Uma Abordagem sobre Atualização Dinâmica em Componentes de Sistemas Orientados a Objetos

Fábio Fagundes Silveira[1], Antônio Maria Pereira de Resende[1,2],
Adilson Marques da Cunha[1], Maria Lúcia Blanck Lisbôa[3]
[1]Divisão de Ciência da Computação - Instituto Tecnológico de Aeronáutica (ITA)
[2]Depto. de Ciência da Computação– Universidade Federal de Lavras (UFLA)
[3]Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
e-mail: [ffs, cunha]@comp.ita.br, tonio@comp.ufla.br, llisboa@inf.ufrgs.br

Resumo

A atualização dinâmica de sistemas é uma atividade crucial em software para missão crítica, que não pode parar a sua execução a qualquer momento. O presente artigo aborda aspectos sobre atualização de componentes em sistemas orientados a objetos, sem a necessidade de interrupções. Nele, discute-se os prós e contras de algumas possíveis soluções que utilizam técnicas existentes. Baseando-se em prospecções realizadas, uma abordagem é proposta, a fim de propiciar a criação de novas soluções ou a melhoria das existentes. Esta abordagem sugere a criação do Serviço de Troca de Referência - STR e do Serviço de Identificação de Processamento – SIP, os quais auxiliam a atualização dinâmica de componentes, eliminando a degradação dos sistemas atualizados causadas pelas técnicas existentes.

Palavras-chave: *Atualização Dinâmica de Componentes, Reflexão Computacional, Programação Orientada a Aspectos*

Abstract

Dynamic software updating is a crucial activity for critical mission software that must not stop its execution at anytime. This paper discusses aspects about components updating in object oriented software without needing to interrupt their executions. Advantages and disadvantages are discussed about techniques used to apply current solutions.Based upon prospections done, an approach is proposed in order to allow either develop new solutions or improve the current ones. This approach suggests an implementation of Reference Exchange Service and Running Identification Service. These services aid dynamic components updating, eliminating the runtime overhead of updated systems which is inherited from current techniques.

Keywords: *Dynamic Software Updating, Computational Reflection, Aspect Oriented Programming*

1. Introdução

Pelo fato de um software representar atividades humanas para operações computadorizadas, ele pode herdar muitas características de processos. Uma das características de maior impacto que o software pode herdar dos processos é a necessidade de manutenções corretivas, adaptativas, perfectivas e preventivas [1].

Durante a atividade de manutenção, um componente deve ser testado antes da entrega. A fase de testes não pode garantir a ausência de defeitos, ela só pode mostrar se defeitos de software estão presentes [2]. Por isso é quase impossível desenvolver um software livre de erros, considerando que ainda não existem técnicas capazes de oferecer tal garantia.

Devido a impossibilidade de prever necessidades futuras de clientes ou aplicações, pode-se considerar cada vez mais difícil a construção de softwares que não requeiram manutenções [3].

A atividade de manutenção normalmente apresenta custos elevados inapropriados nos processos de desenvolvimento de software, caso não sejam utilizadas técnicas apropriadas, ainda mais quando se trata de software para missões críticas, onde atualizações e manutenções representam grandes esforços e preocupações para os responsáveis.

A necessidade de funcionamento contínuo e ininterrupto representa uma das principais características de software críticos como aqueles presentes em Sistemas de Transações Financeiras, Centrais Telefônicas e de Controle de Tráfego, entre outros. A interrupção desses sistemas pode ocasionar grandes perdas para os usuários que se utilizam desses serviços [4].

Este artigo foca a evolução de um software pela atualização dinâmica de seus componentes, simulando um comportamento adaptativo durante o processo de atualização dos componentes defasados, sem se eliminar da memória a instância do software em execução. No decorrer do texto são descritas algumas abordagens encontradas na literatura para atualização dinâmica em componentes de software de missão crítica orientados a objetos, e a partir delas, propõe-se uma nova abordagem capaz de reduzir custos envolvidos na manutenção desse tipo de software.

O artigo está organizado da seguinte forma: a seção 2 caracteriza a atividade de atualização dinâmica de componentes, define o problema endereçado neste trabalho, bem como a solução proposta; a seção 3 apresenta os métodos em que as abordagens existentes estão baseadas, enquanto que a seção 4 demonstra algumas soluções tecnológicas atuais para sistemas legados; já a seção 5 descreve a abordagem proposta e apresenta o seu algoritmo em alto nível de abstração; finalmente, a seção 6 apresenta as conclusões deste artigo.

2. Atualização Dinâmica

Visando padronizar os termos utilizados, o presente trabalho adota a denominação de *componente defasado* para componente ou objeto que necessitar de atualização, e *novo componente* para o componente ou objeto que assumir o lugar do *componente defasado*, provendo as novas funcionalidades requeridas. Já o termo *componente-usuário* refere-se ao componente que se utilizar de serviços de outros componentes.

