`gHisterese`$_1$ = {`nome`: "IndicacaoDeCongestionamento", `origem`: R_3, `prioridade`: prioridadeMedia, `nomeEventoRearm`: "SituacaoNormal", `prioridadeEventoRearm`: prioridadeBaixa, `rearm`: x, `limiar`: y, `gig`: grupoIG}

3.2.7. Um componente `GeradorDeEventoFalhaComHisterese` (`gHisterese`$_2$) para verificar o congestionamento em R_3 a partir da informação obtida do `MonitorInteligente monitor`.

`gHisterese`$_2$ = {`nome`: "Congestionamento", `origem`: R_3, `prioridade`: prioridadeAlta, `nomeEventoRearm`: "SituacaoNormal", `prioridadeEventoRearm`: prioridadeBaixa, `rearm`: x, `limiar`: z, `gig`: grupoIG}

3.2.8. Um `CorrelatorDeEventoFalhaSupressao` (`correlator`) para correlacionar os eventos recebidos de R_2.

correlator = {`contador`: 2, `intervaloDeTempo`: 1800s}

3.2.9. Um `GeradorDeAlarmesHisterese` (`gAlarmes`$_1$) para notificar sobre a ocorrência do congestionamento em R_3 e um `GeradorDeAlarmes` (`gAlarmes`$_2$) para notificar sobre problemas nas interfaces dos roteadores (vide exemplo 2).

3.2.10. Um `GeradorDeLogs` (`glog`) para registrar as ocorrências descritas.

3.3. Interconexão dos componentes

3.3.1. Associamos o `ElementoGerenciadoSnmp R3` e o `GrupoInfo Gerencia grupoIG` ao `Monitor monitor`, informando ao `monitor` qual informação de gerência ele deve obter e de onde.

3.3.2. Associamos os componentes `ElementoGerenciadoSnmp` `R1`, `R2` e `R3` ao `ReceptorDeTrapsSnmp (receptorDeTraps)`.

3.3.3. Associamos os componentes `GeradorLinkDown gLinkDonw`$_1$, `gLinkDown2` e `gLinkDown3`, ao `ReceptorDeTrapsSnmp receptorDeTraps` para que eles verifiquem a ocorrência de problemas em cada um dos roteadores.

3.3.4. Associamos os componentes `GeradorDeEventoFalhaCom Histerese gHisterese1` e `gHisterese2`, ao `Monitor monitor` para que `gHisterese1` e `gHisterese2` obtenham o resultado da monitoração realizada pelo `monitor`.

3.3.5. Associamos o `Monitor monitor` ao `GeradorDeEventoFalha ComHisterese gHisterese1` para que o `monitor` se auto-reconfigure no momento em que uma **indicação** de congestionamento seja verificada.

3.3.6. Associamos o `CorrelatorDeEventoFalhaSupressao correlator` ao `GeradorLinkDown gLinkDown`$_2$, para que ele correlacione o eventos gerados por `gLinkDown`$_2$.

3.3.7. Associamos o `GeradorDeAlarmesHisterese gAlarmes`$_1$ ao `GeradorDeEventoFalhaComHisterese gHisterese`$_1$, para que toda vez que houver um congestionamento em R_3, um alarme seja disparado.

3.3.8. Associamos o `GeradorDeAlarmes gAlarmes`$_2$ aos componentes `Gerador LinkDown gLinkDown`$_1$ e `gLinkDown`$_3$, e ao `CorrelatorDeEventoFalhaSupressao correlator`, para que o `gAlarmes`$_2$ gere um alarme toda vez que ocorrer problemas nas interfaces de R_1, R_2 e R_3.

3.3.9. Associamos o `GeradorDeLogs glog` aos geradores de eventos `gHisterese`$_1$, `gLinkDown`$_1$, `gLinkDown`$_3$ e ao `correlator` para que sejam registradas as devidas ocorrências (estes componentes foram duplicados na figura abaixo apenas para facilitar o entendimento da figura).

O diagrama de seqüência da figura 4.34 mostra o fluxo de eventos resultante das associações descritas. A diagrama foi divido em duas partes por questão de espaço.

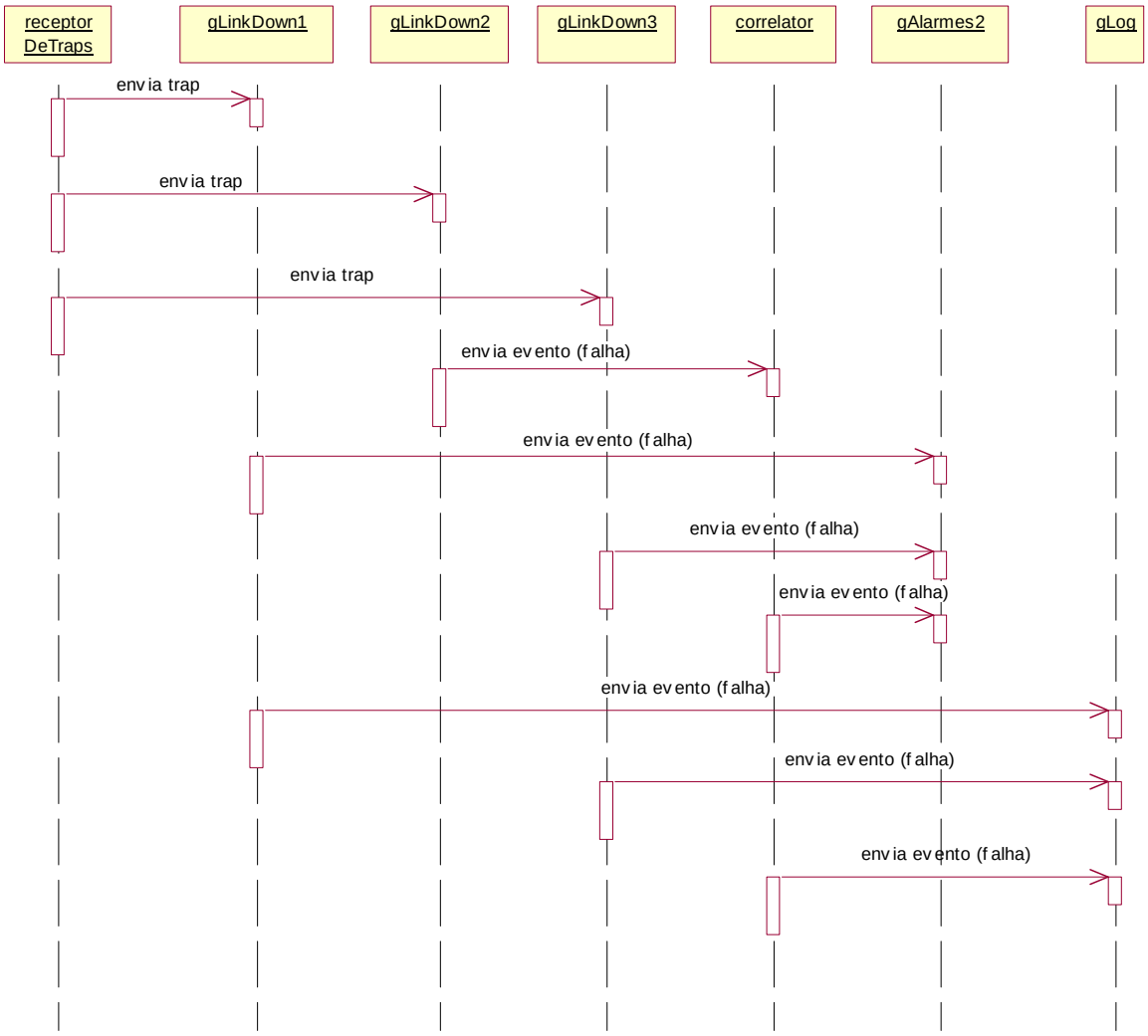


Figura 4.34. Exemplo 3 (parte I)

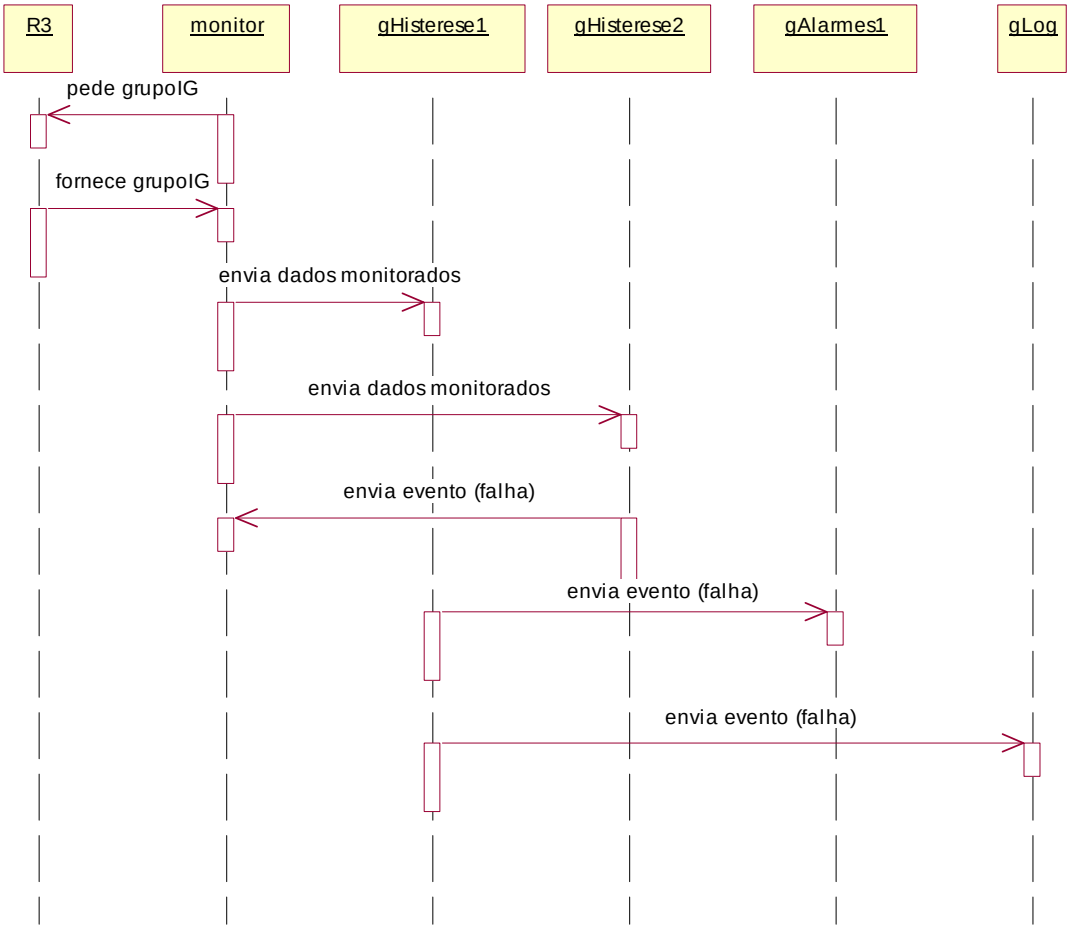


Figura 4.34. Exemplo 3 (parte II)

Capítulo 5

Uma Proposta de Implementação

No capítulo 4, especificamos um *framework* CO para a gerência de falhas. Vimos como utilizar os componentes, classes e interfaces especificados, aproveitando toda a reusabilidade fornecida não só em termos de implementação, mas também em termos de análise e de projeto.

