UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE CIÊNCIAS EXATAS E DA TERRA
DEPARTAMENTO DE INFORMÁTICA E MATEMÁTICA APLICADA
PROGRAMA DE PÓS-GRADUAÇÃO EM SISTEMAS E COMPUTAÇÃO

DISSERTAÇÃO DE MESTRADO

UM AMBIENTE MULTI-MIDDLEWARE PARA DESENVOLVIMENTO DE APLICAÇÕES DISTRIBUÍDAS

André Gustavo Duarte de Almeida

Natal / RN
Fevereiro 2008

Universidade Federal do Rio Grande do Norte
Centro de Ciências Exatas e da Terra
Departamento de Informática e Matemática Aplicada
Programa de Pós-Graduação em Sistemas e Computação

Um Ambiente Multi-Middlware para Desenvolvimento de Aplicações Distribuídas

Dissertação submetida ao Programa de Pós-Graduação em Sistemas e Computação do Departamento de Informática e Matemática Aplicada da Universidade Federal do Rio Grande do Norte como requisito para a obtenção do grau de Mestre em Sistemas e Computação.

André Gustavo Duarte de Almeida

Prof. Dra. Thais Vasconcelos Batista
Orientadora

Natal – RN
Fevereiro de 2008

Catalogação da Publicação na Fonte. UFRN / SISBI / Biblioteca Setorial
Especializada do Centro de Ciências Exatas e da Terra – CCET.

Almeida, André Gustavo Duarte de.

Um ambiente multi-middleware para desenvolvimento de aplicações distribuídas / André Gustavo Duarte de Almeida. – Natal, 2008.

87 f. : il.

Orientador: Profa. Dra. Thais Vasconcelos Batista.

Dissertação (Mestrado) – Universidade Federal do Rio Grande do Norte. Centro de Ciências Exatas e da Terra. Departamento de Informática e Matemática Aplicada. Programa de Pós-Graduação em Sistemas e Computação.

1. Middleware - Dissertação. 2. Desenvolvimento baseado em componentes - Dissertação. 3. Interoperabilidade – Dissertação. 4. Seleção dinâmica. I. Batista, Thais Vasconcelos. II. Título

RN/UF/BSE-CCET CDU: 004.75

Agradecimentos

Agradeço primeiramente a minha família, minhas irmãs Melise e Milena, minha Tia Carmosa, minha super-protetora babá Glória, minha namorada Isabella, pelo apoio dados em todos esses anos de estudo, em especial minha mãe, que não está mais entre nós, porém que fez de tudo para me dar uma educação de qualidade e mesmo depois de deixar esse mundo continuou me enviando a luz para que eu pudesse buscar os meus sonhos.

A minha orientadora Thais, que sempre acreditou no meu trabalho, desde tempo da iniciação cientifica, por ter sempre encontrado tempo em sua agenda para me ajudar, pelas oportunidades que me ofereceu ao longo desse tempo e pelas cobranças que permitiram a conclusão desse trabalho no tempo definido, pelo menos quase no tempo.

A professora Flávia que aceitou de prontidão a participação na banca e que jogou uma nova luz no trabalho, mostrando as potencialidades que o mesmo possui e com as sugestões fornecidas enriquecer o trabalho.

Ao pessoal do projeto de modelagem, Lirisnei, Fabiola e Fred, pelos momentos de debate acadêmico, que muitas vezes abriram novas perspectivas para as minhas decisões.

A Nélio Cacho por ter construído boa parte das ferramentas utilizadas na implementação CORBA, uma vez que a partir delas surgiu a idéia de construir um ambiente multi-middleware.

Ao pessoal do CEFET-UNED-Ipanguaçu, pelo apoio dado para que fosse possível terminar esse texto, permitindo que eu pudesse estar me ausentando da Unidade para melhor me dedicar a esse trabalho.

Por fim a todos que de uma maneira ou de outra contribuíram de forma direta ou indireta ao trabalho, agradeço de coração.

RESUMO

Este trabalho apresenta a proposta de um ambiente multi-middleware para desenvolvimento de aplicações distribuídas, o qual abstrai diferentes plataformas de middleware subjacentes. O trabalho descreve: (i) a arquitetura de referência especificada para o ambiente, (ii) uma implementação que valida tal arquitetura integrando CORBA e EJB, (iii) um estudo de caso ilustrando o uso do ambiente, (iv) a análise de desempenho. O ambiente proposto permite alcançar interoperabilidade de plataformas de middleware, promovendo o reuso de componentes de diferentes plataformas de forma transparente para o desenvolvedor e sem perdas expressivas em termos de desempenho. Na implementação desenvolvemos um plugin para IDE Eclipse que permite que os eventuais desenvolvedores obtenham maior produtividade ao desenvolver aplicações usando o ambiente.

Palavras-Chaves: Middleware, Desenvolvimento Baseado em Componentes, Interoperabilidade, Seleção dinâmica, EJB, CORBA.

ABSTRACT

This work presents a proposal of a multi-middleware environment to develop distributed applications, which abstracts different underlying middleware platforms. This work describes: (i) the reference architecture designed for the environment, (ii) an implementation which aims to validate the specified architecture integrating CORBA and EJB, (iii) a case study illustrating the use of the environment, (iv) a performance analysis. The proposed environment allows interoperability on middleware platforms, allowing the reuse of components of different kinds of middleware platforms in a transparency away to the developer and without major losses in performance. Also in the implementation we developed an Eclipse *plugin* which allows developers gain greater productivity at developing distributed applications using the proposed environment.

Keywords: Middleware, Component Based Development, Interoperability, Dynamic Selection, EJB, CORBA.

Sumário

1 Introdução 10
1.1 Objetivos 12
1.2 Estrutura do Trabalho 14
2 Conceitos Básicos 15
2.1 CORBA 15
2.2 EJB 17
2.3 Aspectos de Interoperabilidade entre EJB e CORBA 20
2.4 LUA 24
2.5 Eclipse 25
2.6 LuaSpace 29
3 Arquitetura 32
3.1 Arquitetura de referência 32
3.2 Camada Multi-Middleware 35
3.3 Seleção Dinâmica 38
3.4 Bridge 40
3.5 Binding 43
4 Implementação 44
4.1 Implementação da Camada Multi-Middleware 44
4.2 Bridge e Mecanismo de seleção dinâmica para CORBA 46
4.3 Bridge e Mecanismo de seleção dinâmica para EJB 51
4.4 Mecanismos de Binding 55
4.4.1 LuaOrb 55
4.4.2 LuaJava 56
4.5 Plugin Eclipse 57
4.5.1 Editor 59
4.5.2 Núcleo de Execução 60
4.5.3 Navegador de Interfaces 62
4.6 Desenvolvendo clientes para o ambiente multi-middleware 63
4.6.1 Clientes escritos em Lua 63
4.6.2 Clientes não escritos em Lua 65
Estudo de caso e Análise de Desempenho 66
4.7 Estudo de Caso 66
4.8 Análise de Desempenho 70

5 Trabalhos Relacionados 73
5.1 Web Services 73
5.2 Gerador Automático de Stubs 75
5.3 J-Integra Espresso 75
5.4 IIOP .Net 77
6 Conclusões 78
6.1 Considerações Finais 78
6.2 Trabalhos Futuros 80
7 Referências 82

Lista de Figuras

Figura 1 - Arquitetura CORBA [em www.cs.wustl.edu/~schmidt/corba-overview.html] 16
Figura 2 - Interface Hello escrita em IDL 17
Figura 3 - Arquitetura do EJB 18
Figura 4 - Bean Calculator na especificação 3.0 19
Figura 5 - Bean Calculator na especificação 2.1 20
Figura 6 - Invocação de EJB através de clientes CORBA 22
Figura 7 - Descritor de implantação do JBOSS 23
Figura 8 - Trecho de código Lua 25
Figura 9 - Plataforma Eclipse Workbench 26
Figura 10 - Arquitetura da plataforma Eclipse 27
Figura 11 - Arquitetura do LuaSpace EPlus 29
Figura 12 - Arquitetura do ambiente de desenvolvimento multi-middleware 33
Figura 13 - Diagrama de classes da Camada Multimiddleware 35
Figura 14 - Diagrama de Seqüência de Utilização da Camada Multi-middleware 38
Figura 15 - Diagrama de Classes do Mecanismo de Seleção Dinâmica 38
Figura 16 - Diagrama de Classes – Mapeamento do Serviço 40
Figura 17 - Diagrama de Classes Bridge – Lado do cliente 42
Figura 18 - Bridge Cliente/Servidor 42
Figura 19 - Implementação da camada Multi-Middleware 45
Figura 20 - Arquitetura do *Discovery Service* 47
Figura 21 - Código de busca de componentes 49

Figura 22 - Implementação do método execute 50
Figura 23 - Código do método *inspectMethod*..... 52
Figura 24 - Representação XML de um Bean 53
Figura 25 - Diagrama de seqüência Seleção dinâmica EJB 55
Figura 26 - Utilização do *binding* LuaOrb 56
Figura 27 - Utilização do binding LuaJava 56
Figura 28 - Trecho de código XML para extensão do editor 58
Figura 29 - Ambiente de Desenvolvimento no Eclipse 59
Figura 30 - Diagrama de classes da implementação do editor..... 59
Figura 31 - Diagrama de classes do núcleo de execução 61
Figura 32 - Configuração de Execução 62
Figura 33 - Discovery View..... 63
Figura 34 - Passos para utilização do ambiente utilizando Lua 64
Figura 35 - Diagrama de classes da aplicação bancária 66
Figura 36 - Especificação IDL da implementação CORBA 67
Figura 37 - Interface Remote do Bean Bank 68
Figura 38 - Assistente para adicionar novo arquivo Lua 69
Figura 39 - Código da aplicação Principal 70
Figura 40 - Gráfico de desempenho(Número de ObjetosxTempo de execução) 72
Figura 41 - J-Integra Espresso[em http://j-integra.intrinsyc.com/support/espresso/doc/] 76

1 INTRODUÇÃO

Plataformas de *middleware* [Bernstein 96] oferecem uma infra-estrutura para facilitar o desenvolvimento baseado em componentes definindo formas padrão para declaração de interfaces de componentes e para comunicação entre eles. A plataforma CORBA (*Common Object Request Broker Architecture*) [OMG 04] tem se destacado entre as demais por ser uma especificação aberta, independente de fabricante e linguagem. Ao endereçar a possibilidade de tratar aplicações escritas em diferentes linguagens, CORBA poderia ter se tornado a plataforma de middleware número um no mundo em termos de utilização, porém, a falta de uma implementação de referência e alto grau de complexidade associado à implementações CORBA, a torna pouco utilizada nos dias de hoje. EJB (*Enterprise Java Beans*) [SUN 03] é uma especificação que define uma arquitetura para o desenvolvimento de componentes utilizando a linguagem *Java*. A grande popularidade de Java, e o fato de ser fornecida uma implementação de referência, tornaram essa especificação uma das mais utilizadas no mundo.

Atualmente uma das principais metas no desenvolvimento de aplicações distribuídas é a interoperabilidade, principalmente devido à grande diversidade de aplicações legadas, base de dados, linguagens de programação e sistemas operacionais que precisam operar em conjunto. O uso de plataformas de middleware como as supracitadas garante um nível de interoperabilidade na qual aplicações distribuídas implementadas em diferentes linguagens e executando em diferentes plataformas de hardware e sistemas operacionais podem interagir, desde que haja o suporte subjacente de uma mesma plataforma de middleware. Entretanto, apesar de tais plataformas procurarem oferecer alguma interoperabilidade entre si, nenhuma delas provê uma maneira simples e eficiente para interagir uma com a outra.

Interoperabilidade em nível de diferentes plataformas de middleware é importante, pois cada plataforma usa diferentes modelos de objetos, cada qual com características e benefícios específicos, e podendo interoperar com determinados sistemas legados. No entanto, há muitas situações em que componentes que seguem diferentes modelos precisam ser combinados para

prover uma solução completa para uma aplicação. Reusabilidade [Frakes 93] é o potencial que um componente possui de ser reusado em mais de um contexto com pouca ou nenhuma adaptação [Katz 93]. O amplo reuso de componentes apenas é possível se houver suporte a coexistência de componentes de diferentes plataformas de *middleware* em uma mesma aplicação e de forma transparente para o programador, de forma a conferir facilidade de reuso.

A especificação CORBA promove o reuso através do suporte a interoperabilidade, especificando uma interface comum para componentes, escrita na linguagem IDL (*Interface Definition Language*). Tal interface é independente de linguagem de programação, e pode ser acessada por qualquer cliente escrito em qualquer linguagem que tenha o *binding* para CORBA. Componentes de outras plataformas de middleware não são tão reusáveis. Por exemplo, o EJB trabalha apenas com a linguagem Java, e no .Net os componentes são especificados para possibilitar a comunicação com outros componentes da própria plataforma. A sistemática de funcionamento do CORBA poderia ser aplicada nesses outros *middlewares* para dar suporte a interoperabilidade. Porém, mesmo com o suporte de CORBA, tal interoperabilidade é dificultada: (i) pelo fato de outras plataformas de middleware não descreverem a interface IDL dos componentes; (ii) por haver incompatibilidade das implementações CORBA não escritas na linguagem específica da plataforma (Java no caso do EJB); (iii) pela ausência de uma implementação de referência para CORBA; (iv) pelo pouco suporte para transporte de objetos complexos entre chamadas remotas de linguagens de programação diferentes.

Outra estratégia amplamente utilizada para se obter interoperabilidade são os *Web Services* [Both 04], onde servidores disponibilizam serviços descritos através de uma interface especificada em WSDL (*Web Service Description Language)*. A interoperabilidade é provida pelo uso de padrões em termos de linguagens e protocolos de comunicação. Entretanto, *Web Services* não suportam o paradigma de conversação entre objetos [Coelho 07], implementando o paradigma de troca de mensagens, que não permite o cliente referenciar objetos remotos. Soluções como WSRF (*Web Services Resource*

Framework) [Granham 06] fornecem aos *Web Services* a capacidade de manter estados transacionais, porém continuam sem permitir o uso de objetos remotos diretamente pelo cliente, cabendo a esse implementar mecanismos para reconstruir tais objetos.

Pelo exposto, observa-se a necessidade de prover um ambiente que suporte interoperabilidade entre diversas plataformas de *middleware,* de forma transparente para as aplicações e que, ao mesmo tempo, não onere o desenvolvedor nem gere um grande impacto no desempenho das aplicações. Para definir a especificação desse ambiente faz-se necessário prover uma arquitetura de referência, que deve: (i) especificar como os componentes das diversas plataformas de *middleware* devem colaborar, (ii) prover mecanismos de seleção dinâmica de componentes, que abstraiam do desenvolvedor as complexidades inerentes à busca de componentes em cada plataforma, (iii) possibilitar a utilização dos componentes de forma transparente sem a necessidade de modificar ou adaptar componentes das diversas plataformas de middleware para uso dentro do ambiente proposto, (iv) permitir que objetos escritos em outras linguagens diferentes da linguagem da plataforma de ambiente sejam acessíveis pela mesma.

Como visto anteriormente, um dos principais problemas da plataforma CORBA foi a falta de uma implementação de referência que seja capaz de mostrar todas as potencialidades da plataforma. A necessidade de desenvolver uma implementação para a arquitetura de referência, a fim de validá-la é mostrar como é possível realizar implementações seguindo a arquitetura. No desenvolvimento de software as ferramentas que dão suporte a essa atividade devem tornar o processo de desenvolvimento mais produtivo, oferecendo ferramentas para geração de código padrão, tais como inicialização de serviços, ou mesmo para configuração de aplicativos como inicialização de servidores de aplicação, instalação de componentes no servidor, etc.

1.1 Objetivos

O objetivo desse trabalho é prover uma arquitetura de referência para desenvolvimento de aplicações distribuídas, chamada arquitetura *multi-middleware,* que seja independente de uma plataforma de *middleware*

específica e ofereça suporte a diversidade de plataformas. A idéia é apresentar uma arquitetura que permita a inserção de novas plataformas de *middleware*, desde que sigam as especificações propostas. Para tal, é necessário definir: (i) uma *camada multi-middleware* que seja responsável por abstrair a complexidade de acessar componentes das diversas plataformas de middleware; (ii) um serviço de seleção dinâmica para cada plataforma de middleware que seja responsável por localizar referências a objetos distribuídos da plataforma middleware de acordo com critérios de busca definidos; (iii) um componente *binding* para realizar o mapeamento entre os tipos específicos de cada plataforma de middleware para os tipos utilizados na linguagem em que é implementada a camada *multi-middleware*; (iv) um componente *bridge* para cada plataforma de middleware que é responsável por abstrair a complexidade inerente de acessar cada plataforma e coordenar as requisições aos componentes *binding* e ao serviço de seleção dinâmica.

Além de propor o ambiente, para validar a especificação, será fornecida uma implementação de referência com suporte as plataformas de middleware CORBA e EJB. Tal implementação será desenvolvida no contexto do LuaSpace [Almeida 06], um ambiente que utiliza uma linguagem de configuração para definição de estrutura de aplicações baseadas em componentes e que dá suporte a reconfiguração dinâmica de aplicações, promovendo o reuso de componentes CORBA. Portanto, o ambiente será expandido, incorporando os elementos da arquitetura de referência. O ambiente LuaSpace foi escolhido pelo seu enfoque em tornar fácil e simples o desenvolvimento de aplicações CORBA, por prover serviços de seleção dinâmica para componentes CORBA e incluir um *binding* entre CORBA e a linguagem Lua[Ierusalimsky 96]. A linguagem Lua será usada na implementação da camada *multi-middleware* por ser uma linguagem interpretada, dinamicamente tipada, com foco na simplicidade de desenvolvimento e adequada para prototipação. A implementação de referência será disponibilizada através de um plugin Eclipse [OTI 03] que fornecerá uma IDE(Integrated Development Enviroment) e um conjunto de ferramentas capazes de abstrair as eventuais complexidades de utilização (configuração)

da infra-estrutura, além de facilitar o processo de desenvolvimento de aplicações no ambiente multi-middleware.

1.2 Estrutura do Trabalho

Esse trabalho está estruturado da seguinte forma. O capítulo 2 apresenta os conceitos básicos relacionados com esse trabalho: conceitos sobre a plataforma CORBA e EJB; aspectos de interoperabilidade entre CORBA e EJB, o framework Eclipse e o ambiente LuaSpace. O capítulo 3 apresenta a arquitetura de referência. O capítulo 4 apresenta a implementação do ambiente *multi-middleware* com suporte as plataformas de middleware CORBA, JEE e também plugin para o Eclipse. O capítulo 5 apresenta um estudo de caso que explora as funcionalidades existentes no ambiente e também apresenta uma análise de desempenho do ambiente. O capítulo 6 contém uma discussão sobre os trabalhos relacionados em especial a comparação com *Web Services*. O capitulo 7 contém as conclusões e os trabalhos futuros.

2 CONCEITOS BÁSICOS

Neste capítulo apresentaremos os conceitos básicos que fornecem fundamentação teórica para o trabalho. Apresentaremos os conceitos sobre as plataformas de *middleware* CORBA (seção 2.1) e EJB (seção 2.2). Discorreremos na seção 2.3, sobre os aspectos de interoperabilidade entre as duas plataformas de *middleware* supracitadas e discutimos os principais problemas relacionados a essa integração. Em seguida, na seção 2.4, apresentamos os conceitos relacionados a linguagem Lua utilizada na implementação da camada *multi-middleware* e na seção 2.5 apresentamos os conceitos relacionados ao *framework* Eclipse, que será a base para o desenvolvimento do plugin do ambiente *multi-middleware.* Por último, na seção 2.6, apresentamos o ambiente LuaSpace.

2.1 CORBA

CORBA (*Common Object Request Broker Architecture*) [OMG 04] é um padrão proposto pela *Object Management Group* (OMG) cujo propósito é permitir interoperabilidade entre aplicações em ambientes distribuídos e heterogêneos. Este padrão estabelece a separação entre a interface de um objeto e sua implementação. Para descrição da interface do objeto, CORBA oferece a linguagem para definição de interfaces (IDL). Para implementação do objeto CORBA, pode ser utilizada qualquer linguagem de programação que tenha o mapeamento (*binding*) para CORBA.

A arquitetura CORBA como ilustrada na Figura 1 é composta por um conjunto de blocos funcionais que usam o suporte de comunicação do ORB (*Object Request Broker*) - o elemento responsável por coordenar as interações entre os objetos interceptando as chamadas dos clientes e direcionando-as para o servidor apropriado.

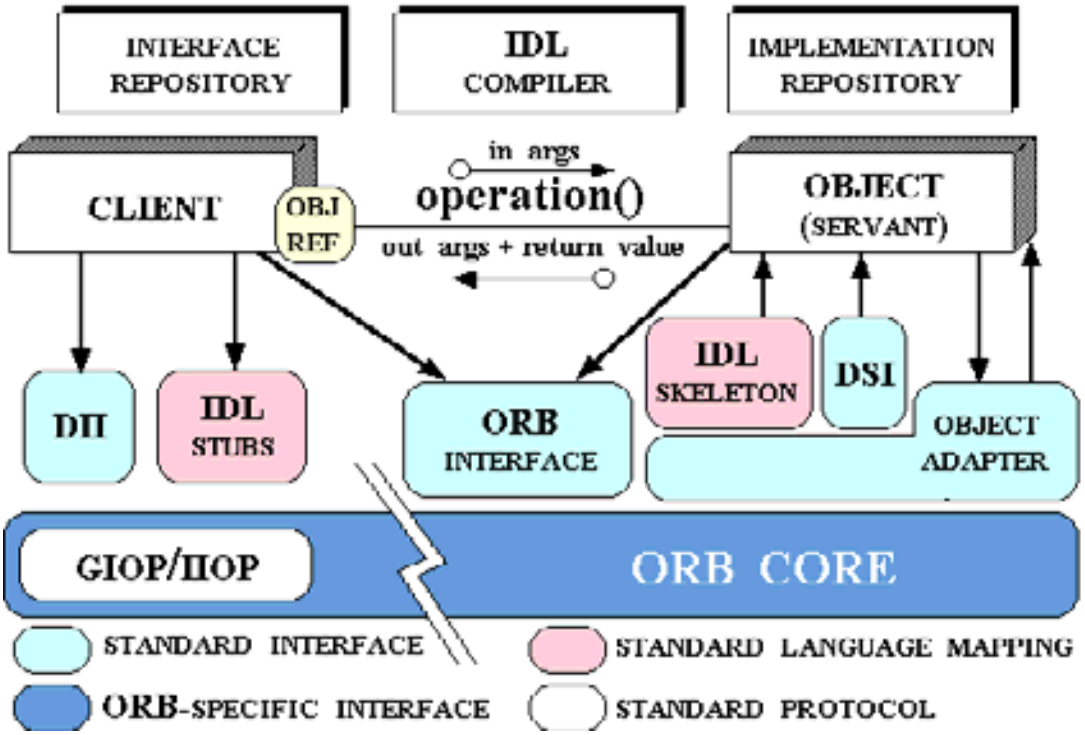


Figura 1 - Arquitetura CORBA [em www.cs.wustl.edu/~schmidt/corba-overview.html]

Todo objeto CORBA possui uma identificação, chamada *referência do objeto*, que é atribuída pelo ORB na criação do objeto. Para usar um objeto, o cliente deve obter a sua referência, pois em uma invocação de um método sobre o objeto, o ORB o identifica através de sua referência.

