4

LuaCCM: Um Modelo de Componentes Dinâmico

Uma das principais preocupações no desenvolvimento de plataformas para o construção de aplicações é a simplicidade de uso. Plataformas de *middleware* como CORBA se mostram muito complexas em diversas situações, em particular no desenvolvimento de aplicações adaptáveis. Isso pode ser justificado pela introdução de mecanismos de flexibilização, que geralmente resultam em maior complexidade de implementação da aplicação, especialmente em linguagens como Java ou C++. Nesse sentido, o uso de uma linguagem dinâmica e flexível em plataformas de *middleware* auxilia a atenuar esse tipo de complexidade [29]. Um exemplo desse uso é o LuaOrb [32], que é uma ferramenta para o desenvolvimento de aplicações CORBA que utiliza a flexibilidade e simplicidade da linguagem Lua para facilitar a utilização da arquitetura CORBA, em especial a criação de objetos dinâmicos através da interface dinâmica de CORBA (ver apêndice A).

Com a introdução do novo modelo de componentes (CCM) na versão 3.0 da especificação CORBA, fez-se necessário avaliar e adaptar os novos recursos e abstrações definidos pelo CCM ao LuaOrb. Isso deve ser feito juntamente com uma análise de como os recursos da linguagem Lua podem simplificar a utilização desses novos conceitos. O modelo CCM foi elaborado para ser aplicado sobre qualquer implementação da arquitetura CORBA, incluindo portanto o LuaOrb. A implementação de um modelo de componentes sobre o LuaOrb permite incorporar as mesmas características presentes no projeto deste, como flexibilidade e simplicidade de uso. Por essas razões, neste trabalho foi desenvolvido um sistema para desenvolvimento de componentes dinâmicos denominado LuaCCM. Neste capítulo, é feita uma discussão detalhada da implementação do LuaCCM, assim como de suas funcionalidades e limitações. O LuaCCM é o alicerce sobre o qual são construídos os recursos fornecidos pelas ferramentas propostas nesse trabalho.

O LuaCCM utiliza a flexibilidade e o dinamismo oferecidos pela linguagem Lua e o LuaOrb para permitir a construção de componentes dinâmicos de forma simples e eficiente. Entretanto, um dos principais desafios em utilizar o modelo CCM na elaboração de um modelo de componentes dinâmico é a adaptação de alguns de seus conceitos para um paradigma mais flexível. Isso se justifica pelo fato do modelo de componentes de CORBA ter sido concebido sem preocupações relativas à possibilidade de componentes poderem ser alterados em tempo de execução. O LuaCCM define recursos adicionais especificamente relacionados a construção e manipulação de componentes dinâmicos.

Em particular, um novo modelo de contêiner projetado para componentes dinâmicos é apresentado.

4.1

Base de Implementação

O LuaCCM é uma implementação do modelo CCM sobre um conjunto de ferramentas, em especial a linguagem Lua. Essa linguagem fornece recursos de reflexão computacional através dos quais é possível realizar alterações nas aplicações em tempo de execução. Além disso, os mecanismos de extensão de Lua permitem definir abstrações que simplificam o desenvolvimento de componentes dinâmicos e do próprio LuaCCM. O LuaCCM também é construído sobre o LuaOrb, que é uma extensão da linguagem Lua que define um mapeamento para arquitetura CORBA, permitindo acessar e implementar objetos CORBA em Lua. Nesta seção, será feita uma pequena apresentação das ferramentas que formam a base de implementação do LuaCCM, inclusive de um modelo dinâmico de programação orientada a objetos definido em Lua, que fornece recursos que facilitam a implementação dos contêiners dinamicamente adaptáveis do LuaCCM.

4.1.1

A Linguagem Lua

Desenvolvida no Departamento de Informática da PUC-Rio, a linguagem Lua [33] tem sido amplamente utilizada na construção de diversas aplicações em domínios diferentes. Essa abrangência se deve primordialmente a flexibilidade e simplicidade da linguagem, aliadas ao seu poder computacional e eficiência. Lua é uma linguagem de extensão, que oferece recursos de programação procedural e descrição de dados, e também é dita interpretada, no sentido de que as instruções e construções da linguagem são traduzidas para um código intermediário, que é então executado. Além disso, é possível criar, através da própria linguagem, porções executáveis a partir de um trecho de código Lua, ou seja, é possível executar um código criado em tempo de execução.

Outra característica importante da linguagem é ser dinamicamente tipada e extensível. Variáveis em Lua podem armazenar quaisquer valores, inclusive funções, que em Lua são valores de primeira classe. Aliado a isso, a linguagem oferece diversos mecanismos reflexivos, que permitem estender a semântica da linguagem, tornando a linguagem extremamente flexível e dinâmica. Apesar dessas características, a linguagem Lua ainda se mantém simples e de fácil utilização.

Os recursos de descrição de dados da linguagem estão relacionados à única estrutura de dados de Lua, que consiste de uma tabela de dispersão que permite armazenar valores indexados por qualquer valor da linguagem. Através

das tabelas de Lua, é possível criar vetores, árvores e demais estruturas. Em particular, em Lua é possível representar objetos através de tabelas contendo valores representando atributos e funções representando métodos, uma vez que funções Lua são valores de primeira classe. Apesar de não ser uma linguagem orientada a objetos, a linguagem Lua fornece recursos através de açúcares sintáticos que permitem utilizar um estilo de programação orientado a objetos. Além disso, através dos mecanismos de extensão da linguagem, é possível implementar mecanismos de compartilhamento de comportamento baseados em hierarquias de classes ou protótipos, além de outros conceitos do paradigma de orientação a objetos.

4.1.2

LuaOrb: O Mapeamento de Lua para Arquitetura CORBA

A arquitetura CORBA define mecanismos para integração de plataformas e linguagens de programação através de regras de mapeamento de dados e interfaces entre diferentes linguagens e um conjunto de protocolos de comunicação. Geralmente, esse mapeamento é feito por código gerado automaticamente por meio de ferramentas fornecidas pelas implementações da arquitetura. Entretanto, com os recursos reflexivos da linguagem Lua, é possível realizar esse mapeamento sem a necessidade de geração de código, fazendo apenas uma tradução dinâmica entre os tipos da linguagem e os tipos IDL. Dessa proposta, surge o LuaOrb[32], que define um mapeamento entre a linguagem Lua e a arquitetura CORBA, permitindo manipular objetos CORBA a partir de aplicações Lua, assim como criar novos objetos servidores escritos em Lua.

Através do LuaOrb, é possível criar objetos Lua que representam um objeto CORBA. Os mecanismos reflexivos de Lua permitem criar um objeto de forma que, quando seus atributos são lidos ou alterados e suas operações são chamadas, todas essas ações sejam repassadas ao objeto CORBA associado utilizando a interface de invocação dinâmica de CORBA (ver apêndice A). Todos os valores trocados são devidamente convertidos entre os tipos IDL e tipos Lua, de acordo com o mapeamento definido pelo LuaOrb[34]. De forma similar, o LuaOrb permite criar objetos CORBA, bastando informar um objeto Lua que forneça a implementação das operações e valores dos atributos de uma interface CORBA. Nesse caso, o LuaOrb cria um objeto dinâmico através da interface de esqueleto dinâmico de CORBA (ver apêndice A), de maneira que sempre que ele recebe uma requisição, a operação de mesmo nome é executada no objeto Lua fornecido como implementação do objeto CORBA.