A partir do momento em que se deseja atualizar dinamicamente um componente de um software, faz-se necessário levar em conta, entre outros, os seguintes fatores:

- Um sistema de missão crítica não pode parar a qualquer momento;
- Manutenções indispensáveis existem para o correto funcionamento desses sistemas;
- Atualizações dos novos componentes podem ocorrer em regime de urgência ou previamente agendado. Em regime de urgência, o novo componente deve ser implantado o mais rápido possível, enquanto que em regime previamente agendado pode-se ter uma atualização eletiva;
- O momento necessário de se instalar um novo componente pode ser totalmente inconveniente para se realizar uma interrupção do sistema;
- Somente os serviços que envolvem o componente defasado podem ser interrompidos temporariamente, durante o processo de carga do novo componente; e
- As dependências existentes entre o software principal e o novo componente, com o intuito de anular qualquer incompatibilidade entre eles com relação às chamadas de serviços, devem ser verificadas no momento da instalação.

É importante ponderar que mesmo as aplicações de missões não críticas também se beneficiam das vantagens das atualizações dinâmicas, quando não se precisa parar suas execuções.

Segundo Hicks [4], as seguintes propriedades definem uma efetiva abordagem de atualizações dinâmicas:

a) **Flexibilidade** - qualquer parte de um sistema em execução deve ser passível de atualização sem que seja necessário realizar uma interrupção em todo o sistema;
b) **Robustez** - riscos de erros e paradas críticas, devido a uma atualização, devem ser minimizados de forma exaustiva, através do uso de mecanismos automáticos que promovam corretas atualizações;

c) **Facilidade de Uso** - espera-se que, quanto mais simples for o processo de atualização, menores serão suas probabilidades de insucesso. Em conseqüência disso, o sistema de atualização deverá ser de fácil utilização; e
d) **Baixo *Overhead*** – atualizações não devem ocasionar impactos no seu desempenho.

As propriedades de Hicks pode ser consideradas como requisitos não funcionais que devem nortear a implementação de uma solução eficiente para a atualização dinâmica.
A aplicação das técnicas existentes de atualização dinâmica em componentes, apesar de cumprirem seus papéis, ainda degradam o desempenho dos sistemas. Isto ocorre porque tais técnicas criam uma nova camada responsável por gerenciar todas as chamadas realizadas ao componente defasado, redirecionando-as para o novo componente. Esse serviço de redirecionamento deve funcionar até a primeira reinicialização do sistema, quando as novas referências são efetivamente atualizadas.
Outra consideração importante sobre os sistemas de missões críticas, consiste no fato da interrupção dos serviços ser inapropriada a qualquer momento, sendo praticável apenas quando o prejuízo ocasionado pela continuidade dos serviços se sobrepõe a necessidade.

O problema endereçado neste artigo consiste em propor uma nova sistemática para atualização dinâmica em componentes. A solução escolhida para resolver o problema acima, consiste em desenvolver uma abordagem sobre atualização dinâmica em componentes de sistemas orientados a objetos, visando aumentar a sua eficiência e reduzir a degradação do desempenho dos sistemas atualizados e o desperdício de recursos envolvidos.

3. Abordagens Existentes

Existem várias implementações possíveis para resolver o problema de atualização dinâmica e, geralmente, elas pertencem a um dos seguintes métodos: a) Vínculo Dinâmico (*Dynamic Linking*); b) Transferência Estática de Estados (*Static State Transfer*); ou c) Transferência Dinâmica de Estados (*Dynamic State Transfer*) [5].

a) **Vínculo Dinâmico** - Atualizações são realizadas trocando-se todas as referências do componente defasado para o novo componente. Desta forma, o componente defasado pode ser removido. Entretanto, tal solução não se encontra presente nas distribuições padrões das linguagens de programação. Uma alternativa específica para a linguagem Java, denominada Gilgul [6], caracteriza-se como uma extensão a esta linguagem, onde todos os ponteiros para um objeto podem ser atualizados através de uma operação de substituição, aumentando a velocidade do processo de atualização dinâmica. Por outro lado, esta extensão dá-se através da modificação da máquina virtual Java, comprometendo em parte o quesito portabilidade.
b) **Transferência Estática de Estados** - Este método consiste em transferir os estados persistentes do componente defasado para o novo componente. Os passos deste método são: salvar o estado do componente defasado; parar sua execução; substituir a versão antiga pela nova versão; e atualizar o estado da versão antiga na nova versão. Se o componente defasado possuir alto nível de criticidade e não puder ser interrompido depois de um estado salvo, o terceiro método (c) pode ser escolhido.
c) **Transferência Dinâmica de Estados**: Gupta [7] descreve uma função de transferência de estados, a qual transporta o estado de um componente em execução para sua nova versão. Hicks [4] introduz um sistema que automaticamente gera tal função embasada em duas diferentes implementações. Esta função é utilizada para atualizar parte do código. Tanto Gupta [7] quanto Hicks [4] enfatizam que o programador deve estar envolvido, informando onde o estado de transferência deverá aparecer. Isto deve-se ao fato de que componentes em execução não podem ser movidos do seu lugar pela função de transferência.

4. Soluções Tecnológicas Existentes

A essência das soluções existentes é prover um gerenciador de atualizações (GA) que capture as chamadas feitas ao componente defasado, colocando-as, por exemplo, numa fila. Após a instalação do novo componente, o gerenciador de atualizações repassa ao novo componente as requisições enviadas ao componente defasado, conforme apresenta a Figura 1.

Essas soluções, apesar de cumprirem o objetivo de atualizar dinamicamente um componente do sistema, geram uma degradação no desempenho, devido a necessidade de desvio das chamadas aos serviços do componente defasado. Esta degradação fere a quarta propriedade de Hicks, baixo *overhead* [4].