O *Repositório de Interfaces* definido no padrão CORBA disponibiliza informações necessárias para a construção de chamadas dinâmicas. Este repositório armazena todas as definições IDL dos objetos CORBA disponíveis para uso. A utilização do repositório de interfaces para a localização de objetos apresenta a restrição de ter de se conhecer a referência do objeto para adquirir mais informações sobre ele.

A Figura 2 mostra um exemplo de descrição de uma interface IDL, *Hello*, que possui apenas um método chamado *print* que recebe como entrada a mensagem a ser impressa.

```
module Tst_Hello
{
   interface Hello
   {
        void print(in string msg);
   };
```

Figura 2 - Interface Hello escrita em IDL

Todas as implementações CORBA devem oferecer suporte ao protocolo IIOP [OMG 04] (*Internet Inter-Orb Protocol*), utilizado em redes TCP/IP. Esse protocolo é usado para envio/recebimento de mensagens entre o cliente e o servidor possibilitando a transmissão de diversos tipos de dados suportados por CORBA, tais como *struct*, *boolean, long* dentre outros.

2.2 EJB

Os elementos que compõem o EJB [Sun 03] são: servidor EJB, *container*, componente (*bean* ou *enterprise bean*), descritor de implantação, interface *home* e interface *remote*. A Figura 3 ilustra a arquitetura do EJB.

O *servidor EJB* gerencia um ou mais *containers* e oferece serviços comuns como transações, segurança, persistência, entre outros, evitando o desenvolvimento deles em cada nova aplicação. Existe uma variedade de servidores EJB disponíveis no mercado: BEA's WebLogic[Bea 07], IBM WebSphere[Jain 07] e JBoss[Fleury 03].

O *container EJB* tem como finalidade oferecer ao programador do componente os serviços disponibilizados pelo servidor. Os serviços são definidos de maneira ortogonal ao componente, ou seja, a especificação dos serviços utilizados na aplicação é separada dos arquivos Java que implementam a lógica da aplicação. Usando uma *semântica declarativa* o programador especifica, em um arquivo XML, chamado *descritor de implantação,* as instruções para implantação do componente, a lista de recursos necessários para os componentes, os papéis de segurança para a aplicação, a informação de autenticação e a lista de controle de acesso para os vários métodos. Isto é possível porque o *container* é o intermediário entre o

cliente e o componente, interceptando todas as chamadas de métodos direcionadas ao componente. Da mesma forma que um *stub* RMI está entre o cliente e o objeto remoto, o *container* EJB está entre o cliente e o componente. O cliente nunca acessa diretamente um método do componente. O acesso é realizado via o *container* que contém o componente, através de duas interfaces que o *container* disponibiliza: a interface *home* e a interface *remote*.

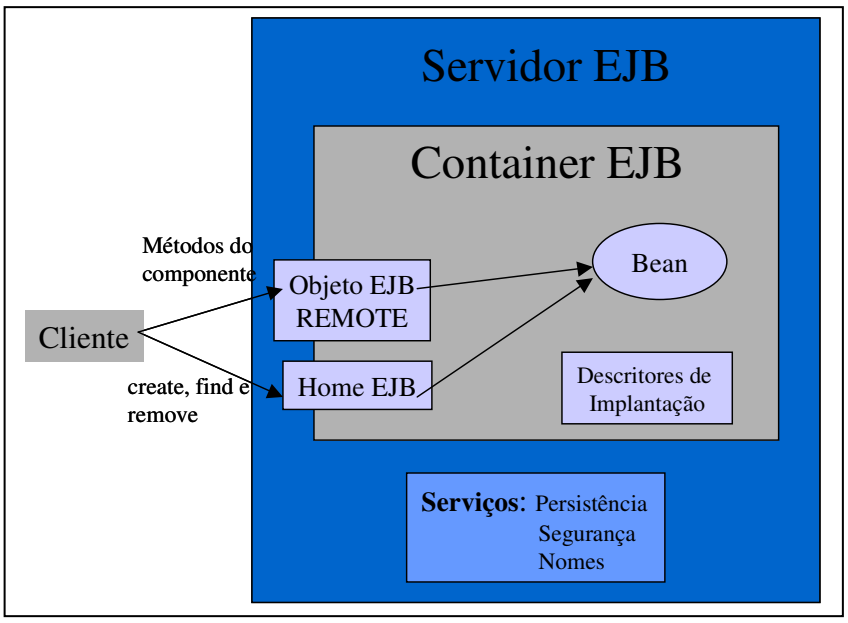


Figura 3 - Arquitetura do EJB

O EJB segue a idéia de *middleware* implícito que elimina a necessidade do programador interagir com o *middleware* para usar serviços adicionais à aplicação. A aplicação apenas declara, no descritor de implantação, quais serviços são necessários. Na execução da aplicação, antes da chamada ao objeto ser realizada, um *interceptador* captura a invocação e executa os serviços declarados no descritor.

A interface *home* é responsável pelo controle das operações de ciclo de vida sobre um componente: criação, remoção e localização do componente.

A interface *remote*, representada na Figura 3 pelo *objeto EJB Remote*, expõe a *interface do componente* que define os métodos que o componente oferece para os clientes. Este objeto é um *proxy* para a instância do

componente. Quando o cliente invoca um método do componente, esse objeto recebe a invocação e direciona para a instância do componente. Antes de fazer o direcionamento, há a interceptação dos serviços requisitados pelo componente.

Todos os aspectos mencionados anteriormente referem-se à especificação 2.1 do EJB que, como vimos anteriormente, utiliza uma série de interfaces e necessita de diversos descritores de implantação para os EJB´s serem colocados no servidor de aplicação. Na especificação 3.0 todas essas interfaces mencionadas anteriormente são substituídas por anotações [SUN 04], funcionalidade da linguagem Java que permite anotar o código para eventuais modificações que são realizadas pelo compilador e não pelo programador. A meta da nova especificação foi retirar a complexidade que existia anteriormente, removendo os descritores de implantação, uso excessivo de interfaces e classes EJB e, sobretudo, a complexidade para manipular *beans* de entidade. Na nova especificação os *beans* de entidade são substituídos por POJO (*Plain Old Java Objects*), que são classes Java com os atributos das entidades e são gerenciados pelo novo mecanismo de persistência chamado de JPA (*Java Persistence API*) que fica responsável por realizar o mapeamento de objetos para o banco de dados. As figuras 4 e 5 mostram a comparação entre trecho de código para um mesmo serviço utilizando a versão 2.1 e 3.0 da especificação EJB. As implementações abaixo referem-se a um *bean* que tem como objetivo receber dois números inteiros e produzir a soma dos mesmos.

```
1 @Remote
2 public interface ICalculator {
3     public int soma(int x,int y);
4 }
5 @Stateless
6 public class Calculator implements ICalculator {
7     public int soma(int x, int y) {
8           return x+y;
9     }
10 }
```

Figura 4 - Bean Calculator na especificação 3.0

```
1 public class CalculatorBean implements SessionBean {
2     public int somar(int x, int y) {
3           return x+y;
4     }
5 }
6 public interface CalculatorHome
   extends javax.ejb.EJBHome
7 {
8   public br.ufrn.app.Calculator create()
     throws javax.ejb.CreateException,java.rmi.RemoteException;
9 }
10 public interface Calculator
  extends javax.ejb.EJBObject
11 {
12   public int somar( int x,int y )
      throws java.rmi.RemoteException;
13 }
```

Figura 5 - Bean Calculator na especificação 2.1

Note que na figura 4 existe apenas uma interface, a *remote*. Em seguida há a classe que implementa a interface definindo a operação de soma. Na especificação 2.1, cujo código está ilustrado na Figura 5, é necessário duas interfaces, além da classe que implementa de fato a soma dos dois números. Podemos observar que é necessário que o programador conheça diversas classes e interfaces quem deve ser implementadas/herdadas. No código da Figura 5 ainda foram omitidos os códigos relativos aos métodos de gerenciamento do ciclo de vida, como métodos para remoção e ativação de objetos.

2.3 Aspectos de Interoperabilidade entre EJB e CORBA

A interoperabilidade entre CORBA e EJB reside na capacidade de que, clientes escritos em qualquer linguagem possam acessar componentes EJB com se estivessem acessando objetos CORBA. Um das metas da arquitetura EJB é fazer com que clientes e servidores CORBA interajam entre si, fazendo com que componentes EJBs sejam utilizados em uma larga gama de sistemas heterogêneos, permitindo que restrições desnecessárias sejam feitas aos arquitetos de software no momento de projetar o sistema ou da integração do mesmo.

A maioria das implementações EJB, tais como a implementação JBoss e a implementação de referência da própria Sun, foram construídas com base no protocolo CORBA-IIOP (*Internet Inter-Orb Protocol*), permitindo que a comunicação CORBA-EJB aconteça seguindo os padrões CORBA para

desenvolvimento de aplicações. Como visto anteriormente, para se desenvolver aplicações CORBA é necessário escrever um contrato do componente que oferece/requisita serviços. Este contrato é escrito na linguagem IDL e, juntamente com o *binding* especifico para cada linguagem, a interoperabilidade é alcançada. Componentes CORBA podem ser escritos em qualquer linguagem desde que tenham associado a si uma especificação IDL, enquanto que componentes EJB são escritos unicamente em Java. Ou seja, para que um cliente CORBA acesse um componente EJB o mesmo deve ter uma interface IDL equivalente para que seja realizada a comunicação. Na especificação [OMG 03] são definidos os procedimentos para mapeamento de Java para IDL. Na versão 2.3 da especificação CORBA o sistema de tipos de dados do IDL foi ampliado para que fosse possível definir o que é chamado *CORBA ValueTypes,* que equivale a objetos locais (se estivéssemos programando localmente), permitindo com que os mesmos sejam passados como parâmetros e retornados por métodos. A instância de uma classe Java é equivalente a um *CORBA ValueType*. Sem a adição desse tipo de objeto seria impossível existir interoperabilidade utilizando componentes escritos em Java.

Existem vários problemas relacionados ao mapeamento EJB para CORBA, que são discutidos em [IONA 01]. Dentre um dos principais problemas citados está a necessidade do cliente não escrito em Java, precisar se familiarizar com detalhes específicos da linguagem.

No desenvolvimento de aplicações EJB, com base em especificações anteriores a 3.0, para cada componente EJB instalado no servidor existem duas interfaces: *Home* e *Remote*. Para que clientes CORBA acessem esse EJB é necessário definir as mesmas interfaces em IDL e gerar os *stubs* necessários para que as mesmas sejam utilizadas. Existem ferramentas que produzem IDL com base em classes Java. Uma delas é o compilador *rmic* disponível em qualquer versão do JSDK (*Java System Development Kit*) da *SUN*. Essas ferramentas seguem a especificação [OMG 03] para construir as IDL.O problema associado a esse tipo de ferramenta é a construção de interfaces não amigáveis, podendo gerar uma infinidade de arquivos que muitas vezes são desnecessários para o funcionamento da aplicação. De posse dos *stubs*, o cliente CORBA pode realizar chamadas para o EJB da

mesma forma como realizaria chamadas para componentes CORBA. Clientes CORBA utilizando estratégias de geração automática de *stubs* podem recorrer, com mais sucesso, ao mapeamento automático de componentes EJB em interfaces IDL A Figura 6 mostra a sistemática de chamada de clientes CORBA a componentes EJB.

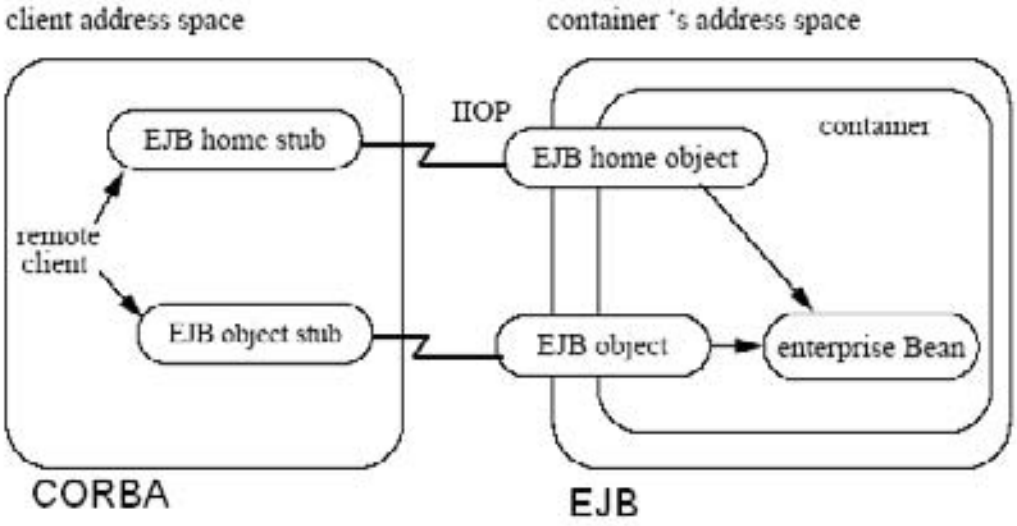


Figura 6 - Invocação de EJB através de clientes CORBA

Para que o componente EJB seja acessado via um cliente CORBA o mesmo precisa ser resolvido pelo serviço de nomes do CORBA, onde normalmente a implementação do serviço de nomes CORBA é feita pelo *container* EJB. O cliente CORBA instancia o serviço de nomes, em seguida realiza a operação de *lookup* (que retorna a referência da interface *home)* e, na seqüência, o procedimento equivale ao mesmo usado em clientes Java acessando EJB: o método *create* é invocado, obtendo referência à interface *remote* acessando assim os métodos de negócio implementados pelo EJB.

A implementação da especificação EJB utilizada neste trabalho é o JBOSS 4.0.4 [Fleury 03], uma das implementações mais utilizadas, disponível sobre a licença GPL e com implementação IIOP, permitindo assim que clientes CORBA façam invocações a EJB´s via esse protocolo, utilizando o JacORB, implementação CORBA em Java integrada ao servidor de aplicações. No JBOSS para que o um componente EJB seja acessível através de um cliente CORBA, é necessário que, no descritor de implantação, seja especificado que

o mesmo aceita requisições via IIOP e RMI [SUN 02] (*Remote Method Invocation*). A Figura 7 mostra o trecho de código com o exemplo de um descritor de implantação do JBOSS para que um componente EJB seja acessível via IIOP.

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss>
<enterprise-beans>
<session>
 <ejb-name>Fibo</ejb-name>
 <jndi-name>ejb/Fibo</jndi-name>

 <invoker-bindings>
  <invoker>
    <invoker-binding-name>stateless-rmi-invoker</invoker-
binding-name>
  </invoker>
  <invoker>
    <invoker-binding-name>iiop</invoker-binding-name>
  </invoker>
 </invoker-bindings>
</session>
</enterprise-beans>
</jboss>
```

Figura 7 - Descritor de implantação do JBOSS

Outra abordagem utilizada alternativamente para acessar componentes EJB a partir de clientes CORBA é a utilização de *Wrappers* que são objetos CORBA implementados em Java que tem como função evitar o envio/recebimento de tipos complexos Java. Supondo que um método Java receba como parâmetro um *java.util.Collection,* se usarmos o mapeamento direto é necessário que o cliente (seja C++, Python, Lua) forneça uma implementação para essa classe, uma vez que a mesma é passada por valor. Tal abordagem torna-se extremamente complexa e tediosa. Os *wrappers* servem para mapear os tipos complexos Java em tipos mais próximos dos tipos que podem ser definidos em especificações IDL. Semelhante à abordagem citada anteriormente, onde todos os tipos complexos devem ter uma implementação na linguagem em que foi

escrito o cliente, essa tática obriga os desenvolvedores EJB implementarem *wrappers* para todos os EJB existentes.

2.4 LUA

A crescente demanda por aplicações que sejam customizáveis em tempo de execução e possibilite o usuário escrever scripts e macros que aumentem a produtividade na utilização das aplicações motivou a criação da linguagem Lua[IERUSALIMSKY 96].

Lua foi desenvolvida na PUC-Rio e projetada para ser integrada de forma prática e simples com outras linguagens convencionais. Lua é dinamicamente tipada, o que significa que os tipos são definidos somente em tempo de execução de acordo com os valores que são atribuídos as variáveis. Os tipos que a linguagem Lua possui são: *boolean,number,string,function, userdata, thread e table*.

As estruturas de controle e sintaxe são semelhantes às linguagens de programação convencionais, possuindo *while,if e for*, definição de funções com passagem de parâmetros e variáveis locais e globais.

O tipo *table* (tabela) implementa *arrays associativos* que são arrays que não são indexados somente por números. No caso da linguagem Lua esses arrays podem também ser indexados por qualquer outro tipo da linguagem diferente de *nil*. Dessa forma o tipo tabela é o pilar para construção das estruturas de dados convencionais (árvores, registros, conjuntos, etc...) na linguagem Lua.

Funções são tratadas como valores de primeira classe, podendo ser passadas como parâmetros para outras funções, serem armazenadas em variáveis e retornadas como resultado. Em Lua não existe passagem de parâmetro por referência, pois é possível retornar mais de um valor nas funções.

O mecanismo de extensão utilizado em Lua é chamado de *metatable* que são tabelas usadas para definir comportamento de tabelas e dados dos usuário(userdata). Uma *metatable* é formada por um par (k,v) onde k é o evento(índice) e v é o *metamethod* associado ao evento. As metables permitem modificar o comportamento padrão do interpretador em determinadas

circunstâncias, tal como chamada de um método inexistente ou aplicar operador a tipos que não previstos para o mesmo.

O trecho de código da figura 9 ilustra como colocar em Lua, via *metatable*, suporte a orientação a objetos.