Neste capítulo, os componentes vistos no capítulo anterior são mais detalhadamente descritos. Na seção 5.1, discutimos o projeto interno dos componentes básicos do *framework*. Destacamos, então, aspectos internos de implementação que tornam o *framework* capaz de executar conforme as características gerais e específicas de cada aplicação de gerência de falhas, como a utilização de padrões de projeto [Gamma *et al*., 1994] e a utilização de características da linguagem de programação (Java) utilizada para especificação dos componentes. Na seção 5.2, outras considerações também são feitas sobre ferramentas de suporte necessárias para a "composição" de aplicações.

5.1. O Projeto Interno dos Componentes Básicos do *Framework*

Na seção 4.2, quando o projeto arquitetural do *framework* CO foi definido, foi sugerida a utilização de uma linguagem de programação orientada a objetos para implementação de cada componente. De fato, como o leitor pode conferir a partir dos exemplos fornecidos no capítulo 4, a linguagem Java foi utilizada.

Para construir componentes de acordo com esta linguagem, algumas "regras" devem ser observadas conforme descrito pelo apêndice B. Cada componente é definido como um conjunto de classes, interfaces e outros recursos (arquivos de configuração, figuras, etc.), de modo que, segundo o projeto arquitetural apresentado no capítulo 4, deve-se estender classes e redefinir métodos para adicionar funcionalidade a um componente semipronto fornecido pelo *framework*.

Sendo assim, embora o programador possa utilizar componentes no momento da configuração visual da aplicação, em termos de implementação o *framework* consiste de classes e interfaces que, ao serem agrupadas, formam os componentes especificados. Estas classes e interfaces estão nos diagramas UML da figura 5.1, presente no final desta seção, e estão devidamente especificados de acordo com a linguagem Java em http://www.dsc.ufpb.br/~raissa/doc/frame.html. A tabela 5.1 mostra como estas classes e interfaces são agrupadas para que cada componente possa desempenhar sua função específica.

Componente	**Classes**	**Interfaces**	**Eventos que consome**	**Eventos que produz**
Elemento GerenciadoSnmp	ElementoGerenciadoAdapter, ElementoGerenciadoSnmp	ElementoGerenciado		
InfoGerenciaSnmp	InfoGerenciaAdapter, InfoGerenciaSnmp	InfoGerencia		
GrupoInfoGerencia	GrupoInfoGerencia			
ConfiguradorDe RequisicaoSnmp	CosfiguradoDeRequisicao, ConfiguradorDeRequisicao Snmp			
Monitor	Monitor			MonitorEvent
ReceptorDeTraps Snmp	ReceptorDeTraps, ReceptorDeTrapsSnmp			TrapEvent
GeradorDeEvento FalhaLimiar	GeradorDeEventoFalha, GeradorDeEventoFalhaLimiar	MonitorListener	MonitorEvent	EventoFalha Event
GeradorDeEvento FalhaComHisterese	GeradorDeEventoFalha, GeradorDeEventoFalha ComHisterese	MonitorListener	MonitorEvent	EventoFalha Event
GeradorDeEvento FalhaTrap	GeradorDeEventoFalha, GeradorDeEventoFalhaTrap	TrapListener	TrapEvent	EventoFalha Event
CorrelatorDe EventoFalha Contador	CorrelatorDeEventoFalha, CorrelatorDeEventoFalha Contador	EventoFalhaListener	EventoFalha Event	EventoFalha Event
CorrelatorDe EventoFalha Supressor	CorrelatorDeEventoFalha, CorrelatorDeEventoFalha Supressor	EventoFalhaListener	EventoFalha Event	EventoFalha Event
CorrelatorDe EventoFalhaCMP	CorrelatorDeEventoFalha, CorrelatorDe	EventoFalhaListener	EventoFalha Event	EventoFalha Event

	EventoFalhaCMP			

Tabela 5.1. As classes e interfaces de cada componente

Cada componente possui pelo menos uma classe, cujo nome é igual ao componente ao qual pertence e cuja responsabilidade é implementar a funcionalidade básica do componente do qual faz parte. Para isto, esta classe pode estender outras classes, pode implementar algumas interfaces e pode ainda ser estendida, no caso de um componente semipronto. Assim, a classe `ElementoGerenciadoSnmp` estende a classe `ElementoGerenciadoAdapter` e é responsável por fornecer informação de gerência SNMP; a classe `Monitor` é responsável por fazer a monitoração síncrona da rede, gerando eventos do tipo `MonitorEvent`; a classe `GeradorDeEventoFalhaLimiar` herda todos os métodos e atributos da classe `GeradorDeEventoFalha`, implementa a interface `MonitorListener`, gera eventos do tipo `EventoFalhaEvent` e ainda deve ser estendida (lembre-se que o componente `GeradorDeEventoFalhaLimiar` é um componente semipronto) para que, de acordo com a funcionalidade adicionada, ela possa finalmente verificar a ocorrência de falhas segundo os limiares pré-configurados; e assim por diante.

Algumas das novas classes até então não mencionadas são resultados de generalizações feitas, como é o caso da classe `GeradorDeEventoFalha`. Outras classes, como `ElementoGerenciadoAdapter` e `InfoGerenciaAdapter` são simplesmente classes de adaptação (padrão de projeto *Adapter* [Gamma *et al*., 1994]) que facilitam a implementação das interfaces `ElementoGerenciado` e `InfoGerencia`, respectivamente[13]. Outras classes e interfaces, entretanto, como as classes `MonitorEvent`, `Monitor` e `ReceptorDeTraps` e as interfaces `MonitorListener` e `TrapListener`, possuem algumas características importantes para o funcionamento interno do *framework* e, portanto, devem ser vistas mais detalhadamente.

[13]Estas classes abstratas implementam a maioria dos métodos especificados pelas interfaces ElementoGerenciado e InfoGerencia, de modo que ao invés de implementar estas interfaces por inteiro, o programador pode simplesmente estender as classes adaptadoras, redefinindo apenas os métodos necessários.

5.1.1. Classes e Interfaces de Uso Interno

Conforme lista a tabela 5.1, outras classes e interfaces até então não citadas foram especificadas. Trata-se de classes e interfaces de uso interno do *framework* que servem para implementar parte da sua funcionalidade interna e que, conseqüentemente, não podem ser diretamente utilizadas pelo programador. É o caso das classes `GeradorDeEventoFalha` e `MonitorEvent` e das interfaces `MonitorListener` e `TrapListener`.

A classe **`GeradorDeEventoFalha`** é uma superclasse da qual os geradores de eventos que identificam falhas na rede herdam todos os métodos e atributos. Entretanto, a sua funcionalidade é reutilizada no momento em que o programador da aplicação estende uma de suas **subclasses**: `GeradorDeEventoFalhaLimiar` ou `GeradorDeEventoFalhaTrap`. Novos componentes não podem ser criados a partir de uma extensão direta da classe `GeradorDeEventoFalha` por dois motivos:

> Esta classe não possui o método abstrato `verificaOcorrenciaEventoFalha` e, conseqüentemente, não obriga o programador a redefini-lo no momento da construção do novo componente. O *framework*, por sua vez, exige (precisa) que isto seja feito para que ele possa identificar falhas na rede a partir da utilização de componentes criados pelo próprio programador.
>
> A classe `GeradorDeEventoFalha` não é, por si só, consumidora de eventos (isto é, não implementa nenhuma interface com este fim) e, assim, não obtém a informação de gerência necessária para a avaliação da ocorrência de falhas. As relações fonte/consumidor, a serem herdadas pelos novos componentes construídos, só são bem identificadas a partir das subclasses `GeradorDeEventoFalhaLimiar` e `GeradorDeEventoFalhaTrap`; ou seja, a classe `GeradorDeEventoFalhaLimiar` é consumidora do `Monitor` e a classe `GeradorDeEventoFalhaTrap` é consumidora do `ReceptorDeTraps`. A construção de novos componentes a partir destas subclasses é que faz com que os novos componentes criados sejam, por herança, consumidores de eventos dos tipos de fontes apropriadas.

Sendo assim, do ponto de vista do programador, apenas os componentes semiprontos `GeradorDeEventoFalhaLimiar` e `GeradorDeEventoFalhaTrap` podem ser

utilizados. A classe `GeradorDeEventoFalha` serve apenas para concentrar algumas das características comuns às suas subclasses.

A classe `MonitorEvent` e as interfaces `MonitorListener` e `TrapListener`, por sua vez, juntamente com as classes `TrapEvent` e `EventoFalhaEvent` e com a interface `EventoFalhaListener`, determinam as associações que podem ser estabelecidas entre os componentes do *framework*, viabilizando a implementação do modelo de execução baseado em fontes e consumidores de eventos conforme descrito pela figura 4.31.

A classe **`MonitorEvent`** representa os eventos gerados pelo componente `Monitor`, os quais contêm, conseqüentemente, os resultados das monitorações realizadas. Para receber eventos deste tipo, um componente deve implementar a interface **`MonitorListener`** e deve se cadastrar na instância do `Monitor` que fornece os eventos de interesse. Assim sendo, as classes `GeradorDeEventoFalhaLimiar` e `GeradorDeEventoFalhaComHisterese` implementam a interface `MonitorListener`, de modo que componentes deste tipo possam ser conectados a componentes do tipo `Monitor`, recebendo a informação necessária à identificação de falhas na rede.

Analogamente, para receber os eventos que descrevem os *traps* gerados na rede, isto é, eventos do tipo `TrapEvent`, um componente deve implementar a interface **`TrapListener`** e deve se cadastrar na instância do `ReceptorDeTraps` responsável pela monitoração assíncrona da rede. A classe `GeradorDeEventoFalhaTrap`, então, implementa a interface `TrapListener` para que componentes deste tipo possam ser conectados a um `ReceptorDeTraps` e, conseqüentemente, possam identificar falhas com base nos *traps* recebidos.

