A Service-based Architecture for ERP Systems
Uma Arquitetura para Sistemas ERP baseada em Serviços

Márcio Vinícius Gagliotti Cesar[1], Rodrigo Rocha Silva[1], Leandro Luque[2]

[1] Pós-Gradução em Engenharia de Software SOA – Veris Faculdades (Grupo Ibemc) – São José dos Campos – SP – Brasil
[2] Faculdade de Tecnologia de São Paulo (FATEC) – Mogi das Cruzes – SP – Brasil
mvgagliotti@gmail.com, rodrigo.silva@veris.edu.br, leandro.luque@fatec.sp.gov.br

Abstract. The ERP systems emerged in the nineties in large industries as IT infrastructure to support their business operations, and gained attention by their integration features, offering advantages over older independent monolithic systems, designed to support single department's needs. Currently, companies face challenging scenarios in which there are many cross-platform systems that need to communicate to each other, enabling the generation of new business solutions faster than ever. SOA is a paradigm that aims to reach these requirements and offer a more manageable environment with reuse. The present paper introduces a service-based architecture for the building of an ERP software kernel with support to dynamic service binding, which become available to other modules as long as to external systems. The proposed architecture is presented in a descriptive form and by UML diagrams. A prototype has been developed as proof-of-concept to verify the advantages of this approach.

Resumo. Os sistemas ERP surgiram nos anos 90 em grandes indústrias como suporte de TI para apoiar suas operações, e se destacaram pela sua característica integradora, oferecendo uma vantagem em relação aos antigos sistemas monolíticos e que atendiam aos departamentos isoladamente. Atualmente, as empresas cada vez mais se deparam com cenários nos quais existe uma grande multiplicidade de sistemas, de diferentes plataformas, e há necessidade de integração destes sistemas para criar novas soluções com a agilidade cada vez maior. SOA é um paradigma que visa atender a estes requisitos e oferecer um ambiente mais propício ao reuso e gerenciamento dos ativos de software. O presente artigo propõe uma arquitetura modular, baseada em serviços, para criação do núcleo de um sistema ERP, com suporte a incorporação dinâmica de módulos e serviços, os quais ficam disponíveis para outros módulos e também para sistemas externos ao ERP. A arquitetura proposta é apresentada de forma descritiva e com o uso de diagramas UML. Apresenta-se também um protótipo desenvolvido com o objetivo de avaliar de forma prática as vantagens desta abordagem.

Palavras-chave: *SOA, Restfull, Web Services, Software Architecture, ERP.*

l. INTRODUÇÃO

Até o início da década de 1990, a maioria das grandes corporações utilizava sistemas de gestão independentes, desenvolvidos sem a visão da empresa como um todo, para atender necessidades específicas de cada departamento - o que gerava dificuldades de integração e obtenção de informações de qualidade.

Neste contexto, surgiram os sistemas integrados de gestão, conhecidos como ERPs (*Enterprise Resource Planning*), que oferecem soluções integradoras e suporte a processos como operacionais, produtivos, administrativos e comerciais.

De acordo com Brehm e Gómez [1], devido aos ERPs atuais não incluírem todas as funções de negócio requeridas pelas empresas, estas precisam manter diversos sistemas em paralelo, o que evidencia a necessidade de integração. Ademais, os departamentos de TI têm se mostrado incapazes de acompanhar a dinâmica da área de negócios no que diz respeito a mudanças e inovações.

Diante deste cenário, a SOA (*Service-Oriented Architecture* - Arquitetura Orientada a Serviço) fornece um modelo que pode auxiliar a minimizar os problemas de integração, além de proporcionar o reuso e o gerenciamento de componentes de software com o alinhamento de TI e negócios.

O objetivo deste trabalho é estudar e propor uma arquitetura orientada a serviços como base para a criação de um sistema ERP. Esta abordagem pode ser justificada pela grande adoção destes sistemas pelas empresas e a necessidade da evolução da arquitetura dos mesmos, para que estes se adéquem aos requisitos impostos pelo mercado, como rápida evolução, rápida resposta a mudanças, necessidade de distribuição em rede dos recursos computacionais, maior controle sobre os módulos, retorno de investimento (ROI), entre outros.