4.1.3

LOOP: Um Modelo Dinâmico de Programação Orientada a Objetos

Toda a arquitetura CORBA, e por conseguinte o modelo CCM, são construídos utilizando o paradigma de orientação a objetos. Apesar disso, existem mapeamentos desses modelos para linguagens não orientadas a objetos (*e.g.*,

C, COBOL, entre outras), mas que resulta em um código mais complexo, além de algumas limitações. Com o intuito de simplificar a adaptação dos conceitos e modelos CCM aos conceitos da linguagem Lua, neste trabalho é definido um modelo de programação orientada a objetos, chamado LOOP (Lua *Object-Oriented Programming*), que é implementado através dos mecanismos de extensão de Lua e fornece mecanismos reflexivos que auxiliam o desenvolvimento de aplicações orientadas a objetos dinamicamente adaptáveis.

Toda implementação do modelo de componentes dinâmico do LuaCCM é composta por classes de objetos definidas utilizando o modelo LOOP. O principal propósito desse modelo é fornecer recursos similares aos encontrados em linguagens orientadas a objetos (*e.g.*, Java, C++), porém com a mesma flexibilidade e dinamismo da linguagem Lua. O modelo LOOP é um modelo dinâmico, no sentido de que permite a redefinição dinâmica de todas as características de classes e objetos, *e.g.*, membros de classe e objetos, relacionamentos de herança e delegação, classe de um objeto, entre outras.

No modelo LOOP, classes podem possuir membros em três níveis de escopo, como descritos abaixo.

privado: acessível apenas através do parâmetro self de chamadas de métodos declarados na classe.

protegido: acessível através do parâmetro self de chamadas de quaisquer métodos de um objeto, independentemente da classe onde o método é definido ou de sua posição na hierarquia de classes.

público: acessível através do parâmetro self de chamadas de métodos do objeto ou através de qualquer referência do objeto. Uma referência para um objeto LOOP é uma tabela através da qual é possível obter e definir membros públicos do objeto e não corresponde necessariamente ao parâmetro self, mas pode ser obtida através da operação self.get_reference().

Classes de objetos podem definir membros (*i.e.*, atributos e métodos) em quaisquer dos níveis de escopo descritos anteriormente, que serão automaticamente incorporados em todas suas instâncias (inclusive instâncias de suas subclasses). Alternativamente, o modelo LOOP também permite a *personalização de objetos*, ou seja, a definição de novos membros ou redefinição dos membros herdados de sua classe em apenas uma instância de objeto. Métodos adicionados numa instância de objeto têm acesso ao estado protegido do objeto, assim como o acesso a um estado privado da instância que não é acessível pelos demais métodos definidos na classe. Esse recurso de (re)definição de membros de uma classe de objetos ou de uma única instância permite realizar adaptações em níveis de abrangência diferentes, como será visto na seção 4.4.2. A figura 4.1 ilustra a declaração completa de uma classe LOOP que implementa uma pilha.

Além do mecanismo de compartilhamento de comportamento através de classes de objetos, o modelo LOOP também fornece o recurso de delegação. Nesse caso, quando um objeto delega seu comportamento a outro objeto, esse

```
require("LOOP")

oo.class.loopdemo.Stack {
  constructor = function(self)
    self.set_private("data", {})
  end,

  private = {
    top = 0,
  },

  public = {
    push = function(self, element)
      self.top = self.top + 1
      self.data[self.top] = element
    end,
    pop = function(self)
      local element = self.data[self.top]
      self.data[self.top] = nil
      self.top = self.top - 1
      return element
    end,
    getsize = function(self)
      return self.top
    end,
  },
}

local stack = oo.class.loopdemo.Stack:new()
stack:push("Um")
stack:push("Dois")
stack:push("Tres")

while stack:getsize() > 0 do
  print(stack:pop())
end
```

Figura 4.1: Exemplo de classe LOOP implementando uma pilha.

é chamado de objeto *delegador*. Quando um objeto delegador recebe uma mensagem, ou seja, o valor de algum atributo é lido ou alterado, ou algum método é chamado, e o delegador não é capaz de tratar a mensagem, por não possuir o atributo ou método, então a mensagem é repassada a outro objeto, chamado de *delegado*, ou seja, o método é executado ou o atributo é lido ou alterado no objeto delegado. No modelo de delegação do LOOP, a execução de um método num objeto delegador é feita com o estado do objeto delegado. Ou seja, o método não é "herdado" do objeto delegado, na realidade a mensagem é repassada ao objeto delegado que a processa e retorna o resultado diretamente ao cliente.

4.2
Contêiners Lua

A implementação de componentes CCM é baseada no modelo de programação do contêiner, que define que todas interações entre o componente e seus clientes sejam feitas através de interfaces oferecidas pelo contêiner. O contêiner é um elemento implementado automaticamente pelo ORB, que fornece um ambiente de execução para os componentes intermediando suas interações e oferecendo serviços. Contudo, parte das responsabilidades do contêiner são dependentes da definição do componente (*e.g.*, portas, interfaces). Por isso, o modelo CCM define que parte da implementação do contêiner seja gerada a partir da definição do componente em IDL 3.0 e seja embutida na implementação do componente. Entretanto, na construção de componentes dinâmicos, a implementação do contêiner não pode ser gerada a partir de um compilador de IDL, pois a definição do componente não é estática. Uma alternativa mais adequada seria adaptar dinamicamente o contêiner para corresponder a uma determinada definição de componente fornecida em tempo de execução.

Para tanto, é necessário elaborar uma implementação dinâmica de um contêiner CCM cujas interfaces externas sejam publicadas através de objetos CORBA dinâmicos, ou seja, através do mecanismo de DSI de CORBA (ver apêndice A). Dessa forma, o contêiner expõe interfaces capazes de serem alteradas dinamicamente, por exemplo para fornecer uma nova faceta através de uma operação adicional (*e.g.*, provide_novafaceta). Além disso, facetas e receptores de eventos implementados como objetos dinâmicos também podem ter suas interfaces alteradas (*e.g.*, adicionar operações ou mudar o tipo de evento consumido). A flexibilidade e o dinamismo da linguagem Lua e o LuaOrb, aliados aos recursos oferecidos pelo modelo dinâmico do LOOP, fornecem os subsídios necessários para implementação de um contêiner adaptável, como veremos em mais detalhes posteriormente.

O contêiner Lua é o componente do servidor LuaCCM onde implementações de componentes escritas em Lua são instaladas. As responsabilidades do contêiner Lua são listadas a seguir:

Obter a implementação do componente: o contêiner Lua é responsável por obter a implementação do componente a partir do *home* correspondente, na forma determinada pelo modelo CCM.

Publicar interfaces: o contêiner Lua é responsável por criar objetos CORBA que implementem as interfaces equivalentes do componente e repassar as chamadas a essas interfaces à implementação do componente de acordo com algum dos modelos de uso de CORBA definidos pelo modelo CCM (ver seção 2.2.5).