```
1 Account={
2     saldo=0,
3     sacar=function(self,v)
4           self.saldo=self.saldo-v
5     end,
6     depositar=function(self,v)
7           self.saldo=self.saldo+v
8     end
9 }
10 function Account:new (o)
11    o = o or {}
12    setmetatable(o, self)
13    self.__index = self
14    return o
15 end
16  a=Account:new({balance=0})
17  a:depositar(1000)
18 a:sacar(560)
19 io.write("Saldo eh: "..a.saldo)
```

Figura 8 - Trecho de código Lua

Nas linhas 1 a 9 é definida uma tabela Lua chamada *Account*. Cada campo da tabela é separado por vírgulas e consistem de um par (chave,valor). A tabela possui 3 campos: *saldo,sacar,depositar*. O campo sacar recebe como valor uma função com dois parâmetros: *self,v*. O parâmetro *self* é a palavra reservada que se refere a própria tabela, podendo acessá-la dentro do corpo da função. Como ilustrado na linha 4, a construção *self.saldo* referencia o campo *saldo* definido na tabela. Das linhas 10 a 15 é definido um construtor para a classe *Account* que recebe como parâmetro uma tabela Lua. Se o parâmetro for passado em branco, uma nova tabela é criada. Na linha 12, é associada uma *metatable* à tabela representada pela variável *o* com a tabela *Account*. Em seguida é utilizada a construção que significa que qualquer chamada a um campo na tabela que não esteja definido na mesma, deve ser repassada a tabela *Account*, fazendo a função de instanciar o objeto. Na linha 16 é instanciado um objeto através do construtor.

2.5 Eclipse

A plataforma Eclipse é uma IDE (*Integrated Development Enviroment*) [OTI 03] de propósito geral, implementada em Java, cujo principal papel é

fornecer um conjunto de ferramentas que possibilitem o desenvolvimento de aplicações nas mais variadas linguagens. A figura 9 ilustra a parte gráfica da plataforma.

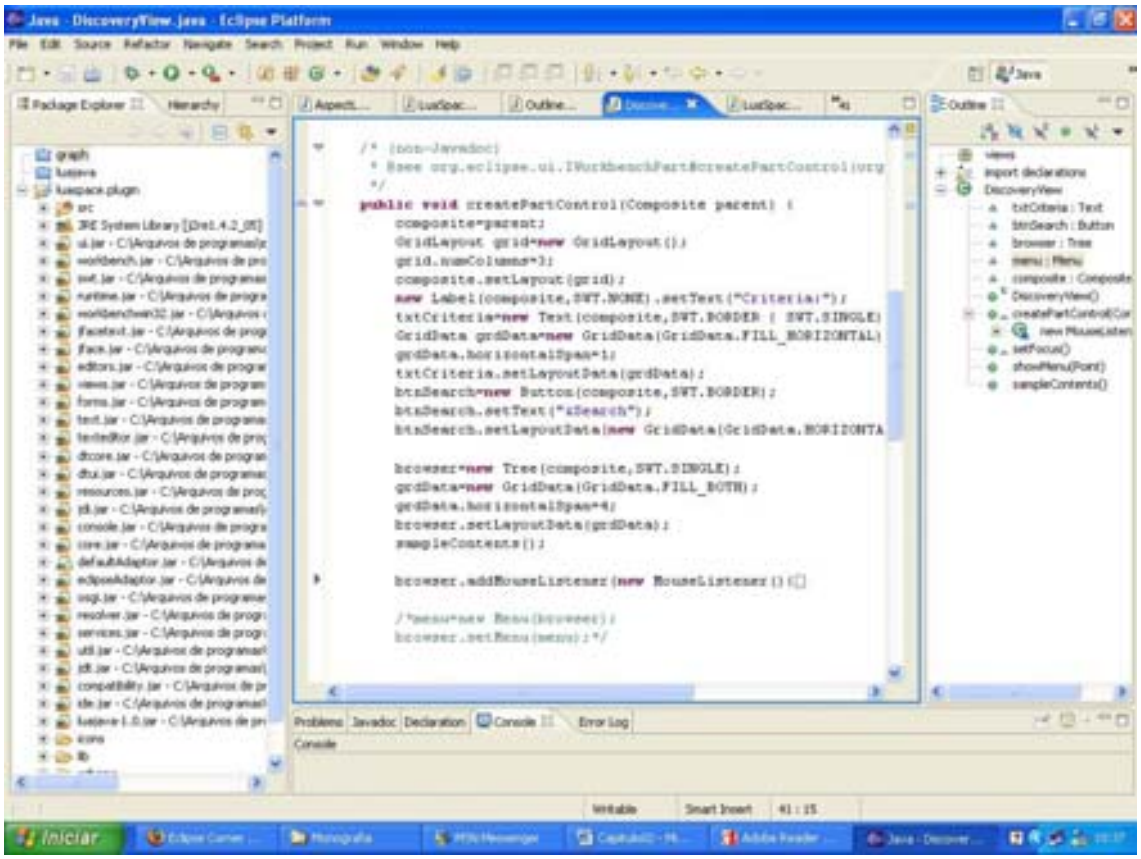


Figura 9 - Plataforma Eclipse Workbench

Em sua distribuição o Eclipse conta com uma série de ferramentas para desenvolvimento de aplicações Java. Porém, sua estrutura permite que novas linguagens e funcionalidades sejam inseridas na plataforma. O Eclipse foi desenvolvido para satisfazer os seguintes requisitos:

- Suportar a construção de uma variedade de ferramentas para desenvolvimento de aplicações.
- Utilizar a popularidade da linguagem Java para o desenvolvimento de ferramentas
- Suportar ferramentas para manipulação dos mais diversos tipos de conteúdo. Ex: Java,C++,GIF,HTML,XML e etc..
- Suportar ambientes baseados ou não em GUI.

- Executar em uma diversidade de sistemas operacionais tais como Linux e Windows

Um *plugin* é a menor unidade da plataforma do Eclipse que pode ser desenvolvida e entregue separadamente. Com exceção do núcleo do Eclipse, todas as demais funcionalidades são implementadas através de plugins. Novas funcionalidades a serem adicionadas a plataforma são desenvolvidas através de plugins que são acoplados à plataforma como ilustra a figura 10 que mostra a arquitetura da plataforma Eclipse.

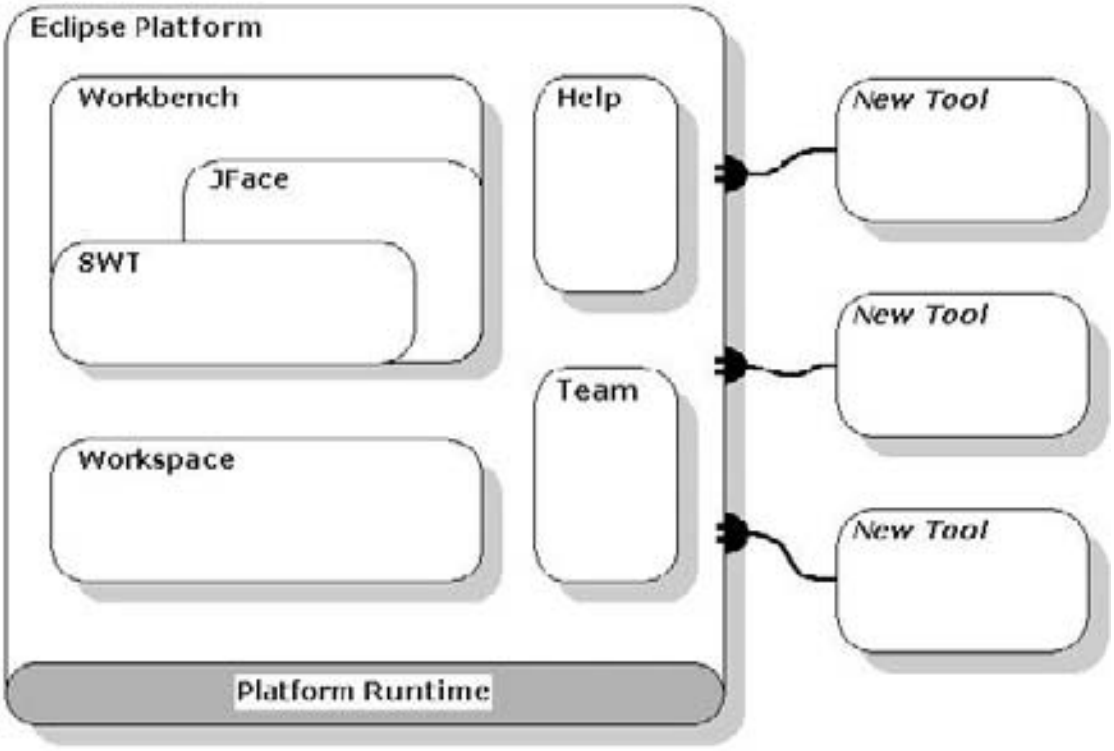


Figura 10 - Arquitetura da plataforma Eclipse

Workbench constitui a parte de UI (*User Interface*) da plataforma que representa o ambiente de trabalho. A API e a sua implementação são baseadas em dois *toolkits*:

- SWT (*Standard Widgets Tookits*) que oferece um gama de objetos para desenvolvimento de aplicações UI integrada com o sistema nativo de janelas, porém independente de plataforma.

- JFace: Toolkit implementado em cima do SWT, que torna mais simples a programação de aspectos relacionados a UI.

A parte do paradigma UI da plataforma do Eclipse é baseada em editores, *views,* e perspectivas. Do ponto de vista do usuário uma janela do *workbench* consiste visualmente de *views* e editores. Uma perspectiva é a forma como os editores e as *views* são organizadas na janela de trabalho. Editores permitem ao usuário abrir, editar e salvar objetos. Eles seguem o ciclo abrir-salvar-fechar semelhante em ferramentas baseadas em sistemas de arquivo, porém mais integradas com o *workbench.* Quando ativo, o editor pode contribuir com ações para a barra de ferramentas do *workbench.*

Views provêem informações sobre algum objeto na qual o usuário esteja trabalhando no *workbench*. Uma *view,* por exemplo, pode prover informações a respeito de um documento que esteja sendo editado. Na figura 8, no canto direito, é apresentada a *Outline View* que mostra informações sobre o arquivo Java sendo editado, tais como métodos, classes e atributos.

As várias ferramentas que estão *plugadas* na plataforma Eclipse atuam sobre um conjunto de arquivos na área de trabalho (*workspace*) do usuário. O *Workspace* consiste de um ou mais *projetos,* onde cada projeto é mapeado em um diretório no sistema de arquivos. Diferentes projetos podem mapear diferentes diretórios, apesar de que, por padrão, os projetos estão situados em um diretório raiz que representa a área de trabalho do usuário.

Pontos de extensão são pontos de um plugin que podem ser estendidos por outros plugin's adicionando novas funcionalidades. Cada ponto de extensão define os requisitos necessários para que uma classe possa implementar as configurações necessárias. O código de um plugin só é carregado quando realmente necessário. Um plugin é composto basicamente de 3 partes: (1) arquivo de *manifest,* que é um arquivo XML que contém informações a respeito das dependências de um plugin; (2) ponto de extensão e recursos externos utilizados pelo plugin; (3) um arquivo jar com as classes utilizadas na implementação do plugin; outros recursos (figuras,ícones, etc.). Para realizar a instalação do plugin o mesmo precisa ser descompactado na pasta *plugins* no diretório principal do Eclipse que, no momento de sua

inicialização, faz uma varredura localizando os plugin's instalados e realizando o carregamento dos mesmos no *workbench* no momento em que são necessários.

2.6 LuaSpace

LuaSpace é um ambiente para desenvolvimento de aplicações distribuídas baseadas em componentes CORBA que integra a plataforma CORBA com a linguagem Lua. O ambiente oferece um conjunto de ferramentas baseadas em Lua com funções estratégicas para facilitar o desenvolvimento de aplicações CORBA e para promover reconfiguração dinâmica. Em LuaSpace a aplicação é escrita em Lua e pode ser composta por componentes implementados em qualquer linguagem que tenha o *binding* para CORBA. Componentes, scripts e *glue code* são os elementos que forma uma aplicação LuaSpace.

A figura 11 mostra a arquitetura do LuaSpace EPlus

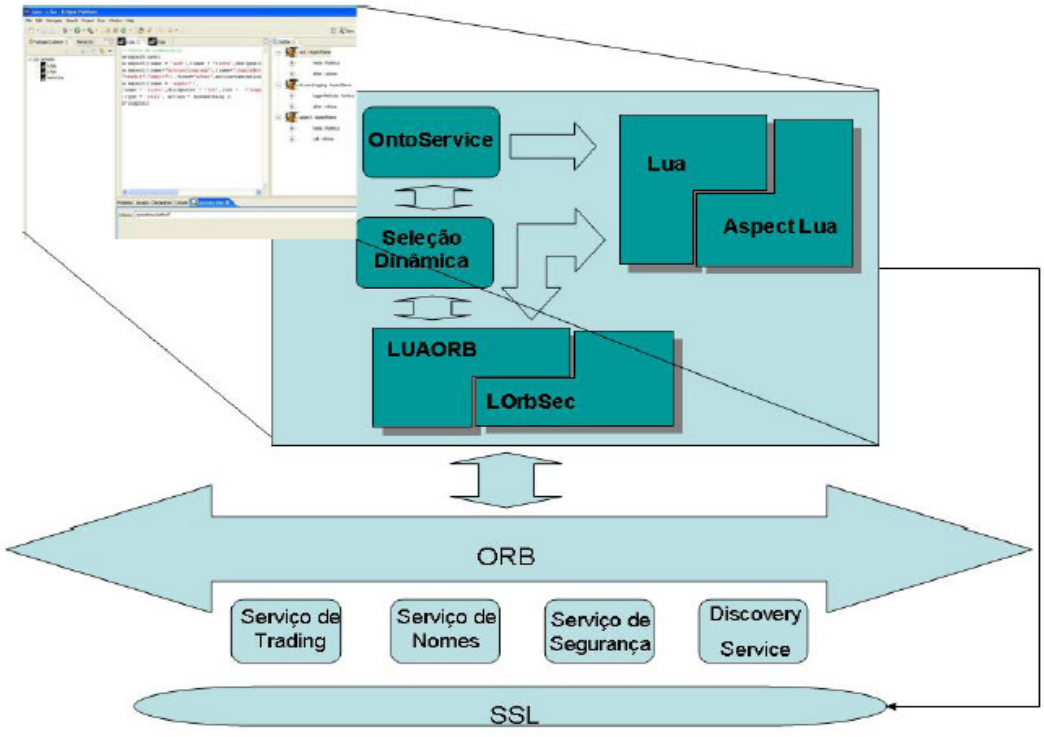


Figura 11 - Arquitetura do LuaSpace EPlus

O LuaSpace EPLUS [Almeida 06] consiste no ambiente LuaSpace com extensões que incluem: (1) a biblioteca de segurança *LOrbSec* [Martins 04]; (2) a incorporação do *Discovery Service* [Cacho 04] ao mecanismo de seleção dinâmica, (3) o *OntoService* [Cacho 05] que combinado ao *Discovery Service* constitui uma poderosa ferramenta para seleção dinâmica de componentes; (4) o *AspectLua* [Fernandes 04] que proporciona a integração de aspectos e componentes e (5) um plugin para a IDE Eclipse que prove mecanismos para utilização das ferramentas que compõe o LuaSpace de forma simples e intuitiva.

A extensão do ambiente em relação à seleção dinâmica e segurança exploram serviços CORBA de forma a evitar soluções proprietárias. O LuaSpace EPlus habilita automaticamente os serviços tornado-os operacionais para uso a qualquer momento. Em LuaSpace o script de configuração da aplicação é submetido ao interpretador Lua que o executa fazendo chamadas a cada diferente componente da arquitetura para tratar os comandos. LuaOrb é acionado nos casos de chamadas que fazem acesso a objetos CORBA. LuaOrb faz o mapeamento entre a chamada Lua e a chamada correspondente na plataforma CORBA. LuaOrb também é invocado quando são feitas chamadas para instalação dinâmica de objetos Lua a um servidor remoto.

O *Discovery Service* permite que a aplicação do usuário registre e consulte componentes seguindo as definições de conceitos presentes no OntoService. O Discovery interage com o OntoService para verificar se os componentes registrados seguem corretamente as definições ontológicas. O Discovery auxilia no processo de busca de componentes através da substituição de termos equivalentes.

AspectLua é uma extensão da linguagem Lua que oferece suporte a programação orientada a aspectos. Os programas são submetidos ao interpretador Lua que executa tanto os comandos Lua quanto AspectLua. A ordem com que os programas devem ser submetidos ao interpretador deve seguir a seguinte ordem: implementação de AspectLua (que permite o uso de AOP), a aplicação e a definição dos aspectos, podendo ser utilizado um arquivo auxiliar contendo a ordem correta da carga dos arquivos. Como

AspectLua é uma extensão da própria linguagem Lua, a utilização em conjunto com as demais ferramentas é realizada de forma natural, não sendo necessária nenhum adaptação das demais ferramentas.

A interface gráfica é construída sobre um plugin para IDE Eclipse com janelas para edição de código, assistentes para geração de código, navegador de aspectos integrado com o editor mostrando os pontos de interceptação, navegador de interfaces, configurador de preferências além dos demais utilitários fornecidos pelo Eclipse para o desenvolvimento de aplicações baseadas no LuaSpace.

3 ARQUITETURA

Neste capitulo apresentaremos a arquitetura de referência do ambiente multi-middleware (seção 3.1), apresentando sua visão geral e em seguida explicitando todos os componentes através do uso de diagramas de classes que mostram todos os detalhes dos componentes da arquitetura de referência. A seção 3.2 apresenta a camada multi-middleware, A seção 3.3 apresenta o mecanismo de seleção dinâmica. Na seção 3.4 mostramos o componente de *bridge* explorando tanto o lado cliente como o lado servidor e na seção 3.5 o mecanismo de *binding*.

3.1 Arquitetura de referência

A Figura 12 ilustra a arquitetura de referência do ambiente *multi-middleware* para desenvolvimento de aplicações distribuídas. Os elementos chave da interoperabilidade encontram-se em destaque: (i) a *camada multi-middleware* que é responsável por abstrair a complexidade de acesso do cliente/desenvolvedor; (ii) a *bridge*, que tem como objetivo detectar o registro de componentes na plataforma de middleware específica (p.ex. em CORBA a *bridge* deve monitorar os registros de componentes no repositório de interfaces, e no JEE monitorar o *deploy* do componente) e notificar o serviço de seleção dinâmica para que o mesmo mantenha também o registro de tal componente*; (iii)* o *binding middleware*, responsável por tratar do mapeamento de tipos de dados e delegar a execução das chamadas aos componentes da plataforma; (iv) o *mecanismo de seleção dinâmica*, que tem como objetivo localizar componentes em cada plataforma de middleware.

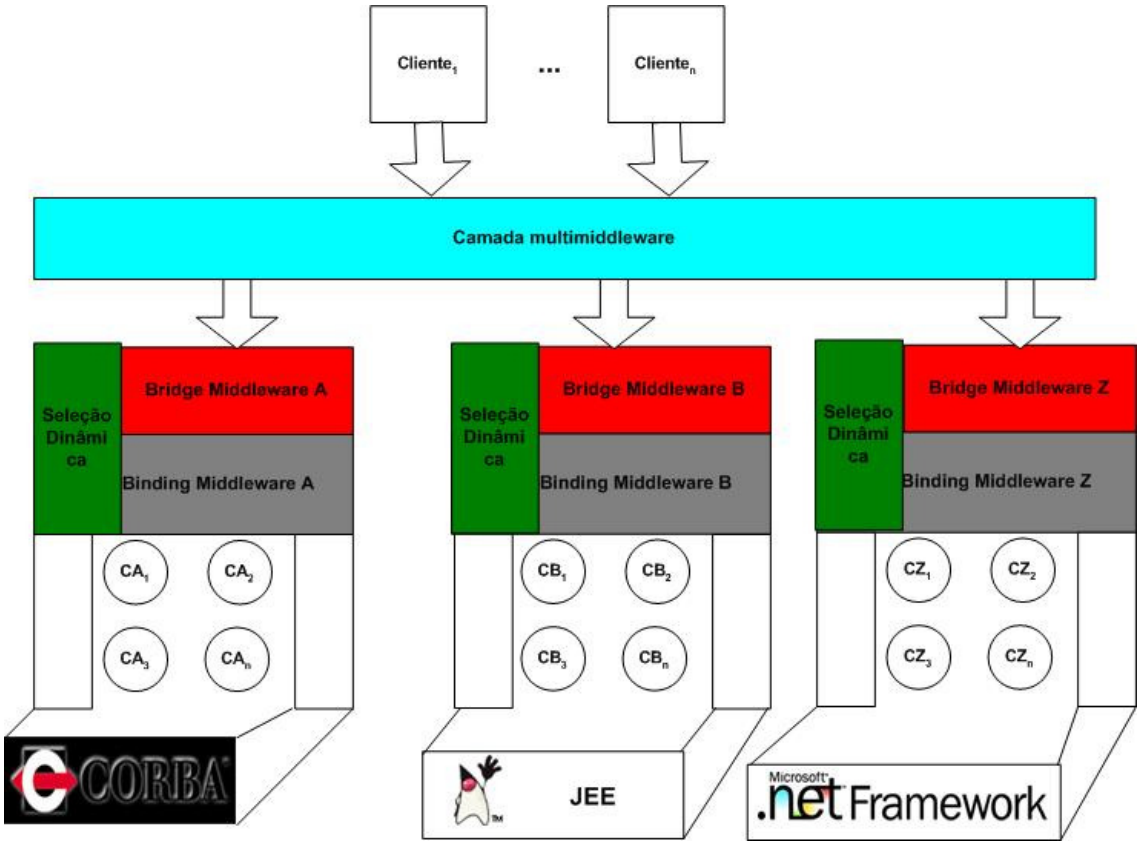


Figura 12 - Arquitetura do ambiente de desenvolvimento multi-middleware

A camada *multi-middleware* é responsável por acessar as diversas plataformas de middleware subjacentes de modo transparente para o desenvolvedor ou cliente que esteja requisitando algum serviço. Ela atua como um *Facade* [Gamma 05], delegando as requisições para cada plataforma de middleware que se registra com a camada. Na camada *multi-middleware* existe um componente que implementa a interface *IManager* e que atua como um gerenciador, recebendo o registro da referência dos *bridges* de cada plataforma especifica, cabendo a ele gerenciar como as solicitações devem ser feitas a esses *bridges.*

O mecanismo de seleção dinâmica é implementado para cada plataforma, de forma a lidar com as peculiaridades relativas a busca de objetos em cada uma delas. Tal mecanismo não lida com mapeamentos ou invocações de métodos. Essas tarefas cabem ao componente *bridge*. O objetivo do mecanismo de seleção é localizar objetos, utilizando critérios de busca que se baseiam em assinatura de métodos e em propriedades definidas pelos

serviços. Cada mecanismo de seleção tem liberdade para definir como devem ser armazenadas as informações sobre os serviços oferecidos. Porém, deve haver uma forma padrão para especificar critérios de busca, a fim de que seja possível garantir a combinação de critérios de busca e a uniformidade na seleção dos componentes.

Para cada plataforma existe um componente *bridge* cujo objetivo é abstrair a complexidade inerente de acessar os diversos serviços das plataformas de *middleware* seguindo as definições especificadas pela camada *multi-middleware*. Portanto, o componente *bridge* provê mecanismos para invocação de métodos (que, no final, são delegados ao *binding)*, passagem e recebimento de valores enviados pelos diversos componentes da plataforma. Esse componente também tem a função de detectar os componentes existentes na sua plataforma de *middleware* subjacente e notificar o mecanismo de seleção dinâmica da camada *multi-middleware* da disponibilidade do mesmo mantendo, assim, um registro do componente, da sua localização, e principalmente da especificação da sua interface. Essa interface é descrita de acordo com a plataforma, por exemplo, em CORBA é a interface IDL e na nossa implementação de referência, para a plataforma JEE, utilizamos um mapeamento XML para representar as interfaces dos EJB de forma acessível ao mecanismo de seleção dinâmica. Todo processo de notificação e registro dos serviços deve ser realizado de forma transparente para os usuários da camada *multi-middleware*.

O *Binding Middleware* é responsável por mapear os tipos de dados especificados em cada plataforma de *middleware* para os tipos de dados utilizados pela implementação da camada *multi-middleware*, além de ser responsável por enviar/receber as requisições provenientes da camada superior. Em cada plataforma há ainda os diversos componentes representados na Figura 12 pelos círculos na parte inferior da representação do *binding*. A arquitetura foi projetada para suportar um número qualquer de plataformas de *middleware* desde que os elementos citados estejam presentes, uma vez que os detalhes de mapeamento de tipos, invocação de métodos e demais aspectos relacionados às diversas plataformas de *middleware* existentes ficam a cargo dos componentes *bridge* e *binding*.

3.2 Camada Multi-Middleware

A figura 13 mostra o diagrama de classes da camada *multi-middleware.*

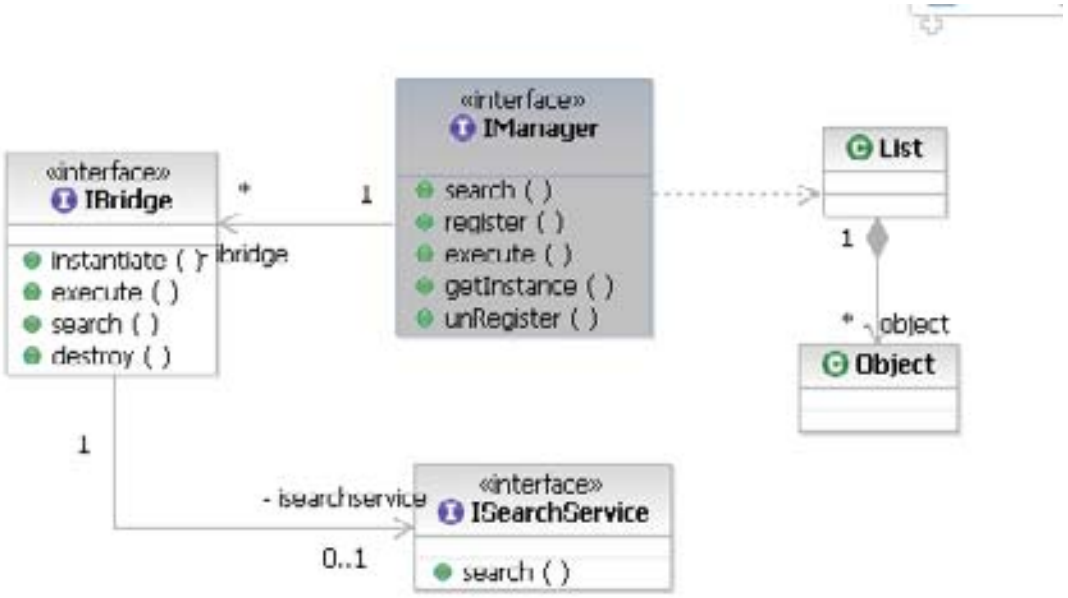


Figura 13 - Diagrama de classes da Camada Multimiddleware

A interface *IManager* funciona como uma fachada, responsável por registrar os *bridges* de cada plataforma, podendo assim delegar para o *bridge* específico de cada plataforma a execução e localização dos serviços.

O método *register* recebe como parâmetro um objeto que implementa a interface *IBridge* e registra na lista de *bridges*. De forma inversa o método *unregister* remove o registro do bridge.

O método *getInstance* retorna uma instância da classe que implementará a interface, de forma a utilizar o padrão *Singleton* [Gamma 99], evitando a criação de diversas instâncias da classe que implementa a interface *IManager.* Como o *IManager* implementa o padrão de projeto de fachada, só existe uma instância do *IManager* por execução do ambiente, de forma que o *IManager* atua gerenciando as requisições delegadas a camada *multi-middleware*.

O método *search* recebe como parâmetro uma *string* contendo o critério de busca dos componentes que estão instalados nas diversas plataformas de

middleware retornando um objeto da classe *List* que contém todas as referências aos objetos que satisfazem o critério passado como parâmetro.

O método *execute* realiza o mesmo procedimento do método *search*, porém, já realiza a chamada de métodos sobre os objetos que foram retornados pelo critério de busca. O método *search* pode ser utilizado em tempo de desenvolvimento através da integração com a IDE, permitindo que o desenvolvedor saiba, previamente, quais componentes estão disponíveis nos diversos servidores de aplicações das plataformas de *middleware* ou mesmo em tempo de execução, para que o desenvolvedor selecione de forma mais precisa qual método/componente ele deve executar, uma vez que o método *execute* invoca todos os métodos que satisfazem o critério de busca definido.

As classes *List* e *Object* representam, respectivamente, a lista e as referências dos objetos contidas nessa lista que satisfazem o critério de busca. Note que a referência aos objetos retornados pode ser de uma infinidade de tipos, que são mapeados pelo *binding*, para o objeto equivalente na linguagem utilizada na implementação do ambiente.

A interface IBridge possui três métodos *doSolicitation*, *instantiate* e *search* e *destroy* que servem, respectivamente, para invocar a execução de um método de um componente distribuído, para instanciar um objeto nativo da linguagem utilizada pela plataforma de *middleware*, sendo essa última delegada ao *binding* correspondente, o método search que delega ao serviço de seleção dinâmica específico a localização dos objetos que atendem o critério passado como argumento e finalmente o método *destroy* remove uma instância de um objeto nativo da linguagem utilizada pela plataforma de *middleware*. As demais interfaces e classes existentes no ambiente foram omitidas no diagrama em questão para efeito de simplificação.

Ao utilizar a camada *multi-middleware* o desenvolvedor não precisa mais se preocupar em carregar configurações, componentes relativos a uma plataforma de *middleware* especifica, delegando todas essas responsabilidades à camada *multi-middleware*. Por exemplo, supondo que existam dois componentes, um implementado em C++, disponibilizado no CORBA, e outro implementado em Java, utilizando JEE, ambos possuindo um método *print* que

recebe como parâmetro uma mensagem. Com a camada *multi-middleware* basta invocar *execute* da interface *IManager*, passando como parâmetro o nome do método *print* e a mensagem. A camada delega a invocação a cada *bridge* registrado para que sejam localizados todos os objetos que satisfaçam o critério de busca. Em seguida esses métodos são executados.

A Figura 14 ilustra o diagrama de seqüência correspondente à utilização da camada *multi-middleware*:

1. O cliente recupera a referência ao *IManager* que atua como gerenciador das requisições a camada multi-middleware;
2. Em seguida os *bridges* das plataformas de *middleware* especificas devem registrar-se no gerenciador, para que seja possível a busca/execução dos métodos implementados pelo diversos componentes existentes nas plataformas de *middleware* representados pelo *bridge*.
3. Na seqüência o cliente invoca o método *execute* passando como parâmetros a assinatura do métodos e os parâmetros, devemos frisar que o método execute só retorna um resultado de acordo com a disponibilidade do objeto, o método execute é delegado para cada bridge.
4. Cada bridge invoca o método *search* com base nos critérios de seleção que foram passados como parâmetro para método execute.
5. Cada *bridge*, por sua vez, delega ao *binding* que invoque o serviço de seleção dinâmica, passando como parâmetro o critério de busca. O retorno dessas chamadas é a referência aos componentes que satisfizeram o critério de busca.
6. Em seguida, o cliente invoca o método de negócio implementado pelo componente que atendeu os critérios de busca, cabendo ao *bridge* solicitar que o *binding* faça o mapeamento dos parâmetros, invoque o método do objeto e solicite ao *binding* que mapeie o retorno desse método para um objeto equivalente na linguagem utilizada na plataforma *multi-middleware*.

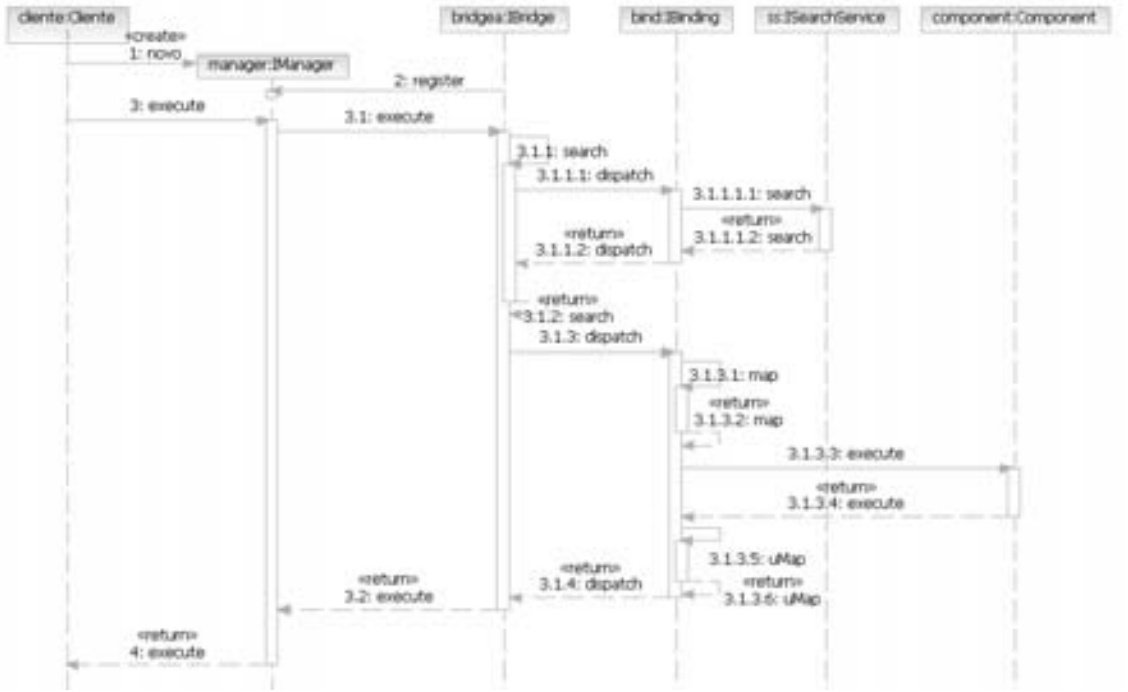


Figura 14 - Diagrama de Seqüência de Utilização da Camada Multi-middleware

3.3 Seleção Dinâmica

A figura 15 mostra o diagrama de classes do mecanismo de seleção dinâmica que deve ser implementado para cada plataforma de *middleware* que fará parte do ambiente *multi-middleware*.

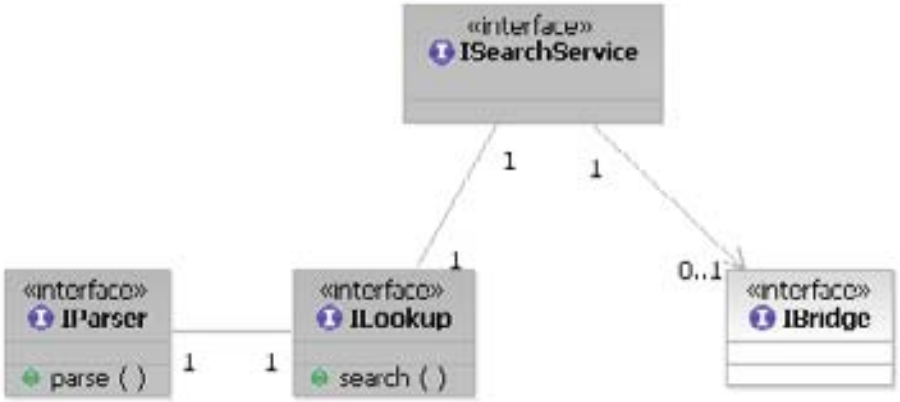


Figura 15 - Diagrama de Classes do Mecanismo de Seleção Dinâmica

A interface *ISearchService* encapsula os métodos para acessar as funções de localização de objetos. A interface *ILookup* é responsável por consultar os repositórios para localizar objetos que satisfaçam os critérios. Os repositórios dependem, exclusivamente, da estratégia utilizada para cada

plataforma de *middleware*. Para implementações baseadas em CORBA pode-se utilizar o repositório de interfaces. Para EJB pode-se utilizar um banco de dados contendo informações sobre os serviços ofertados ou mesmo outra estratégia existente. A forma como a informação será armazenada dependerá de cada implementação, cabendo à interface *ILookup* acessar esses repositórios e retornar a referência aos serviços. A interface *IParser* é responsável por realizar o *parser* do critério de busca passado como parâmetro. O critério de busca pode conter nome e valores de propriedades, combinados com os operadores *and*, *or* e *not*. A tabela 1 contém o conjunto de nomes e propriedades que podem ser utilizados no critério de busca.

Propriedade	**Descrição**
Servicename	Nome do serviço
OperationName	Nome da operação
ParamName	Nome do Parâmetro
OperationExceptionName	Nome de exceção gerada por uma operação
AttributteName	Nome de atributo

Tabela 1 - Nome de propriedades

Em [Cacho 04] é definido um conjunto maior de propriedades bem como funcionalidades, porém todas elas são específicas para uso com CORBA. A idéia de utilizar esse conjunto reduzido de propriedades é permitir compatibilidade com as diversas plataformas de *middleware* existentes. Com critérios mais genéricos garantimos a flexibilidade do ambiente.

A interface *IBridge* é a responsável por chamar o serviço de seleção dinâmica. Um exemplo de busca, seguindo o exemplo citado na seção anterior, seria a chamada à interface *ISearchService* passando como parâmetro *operationName=='print'*. A interface *IParser* realizaria a analise léxica desse critério e buscaria os componentes que satisfaçam tal critério, retornando a referência dos mesmos.

3.4 Bridge

O componente *bridge* é responsável por realizar a ponte entre a camada *multi-middleware* e os componentes das plataformas de *middleware*. Para cada plataforma de *middleware* utilizada deve existir um componente *bridge* correspondente. Uma de suas principais funções é detectar quando serviços são instalados e extrair informações sobre a interface do serviço. A outra função é delegar ao serviço de seleção dinâmica a procura pelos objetos existentes e coordenar, junto com o componente *binding,* o mapeamento e execução dos métodos dos objetos. A figura 16 apresenta o diagrama de classes do *bridge*, referente a parte que é responsável por mapear os serviços disponíveis de tal forma que o serviço de seleção dinâmica possa consultá-los.

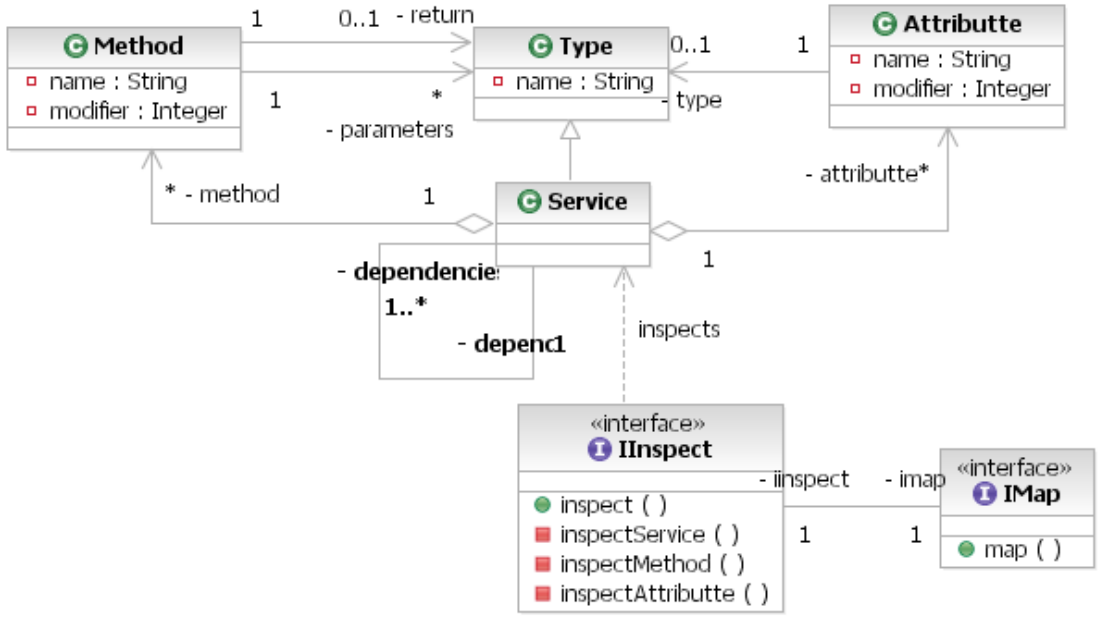


Figura 16 - Diagrama de Classes – Mapeamento do Serviço

A interface *IInspect* é responsável por inspecionar o serviço que foi disponibilizado e gerar o mapeamento adequado (por exemplo, na implementação de referência para a plataforma CORBA, especificar corretamente a IDL ou para EJB uma especificação XML das interfaces dos componentes) para o mesmo de forma a possibilitar que o serviço de seleção dinâmica localize-o. O formato desse mapeamento é arbitrário, ou seja, desde que o serviço de seleção dinâmica consiga consultá-lo.

A interface *IMap* é responsável por definir o método de mapeamento necessário. A classe *Service* representa um serviço a ser inspecionado. Ela representa o mesmo conceito de uma classe em Java que possui atributos e métodos, representados respectivamente pelas classes *Attributte* e *Method*. Consideramos que o serviço também faz parte dos tipos de dados da linguagem a ser utilizada. A implementação deve-se valer de funcionalidades de reflexão, para que seja possível analisar o conteúdo do serviço e extrair informações tais como tipos dos parâmetros, nome de métodos, nome de atributos, exceções geradas pelos métodos e assim por diante. O sucesso da utilização do mecanismo de *bridge* está diretamente vinculada questão do que a linguagem oferece em termos de reflexão, caso a linguagem não possua mecanismos de reflexão, o *bridge* não poderá ser implementado, uma vez que ficaria impossível analisar as especificações do serviços em tempo de execução. Quando o serviço é disponibilizado no servidor este deve notificar o *bridge*, passando a informação sobre o serviço que está sendo instalado. O *bridge* então inspeciona o serviço e gera o mapeamento de acordo como é definido pela implementação da interface *IMap*. Dessa forma, o serviço de seleção dinâmica está apto a consultar o repositório de dados onde está guardado o mapeamento. Para que um componente seja mapeado e, consequentemente, disponibilizado para o serviço de seleção dinâmica, o mesmo precisa ser inspecionado, através da chamada ao método *inspect*, que recebe como parâmetro a referência do componente. Supondo que existisse um *bean* de negócio Java chamado *br.ufrn.PrintService*, o método *inspect* receberia o nome da interface do serviço, que precisa, necessariamente, estar no mesmo servidor de aplicação do *bridge.*

A figura 17 mostra a parte do componente *bridge* que fica do lado do cliente do serviço, interagindo com a camada *mullti-middleware*. Essa função do bridge tem como objetivo articular com o serviço de seleção dinâmica e o *binding* a localização, execução e mapeamento dos dados.

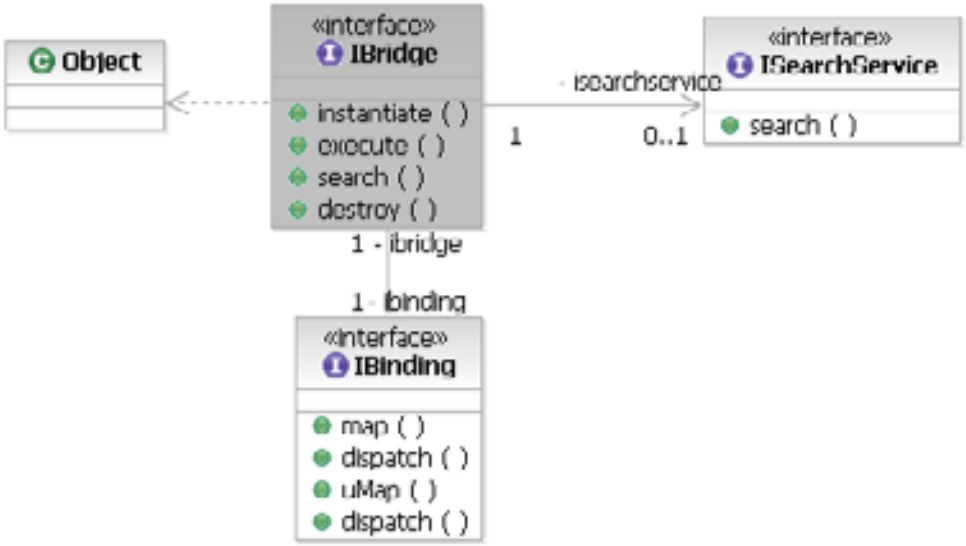


Figura 17 - Diagrama de Classes Bridge – Lado do cliente

A interface *IBridge,* como mencionado anteriormente, é responsável por delegar ao serviço de seleção dinâmica a procura por objetos com base nos critérios definidos, por intermédio da implementação da interface *ISearchService*. A interface *IBinding* representa o componente *binding* responsável por mapear os dados entre a linguagem utilizada pela plataforma de middleware e pelo ambiente *multi-middleware*. A interface *IBridge* ainda possui a função de instanciar objetos definidos pelo usuário em linguagem diferente da utilizada na camada *multi-middleware*, para tal a implementação *binding* é invocada recebendo os parâmetros do eventual construtor da classe, de acordo com a linguagem que é tratada pelo bridge.

A figura 18 apresenta como funciona o bridge, evidenciando os lados cliente e servidor do mesmo.

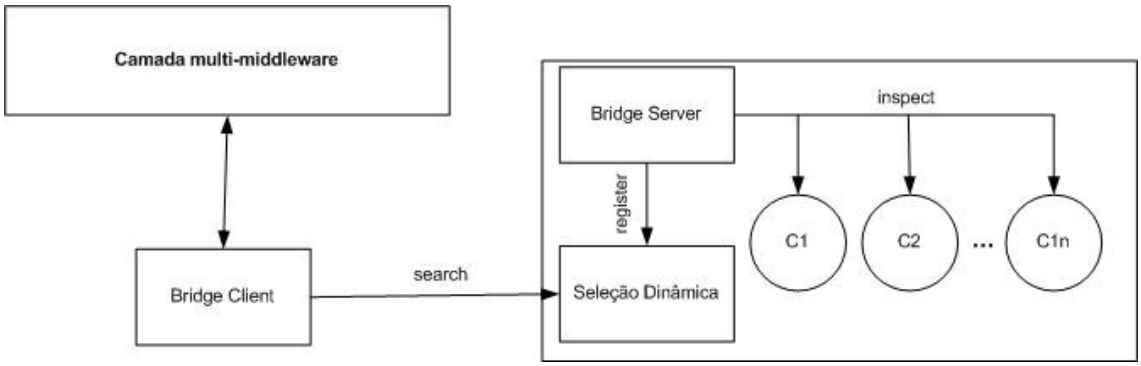


Figura 18 - Bridge Cliente/Servidor

3.5 Binding

O componente *binding* tem como função permitir que objetos escritos na linguagem do ambiente *multi-middleware* sejam vistos como objetos da linguagem da plataforma de *middleware* e vice-versa. Nesse trabalho não definimos como deve ser implementado esse componente, apenas especificamos requisitos que o mesmo deve satisfazer: (i) mapear tipos de dados da linguagem da plataforma de *middleware* para a linguagem da camada *multi-middleware*, (ii) permitir a execução dos métodos de objetos, os quais não estejam do lado servidor, para que soluções existentse, tais como os *bindings* das linguagens para CORBA, enquadrem-se dentro das necessidades do ambiente.

O *binding* pode ainda receber a execução dos métodos dos objetos implementados pelo usuário na linguagem da plataforma de *middleware* específica. As técnicas de mapeamento consistem em delegar a execução desses métodos para os objetos originais, fazendo com que o retorno dessa execução seja mapeado, de tal forma que a preocupação fundamental desse tipo de ferramenta seja mapear os dados sem se preocupar com o comportamento definido nas classes dos objetos em questão.

O *binding* LuaOrb [Cerqueira 99], por exemplo, é a implementação utilizada para mapear objetos CORBA para objetos Lua e vice-versa. Esse *binding* utiliza mecanismos de reflexão, tipagem dinâmica e construção de chamadas em tempo de execução para criação de *bindings* dinâmicos entre Lua e componentes CORBA.

Outro exemplo é o MCORBA [Jeffery 99], um *binding* CORBA para a linguagem Mercury [Somogy 95]. Tal linguagem foi especificada para ser trabalhada em times de desenvolvedores atuando em projeto de grande porte. O *binding* segue a mesma filosofia de funcionamento dos *bindings* para CORBA: com base nos elementos da linguagem gera-se a especificação IDL equivalente e dinamicamente gera-se *stubs* e *skeletons* que são utilizados na chamada dos componentes distribuídos.

4 IMPLEMENTAÇÃO

Neste capitulo apresentamos a implementação do ambiente, detalhando todos os componentes que compõem a plataforma. A seção 4.1 apresenta a implementação da camada *multi-middleware.* A seção 4.2 a implementação do mecanismo de bridge e seleção dinâmica para a plataforma CORBA, na seção 4.3 discutimos sobre os mesmos componentes da seção 4.2 porém com ênfase na implementação EJB, na seção 4.4 discutimos sobre os bindings utilizados na implementação do ambiente, na seção 4.5 apresentamos a implementação do plugin Eclipse que dá suporte ao desenvolvimento de aplicações utilizando o ambiente multi-middleware e na seção 4.6 mostramos quais passos necessários para escrever um cliente utilizando a linguagem Lua e para um cliente não escrito na linguagem Lua.

4.1 Implementação da Camada Multi-Middleware

Como citado no capitulo anterior, a camada *multi-middleware* atua como uma fachada para os componentes *bridge*, de tal forma que o desenvolvedor não precise se preocupar com os detalhes específicos de cada plataforma de middleware representadas pelo *bridge* especifico. Como mostrado no diagrama de classe da Figura 13, a interface *IManager* possui 3 métodos: *search*, *execute* e *register* que servem respectivamente para localizar objetos, executar chamadas de métodos e registrar *bridges*. A figura 19 mostra o trecho de código Lua da implementação da interface I*Manager*.