A abordagem de serviços como arquitetura para sistemas ERP foi adotada no trabalho de Brehm e Gómez [1], entretanto, a arquitetura proposta por ambos é fortemente baseada nos padrões WS-* e SOAP. A contribuição deste trabalho está em propor uma arquitetura mais genérica e implementável por qualquer padrão de comunicação baseado em serviços, inclusive o próprio WS*, e outros, como o padrão RESTful [2].

2. CARACTERÍSTICAS DE SISTEMAS ERP

Os sistemas ERP tipicamente são compostos por módulos e fazem uso de uma base de dados centralizada. Os módulos agregam as funções de negócio normalmente associadas a cada departamento da empresa. Desta maneira, as informações podem ser compartilhadas pelos diversos módulos, sem que haja duplicidade, aumentando a eficiência dos processos. A Figura 1 ilustra a estrutura clássica de um sistema ERP.

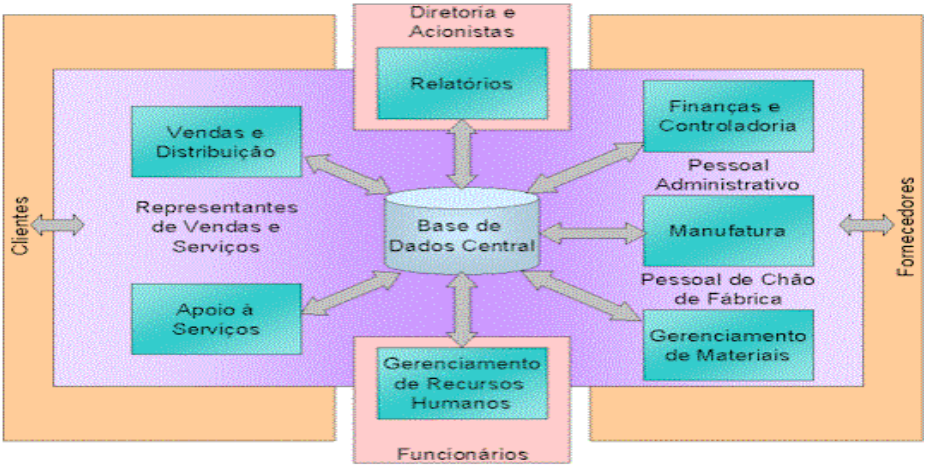

Figura 1. Estrutura de sistemas ERP. Fonte: Davenport [3].

3. ARQUITETURA DE SOFTWARE E SOA

Grandes sistemas computacionais não são estáticos e envolvem longos períodos de desenvolvimento. À medida que estes sistemas crescem, incorporam melhorias e novas funcionalidades, a complexidade para mantê-los cresce proporcionalmente. Por esta razão, existe a necessidade de se documentar as decisões de projeto e a estrutura dos componentes que compõem o sistema.

A disciplina que lida com estas questões é a Arquitetura de Software. A arquitetura de software é o conjunto de estruturas que incluem componentes, com suas propriedades externas e os relacionamentos entre eles, constituindo uma abstração do sistema e auxiliando o gerenciamento da complexidade [4].

A arquitetura de software é um dos principais habilitadores em proporcionar ganhos efetivos em agilidade e eficiência na manutenção e evolução dos sistemas de informação corporativos, o que é considerado um fator importante em ambientes competitivos.

A partir destas definições, é possível concluir que a arquitetura de software incorpora não só aspectos de funcionalidades que o sistema deve oferecer para atender aos requisitos de negócio (requisitos funcionais), como também aspectos de qualidade ou não comportamentais (requisitos não funcionais).

Uma vez definido o conceito de arquitetura de software, é importante conceituar a arquitetura SOA. Na visão de negócios, SOA oferece um modelo de estruturação da área de TI para que esta se alinhe com os objetos de negócio da organização de maneira flexível e ágil.