Esta degradação de desempenho perdura até o momento oportuno para a reinicialização do sistema, quando as referências aos novos componentes são corretamente restabelecidas.

A seguir, são apresentadas três soluções tecnológicas de atualizações dinâmicas para sistemas legados. As duas primeiras utilizam a tecnologia de Reflexão Computacional, enquanto que a terceira é aplicada utilizando-se o Paradigma da Orientação a Aspectos.

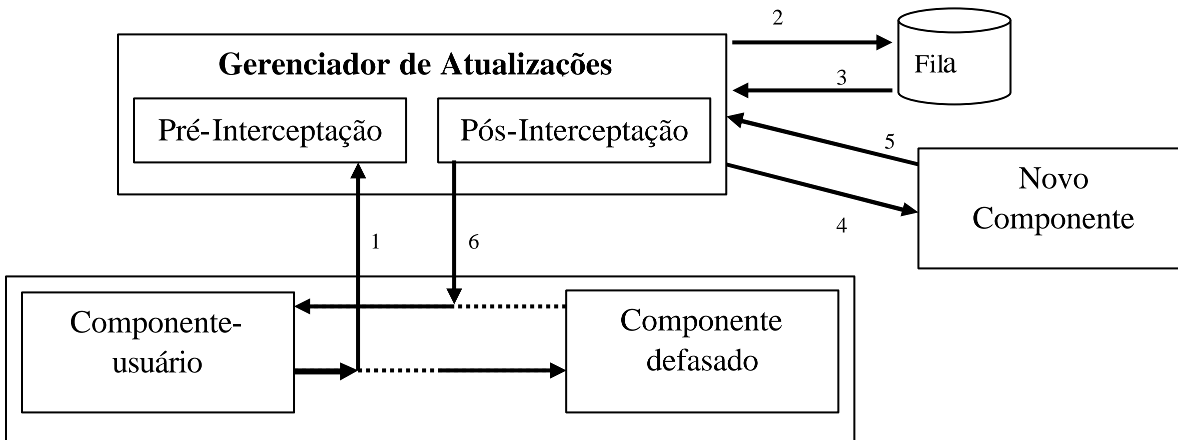

Figura 1: Processo de Gerenciamento de Atualizações

Na Figura 1, a numeração de um a seis, representa os passos realizados durante uma atualização dinâmica em componentes, aplicando as técnicas existentes.

1) Gerenciador de atualizações (GA) - intercepta todas as chamadas ao componente defasado;
2) O GA armazena as chamadas interceptadas na fila enquanto o novo componente é instanciado;
3) Após instanciar o novo componente, o GA obtém a relação de chamadas armazenadas na Fila;
4) O GA repassa tais chamadas ao novo componente;
5) O novo componente atente as requisições repassadas, retornando-as ao GA; e
6) O GA retorna ao componente-usuário os resultados.

Note-se que, o GA após realizar atualizações, deverá permanecer interceptando chamadas ao componente defasado, pois o componente-usuário não possui a referência do novo componente. Desta forma, o GA deverá permanecer em execução até a próxima interrupção do sistema.

4.1. Interceptações pelo Ambiente de Execução

Para linguagens interpretadas, entende-se por ambiente de execução o interpretador, atualmente chamado de Máquina Virtual (VM), enquanto que para linguagens compiladas, entende-se o sistema operacional.
Nessa solução é utilizada reflexão de meta-objetos, a qual utiliza reflexão computacional comportamental [8], permitindo que meta-objetos manipulem de maneira conveniente o comportamento dos componentes a serem atualizados.
Para tornar possível a utilização da reflexão de meta-objetos, pode ser necessário prover serviços específicos nos ambientes de execução. Como exemplo, cita-se o protocolo de meta-objetos (MOP) Guaraná, implementado a partir da modificação da máquina virtual da linguagem Java, denominada *Kafee OpenVM* [9].

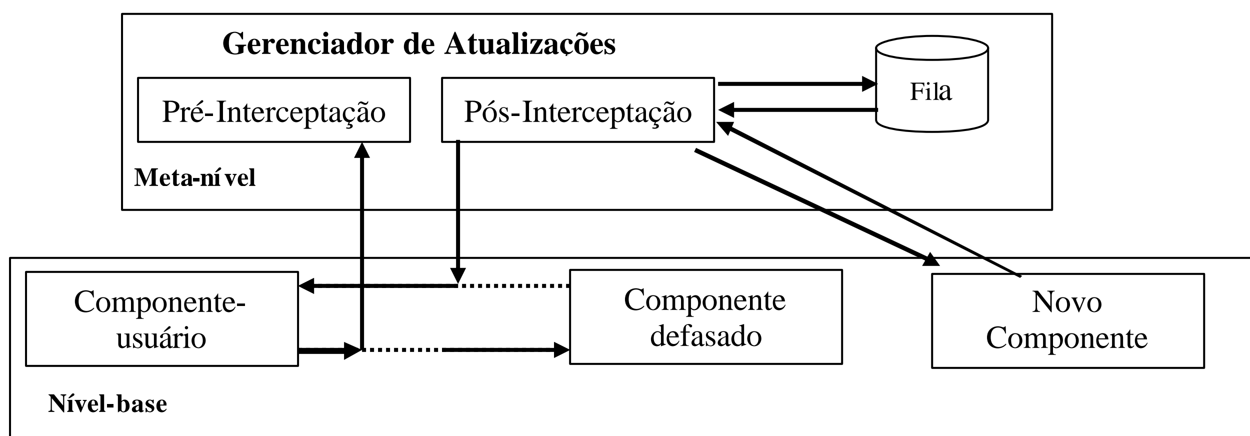

Figura 2: Processo de Gerenciamento de Atualizações implementado por Meta-Objetos