```
1 Manager={
2     instance=self,
3     bridges={},
4     getInstance=function(self)
5           return instance
6     end,
7     register=function(self,bridge)
8           self.bridges[table.getn(self.bridges)+1]=bridge
9     end,
10    search=function(self,criteria)
11          result={}
12          i=0
13          local tmp
14          for k,bridge in pairs(self.bridges)
15                tmp=bridge:search(criteria)
16                for k,ret in pairs(tmp)
17                      results[i]=ret
18                      i=i+1
19                end
20          end
21    end,
22    execute=function(self,method,parameters)
23          result={}
24          i=0
25          local tmp
26          for k,bridge in pairs(self.bridges)
27                tmp=bridge:execute(method,parameters)
28                for k,ret in pairs(tmp)
29                      results[i]=ret
30                      i=i+1
31                end
32          end
33          return result
34    end
35 }
```

Figura 19 - Implementação da camada Multi-Middleware

Na linha 1 definimos uma tabela Lua que representa a implementação da interface *IManager*. Na linha 2 é adicionado um atributo à classe chamado *instance*, que recebe o valor *self*, uma palavra reservada Lua que indica a referência da própria classe (reflexão). Na linha 3 é definido o atributo *bridges,* que é a tabela Lua que guardará a lista de *bridges* que serão registrados na camada multi-middleware. Nas linhas 4 a 6 está a implementação do método *getInstance* que implementa o padrão *Singleton*. As linhas 7 a 9 definem o método *register*, que recebe uma tabela Lua com a implementação do lado do cliente do *bridge* de cada plataforma específica. Nas linhas 10 a 21 temos a implementação do método *search*, que percorre a lista de *bridges* registrados delegando a execução do método *search* para cada *bridge*, passando o critério

de seleção como parâmetro. A execução do método *search* de cada *bridge* pode retornar várias referências a objetos que atendam o critério de busca, então, nas linhas 16 e 17, cada referência é extraída da lista e colocada na lista que conterá todas as referências encontradas, independente de qual plataforma de *middleware* a referência seja proveniente. As linhas 22 a 36 mostram a implementação do método *execute* que funciona de forma semelhante ao método *search*, porém, em vez do critério de busca, o método recebe o nome do método e os parâmetros do método a ser invocado. O retorno dessa chamada consiste na lista dos valores obtidos com a execução do método.

4.2 Bridge e Mecanismo de seleção dinâmica para CORBA

Na introdução foi citado o motivo da escolha do LuaSpace para receber a implementação de referência do ambiente multi-middleware. LuaSpace dispõe de um serviço de seleção dinâmica, que tem suporte a múltiplos critérios de busca, balanceamento de carga e busca síncrona e assíncrona.

O mecanismo de seleção dinâmica supracitado é o *Discovery Service* [Cacho 04] que oferece uma maneira simples e uniforme de buscar componentes considerando diferentes critérios: nomes de métodos, assinatura de métodos, exceções e propriedades não funcionais.

Esse serviço oferece os mecanismos básicos do *Serviço de Trading* tais como interfaces para exportar e importar serviços, para administrar e interagir com *traders* e/ou outros serviços CORBA remotos agrupados em uma federação. Além disso, o serviço oferece um mecanismo de *callback* para notificar a aplicação quando acontece o registro de um novo componente que satisfaz um critério previamente estabelecido. A figura 20 mostra a arquitetura do *Discovery Service*, como apresentado em [Cacho 04].

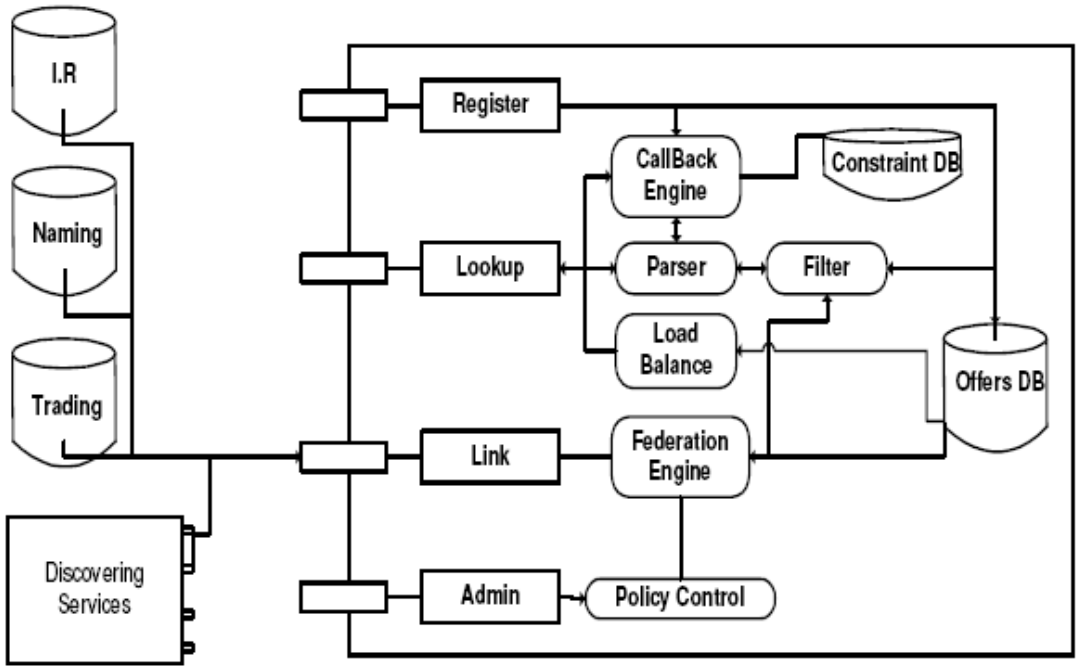


Figura 20 - Arquitetura do *Discovery Service*

As principais interfaces para o serviço de seleção dinâmica são: *Register, Lookup, Link* e *Admin*.

A interface *Register* é responsável por armazenar as novas ofertas de serviço no *OffersDB*. Essa interface agrupa novos serviços de acordo com suas propriedades funcionais e não-funcionais. Esse agrupamento será utilizado pelo módulo de balanceamento de carga (*LoadBalance*) para determinar o componente que satisfaz o requisito especifico. Outra função da interface *Register* é informar ao *Callback Engine* que novas ofertas de serviços foram registradas.

A interface *Lookup* é responsável por consultar os repositórios utilizados pelo serviço de descoberta para localizar objetos que satisfaçam determinados critérios. Dois tipos de consultas podem ser feitas síncronas e assíncronas. Para consultas síncronas, a interface *Lookup* invoca o módulo *Parser* que decompõe os critérios de busca e os encaminha para o módulo *Filter* processar os resultados. Para consultas assíncronas, a interface *Lookup* invoca o módulo *CallBack Engine* de forma a registrar o critério de busca no *constraint DB*. Esse módulo usa o módulo *Parser* para encontrar componentes que satisfazem os critérios estabelecidos. Após processar os resultados, a interface

Lookup retorna a lista de identificadores de ofertas. O balanceamento de carga é opcional e é usado apenas quando o cliente especifica na restrição da busca que o resultado deve conter os serviços menos sobrecarregados. Nesse caso, quando o módulo *Parser* termina o processo de filtragem, ele encaminha os resultados para o módulo *Load Balance*. Esse módulo ordena os resultados usando um mecanismo *roundrobin*. Depois da ordenação, o módulo *Load Balance* retorna os resultados para a interface *Lookup*. Para manter o controle do uso dos serviços, o módulo *Load Balance* mantém um contador de uso que é incrementado sempre que o método *getServiceReference* é invocado para um determinado identificador de oferta *(OfferId)*.

A interface *Admin* mantém os mecanismos de proteção interna do Serviço de Descoberta. O módulo *Policy Control* pode limitar o uso dos recursos de forma a evitar uma sobrecarga na busca que trata uma grande quantidade de dados. Os parâmetros *max return offers* e *max match card* são usados para definir os limites.

A interface *Link* permite a interação com outros Serviços de Descoberta e com os três repositórios CORBA: Nomes, *Trading* e o Repositório de Interfaces. Ele também permite a propagação da busca para outros mecanismos.

O *Discovery Service* não só possui os requisitos necessários para um serviço de seleção dinâmica, de acordo com a seção 3.4, como também possui mais funcionalidades, como o balanceamento de carga e busca assíncrona. Isso permite que outros serviços possam ser utilizados desde que tenham uma interface comum, de acordo com o especificado na seção 3.4.

Os serviços CORBA só podem ser acessados se possuírem uma interface comum, IDL, que não contém detalhes de implementação. O *Discovery Service* baseia-se principalmente no repositório de interfaces para fazer as buscas por serviços com base em propriedades como nome e assinatura de métodos. Dessa forma, não é necessária a construção de um mecanismo que crie uma interface comum para o serviço, como mostrado na seção 3.5, onde o *bridge* do lado servidor deve inspecionar os serviços publicados, construir uma representação para a interface do serviço e em

seguida notificar o serviço de seleção dinâmica. No caso do CORBA para que um serviço seja publicado é necessário que o mesmo possua essa interface comum. Então cabe ao *bridge* do lado do servidor, a cada vez que um componente registre uma implementação para uma interface publicada, notificar o serviço de seleção dinâmica, tornando aquele serviço disponível para consulta posteriormente.

Na Figura 21 temos o trecho de código referente ao método *search* do bridge, que é responsável por delegar ao serviço de seleção dinâmica.

