3
Componentes em Lua

O OiL (*ORB in Lua*) é uma implementação na linguagem Lua de um *middleware* compatível com CORBA. O objetivo do OiL é ser portável, flexível e simples. O uso de Lua facilita essas três características: a linguagem, juntamente com a biblioteca LuaSocket (52), nos fornece uma camada de portabilidade que foi usada para a implementação do *middleware*. O OiL é construído sobre essa camada, fazendo com que ele esteja disponível em diferentes plataformas de forma transparente. A linguagem Lua, por ser ela mesma mais flexível em relação a uma linguagem compilada e estaticamente tipada, facilita a construção de um sistema mais dinâmico. A simplicidade no uso do OiL vem dos próprios mecanismos fornecidos por Lua, como por exemplo a coleta de lixo e a criação e manipulação mais fácil de estruturas de dados.

A implementação atual do OiL nos fornece uma ferramenta para investigar e prototipar técnicas de implementação de *middlewares* e abstrações de programação distribuída. O OiL já foi utilizado para a pesquisa em adaptação dinâmica no nível de aplicação (53), em mecanismos para traversia de *firewalls* (54), computação em grade (55) e políticas de escalonamento (56).

Apesar de Lua não ser uma linguagem orientada a objetos, ela pode ser estendida de forma a prover algumas funcionalidades de uma linguagem orientada a objetos, tais como herança e variáveis de classe. O modelo utilizado na implementação do OiL é o LOOP, um modelo dinâmico de programação orientada a objetos desenvolvido no Grupo de Sistemas Distribuídos da PUC-Rio e utilizado em outros projetos de *middlewares* dinamicamente adaptáveis (57). O LOOP implementa o comportamento compartilhado de objetos de uma dada classe através de uma *metatable* (05) e mantém dados de um objeto específico (estado do objeto) em uma tabela Lua que representa o objeto. Esse modelo já oferece suporte a adaptações com a possibilidade da mudança desse comportamento compartilhado, tornando-o disponível para todos os objetos da mesma classe no momento da mudança, através da troca da implementação da *metatable*.

Entretanto, para este trabalho foi interessante adotar uma extensão do

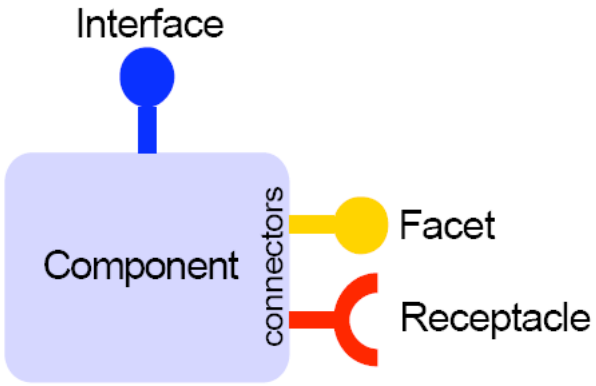

Figura 3.1: Estrutura de um componente.

LOOP com suporte a componentes em Lua, para poder definir de forma explícita a relação entre partes de um *middleware*. O modelo foi baseado em outros modelos de componentes, como o CORBA Component Model (CCM), o OpenCOM e o Fractal. Neste modelo, usa-se a nomenclatura do CCM, ou seja, um componente é formado de portas chamadas facetas e receptáculos. Além disso, ele exporta uma interface principal, que é a maneira através da qual o componente obtém acesso diretamente à sua implementação. Uma representação gráfica de um componente é apresentada na figura 3.1.

Este modelo provê interfaces simples para a criação de componentes, especificando as facetas e receptáculos para cada um deles. Uma breve descrição do modelo será apresentada a seguir, assim como as formas de construção de componentes e as opções de ligação entre eles.

3.1 Tipos de componente

O modelo permite a definição de tipos, que é a descrição de quais são as portas exportadas por um dado componente.

O modelo também permite herança de tipos, usando a mesma sintaxe do LOOP. Um componente filho herda as definições de portas do componente pai. Um exemplo de herança está na listagem 3.1, nas linhas 12 a 14: o tipo `InvokeProtocolType` herda as facetas do tipo ProtocolType.

3.2 Portas

As facetas são as interfaces que um componente provê, ou seja, conjuntos de funções exportadas para serem usadas por outros componentes. Um componente pode ter uma ou mais facetas, que são usadas pelos receptáculos de

outros componentes, depois de feita a ligação entre eles. A especificação de um componente com uma faceta é mostrada nas linhas 4 a 6 da listagem 3.1.

Já os receptáculos são declarações de quais interfaces o componente em questão necessita para o seu funcionamento. Através da declaração do que é necessário para um dado componente, um configurador externo consegue descobrir se todos os pré-requisitos para um sistema estão resolvidos e, com isso, criar um mecanismo para garantir que o sistema funcionará corretamente. No modelo que adotamos podemos dividir os receptáculos em dois tipos: simples e múltiplos.

Receptáculos simples

Neste tipo de receptáculo, há apenas uma faceta ligada a um determinado receptáculo, e este é o tipo mais comum. Na listagem 3.1, linhas 8 e 9, mostramos a especificação de dois receptáculos simples no tipo de componente `ProtocolType`.