Desta forma, as aplicações executadas nessa máquina virtual podem sofrer atualizações dinâmicas, através do desenvolvimento de um gerenciador de atualizações baseado em meta-objetos. Todas as chamadas feitas aos componentes defasados configurados para atualização são interceptadas pelo ambiente de execução e entregues (reificadas) ao(s) meta-objeto(s) correspondente(s), neste caso, denominado gerenciador de atualizações. A partir deste momento, o gerenciador de atualizações pode armazenar as chamadas aos componentes defasados e redirecioná-las quando obtiver a referência do novo componente. A Figura 2 apresenta esta solução.
Note-se que, a Figura 2 implementa o mesmo modelo apresentado na Figura 1, utilizando meta-objetos para interceptar as chamadas ao componente defasado.
Essa solução apresenta maior flexibilidade, pois a instalação desse gerenciador de atualizações (meta-objetos) pode ser instanciado a qualquer momento. Entretanto, torna-se mister ressaltar que esta flexibilidade é diretamente dependente do ambiente de execução, ou seja, este deve estar preparado ou estendido para oferecer as devidas funcionalidades. Uma solução em camadas, utilizando esta abordagem, onde são descritos os diversos elementos da arquitetura, entre eles o gerenciador de atualizações, pode ser encontrada em [10].

4.2. Interceptações por *Proxies*

Um *proxy* propicia que chamadas a métodos de objetos ocorram indiretamente através dele, o qual age como um substituto ou representante do objeto ao qual se refere [11]. Assim, podem ser utilizados para controlar os acessos aos objetos que representam, como mostrado na Figura 3.

Esta abordagem é mais simples e mais facilmente portada entre diversos ambientes de execução. O problema desta solução é garantir que o *proxy* encapsule o componente defasado (objeto-alvo) o mais cedo possível. Caso contrário, se outro componente já possuir uma referência direta para o componente defasado, então não haverá nenhuma interceptação das chamadas feitas ao componente a ser atualizado.

Logo, esta abordagem diminui a flexibilidade, pois exige uma vinculação entre *proxy* e componente defasado antes que qualquer componente usuário obtenha uma referência ao componente defasado. Para algumas aplicações, isso não constitui necessariamente uma dificuldade, enquanto que, para outras, inviabiliza. No caso específico de atualizações dinâmicas, somente aplicações que incorporem o conceito de *proxies* em seus componentes poderiam se utilizar dessa solução. Para sistemas que não possuem tal característica, torna-se necessário implementar o conceito de *proxies* sobre os componentes e, após a primeira interrupção em sua execução, utilizá-los para atualizar os componentes dinamicamente.

Cabe ressaltar que a interceptação neste caso é puramente conceitual, pois o *proxy* é chamado explicitamente por qualquer componente que requisite um serviço do componente defasado.

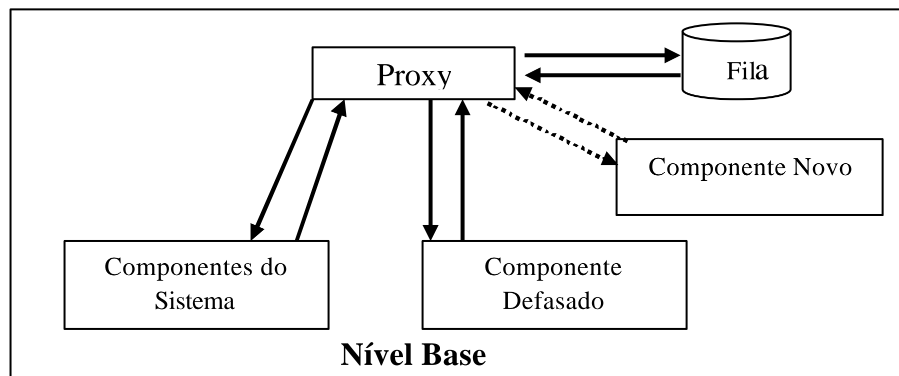

Figura 3: Processo de Gerenciamento de Atualizações implementado por Proxy

Note-se que a única diferença entre os processos mostrados pelas figuras 2 e 3, é que na figura 3 a interceptação é feita por *proxy*.

4.3. Interceptação por Programação Orientada a Aspectos - POA

O Desenvolvimento de Software Orientado a Aspectos (DSOA) doutrina que as preocupações pertinentes ao desenvolvimento de um sistema devem ser quebradas em diferentes dimensões [12]. Na prática, equivale a dizer que não se deve desenvolver um software preocupando-se ao mesmo tempo com questões como persistência, distribuição, segurança, auditoria, dentre outros.