Do ponto de vista técnico, SOA define um estilo de arquitetura, no qual os componentes principais são os serviços - realizações de funcionalidades de negócio auto-suficientes, que fornecem uma interface ou contrato que os descrevem e podem

ser invocados por um consumidor, que através desta invocação obtém e/ou modifica dados do sistema fornecedor [5].

Uma importante característica de serviços no âmbito de SOA é que estes devem ser agnósticos de tecnologia. Isto significa dizer que um serviço pode ser implementado em uma plataforma tecnológica diferente da plataforma usada pelo consumidor. Para isto, é necessário que ambos utilizem o mesmo padrão para a troca de mensagens.

A SOA endereça requisitos de qualidade, como baixo acoplamento, distribuição dos componentes e independência de plataforma tecnológica, pois tais requisitos permitem que sistemas sejam mais dinâmicos, mais aptos a se integrarem com outros sistemas e com maior capacidade para reuso e para criação de novos componentes a partir de outros já existentes.

4. ARQUITETURA PROPOSTA

A arquitetura tradicional dos ERPs, conforme descrito anteriormente, define uma separação em módulos, que compõem as diversas funcionalidades do sistema. Cada módulo é responsável por oferecer uma determinada funcionalidade de negócio.

A arquitetura proposta mantém esta característica e define que cada módulo é composto por serviços. Os serviços são as unidades executáveis de software, que devem ser implementadas de acordo com o paradigma de SOA. É importante notar que os serviços são executáveis e módulos são agrupamentos lógicos para uma determinada classe de serviços.

Uma das premissas da arquitetura proposta é ser suficientemente abstrata para poder ser implementada em qualquer plataforma tecnológica. Para tal, é definido um componente central, o qual se convencionou chamar de *kernel*, que gerencia a troca de mensagens entre os módulos, por meio de invocações de serviços.

A comunicação entre os módulos não ocorre diretamente, o *kernel* faz a intermediação de toda requisição de serviço. A Figura 2 ilustra o diagrama de componentes da arquitetura. A ideia desta abordagem é manter um baixo acoplamento entre os módulos, pois basta que estes conheçam e direcionem as requisições para o *kernel* que por sua vez saberá como se comunicar com a aplicação que implementa o serviço solicitado, que pode eventualmente estar hospedada em outro servidor na rede.

Para realizar a comunicação com os módulos, é mantido um registro interno com os metadados de módulos e serviços. A arquitetura define um componente *AplicationServer*, que deve ser um servidor de aplicação ou container web. Este componente funciona como a infraestrutura para o *kernel* e a implementação pode se valer dos diversos produtos disponíveis no mercado, tais como JBoss AS ou Apache Tomcat, dentre outros.

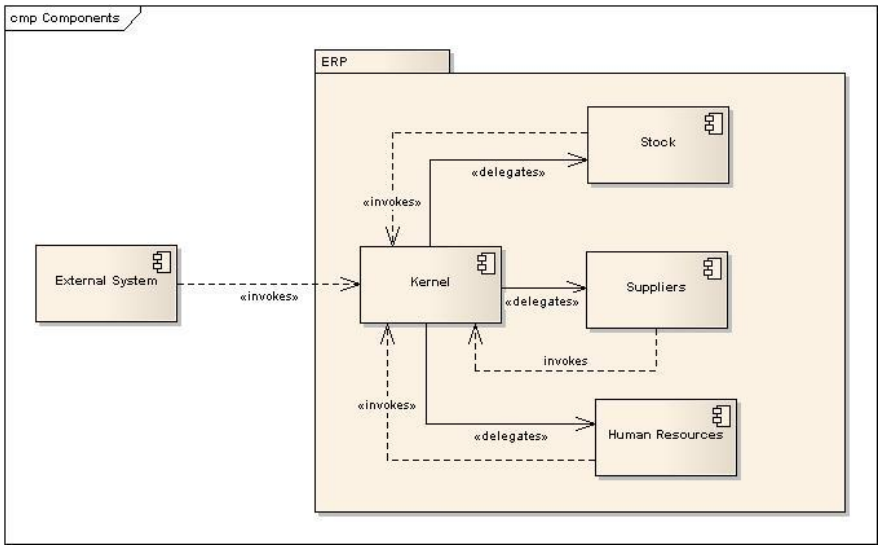

Figura 2. Ilustração da arquitetura proposta.