Gerenciar conexões: o contêiner Lua é responsável por manter a lista das conexões estabelecidas pelo componente com os demais, assim como implementar as operações para conectar e desconectar componentes.

Ativar o componente: o contêiner Lua é responsável por ativar os diversos segmentos que compõe a implementação do componente seguindo uma das políticas de gerência do ciclo de vida definidas no modelo CCM (ver seção 2.2.5).

Neste trabalho, a implementação do contêiner Lua não oferece nenhum dos recursos do contêiner CCM relacionados a serviços de objetos, tais como eventos, transações, persistência ou segurança. Além disso, o contêiner não oferece suporte a políticas de transações ou persistência gerenciadas pelo contêiner. Por isso, o contêiner Lua não provê suporte a categorias de componentes com estado persistente, como é o caso de componentes de processo e entidade. Entretanto, o contêiner Lua dá suporte a emissão e recepção de eventos através das portas de eventos CCM, apesar de não utilizar o serviço de eventos de CORBA para isso.

4.2.1

Estrutura dos Contêiners Lua

O contêiner é responsável por publicar as interfaces dos componentes instanciados nele. Cada componente LuaCCM é publicado através de um conjunto de objetos dinâmicos, que são criados sob demanda através do LuaOrb pelo contêiner Lua. Em especial, um objeto dinâmico é criado para representar a interface principal do componente, ou seja, um objeto que implementa a interface equivalente do componente. Através dele é possível obter referências para as facetas do componente, assim como estabelecer conexões com outros componentes. Os demais objetos dinâmicos que compõe o componente representam suas facetas e consumidores de eventos. Além dos objetos dinâmicos que publicam as interfaces externas do contêiner, um objeto Lua, que implementa as interfaces internas, é criado para representar o contexto do componente, ou seja, permitir que o componente tenha acesso às conexões em seus receptáculos e fontes de eventos. A figura 4.2 ilustra a estrutura dos contêiners Lua.

A implementação de todas as operações da interface equivalente do componente são automaticamente geradas pelo contêiner em tempo de execução, a exceção das operações que fornecem os serviços do componente, ou seja, operações das interfaces oferecidas e de suas facetas. Quando um novo componente é instalado em um contêiner Lua, este cria um subcomponente denominado *Gerenciador de Definição*, que é associado àquela nova definição de componente. O Gerenciador de Definição é responsável por obter a definição do componente a partir de um repositório de interfaces de componentes e, a partir disso, gerar a implementação das operações da interface equivalente do componente, que outrora seriam geradas pelo compilador de IDL.

O Gerenciador de Definição utiliza a implementação gerada da interface equivalente do componente para definir uma classe LOOP. Essa classe define um envoltório para a implementação do componente. O envoltório é um objeto

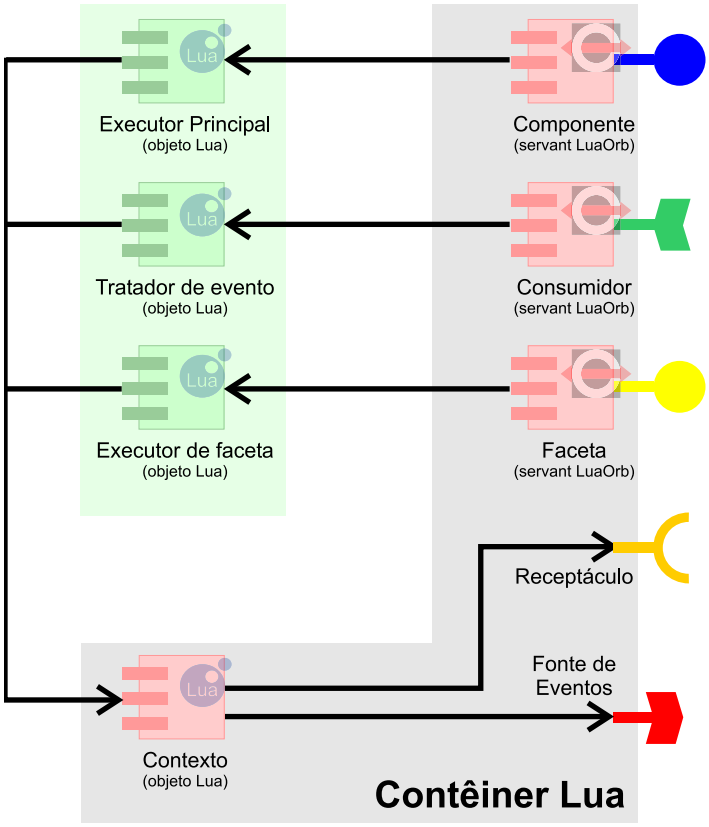

Figura 4.2: Estrutura do contêiner Lua.

que implementa as funcionalidades do contêiner para uma instância de componente (*e.g.*, publicação de interfaces, gerenciamento de conexões, ativação, etc.). Cada Gerenciador de Definição é responsável por gerenciar a classe do envoltório que implementa as funcionalidades do contêiner para uma determinada definição de componente, ou seja, o Gerenciador de Definição é capaz de alterar as funcionalidades do contêiner para as instâncias de uma determinada definição de componente, alterando a classe de objetos a partir da qual os envoltórios de componente são instanciados. Essas classes são dispostas numa hierarquia similar à hierarquia de definições de componentes definida em IDL. Dessa forma, quando uma alteração é feita na definição de um componente, o Gerenciador de Definição correspondente altera a classe de objetos de envoltório do componente correspondente, implicando numa alteração nos serviços fornecidos pelo contêiner para todas as instâncias daquele componente. Inclusive, o modelo dinâmico do LOOP faz com que a adaptação seja refletida automaticamente para todas as instâncias de componentes cuja definição herda do componente alterado originalmente.

Para cada componente instanciado é criado um envoltório a partir da classe LOOP definida pelo Gerenciador de Definição correspondente. O envoltório define implementação somente das operações da interface equivalente que são relacionadas às tarefas do contêiner. As demais operações são delegadas ao executor do componente através do mecanismo de delegação do LOOP. Além disso, o envoltório também cria um objeto de contexto, que é repassado

ao executor do componente através da operação set_session_context, sempre que este é ativado. Assim como o envoltório, o objeto de contexto também é um objeto LOOP que tem sua classe definida pelo Gerenciador de Definição, mas implementa as interfaces internas do contêiner que são oferecidas à implementação do componente. A figura 4.3 ilustra a arquitetura utilizada para gerenciar definições de componentes no LuaCCM, onde participam Gerenciadores de Definição e objetos de envoltório e contexto.