Uma filosofia de desenvolvimento orientado a aspectos passível de aplicação aborda o conceito de linha de produto. Após a estrutura básica ter sido desenvolvida, outras características são agregadas à linha de produção, como por exemplo, numa montadora de automóveis. Quando aplicado ao software, tem-se um modelo de negócio que poderia receber na linha de produção as camadas de interfaces de entrada e saída, persistência, segurança, auditoria, entre outras. Neste caso, os problemas podem ser tratados separadamente, no momento adequado, e por diferentes equipes especializadas.

Isto é possível porque o compilador de aspectos, *weaver*, possui a capacidade de interceptar as chamadas a métodos em tempo de compilação ou execução, podendo desviar o fluxo do programa [13]. Assim, como ocorre na solução de meta-objetos, pode-se interceptar todas as chamadas aos componentes defasados, armazená-las e redirecioná-las ao novo componente, após este ter sido previamente instalado, como mostrado na Figura 4.

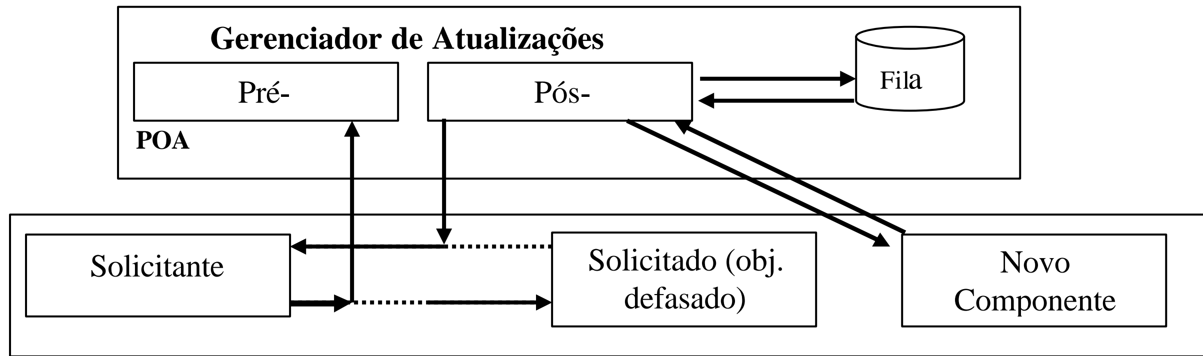

Figura 4: Processo de Gerenciamento de Atualização por Orientação a Aspectos

Note-se que, a única diferença entre os processos mostrados pelas figuras 3 e 4, é que na figura 4 a interceptação é feita por POA.
Considerando as técnicas para atualização dinâmica apresentadas e as propriedades de Hicks [4], discutidos na seção 2 constata-se que a quarta propriedade de Hicks [4], baixo *overhead,* não é completamente atendida.

5. Abordagem Proposta

Considerando-se, por exemplo, um cenário em que uma grande empresa de eletrodomésticos possua cinco Gerentes de Vendas, um para cada região do País. Infelizmente, a empresa não mais poderá contar com o trabalho eficiente do Gerente da Região Sul, que está pronto para ir trabalhar numa empresa de computadores. Para sua vaga, ele indicou um profissional de confiança que, após análise de currículo e entrevista, foi aceito para o cargo. No último dia de trabalho do Gerente de Vendas da Região Sul, a Secretária do Departamento de Vendas telefona para todos os 20 parceiros das sub-regionais da Região Sul para informar-lhes o nome e telefone do novo Gerente. Agindo assim, a secretária está solicitando que o cliente redirecione suas necessidades ao novo Gerente de Vendas responsável por atender as suas necessidades. O novo Gerente de Vendas, ao iniciar seu trabalho, terá que ler documentos, entender o processo e normas da empresa, e buscar se interar das informações relevantes para desempenho de sua atividade. Somente após engajar-se no cotidiano da empresa, o processo poderá ser executado com pelo menos a mesma eficiência anterior.
Para a secretária poder realizar esta tarefa, ela deve:

- Manter uma relação de todos os parceiros da empresa;
- Discernir quais parceiros dependem do Gerente de Vendas que está saindo da empresa;
- Conhecer os números de telefone do antigo e do novo Gerente de Vendas; e
- Ser capaz de interceptar as chamadas telefônicas para o Gerente de Vendas antigo, a partir do momento que o mesmo não mais responder pela empresa. Após a interceptação, a mensagem deve ser redirecionada para o novo Gerente de Vendas.

Para que o trabalho da secretária tenha completo sucesso, o cliente deve:

- *Ser capaz de guardar o telefone do Novo Gerente e utilizá-lo quando necessário*.

Este cenário é bastante conveniente, pois mostra uma interação simples, eficaz e eficiente. Entretanto, implementar esta solução para resolver as dificuldades de troca de componentes está longe de ser tão trivial quanto um telefonema.
Para a aplicação em questão ressalta-se que um objeto não pode ser instanciado sobre outro objeto e ocupar o mesmo espaço de memória. Por isso, repassar o telefone entre os gerentes não constitui uma solução computacional apropriada, mesmo isso sendo possível no mundo real. Neste caso, o número de telefone equivale ao endereço de memória dos componentes.

No exemplo acima, por analogia, o gerenciador de atualizações nos sistemas computacionais assume o papel das secretárias no mundo real, e as atividades por elas realizadas equivalem aos requisitos do gerenciador de atualizações, e deve:

- Manter informações de todos os componentes do sistema;
- Relacionar os componentes que dependem de um outro componente específico;
- Conhecer a referência do componente defasado e do novo componente;
- Ser capaz de interceptar as chamadas de métodos do sistema; e
- Verificar a compatibilidade das assinaturas dos métodos chamados e dos métodos oferecidos pelo sistema.

