Um *Design Pattern* para Configuração de Arquiteturas de Software

Sérgio Teixeira de Carvalho (sergiotc@ic.uff.br)
Jonivan Coutinho Lisbôa (jlisboa@ic.uff.br)
Orlando Gomes Loques Filho (loques@ic.uff.br)

Universidade Federal Fluminense – Instituto de Computação
Niterói-RJ - Brasil

RESUMO

Este trabalho apresenta a descrição de um *design pattern*, chamado ***Architecture Configurator***, que modela o processo de implantação da configuração de um sistema de software. Os *patterns* são meios utilizados para documentar situações recorrentes em desenvolvimento de software. Sua utilização no estudo de implantação de arquiteturas visa facilitar o estudo comportamental de um sistema, sem que o projetista precise ater-se a detalhes de implementação. Além disso, o uso de *patterns* possibilita obter alguns requisitos desejados na implementação de arquiteturas, como separação de interesses, reutilização de componentes, facilidade de manutenção e extensão do sistema, entre outros.

Palavras-chave : *design patterns*, arquiteturas de software, configuração de software.

INTRODUÇÃO

A concepção de um sistema de software parte da definição de uma arquitetura, que o descreve em termos dos componentes que o integram – os módulos e conectores – e das ligações feitas entre eles, através de pontos de interação específicos – as portas.

A descrição de uma arquitetura pode ser realizada de maneira formal, através de uma linguagem de descrição arquitetural (ADL – *Architecture Description Language*). O produto da descrição arquitetural de um sistema é a sua **configuração**, ou seja, a estrutura topológica da aplicação. Na configuração, estão definidos os pontos de interação de cada módulo e cada conector, e também a maneira como os componentes irão interagir entre si – as ligações entre eles. A configuração é abstrata, e deve ser implementada mediante a criação das instâncias dos componentes, e a realização das ligações especificadas.

Quando bem definido, o projeto de uma arquitetura pode fornecer um nível de abstração que permite a análise do comportamento do sistema como um todo, sem a necessidade de se conhecer detalhes de implementação. Para conseguir isso, pode ser útil o reaproveitamento de experiências anteriores na implantação de arquiteturas de software. Isso é possível através da utilização de *patterns*, que são meios de se documentar situações recorrentes em desenvolvimento de software.

A descrição de *patterns* deve seguir o formato Contexto-Problema-Solução. Esse formato apresenta o contexto no qual o problema deve ser tratado, descreve o problema em si, e apresenta a solução empregada. Existem vários formatos padronizados de descrição, entre eles o padrão GoF, de *Gang of Four*, referência aos autores do primeiro catálogo a ter aceitação como uma forma padronizada para descrição de *patterns* [GHJ95].

Este trabalho apresenta uma descrição do *design pattern* ***Architecture Configurator***, que fornece uma base para a implementação de configurações arquiteturais. Para isso, fundamenta-se nos mecanismos de interceptação, encaminhamento e manipulação de requisições realizadas entre os componentes do sistema, e também na interligação entre eles [Car01].

A seguir, a descrição do ***Architecture Configurator***, segundo o formato GoF.

Nome

Architecture Configurator.

Objetivo

O *Architecture Configurator* modela o processo de implantação da configuração de um sistema, obtida mediante a descrição arquitetural do mesmo.

Contexto

A arquitetura de uma aplicação envolve componentes e um ou mais conectores. Os componentes interagem uns com os outros, requerendo e/ou fornecendo serviços, e os conectores intermediam essa interação. Um componente, neste contexto, refere-se a um processo, objeto, *procedure*, ou qualquer pedaço de código ou dados identificável.

Exemplo

Como exemplo pode-se considerar uma possível arquitetura para a aplicação Produtor-Consumidor com *buffer* limitado (PCB) apresentada na figura 1. Ela é composta de três módulos (**Produtor**, **Consumidor** e **Buffer**) e um conector (**Guarda**). Os módulos interagem entre si, requerendo ou fornecendo algum tipo de serviço, e possuem, cada um, sua funcionalidade intrínseca (requisitos funcionais): o **Produtor** produz itens requerendo o serviço específico para armazenar itens no **Buffer** (***put***); o **Buffer** armazena/recupera itens, fornecendo os respectivos serviços (***put***/***get***); o **Consumidor** consome itens, requerendo o serviço específico para retirar itens do **Buffer** (***get***).

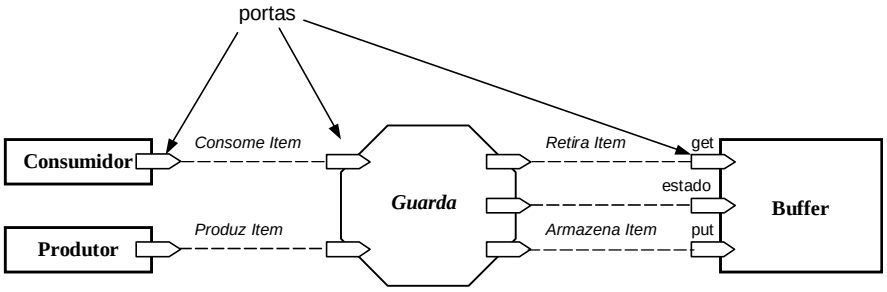

Figura 1 Arquitetura da aplicação Produtor-Consumidor com *buffer* limitado. Os componentes **Produtor**, **Consumidor** e **Buffer** agem como se o conector **Guarda** não existisse.

Os conector **Guarda** tem a função de intermediar a interação entre os componentes, tratando da sincronização de acesso ao **Buffer**, que é considerado um requisito não-funcional. **Guarda** retarda a execução de **Consumidor** se **Buffer** estiver vazio e retarda a execução de **Produtor** se **Buffer** estiver cheio.

