Separação de Interesses em Sistemas Distribuídos

Alexandre Sztajnberg
IME-UERJ/COPPE-UFRJ
Rua São Francisco Xavier, 524 – 6º andar
22550-000, Rio de Janeiro RJ
alexszt@ime.uerj.br

Marcelo Lobosco **Orlando Loques**
Computação Aplicada e Automação – CAA
Universidade Federal Fluminense – UFF
Rua Passo da Pátria, 156
24210-240, Niterói, RJ
{lobosco, loques}@caa.uff.br

Abstract

Separation of concerns is an approach intended to separate the conception of functional and non-functional requirements of the applications, maximizing component reusability chances. In this work we present and briefly evaluate four contemporary proposals that use separation of concerns concepts: computational reflection, composition-filters, aspect-oriented programming and configuration, and how they can be used on the context of modern distributed system design. After analyzing these proposals, configuration combined with aspect oriented programming was selected for it presents features like reusability, dynamic evolution capacity, and , considered important to system application design.
Keywords*: separation of concerns, computational reflection, composition-filters, aspect-oriented programming, configuration, Java, distributed systems.*

Resumo

Separação de interesses é uma estratégia que preconiza a concepção de requisitos funcionais e requisitos não-funcionais das aplicações de forma separada aumentando as chances de reusabilidade dos componentes projetados. Neste trabalho apresentamos e avaliamos quatro propostas atuais que utilizam o conceito de separação de interesses: reflexão computacional, filtros de composição, programação orientada a aspectos e configuração e como podem ser utilizadas na concepção de sistemas distribuídos. Após a análise destas propostas, a abordagem de configuração combinada a programação orientada a aspectos foi selecionada, por apresentar, mais evidentes, características como reusabilidade, capacidade de evolução dinâmica e abrangência, consideradas importantes na concepção de sistemas aplicativos.
Palavras-chaves*: separação de interesses, reflexão computacional, filtros de composição, programação orientada a aspectos, configuração, evolução dinâmica, sistemas distribuídos.*

1 - Introdução

Um dos grandes problemas apresentados na concepção de sistemas de *software* é a dificuldade em se separar o que o sistema deve fazer (definido pelos requisitos funcionais) daquilo que ele deve atender (definido pelos requisitos operacionais). Este problema, conhecido como separação de interesses (*separation of concerns*), leva quase sempre a uma codificação confusa, onde trechos de código que exprimem a funcionalidade da aplicação se entrelaçam com os trechos de códigos associados aos requisitos não-funcionais que devem ser atendidos[5]. Isto cria dificuldades durante todo o ciclo de vida da aplicação, e para o reaproveitamento de código para outras aplicações: por melhor que tenha sido a fase de projeto do sistema e a sua implementação, acaba-se misturando requisitos dos dois domínios a serem atendidos.

Aplicações concorrentes e distribuídas, onde os aspectos de coordenação, distribuição e comunicação são concebidos e implementados de forma entrelaçada à própria funcionalidade da aplicação, tornam evidente o problema da separação de interesses. No contexto destas aplicações vários benefícios são obtidos se os diversos aspectos puderem ser planejados e programados separadamente. Quatro propostas atuais, reflexão computacional [2], filtros de composição [1], programação orientada a aspectos [3, 5] e configuração [4, 6, 7, 8, 9], apresentam características que permitem a implementação de aplicações distribuídas segundo o conceito de separação de interesses e são comentadas individualmente nas seções 2, 3, 4 e 5. Na seção 6 são apresentadas algumas observações sobre estas propostas, e a partir de comparações entre as mesmas, as razões que levaram à escolha do paradigma de configuração como mais apropriado para a concepção e evolução de aplicações distribuídas. Finalmente, na seção 7, apresentam-se algumas conclusões e as perspectivas de trabalhos futuros envolvendo um modelo baseado em uma combinação de configuração e programação orientada a aspectos.