Com intuito de dividir as responsabilidades, o *kernel* é composto por cinco componentes principais: *ModuleManager*, *Router*, *ServiceInvoker*, *CompositeExecutor* (camada *back-end*) e *PanelControl* (*front-end*).

O *ModuleManager* é responsável por manter os metadados de serviços registrados no ERP. Os metadados são informações referentes aos serviços, como a URI (*Unified Resource Identificator*), que corresponde ao endereço físico do serviço na rede; *version*, que define a versão do serviço e os *schemas* de entrada e saída, que determinam o formato dos dados de entrada e saída do serviço. Este componente oferece serviços para manter registros de módulos e serviços, seus endereços físicos e versões, habilitar e desabilitar serviços. Este é um componente que segue o formato de um serviço, ou seja, ele não mantém estado (*Stateless*) e oferece métodos coesos para leitura e escrita.

O componente *Router* atua como um mediador entre o consumidor e o fornecedor de um serviço. É o componente que atua em tempo de execução, direcionando as invocações de serviços aos componentes físicos que os implementam, permitindo atender os requisitos de centralização e baixo acoplamento definidos na arquitetura. O comportamento deste componente pode ser associado ao padrão de projeto *Mediator*, definido por Ghama et al. [6] como um intermediador entre objetos que controla e coordena as interações entre eles, evitando o referenciamento explícito. O algoritmo implementado por este componente pode ser descrito da seguinte forma: o *Router* recebe uma requisição de um consumidor para um serviço de um determinado módulo, usa o *ModuleManager* para descobrir o endereço físico do serviço, direciona a requisição a este endereço, por meio do componente *ServiceInvoker* , obtém o resultado e retorna ao consumidor.

Complementando este componente, são definidos dois pontos de extensão no algoritmo, antes da invocação do serviço e após o recebimento do retorno da

invocação, nos quais é possível customizar o seu comportamento, através da sobrescrita de métodos *template*. O padrão *Template Method* foi adotado para esta finalidade. Segundo Ghama et al. [6], este padrão de projeto permite redefinir certos passos de um algoritmo, sem alterar a sua estrutura. Esta funcionalidade permite mudar o comportamento padrão do componente, de acordo com a necessidade, habilitando uma série de melhorias a serem incorporadas, sem contudo alterar o componente.

Como exemplos de uso desta funcionalidade, pode-se citar a adição de um *log* de chamadas de serviços e, a partir deste *log*, geração de estatísticas sobre os serviços, ou ainda adicionar um controle de segurança. O componente *ServiceInvoker* tem por finalidade isolar os detalhes técnicos da chamada a um serviço, como APIs e protocolos de comunicação usados. Este componente define uma interface para a invocação de um serviço e obtenção do retorno.

Na arquitetura, é possível definir dois tipos de serviços: simples e compostos. Os serviços simples são considerados de alta coesão e com maior possibilidade de reuso. Os serviços compostos são aqueles que internamente acessam de forma orquestrada outros serviços, compondo uma funcionalidade de negócio. Do ponto de vista de usabilidade, estes serviços são vistos como um único serviço e também podem ser reusados por outros serviços compostos.

O componente *CompositeExecutor* é responsável por executar os serviços compostos. Os serviços compostos, como o nome já estabelece, executam mais de um serviço de forma orquestrada, ou seja, a resposta de uma chamada é passada como parâmetro de entrada para o serviço subsequente, e assim por diante até que se encerre a execução. Para lidar com questão de transações envolvendo serviços, é possível definir que um serviço dentro da composição é um serviço transacional e define um segundo serviço, responsável por desfazer a ação de sua chamada. O *CompositeExecutor* então quando recebe um retorno de erro de um serviço, faz o *rollback* das chamadas transacionais já executadas anteriormente.

Por fim, o componente *PanelControl* estabelece uma interface de usuário para configuração do *kernel* e oferece operações para manutenção de módulos e serviços. A Figura 3 exibe um diagrama de componentes que compõem o *kernel*.