A implementação das interfaces `MonitorListener` e `TrapListener`, contudo, não é tarefa do programador que constrói novos componentes do tipo `GeradorDeEventoFalhaLimiar` ou `GeradorDeEventoFalhaTrap`. Conforme dito, estas interfaces são implementadas **internamente** pelas classes `GeradorDeEventoFalhaLimiar` e `GeradorDeEventoFalhaTrap`, de modo que os novos componentes criados a partir da extensão destas classes se tornam, por herança, consumidores de eventos do tipo `MonitorEvent` e `TrapEvent`, respectivamente. Então, conforme vimos no capítulo 4, para construir novos componentes para identificação de falhas, basta estender as classes `GeradorDeEventoFalhaLimiar` e `GeradorDeEventoFalhaTrap`, redefinindo o

método `verificaOcorrenciaEventoFalha`. Feito isto, o novo componente criado já pode ser instanciado, configurado e conectado a componentes do tipo `Monitor` ou `ReceptorDeTraps` para receberem a informação de cuja análise serão identificadas as falhas na rede.

Esta implementação interna das interfaces `MonitorListener` e `TrapListener` não só parece "confortável" para o programador, mas também permite que o *framework* implemente parte de sua própria funcionalidade. É através dos métodos `processaDadosMonitorados` (interface `MonitorListener`) e `processaTrap` (interface `TrapListener`) que o *framework* (1) trata a informação de gerência resultante da monitoração realizada, (2) **executa o método `verificaOcorrenciaEventoFalha`** e (3) gera novos eventos quando as falhas observadas ocorrem. O programador, portanto, deve apenas reutilizar a funcionalidade embutida no *framework*. Para que este algoritmo não seja modificado, o que interferiria no fluxo de execução interno do *framework*, os métodos `processaDadosMonitorados` e `processaTrap` são do tipo *final*. Métodos Java do tipo *final* não podem ser redefinidos nas subclasses que os herdam, de modo que, neste caso, a implementação feita pelo *framework* pode ser apenas reutilizada.

Os métodos **`processaDadosMonitorados`** e **`processaTrap`**, portanto, são **métodos *template*** (padrão de projeto *Template Method* [Gamma *et al*., 1994]) que, uma vez escritos, não podem ser modificados. Contudo eles executam outros métodos definidos pelo próprio usuário do *framework* dentro de seu fluxo de controle, (no caso, o método `verificaOcorrenciaEventoFalha`), possuindo além de partes fixas, partes variáveis através das quais alguma funcionalidade particular pode ser adicionada. É, também, dentro destes métodos *template* que estruturas de dados internas necessárias para o funcionamento dos componentes são atualizadas. No caso do `GeradorDeEventoFalhaLimiar,` que pode receber informação de diferentes monitores, uma estrutura de dados própria é mantida, de modo a organizar e disponibilizar a informação recebida dos diferentes monitores; esta informação é atualizada através do método `processaDadosMonitorados`.

Um outro método *template* resulta da implementação da interface `EventoFalha Listener` pela classe `CorrelatorDeEventoFalha`. Ao implementar esta interface, a classe `CorrelatorDeEventoFalha` torna o método **`processaEventoFalha`** um método do tipo *final* que simplesmente executa o método `correlaciona` definido pelo

programador. Os novos componentes criados a partir da extensão desta classe podem ser, então, automaticamente conectados a qualquer tipo de gerador de evento (subclasses de `GeradorDeEventoFalha`), já que, por herança, eles também implementam a interface `EventoFalhaListener`, podendo receber os eventos de interesse para a correlação a ser realizada. Outros componentes criados diretamente a partir da implementação desta interface, entretanto, podem/devem redefinir o método `processaEventoFalha`, incluindo nele qualquer ação cabível a ser realizada diante da ocorrência de determinado tipo de falha.

5.1.2. Classes Ativas

Outras duas classes devem ser destacadas. Trata-se das **classes ativas** `Monitor` e `ReceptorDeTraps`. Estas classes implementam a interface Java `java.io.Runnable`, abastecendo o *framework* com a informação de gerência necessária para que os eventos sejam trocados entre os componentes interconectados. O fluxo de execução interno do *framework*, portanto, começa com a ativação destas classes.

Como fontes de eventos, as classe `Monitor` e `ReceptorDeTraps` possuem, juntamente com a classe `GeradorDeEventoFalha`, os métodos necessários para a gerência dos componentes nelas cadastrados. A saber:

- Classe `Monitor`: `addMonitorListener`, `removeMonitorListener` e `disparaProcessaDadosMonitorados`;
- Classe `ReceptorDeTraps`: `addTrapListener`, `removeTrapListener` e `disparaProcessaTrap`;
- Classe `GeradorDeEventoFalha`: `addEventoFalhaListener`, `removeEventoFalhaListener` e `disparaProcessaEventoFalha`;

Os métodos `addMonitorListener`, `addTrapListener` e `addEventoFalhaListener` efetuam o cadastro de componentes interessados em receber os novos eventos gerados; os métodos `removeMonitorListener`, `removeTrapListener` e `removeEventoFalhaListener` cancelam o cadastro de componentes previamente cadastrados; e os métodos `disparaProcessaDadosMonitorados`, `disparaProcessaTrap` e `disparaProcessaEventoFalha` enviam cada novo evento gerado para todos os componentes cadastrados. O envio de um novo evento consiste em executar o

método da interface implementada por cada componente cadastrado, repassando o evento gerado. Assim, o `Monitor` executa o método `processaDadosMonitorados`, o `ReceptorDeTraps` executa o método `processaTrap` e o `GeradorDeEventoFalha` executa o método `processaEventoFalha` de cada componente cadastrado, para que tais componentes possam tratar a informação recebida.

Então, uma vez que os componentes estejam cadastrados às classes ativas `Monitor` e `ReceptorDeTraps` e uma vez que estas classes sejam ativadas, a rede passará a ser permanentemente monitorada, não faltando a informação necessária para que outros componentes possam realizar sua função. Uma outra classe chamada **`FwGF`** é a classe responsável por inicializar as classes ativas instanciadas, dando início à execução da aplicação de acordo com a configuração feita visualmente pelo usuário do *framework*. Com isso, a rede passa a ser efetivamente gerenciada conforme a configuração realizada.

5.2. Ferramentas para a "Composição" de Aplicações

Uma vez que os componentes sejam implementados e estejam finalmente disponíveis para uso, a aplicação pode ser "composta" visualmente. Em outras palavras, os componentes podem ser instanciados e interconectados através de um ambiente gráfico para que, a partir daí, o *framework* possa executar de acordo com uma configuração específica. Sendo assim, a primeira ferramenta de suporte, necessária para que uma aplicação seja construída com base em componentes, é um **editor gráfico**.

Um editor gráfico para os componentes especificados deve permitir:

configurar as propriedades de cada componente instanciado; e

interconectar os componentes instanciados.

Algumas ferramentas como o *BeanBox* (vide apêndice B) permitem que isto seja feito. Entretanto, um editor mais apropriado e especialmente projetado para os componentes especificados deve ser utilizado.

Primeiramente, o editor deve fornecer um editor especial para propriedades do tipo `Map`. `Map` é uma interface do JDK (*Java Development Kit*) 1.2 que mapeia chaves - nomes, por exemplo - para valores (`TreeMap` e `HashMap` são exemplos de implementações desta interface). Isto se faz necessário já que componentes do tipo `GrupoInfoGerencia` devem

ser implementados internamente através de estruturas deste tipo (vide http://www.dsc.ufpb.br/~raissa/doc/frame.html) e, sendo assim, tal editor é indispensável para que se possa adicionar unidades de informação de gerência (`InfoGerencia`) a um grupo de informação qualquer. Ferramentas como o *BeanBox*, fornecem apenas editores para propriedades com tipos nativos da linguagem suportada.

Depois, o editor gráfico deve fornecer editores especiais para simplificar a interconexão dos componentes. Tais editores devem conhecer a semântica dos componentes a serem interconectados para que as associações feitas pelo programador possam ser entendidas e validadas. Por exemplo, é necessário fornecer um editor que permita associar a instância de um `ElementoGerenciado` a uma instância de um `Monitor`, de modo que o `Monitor` passe a controlar a freqüência com que o `ElementoGerenciado` deve fornecer determinada informação de gerência. Editores que permitam interconectar fontes e consumidores de eventos também são necessários.

Além disso, o programador deve poder utilizar um **banco de dados de topologia** que armazene a configuração da rede a ser gerenciada, ou seja, a configuração dos componentes do tipo `ElementoGerenciado` que devem ser instanciados. Isto facilita a construção de novas aplicações de gerência para a mesma configuração de rede e implica em possibilitar a integração entre o editor gráfico e o banco de dados de topologia onde a informação de interesse está armazenada (este banco de dados poderia ser atualizado pelo programador ou através de ferramentas de descobrimento automático de topologia).

Assim, pelo menos duas ferramentas básicas de suporte à “composição” de aplicações devem estar disponíveis: um editor gráfico e um banco de dados de topologia da rede. Uma vez que a configuração seja concluída, aproveitando todas as facilidades que tais ferramentas oferecem, o *framework* deve ser ativado para que a rede possa ser devidamente gerenciada, reutilizando toda a funcionalidade nele embutida.