O conector intercepta, analisa e encaminha as requisições feitas por **Produtor** e **Consumidor** através de pontos de acesso específicos, que são as portas definidas nos módulos e nos próprios conectores, e que podem ser observadas na figura 1. As portas representam serviços que poderão ser interceptados para o respectivo tratamento não-funcional. Tal interceptação é realizada pelos conectores, de forma transparente em relação aos componentes (módulos) envolvidos. Isto é, os componentes ignoram quaisquer mecanismos de intermediação existentes entre eles.

A descrição da arquitetura da figura 1 parte da definição dos tipos correspondentes aos componentes (**Produtor**, **Consumidor** e **Buffer**), do tipo correspondente aos conector (**Guarda**) e ainda, das portas presentes em componentes e conectores (***put***, ***get***, ***estado***). Estas definições são feitas através de uma ADL (*Architecture Description Language*). O código 1 mostra a descrição arquitetural da aplicação PCB utilizando-se a ADL Babel [Mal96], empregada no ambiente de suporte a arquiteturas R-RIO (*Reflective-Reconfigurable Interconnectable Objects*) [Lob99].

Os tipos para as portas, componentes e para o conector estão definidos entre as linhas 02 e 26. A partir da linha 27, há instruções para instanciar os tipos definidos (*instantiate*), interligar as instâncias entre si (*link*) e iniciar a aplicação (*start*).

```
1     module PCB {
2           port Put (Any Item) (void);
3           port Get (void) (Any Item);
4           port Estado (void) (String);
5
6           module Tipo_Buffer {
7                 inport Put;
8                 inport Get;
9                 inport Estado;
10          } buffer;
11
12          module Tipo_Produtor {
13                outport Put;
14          } produtor;
15
16          module Tipo_Consumidor {
17                outport Get;
18          } consumidor;
19
20          connector Guarda {
21                inport Put;
22                inport Get;
23                outport Put;
24                outport Get;
25                outport Estado;
26          } guarda;
27
28          instantiate buffer, produtor, consumidor, guarda;
29          link produtor to buffer by guarda;
30          link consumidor to buffer by guarda;
31
32    } pcb;
33
34    start pcb;
```

Código 1 Descrição da arquitetura da aplicação Produtor-Consumidor com *buffer* limitado (PCB).

A definição da configuração é abstrata, tornando-se concreta no momento em que as instâncias de componentes e conectores são criadas como objetos reais, e as respectivas portas de cada instância são interligadas, de acordo com a descrição da arquitetura, de modo que possa acontecer a troca de mensagens que representarão a requisição de serviços e as respostas obtidas. Assim, completa-se o processo de Programação da Configuração, ou seja, a implantação da arquitetura em questão.

Problema

A composição de uma aplicação parte do estabelecimento dos componentes envolvidos e das suas interações. A definição de um sistema em termos dos componentes e das interações entre estes componentes é chamada Arquitetura de um Sistema de Software [SDZ96], a qual é descrita através de uma ADL (*Architecture Description Language*). O arquiteto, ao utilizar uma ADL, seleciona e especifica os componentes do sistema e os conectores. Este processo é chamado Programação da Configuração, ou simplesmente, Configuração.

Para a aplicação do exemplo anterior, a arquitetura pode ser descrita como no código 1, com dois componentes e um conector, e suas respectivas portas. Instâncias dos componentes e do conector podem ser criadas e configuradas, formando-se a topologia da arquitetura.

Uma vez descrita a arquitetura, sua configuração deve ser implementada, ou seja, o mecanismo que concretiza a troca de mensagens (requisições e respostas) entre as instâncias de componentes e conectores deve ser estabelecido, conforme definido pela configuração arquitetural.

Forças

Uma solução para o problema da implementação de arquiteturas deve considerar o seguinte:

- As propriedades da arquitetura, como a reutilização de componentes e conectores, a separação de requisitos e a abstração, devem ser preservadas na sua implementação;

- Os componentes e conectores devem ser independentes quanto à sua definição. Esta ortogonalidade permite que modificações em componentes não afetem conectores, e vice-versa;

- Os componentes de um sistema não devem ter suas interfaces e comportamentos modificados com o objetivo de adicionar requisitos não-funcionais ao mesmo;

- Os componentes e conectores devem preferencialmente ser coesos quanto aos seus requisitos;

- A especificação da interface dos componentes deve tratar preferencialmente dos requisitos funcionais dos mesmos e deve ser claramente definida;

- Os conectores podem ser utilizados para estender a arquitetura quanto a novos serviços não-funcionais;

Solução

A figura 2 apresenta uma solução para a implementação da arquitetura de Produtor-Consumidor com *buffer* limitado. As classes **Produtor**, **Consumidor** e **Buffer Real** representam os componentes e a classe **Guarda** representa o conector da arquitetura. A programação da configuração está representada pela classe **Configuração (Descrição/Execução)**.