Por outro lado, todos os componentes do sistema, com exceção do componente defasado (gerente que está saindo), devem ser capazes de trocar internamente uma referência defasada por outra nova, da mesma forma que os parceiros da empresa devem receber e armazenar o telefone do novo Gerente de Vendas. Assim, os componentes-usuários, que dependem do componente defasado, podem referenciar o componente correto após uma atualização do sistema. Este requisito é especificado como segue:

- *Os componentes devem ser capazes de trocar uma referência antiga por outra nova.*

Por último, o novo componente deve ser capaz de configurar seu estado para ficar idêntico ao componente defasado, quando este começou a ser atualizado. Portanto, o novo componente deve:

- *Ser capaz de capturar e assumir o estado do componente defasado.*

São apresentados abaixo os resultados das investigações e discussões sobre a possibilidade de atender cada um dos requisitos citados e a viabilidade de suas implementações.

a) Manter informações sobre todos os componentes do sistema

Durante a análise do problema, levantou-se algumas soluções possíveis para resolvê-lo, como por exemplo, arquivos de configuração e/ou reflexão computacional.
A reflexão computacional apresenta-se, à primeira vista, como uma boa solução, pois pode permitir relacionar as dependências de um componente. Porém, não oferece suporte para identificar quais componentes dependem de um componente especifico.
A solução encontrada, para o cumprimento deste requisito, foi analisar as dependências a partir da classe que contém o método principal, por exemplo, em Java e C, o método main(). Neste caso, é possível rastrear todas as chamadas, identificar os componentes de quem a classe principal depende e continuar a busca até mapear toda a árvore de chamadas de componentes. Devido ao custo computacional deste processamento, a construção desta árvore deve ser realizada uma única vez, e o arquivo contendo as informações deve ser mantido pelo gerenciador de atualizações, fazendo as devidas alterações quando ocorrerem atualizações no sistema.
Para implementar esta solução, deve-se instrumentar o código-fonte ou o código objeto do programa. No caso da linguagem Java, por exemplo, existem ferramentas que facilitam a realização de análise do código-fonte, como por exemplo o Java Compiler Compiler [14], ou de código objeto, como o ByteCode Engineering Library [15].

b) Relacionar os componentes que dependem de um outro componente específico

Ao se processar o mapeamento da árvore de chamadas é possível identificar os componentes que chamam os serviços de outros componentes. Para o cumprimento deste requisito, pode-se criar facilmente uma relação contendo o nome dos componentes e de quem ele depende. Assim, a

qualquer momento, o gerenciador de atualizações pode fornecer a identificação do componente defasado a ser atualizado.

c) Conhecer a referência do componente defasado e do novo componente

Para atender a este requisito, o gerenciador de atualizações deve iniciar uma instância do novo componente e guardar a referência. A referência do componente defasado pode ser obtida pelo *proxy*, POA ou Reflexão Computacional.

d) Ser capaz de interceptar as chamadas de métodos do sistema

Para o cumprimento deste requisito, a interceptação de chamadas pode ser realizada por *proxy*, POA ou meta-objetos. Pelo fato do *proxy* atuar como um representante de um ou mais objetos, a interceptação é apenas em nível conceitual, pois, na verdade, a chamada a ele é realizada de forma explícita.

No caso de POA e meta-objetos a interceptação é um pouco mais elaborada. A chamada ao componente defasado é interceptada para tratamento e possível redirecionamento.

e) Verificar a compatibilidade das assinaturas dos métodos chamados e dos disponíveis no sistema

Este requisito serve para garantir a integridade do sistema. Sempre que houver atualização de componentes, o gerenciador de atualizações deve assumir a responsabilidade de verificar se todos os métodos que o novo componente chama são atendidos pelos demais componentes-usuários e vice-versa. Isto deve ser feito verificando se há correspondências das assinaturas desses métodos. Este requisito impede que um componente chame serviços inexistentes de outro componente.

No exemplo do mundo real, apresentado anteriormente, este requisito equivale à análise de currículo e entrevista com o candidato à vaga. Neste caso, tanto o funcionário quanto a empresa podem apresentar necessidades que o outro não pode suprir, gerando uma incompatibilidade entre as partes.

Para atender este requisito é necessário dividí-lo em duas partes. A primeira consiste em descobrir quais são os serviços que o novo componente oferece aos componentes usuários. Na prática, a implementação pode ser feita aproveitando-se do poder de introspecção dos pacotes de reflexão, como ocorre em Java e C++, por exemplo.

A segunda parte consiste em levantar quais são os serviços que os componentes usuários chamam do componente defasado. Esta informação está na relação criada durante o processamento do mapeamento e criação da árvore de chamadas que também deve capturar as assinaturas dos métodos.

Analisando as informações de dependências armazenadas num arquivo, obtidas pelo pacote de reflexão e processamento da árvore de chamadas, é possível identificar se o novo componente atende as requisições solicitadas (interfaces) pelos demais componentes e vice-versa.

f) Os componentes devem ser capazes de trocar uma referência antiga por outra nova

Os componentes do sistema devem ser capazes de trocar na memória uma referência defasada por outra nova, da mesma forma que os parceiros da empresa conseguem receber e armazenar o telefone do novo Gerente de Vendas. Esta habilidade permite que as novas chamadas realizadas pelos componentes do sistema possam ser feitas diretamente ao novo componente instalado.