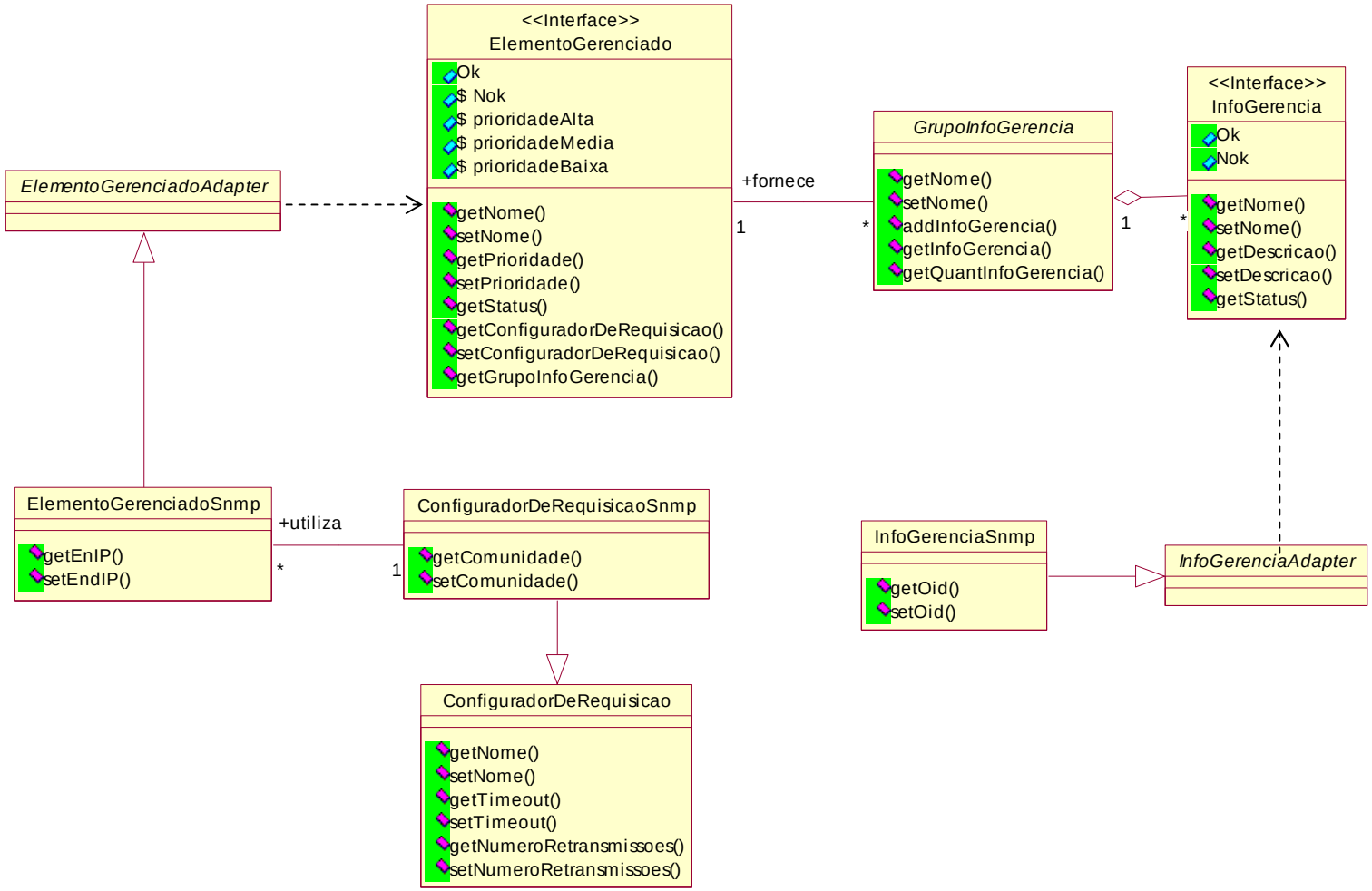


Figura 5.1. Diagrama de classes (parte I)

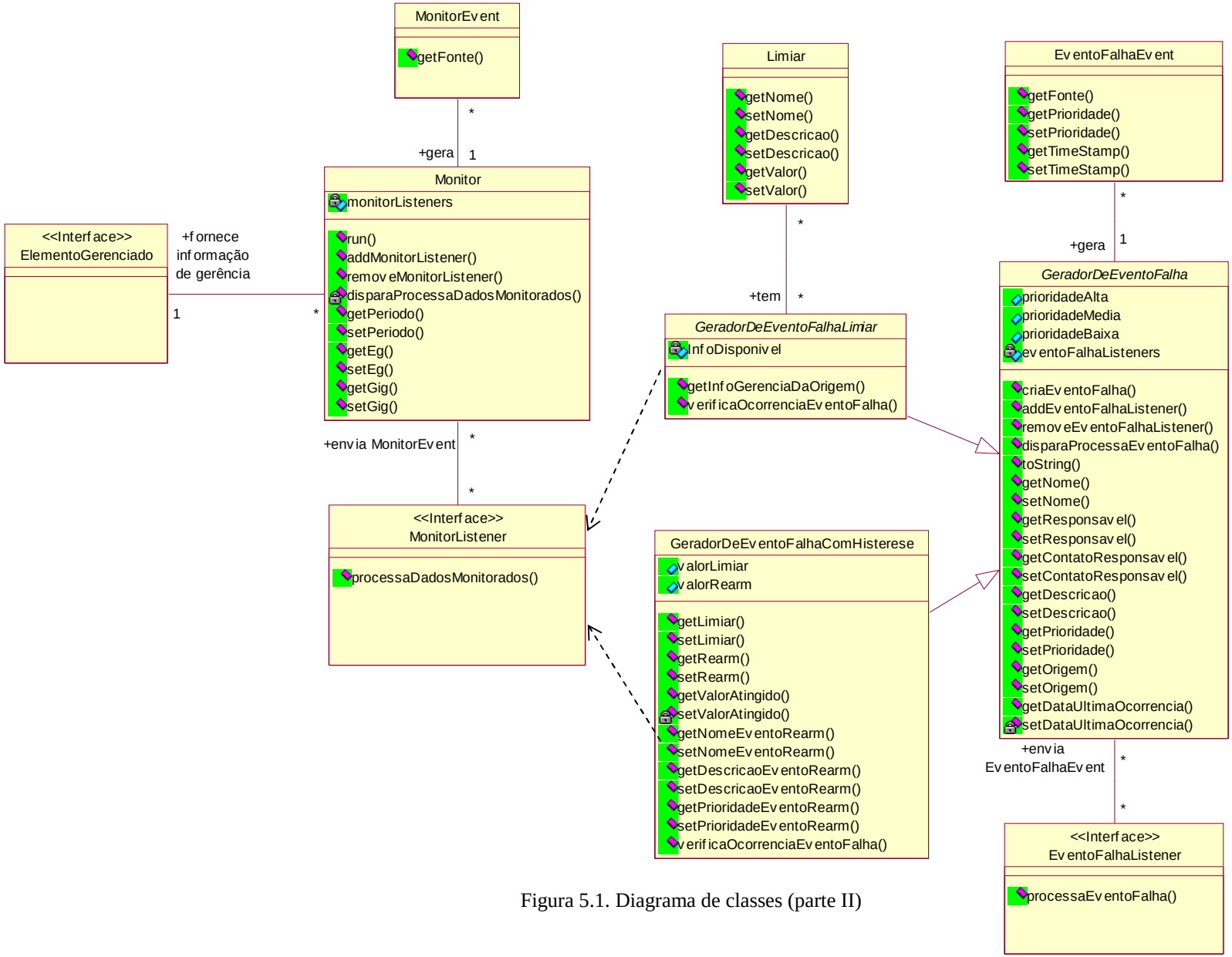


Figura 5.1. Diagrama de classes (parte II)

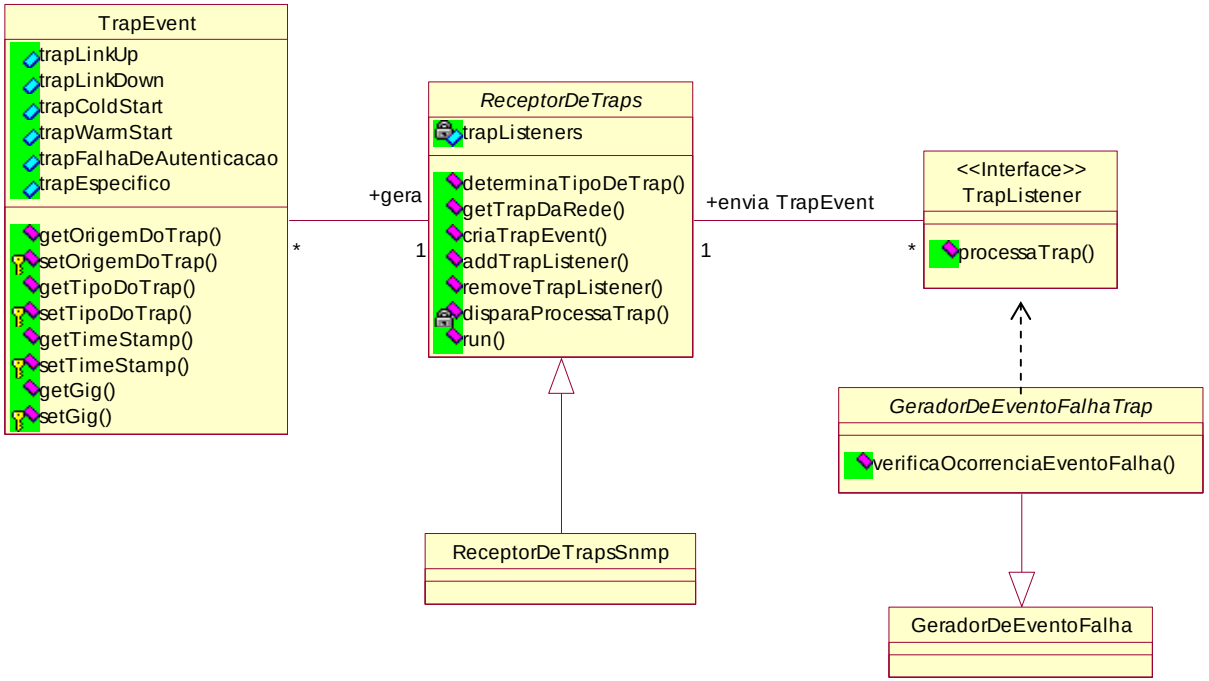


Figura 5.1. Diagrama de classes (parte III)

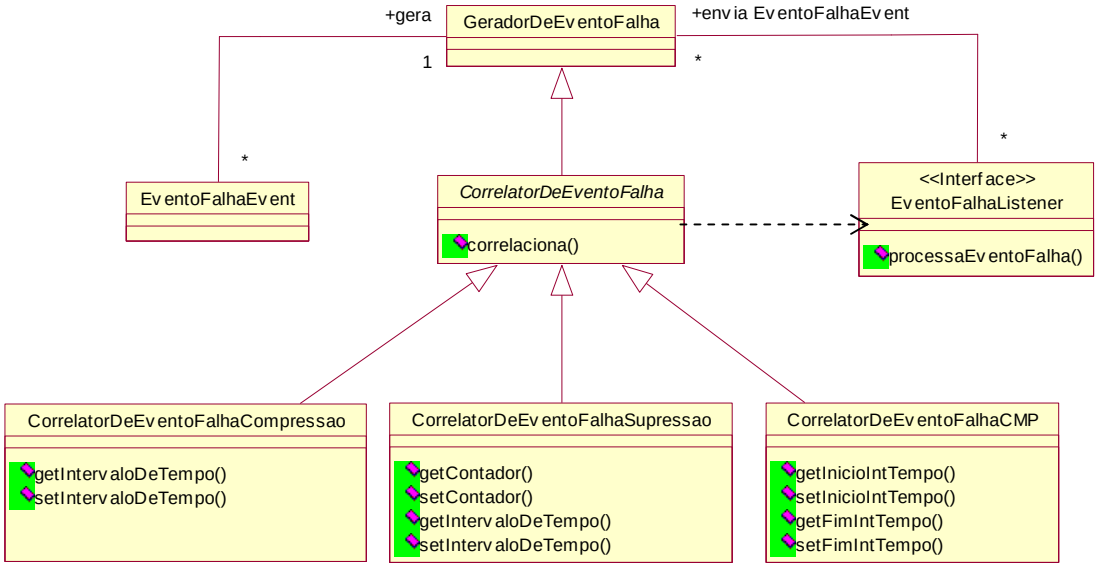


Figura 5.1. Diagrama de classes (parte IV)

Capítulo 6

Conclusões

Neste trabalho, especificamos uma solução cujo principal objetivo é facilitar o desenvolvimento de aplicações de gerência de falhas em redes de computadores. Trata-se de um *framework* baseado em componentes de software reutilizáveis - um *framework* CO - que busca fornecer um nível de abstração mais elevado para a realização das tarefas de gerência de falhas.