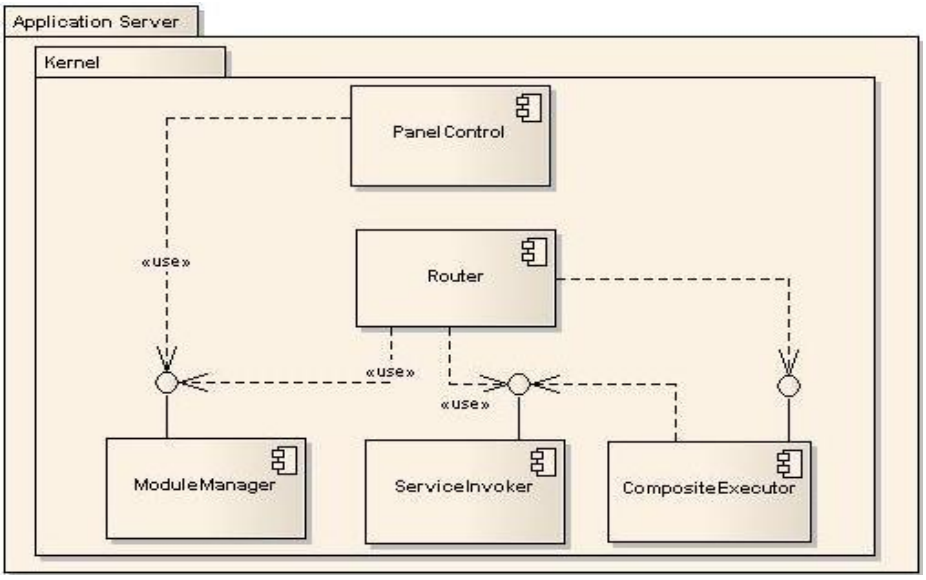

Figura 3. Arquitetura interna do *kernel*.

5. IMPLEMENTAÇÃO E PROVA DE CONCEITO DE ARQUITETURA

5.1 Implementação do *Kernel*

Para a implementação do componente *kernel*, foi usada a plataforma Java. Além do SDK Java, foram usadas bibliotecas Java para necessidades específicas do projeto. A Tabela 1 descreve as tecnologias usadas.

Tabela 1. Lista das tecnologias usadas na implementação do *Kernel*.

Tecnologia	Propósito
Maven	Organização dos componentes e controle de dependências
Apache Tomcat	Contêiner Java para aplicações servidoras na web.
Servlet API	API Java para aplicações executáveis em contêineres web
Apache Http Components	API para troca de mensagens com o protocolo HTTP

Dom4J	API para leitura de XML
Maven (http://maven.apache.org/)	Organização dos componentes e controle de dependências

5.2 Implementação do Processo de Negócio

Para testar a arquitetura proposta, foi escolhido um processo de negócio de exemplo para servir de base para a criação dos serviços. Um processo genérico de Cotação de Compras foi modelado com a notação BPMN (*Business Process Modeling Notation*). Foi utilizada a ferramenta *Enterprise Architect* para o desenho do processo.

Os serviços foram implementados na linguagem Java usando Servlets e o servidor web Tomcat. Para testar a invocação dos serviços e avaliar as respostas, foi usada a API Java Swing para criar um aplicativo onde é possível definir o serviço a ser invocado para testar as requisições aos serviços. A Figura 4 ilustra a interface do aplicativo de testes.