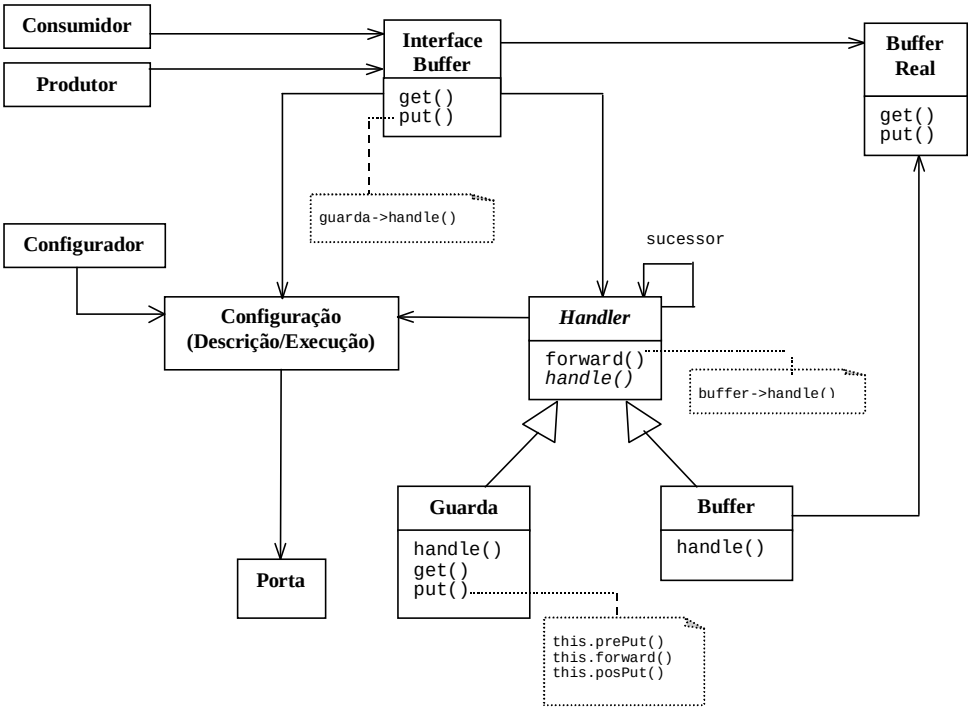

Figura 2 Aplicação Produtor-Consumidor com *buffer* limitado. A classe Configuração (Descrição/Execução) representa a configuração da aplicação.

Configurador é uma classe que define, de um modo geral, um mecanismo que interpreta as instruções de uma ADL, e a partir da descrição interpretada, define os tipos para componentes, conectores e portas presentes na aplicação, e também procede com a instanciação e inteligação dos mesmos, para permitir a execução da aplicação. Para a realização de tais coisas, são invocados os serviços da classe **Configuração**. Esse mecanismo de interpretação de uma ADL, que torna possível a implementação de uma arquitetura a partir de sua descrição, é o foco principal deste trabalho. Essa é a funcionalidade principal do ambiente de suporte a configuração desenvolvido e descrito mais adiante.

Configuração (Descrição/Execução), ou apenas **Configuração**, recebe as solicitações de **Configurador** para realizar a configuração de componentes e conectores, e iniciá-los. Ela mantém duas categorias de informações: de descrição e de execução da arquitetura. Informações de descrição referem-se à arquitetura descrita através de uma ADL (código 1), enquanto as informações de execução relacionam-se às instâncias – referências – dos componentes e conectores durante o processo de execução da aplicação.

A classe abstrata **Handler** é responsável pelo encadeamento entre conectores e componentes conforme a descrição arquitetural, disponibilizada por **Configuração**. **Handler** utiliza-se também das informações de execução da mesma classe, uma vez que necessita das referências aos objetos que representam componentes e conectores no espaço de execução da aplicação. No modelo, as classes **Guarda** e **Buffer** são encadeadas por **Handler**. **Buffer** representa a classe **Buffer Real** no encadeamento e possui uma referência a esta última.

As requisições invocadas por **Produtor** e **Consumidor** são interceptadas pela classe **Interface Buffer**, que possui a mesma interface de **Buffer Real**. **Interface Buffer** busca em **Configuração** a referência ao conector **Guarda** e invoca a operação *handle()* do mesmo.

Conforme a porta de entrada configurada no conector **Guarda**, *handle()* invoca a operação *get()* ou *put()*. Ambas operações tratam da sincronização (requisito não-funcional) e invocam *forward()*, responsável por dar seguimento ao encadeamento controlado por **Handler**. Na seqüência, **Buffer** tem sua operação *handle()* solicitada, a qual encaminha a requisição original para **Buffer Real**, finalizando o encadeamento.

Nesse modelo, as portas de entrada configuradas na descrição da arquitetura são mapeadas para operações. Por exemplo, as duas portas de entrada definidas no conector **Guarda** (figura 1) são mapeadas para as operações *get()* e *put()* da classe **Guarda**.

Algumas propriedades apresentadas:

- A reutilização de componentes e conectores é possível uma vez que suas funcionalidades estão separadas dos processos de interceptação e encaminhamento das requisições, os quais são realizados pelas classes **Interface Buffer** e **Handler** respectivamente.

- Modificações funcionais ou de interface de componentes não afetam os conectores, e vice-versa. Isto ocorre porque classes distintas estão definidas para componentes e conectores e a classe **Porta** mantém os pontos de acesso dos mesmos, independentemente das interfaces.

- Mapeamentos de portas de entrada para operações definidas nas classes, e de portas de saída para invocações a operações, são mantidos pela classe **Porta**. Isto torna componentes e conectores livres para terem suas interfaces modificadas de forma transparente, sem afetar os demais elementos da arquitetura implementada.

- A classe **Configuração** mantém informações da arquitetura configurada, tanto sua descrição quanto sua execução, e disponibiliza operações suficientes para a recuperação das mesmas por parte das classes **Interface Buffer** e **Handler**.

- Um conector tem a responsabilidade de rotear as mensagens para outros conectores e componentes interligados.

- Mais conectores podem ser definidos para refletir novos serviços não-funcionais.

Estrutura e Participantes

Os principais participantes no *design pattern* são:

- **Cliente**: requisita um serviço de **Servidor** invocando uma de suas operações (*serviço1()*, *serviço2()*, etc.) através de **Interface Servidor**.

- **Servidor**: contém as operações relacionadas aos serviços oferecidos.

- **Interface Servidor**: responsável pela interceptação das invocações ao Servidor. Ele solicita de **Configuração** a referência do conector configurado e invoca a operação *handle()* deste conector.

- **Handler**: classe abstrata responsável por encadear os conectores e componentes. O encadeamento é feito através da operação *forward()* que utiliza-se de **Configuração** para encontrar o sucessor, e invoca *handle()* do mesmo, seja ele conector ou componente.
- **Conector Handler**: encapsula preferencialmente aspectos não diretamente relacionados à funcionalidade de **Cliente** e **Servidor** (além de quaisquer mecanismos auxiliares), através da implementação do método *handle*(). Representa os conectores no nível da configuração.