2 – Reflexão Computacional

Neste modelo, os aspectos funcionais e não-funcionais são separados com o uso de objetos-base e meta-objetos. Os objetos-base implementam em seus métodos a funcionalidade da aplicação. Já os meta-objetos implementam os procedimentos de controle que determinam o comportamento dos objetos-base a ele associados. Os objetos-base podem não estar cientes da existência dos meta-objetos, sendo que também não é obrigatório que todo objeto tenha um meta-objeto a ele ligado. As chamadas aos métodos do objeto-base são desviadas de modo a ativar meta-métodos, permitindo desta forma o meta-objeto supervisionar e modificar o comportamento do código funcional residente no objeto-base ou ainda adicionar novas funcionalidades a seus métodos. A Figura 1, retirada de [2], ilustra o modelo.

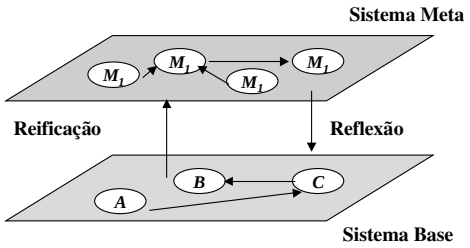

Figura 1- Objetos A, B e C implementam a solução de um problema específico. Objetos meta-nível M_1...M_4 controlam a execução dos objetos A, B e C e implementam parte de seu comportamento.

No desenvolvimento de aplicações distribuídas, os objetos-base implementam as funções específicas da aplicação e os aspectos de distribuição, comunicação e coordenação são implementados em meta-objetos. Em um meta-objeto é possível impor exclusão mútua ou sincronização de métodos ou estabelecer a comunicação entre objetos distribuídos sem que os objetos-base tenham sido planejados para isso. Atualmente a utilização de reflexão computacional está restrita a linguagens com extensões especializadas que oferecem mecanismos para associar objetos-base à meta-objetos e interceptar chamadas ao objeto-base para o meta-objeto.

3 – Filtros de Composição

O modelo de filtros de composição estende o modelo de orientação a objetos convencional. Seu modelo de objetos pode ser apresentado como duas partições separadas: a primeira suporta o modelo convencional de objetos, que é envolvido por uma segunda camada encapsuladora que contém extensões, que utilizam os chamados filtros de composição. A primeira parte, também chamada de implementação, é o núcleo do objeto; a segunda parte, chamada de interface, é usada para a gerência das mensagens que entram e saem do objeto. O modelo resultante pode ser visto na Figura 2, retirada de [1]:

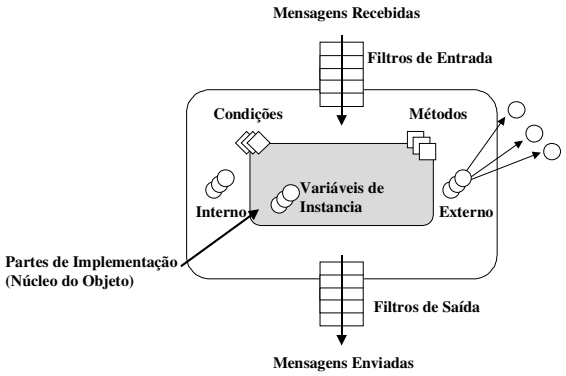

Figura 2– Os componentes do modelo de filtros de composição

A interface recebe as mensagens que chegam, processa-as e então opcionalmente as repassa para a implementação, onde levam à execução de métodos. Esse processamento é feito pelos filtros de composição: os filtros de entrada lidam com as mensagens recebidas pelo objeto, enquanto os filtros de saída lidam com

as mensagens que são enviadas pelo objeto. Considerando que todas as atividades no modelo de computação orientado a objetos são executadas através de invocação de mensagens, os filtros podem virtualmente controlar todo aspecto da computação orientada a objetos. Assim, os filtros de composição aplicam uma forma de reflexão computacional nas mensagens.