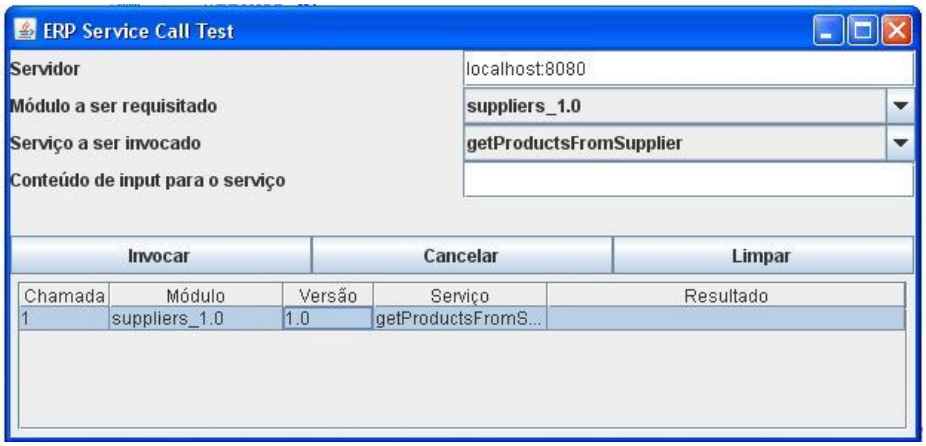

Figura 4. Aplicativo de testes.

Na modelagem do processo de cotações, foram definidos dois atores, o operador de estoque e o gerente de cotações. O processo inicia-se quando o operador de estoque identifica os produtos com estoque crítico, então, são obtidos os fornecedores homologados para estes produtos e uma solicitação de cotação é criada e enviada a estes fornecedores. O gerente de cotações recebe as cotações dos fornecedores e toma a decisão sobre qual cotação é a vencedora. A Figura 5 demonstra o processo de cotação de compras. A partir deste cenário, foi possível definir os seguintes serviços de negócio:

- obterProdutosEmEstoqueCritico;
- obterFornecedoresHomologados;
- manterSolicitacaoDeQuotacao;
- manterCotacoes;

- definirCotacaoVencedora

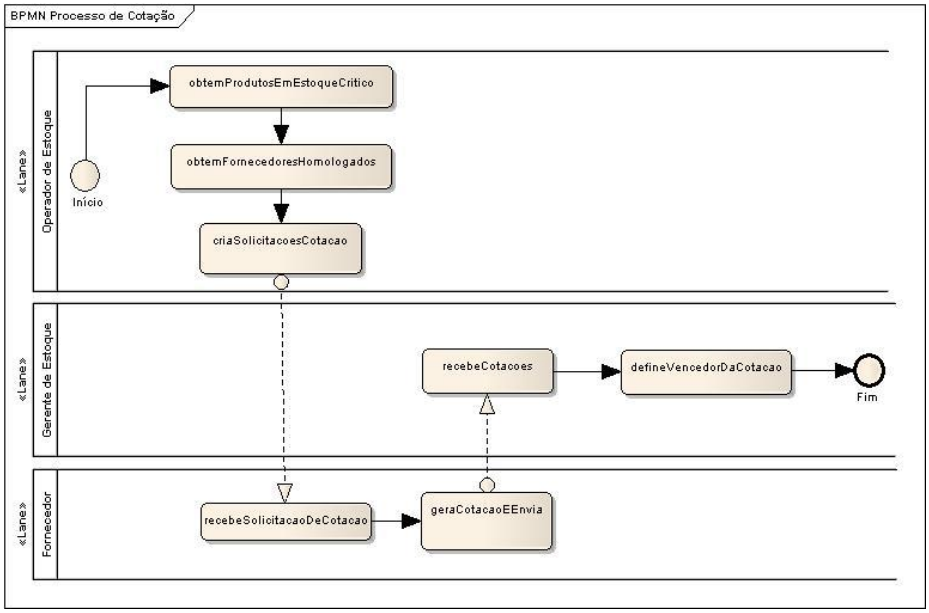

Figura 5. Processo de negócio "Cotação de Compras".

Conforme descrito anteriormente, o componente *ModuleManager* é o responsável por manter os metadados dos serviços. A Figura 6 exibe o XML com o mapeamento do serviço *obterProdutosEmEstoqueCritico*. O atributo URI define o caminho para a invocação do serviço. Os atributos *inputSchema* e *outputSchema* definem respectivamente o formato de entrada e saída que o serviço trabalha.

```
<?xml version="1.0" encoding="UTF-8"?>
<service>
    <name>obterProdutosEmEstoqueCritico</name>
    <description>Retorna uma coleção de produtos em estoque crítico</description>
    <uri>http://200.136.197.8/obterProdutosEmEstoqueCritico</uri>
    <outputSchema>
        <![CDATA[
            <?xml version="1.0"?>
            <schema xmlns="http://www.w3.org/2001/XMLSchema"
            xmlns:est="estoque_1.0">
                <include schemaLocation="estoqueTypes_1.0.xsd"/>
                <xs:element name="est:produtos">
            </xs:schema>
        ]]>
    </outputSchema>
</service>
```

Figura 6. Mapeamento do serviço *obterProdutosEmEstoqueCritico*.