```
1 function search(conditions)
2     components = {}
3     components = Lookup:search(conditions)
4     local result = {}
5     for i,id in ipairs(components) do
6           local object = Lookup:getServiceReference(id)
7           if (object) then
8                 table.insert(result,1,object)
9           end
10    end
11    return result
12 end
```

Figura 21 - Código de busca de componentes

Na linha 1 temos a declaração do método *search* que recebe como parâmetro um *String* contendo as condições a serem passadas ao serviço de seleção dinâmica. Na linha 2 é criada uma tabela Lua para receber a lista de referências dos objetos encontrados. Na linha 4 é declarada uma tabela Lua que receberá as referências dos objetos encontrados. Na linha 5 a 9 a lista de componentes encontradas é percorrida. Na linha 6 o método *getServiceReference* retorna a referência(IOR) do objeto CORBA e caso a mesma não seja nula é adicionada a tabela *result* que é retornada.

A figura 22 mostra o trecho de código do *bridge* do método *execute* que recebe como parâmetro o nome do método a ser localizado e os parâmetros para execução desse método.

```
1 function execute(service_name, service_signature)
2     conditions = "(operationame == '"..service_name.."') "
3     components = {}
4     components = Lookup:search(conditions)
5     local flagExe = 0
6     for i,id in ipairs(components) do
7           local object = Lookup:getServiceReference(id)
8           if (object) then
9                 flagExe = 1
10                local ior_obj = Lookup:getServiceIOR(id)
11                local OfferDescrible = Register:describe(id)
12                local Idl_service =
OfferDescrible["description"]["id"]
13                result = doSolicitation(object,ior_obj,
                  Idl_service,service_name,  service_signature)
14                return unpack(result)
15          end
16    end
17    if (flagExe == 0) then
18          print("No component was found to
execute:",service_name)
19    end
20 end
```

Figura 22 - Implementação do método execute

A linha 1 contém a declaração do método *execute* que recebe dois parâmetros: o nome do método e os parâmetros a serem passados para execução. Na linha 4 é feita a consulta ao serviço de seleção dinâmica passando como parâmetro a *String* com o critério de busca que é o nome do método a ser executado. Na linha 5 é declarado um *flag* para verificar se foi possível encontrar algum objeto que atenda os critérios de seleção que esteja disponível para receber a chamada do método. Entre as linhas 6 a 16 a lista de componentes é percorrida. Na linha 10 a referência (IOR) do componente é recuperada em seguida, nas linhas 11 e 12 a descrição da IDL(nome) também é recuperada para que o *bridge* possa executar corretamente o método. Na linha 13 o método *doSolicitation*, que tem como função executar a chamada ao método consultado, recebe como parâmetro a referência IOR do objeto que implementa o método, o nome da IDL que descreve o serviço, o nome do método a ser executado e a lista de parâmetros a serem passados para execução do mesmo. Caso nenhum componente tenha sido encontrado para execução, é exibida uma mensagem informando que não existe componente disponível para execução.

4.3 Bridge e Mecanismo de seleção dinâmica para EJB

Um dos principais motivos para que CORBA não tenha se tornado a principal plataforma de middleware disponível no mercado foi a falta de simplicidade no desenvolvimento de aplicações distribuídas e também a falta de uma implementação de referência para a especificação CORBA. Os componentes EJB principalmente em sua versão 3.0 trouxeram uma novo paradigma no desenvolvimento de aplicações distribuídas, facilitando sobretudo a vida do desenvolvedor, que não precisa mais conhecer detalhes de baixo nível, como controle de persistência, controle de transações, segurança dentre outros.

O *bridge,* como visto na seção 3.5, tem como uma das funções inspecionar os componentes instalados no servidor e construir uma representação da interface de acesso aos mesmos, tal qual existe a IDL para CORBA. Como aplicações EJB são construídas para serem acessadas por aplicações Java, não se faz necessário construir tal representação, cabendo somente ao cliente saber a referência do *bean* que foi instalado no servidor e ter os devidos arquivos *.class* no lado do cliente. Na seção 4.2 vimos que o mecanismo de seleção dinâmica utiliza o repositório de interfaces para consultar serviços com base em critérios como nome e assinatura de métodos. No caso dos EJB´s utilizaremos uma estratégia parecida, onde para cada componente instalado no servidor, o *bridge* do lado do servidor deve inspecionar esse componente e construir uma representação da interface de acesso a esse componente e armazenar essa representação dentro de um repositório, que é acessível pelo mecanismo de seleção dinâmica. Para analisar o conteúdo do componente é feito uso da API Java java.reflect, que permite inspecionar definições de classes e interfaces. Quando um componente é instalado no servidor, o bridge deve receber como entrada o nome completo (ex: *br.ufrn.bean.IInterface*) da interface do bean. Em seguida a análise é feita de maneira decrescente, partindo da interface, inspecionando os membros e em seguida os métodos da interface. Para evitar que todas as classes referenciadas pela interface fossem inspecionadas, o que poderia causar um overhead muito alto, não é feito a análise de herança das classes e interfaces. Isso implica em dizer que, dada uma interface A que herda de uma

interface B, apenas os métodos declarados em A estarão disponíveis para consulta, através do serviço de seleção dinâmica. Para que essa limitação não ocorra, o grau de inspeção pode ser definido pelo desenvolvedor que realiza a instalação do componente, de tal forma que ele pode especificar quantos níveis de herança o desenvolvedor deseja que sejam inspecionados. No caso das interfaces A e B, a representação da interface A mostraria os métodos da interface A e B, em vez de ser criada uma representação para interface A e outra para B.

A Figura 23 mostra um trecho de código da classe *Inspect* que implementa a interface *IInspect* exibida no diagrama de classes da Figura 15. No trecho de código é evidenciando o comportamento do método *inspectMethod*.

```
1 public Metodo inspectMethod(Method m,int nivel,Service owner){
2           Metodo met=new Metodo();
3           met.setModificador(m.getModifiers());
4           met.setNome(m.getName());
5 met.tipoRetorno=inspectService(
m.getReturnType().getCanonicalName(),nivel+1);
6           met.tipoRetorno.dependecies.add(owner);
7           Class<?> parameters[]=m.getParameterTypes();
8           for(int i=0;i<parameters.length;i++){
9 Classe p = inspectService(parameters[i].getCanonicalName(),
nivel+1);
10                p.dependecies.add(owner);
11                met.parametros.add(p);
12          }
13          Class<?> exceptions[]=m.getExceptionTypes();
14          for(int i=0;i<exceptions.length;i++){
15                if
(!(exceptions[i].getCanonicalName().equals("java.rmi.RemoteExceptio
n"))){
16                      Classe p=new Classe();
17                      p.setNome(exceptions[i].getCanonicalName());
18                      p.setPrimitivo(false);
19                      p.setExcecao(true);
20                      mapa.put(p.getNome(), new Token(p,nivel));
21                      p.dependecies.add(owner);
22                      met.excecoes.add(p);
23                }
24          }
25          return met;
26    }
```

Figura 23 - Código do método *inspectMethod*

A linha 1 contém a declaração do método *inspectMethod*, que recebe como parâmetros, uma instância da classe *java.lang.reflect.Method* que contém as informações sobre um método Java, tais como modificadores, assinatura do

método, o nível da análise, que se for 0 indica que não é necessário fazer analise de informações sobre tipos de retorno e os parâmetros do método analisado e, por fim, um objeto da classe *Service*. Tal classe foi exibida na Figura 15 da seção 3.5. Nas linhas 2 a 4 é criada uma instância da classe método e armazenado as informações sobre os modificadores e nome do método. Na linha 5 é feita uma chamada ao método *inspectService* recebendo como parâmetro a referência ao tipo de retorno do método inspecionado e o nível. Na linha 7 é declarado um array de *Class<?>* que contém as informações sobre os parâmetros que são passados para o método atualmente analisado, nas linhas 8 a 12 então para cada parâmetro passado é feito a análise sobre o tipo do mesmo. Em seguida, nas linhas 13 a 24, são analisadas as exceções que o método pode retornar. No caso de exceções não analisamos aspectos relacionados a cadeia de heranças de uma exceção.

Após as classes serem inspecionadas é realizado o mapeamento para que seja construída uma representação da interface acessível pelo serviço de seleção dinâmica. A figura 24 contém um exemplo da representação XML obtida para a interface do *bean Bank*, que será discutido no estudo de caso do capítulo 5.

```
1 <classe name="br.ufrn.bank.interfaces.BankInterface"
beanname="BankBean">
2         <method name="criarConta"
returnType="br.ufrn.bank.model.ContaBancaria">
3             <parameter type="int"/>
4             <parameter type="java.lang.String"/>
5             <parameter type="java.lang.String"/>
6         </method>
7         <method name="criarConta" returnType="void">
8             <parameter type="int"/>
9         </method>
10         <method name="getConta"
returnType="br.ufrn.bank.model.ContaBancaria">
11             <parameter type="int"/>
12        </method>
13 </classe>
```

Figura 24 - Representação XML de um Bean

A linha 1 mostra o elemento *classe* que representa uma classe ou interface Java. Esse elemento tem pelo menos um atributo que é o nome completo da classe ou interface e, sendo uma interface de bean, há outro atributo chamado *beanname*, que diz respeito a referência do bean no serviços

de nome e diretório (JNDI). Nas linhas 2 a 12 há vários métodos pertencentes a interface *br.ufrn.bank.interfaces.BankInterface*. Na linha 2 há o elemento método que guarda as informações sobre o método *criarConta,* que retorna um objeto da classe *br.ufrn.bank.model.ContaBancaria*, recebendo como parâmetros um inteiro e duas *Strings*.

Uma vez realizado o mapeamento, a representação obtida é armazenada no servidor onde o bridge e o serviço de seleção dinâmica estão instalados. Agora o serviço de seleção dinâmica está apto a receber consultas. O processo de consulta a um serviço é ilustrado através do diagrama de seqüência da Figura 25:

1. O cliente realiza uma chamada ao método *lookup* da classe *InitialContext* que implementa a interface IContext, responsável por acessar o serviço de nomes e diretórios (JNDI), passando como parâmetro o nome do bean que representa o serviço de seleção dinâmica, que então retorna a referência desse objeto.
2. De posse da referência o método *narrow* da classe *PortableRemoteObject* é invocado para que seja possível transformar a referência é um objeto que representa a interface *Remote* do bean do serviço de seleção dinâmica, chamada de ILookup
3. O cliente então invoca o método *search*, passando como parâmetro o critério de busca, da interface *ILookup*, que é transmitida para a implementação de fato da interface localizada no servidor
4. A implementação da interface *ISearchService*, apresentada na figura 14, recebe a chamada e delega ao mecanismo de parser, que realiza a analise léxica do critério de busca e o transforma para que o mecanismo do XPATH possa interpretá-lo.
5. Com o critério de busca devidamente analisado, a classe que implementa *ISearchService* invoca o método query da classe *XPATH* que consultará o repositório de interfaces que contém as representações XML dos beans instalados no servidor, retornando

em caso positivo a referência para o serviço que foi selecionado com base nos critérios de busca.

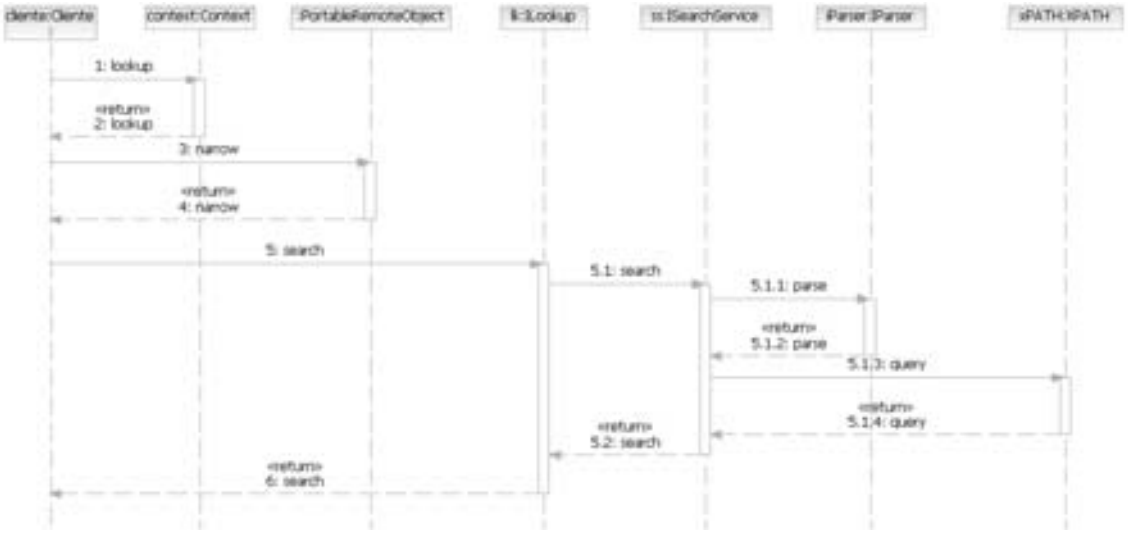


Figura 25 - Diagrama de seqüência Seleção dinâmica EJB

O retorno da execução do mecanismo de seleção dinâmica é a entrada do bean no JNDI e o nome da interface *remote* do mesmo. De posse dessas duas informações é possível criar uma referência ao bean do lado do cliente.

4.4 Mecanismos de Binding

4.4.1 LuaOrb

O *binding* utilizado para plataforma CORBA foi o LuaOrb [Cerqueira 99] baseado na interface de invocação dinâmica de CORBA (DII) que possibilita o acesso dinâmico a componentes CORBA via Lua da mesma forma que se utiliza qualquer objeto Lua. Além disso, LuaOrb usa a interface de esqueleto dinâmico (DSI) para permitir a instalação dinâmica de objetos Lua como servidores CORBA. LuaOrb utiliza os *metatables* para que as chamadas sobre os objetos CORBA possam ser direcionadas para que esse *binding* possa tratá-las.

O trecho de código da Figura 26 mostra uma das utilizações do LuaOrb, para criar um *Proxy* para o serviço de seleção dinâmica, permitindo que as requisições para tal serviço sejam delegadas. Esse trecho está no construtor do *bridge* para plataforma CORBA.