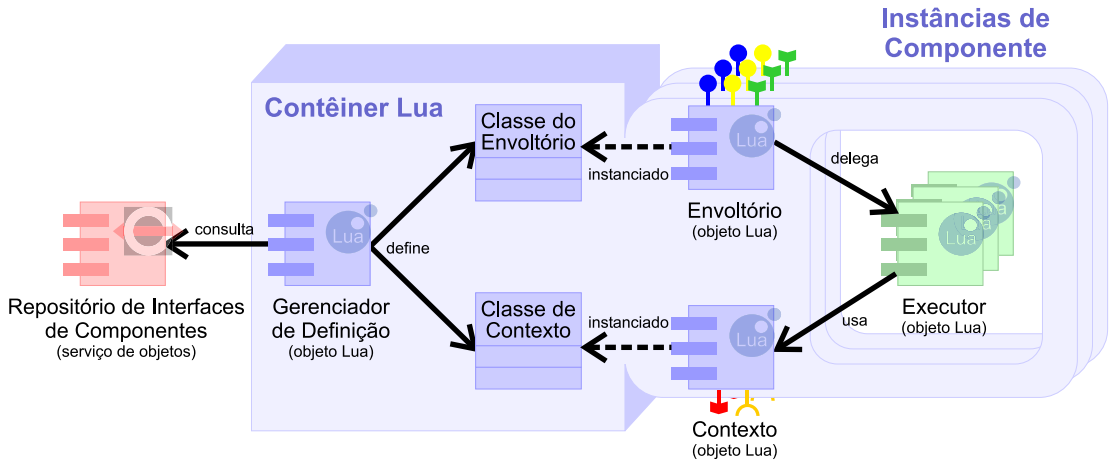

Figura 4.3: Arquitetura de gerência de definições de componentes do LuaCCM.

4.2.2

Interfaces do Contêiner

O modelo CCM define um conjunto de interfaces através das quais o contêiner interage com a implementação do componente (*i.e.*, interfaces internas e de *callback*) e o ambiente externo (*i.e.*, interfaces externas). As interfaces do contêiner Lua são implementadas pelo objeto de contexto e o envoltório. O modelo LuaCCM define classes base para as classes de envoltórios e objetos de contexto, denominadas LuaCCMObject e LuaCCMContext respectivamente. Essas classes definem os serviços comuns oferecidos pelo contêiner a todos os componentes LuaCCM e seus clientes. Além disso, essas classes também implementam serviços relacionados a adaptação dinâmica do contêiner, como será visto na seção 4.4.3.

Quando a definição de um componente não define nenhum componente base, as classes do envoltório e do objeto de contexto herdam diretamente das classes bases LuaCCMObject e LuaCCMContext. Caso contrário, essas classes herdam diretamente das classes do envoltório e do objeto de contexto definidas para o componente base. Dessa forma, a hierarquia de classes criada é capaz de refletir alterações resultantes de alguma mudança na definição de um determinado componente por todas suas instâncias, diretas ou indiretas.

Além das interfaces oferecidas pelo contêiner, o modelo CCM define um conjunto de interfaces que devem ser oferecidas pela implementação do

componente ao contêiner, denominadas de interfaces de *callback*. Entretanto, como a linguagem Lua é dinamicamente tipada e não apresenta o conceito de declaração de interfaces, a geração de interfaces de *callback* não se faz necessária. É necessário, entretanto, que a implementação do componente forneça as operações esperadas pelo contêiner, inclusive quando o contêiner é alterado dinamicamente para refletir alguma mudança na definição do componente. Por essa razão, é necessário utilizar algum mecanismo que permita alterar a implementação do componente para que este se adeque à sua nova definição. Os recursos do LuaCCM para adaptação das implementações do componente serão discutidos na seção 4.4.2

4.2.3

Gerenciamento de Portas

Cada porta de cada instância de componente LuaCCM é implementada por um subcomponente do contêiner Lua, que é responsável por gerenciar as conexões estabelecidas através dessa porta. Há dois tipos de implementações de porta no LuaCCM: ativa e passiva. Implementações de porta ativas são utilizadas para portas que publicam alguma interface e mapeiam operações para segmentos do executor do componente, ou seja, facetas, consumidores de eventos e a interface principal do componente (a interface principal do componente é tratada como uma porta especial). Uma porta ativa é um o objeto dinâmico que implementa a interface correspondente à porta. Esse objeto intercepta todas as interações do meio externo com o componente e as delega ao executor correspondente.

Implementações de porta passivas são criadas para portas que estabelecem conexões com outros componentes, ou seja, receptáculos, emissores e publicadores de eventos. A implementação de uma porta passiva mantém a lista de referências dos objetos conectados a uma determinada porta e é responsável por gerar os identificadores das conexões (*cookies*). Assim como a implementação de porta ativa, a implementação passiva também intercepta todas as interações da implementação do componente com os objetos conectados à porta. Os recursos de interceptação das interações através das portas é utilizado na ativação do componente, assim como no mecanismo de adaptação dinâmica através de interceptadores, como será visto na seção 4.4.1.

A criação das implementações das portas é feita sob demanda durante a execução das operações do envoltório do componente e do objeto de contexto. Dessa forma, quando alguma porta é adicionada à definição de um componente, a execução das operações adicionadas ao envoltório ou objeto de contexto do componente (*e.g.*, provide_nomedafaceta) automaticamente criam a implementação para a nova porta quando são chamadas.

4.2.4

Obtenção de Executores

O modelo CCM define que o executor de um componente seja obtido através da operação create da implementação do seu *home*, que deve devolver um objeto que pode apresentar interfaces diferentes de acordo com o tipo de executor do componente (ver seção 2.2.1). No caso de um componente segmentado, o objeto devolvido é um localizador de executores, que é um objeto que fornece a operação obtain_executor, através da qual o contêiner obtém os diversos segmentos do componente, inclusive o segmento principal. Já no caso de um componente monolítico, o objeto devolvido pode tanto ser o próprio executor do componente, fornecendo diretamente as operações das interfaces oferecidas, ou um localizador de executores, como no caso dos componentes segmentados.

Os *homes* do LuaCCM implementam a forma de obtenção dos executores de componentes seguindo o modelo de uso de CORBA correspondente a categoria do componente, ou seja, *stateless* para componentes de serviço e *conversational* para componentes de sessão. O modelo de uso de CORBA *durable* não é considerado, pois a implementação atual do LuaCCM não dá suporte a componentes persistentes, ou seja, componentes de entidade ou processo. A classe ServiceHome define o comportamento de *homes* de componentes de serviço. Esse tipo de *home* obtém apenas uma única instância de executor de componente e a utiliza na criação de todas as instâncias do componente. Note, que é possível alterar a implementação do LuaCCM para que ele crie um repositório de executores ao invés de utilizar apenas uma única instância. Já os componentes de sessão são criados por instâncias da classe SessionHome. Tais *homes* obtêm uma nova instância de executor do componente para cada nova instância do componente que é criada.

4.2.5

Ativação de Executores

Através do mecanismo de interceptação oferecido pelos gerenciadores de porta, é possível definir ativadores de segmentos, que são subcomponentes do contêiner responsáveis por empregar alguma das políticas de ciclo de vida definidas pelo modelo CCM. A implementação atual do LuaCCM define dois ativadores: MethodActivator e ContainerActivator, que implementam as políticas *method* e *container*, respectivamente. Os ativadores de segmentos são associados aos gerenciadores de porta ativos e interceptam todas as interações dos clientes, ativando e desativando o segmento de acordo com a política adotada.