Esta tarefa é executada via uma chamada de método que os componentes devem disponibilizar. O gerenciador de atualizações passa a referência do componente defasado e do novo componente como argumentos. O objeto tem que prover um **Serviço de Troca de Referências - STR**, que o torne capaz de realizar uma introspecção na memória do computador onde ele está localizado, trocando todas as referências antigas pelas novas.

Torna-se importante ressaltar que as linguagens de programação ofereçam suporte a este serviço ou que componentes sejam construídos para este fim. Esse serviço deve ser atômico, ou seja, a partir do momento que se inicia, o processamento não deve parar sua execução, emulando uma única instrução de processador. A atomicidade é importante para evitar possíveis inconsistências de referências ocasionadas pelas constantes trocas de contextos e estados entre os processos carregados na memória.

Ao se atender a este requisito, a degradação de desempenho dos sistemas atualizados é eliminada, atendendo completamente a quarta propriedade de Hicks [4], baixo *overhead*.

g) Ser capaz de capturar e assumir o estado do componente defasado

Após iniciar o processo de enfileirar as requisições ao componente defasado, o gerenciador de atualizações pode instanciar o novo componente e liberá-lo para atualizar seus estados, de acordo com os valores dos estados do componente defasado. Para isso, faz-se necessário que o novo componente ofereça um serviço que permita instruí-lo a assumir o estado do componente defasado. Logo, a referência do componente defasado é passada como argumento do método e os valores podem ser capturados do componente defasado e configurados no novo componente.

Quando existir atributos privados, que não possuem métodos de leitura e escrita, este serviço somente funcionará se a classe do novo componente for a mesma do componente defasado. De acordo a programação orientada a objetos, a declaração de atributo como privado faz com que a sua visibilidade esteja disponível apenas para objetos pertencentes a mesma classe.

Porém, é factível a situação em que o sistema é remodelado e uma classe substituída por duas, e os dois atributos privados existentes distribuídos entre elas. Neste caso, a solução por *proxy* torna-se viável se conceitos, como por exemplo, de reflexão computacional ou POA, forem utilizados, propiciando as devidas alterações estruturais no *proxy* para representar o(s) novo(s) componente(s). Especificamente em Java, um *proxy* pertence ao pacote de reflexão criando o conceito de *proxies* reflexivos.

Uma solução por meta-objetos ou POA ainda permanece possível. Meta-objetos tem acesso irrestrito a todos os campos de um objeto que ele controla. A POA, com o pacote AspectJ, permite a declaração de um aspecto como privilegiado e permite seu acesso a qualquer campo da classe independente da sua declaração.

Abaixo apresenta-se um algoritmo em alto nível de abstração para a abordagem proposta.

1. Identificar a necessidade de atualização de um sistema que pode estar agendada a priori;
2. Enfileirar todas as chamadas para o componente defasado.
3. Instanciar o novo componente;
4. Identificar o novo componente e listar todas as chamadas que ele faz para classes externas a ele:
   4.1. Verificar se os componentes externos são capazes de atender todas as chamadas ao novo componente;
5. Identificar todas as chamadas que os componentes usuários podem fazer para o novo componente:
   5.1. Verificar se o novo componente é capaz de atender a todas as chamadas dos demais componentes usuários;
6. Se novo componente é compatível com o sistema e o componente defasado estiver sem processamento então:
   6.1. Transferir o estado do componente defasado para o novo componente;
   6.2. Substituir as referências ao componente defasado;
7. Utilizando a abordagem FIFO (*First in First out)*, redirecionar as chamadas da fila para o novo componente;

8. Eliminar o componente defasado.

Um fator complicador existente neste algoritmo e, ao mesmo tempo extremamente relevante, consiste em responder algumas perguntas, tais como:

- Como saber se o componente defasado está processando alguma informação solicitada por um componente usuário?
- Como saber se o componente defasado não está esperando uma resposta de um outro componente a quem requisitou um serviço?

Neste ponto, torna-se importante salientar a necessidade dos ambientes de execuções proverem **Serviços de Identificação de Processamento - SIP**. Esses serviços permitirão ao gerenciador de atualizações identificarem o momento apropriado em que o componente defasado não se encontrar mais em uso, podendo continuar a execução do algoritmo supracitado, a partir do passo 6.1.

6. Conclusão

O problema endereçado neste artigo consiste em propor uma nova sistemática para atualização dinâmica em componentes. A solução escolhida para resolver o problema acima consiste em desenvolver uma abordagem sobre atualização dinâmica em componentes de sistemas orientados a objetos, visando aumentar a sua eficiência e reduzir a degradação do desempenho dos sistemas atualizados e o desperdício de recursos envolvidos.

Inicialmente, foram investigadas as soluções existentes por meta-objetos, *proxy* e POA, constatando-se que, apesar de cumprirem seus papéis, representam apenas soluções paliativas pelo fato das soluções existentes degradarem o desempenho do sistema proporcionalmente ao número de atualizações realizadas, e por diminuírem a quantidade de interrupções sem eliminá-las.