O conceito de filtros de composição admite filtros especializados em determinados aspectos não-funcionais. Em [1] é apresentada a linguagem Sina, uma linguagem orientada a objetos que implementa a idéia de filtros de composição e oferece filtros do tipo ***wait*** e ***meta*** que podem ser usados na concepção de aplicações distribuídas. Com um filtro *wait* programado na entrada de um objeto é possível impor sincronização aos métodos do núcleo do objeto através de bloqueio das mensagens baseado no estado do núcleo do objeto. O filtro meta permite manipular as invocações que chegam ao objeto ou sem dele. Neste sentido, mensagens de invocação de um método podem ser encaminhadas para objetos remotos através de uma rede de comunicação, sem que isto esteja programado no núcleo do objeto.

4. Programação Orientada a Aspecto (*Aspect-Oriented Programming*)

O paradigma de programação orientada a aspectos (AOP) procura solucionar o problema da separação de interesses através do uso de diferentes linguagens, cada qual especializada no atendimento de uma classe específica de requisito ou aspecto. A partir deste ponto um pré-compilador, denominado a*spect weaver*, combina as descrições dos códigos dos diferentes aspectos, gerando um único código para a aplicação (Figura 3). Do ponto de vista do programador, o requisito funcional (denominado nesta abordagem de componente) é programado independentemente de cada requisito não-funcional (denominado de aspecto), apesar do código gerado pelo pré-compilador entrelaçá-los.

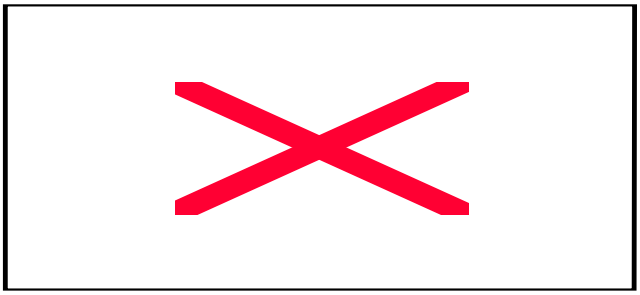

Figura 3 – O papel do *aspect weaver*

Em [5] é apresentada a linguagem D composta de uma linguagem base, JCore, e duas linguagens de aspecto, Cool e Ridl. JCore é um subconjunto de Java em que foram retiradas todas as construções relativas a sincronização, nativas de Java. Através de JCore a funcionalidade de uma aplicação distribuída é programada. Cool é a linguagem em que se descrevem os aspectos de sincronização e concorrência relativos a aplicação descrita em JCore. Ridl, por sua vez, é a linguagem através da qual se descrevem os aspectos que permitem invocações a objetos remotos, permitindo que a aplicação descrita em JCore seja composta por objetos distribuídos.

5 - Configuração

No paradigma de configuração, a estrutura de uma aplicação é concebida descrevendo-se seus componentes e como estes se interligam através de conectores. Para este fim é usada uma linguagem de configuração (textual ou gráfica)[9]. Os componentes, ou módulos, podem corresponder a unidades básicas de linguagens de programação, como classes e métodos, ou mesmo a outros objetos, como tabelas e arquivos. Cada módulo possui uma interface (assinatura) e uma especificação, que define sua funcionalidade. Já os conectores são responsáveis pelas ligações entre módulos, tendo também o papel de mediadores em suas interações. Pode-se citar como exemplo de conector o mecanismo de RPC (*Remote Procedure Call*), responsável pelo envio de pedidos de computação do cliente para o servidor e pela devolução das respostas do servidor para o cliente, controle de erros de transporte e também pelo empacotamento e desempacotamento dos eventuais parâmetros e respostas.

Os módulos são projetados de forma que as interfaces pelas quais eles podem se comunicar com outros módulos sejam compatíveis com os conectores (Figura 4). Diferentes conectores são disponíveis para o suporte de diferentes técnicas de comunicação (por exemplo, chamada remota de procedimento, *pipelines*, *sockets*, CORBA, etc.), definindo regras sobre o tipo de interface com a qual eles trabalham, garantias sobre a interação, compromissos com a qualidade dos serviços como ordenação ou desempenho, dentre outras.