4.3
Implantação de Componentes

O modelo CCM define uma arquitetura para implantação de componentes e um formato para distribuição de componentes em pacotes de software. Essa arquitetura é formada por um conjunto de interfaces que compõem o *framework* de implantação de componentes CCM. O LuaCCM implementa apenas as interfaces ::Components::ComponentServer e ::Components::Container. Ambas interfaces são fornecidas pelo servidor de componentes LuaCCM. A interface ::Components::ComponentServer oferece a operação create_container através da qual é possível criar contêiners Lua dado uma lista com valores de configuração do contêiner. Os contêiners Lua criados através dessa operação implementam a interface ::Components::Container, que oferece a operação install_home, através da qual é possível instalar pacotes de componentes Lua. A operação install_home recebe três parâmetros. O primeiro parâmetro é o identificador da implementação, que nesta versão do LuaCCM, é uma cadeia de caracteres contento a localização do pacote do componente no sistema de arquivos do servidor. Ou seja, o pacote de componentes deve ser previamente disponibilizado no ambiente do servidor. O segundo parâmetro é a especificação de um ponto de entrada do pacote a partir do qual a implementação do *home* é obtida. O terceiro e último parâmetro é uma lista de valores a serem utilizados na configuração do *home*. Nesta versão do LuaCCM, esse terceiro parâmetro é utilizado para informar características da implementação dos componentes que não são descritas em IDL 3.0, como a categoria e a política de ciclo de vida do componente.

O ponto de entrada do pacote depende da linguagem utilizada na implementação do componente. Entretanto, antes de definir o ponto de entrada, é necessário definir o formato dos pacotes de componentes LuaCCM. Os pacotes de componentes CCM são compostos basicamente pela implementação do componente e pelos arquivos XML de descrição, denominados *descritores*. Esses arquivos XML contêm informações sobre a definição e implementação dos componentes do pacote, que são utilizadas por ferramentas de implantação para criação de contêiners adequados e na configuração de sistemas de componentes. Entretanto, como a linguagem Lua fornece mecanismos de descrição de dados de manipulação mais simples do que os arquivos XML, o LuaCCM utiliza um formato de descritor de componentes próprio, escrito em Lua. O descritor de componentes Lua é mais resumido que os arquivos XML do modelo CCM, contento apenas as informações necessárias para implantação dos componentes nos servidores LuaCCM. Entretanto, vale a pena notar que essa abordagem não impede a utilização de arquivos XML, pois o LuaCCM pode ser estendido para permitir a leitura de arquivos XML e utilizar essa informação para gerar o descritor de componentes LuaCCM correspondente. Como exemplo, considere a declaração de componente da figura 4.4, onde é definido um componente repetidor e seu componente *espelho*[1], que é utilizado para testar

[1]Componente espelho é um componente que tem uma porta diferente para ser conectada a cada porta do componente original, de forma que quando conectado ao componente, possa

o componente repetidor. A figura 4.5 ilustra um descritor LuaCCM correspondente ao componente ::repeater::RepeaterComponent da figura 4.4.

```
1  module repeater {
2    interface FooInterface {
3      void operation();
4    };
5
6    eventtype FooEvent {};
7
8    interface Tester {
9      void test();
10   };
11
12   component RepeaterComponent {
13     provides       FooInterface operation_in;
14     uses multiple  FooInterface operation_out;
15     emits          FooEvent     single_event_out;
16     publishes      FooEvent     multiple_event_out;
17     consumes       FooEvent     event_in;
18   };
19
20   home RepeaterHome manages RepeaterComponent {};
21
22   component RepeaterTester supports Tester {
23     uses      FooInterface operation_caller;
24     provides  FooInterface operation_receiver;
25     consumes  FooEvent     event_receiver_1;
26     consumes  FooEvent     event_receiver_2;
27     emits     FooEvent     event_sender;
28   };
29
30   home RepeaterTesterHome manages RepeaterTester {};
31 };
```

Figura 4.4: Declaração de um componente repetidor e seu componente de teste.

Para a implantação de um componente no servidor LuaCCM, apenas dois elementos são necessários: a implementação do *home* do componente e a definição desse *home* e do componente gerenciado. A implementação do *home* é dada por um objeto Lua. Já a definição do *home* e do componente pode ser fornecida de duas formas: através do descritor de componentes LuaCCM, que consiste de uma tabela contendo as informações sobre a estrutura do componente (*i.e.*, portas, interfaces, eventos, etc.), ou simplesmente informando o nome da definição do *home*. Nesse caso, o servidor consulta um repositório de interfaces de componentes para obter as informações sobre a estrutura do componente. Ou seja, através do uso dos descritores LuaCCM, é possível manipular componentes sem a necessidade de um repositório de interfaces. Vale ressaltar entretanto, que mesmo com o uso de descritores, o LuaOrb consulta o repositório de interfaces para obter as informações sobre as interfaces das portas do componente, de forma a enviar e receber as requisições CORBA apropria-

controlar todos os seus canais de comunicação. Componentes espelhos são utilizados para implementar testes automatizados para uma determinada definição de componente[35].

damente. Contudo, É possível alterar a implementação do LuaOrb, incluindo mecanismos para alimentar diretamente as bases de dados do LuaOrb com as informações necessárias para construção de requisições.

Para tornar essa implementação do LuaCCM mais simples, o formato de pacote adotado é um simples arquivo contendo código Lua, que quando executado deve instalar no ambiente Lua a implementação do componente. Essa instalação deve consistir da criação do executor do *home* e sua definição correspondente, que pode ser uma tabela contendo os campos do descritor LuaCCM ou o nome do *home*. Nesse último caso, o repositório de interfaces de componentes deve conter a definição do *home*. Além disso, a instalação do componente deve criar uma tabela contendo a implementação criada e a definição de componente correspondente e armazená-la numa variável global, que será utilizada como ponto de entrada do pacote para aquela implementação do componente. Note que dessa maneira, se cada implementação de componente for armazenada numa variável global diferente, várias implementações de componentes podem ser embutidas num único pacote, de forma quando o pacote fosse carregado todas as implementações seriam instaladas e poderiam ser obtidas através dos seus respectivos pontos de entrada. Como exemplo, considere o pacote de componentes ilustrado na figura 4.6, onde é definida a implementação do componente repetidor definido na figura 4.4.