Receptáculos múltiplos

Os receptáculos múltiplos servem para ligar mais de um componente em um dado receptáculo. Este modelo de componentes oferece receptáculos de lista, onde os componentes ligados não possuem um identificador explícito. Se for necessário identificar cada componente ligado a um receptáculo, é interessante usar um receptáculo de tabela associativa. Há também o receptáculo de conjunto, que se comporta da mesma maneira que um receptáculo de lista, exceto por não permitir repetições de componentes associados a ele. Na listagem 3.1, linha 18, criamos um receptáculo de lista no tipo de componente `ServerBrokerType`.

```
1
2  ...
3  -- creating the component types
4  CodecType = component.Type{
5    codec = port.Facet,
6  }
7  ProtocolType = component.Type{
8    channels = port.Receptacle,
9    codec = port.Receptacle,
10 }
11 -- the type InvokeProtocol inherits the facets and receptacles of Protocol
12 InvokeProtocolType = component.Type({
13   interface = port.Receptacle,
14 }, ProtocolType)
15
16 ServerBrokerType = component.Type{
17 ...
```

```
18    ports = port.ListReceptacle,
19  ...
20  }
```

Listagem 3.1: Exemplo da criação de um tipo

3.3 Instanciação de componentes

Para instanciar um componente, cria-se uma fábrica para um dado tipo. A criação da fábrica é feita através de uma chamada para o tipo, passando como parâmetro uma tabela com um objeto, chamado construtor principal, e com implementações específicas para facetas, se for o caso. Os construtores podem ser quaisquer objetos executáveis que devolvam a implementação de um novo componente sempre que chamados.

Um exemplo de criação de uma instância de um componente é demonstrada na listagem 3.2: na linha 7, especifica-se que a implementação de um tipo `Codec` é o objeto `CodecImpl`, criando a fábrica `CodecFactory`. Nas linhas 11 a 14, cria-se uma fábrica `InvokeProtocolFactory`, a partir de um objeto `ProtocolImpl` e de um objeto que implementa a faceta *invoker*, o `InvokeProtocolImpl`. As fábricas são então utilizadas, nas linhas 16 e 17 para criar duas instâncias de componentes.

```
1  -- getting the constructors for each component type
2  ProtocolImpl = require "oil.corba.Protocol"
3  CodecImpl    = require "oil.corba.Codec"
4  InvokeProtocolImpl = require "oil.corba.Invoke"
5
6  -- the implementation for the Codec is the main object
7  CodecFactory = CodecType{ CodecImpl }
8
9  -- the Invoke Protocol factory uses a different object
10 -- to implement the "invoker" facet
11 InvokeProtocolFactory = ProtocolType{
12   ProtocolImpl,
13   invoker = InvokeProtocolImpl,
14 }
15 -- creating two components, one of each type
16 codec1 = CodecFactory()
17 prot1 = ProtocolFactory()
18
19 -- binding components
20 prot1.codec = codec1.codec
```

Listagem 3.2: Exemplo da criação de componentes

3.4 Conexões facetas-receptáculos

Para a criação de um sistema baseado em componentes, é necessário instanciá-los e conectá-los de acordo com a necessidade e com as regras designadas pelas facetas e receptáculos, através de um processo chamado ligação (*binding*). Em alguns modelos, há a definição de tipos para facetas e receptáculos, ou mesmo regras para a ligação entre eles, para que durante a configuração não se possa fazê-la de maneira incorreta. No entanto, o modelo utilizado aqui não oferece esse tipo de controle, deixando a cargo do desenvolvedor a responsabilidade de conectá-los de maneira coerente. Na linha 20 da listagem 3.2 mostramos a conexão da faceta *codec* de um `Codec` com um receptáculo de mesmo nome de um `Protocol`.

3.5 Tipos de modelos de componentes

Um dos objetivos pelos quais adotamos esse modelo de componentes do LOOP na implementação do *middleware* foi a possibilidade de inserir mecanismos para flexibilizar as conexões entre módulos diferentes. O mecanismo que o LOOP nos oferece inclui portas com capacidade de interceptação, mediante o registro de funções que são associadas a uma operação em uma faceta ou receptáculo. Essa função registrada pode ser chamada antes ou depois da operação ser executada. A interceptação pode ser específica a uma porta em todos os componentes de um dado tipo, ou mesmo em uma instância específica de um componente, e é associada a eventos específicos em uma porta, como por exemplo uma chamada de método em uma faceta.

O LOOP oferece pacotes que implementam diferentes "sabores" de modelos de componentes, com flexibilidade crescente. No primeiro pacote, *Base*, usa-se o próprio objeto de implementação do componente para armazenar a implementação das facetas e as conexões dos receptáculos, tornando o acesso a essas informações mais eficiente. No entanto, essa escolha não permite usar o mecanismo de portas com capacidade de interceptação.

O pacote *Wrapped* também guarda as referências para as portas no próprio objeto de implementação do componente, mas permite o uso de portas com capacidade de interceptação. No pacote *Contained*, o estado do componente e as referências das portas ficam em um objeto de contexto, criando mais uma camada de indireção, para o caso de objetos que não podem guardar as referências para as portas. O pacote *Dynamic* adiciona o suporte a criação e remoção de portas dinamicamente, seja de todos os componentes de um mesmo tipo ou de uma dada instância de um componente.