- **Servidor Handler**: representa o **Servidor** no encadeamento mantido por **Handler**. Ele encaminha ao **Servidor**, através de sua operação *handle*(), a requisição vinda originalmente de **Cliente**.

- **Configurador**: permite a definição dos tipos, instâncias e das ligações entre componentes e conectores, conforme determinada ADL. Funciona como um interpretador, recebendo instruções de configuração e as executando junto à classe **Configuração**.

- **Configuração (Descrição/Execução)** ou **Configuração**: mantém a configuração da aplicação, contendo informações da descrição e da execução da arquitetura. A descrição é recebida pela classe **Configurador** e contém tipos para componentes, conectores e portas, além de instâncias e ligações. A execução mantém as referências aos componentes e aos conectores. Todos os participantes do *design pattern* podem consultar **Configuração** sempre que necessitarem de informações a respeito da estrutura da aplicação. **Configuração** mantém referências às portas configuradas através de associação com **Porta**.

- **Porta**: mantém as portas configuradas junto aos conectores e componentes e permite o mapeamento das mesmas para declarações de operações (assinaturas). Qualquer acesso a **Porta** é feito através de **Configuração**.

No exemplo de aplicação citado anteriormente, as classes **Produtor** e **Consumidor** têm o papel de **Cliente**, a classe **Buffer Real** tem o papel de **Servidor**, a classe **Interface Buffer** tem o papel de **Interface Servidor**, a classe **Guarda** tem o papel de **Conector Handler** e a classe **Buffer** tem o papel de **Servidor Handler**.

A figura 3 apresenta uma tabela, na qual é feito um resumo de todas as classes participantes, suas responsabilidades e colaboradores.

Classes	Responsabilidades	Colaboradores
Cliente	- Utiliza a interface fornecida por Interface Servidor para requisitar um serviço particular;	**Interface Servidor**
Servidor	- Implementa um ou mais serviços particulares;	-
Interface Servidor	- Fornece a interface do servidor aos clientes; - Recupera a referência do conector interligado ao cliente no nível da configuração, ou do próprio servidor, caso não haja nenhum conector envolvido; - Repassa a solicitação do cliente ao conector recuperado, ou ao servidor, no caso de não haver conectores; - Retorna ao cliente resposta oriunda do servidor;	**Servidor** **Configurador** **Conector Handler**
Handler	- Serve como classe abstrata base para o servidor e para os conectores; - Realiza o encadeamento de conectores e componentes a partir da configuração estabelecida;	**Configuração**
Conector Handler	- Implementa serviços relacionados à funcionalidade de um conector; - Invoca uma de suas operações correspondente à porta de entrada requisitada; - Requisita, junto ao próximo conector ou componente configurado, a operação correspondente a uma de suas portas de saída. Essa requisição é feita através da operação *forward()* da classe Handler.	**Configuração**
Servidor Handler	- Encaminha ao servidor a requisição vinda originariamente do cliente;	**Servidor**
Configurador	- Define a arquitetura da aplicação a partir de uma ADL; - Recebe instruções a partir de uma determinada ADL e invoca serviços da classe Configuração para executá-las;	**Configuração**
Configuração	- Fornece a configuração estabelecida entre componentes e conectores; - Disponibiliza serviços para configurar componentes e conectores e iniciar a aplicação; - Mantém a descrição da arquitetura, bem como informações quanto à execução da mesma;	**Porta**
Porta	- Fornece as portas configuradas de conectores e componentes com suas respectivas assinaturas;	-

Figura 3 Tabela que resume as Classes, Responsabilidades e Colaboradores.

O diagrama de classes na figura 4 ilustra a estrutura do *design pattern* e um possível diagrama de instâncias em tempo de execução aparece na figura 5.