O LuaCCM define uma biblioteca de funções que são utilizadas para manipular o servidor de componentes LuaCCM. Basicamente, as funções dessa biblioteca permitem definir o repositório de interfaces de componentes que será utilizado para obter as informações sobre definições de componentes, obter uma referência para o servidor LuaCCM e a função luaccm.createhome, que fornece um mecanismo alternativo de implantação de componentes. Essa função permite implantar componentes localmente sem a necessidade de definição de um pacote. Vale ressaltar que para implantar componentes em servidores remotos, é necessário definir um pacote com o componente e disponibilizá-lo no ambiente de execução do servidor remoto. A função luaccm.createhome recebe o executor do *home* do componente e sua definição por parâmetro (*i.e.*, descritor LuaCCM ou o nome da definição do *home* no repositório de interfaces de componentes). Opcionalmente, a função luaccm.createhome também recebe a referência de um contêiner local, onde o componente deve ser instalado, e uma lista de configuração contendo informações sobre a implementação do componente que não são descritas em IDL, como a categoria ou a política de gerenciamento de ciclo de vida do componente. A figura 4.7 ilustra a implantação do componente repetidor do pacote da figura 4.6 através da arquitetura de implantação de componentes definida pelo modelo CCM. Posteriormente, ainda na figura 4.7, uma implementação componente de teste da figura 4.4 é implantada localmente através da função luaccm.createhome. Por fim, duas instâncias dos componentes são criadas e adequadamente conectadas para a realização do teste.

```
RepeaterDescriptor = {
  home = {
    repid = "IDL:repeater/RepeaterHome:1.0",
    absolute_name = "::repeater::RepeaterHome",
  },
  repid = "IDL:repeater/RepeaterComponent:1.0",
  absolute_name = "::repeater::RepeaterComponent",
  name = "RepeaterComponent",
  category = "service",
  level = "extended",
  lifetime = "method",
  ports = {
    facets = {
      operation_in = {
        repid = "IDL:repeater/FooInterface:1.0",
        iface = "::repeater::FooInterface",
      },
    },
    receptacles = {
      operation_out = {
        multiple = true,
        repid = "IDL:repeater/FooInterface:1.0",
      },
    },
    emitters = {
      single_event_out = {
        repid = "IDL:repeater/FooEvent:1.0",
        consumer_repid = "IDL:repeater/FooEventConsumer:1.0",
      },
    },
    publishers = {
      multiple_event_out = {
        repid = "IDL:repeater/FooEvent:1.0",
        consumer_repid = "IDL:repeater/FooEventConsumer:1.0",
      },
    },
    consumers = {
      event_in = {
        repid = "IDL:repeater/FooEvent:1.0",
        event_type = "::repeater::FooEvent",
      },
    },
  },
},
```

Figura 4.5: Exemplo de descritor de componente LuaCCM.

```
RepeaterEntryPoint = {
  executor = {
    create = function(self)
      return {
        set_session_context = function(self, context)
          self.context = context
        end,
        get_operation_in = function(self)
          return self
        end,
        operation = function(self)
          local connecitons =
            self.context:get_connections_operation_out()
          for index, connection in ipairs(connecitons) do
            connection.objref:operation()
          end
        end,
        push_event_in = function(self, event)
          self.context:push_single_event_out(event)
          self.context:push_multiple_event_out(event)
        end,
      }
    end,
  },
  definition = "::repeater::RepeaterHome",
}
```

Figura 4.6: Exemplo de pacote de componente LuaCCM.

4.4 Componentes Dinâmicos

O LuaCCM é um modelo de componentes baseado no modelo CCM da OMG, que fornece recursos de adaptação em ponto pequeno no nível dos componentes, ou seja, permite realizar alterações nos componentes da aplicação, que serão chamados neste trabalho de *componentes dinâmicos*. Apesar de ser possível construir componentes dinâmicos utilizando objetos dinâmicos através do recurso de DSI de CORBA (ver apêndice A), essa abordagem é muito custosa, pois deixa toda a implementação das tarefas relacionadas ao contêiner para o desenvolvedor do componente. Por isso, o LuaCCM define o modelo de contêiner dinâmico, que simplifica o desenvolvimento desse tipo de componentes. Além disso, faltam abstrações e mecanismos no modelo CCM para manipulação de componentes dinâmicos (ver seção 3.3.3), em especial mecanismos para:

Introspecção: O modelo CCM define um conjunto de interfaces que permitem consultar e acessar as portas de um componente (*i.e.*, Navigation, Receptacles e Events). Apesar de tais interfaces não terem sido elaboradas para esse fim, elas podem ser utilizadas como mecanismo de introspeção de componente dinâmicos, pois permitem manipulá-los independentemente das interfaces equivalentes geradas a partir da definição do com-

```
-- Implantacao atraves da arquitetura de implanatcao CCM
server = luaccm.getserver() -- servidor local
container = server:create_container({})
repeater_home = luaorb.narrow(container:install_home(
  "luaccm/demo/repeater/package.lua",
  "RepeaterEntryPoint", {} ))

-- Implantacao atraves da funcao luaccm.createhome
tester_home = luaccm.createhome({
    create = function()
      return {
        set_session_context = function(self, context)
          self.context = context
        end,
        get_operation_receiver = function(self)
          return self
        end,
        operation = function(self)
          self.called = true
        end,
        push_event_receiver_1 = function(self, event)
          self.receiver1 = true
        end,
        push_event_receiver_2 = function(self, event)
          self.receiver2 = true
        end,
        test = function(self)
          self.context:get_connection_operation_caller():operation()
          self.context:push_event_sender({})
          assert(self.called, "Test failed!")
          assert(self.repeater1, "Test failed!")
          assert(self.repeater2, "Test failed!")
          print("Test successful.")
        end,
      }
    end,
  },
  "::repeater::RepeaterTesterHome")

-- Instanciacao e configuracao dos componentes.
repeater = repeater_home:create()
tester = tester_home:create()

repeater:connect_operation_out(
  tester:provide_operation_receiver())
repeater:connect_single_event_out(
  tester:get_consumer_event_receiver_1())
repeater:subscribe_multiple_event_out(
  tester:get_consumer_event_receiver_2())
tester:connect_operation_caller(
  repeater:provide_operation_in())
tester:connect_event_sender(
  repeater:get_consumer_event_in())

tester:test()
```

Figura 4.7: Exemplo de implantação de componentes LuaCCM.

ponente em IDL 3.0. Entretanto, o modelo CCM não define nenhum mecanismo similar para ser acessado localmente pelo executor do componente através das interfaces internas do contêiner, ou seja, interfaces locais que permitam consultar as características atuais do componente, assim como acessar suas portas independentemente das interfaces equivalentes. Isso pode ser contornado fazendo com que a implementação utilize as interfaces externas de introspeção para consultar a estrutura atual do componente, introduzindo apenas uma sobrecarga de processamento.

Alteração de definição: Assim como consultar a estrutura do componente, é necessário fornecer mecanismos que permitam alterá-la, por exemplo através da adição de novas portas. Contudo, o modelo CCM não define nenhum mecanismo que possa ser utilizado para alterar dinamicamente a definição de um componente. Por essa razão, é necessário definir mecanismos que permitam alterar a definição de um componente dinamicamente, como por exemplo uma meta-interface oferecida pelo componente que permita alterar suas portas, interfaces oferecidas, o componente base, etc.

Alteração de implementação: Alterações na definição do componente geralmente acarretam em alterações na implementação, portanto também é necessário definir mecanismos que permitam alterar a implementação do componente. A flexibilidade e o dinamismo da linguagem Lua aliados aos recursos oferecidos pelo modelo dinâmico do LOOP funcionam como mecanismos de alteração da implementação, tanto do contêiner como dos próprios componentes, uma vez que componentes implementados em Lua podem ser dinamicamente alterados (*e.g.*, substituindo os métodos do objeto que representa a implementação do componente). Além disso, componentes implementados por objetos LOOP podem ter suas instâncias alteradas como um todo através da alteração da definição de sua classe.