Em princípio, a funcionalidade do módulo independe do tipo de conector utilizado, o que torna possível sua substituição em tempo de execução, permitindo adequar dinamicamente a aplicação às necessidades de estilo ou qualidade de conexão entre componentes.

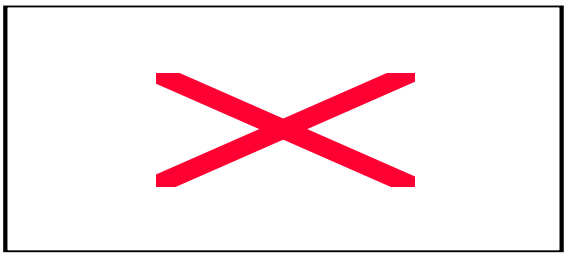

Figura 4 - Módulos independentes e módulos interconectados

Sistemas distribuídos são facilmente concebidos, descrevendo-se os módulos componentes e seus conectores de interligação. A estrutura da aplicação, incluindo o arranjo da distribuição dos módulos e aspectos específicos da comunicação necessários para realizar cada interligação podem ser descritos através da linguagem de configuração. Componentes e conectores utilizados em uma aplicação devem ser programados e estar disponíveis em uma biblioteca de componentes antes de se executar a aplicação. A configuração da aplicação independe da linguagem de programação com que seus componentes são implementados. Após a descrição da configuração, a estrutura descrita deve ser instanciada e então sua execução pode ser iniciada.

6 – Escolhendo um Paradigma para a Implementação da Plataforma

Para possibilitar a seleção do paradigma mais adequado a concepção de sistemas distribuídos, as propostas apresentadas são comparadas segundo os critérios de (1) reusabilidade, (2) separação de aspectos, (3) evolução dinâmica e (4) abrangência. Estes critérios foram considerados importantes para ferramentas ou ambientes de desenvolvimento de aplicações distribuídas, a partir de um estudo desenvolvido em [11].

	Reusabilidade
Reflexão	Objetos-base e meta-objetos podem ser reutilizados. Requer o planejamento do meta-nível visando permitir a reusabilidade.
CF	Filtros não são entidades independentes, portanto só podem ser reutilizados reescrevendo-se um novo código para um novo objeto. Entretanto, tipos de filtros disponíveis podem ser utilizados em aplicações distintas.
AOP	A reutilização de código em AOP é equivalente a da tecnologia de objetos. Os objetos relativos aos aspectos funcionais de uma aplicação podem ser reutilizados. Entretanto a reutilização dos aspectos não-funcionais descritos para determinada aplicação torna-se difícil. É necessário que se projete a aplicação em seus aspectos funcionais e, de acordo com os requisitos não funcionais, adaptar a descrição dos outros aspectos à mesma.
Configuração	Aplicações podem ser construídas a partir de módulos e conectores já desenvolvidos e disponíveis. Requer padronização de interface e estilos de interação. Módulos e conectores disponíveis também podem ser reutilizados na composição de módulos complexos.
	Separação de Aspectos
Reflexão	Aspectos funcionais e não-funcionais são separados em objetos e meta-objetos. Entretanto a separação de aspectos não-funcionais dentro de um mesmo meta-objeto ou através da composição de meta-objetos está ao encargo do programador.
CF	O núcleo do objeto implementa a funcionalidade. Filtros implementam aspectos não-funcionais. Filtros podem ser encadeados, permitindo a ação de filtros consecutivos para compor um comportamento específico. Aspectos não-funcionais só podem ser programados se filtros de tipos adequados estiverem disponíveis na linguagem.