A abordagem proposta apresentou um conjunto de requisitos como alternativas de solução que tornam possível eliminar as interrupções necessárias para restabelecer apropriadamente as referências entre os componentes usuários e os novos componentes. Uma das principais contribuições deste artigo consistiu da constatação de que a abordagem proposta elimina a degradação de desempenho dos sistemas atualizados, atendendo completamente a quarta propriedade de Hicks, propiciar baixo *overhead*.

Os autores deste trabalho de pesquisa acreditam que a implementação desta abordagem em diversos sistemas dinâmicos em diferentes domínios do conhecimento poderá propiciar a validação desta abordagem.

Devido à complexidade do problema e ao custo envolvido na tarefa de manutenção e atualização de software de missão crítica, os autores propõem a inserção de dois novos serviços para que a abordagem conceitual possa ser aplicada em sua plenitude. O **Serviço de Troca de Referências,** que deve capacitar os componentes-usuários a atualizarem as referências ao componente defasado pela referência ao novo componente, e o **Serviço de Identificação de Processamento,** que consiste em identificar se um componente encontra-se em processamento, aguardando uma resposta ou livre para retirá-lo da memória.

Para que seja dada continuidade a esta investigação, sugere-se a realização de experimentos de atualização dinâmica com substituição de componentes centralizados e distribuídos, objetivando resolver problemas de segurança envolvendo carga dinâmica de classes, entre outros. Além disso, pesquisar novas abordagens que propiciem atualizações dinâmicas em nível de atributos e serviços, visando diminuir a granularidade dessas atualizações, tornará possível eliminar a necessidade de substituir um componente inteiro, como por exemplo, uma classe.

Finalmente sugere-se que as soluções existentes devem ser consideradas paliativas para a atualização de Sistemas de Missões Críticas, não devem ser aceitas como soluções definitivas, e a

comunidade científica e tecnológica deve direcionar esforços para adotar a abordagem proposta, enfatizando a necessidade de implementar os serviços **STR** e **SIP**.

Referências

[1] Pressman, R. S. **Engenharia de Software**. São Paulo: Mcgraw-Hill Interame, 5ª Edição, 2002.

[2] MYERS, G. **The Art of Software Testing**. New York: John Willey & Sons, 1979.

[3] Lyu, Janghoon; Kim, Youngjin; Kim, Yongsub; Lee, Inhwan. **A procedure-based dynamic software update**. In: The International Conference on Dependable Systems and Networks, 2001. Proceedings… The International Conference on, Vol., Iss., 2001. Pages:271-280. IEEE, 2001.

[4] Hicks, Michael; Moore, Jonathan T.; Nettles, Scott. **Dynamic Software Updating**. In: Special Interest Group on Programming Languages- SIGPLAN 2001 conference. Snowbird, Utah, United States. Proceedings... New York, NY, USA: ACM Press, 2001.

[5] Bialek, Robert Pawel. **The Architecture of a Dynamically Updatable, Component-based System**. In: Computer Software and Applications Conference-COMPSAC 2002. Proceedings... 26th Annual International, Vol., Iss., 2002 Pages: 1012- 1016. IEEE, 2002.

[6] Costanza , Pascal. **Transmigration of Object Identity: The Programming Language Gilgul**. Disponível em: <citeseer.ist.psu.edu/481427.html>. Acesso em: 10 abr. 2004.

[7] Gupta, D., Jalote, P. **On-line Software Version Change Using State Transfer Between Process**. Software Practice and Experience, Vol23, No. 9, Sep. 1993.

[8] Lisbôa, M. L. B. **Reflexão computacional no modelo de objetos**. Porto Alegre: CPGCC da UFRGS, 1998.

[9] OLIVA, A.; BUZATO, L. E. **Composition of Meta-Objects in Guaraná**. Campinas: Instituto de Computação, Universidade Estadual de Campinas – UNICAMP, 1998. Disponível em: <http://www.dcc.unicamp.br/~oliva/guarana>. Acesso em: 15 jan. 2004.

[10] Bialek, Robert. **Dynamically Updatable Component-based System (DUCS)**. In: Object-Oriented Programming Systems, Languages and Applications Conference – OOPSLA. Anaheim, CA, USA, 2003. Proceedings... New York, NY, USA: ACM Press, 2003.

[11] Blosser, Jeremy. **Explore the Dynamic Proxy API**. JavaWorld Magazine. Disponível em <http://java.sun.com/developer/technicalArticles/DataTypes/proxy/>. Acesso em: 15 mar. 2004

[12] IBM Research Group. **Multi-Dimensional Separation of Concerns: Software Engineering using Hyperspaces**. IBM Corporation. Disponível em: <http://www.research. ibm.com/hyperspace/>. Acesso em: 08 set. 2003

[13] The aspect team. **The Aspect Programming Guide**. Xerox Corporation, Palo Alto Reserch Center. Disponível em: <http://www.research.ibm.com/hyperspace/ConcernSpaces.htm>. Acesso em: 13 set. 2003.

[14] Sun Microsystem. **JavaCC:JavaCC Home**. Disponível em <https://javacc.dev.java.net/>. Acesso em: 10 abr. 2004.

[15] Apache Software Foundation. **BCEL- Bytecode Engineering Library**. Disponível em: <http://jakarta.apache.org/bcel/>. Acesso em: 25 jan. 2004