4.4.1 Formas de Adaptação

É necessário definir os mecanismos de adaptação disponíveis para alterar a implementação de componentes LuaCCM. Basicamente, duas formas de adaptação são fornecidas: a adição de um novo comportamento através de novas portas ou a alteração do comportamento existente através da substituição da implementação ou interceptação das portas do componente. O LuaCCM provê os recursos necessários para alterar a estrutura de um componente através da adaptação do contêiner, assim como interceptar todas as interações através das portas do componente.

A adição de novas portas ou a substituição da implementação de portas existentes é feita fornecendo um novo segmento que é associado ao executor do componente. O novo segmento adicionado dinamicamente é tratado como os demais segmentos do componente, ou seja, ele é independentemente ativado e

tem acesso ao objeto de contexto do componente. Inclusive, o LuaCCM fornece a operação get_executor no objeto de contexto, que permite acessar os outros segmentos que implementem as demais portas. Da mesma forma, o objeto de contexto oferece a operação get_interceptor, através da qual é possível acessar os interceptadores associados as portas do componente.

Entretanto, a redefinição de componentes pode ser muito destrutiva, pois impossibilita desfazer a adaptação. O uso de interceptadores permite alterar um comportamento existente sem reimplementá-lo completamente, como ocorre no caso da redefinição. As alterações feitas através de interceptação também podem ser desfeitas pela remoção do interceptador. No LuaCCM, interceptadores são definidos como objetos Lua que fornecem as operações before e after. No caso da interceptação de chamadas às operações da interface do componente ou de alguma de suas facetas ou receptáculos, a operação before é utilizada para tratar os parâmetros da chamada, podendo cancelá-la e a operação after é utilizada para tratar o retorno da chamada. No caso da interceptação de emissão ou recebimento de eventos, a operação before é chamada para tratar o evento antes dele ser repassado ao seu destino e a operação after é chamada depois dele ter sido enviado ou tratado pelo executor do componente. A definição de um interceptador LuaCCM pode ser ilustrada da seguinte forma:

```
interceptor = {
    before = function(self, request)
        <implementação>
    end,
    after = function(self, request)
        <implementação>
    end,
}
```

Os interceptadores Lua podem definir outras operações além das operações de interceptação before e after. Além disso, essas operações podem utilizar o parâmetro self para guardar estado ou acessar outras operações do interceptador. O parâmetro request é uma tabela que identifica a requisição sendo interceptada e contém os seguintes campos:

context objeto de contexto do componente.

segment segmento que implementa a porta interceptada.

port nome da porta interceptada.

name nome da operação interceptada.

params lista dos valores dos parâmetros da operação interceptada.

results lista dos valores de retorno da operação interceptada.

Vale ressaltar que uma única tabela é passada para a operação before e after numa requisição, ou seja, é possível adicionar valores à tabela request durante a operação before e depois acessá-los na operação after correspondente. O interceptador pode substituir os valores dos parâmetros utilizados na chamada da operação interceptada, assim como os valores de retorno devolvidos ao cliente. Isso é feito através dos campos params e results. Após a execução da operação before, o campo params deve conter os valores de parâmetros que serão utilizados na chamada da operação no executor do componente. O mesmo acontece após a operação after, pois o campo e results deve conter os valores de retorno que serão devolvidos do cliente da requisição. Note que se nenhuma substituição é feita, então a tabela request não é alterada. Durante a execução da operação before, o valor do campo results é nil. Mas, se após a execução da operação before o campo results for definido, então a chamada da operação é cancelada e os valores do campo results são utilizados como valores de retorno da chamada. Vale ressaltar também, que quando a operação é cancelada, a operação after correspondente nunca é executada.

No caso da interceptação de portas orientadas a eventos, ela é feita como nos demais casos, ou seja, a interceptação é feita nas chamadas das operações de envio de eventos (*i.e.*, dos objetos consumidores e segmentos que implementam as fontes de eventos). Portanto, o nome da operação interceptada pode variar bastante, sendo inclusive dependente da implementação do LuaCCM (*e.g.*, operação deferred_push_<*tipo do evento*>, para fontes de eventos, e a operação push_<*nome da porta*>, para receptores de eventos). Por essa razão, não é recomendada a utilização do campo name por interceptadores de portas orientadas a evento. Entretanto, todas as operações de envio de eventos possuem assinatura similares, pois não possuem valor de retorno e recebem como único parâmetro o evento sendo recebido ou enviado. Por isso, é possível acessar e substituir eventos recebidos ou enviados através do campo params[1]. Para cancelar o envio ou tratamento de um evento basta definir o campo results como uma tabela vazia durante a execução operação before.

4.4.2

Níveis de Adaptação

Como visto na seção 4.1.3, o modelo de classes dinâmicas do LOOP permite realizar alterações no conjunto de todas as instâncias de uma mesma classe ou numa única instância, através da personalização de objetos. Esses recursos permitem realizar alterações em diferentes níveis de abrangência. O LuaCCM utiliza esses recursos para permitir adaptações de componentes em três níveis diferentes, dentro de um mesmo servidor (processo):

Por definição: A alteração é feita no nível da definição do componente, ou seja, a alteração abrange todas as instâncias de todas as diferentes implementações de uma determinada definição de componente.

Por *home*: A alteração é feita no nível de um *home* de componente, ou seja, a alteração abrange todas as instâncias de componente criadas a partir de um determinado *home*.

Por instância: A alteração é feita no nível de uma instância do componente, ou seja, ela só é refletida em uma única instância.

Quando ocorre uma alteração na definição de um componente, o contêiner se adapta através da alteração das classes de envoltório e de contexto definidas pelo Gerenciador de Definição associado àquela definição de componente. Mas quando a alteração é feita em apenas uma única instância do componente, o contêiner se adapta alterando os objetos de envoltório e de contexto associados a essa instância, de forma que esses objetos forneçam os serviços adequados à nova definição de componente. Entretanto, adaptações numa única instância ou sobre todas as instâncias de um componente podem ser inadequadas, como no caso de uma adaptação feita para melhorar a eficiência de uma determinada implementação de componente. Nesse caso, é importante introduzir a possibilidade de realizar a adaptação num nível intermediário, como por exemplo, no nível de um *home*. Cada *home* pode ser utilizado para representar uma implementação diferente de um componente, fazendo com que adaptação no nível de uma implementação do componente possa ser feita através do *home*.

A adaptação no nível de um *home* é feita através de duas classes adicionais criadas para cada *home* instalado no contêiner. Essas classes herdam das classes do envoltório e do objeto de contexto definidas pelo Gerenciador de Definição do componente gerenciado e são utilizadas na criação de componentes a partir daquele *home*. Portanto, quando alguma adaptação é feita sobre as instâncias de um único *home*, ela é aplicada sobre essas classes, de forma que automaticamente todas as instâncias de componentes criadas a partir daquele *home* reflitam a alteração.