AOP	Aspectos são definidos através de linguagens de aspecto específicas.
Configuração	Módulos representam os componentes funcionais da aplicação. Conectores permitem a interligação de módulos e a configuração de requisitos não-funcionais da aplicação. Estilos diferentes de interconexão dos componentes são planejados e implantados separadamente nos conectores. A composição de conectores implementando diferentes aspectos fica ao encargo do programador.
	Evolução Dinâmica
Reflexão	Extensibilidade é provida por meta-objetos; Na maioria das linguagens reflexivas disponíveis, meta-objetos são definidos estaticamente. Extensibilidade dinâmica de objetos-base deve ser suportada por um ambiente especializado que permita a criação de meta-objetos e a associação destes com o objeto base em tempo de execução.
CF	Se o objeto deve ser adaptado, sua interface tem que ser modificada. Novos filtros não podem ser acrescentados dinamicamente à interface do objeto. Entretanto os filtros permitem que o comportamento do objeto seja adaptado segundo as mensagens que chegam à interface, e de acordo com o comportamento dos filtros e o estado do próprio objeto.
AOP	Não permite. A aplicação terá que ser reprogramada para admitir nova funcionalidade.
Configuração	Obtida pela capacidade de (re)configuração de módulos e conectores simples ou compostos e pela reestruturação da aplicação durante sua operação.
	Abrangência
Reflexão	Linguagens de programação ou ambientes de execução com características reflexivas ainda não estão amplamente disponíveis; Não existem características padronizadas aceitas universalmente para se definir reflexão. Podem resolver vários tipos de problemas. Linguagens reflexivas são especializadas.
CF	A aplicação das idéias de filtros de composição demanda uma linguagem especializada e o acréscimo de novos tipos de filtros não é factível por desenvolvedores de aplicação. Pode resolver vários problemas cujos aspectos não-funcionais possam ser implementados pelos tipos de filtros disponíveis.
AOP	Linguagens particulares para cada aspecto devem estar disponíveis. Para o acréscimo de novos aspectos, novas linguagens deve ser concebidas e o *weaver* modificado.
Configuração	Linguagens e ambientes de configuração estão disponíveis. Entretanto, não existem padrões ou implementações específicas aceitas universalmente. Permite a concepção de uma grande gama de aplicações. Tipos de conectores são geralmente pré-definidos na linguagem de configuração ou podem ser adaptados à mesma. A facilidade composição de elementos permite a combinação de funcionalidades para atender a aplicações com requisitos diversos.

Tabela 1: comparação das propostas avaliadas

Através da comparação das propostas avaliadas, constatou-se que a configuração atende aos requisitos de reusabilidade, separação de aspectos, evolução dinâmica e abrangência de maneira mais flexível que as demais. Dentre as razões que tornam atrativo o uso de configuração na busca de soluções para o problema da separação de interesses, destacam-se:

- *a passagem da análise para a implementação é natural*, uma vez que este paradigma oferece correspondência direta entre o projetado e o implementado, dada as estruturas modularizadas tanto do modelo gerado durante a análise quanto do modelo do paradigma de configuração;

- *o entendimento do sistema como um todo é facilitado*. A linguagem de configuração usada para a descrição do sistema cria a possibilidade de verificação da correção do mesmo e a representação gráfica tende a facilitar sua compreensão;

- *é natural a separação dos interesses*. Tanto os componentes quanto os conectores podem implementar requisitos específicos do sistema;

- *possui poder computacional equivalente à reflexão, aos filtros de composição e à programação orientada a aspectos;*

- *em geral, a configuração mostra-se mais flexível e genérica em relação as demais abordagens*. Parte dos sistemas reflexivos, os filtros de composição e a programação orientada a aspectos não permitem mudanças dinâmicas nos sistemas e são dependentes da utilização de linguagens especiais de programação, diferentemente da configuração que em princípio só depende de um suporte de execução razoavelmente simples.