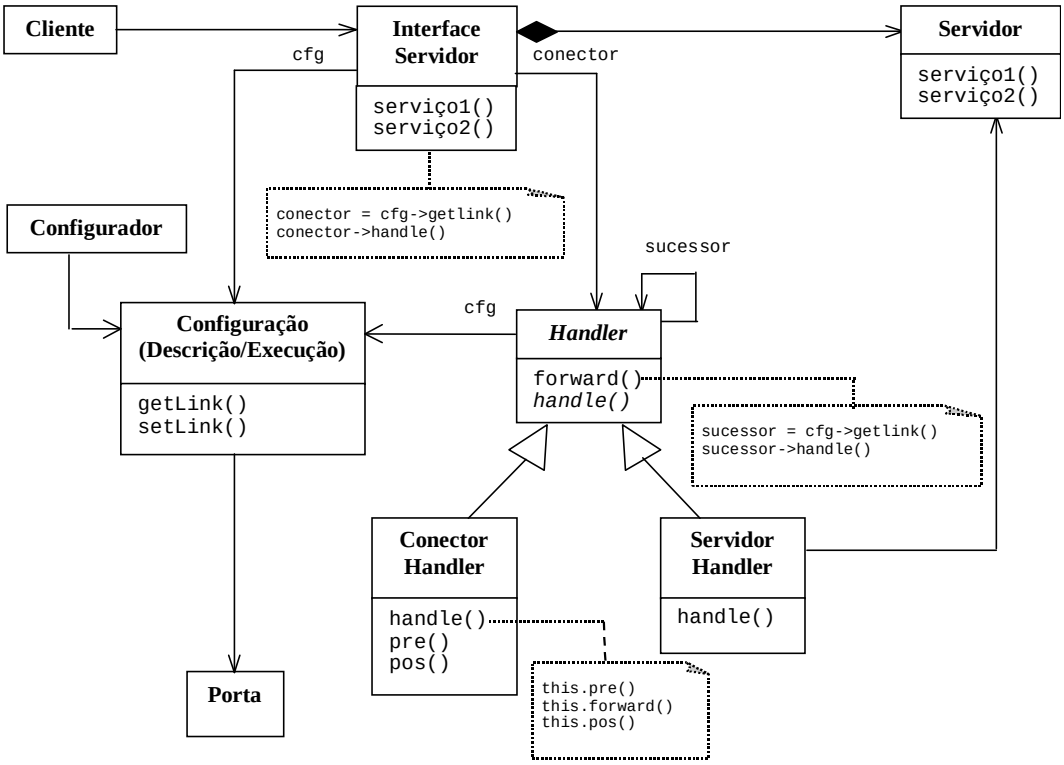

Figura 4 Diagrama de classes do *design pattern Architecture Configurator*

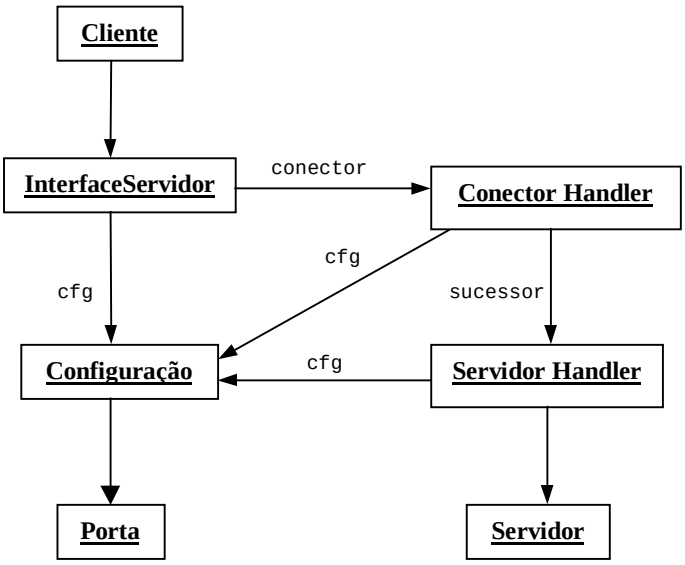

Figura 5 Um possível diagrama de objetos de *Architecture Configurator*.

Colaborações

Uma vez descritos tipos para portas, componentes e conectores através de uma ADL, torna-se necessário estabelecer, ainda no nível de descrição arquitetural, a estrutura do sistema de software. É a partir da descrição das interligações entre componentes e conectores que inicia-se a colaboração entre os participantes do *design pattern Architecture Configurator*.

A colaboração é composta basicamente de duas fases: (i) estabelecimento da configuração arquitetural; (ii) implementação da configuração arquitetural.

A primeira fase é iniciada pelo **Configurador** e descreve as interligações entre componentes e conectores. **Configurador** interpreta a descrição arquitetural, identificando as instâncias de componentes e conectores, e procedendo com sua instanciação. A referência às instâncias fica armazenada na classe **Configuração**. Após isso, **Configurador** identifica as ligações previstas entre componentes e/ou conectores, e a cada instrução referente a uma ligação, é invocada na classe **Configuração** a operação para realizar tal ligação. Ainda nesta fase, as instâncias de componentes e conectores são iniciados, através de instrução apropriada.

A segunda fase trata do mecanismo de interceptação e encaminhamento de requisições oriundas de componentes clientes. Este mecanismo compreende basicamente a organização dos componentes e conectores, realizada pela classe **Handler**, conforme a configuração estabelecida na primeira fase.

A primeira fase está retratada na figura 6. O diagrama de seqüência ilustra a interligação de **Cliente** a **Servidor Handler** através de **Conector Handler** e a iniciação dos mesmos. Instâncias de componentes e conectores são mantidas pela classe **Configuração**.

Note-se que **Servidor** não aparece no diagrama de seqüência da primeira fase, uma vez que é iniciado pelo objeto correspondente à classe **Servidor Handler**. Para cada classe que oferece serviços (**Servidor**) há uma classe respectiva associada (**Servidor Handler**), facilitando o processo de encadeamento de objetos realizado pela classe **Handler** do *design pattern*. Esta solução é baseada no *pattern Object Recursion* apresentado em [HFR99].

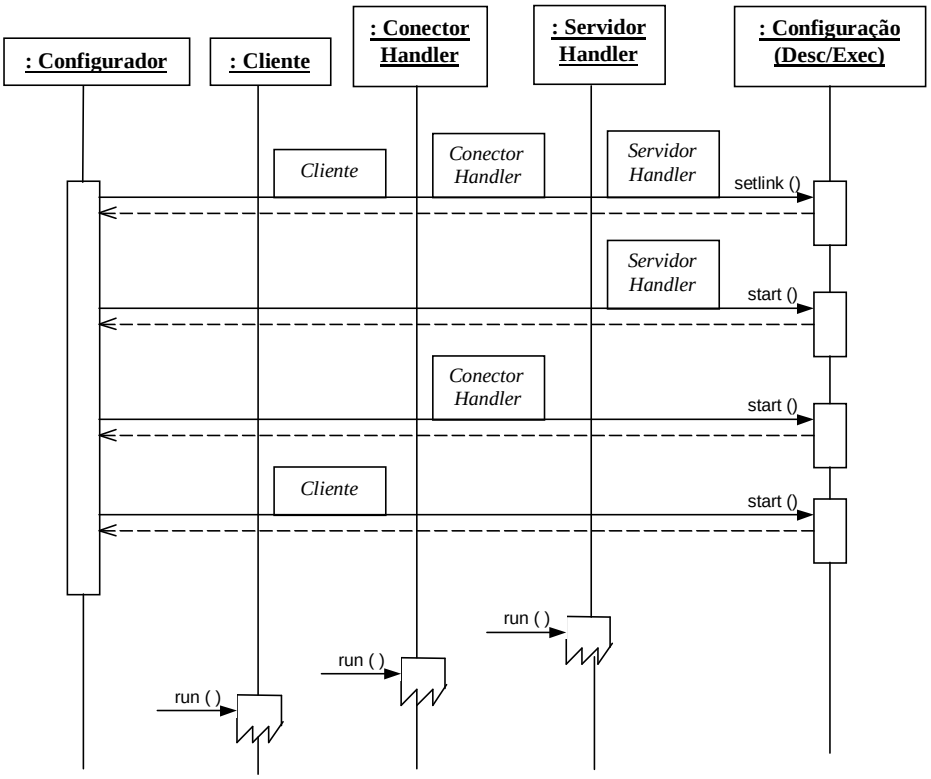

Figura 6 Diagrama de Seqüência (Colaboração). Primeira fase.