O componente *CompositeExecutor* permite a execução de serviços compostos. Segundo Josutis [5], na terminologia SOA, a composição de novos serviços a partir de

serviços já existentes é chamada de "Orquestração". Para demonstrar a orquestração de serviços na arquitetura proposta, foi criado o serviço composto *iniciarProcessoCotacacao*.

Este serviço faz uso dos serviços *obterProdutosEmEstoqueCritico*, *obterFornecedoresHomologados* e *gerarSolicitacaoDeCotacao*, para gerar solicitações de cotação aos fornecedores homologados a fornecer os produtos em estoque crítico em um dado momento.

A Figura 7 apresenta o XML de mapeamento do serviço composto. Os dois primeiros serviços são executados em sequência e a as respostas de ambos é a entrada para o terceiro serviço que gera a solicitação.

```
<?xml version="1.0" encoding="UTF-8"?>
<compositeService name="iniciarProcessoCotacao" version="1.0">
    <sequence id="produtosEFornecedores">
        <service module="estoque" name="obterProdutosEmEstoqueCritico" />
        <service module="estoque" name="obterFornecedoresHomologados" />
    </sequence>
    <sequence>
        <service module="estoque" name="gerarSolicitacaoCotacao"/>
    </sequence>
</compositeService>
```

Figura 7. Serviço Composto *iniciarProcessoCotacao*.

5. CONCLUSÃO

Este trabalho permitiu o estudo e a aplicação dos conceitos do paradigma SOA para a criação de uma arquitetura para sistemas ERP. Observou-se que a adoção de princípios de SOA, como baixo acoplamento e estruturação do software na forma de serviços agnósticos de tecnologia podem trazer benefícios aos sistemas ERPs, no que tange à manutenção e evolução destes sistemas de forma rápida e à integração deste com outros sistemas.

Deve-se ressaltar, no entanto, que tais benefícios aumentam a complexidade do desenvolvimento e da própria arquitetura como um todo e podem impactar em requisitos não funcionais como desempenho, uma vez que a camada de serviços lida com operações de *parsing* de arquivos e dados. Para minimizar os esforços com desenvolvimento, é necessário o uso de ferramentas, ambientes de desenvolvimento e bibliotecas que auxiliem na criação dos serviços.

A questão do desempenho pode ser minimizada com uso de *caching* dos metadados dos serviços, além do suporte para execução de serviços em paralelo, quando isto for possível. O modelo de serviços RESTfull também endereça a questão da performance, pois exige menos esforço de *parsing* das mensagens que o modelo WS*, sendo uma solução que vem ganhando atenção pelo mercado.

6. REFERÊNCIAS

1. N. Brehm and Gómez, J. M., Service-oriented Development of Federated ERP Systems. Workshop on software engineering methods for service-oriented Architecture, (2007).
2. R. T. Fielding, Architectural Styles and the Design of Network-based Software Architectures. Doctoral Dissertation, University of California, Irvine, (2000).
3. T. H. Davenport, Putting the Enterprise into the Enterprise System, *Harvard Business Review*. pp.121-131, (1998).
4. A. C. Varoto, Visões em arquitetura de software. 2002. 108f. Dissertação (Mestrado em Ciência da Computação) – Instituto de Matemática e Estatística, Universidade de São Paulo, (São Paulo, 2002).
5. N. M. Josuttis, SOA in practice. (O' Relly, 2007).
6. E. Ghama, R. Helm, R. Johnson and Vlissides, J. M., Design patterns: elements of reusable object oriented software. (Addison Wesley, 1994).