```
1 local search_service=luaorb.createproxy(
                        readIOR("./search.ref"),
                        "IDL:CosDiscovering/SearchComponents:1.0")
2     Lookup = search_service.getLookup
```

Figura 26 - Utilização do *binding* LuaOrb

Na linha 1 o método *createProxy* é invocado passando como parâmetros a referência IOR do serviço de seleção dinâmica e também o nome completo da IDL que contém a definição do serviço, o método então retorna a referência ao serviço de seleção dinâmica. Na linha 2 é recuperada referência à implementação da classe *ILookup*, que recebe a requisição de busca feita pela camada multi-middleware.

4.4.2 LuaJava

LuaJava [Cassino 99] é uma ferramenta de *scripting* que permite objetos Java serem manipulados como objetos Lua e vice-versa. O fato da linguagem ser dinamicamente tipada e a possibilidade de adicionar métodos a uma tabela Lua após a mesma ter sido declarada, facilita a integração com objetos Java. LuaJava utiliza o conceito de *proxy*, que consiste em criar uma representação para um objeto Java, que então pode ser acessado da mesma forma que se acessa um objeto Lua, permitindo com que o desenvolvedor não precise conhecer detalhes de sintaxe da linguagem Java para trabalhar com objetos Java.

Existem basicamente duas chamadas de métodos que são feitas utilizando o LuaJava: *newInstance* e *bindClass*, que servem respectivamente para instanciar um novo objeto e criar o *proxy*, que é a representação Lua do objeto Java, e para criar uma associação com uma classe, sem instanciar um objeto da mesma classe, permitindo dessa forma acessar membros estáticos de classes. O trecho de código apresentado na Figura 27 que utiliza diretamente as chamadas de métodos citadas anteriormente.

```
1 bridge=luajava.bindClass("br.ufrn.luaspace.Bridge")
2 param=luajava.newInstance("br.ufrn.luaspace.Parameters")
3 bridge:search(criteria)
```

Figura 27 - Utilização do binding LuaJava

A linha 1 contém a chamada ao método *bindClass*, passando como parâmetro o nome da classe que representa a implementação Java do *bridge*, que deve invocar o serviço de seleção dinâmica implementada na linguagem Java. A linha 2 contém a chamada ao método *newInstance*, passando como parâmetro o nome da classe que tem como função empacotar os parâmetros que são passados para *bridge* no momento da execução de um método de negócio de um objeto distribuído cuja referência será retornada pelo serviço de seleção dinâmica.

Um aspecto fundamental que deve ser levado em conta é como funciona a integração entre a biblioteca LuaOrb e LuaJava. A ferramenta *LuaJava* para acessar objetos Lua utiliza a *Java Native Interface* para acessar código de aplicações nativas, tais como o LuaOrb, que é escrito em C++. O LuaJava acessa a API Lua escrita em C, através do JNI. Para executar uma aplicação utilizando LuaJava é preciso iniciar uma máquina virtual Java, que então é responsável por delegar as chamadas à API Lua. Como LuaOrb é uma biblioteca, escrita em C++, então é possível invoca-la através do LuaJava, utilizando a JNI.

4.5 Plugin Eclipse

O primeiro aspecto a ser observado ao desenvolver um plugin para o Eclipse é verificar quais pontos o plugin irá contribuir para com a plataforma. Ao desenvolver um plugin que irá dar suporte ao ambiente multi-middleware, onde os programas são escritos na linguagem Lua, que não possui suporte nativo na IDE Eclipse e tendo em vista as necessidades que devem ser satisfeitas para utilização das ferramentas que compõem o ambiente, os pontos que devem ser estendidos para satisfazer os requisitos são os seguintes:

- Editor de texto que dê suporte a edição de códigos Lua, com destaque de coloração de sintaxe, *code completion* e demais utilitários comuns a um editor de código;
- Configuração de execução, que define como os programas serão executados, configurando os programas externos ao Eclipse que devem ser invocados para execução correta dos programas;

- Assistentes que auxiliam o programador a gerar código referente à utilização das ferramentas que compõe o ambiente;

- Páginas de preferências que são utilizadas para configurar as preferências do plugin, tais como localização de ferramentas, localização de programas executáveis, as cores utilizadas para destaque de sintaxe;

- Visões que são janelas que compõem o ambiente e têm o intuito de prover outras formas de interação com o desenvolvedor

O trecho de código da figura 28 ilustra o ponto de extensão que adiciona o plugin ao editor para o ambiente.

```
1 <extension
         point="org.eclipse.ui.editors">
2       <editor
             class="br.ufrn.luaspace.ui.editors.LuaEditor"
             contributorClass="
br.ufrn.luaspace.ui.editors.LuaEditorActionContributor"
             default="true"
             extensions="lua"
             icon="icons/file.gif"
             id=" br.ufrn.luaspace.ui.editors.LuaEditor"
             name="Lua File Editor">
3       </editor>
4    </extension>
```

Figura 28 - Trecho de código XML para extensão do editor

Na linha 1 é definido que o ponto a ser estendido é de um editor, através da tag XML *extension* e o atributo *point* informa qual será o ponto de extensão. Na linha 2 a tag editor possui uma série de atributos: a classe que herda da classe *TextEditor* que representa uma editor básico de texto, a classe que representa uma Action do Eclipse, permitindo a utilização de menus pop-up dentro do editor de texto, se esse editor deve ser utilizado sempre para a extensão lua, o ícone que deve aparecer na aba e no arquivo que ficará na área de projeto, a identificação do editor e o nome do editor.

A figura 29 mostra um *screenshot* do ambiente sendo utilizando no Eclipse. Nas áreas em destaque encontra-se o editor de texto que além de suporte a coloração de sintaxe é possível utilizar *code completion*, para acessar as bibliotecas Lua existentes, além de métodos específicos dos *binding LuaOrb* e

LuaJava. Foi construída também uma *view* para consultar os serviços de seleção dinâmica em tempo desenvolvimento, permitindo que o desenvolver saiba quais são os serviços que estão disponíveis naquele momento.

Figura 29 - Ambiente de Desenvolvimento no Eclipse

4.5.1 Editor

A figura 30 mostra o diagrama de classes correspondente à implementação do editor.

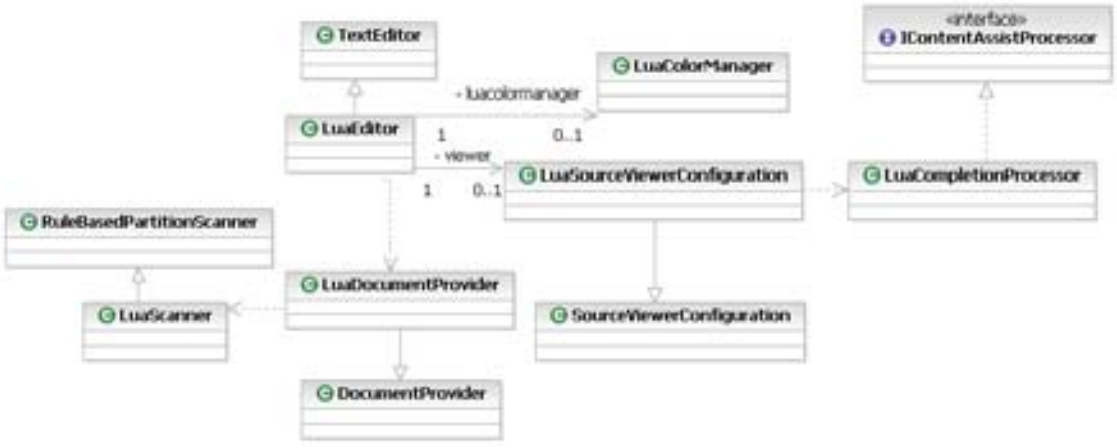


Figura 30 - Diagrama de classes da implementação do editor

A classe *LuaEditor* representa a classe principal da implementação, essa classe que será atribuída a função de implementar um editor de texto. A classe *LuaEditor* herda da classe *TextEditor* que é uma classe padrão da arquitetura do Eclipse para implementação de editores de texto, oferecendo funcionalidades básicas de um editor, como recursos de copiar, colar e desfazer. Para gerenciar o esquema de cores do editor, permitindo o destaque das palavras reservadas da linguagem Lua, comentários, strings e números, é utilizada a classe *LuaColorManager*. Para gerenciar o comportamento de um editor, no que diz respeito à sua visualização, é necessário definir uma classe que herde da classe *SourceViewerConfiguration*, classe essa responsável por gerenciar a coloração de sintaxe, auto-identação e completação de código. A classe *LuaSourceViewerConfiguration* implementa este comportamento e o auxilio da classe *LuaCompletionProcessor*, realiza o *code completion*. A classe *LuaDocumentProvider* tem a função de gerenciar o documento que está sendo editado no editor no momento, dessa forma é possível ter um modelo do documento sendo editado, que pode ser escaneado para localizar as palavras reservadas e assim realizar o destaque dessas palavras. Note que o modelo aqui segue o princípio do padrão MVC (*Model-View-Controler*), onde o editor em si mostra a visão (View), através do *SourceViewerConfiguration*, do modelo (Model) do documento que está sendo editado.

4.5.2 Núcleo de Execução

A figura 31 mostra o diagrama de classes da parte central do plugin responsável pela coordenação de execução de aplicações dentro do ambiente multi-middleware de gerenciamento de recursos (arquivos, projetos...).

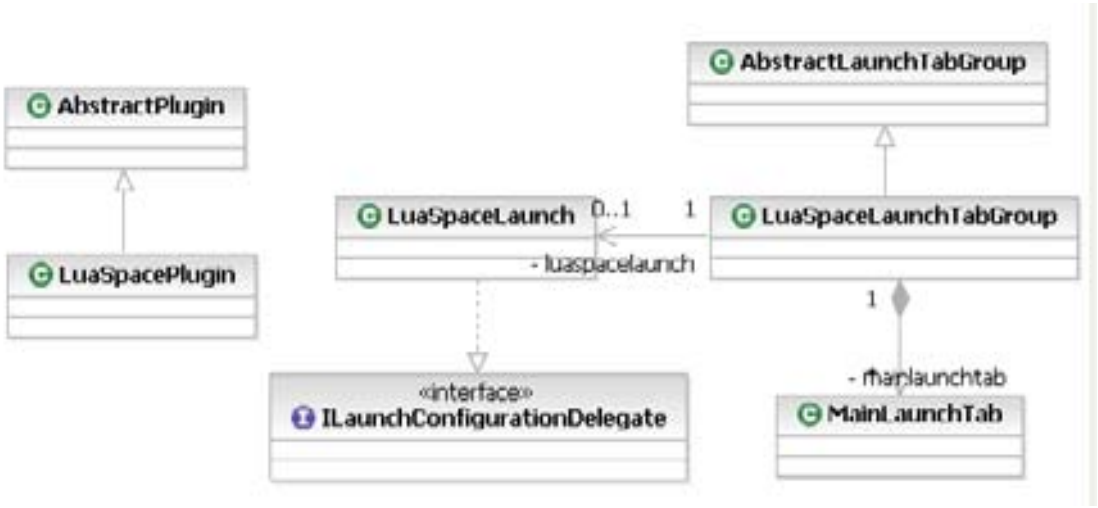


Figura 31 - Diagrama de classes do núcleo de execução

A classe estática *LuaSpacePlugin* é a classe central do plugin. Essa classe é responsável por controlar o ciclo de vida do plugin, tendo acesso aos recursos utilizados pelo plugin, configurações externas e realização de log das operações executadas internamente (dentro do plugin). A classe *LuaSpaceLaunch* implementa a interface *ILaunchConfigurationDelegate* responsável pela chamada de execução do *LuaOrb*. A classe *LuaSpaceLaunch* recebe a configuração de execução da classe *LuaSpaceTabGroup* que representa a parte UI onde o programador configura as informações de execução, tal como arquivo a ser passado para o interpretador. A figura 32 ilustra esse processo.

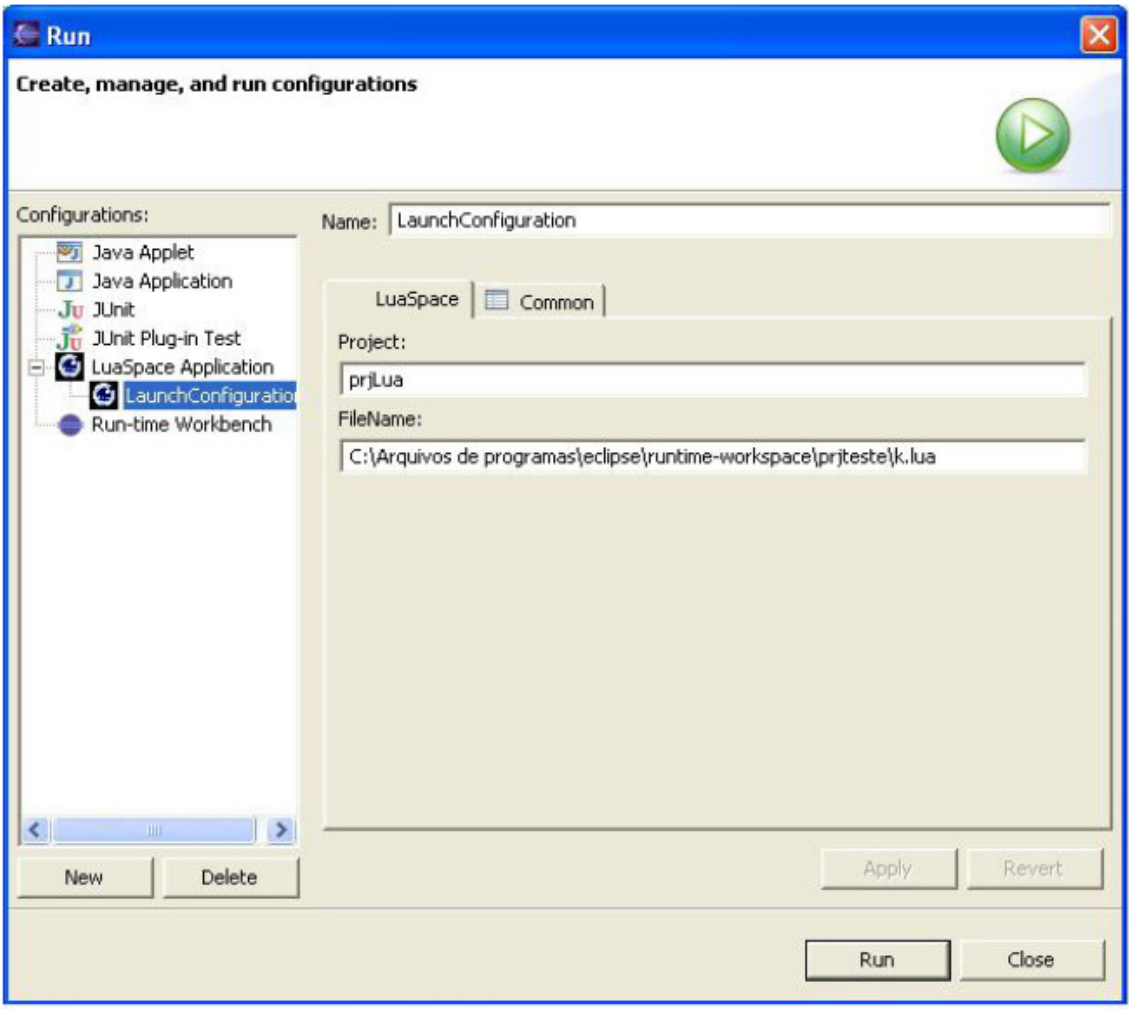


Figura 32 - Configuração de Execução

Através da tela apresentada na figura é solicitada a interação do programador para informar dois aspectos: qual projeto que deve ser executado e o arquivo a ser repassado para o interpretador Lua configurado. Caso o programador não informe os parâmetros, será selecionado o projeto e arquivo que estão sendo editados na tela principal (comportamento padrão do configurador). Quando o botão *run* é pressionado a configuração editada é repassada para classe *LuaSpaceLaunch* que realiza os procedimentos necessários.

4.5.3 Navegador de Interfaces

Os mecanismos de seleção dinâmica são as ferramentas do ambiente multi-middleware que permitem seleção dinâmica baseada em critérios de busca. Além disso, é possível navegar pelas interfaces que estão nos repositórios configurados para armazenar objetos. No plugin foi implementada

uma *View* que permite adicionar mais funcionalidades ao ambiente de desenvolvimento. A figura 33 mostra um *screenshot* da *view* para realizar seleção dinâmica em tempo de desenvolvimento.

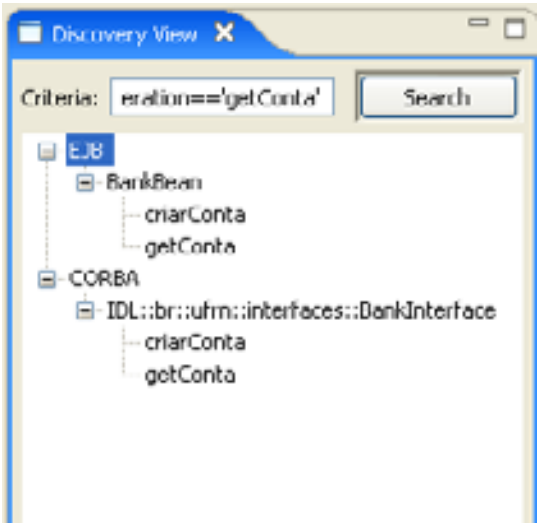


Figura 33 - Discovery View

O mecanismo de seleção dinâmica executa em paralelo à interface gráfica e trata todas as requisições de busca enviadas pela interface, retornando o vetor de objetos que descreve as interfaces que satisfazem os critérios de seleção, incluindo os métodos e demais atributos que a interface possua. Como os serviços de seleção dinâmica são acessados via o bridge do lado cliente, então através do LuaJava é feito o envio do critério de busca aos bridges que estão instalados no ambiente, no caso da implementação de referência, bridges para CORBA e EJB. Então para cada bridge instalado é feita a notificação ao serviço de seleção dinâmica respectivo e o resultado é recebido pelo plugin, que então constrói uma lista contendo o resultado da busca, permitindo que o desenvolvedor possa consultar os repositórios existentes em tempo de desenvolvimento. Para que isso seja possível os *bridges* precisam estar corretamente inicializados e configurados.

4.6 Desenvolvendo clientes para o ambiente multi-middleware

4.6.1 Clientes escritos em Lua

Como a camada multi-middleware foi desenvolvida utilizando a linguagem Lua clientes escritos na mesma linguagem são triviais. O seguinte diagrama de

atividades mostra o fluxo de passo necessário para se utilizar o ambiente multi-middleware através de um cliente Lua.

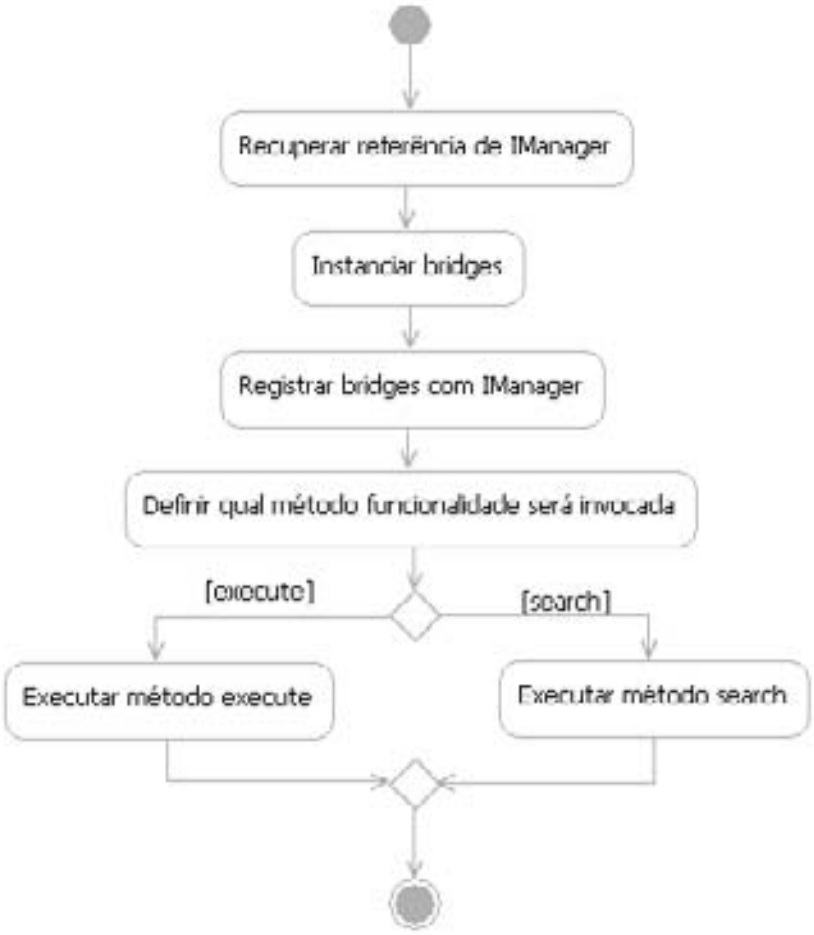


Figura 34 - Passos para utilização do ambiente utilizando Lua

Os passos são os seguintes:

1. Recuperar a referência ao *IManager*: Primeiro devemos recuperar a referência ao gerenciador de requisições da camada multi-middleware que receberá os registro dos *bridges* e delegará aos mesmos as requisições realizadas;
2. Instanciar bridges: O desenvolvedor deve instanciar os bridges disponíveis para registrar na camada multi-middleware. No caso da implementação de referência devemos instanciar *BridgeCorba* e *BridgeEJB*;
3. Registrar *Bridges* com *IManager*: O método *register* deve ser invocado passando como parâmetro a instância do bridge que se deseja registrar com a camada multi-middleware;

4. Definir qual funcionalidade será invocada: como visto anteriormente através da camada multi-middleware é possível realizar a consulta a referência de um serviço com base em critérios de busca ou executar um método remoto com base na sua assinatura.

Ao final teremos o resultado da execução.

4.6.2 Clientes não escritos em Lua

Clientes não escritos em Lua precisam construir uma camada que acesse o código Lua da forma como foi descrito na seção anterior. Para tal, será necessária a utilização de algum mecanismo de *binding* para realizar o mapeamento entre as requisições Por exemplo, se tivéssemos um cliente escrito em C++ que precisa acessar a camada multi-middleware para realizar uma consulta ao serviço de seleção dinâmica deveríamos utilizar o LuaBind [Wallin 03], que consiste em uma biblioteca que permite a criação de aplicações que acessem código Lua e vice-versa. Dessa forma, deveria ser construída uma camada que receberia as requisições do cliente C++ e, através do LuaBind, acessaria o cliente descrito na seção anterior passando como parâmetro os critérios de busca/execução desejados.

ESTUDO DE CASO E ANÁLISE DE DESEMPENHO

Neste capítulo apresentamos um estudo de caso, que consiste em uma aplicação bancária que visa mostrar as potencialidades do ambiente e das ferramentas associadas e também fazer uma análise de desempenho, comparando o tempo de execução de chamadas remotas dentro e fora do ambiente *multi-middleware*.

4.7 Estudo de Caso

Nesta seção discutiremos um estudo de caso que mostra as potencialidades do ambiente *multi-middleware* e consiste de uma aplicação bancária para o Banco Money. O banco foi formado a partir da aquisição de vários bancos menores. Os sistemas de informação desses bancos compartilham a mesma base de dados, porém foram implementados utilizando plataformas de *middleware* diferentes, existindo uma implementação em CORBA e outra em EJB 3.0. A figura 35 mostra o diagrama de classes da aplicação bancária considerada.

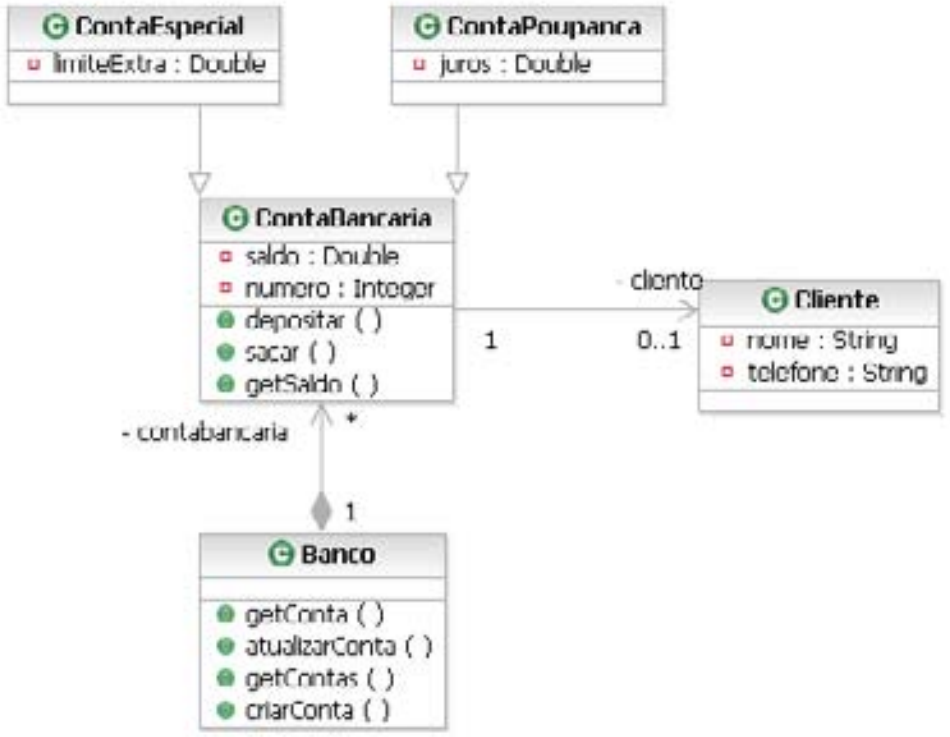


Figura 35 - Diagrama de classes da aplicação bancária

Na aplicação desenvolvida com o uso de CORBA as classes Cliente, *ContaBancaria*, *ContaEspecial* e *ContaPoupança* são especificadas através de *valuetypes* e classe Banco como uma interface CORBA. A figura 36 mostra a especificação IDL da implementação CORBA.