A iniciação de **Cliente** dará princípio à segunda fase da colaboração, retratada na figura 7. A seqüência de colaboração inicia-se a partir da invocação, por parte da instância de **Cliente**, a um serviço (*servico1()*, na figura) oferecido por um **Servidor**. Todas as invocações a partir do Cliente são interceptadas por um objeto **Interface Servidor**. Este encaminha a invocação através da cadeia de conectores e componentes, representados por instâncias de **Conector Handler** e **Servidor Handler**. Cada uma das instâncias de **Conector Handler** representa, preferencialmente, um aspecto não-funcional, implementado por um conector. Ou seja, a cada conector configurado, deve estar associada uma instância de **Conector Handler**.

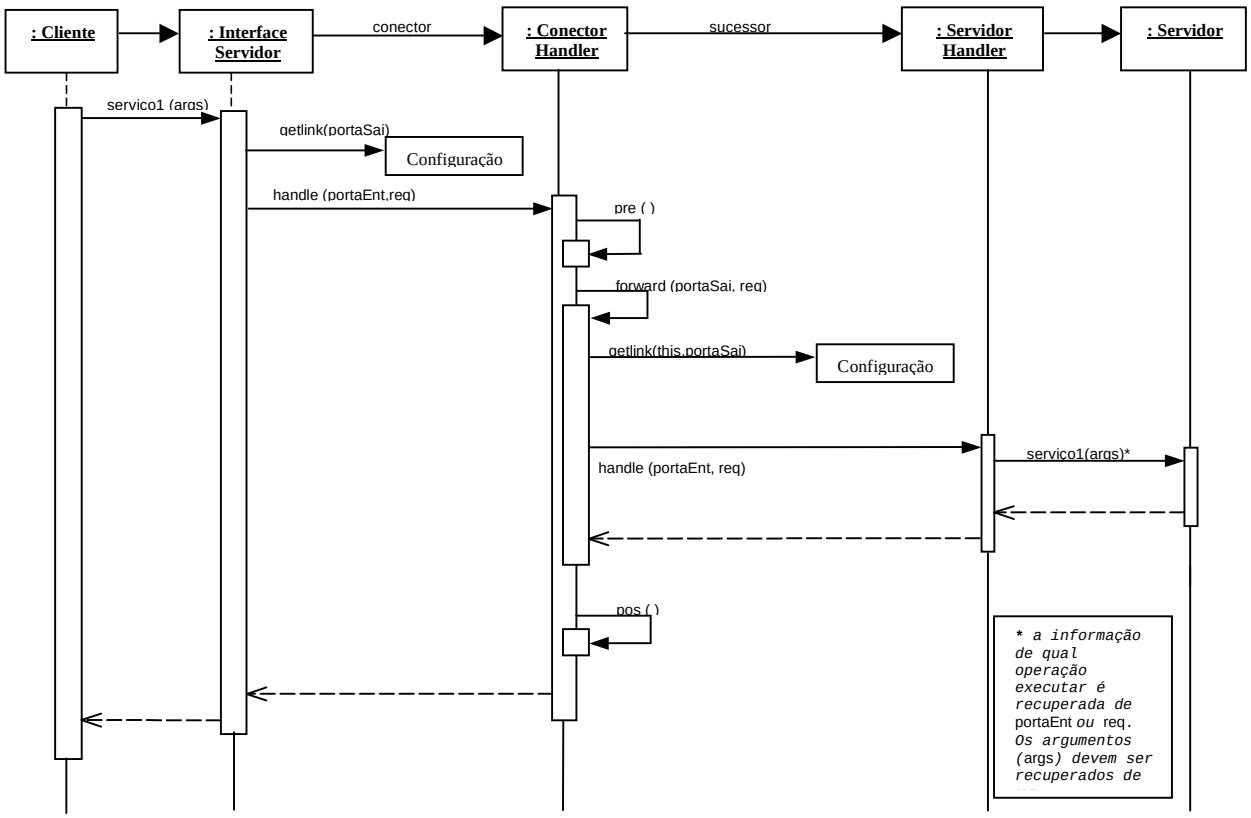

Figura 7 Diagrama de Seqüência (Colaboração). Segunda fase.

A invocação de um serviço corresponde a uma das portas de saída configuradas para o componente **Cliente**. A assinatura da porta é passada a **Interface Servidor**, que, de posse disso, obtém em **Configuração** a referência ao primeiro conector ligado a **Cliente**, e a porta de entrada ligada à sua porta de saída em questão. Feito isso, o método *handle()* de conector é invocado, com argumentos referentes à porta de entrada e à requisição realizada por Cliente.

O método *handle()* começa sua operação invocando o método *pre()* do respectivo conector, responsável pelos aspectos não-funcionais que devem ser executados antes do encaminhamento da requisição para o próximo conector/componente configurado. Após a execução de *pre()*, o fluxo deve seguir por alguma porta de saída do conector. Definida a porta de saída, a operação *forward()* do conector é invocada. Tal operação utiliza a classe **Configuração** para obter a referência ao próximo componente/conector configurado, de modo análogo ao explicado anteriormente, e a requisição segue no encadeamento.

O final do encadeamento ocorre quando **Servidor Handler** é encontrado e tem sua operação *handle()* requisitada. Esta operação tem funcionamento análogo à de **Conector Handler**, porém não invoca *forward()*, e sim concretiza a requisição junto a **Servidor**, invocando o método que implementa o serviço desejado.