Vimos que outras APIs e linguagens disponíveis para o desenvolvimento de aplicações de gerência exigem considerável experiência de programação por parte do programador de aplicações e que, conseqüentemente, não oferecem abstrações de mais alto nível para realizar as tarefas associadas à gerência de redes de uma maneira mais simples e mais rápida.

Assim, ao levantar os requisitos que caracterizam um nível de abstração mais adequado, fomos capazes de especificar o projeto de uma solução que visa fornecer mecanismos mais elaborados para identificação e diagnóstico de falhas. Diante dos requisitos levantados, propomos um *framework* CO que fornece não só reutilização de código, mas também reutilização de análise e de projeto, e que, além disso, permite a construção *visual* de aplicações.

Um *framework* CO torna possível a reutilização de análise ao descrever os componentes importantes que estão diretamente relacionados ao domínio do problema e as relações entre estes componentes; o projeto é reutilizado à medida em que a arquitetura do próprio *framework* captura as principais decisões de projeto associadas ao domínio do problema - ele contém algoritmos abstratos e define as interfaces e restrições que uma implementação deve satisfazer; o código pode ser reutilizado uma vez que um novo componente criado pelo usuário pode “herdar” toda a implementação do componente

semipronto do qual ele é derivado; e os componentes disponíveis podem ser graficamente manipulados através de uma ferramenta de composição visual de aplicações. Com tudo isso, o programador pode se concentrar nas particularidades de sua aplicação enquanto conta com o alto grau de reusabilidade fornecido. Ao especificar a solução, identificamos os principais componentes do *framework* e definimos a forma de interconectá-los. Definimos, também, como o programador pode construir novos componentes para modificar e/ou estender o comportamento do *framework.*

Enfim, ao propor um *framework* CO para o desenvolvimento de aplicações de gerência de falhas, vimos que é possível elevar o nível de abstração fornecido, facilitando o desenvolvimento deste tipo de aplicação. A arquitetura proposta e as abstrações especificadas atendem aos requisitos listados, de modo que, arquiteturalmente falando, alcançamos os objetivos propostos de forma satisfatória.

6.1. Trabalhos Futuros

A conclusão deste trabalho nos permite identificar alguns encaminhamentos futuros no sentido de aperfeiçoar e estender a especificação de um *framework* baseado em componentes de software reutilizáveis para aplicações de gerência de falhas. Outros trabalhos ainda são necessários de modo a incluir outras características na solução proposta, inclusive para melhorar o atendimento a alguns requisitos levantados.

Em primeiro lugar, é preciso especificar novos componentes para o gerenciamento de alarmes e de históricos (*logs*), de modo a permitir um melhor tratamento das falhas identificadas. A especificação atual não detalha nenhum componente que possa ser diretamente utilizado pelo programador para estes fins. A integração com sistemas de *trouble-ticketing* também deve ser possibilitada para que o requisito F8, identificado na seção 4.1, seja devidamente atendido.

Em seguida, para permitir utilizar algoritmos de correlação de eventos mais elaborados, é preciso incluir informação sobre a topologia da rede no *framework.* De fato, muitos algoritmos se baseiam nos relacionamentos existentes entre as entidades gerenciadas da rede para identificar e diagnosticar falhas. O *framework* possui alguns componentes que implementam algoritmos mais simples mas isto pode não ser suficiente. Deve ser investigada, inclusive, a possibilidade de integração entre o *framework* e outras ferramentas de

descobrimento automático de topologia capazes de fornecer a informação que o *framework* precisa sobre a configuração física e lógica da rede.

Em termos de configurabilidade, outros pontos podem ser destacados. A saber:

Toda a interface gráfica de utilização do *framework* deve ser definida. Isto inclui a especificação e a implementação de uma ferramenta visual para manipulação de componentes conforme discutido na seção 5.2, bem como a especificação e a implementação de uma interface gráfica através da qual a atividade resultante da execução do *framework* possa ser acompanhada (um mapa onde se pode visualizar o estado da rede, por exemplo). Neste último caso, deve-se investigar a possibilidade de integração entre o *framework* proposto neste trabalho e outros *frameworks* especialmente desenvolvidos para a implementação de interfaces gráficas. Um exemplo deste último tipo de *framework* pode ser encontrado em [ILOG, 1999].

Apenas as classes que implementam os serviços oferecidos pelos componentes foram especificadas. Falta definir as classes relacionadas à configuração de cada componente e que, portanto, determinam como cada componente deve ser manipulado por uma ferramenta visual. Isto consiste em especificar os editores de propriedades especiais que cada componente requer e as classes que contêm informação adicional sobre as características de cada componente, como as propriedades e métodos que ele suporta, referências a editores de propriedades especiais, arquivos da dados que ele utiliza, etc. No caso particular da linguagem Java, trata-se de utilizar a API *JavaBeans* [Sun Microsystems, 1999] para definir como cada componente deve ser configurado.

Devem ser investigadas formas alternativas de melhorar a configurabilidade da aplicação a partir dos componentes disponíveis, em atendimento ao requisito F9. Para isto, seria necessária a definição de **módulos de política de gerência** que agrupassem os componentes necessários para a realização de determinada tarefa, instanciando-os e interconectando-os de forma automática, em tempo de construção visual da aplicação. O programador instanciaria um destes módulos em vez de componentes isolados. Por exemplo, seria extremamente útil que houvesse um módulo de configuração para identificação de falhas na rede. Isto poderia

incluir a instanciação automática de um componente do tipo `ElementoGerenciado`, um `Monitor`, um `GeradorDeEventoFalha` e um `EventoFalhaListener`, já devidamente interconectados. O programador configuraria o módulo como um todo, reutilizando toda a funcionalidade resultante da combinação dos componentes pertencentes ao módulo de configuração.

No caso do terceiro item supracitado, a questão vai além de melhorar a configurabilidade da aplicação e consiste em evoluir um pouco mais em termos de processo de desenvolvimento para que se possa identificar as formas adequadas de agrupar os componentes até agora disponíveis. Ao fornecer "supercomponentes" que naturalmente provêem um grau de reusabilidade ainda maior que aquele fornecido por componentes isolados, o programador pode construir sua aplicação sem precisar saber como componentes individuais são interconectados e sem precisar saber como eles realizam suas tarefas específicas. Isto eleva ainda mais o nível de abstração fornecido (em [Johnson & Roberts, 1998], um *framework* que fornece este nível de abstração é chamado um *black-box framework*), tornando a utilização do *framework* ainda mais simples.

Também deve-se investigar melhor a independência do *framework* proposto com relação aos diferentes padrões de gerência utilizados, principalmente no que diz respeito ao modelo de dados empregado por cada padrão. O padrão de gerência OSI, por exemplo, utiliza um modelo de dados orientado a objetos enquanto o padrão de gerência Internet utiliza um modelo de dados baseado em variáveis escalares. Deve-se averiguar como acomodar melhor as características próprias de cada padrão, haja vista que este trabalho está fundamentalmente voltado para o padrão de gerência Internet.

Por fim, a solução proposta deve ser implementada e também deve ser estendida para permitir realizar uma gerência de redes distribuída. Isto tornará a solução mais escalável, adequando-a às grandes redes de computadores atualmente utilizadas.

Bibliografia

[Beck & Gamma, 1998] Beck, K., Gamma, E., *Test Infected: Programmers Love Writing Tests*. 1998, *On-line*: http://www.dsc.ufpb.br/~jacques/cursos/1999.1/apoo/material/testing/junit.htm.

[Case *et al*., 1990] Case, J. D., Fedor, M. S., Schoffstall, M. L., Davin, J. R., *A Simple Network Management Protoco*l. Request for Comments 1157, DDN Network Information Center, SRI International, May 1990.

[Case *et al.*, 1993] Case, J. D., McCloghrie, K., Rose, M. T., Waldbusser, S., *Introduction to version 2 of the Internet-standard Network Management Framework.* Request for Comments 1441, SNMP Research Inc., Hughes LAN Systems Inc., Dover Beach Consulting Inc., Carnegie Mellon University, April 1993.

[Case *et al*., 1998] Case, J. F., Mundy, R., Partain, D., Stewart, B., *Introduction to Version 3 of the Internet-standard Network Management Framework*, Internet-draft, June 1998.

[DMTF, 1998] DMTF - Distributed Management Task Force, *DMTF Desktop Management Interface Specification*. Version 2.0, April 1998.

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

[Duarte, 1997] Duarte Jr, E. P., *SNMP-based Fault-Tolerant Network Monitoring*. Depto. de Informática, Universidade Federal do Paraná, Relatório Técnico, May 1997.

[Ezhilchelvan, 1994] Ezhilchelvan, P. D., *Dependability of Computer Systems*. Department of Computing Science, University of Newcastle upon Tyne, England, 1994.

[Flanagan, 1997] Flanagan, D., *Java in a Nutshell – A Desktop Quick Reference*. Second Edition, O'Reilly & Associates, May 1997.

[Fowler & Scott, 1998] Fowler, M., Scott, K., *UML Distilled – Applying the Standard Object Modeling Language*. Addison-Wesley, 1998.

[Gamma *et al.*, 1994] Gamma, E., Helm, R., Johnson, R., Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[Gibson *et al*., 1996] Gibson, J., Terplan, K., Huntington-Lee, *HP Open View – A Manager's Guide*. McGraw-Hill, 1996.

[Gosselin, 1999] Gosselin, C., *A Tool made in Perl using the HP OpenView SNMPv1 API*. *On-line*: http://www.info.uqam.ca/~gosselin/programm.htm.

[Govoni, 1999] Govoni, J., *Java Application Frameworks*. John Wiley & Sons, 1999.

[Hewlett Packard, 1998a] Hewlett Packard Company. *SNMP Developer's Guide*. Edition 1, November 1998.

[Hewlett Packard, 1998b] Hewlett Packard Company. *HP OpenView Windows Developer's Guide*. Edition 1, November 1998.

[Houck *et al.*, 1995] Houck, K., Calo, S., Finkel, A., *Towards a Practical Alarm Correlation System*. In IFIP/IEEE International Symposium on Integrated Network Management, IV (ISINM'95) [1995], pp 226-237.