```
1 valuetype Cliente{
2     public string name;
3     public string telefone;
4     factory create(in string nome,in string telefone);
5 };
6 valuetype ContaBancaria {
7  factory create (in unsigned short numero,in double saldo,in
entity::Cliente cliente);
8  void depositar(in double valor);
9  void sacar(in double valor);
10  double getSaldo();
11  public entity::Cliente cliente;
12  public unsigned short numero;
13  public double saldo;
14 };
15 module br{
16 module ufrn{
17 module interfaces{
18 interface BankInterface{
19 ::entity::ContaBancaria getConta(in short numero);
20 void atualizaConta(in ::entity::ContaBancaria conta);
21 sequence<::entity::ContaBancaria> getContas(in short qtd);
22 ::entity::ContaBancaria criarConta(in string nome,in string
telefone,in unsigned short numero);
23 };};};};
```

Figura 36 - Especificação IDL da implementação CORBA

Nas linhas de 1 a 14 temos a declaração dos *valuetypes* que representam respectivamente as classes de entidade Cliente e ContaBancaria. ValueTypes representam o estado de um objeto, ou seja, podem ser passados por valor, porém sua implementação é local, no sentindo que o cliente precisa disponibilizar uma implementação para os métodos que compõem o *valuetype*. O método *create* funciona como um construtor, por isso a palavra *factory* (linhas 4 e 7) antes do método indica que o método constitui um inicializador para o objeto.Na linhas 15 a 23 está a declaração da interface *BankInterface* que representa a classe Bank do diagrama de classe da Figura 35. Na linha 19 temos o método *getConta* que recebe como parâmetro um número e retorna a conta bancária equivalente. Na linha 20 o temos o método *atualizaConta* que recebe como parâmetro uma conta bancária que deve ser atualizada. Na linha 21 o método *getContas* que recebe o número de contas que devem ser criadas e retorna uma seqüência de objetos do tipo *ContaBancaria* e na linha 22 a

declaração do método *criarConta* que recebe como parâmetros o nome e telefone do cliente e o número a ser atribuído à conta bancaria. O ORB (*Object Request Broker*) utilizado nessa implementação é o MICO na versão 2.3.11 e o servidor foi implementado utilizando-se a linguagem Lua.

Na implementação Java do diagrama de classes da Figura 34, foi utilizado a versão 3.0 da especificação EJB, uma vez que reduz substancialmente o número de interfaces e o processo de instalação do EJB, uma vez que não se faz mais necessário escrever descritores de implantação. O servidor de aplicação utilizado foi o JBOSS na versão 4.2, que é a primeira versão desse servidor a oferecer suporte a especificação 3.0 do EJB. A figura 37 apresenta o trecho de código referente a interface *remote* referente a classe *Bank*.

```
1 @Remote
2 public interface BankInterface {
3     public ContaBancaria criarConta(String nome,String telefone,
int numeroConta);
4     public ContaBancaria getConta(int numeroConta);
5     public void atualizarConta(ContaBancaria c);
6     public List<ContaBancaria> getContas(int qtd);
}
```

Figura 37 - Interface Remote do Bean Bank

Na linha 1 temos a anotação que identifica a interface como *Remote*. Na linha 3 temos a declaração do método *criarConta* recebendo como parâmetro o número da conta, nome e telefone do futuro cliente. A linha 4 temos o método *getConta* que recebe como parâmetro o número da conta. As linhas 5 e 6 apresentam os métodos *atualizarConta* e *getContas*.

Com as implementações devidamente instaladas, agora é necessário registrar as mesmas com o serviço de seleção dinâmica equivalente para cada plataforma, procedimento este feito de forma automática para EJB, porém para a aplicação CORBA é necessário registrar manualmente o *servant* no serviço de seleção dinâmica. Primeiro passo é criar um projeto Lua dentro Eclipse e em seguida adicionar o arquivo *main.lua* que conterá o código da aplicação principal. A figura 38 mostra um *screenshot* do assistente utilizado para adicionar um o arquivo *main.lua* ao projeto.

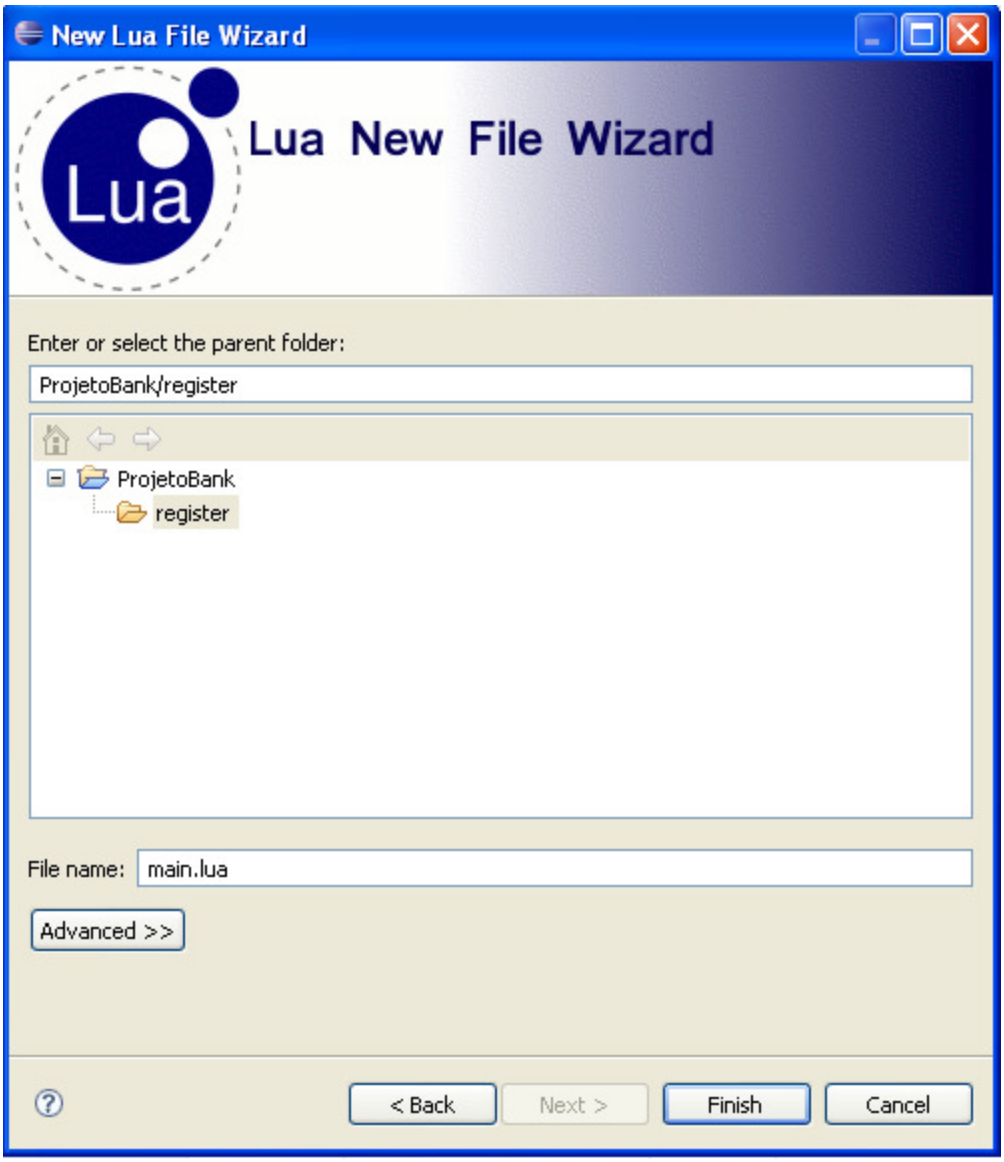


Figura 38 - Assistente para adicionar novo arquivo Lua

O primeiro procedimento que deve ser realizado é o registro dos *bridges* para as plataformas CORBA e EJB. Em seguida devemos localizar os componentes que possuam o método *getConta* ou *criarConta*, independente dos parâmetros que sejam passados. A figura 39 apresenta o código da aplicação principal.