Uma explicação mais detalhada sobre o funcionamento dos métodos *handle()* e *forward()* presentes nos *handlers* que compõem o encadeamento de componentes e conectores, será dada mais adiante, no capítulo que explica detalhes de implementação para a utilização do ambiente de configuração proposto por este trabalho.

No contexto das classes **Interface Servidor** e **Handler**, portas de saída são representadas por invocações de operações, e portas de entrada são representadas por operações declaradas na interface de componentes e /ou conectores (assinaturas).

É importante ressaltar que podem ser criadas várias classes que implementam aspectos não-funcionais, e também várias instâncias de cada uma delas. Em outras palavras, seria possível ter classes **Conector Handler 1**, **Conector Handler 2**, **Conector Handler 3**, etc., cada qual com uma ou mais instâncias. A mesma observação vale para as classes **Cliente**, **Servidor** e **Servidor Handler**.

Deve ser levado em conta o fato de que cada classe **Servidor** criada deve ter associada a si uma classe **Interface Servidor** específica.

Conseqüências

A utilização do *design pattern Architecture Configurator* traz as seguintes consequências:

- ***conectores independentes***: cada conector pode ser construído levando-se em conta um aspecto não-funcional diferente e podem ser inter-relacionados sem que conheçam a configuração da aplicação.

- ***transparência***: componentes requisitam e fornecem serviços transparentemente em relação aos conectores configurados.

- ***responsabilidade flexível***: conectores têm amplo poder de manipular e analisar as informações por eles recebidas e podem encaminhá-las ou não a outro conector ou ao **Servidor**. A operação *forward()* busca o sucessor na seqüência de conectores e pode ser invocada ou não pelos mesmos, conforme os requisitos da aplicação.

- ***conectores adaptáveis***: os conectores podem ser concebidos independentemente dos componentes os quais intermediarão. Uma vez implementados e configurados em uma determinada aplicação, não necessitam de alterações em decorrência de alguma modificação da interface de um dos componentes intermediados.

- ***conectores podem ser usados para estender aplicações já existentes***: este *design pattern*, ao tornar possível a organização dos conectores, facilita a implementação de novos requerimentos que surgem na aplicação. Adaptações que seriam necessárias aos componentes para suprir tais requerimentos podem ser realizadas em novos conectores, os quais podem ser construídos e configurados independentemente de outros existentes.

- ***conectores genéricos***: um determinado conector pode ser desenvolvido independentemente da quantidade de portas dos componentes que irá intermediar, e da diversidade de assinaturas das mesmas.

Patterns Relacionados

Interface Servidor possui a mesma interface de **Servidor**, com o objetivo de controlar o acesso ao mesmo. O controle de acesso ao **Servidor** foi baseado no *design pattern Proxy* [GHJ95], sendo que **Servidor** corresponde a *RealSubject* de *Proxy* e **Interface Servidor** corresponde à classe *Proxy* do mesmo *design pattern*. O **Servidor** desconhece a existência de **Interface Servidor**.

A interligação dos conectores e componentes como desenhada na estrutura de *Architecture Configurator* segue o *design pattern Object Recursion* [HFR99], no qual sua classe abstrata **Handler** corresponde a **Handler** de *Architecture Configurator*, suas classes *Terminator* e *Recurser* correspondem respectivamente a **Servidor Handler** e **Conector Handler** e, finalmente, sua classe *Initiator* corresponde a **Interface Servidor**. *Object Recursion* é utilizado no *Chain of Responsibility* de [GHJ95].

O *design pattern Component Configurator*, disponível em [SSR00], permite que uma aplicação configure seus componentes (***link*** e ***unlink***) em tempo de execução, sem necessidade de quaisquer modificações e/ou recompilações dos mesmos. O processo é feito separando-se a implementação dos componentes de suas operações de controle, tais como *init(), fini(), suspend(), resume()*. O *pattern* possui duas classes importantes: *Component Repository* e *Component Configurator*. Ambas classes têm funcionalidades semelhantes à Configuração e Configurador de *Architecture Configurator*.

A diferença básica entre *Component Configurator* e *Architecture Configurator* relaciona-se aos aspectos arquiteturais. O primeiro *pattern* contempla o espaço de endereços da aplicação, armazenando as referências dos componentes na classe *Component Repository*, sem ater-se à configuração arquitetural da mesma. *Architecture Configurator*, por sua vez, mantém a descrição da

arquitetura e informações de execução da mesma. Assim, *Architecture Configurator* organiza os componentes e conectores de acordo com a descrição arquitetural mantida pela classe Configuração.

Entretanto, *Architecture Configurator* pode ser empregado em conjunto com *Component Configurator*, com o objetivo de permitir a reconfiguração de componentes e conectores da aplicação em tempo de execução.

Interceptor [SSR00] é um *architectural pattern* que permite a adição de serviços a determinado *framework*, de forma transparente, tornando-o extensível. Este *pattern* relaciona-se ao *design pattern Architecture Configurator* no sentido de ambos permitirem a extensão da aplicação com serviços não previstos nos seus componentes básicos. Entretanto, o *Interceptor* trata da extensão de *frameworks*, que são estruturas inacabadas que servem de base para o desenvolvimento de aplicações, enquanto que o *Architecture Configurator* permite a extensão de arquiteturas mais genéricas, através da incorporação de conectores que podem encapsular serviços não-funcionais à aplicação.

Outro *design pattern* relacionado é o *Facade* [GHJ95], o qual pode auxiliar na composição de **Configuração** e **Porta**. No modelo da figura 4, **Configuração** serve de *facade* para a funcionalidade de **Porta**, com o objetivo de distinguir as funções de **Porta** e **Configuração**. Entretanto, uma terceira classe pode ser definida para servir como *facade* para **Configuração** e **Porta** com o objetivo de tornar unificada a interface de acesso a ambas classes, simplificando a implementação de conectores e componentes.