[ILOG, 1999] ILOG Corporate Information Company. *ILOG TGF - Powerful Framework for Network and Service Management User Interfaces*. White Paper, 1999. *On-line*: http://www. ilog.com/products/tgf.

[ISO/IEC 7498:1984] Information Processing Systems - *Open Systems Interconnection: Basic Reference Model*. International Organization for Standardization and International Electrotechnical Committee, International Standard 7498, 1984.

[ISO/IEC 8824:1987] Information Processing Systems - *Open Systems Interconnection. Specification of Abstract Syntax Notation One (ASN.1)*. International Organization for Standardization, International Standard 8824, December 1987.

[ISO/IEC 8825:1987] Information Processing Systems - *Open Systems Interconnection. Specification of Basic Encoding Rules for Abstract Notation One (ASN.1)*. International Organization for Standardization, International Standard 8825, December 1987.

[Ierusalimschy *et al*., 1996] Ierusalimschy, R., Figueiredo, L., Celes, W., *Lua – An Extensible Language*. Software: Practice and Experience, 26(6), 1996.

[Jakobson & Weissman, 1993] Jakobson, G., Weissman, M. D., *Alarm Correlation*. IEEE Network, 7(6):52-59, November 1993.

[Johnson & Roberts, 1998] Johnson, R. E., Roberts, D., *Evolving Frameworks – A Pattern Language for Developing Object-Oriented Frameworks*. University of Illinois, 1998.

[Johnson & Foote, 1991] Johnson, R. E., Foote, B., *Designing Reusable Classes*. Journal of Object-Oriented Programming, June/July 1991.

[Johnson & Russo, 1991] Johnson, R. E., Russo, V. F., *Reusing Object-Oriented Designs*. University of Illinois, 1991.

[Lajoie & Keller, 1994] Lajoie, R., Keller, R. K., *Design and Reuse in Object-Oriented Frameworks – Patterns, Contracts and Motifs in Concert.* Proceedings of the 62nd Congress of the Association Canadienne Francaise pour l'Avancement des Sciences, Montreal, Canada, May 1994.

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

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

[Leinen, 1999] Leinen, S., *SNMP Support for Perl 5*. 1999. *On-line*: http://www.switch.ch/misc/leinen/snmp/perl/index.html.

[Lima *et al*., 1998] Lima, M. E., Moura, A. L., Ishikawa, E., Rodriguez, N., *Aplicações de Gerência Extensíveis*. XVI Simpósio Brasileiro de Redes de Computadores – SBRC. pp 125-143, Rio de Janeiro, Brasil, Maio 1998.

[Lima, 1998] Lima, M. E., *LuaMan: Uma Plataforma para Desenvolvimento de Aplicações de Gerenciamento Extensíveis*. Dissertação de Mestrado, Depto. de Informática, PUC-Rio de Janeiro, Janeiro 1998.

[Madruga & Tarouco, 1994] Madruga, E. L., Tarouco, L. M. R., *Fault Management Tools for a Cooperative and Descentralized Network Operations Environment*. IEEE Journal on Selected Areas in Communications, Vol. 12, No. 6, pp. 1121-1130, August 1994.

[Meira & Nogueira, 1997] Meira, D. M., Nogueira, J. M. S., *Métodos e Algoritmos para Correlação de Alarmes em Redes de Telecomunicações*. Simpósio Brasileiro de Redes de Computadores, 1997, pp. 79-98, São Carlos, SP, Brasil, Maio 1997.

[Meira, 1997] Meira, D. M., *Um Sistema para Correlação de Alarmes em Redes de Telecomunicações*. Tese de Doutorado, Depto. de Informática, Universidade Federal de Minas Gerais, 1997.

[Meira & Lages, 1998] Meira, D. M., Lages, N. A. C., *SIS: Um Sistema Integrado de Supervisão para Telecomunicações*. Relatório Técnico TELEMIG/UFMG/88, Depto. de Informática, Universidade Federal de Minas Gerais, Belo Horizonte, Brasil, Setembro 1988.

[Mellquist, 1997] Mellquist, P. E., *SNMP++: An Object-Oriented Approach to Developing Network Management Applications*. Prentice-Hall, 1997

[Oates, 1995] Oates, T., *Fault Identification in Computer Networks: A Review and a New Approach*. Computer Science Technical Report 95-113, University of Massachusetts, 1995.

[Raman & Raman, 1999] Raman, L. G., Raman, L., *Fundamentals of Telecommunications Network Management*. IEEE Series on Network Management, IEEE, March 1999.

[Rational, 1999] Rational Company, *A Rational Approach to Software Development using Rational Rose 4.0*. White Paper, 1999. *On-line*: http://www.rational.com/products/rose/prodinfo/whitepapers.

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

[Rose, 1990] Rose, M. T., *The Simple Book: An Introduction to Management of TCP/IP-based Networks*. Prentice-Hall, 1990.

[Rose & McCloghrie, 1990a] Rose, M. T., McCloghrie, K., *Management Information Base Network Management of TCP/IP based Internets*. Request for Comments 1156, DDN Network Information Center, SRI International, May 1990.

[Rose & McCloghrie, 1990b] Rose, M. T., McCloghrie, K., *Structure and Identification of Management Information for TCP/IP based Internets*. Request for Comments 1155, DDN Network Information Center, SRI International, May 1990.

[Rose & McCloghrie, 1991] Rose, M. T., McCloghrie, K., *Management Information Base for Network Management of TCP/IP-based Internets: MIB-II*. Request for Comments: 1213, Hughes LAN Systems, Inc., Performance Systems International, March 1991.

[Rose & McCloghrie, 1995] Rose, M. T., McCloghrie, K., *How to Manage Your Network Using SNMP: The Network Management Practicum*, Prentice-Hall, 1995.

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

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

[Sauvé, 1999] Sauvé, J. P., *WebManager: A Web-Based Network Management Application*. LANOMS - Latin American Network Operations and Management Symposium, Rio de Janeiro, Brasil, Dezembro 1999.

[Schönwälder & Langendörfer, 1996] Schönwälder, J., Langendörfer, H., *How to Keep Track of Your Network Configuration*. Technical University of Braunschweig, Germany, November 1993.

[Schönwälder & Langendörfer, 1995] Schönwälder, J., Langendörfer, H., *Tcl Extensions for Network Management*. In Proceedings 3rs Tcl/Tk Workshop, Toronto, Canada, 1995.

[Schönwälder, 1997] Schönwälder, J., *Scotty - Tcl Extensions for Network Management Applications*, 1997. *On-line*: http://wwwsnmp.cs.utwente.nl/~schoenw/scotty/.

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

[Simões *et al.*, 1994] Simões, P. A. F., Brites, A. C. S. C., Leitão, P. M. C. , Monteiro, E. H. S., Fernandes, F. P. L. B., *A High-Level Notation for the Specification of Network Management Applications*. University of Coimbra, Portugal, May 1994.

[Sloman, 1994] Sloman, M., *Network and Distributed Systems Management*. Addison-Wesley, 1994.

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

[Tanenbaum, 1998] Tanenbaum, A. S., *Modern Operating Systems*. Prentice-Hall. 1998.

Apêndice A

Frameworks

> “ O foco é a reutilização de análise e projeto, e não a reutilização de código.”
>
> [Gamma *et al.*, 1994]

A reusabilidade de software é reconhecida como uma importante maneira de aumentar a produtividade no processo de desenvolvimento de software. A idéia é não desenvolver nada do que já existe, mas apenas reutilizar. Isto diminui o tempo de desenvolvimento e permite a construção de produtos de software mais robustos, o que é primordialmente necessário já que softwares cada vez mais complexos precisam ser desenvolvidos num espaço de tempo cada vez menor.

Produzir software reutilizável implica em desenvolver unidades de software genéricas e extensíveis. Projetistas devem prever a utilização futura do software a ser produzido e devem incluir os requisitos previstos no projeto corrente. Tradicionalmente, isto tem sido feito através de bibliotecas de funções procedurais, específicas de um determinado domínio de problema, ou através de bibliotecas de classes reutilizáveis.

O problema é que se torna difícil prover um comportamento padrão e introduzir conhecimento sobre o domínio do problema numa biblioteca [Landin & Niklasson, 1998]. Uma biblioteca como estas contém pequenos módulos funcionais a partir dos quais o programador deve construir uma determinada aplicação e desenvolver uma aplicação com base em pequenos módulos exige que a comunicação entre os diferentes módulos utilizados ainda seja definida.

Estes “problemas” podem ser aliviados através da utilização de *frameworks*. O programador não precisa saber como ou quando “chamar” cada função; o *framework* faz isto para ele. Diz-se, portanto, que *frameworks* se baseiam no *Hollywood Principle* - *“Don’t call us, we’ll call you”*. A comunicação entre os módulos funcionais do *framework* já está

internamente definida e o programador não precisa se preocupar com isto [Lajoie & Keller, 1994].

Geralmente, um *framework* consiste de um conjunto de classes e poderia, então, ser considerado uma biblioteca de classes. Isto não é inteiramente verdade, desde que numa biblioteca de classes cada classe é vista individualmente e a maioria das classes num *framework* são dependentes umas das outras, não tendo utilidade fora deste contexto. Assim, *frameworks* provêem reusabilidade num nível mais alto de abstração a partir da funcionalidade resultante da combinação (interação) de suas classes, fornecendo além de reutilização de código, reutilização de análise e de projeto. Aplicações podem ser desenvolvidas ao utilizar o *framework* como ponto de partida, sendo possível (e necessário) escrever pequenos trechos de código para modificar ou estender o comportamento do *framework*.

A.1. O que é um *Framework*?

Atualmente, o conceito de *framework* está intimamente ligado à orientação a objetos. *Frameworks* são implementados através de linguagens orientadas a objetos principalmente porque elas suportam os conceitos de herança, polimorfismo e *binding* dinâmico. Os detalhes associados a cada um destes conceitos não serão descritos aqui, mas são estas as características que viabilizam o desenvolvimento de *frameworks*.

Frameworks OO são definidos de duas maneiras similares em [Johnson & Foote, 1991] e [Johnson & Russo, 1991]: “Um *framework* é um conjunto de classes que captura um projeto abstrato para soluções de uma família de problemas relacionados.” e “Um *framework* é um conjunto de objetos que colaboram entre si para assumir um conjunto de responsabilidades de aplicações pertencentes a um domínio de problema específico.”.

Com base nas definições acima, pode-se concluir que um *framework* determina a arquitetura das aplicações construídas com base nele. As decisões de projeto comuns às aplicações pertencentes a determinado domínio de problema são capturadas de forma que o programador possa se concentrar nas particularidades de sua aplicação. O domínio de problema precisa, então, ser bem delimitado e bem caracterizado para que as generalidades possam ser devidamente capturadas. Quando se utiliza um *framework* bem projetado e bem documentado, conseqüentemente, análise, projeto e código podem ser reutilizados.