```
1 dofile("BridgeCorba.lua")
2 dofile("BridgeEJB.lua")
3 ManageService:getInstance():register(BridgeCORBA())
4 ManageService:getInstance():register(BridgeEJB())
5 local
resultados=ManageService:getInstance():search("operationname=='cria
rConta' || operationname=='getContas'")
6 for k,component in pairs(resultados) do
7     cc=component:criarConta("André","3206-3265",1)
8     cc:depositar(100)
9     io.write(cc:getSaldo())
11 end
```

Figura 39 - Código da aplicação Principal

Nas linhas 1 e 2 os bridges para as plataformas CORBA e EJB são carregados. Nas linhas 3 e 4 os bridges são registrados na camada multi-middleware. Na linha 5 o método *search* da camada multi-middleware é invocado passando como parâmetro a combinação de critério de busca *operationname=='criarConta' || operationname='getContas'* que procura todos os componente que possuam o método *criarConta* ou *getContas*. O retorno é a referência aos objetos que satisfazem o critério de busca informado. Então na linha 6 é iniciado um loop pela tabela que contém a referência dos componentes. Na linha 7 o método *criarConta* é invocado, passando como parâmetro o nome, telefone do cliente e o número da conta corrente. Na linha 9 o método *depositar* da entidade *ContaBancaria* é invocado e na linha seguinte o saldo da mesma é impresso na tela.

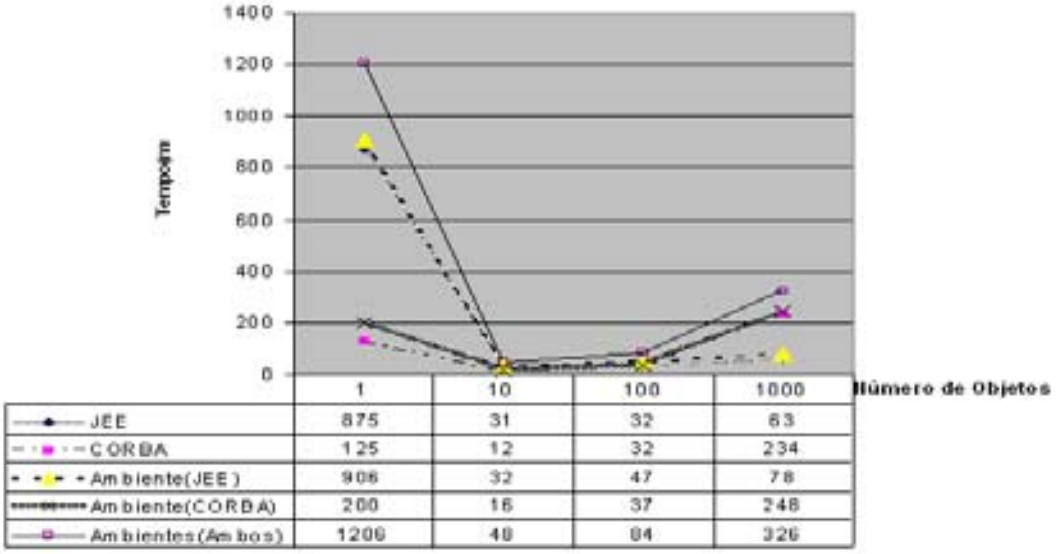
4.8 Análise de Desempenho

A Figura 40 apresenta o gráfico da análise de desempenho do ambiente *multi-middleware*. A análise foi feita utilizando o estudo de caso apresentado, acrescentando na classe *Banco* o método *getContas*, o qual recebe um inteiro com a quantidade de contas a serem criadas e, em seguida, retornadas para o cliente. Foi feita uma simulação com chamadas consecutivas a esse método, criando-se uma variável de simulação representando o número de objetos a serem retornados por ele. Tal variável foi configurada com os valores 1,10,100 e 1000. Na primeira vez que a chamada ao método foi realizada, a variável foi configurada para o valor 1 (indicando que apenas 1 objeto deveria ser retornado por *getContas*), e nessa chamada foi contado o tempo de inicialização (carregar *bean*, executar busca). Em seguida foi medido o tempo

transcorrido (em milisegundos) antes da chamada do método *getContas* e imediatamente após a chamada. Isso foi feito respectivamente para: (i) um cliente em Java do *Bean*, (ii) um cliente CORBA da implementação CORBA do estudo de caso, (iii) utilizando o ambiente *multi-middleware* somente com o *bean* instalado no servidor, (iv) novamente utilizando o ambiente com apenas a implementação CORBA executando e (v) com as duas implementações (EJB e CORBA) em execução no ambiente *multi-middleware*.

O gráfico mostra que, inicialmente, o tempo de execução em todos os cinco casos é alto devido a carga inicial dos serviços necessários para o funcionamento da plataforma de *middleware*. Por exemplo, no início da execução de CORBA são carregados os serviços de repositórios de interface e de nomes. Para as simulações com uso do ambiente *multi-middleware* é levado em conta o tempo de inicialização mencionado anteriormente mais o tempo que o serviço de seleção dinâmica específico de cada plataforma leva para fazer busca de componentes. A avaliação seguinte refere-se a execução do método *getContas* para chamadas com os valores de 10,100 e 1000 objetos a serem retornados. Podemos verificar que o tempo varia muito pouco entre as chamadas feitas a partir de uma plataforma específica (J2EE, por exemplo) e, com a mesma plataforma no ambiente *multi-middleware*. A diferença acontece somente no momento do mapeamento dos dados pelo *binding* específico e no momento inicial onde a busca é realizada pelo mecanismo de seleção dinâmica. Concluímos a partir do gráfico que o tempo total incluindo a busca dos componentes e o uso do *binding* não se distancia do tempo de execução levado pelo uso das plataformas isoladas. A diferença de tempo de execução que ocorre entre as chamadas com 100 e 1000 objetos é explicada pelo aumento na quantidade de objetos a serem retornados e relaciona-se principalmente ao retorno do método na plataforma CORBA. A complexidade relacionada ao método, o qual retorna uma lista de tipos complexos, torna a sua execução mais lenta quando utilizando a plataforma CORBA, uma vez que o suporte de tipos complexos é mais custoso em termos de desempenho, devido aos procedimentos de empacotamento e desempacotamento (*marshalling* e *unmarshalling*) utilizados pelo protocolo IIOP do CORBA. Para medir esses tempos utilizamos, antes e depois da invocação dos métodos, a

chamada *System.currentMilis()*, que retorna o número de millisegundos transcorridos de 1 de Janeiro de 1970 até o momento atual.



	1	10	100	1000
JEE	875	31	32	63
CORBA	125	12	32	234
Ambiente(JEE)	906	32	47	78
Ambiente(CORBA)	200	16	37	248
Ambientes(Ambos)	1206	48	84	326

Figura 40 - Gráfico de desempenho(Número de ObjetosxTempo de execução)

5 TRABALHOS RELACIONADOS

Neste capítulo apresentamos alguns trabalhos relacionados, dando ênfase na comparação de os Web Services e WSRF(Web Services Resource Framework), que constituem a principal alternativa existente para fornecer interoperabilidade entre plataformas de middleware.

5.1 Web Services

A tecnologia de Serviços Web (*Web Services*) provê uma solução para a integração de sistemas e a comunicação de diferentes aplicações, permitindo que dados sejam enviados/recebidos através de mensagens em formato XML.

Os Web Services constituem um mecanismo padrão para publicações e subscrições de serviços de software através de uma intranet ou Internet. Clientes desses Web Services localizam serviços através da UDDI (*Universal Discovery, Description and Integration*), onde as interfaces desses serviços são descritas utilizando WSDL (*Web Services Description Language*). Para os clientes que acessam esses serviços, parece que está sendo feito uma chamada ao método localmente, onde na verdade a requisição é transformada em XML, baseado no protocolo SOAP, e enviado pela rede para aplicação servidora que então retorna como resultado um documento XML, também baseado no protocolo SOAP.

Essa sistemática de comunicação permite flexibilidade com relação às partes envolvidas, permitindo o uso de diversos tipos de dispositivos de hardware e sistema de software que possuam suporte a comunicação utilizando XML como formato principal das mensagens enviadas. Ao tratar de objetos complexos, por exemplo, classes que não sejam simples *beans* contendo apenas dados representados por tipos primitivos (ex: inteiro, string, etc...), onde o comportamento do objeto precisa ser utilizado pelo cliente, os Web Services não satisfazem esses requisitos.

Os *Web Services* através do WSRF (*Web Services Resource Framework*) [Granham 06] podem realizar conversações que não são orientadas a objetos [Hopkins 05], nas quais o estado de um objeto é expresso

através de um documento XML, porém, o seu comportamento (métodos) não pode ser transmitido, fazendo com que o cliente precise implementar tais métodos se desejar fazer com que o comportamento siga o do objeto original. Em nossa proposta, o ambiente *multi-middleware* é orientado a objetos e pode receber objetos provenientes do retorno dos métodos, sendo tal capacidade possibilitada pelos componentes que compõem o ambiente.

O ambiente multi-middleware não se preocupa em desenvolver um protocolo de comunicação, como é feito para o Web Services que utiliza o protocolo SOAP e o protocolo de comunicação utilizado é inerente de cada plataforma de middleware, para o CORBA é utilizado o protocolo IIOP que é o padrão para essa plataforma. Dessa forma não é necessário adaptar os serviços existentes para um eventual protocolo que fosse definido pelo ambiente, como acontece com Web Services. O diferencial dos Web Services nesse ponto é a comunicação entre as plataformas de middleware sem a necessidade um ambiente comum, como é feito no ambiente multi-middleware. Em Web Services CORBA pode acessar JEE através de Web Services e vice-versa, enquanto na solução proposta nesse trabalho para que a interação ocorra é obrigatória a utilização da camada multi-*middleware*.

Através do mecanismo de UDDI é possível consultar e invocar serviços web. UDDI permite que sejam realizadas consultas aos repositórios contendo as interfaces dos serviços oferecidos. O mecanismo de seleção dinâmica não só permite a consulta a repositórios de interfaces e invocação desses serviços, como permite expressar consultas mais complexas aos serviços, através da combinação de critério de busca, utilizando uma sintaxe simples, porém poderosa, permitindo que o desenvolvedor busque serviços em diversas plataformas de middleware. A busca pode ser feita por nome de métodos, serviços e inclusive exceções lançadas por métodos, podendo combinar todas essas opções através de operadores lógicos. Além disso, é possível realizar esses procedimentos sem precisar adaptar componentes das diversas plataformas de middleware existentes. Ou seja, o mecanismo de seleção dinâmica é muito mais expressivo do que o UDDI.

5.2 Gerador Automático de Stubs

Em [Chiang 07] é discutido um gerador automático de *Wrappers* CORBA, os quais escondem a complexidade do código do servidor para as aplicações cliente. Aplicado inicialmente para prover transparência de acesso a aplicações Mainframe, o gerador pode ser utilizado para integrar diferentes plataformas de middleware, porém nenhuma implementação especifica é discutida e não fica claro o nível de complexidade relacionado à geração desses *wrappers*, bem como desempenho dos mesmos. Essa proposta de utilização de wrappers segue a sistemática discutida em [IONA 01], onde CORBA Wrappers são construídos para permitir que clientes CORBA possam acessar componentes EJB sem precisarem se preocupar em lidar com objetos complexos que possam eventualmente ser retornados da execução de um método de negócio de um EJB. Essa solução gera uma quantidade enorme de arquivos, uma vez que o mapeamento CORBA-Java não é trivial, como visto na seção 2.3, além do mais é uma solução orientada a implementação, onde não é definida uma arquitetura de referência que permita uma eventual implementação por terceiros para outra plataforma de middleware.

Na estratégia apresentada ainda não é mencionado nada em relação a serviços de seleção dinâmica, de forma que o programador é obrigado a conhecer detalhes de especificação dos serviços que deseja invocar.

5.3 J-Integra Espresso

A solução J-Integra Espresso [J-Integra 07] propõe-se a realizar a integração entre as principais plataformas de middleware existentes: .Net, CORBA e JEE. Tal solução consiste na implementação de um ORB (*Object Request Broker*) na linguagem C# e segue a especificação definida pela OMG. Dessa forma o protocolo de comunicação é o IIOP, o que permite que aplicações escritas em .Net acessem aplicações CORBA e, por conseqüência, aplicações EJB, através do protocolo RMI-IIOP discutido na secão 2.3. A figura 41 mostra a arquitetura da solução J-Integra Espresso, mostrando que o elemento principal é o protocolo IIOP, que permite a comunicação com CORBA e JEE.

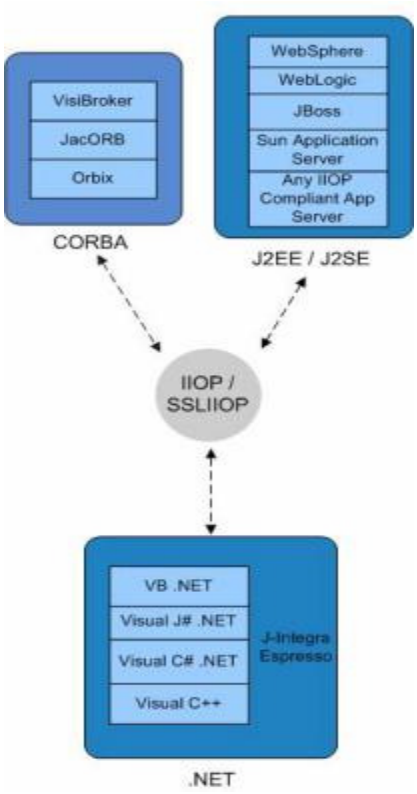


Figura 41 - J-Integra Espresso[em http://j-integra.intrinsyc.com/support/espresso/doc/]

J-Integra Espresso define um conjunto de regras para mapeamento IDL para C#, que seguem os mesmos procedimentos utilizando no mapeamento Java-IDL. Outro ponto a ser analisado é que diversas classes utilitárias Java que vem em sua distribuição, tais como *HashMap*, *ArrayList* são incorporadas à solução. Porém, a solução recai no procedimento tradicional no desenvolvimento de aplicações que precisam acessar servidores EJB a partir de clientes CORBA, é necessário definir uma IDL para o serviço EJB e utilizar um compilador específico que irá gerar *stubs* e *skeletons* para .Net para só então ser construído o cliente. Na nossa proposta como o objetivo é tornar os serviços disponíveis dentro da camada *multi-middleware*, não é necessário fazer esses passos descritos anteriormente. Além disso, é definida uma arquitetura de referência que não se limita a uma plataforma de middleware especifica, permitindo que o desenvolvedor construa os próprios elementos de *bridge*, seleção dinâmica e *binding*. No trabalho mencionado não existe nenhuma referência a serviços de seleção dinâmica que estejam disponibilizados em conjunto com a solução. A solução é proprietária, existindo apenas um demo para utilização dos serviços básicos que a compõem.

5.4 IIOP .Net

A solução IIOP .Net [Ullmann 03] permite aplicações .Net possam acessar aplicações CORBA e EJB. Tal solução consiste em adicionar suporte ao protocolo IIOP à plataforma .Net, aumentando a capacidade do .Net Remoting para comunicação entre aplicações que estão em domínios diferentes. IIOP .Net atua como um ORB mapeado permitindo que objetos .Net seja acessados por outros ORB´s. Da mesma forma como feito em [J-Integra 07], a solução define um conjunto de regras para mapeamento entre tipos de dados do .Net e CORBA. Segue a premissa que um compilador deve ser utilizado para gerar skeletons e stubs a partir de um arquivo IDL que deve seguir as especificações definidas pela solução, o que acaba recaindo nos mesmos problemas visto na seção 2.3 para realizar interoperabilidade entre aplicações JEE e CORBA. A solução apresentada nesse trabalho procura definir uma arquitetura de referência sem se preocupar com detalhes de implementação como foi feita no IIOP .Net, onde o mecanismo de binding é responsável por mapear os tipos de dados da plataforma de middleware subjacente para o tipo equivalente da camada multi-middleware.

No IIOP .Net não é discutido nenhuma maneira de realizar seleção dinâmica de componentes.

6 CONCLUSÕES

Neste trabalho apresentamos as conclusões obtidas no desenvolvimento do trabalho (seção 7.1) bem como algumas ações futuras (seção 7.2) que devem ser feitas para complementar o ambiente, tanto e termos de revisão da arquitetura de referência bem como na implementação base.

6.1 Considerações Finais

Esse trabalho apresentou um ambiente *multi-middleware* para desenvolvimento de aplicações distribuídas, que permite integrar diversas plataformas de middleware, sem perder as características intrínsecas de cada uma delas. O ambiente fornece uma forma simples de programadores reusarem componentes de diferentes plataformas sem precisar saber em qual plataforma o componente encontra-se, nem em qual linguagem é implementado. Isso é possível pois a camada multi-middleware tem o registro de todas as plataformas de middleware subjacentes e, ao receber uma invocação do cliente, repassa-a para o mecanismo de seleção dinâmica das plataformas selecionar os componentes que oferecem o serviço. Uma vez selecionados, a *bridge* de cada plataforma realiza os procedimentos para invocação do serviço e repassa ao *binding* que mapeia os parâmetros da invocação solicitada pelo cliente, invoca o método do componente da plataforma de *middleware* por ele tratada, e, ao receber o resultado, mapeia-o para um tipo especifico do ambiente multi-middleware antes de ser entregue ao cliente. O ambiente tem o diferencial de prover suporte a conversão orientada a objetos, resolvendo os problemas de incompatibilidade entre os dados das plataformas de *middleware*. O mecanismo de seleção dinâmica permite a localização de componentes combinada com a subseqüente invocação da execução dos métodos dos componentes localizados, sem a necessidade de conhecer em detalhes as interfaces dos serviços ou mesmo os seus nomes.

Prover ferramentas que permitam o rápido desenvolvimento de aplicações torna-se extremamente necessário para que o desenvolvedor não precise realizar diversas configurações manuais, memorizar detalhes de inicialização que não fazem parte da lógica de negócio, além de promover um aumento da

produtividade de modo geral. A IDE Eclipse é um dos ambientes integrados de desenvolvimento mais utilizados no mundo, principalmente por programadores Java, pela diversidade de recursos oferecidos e pela capacidade estender as funcionalidades existentes, de tal forma que requisitos demandados por algum programador, que não existam no ambiente possam ser construídos e integrados ao ambiente. A implementação de referência do ambiente multi-middleware consiste também em uma das contribuições do trabalho tanto do ponto de vista de validação da arquitetura proposta quanto da facilidade de desenvolvimento para programadores e também como referência para a implementação de outras ferramentas baseadas na arquitetura multi-middleware proposta nesse trabalho

Os *WebServices*, como mencionado na seção de trabalhos relacionados, têm sido a solução mais utilizada para alcançar interoperabilidade entre diversas plataformas de middleware, bem como com aplicações legadas. Nosso objetivo não foi produzir uma solução melhor ou pior que *WebServices,* mas apontar outra solução para se alcançar interoperabilidade. Apesar da solução que propomos integrar diversos elementos e diferentes tecnologias dentro de um ambiente, os desenvolvedores apenas precisam registrar seus componentes no serviço de seleção dinâmica da sua plataforma de middleware, através do mecanismo de *bridge*. As demais funções que permitem a interoperabilidade são realizadas de forma transparente pela plataforma multi-middleware. Dessa forma não é necessário que o desenvolvedor adapte os componentes existentes para o ambiente. A única demanda necessária é a instalação apropriada dos mecanismos que compõe o ambiente.

A implementação de referência foi desenvolvida utilizando-se o sistema operacional Windows XP, a implementação CORBA utilizada foi a MICO e compilador C++ utilizando foi VC++ na versão 7.0. Para implementação Java foi utilizado JSDK na versão mais recente, 6.0, e o servidor de aplicações foi o JBOSS na versão 4.2. A versão do Eclipse empregada foi a 3.2.

6.2 Trabalhos Futuros

O ambiente multi-middleware prevê em sua arquitetura a cooperação entre diversas plataformas de middleware, que interagem através da camada multi-middleware, além de um serviço de seleção dinâmica, que permite a seleção de componentes com base em variados critérios de busca.

Lidar apenas com diversas plataformas de middleware, do ponto de vista de realizar chamadas de métodos e realizar seleção dinâmica de componentes, não é suficiente para permitir total homogeneidade do ponto de vista de todos os serviços implementados por cada uma delas. Uma vez que cada plataforma de middleware define aspectos relacionados a controle de transações, segurança e tratamento de exceções, seria necessário que um ambiente multi-middleware endereçasse todos esses aspectos de forma transparente para o desenvolvedor que utiliza a camada multi-middleware, podendo definir, por exemplo, um mecanismos homogêneo para tratamento de exceções que fosse gerenciado de forma independente da plataforma de middleware utilizada.

Os principais pontos a serem estudados com intuito de tornar o ambiente mais robusto e, por conseqüência, mais apto a integrar essas plataformas são os seguintes:

- Prever na arquitetura uma forma de adicionar suporte a ontologias, associada ao serviço de seleção dinâmica, permitindo a construção de critérios bem mais ricos em termos de consulta a componentes;
- Analisar aspectos de segurança para aplicação desenvolvida dentro do ambiente multi-middleware, uma vez que os detalhes de segurança são de responsabilidade de cada uma das plataformas de middleware, inviabilizando uma sistemática única para garantir a segurança da aplicação;
- Estudar e conceber um mecanismo de tratamento de exceções e transações que sejam independentes de plataforma de middleware;
- Adicionar a implementação de referência a plataforma .Net, utilizando a ferramenta *LuaInterface* [Mascarenhas 04];

- Analisar quais ferramentas são necessárias para melhorar o suporte ao desenvolvimento de aplicações multi-middleware utilizando ambiente em especial o plugin do Eclipse

7 REFERÊNCIAS

ALMEIDA, A. D., Batista, T. and Cacho, N. **LuaSpace EPLUS: Um ambiente para desenvolvimento de aplicações CORBA no Eclipse**. In: Anais do XIV Simpósio Brasileiro de Redes de Computadores (SBRC'2006), SBC, pages 1315-1330, ISBN85-7669-002-0, Curitiba, Paraná, Maio 2006.

ALMEIDA, A. D., Cacho, N. and Batista, T.(2004) **LuaSpace Plus: Um Ambiente Visual para Desenvolvimento de Aplicações CORBA** In: Anais do XVIII Simpósio Brasileiro de Engenharia de Software (SBES'2004), SBC, pages 163-177, ISBN85-7669-002-0,Brasília, DF, Outubro 2004.

ALUR, D., Crupi, J. and Malks D.(2004) **Core J2EE Patterns: Best Practices and Strategies**. Sun Microsystems.

BATISTA, T. and Carvalho, M. **Component-Based Applications: A Dynamic Reconfiguration Approach with Fault Tolerance Support**. In Software Composition Workshop (SC) - affiliated to European Joint Conferences on Theory and Practice of Software (ETAPS), Grenoble - FR, April 2002. Published in Electronic Notes in Theorical Computer Science, Vol. 65, Number 4, 2002. http://www.elsevier.nl/locate/entcs/volume65.html

BATISTA, T.V., Cerqueira, R., Rodriguez, N. **Enabling Reflection and Reconfiguration in CORBA**. In *Workshop Proceedings of the International Middleware Conference*, Rio de Janeiro - Brazil, pp. 125-129, 2003.

BATISTA, T.V., Rodriguez, N. **Dynamic Reconfiguration of Component-based Applications**. In *Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE*), Limerick, Ireland, 2000.

BEA Systems(2007), Bea Weblogic Server 10: The Rock Solid Foudantion for SOA, Disponivel em :

http://www.bea.com/content/news_events/white_papers/BEA_WL_Server10_wp.pdf

BEACH, B., **Connecting software components with declarative code glue**, Proceedings of th 14th International Conference on Software Engineering May(1992) 120-137.

BERNSTEIN, P. **Middleware**. Communications of the ACM, 39(2), February 1996.

BOOTH, D. et all.(2004) **Web Services Architecture Specification**. Disponivel em http://www.w3.org/TR/ws-arch/

BROSE, G. **JacORB: Implementation and Design of a Java ORB**. In Proceedings of Dais 97, IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems, Cottbus, Germany, Sempteber 1997. Chapman & Hall.

CACHO, N., Batista, T. and Elias, G. (2004) **Um Serviço CORBA para Descoberta de Componentes**, In: Anais do XVIII Simpósio Brasileiro de Engenharia de Software (SBES'2004), SBC, pages 273-288, ISBN85-7669-0020, Brasilia, DF,Outubro 2004.

CACHO,N.,Batista T. and Matoso A. (2005) **Reuso e Interoperabilidade de Componentes através de Ontologias** Artigo Submetido ao SBES 2005.

CASSINO, C. and Ierusalimschy, R. **LuaJava – Uma Ferramenta de Scripting para Java** In Simposio Brasileiro de Linguagens de Programação(SBPL´99)1999

CERQUEIRA, R., Cassino, C. and Ierusalimschy, R. (1999) **Dynamic Component Gluing Across Different Componentware Systems**. In International Symposium on Distributed Objects and Applications (DOA´99), 362-371, Edinburgh, Scotland, September 1999. OMG, IEEE Press.

CHIANG, C-C. ACM **Automatic software wrapping** In Proceedings of the 45th annual southeast regional conference, p. 59-64, Winston-Salem, North Carolina USA March 2007.

CHIANG, C-C. **The use of adapters to support interoperability of components for reusability**. Information and Software Technology, Volume 45, Number 23, March 2003, pp. 149-158.

COELHO, O.(2007) **Escolhendo entre Web Services, Enterprise Services e Remoting**. Disponível em http://msdn.com/brasil/msdn/Tecnologias/arquitetura/Escolhendo.mspx

FERNANDES, F. and Batista, T. **Dynamic Aspect-Oriented Programming: An Interpreted Approach** In: *Proceedings of the 2004 Dynamic Aspects Workshop (DAW04)*, Lancaster - England. pp. 44 – 50, 2004.

FERNANDES, F., A. **Combinando Aspectos e Componentes: Uma Abordagem interpretada**. Tese de Mestrado, Departamento de Informática e Matemática Aplicada, Natal, RN, 2004.

FISHER, M., Lai, R. et all(2006). **Java EE and .Net Interoperability: Integration strategies, patterns and best practices**. Prentice Hall, April 21, 2006.

FLEURY, M. and Reverbel F.(2003) **The JBoss Extensible Server**, Lecture Notes in Computer Science, Springer Berlin / Heildelberg, p. 344-373.

FRAKES, B. and LEA, D. **Design for Reuse and Object Oriented Reuse Methods**. Disponivel em http://www.umcs.maine.edu/~ftp/wisr/SEN-pap/node1.html

FRIEDMAN, J.(2001) **An Introduction to Reflection Oriented-Programming** Disponivel em: http://www.cs.indiana.edu/~jsobel/rop.html

GAMMA, E., Helm R., Johnson R. e Vlissides(2005) **Padrões de Projeto: Soluções reutilizáveis de software orientado a objetos**. Editora Bookman pp.179-186.

GRANHAM, S. et all (2006**). Web Services Resource Framework Specification**. Disponível em http://docs.oasis-open.org/wsrf/wsrf-ws_resource1.2-spec-os.pdf

IERUSALIMSKY, R., Figueiredo, L. H., and Celes, W. (1996) **Lua – an extensible extension language**. *Software: Practice and Experience*, 26(6):635-652.

IONA Technologies(2001) **CORBA-EJB Interoperability White Paper**

ISSARNY, V. and Bellissard, L. and Riveill, M. and Zarras, A. (1999) **Component-Based Programming of Distributed Applications**. Advances in Distributed Systems pp.327-353
Jain,A.(2007) **What´s new in WebSphere Application Server Community Edition**.

J-INTEGRA(2007). J-Integra Espresso Documentation. Disponivel em: http://j-integra.intrinsyc.com/support/espresso/doc/

JEFFERY, D., Dowd, T. and Somogyi Z. MCORBA: **A CORBA Binding for Mercury**. In *Proceedings of the First International Workshop on Practical Aspects of Declarative Languages* , San Antonio, Texas, January 1999, Lecture Notes in Computer Science 1551, Springer Verlag, Pages 211-227

KATZ, S., DABROWSKI, C. and LAW, M. **Glossary of Software Reuse Terms.** Prepared for The Department of Defense/Ballistic Missile Defense Organization, by U.S. Department of Commerce/National Institute of Standards and Technology, October 1993.

MACHALE, C.(2007) **CORBA Explained Simply**, Disponível em www.ciaramchale.com

MARTINS, S., Cacho, N. and Batista, T. (2004) **Uma Biblioteca para Segurança de Aplicações CORBA**. In: Anais do 22o. Simpósio Brasileiro de Redes de Computadores (SBRC'2004), Gramado, RS, Maio 2004, ISBN 85-88442-79-5, pp. 511-524.

MEIJER, E. and Gough, J. (2002). **Technical Overview of the Common Language Runtime.Technical report**, Microsoft Research. Disponível em http://research.microsoft.com/~emeijer/Papers/CLR.pdf.

MICROSOFT (2002). **ECMA C# and Common Language Infrastructure Standards**. Disponível em http://msdn.microsoft.com/net/ecma/

MICROSOFT(2005). **Microsoft Visual Studio 2005 Overview** Disponível em http://msdn2.microsoft.com/pt-br/vstudio/aa700919.aspx

OMEX(Object Middleware Experts) Java to Corba Bridge 2.0(2001), Disponível em: http://www.omex.ch/products-jcorbabridge.htm

OMG (2004) **Common Request Broker Architecture: Technical Report Revision 3.0.3**.

OMG(Object Management Group) **Java to IDL Mapping**(2003).

OTI(2003). **Eclipse Platform Overview**. Disponível em: http://www.eclipse.org/articles/index.html.

PARRIGTON, D. G.(1995) **A Stub Generation System for C++**, TechnicalReport: Newcastle-CS#TR95-510, University of Bologna.

PUDER, A. and Römer, K.(2000) **MICO: An Open Source CORBA Implementation** 3º Edition, Morgan Kauffman.

RINE, D.,Nada, N., Jaber, K. **Using adapters to reduce interaction complexity in reusable component-based software development**, Proceedings of the fifth Syposium on Software Reusability May(1999) 37-43.

SIERRA, K. and Bates, B.(2003). **Head First EJB**. O´Reilly, May 20 2003.

SOMOGY, Z., Henderson, F. and Conway, T. **Mercury: An Efficient Purely Declarative Logic Programming Language** Presented at *ACSC'95*, Adelaide, South Australia.

SUN(1997) **JavaBeans Specification**, Disponível em:

SUN(2002) **Java RMI Over IIOP Specification**, Disponível em http://java.sun.com/j2se/1.4.2/docs/guide/rmi-iiop/

SUN(2004) **Annotations**, Disponivel em: http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html

SUN(2003) **Enterprise Java Beans Specification**

SZYPERSKI, C. **Beyond Object-Oriented Programming**, 2nd ed. Addison-Wesley, 2002.

WALLIN, D. and Norberg A(2003). Luabind: A library to integrate C++ and Lua. Disponivel em: http://www.rasterbar.com/products/luabind/docs.html

ULLMANN, D. (2003) **IIOP .Net,** Disponível em: http://iiop-net.sourceforge.net/index.html