O *design pattern Architecture Configurator* pode ser empregado na implementação do *architectural pattern Reflection* [HFR99]. Este *pattern* separa a arquitetura de um sistema em dois níveis: nível base (*base level*) e meta-nível (*meta level*). O primeiro define a lógica da aplicação e o segundo mantém informações a respeito da própria estrutura e comportamento do sistema. O meta-nível consiste de metaobjetos (*metaobjects*) com uma interface que permite ao nível base acesso ao meta-nível. As classes **Cliente** e **Servidor** de *Architecture Configurator* podem compor o nível base, enquanto os conectores e suas respectivas portas podem compor o meta-nível, fazendo o papel de metaobjetos.

Usos Conhecidos

O *design pattern* apresentado tem seu uso no ambiente de suporte R-RIO [Lob99]. No R-RIO os componentes são objetos instanciados a partir de classes escritas em Java e conectores podem encapsular protocolos de comunicação e aspectos relacionados à interação [SLL99] entre os componentes. R-RIO permite a instanciação de componentes em um ambiente distribuído e mantém a configuração da aplicação implementada em gerentes localizados nos *hosts* do sistema distribuído. A configuração é alimentada por um interpretador central, ao qual recebe instruções baseadas na ADL Babel.

Os gerentes do R-RIO têm funcionalidade equivalente à classe **Configuração** do *Architecture Configurator*, mantendo informações da descrição da configuração arquitetural, bem como da execução do sistema. O interpretador, por sua vez, tem associação com a classe **Configurador**.

O conceito de porta de entrada e saída com o respectivo mapeamento para operações, é usado pelo R-RIO, tanto para componentes quanto para conectores. Outro aspecto do R-RIO refere-se ao uso do mecanismo de reflexão estrutural da linguagem Java para gerar de forma automática um *proxy* para o componente **Servidor**, nos mesmos moldes da classe **Interface Servidor** de *Architecture Configurator*.

O conceito de conector, como colocado neste *design pattern*, aparece em ADLs como ACME [GMW97], Babel [Mal96], C2 [Med99], UniCon [SDZ96] e *Wright* [AG97]. Em todas elas existe a distinção entre o tipo de conector e a instância de conector.

Propriedades em relação às portas de componentes e conectores, representadas pela classe **Porta**, aparecem em UniCon na forma de listas chamadas Listas de Propriedades. Portas de componentes são denominadas *players* e portas de conectores são denominadas *roles* em UniCon.

O conceito de encaminhamento/roteamento de mensagens atribuído ao conector aparece na linguagem C2. O conector C2 tem a responsabilidade primária de realizar o roteamento e o *broadcast* de mensagens e, secundariamente a definição e implementação de políticas de filtragem de mensagens. Tais políticas têm paralelo com a interceptação e manipulação descritas neste *design pattern*.

Conclusão

Este texto apresentou o *design pattern Architecture Configurator*, proposto para servir de base à implementação da configuração de aplicações.

Arquiteturas definidas em alguma ADL podem ser implementadas através deste *design pattern*, uma vez que ele emprega características próprias dos conectores e características inerentes ao processo de configuração de componentes e conectores. O *pattern* faz a leitura da arquitetura e a implementa utilizando propriedades de conectores como a interceptação, manipulação e encaminhamento de mensagens ou requisições. As interconexões entre componentes e conectores são interpretadas conforme seus elos de ligação definidos pelas portas.

Cada componente ou conector participante da arquitetura pode ser implementado de forma autônoma e integrado de acordo com a configuração arquitetural.

A solução descrita no *Architecture Configurator* não apresenta novidades conceituais quanto às propriedades de configuração, de conectores ou mesmo de arquiteturas de software, mas reforça estes pontos, propondo um modelo-base sobre o qual a configuração entre componentes e conectores pode ser realizada.

REFERÊNCIAS

[AG97] R. Allen, D. Garlan. *A Formal Basis for Architectural Connection*. ACM Transactions on Software Engineering and Methodology, vol. 6, no. 3, pág. 213-249, julho 1997.

[Car01] Carvalho, S. *Um Design Pattern Para a Configuração de Arquiteturas de Software*. Dissertação de Mestrado. IC/UFF, Niterói-RJ, maio de 2001.

[GHJ95] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, EUA, 1995.

[GMW97] D. Garlan, R. Monroe, D. Wile. *Acme: An Architecture Description Interchange Language*. CASCON'97. Novembro, 1997.

[HFR99] N. Harrison, B. Foote, H. Rohnert. *Pattern Languages of Program Design 4*. Software Pattern Series. Addison-Wesley, 1999.

[Lob99] M. Lobosco. *Um Ambiente para Suporte à Construção e Evolução de Sistemas Distribuídos*. Dissertação de Mestrado. IC/UFF. Março, 1999.

[Mal96] V. V. Malucelli. *Babel – Construindo Aplicações por Evolução*. Dissertação de Mestrado. DEE / PUC-RJ. Fevereiro 1996.

[Med99] N. Medvidovic. *Architecture-Based Specification–Time Software Evolution*. Tese de Doutorado (PhD). Universidade da Califórnia, Irvine, 1999.

[SDZ96] M. Shaw, R. Deline, G. Zelesnik. *Abstractions and Implementations for Architectural Connections*. Third International Conference on Configurable Distributed Systems. Maio 1996.

[SLL99] A. Sztajnberg, M. Lobosco, O. Loques. *Configurando Protocolos de Interação na Abordagem R-RIO*. Simpósio Brasileiro de Engenharia de Software. Florianópolis, Santa Catarina. 1999.

[SSR00] D. Schmidt, M. Stal, H. Rohnert, F. Buschmann. *Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects*. Volume 2. John Wiley & Sons, 2000.

[Sun00] Sun Microsystems. *Java 2 Plataform, Standard Edition Documentation*. http://java.sun.com/products/jdk/1.3/docs/index.html. Maio 2000.