Para configurar o *framework*, o programador pode estender classes abstratas, redefinindo os métodos necessários. O *framework* se encarrega de "chamar" os novos métodos definidos, de modo que, ao contrário de bibliotecas de classes, onde a aplicação faz referência às classes disponíveis, é o *framework* quem "chama" o código escrito pelo programador. Veja a figura A.1 [Landin & Niklasson, 1998].

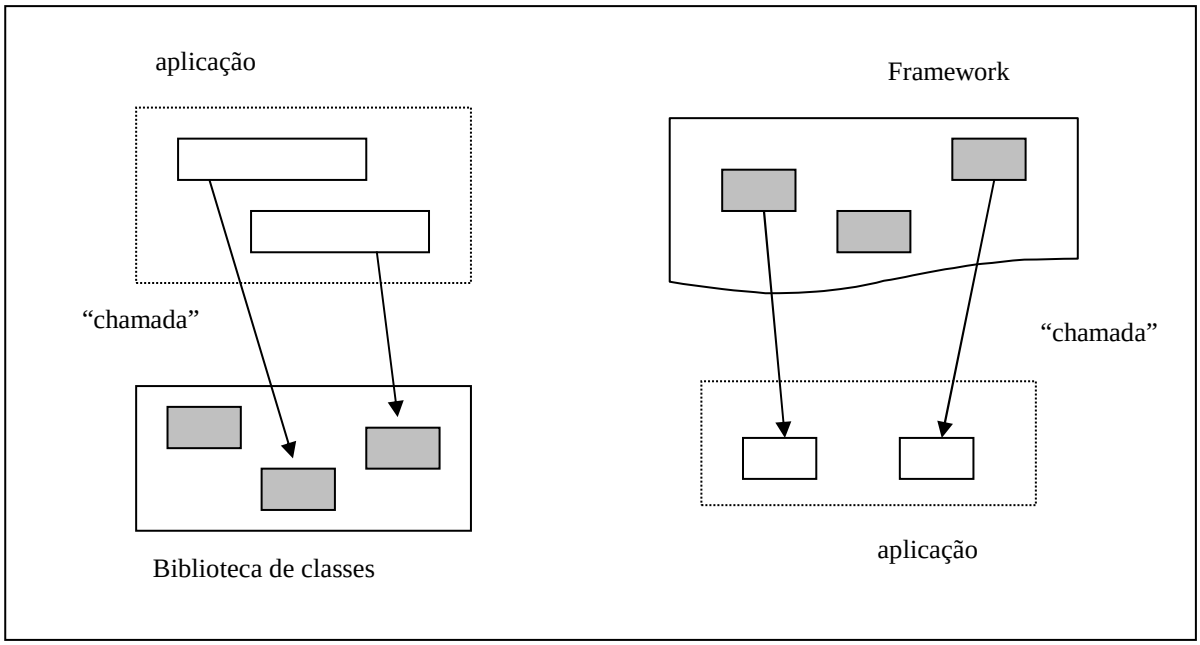


Figura A.1. Diferença no fluxo de controle entre *frameworks* e bibliotecas de classes

Com a utilização de *frameworks*, aproveitam-se não só as decisões de projeto normalmente tomadas por especialistas no domínio de problema para o qual o *framework* foi desenvolvido, mas também reduzem-se os custos com manutenção e testes. Quando várias aplicações são desenvolvidas com base no mesmo *framework*, apenas o *framework* em si e o código que é diferente para as diversas aplicações deve ser mantido. Isto também significa que mudanças têm que ser feitas num único lugar, garantindo consistência. Além disso, todos os testes feitos no *framework* são reutilizados pelas diversas aplicações que o utilizam.

A maximização da reusabilidade tem seu ônus. De fato, desenvolver um *framework* é mais difícil do que construir aplicações isoladas ou módulos de bibliotecas, já que a arquitetura do *framework* tem que ser definida, assim como a interação entre os módulos internos [Johnson & Foote, 1991]. Apesar disso, muitos *frameworks* têm sido desenvolvidos, pois na relação custo-benefício há uma proporcionalidade que, em geral, observa-se e que no

final das contas, compensa. Um exemplo de *framework* pode ser encontrado em [Beck & Gamma, 1998]. Trata-se de um *framework* para teste de unidade em Java.

Apêndice B

Componentes de Software Reutilizáveis

> "Se você busca melhoramentos consideráveis em termos de produtividade de software, uma das coisas mais importantes a serem feitas é parar de escrever aplicações do princípio toda vez que você inicia um novo projeto. Em vez disso, você deve construí-las usando componentes de software que já existem".
>
> [D'Souza & Wills, 1999]

Há muito tempo que a idéia de maximizar a reutilização de coisas prontas é utilizada. 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 ter muitas possibilidades de configuração e os produtos finais podem ser fabricados de forma mais rápida e mais confiável.

Nos últimos anos, a mesma coisa tem acontecido na produção de software e o que se observa é que muitas aplicações são, agora, construídas com base em *frameworks* diversos ou resultam da fusão de aplicações existentes. É possível, por exemplo, utilizar "software de prateleira" como um pacote de calendário, um processador de textos e uma planilha, e reunir esses componentes heterogêneos usando *scripts*, de modo a resolver um problema particular de determinado domínio de problema.

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. Padrões como CORBA (*Common Object Request Broker Architecture*) [Siegel, 1996] e COM (*Component Object Model*) [Rofail & Shohoud, 1999], permitem conectar componentes construídos em diferentes linguagens e plataformas, e APIs como *JavaBeans* [Sun Microsystems, 1999] permitem desenvolver componentes de acordo com as características de determinado tipo de

linguagem de programação. Ferramentas gráficas ajudam o programador a “compor” aplicações ao permitir a configuração e conexão dos componentes necessários à aplicação de forma *visual*.

Neste apêndice, são destacadas algumas definições e as características genéricas de um componente de software. Em particular, é apresentado como a linguagem Java pode ser utilizada para desenvolver componentes.

B.1. Características Gerais

Há muitas definições para um “componente”. Segundo [D’Souza & Wills, 1999], um componente “é um pacote coeso de artefatos de software e uma unidade de entrega (*delivery unit*) que pode ser desenvolvido de forma independente e que pode ser conectado a outros componentes para construir algo maior”. De acordo com esta definição, são exemplos de componentes:

usar um *framework* de alocação de recursos para modelar problemas que variam da alocação de salas para seminários até o escalonamento de tempo de máquina para a produção de lotes de peças numa fábrica;

usar construções pré-definidas de uma linguagem de programação numa infinidade de contextos:

```
for(...; ...; ...) {
    ...
}
```

usar um *framework* de classes e interfaces para construir componentes que serão utilizados para construir diversas aplicações de gerência de falhas.

Um componente contém código executável, código fonte, especificações, testes, documentação e tudo o mais que um pacote de *software* pode conter. Em termos de implementação e ainda segundo [D’Souza & Wills, 1999], componentes (a) têm interfaces explícitas e bem definidas para os serviços que oferece, (b) têm interfaces explícitas e bem definidas para os serviços que requer e (c) podem ser conectados a outros componentes, talvez após a configuração de algumas propriedades, sem modificar os componentes em si. Um componente consiste tipicamente de várias classes (código binário), definições de interfaces e outros recursos que são usados para a sua configuração (arquivos de dados contendo formulários, parâmetros, imagens, etc.).

A utilização de componentes deve permitir que todo ou quase todo o trabalho seja feito pela composição de pedaços existentes. Isso significa que, no processo de desenvolvimento de software, novas etapas podem surgir, como é o caso da **etapa de composição de aplicações**, que consiste em juntar componentes para obter a funcionalidade de cada aplicação. A composição pode ser feita por "terceiros" que nem ao menos têm acesso a detalhes de implementação do componente. A configuração de componentes é feita através da modificação de seus atributos (*Attribute Programming*) e, para isso, ferramentas de composição de aplicações, ou simplesmente ferramentas visuais, como *Delphi* ou *Visual Basic*, são freqüentemente utilizadas (*Visual Programming*).

Sendo assim, o uso de componentes traz mudanças no processo de desenvolvimento, exigindo uma nova postura do programador de aplicações que simplesmente terá que selecionar componentes e configurá-los conforme exige determinada situação. A composição da aplicação é o foco, e não sua implementação. A implementação, por sua vez, está embutida nos componentes utilizados.

B.2. Construindo Componentes em Java

Quem já usou *Delphi* ou *Visual Basic* para desenvolver aplicações, já é familiar com a noção de um *bean*. A idéia é a mesma; a linguagem de programação é diferente. Um *bean* é um componente de software escrito em Java. Mais especificamente, um *bean* é um componente de software reutilizável que pode ser visualmente manipulado através de uma ferramenta gráfica. A API utilizada para a construção de componentes de software em Java chama-se *JavaBeans* (`java.beans`).

Um *bean* pode ser composto de várias classes cujas interfaces externas são descritas através de suas **propriedades**, de seus **métodos** e dos **eventos** gerados.

* Propriedades são atributos de objetos que podem ser lidos e atualizados através de métodos de acesso. Elas podem ser modificadas em tempo de composição e representam os atributos de estado e de comportamento através dos quais *beans* podem ser configurados.

* Métodos são os serviços que o cliente pode utilizar.

* Eventos representam mudanças de estado que são exteriorizadas para o ambiente onde o *bean* está inserido. É através da troca de eventos que os *beans* se comunicam uns com os outros.

Propriedades e métodos representam os serviços oferecidos pelo *bean*. Eventos representam notificações geradas diante de uma mudança de estado. Não há nenhuma maneira de representar explicitamente os serviços usados pelo componente (aliás, as linguagens têm se concentrado apenas em especificar os serviços oferecidos por componentes e não os serviços usados).

Tipicamente, uma ferramenta visual para a manipulação de *beans* contém o conjunto de *beans* que o programador da aplicação pode instanciar e configurar. Ele deve "arrastar" o *bean* para uma área central e configurá-lo através de um editor de propriedades. A figura B.1 mostra uma ferramenta chamada *BeanBox*.