7 - Conclusões e Trabalhos Futuros

Mesmo tendo sido selecionada como a proposta mais adequada para a concepção de aplicações distribuídas, a configuração apresenta alguns pontos que ainda merecem pesquisa. Por exemplo, não existem definições nas linguagens de configuração para se expressar diferentes aspectos não-funcionais de uma aplicação. Desta forma, visando suprir essa deficiência, estamos propondo um modelo de configuração orientado à separação de aspectos [12]. Neste modelo, aspectos funcionais da aplicação são encapsulados nos módulos e aspectos não-funcionais ou operacionais da aplicação são contemplados nos conectores. Assim, os conectores tem o papel de interligar os módulos, definindo a estrutura da aplicação e capturar aspectos não-funcionais como a coordenação, distribuição, comunicação, tolerância a falhas, entre outros. Para maximizar a reusabilidade, além utilização da composição de módulos para a formação de módulos mais complexos, conectores também podem ser compostos. Isso permite que durante a concepção da aplicação, requisitos não-funcionais sejam atendidos através da composição de conectores que implementem aspectos não-funcionais mais simples.

Dois caminhos estão sendo seguidos em trabalhos em andamento: a formalização do modelo de configuração associado a uma linguagem de configuração orientada a aspectos [12] e a implementação de um protótipo contemplando a configuração dos aspectos de coordenação e distribuição [10] onde os conceitos do modelo possam ser testados. Na linguagem de configuração estão sendo introduzidos inicialmente aspectos de coordenação (que permitem a descrição de características de concorrência e sincronização), distribuição e comunicação para a realização de avaliações preliminares do modelo.

9 – Bibliografia

1. BERGMANS, Lodewijk M. J. "*The Composition-Filters Object Model*". Artigo derivado do capítulo 2 de "*Composing Concurrent Objects*", PhD thesis, University of Trese, junho de 1994.
2. GOLM, Michael. "*Design and Implementation of a Meta Architecture for Java.*" Diplomarbeit im Fach Informatik, Friedrich-Alexander-Universität Erlangern-Nürnberg, janeiro de 1997.
3. KICZALES, Gregor; *et alli*. "*Aspect-Oriented Programming*". Xerox Palo Alto Research Center, PARC Technical Report, fevereiro de 1997.
4. KRAMER, Jeff; *et alli*. "*Configuring Distributed Systems*". Proceedings of 5th. ACM SIGOPS on Models and Paradigms for Distributed Systems Structuring, setembro de 1992.
5. LOPES, Cristina V.; KICZALES, Gregor. "*D: A Language Framework for Distributed Programming*". PARC Technical Report, fevereiro de 1997.
6. LOQUES, Orlando; *et alli*. "*A Configuration Approach for Distribuited Object-Oriented System Customization*". Proceedings of Third International Workshop on Objected-Oriented Real-Time Dependable Systems, fevereiro de 1997.
7. SHAW, Mary; *et alli*. "*Abstractions for Software Architecture and Tools to Support Them*". IEEE Transactions on Software Engineering, Volume 21, Nº. 4, abril de 1995.
8. SZTAJNBERG, Alexandre; LOQUES, Orlando. "*Configuração e Reflexão Computacional em Aplicações Multimídia Distribuídas*". 15º Simpósio Brasileiro de Redes de Computadores, março de 1997.
9. WERNER, J. A. V. "*Ambiente Gráfico para a Construção, Configuração e Gerenciamento de Aplicações Distribuídas*". Dissertação de Mestrado, Departamento de Engenharia Elétrica, PUC-RJ, Rio de Janeiro, RJ, 1995.
10. LOBOSCO, M., Sztajnberg, A., Loques, O., "*Uma Plataforma para o Suporte à Evolução de Sistemas Distribuídos*", 2º Workshop de Sistemas Distribuídos, Curitiba, junho de 1998.
11. SZTAJNBERG, A. "Requisitos para Concepção de Sistemas Distribuídos", Relatório de Pesquisa, GTA/COPPE/UFRJ, 1997.
12. SZTAJNBERG, A. "Proposta de um Modelo de Configuração orientado a Separação de Aspectos", Relatório de Pesquisa, GTA/COPPE/UFRJ, 1997.