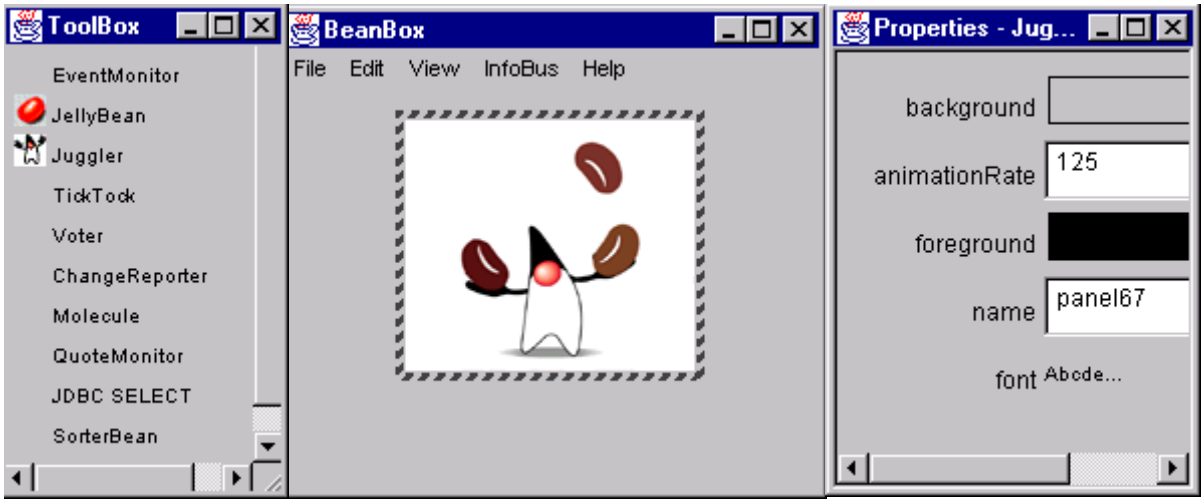


Figura B.1. *BeanBox*

Ferramentas visuais como o *BeanBox*, descobrem as propriedades e métodos de um *bean* e os eventos por ele tratados através de um mecanismo conhecido como **introspecção**. Isto é feito 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, algumas vezes conhecidas como "padrões de projeto *JavaBeans*" [Flanagan, 1998]. Então, para cada *bean* a ser escrito, as regras básicas abaixo devem ser observadas.

1. Propriedade - deve-se escrever um par de métodos de acesso `Y get<X>` e `set<X>(Y)` para definir uma propriedade com nome `X` e tipo `Y`.
2. Evento - para cada evento `E` que um *bean* deve enviar para os demais componentes cadastrados, deve-se escrever um par de métodos `add<E>Listener(Elistener)` e `remove<E>Listener(Elistener)` e definir as assinaturas dos métodos que a interface `EListener` deve suportar.
3. Método - qualquer método com qualquer nome pode ser definido.

Também é possível implementar *beans* sem seguir regras de nomeação, só que, neste caso, o *bean* deve fornecer, explicitamente, informação sobre suas propriedades, métodos e eventos através da interface `BeanInfo` (`java.beans`). Detalhes não serão discutidos aqui.

Uma vez que os *beans* sejam configurados, eles devem ser salvos, afinal este é o resultado da etapa de composição de aplicações. Desta maneira, *beans* são **persistentes** e Java provê uma forma muito simples de se ter isto. Um mecanismo automático de serialização através do qual determinada configuração de *beans* pode ser salva e recuperada em outro momento pode ser utilizado. Tudo que o *bean* deve fazer é implementar a interface `java.io.Serializable`. Normalmente, a ferramenta visual salva e restaura determinada configuração através de comandos de *menu*.

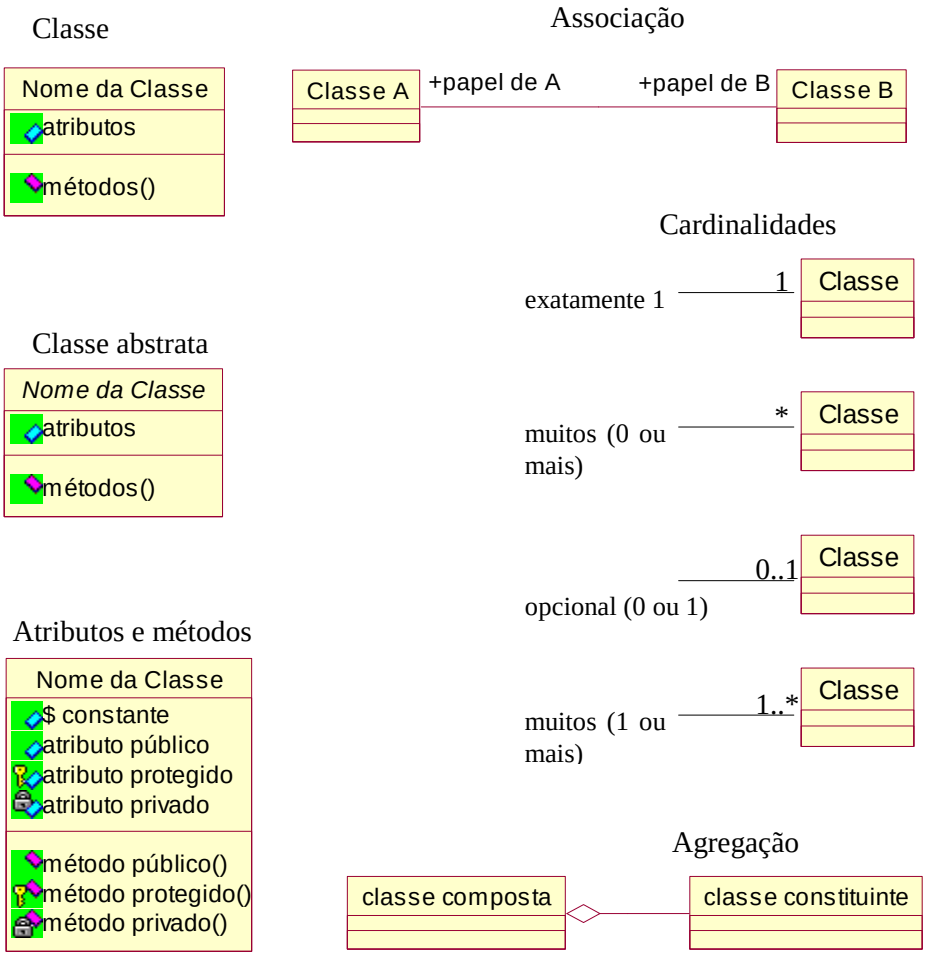
Componentes Java compilados são empacotados em arquivos do tipo JAR. Estes arquivos incluem as classes que implementam os serviços dos componentes, as classes adicionais que fornecem informação explícita sobre propriedades, métodos e eventos do *bean*, e qualquer informação adicional. Para que um *bean* 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.

Em resumo, *beans* podem ser configurados, são persistentes e podem ser visualmente manipulados. A linguagem Java suporta o conceito de componentes através da API *JavaBeans* e de outros mecanismos genéricos como reflexão e serialização. Programadores avançados constróem componentes com base no suporte fornecido e estes componentes são utilizados para compor aplicações através de um ambiente gráfico. A API *JavaBeans* faz parte do JDK1.1 e qualquer ferramenta compatível com ele suporta implicitamente os conceitos e características envolvidos.

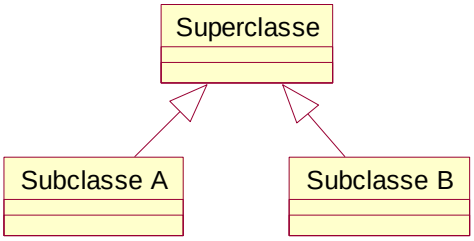
Apêndice C

A Notação UML

A ferramenta *Rational Rose* [Rational, 1999] foi utilizada para a construção dos gráficos utilizados neste trabalho. Trata-se de uma ferramenta *CASE* para especificação de modelos UML (*Unified Modeling Language*). A notação que segue, portanto, está associada aos recursos que a ferramenta oferece e difere ligeiramente da notação encontrada em algumas referências bibliográficas, como em [Rumbaugh *et al.*, 1999a] e [Fowler & Scott, 1999]. Apenas os símbolos utilizados são destacados.



Generalização



Interfaces

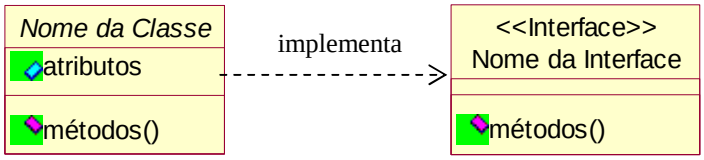
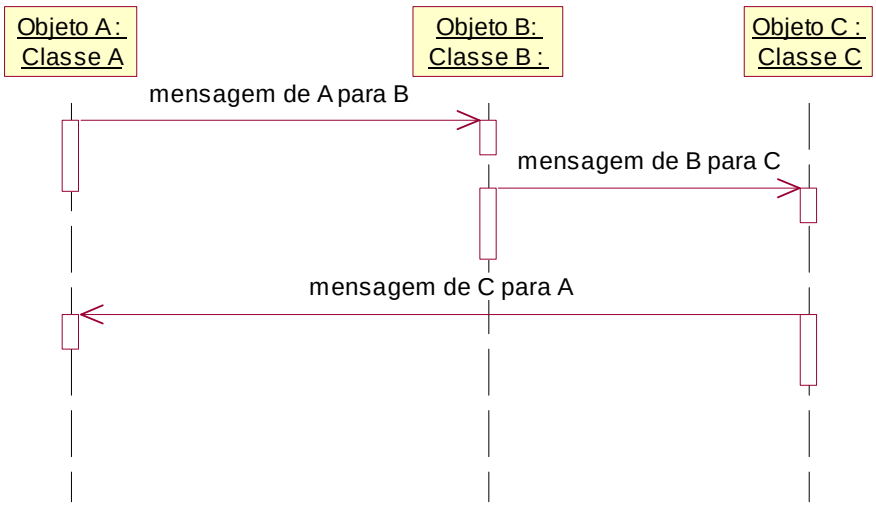


Diagrama de Seqüência



Glossário

API	Application Programming Interface
ASN.1	Abstract Syntax Notation One
BER	Basic Enconding Rules
CMP	Controle de Manutenção Programada
CO	Component-Oriented
CORBA	Common Object Request Broker Architecture
COM	Component Object Model
DMI	Desktop Management Interface
DMTF	Distributed Management Task Force
ECS	Event Correlation Services engine
IP	Internet Protocol
ISO	International Standards Organization
ITU-T	International Telecommunications Union
JMX	Java Management Extensions
JDK	Java Development Kit
MIB	Management Information Base
OO	Orientação a Objetos
OSI	Open System Interconnection
OVSNMP	OpenView SNMP
SMI	Structure of Management Information
SNMA	Specification of Network Management Applications
SNMP	Simple Network Management Protocol
SNMPv1	Simple Network Management Protocol version 1
SNMPv2	Simple Network Management Protocol version 2
SNMPv3	Simple Network Management Protocol version 3
Tcl	Tool Command Language
TCP	Transmission Control Protocol
TMN	Telecommunications Management Network

UML	Unified Modeling Language