As alterações de classes e objetos de envoltório e contexto permitem adaptar o contêiner para uma alteração na estrutura do componente. Contudo, é necessário fornecer mecanismos similares para alterar os executores dos componentes. Isso poderia ser feito através do uso de classes LOOP na implementação desses executores. Entretanto, como o LuaCCM não exige a utilização do modelo LOOP na implementação de componentes, é necessário definir um mecanismo alternativo para adequar os executores à nova estrutura dos componentes adaptados.

Alterações nos receptáculos e fontes de eventos do componente não implicam necessariamente em alteração na implementação do componente, uma vez que representam serviços fornecidos pelo contêiner e apenas acarretam adaptações no próprio contêiner. Entretanto, alterações em facetas e receptores de eventos implicam que a implementação passe a fornecer novos serviços. Esses novos serviços podem ser implementados por novos segmentos do executor, criados à medida que clientes solicitem o serviço e incluídos na implementação do componente separadamente. Assim, é possível adicionar facetas e receptores de eventos a componentes, bastando informar um novo segmento

que implemente a interface oferecida pela faceta, ou um tratador de evento no caso da adição de um novo receptor de eventos.

Quando a implementação de uma nova faceta ou receptor de eventos é fornecida, ela é adicionada ao envoltório como um novo campo do objeto. Dessa forma, quando um cliente solicita uma faceta do componente ou envia um evento, o envoltório verifica se existe um campo que defina o segmento a ser utilizado na implementação da porta. A implementação do segmento pode ser fornecida como um objeto Lua ou uma função. No primeiro caso, o objeto é compartilhado por todas as instâncias de componentes como o segmento que implementa a nova porta. No caso da implementação ser fornecida como uma função, essa função é chamada para construir um novo segmento para cada instância do componente. Caso nenhum campo seja encontrado, o envoltório solicita ao executor do componente que este forneça o segmento adequado, seguido as normas do modelo CCM. Note que dessa forma, o campo contendo a implementação da nova porta pode ser adicionado a várias instâncias através da alteração de sua classe, pois o mecanismo de herança do modelo LOOP faz com que o campo seja propagado por todas as instâncias da classe. Isso permite que a definição da implementação de portas também possa ser feita em níveis de abrangência diferentes. Uma abordagem similar é utilizada na definição de interceptadores, ou seja, a implementação de interceptadores é disponibilizada a todas as instâncias do componente como um campo do envoltório do componente através da classe definida pelo Gerenciador de Definição (nível do componente) ou da classe definida pelo *home* (nível do *home*), ou inclusive na própria instância do envoltório (nível da instância).

4.4.3

Interfaces de Adaptação

Os mecanismos de adaptação do LuaCCM são oferecidos através da interface LuaCCM::Adaptable. Através dessa interface, é possível adicionar e remover portas, alterar os segmentos do executor e inclusive definir interceptadores capazes de interceptar interações através das portas do componente. As operações para adição de facetas e receptores de eventos recebem como parâmetro um trecho de código Lua que deve devolver a implementação da porta a ser adicionada. Novamente, a implementação pode ser um objeto que implementa o novo segmento, que será compartilhado por todas as instâncias, ou uma função que será executada para obter uma nova instância do segmento sempre que necessário. Alternativamente, o trecho de código fornecido pode alterar a implementação dos componentes de forma que elas se adequem à nova definição de componente (*e.g.*, redefinir a classe LOOP que implementa o componente). Nesse caso, nenhum valor deve ser devolvido pelo trecho de código Lua. A interface LuaCCM::Adaptable é apresentada na figura 4.8.

Os componentes LuaCCM automaticamente oferecem a interface LuaCCM::Adaptable através da faceta denominada adaptation. Essa faceta pode ser obtida através da operação provide_facet("adaptation") da interface de introspeção de CCM Components::Navigation. Da mesma forma, é possível incluir

```
#include <CCM.idl>

module LuaCCM {

  typedef string LuaCode;

  exception PortNameAlreadyExists {};
  exception InvalidComponent {};
  exception InvalidPortName {};
  exception InvalidInterceptor {};
  exception NoInterceptor {};
  exception LuaCodeError {
    string message;
    string stack;
  };

  interface Adaptable {
    void add_facet(in string name,
                   in string iface,
                   in LuaCode code)
      raises (PortNameAlreadyExists, LuaCodeError);
    void add_receptacle(in string name,
                        in string iface,
                        in boolean ismultiple)
      raises (PortNameAlreadyExists);
    void add_emitter(in string name,
                     in string event_type)
      raises (PortNameAlreadyExists);
    void add_publisher(in string name,
                       in string event_type)
      raises (PortNameAlreadyExists);
    void add_consumer(in string name,
                      in string event_type,
                      in LuaCode code)
      raises (PortNameAlreadyExists, LuaCodeError);
    void remove_port(in string name)
      raises (InvalidPortName);
    void set_base(in string component_name)
      raises (InvalidComponent);
    void set_implementation(in string name,
                            in LuaCode code)
      raises (InvalidPortName, LuaCodeError);
    void intercept(in string point,
                   in LuaCode code)
      raises (InvalidPortName, InvalidInterceptor, LuaCodeError);
    void unintercept(in string point)
      raises (InvalidPortName,NoInterceptor);
  };

  interface AdaptableContainer : ::Components::Container {
    Adaptable get_component_adaptor(in string absolute_name)
      raises (InvalidComponent);
  };

};
```

Figura 4.8: Módulo de interfaces do LuaCCM.

na definição do componente a cláusula provides LuaCCM::Adaptable adaptation, que resultará na geração da operação provide_adaptation, que também pode ser utilizada para obter a interface de adaptação do LuaCCM para uma instância de componente. Alternativamente, os componentes LuaCCM também podem disponibilizar a interface de adaptação como uma interface oferecida. Portanto, componentes LuaCCM que definam a cláusula supports LuaCCM::Adaptable oferecem as operações de adaptação na interface principal do componente. Alterações feitas através da interface de adaptação fornecida por uma instância de componente resultam em alterações no envoltório e no objeto de contexto da instância do componente. Ou seja, alterações feitas utilizando a faceta de uma instância ou a interface oferecida por uma instância de componente são feitas apenas no nível daquela instância.

Assim como as instâncias de componentes, os *homes* de componentes LuaCCM também implementam a interface de adaptação. Portanto, basta incluir na definição do *home* do componente a cláusula supports LuaCCM::Adaptable. Alterações feitas através da interface de adaptação oferecida por um *home* de componente resultam em alterações nas classes de envoltório e de contexto definidas pelo *home* no momento de sua criação e que são utilizadas na criação das instâncias de seus componentes. Ou seja, alterações feitas utilizando a interface oferecida por um *home* são feitas no nível do *home* e são válidas para todas as instâncias criadas a partir dele.

Alterações no nível da definição de um componente são feitas através da interface LuaCCM::AdaptableContainer, que é ilustrada na figura 4.8. Essa é a interface implementada pelos contêiners Lua e define a operação get_component_adaptor, que devolve um objeto que implementa a interface de adaptação para a definição de componente instalada no contêiner, cujo nome é dado como parâmetro. Alterações feitas através desses objetos de adaptação obtidos através do contêiner Lua resultam em alterações nas classes de envoltório e de contexto definidas pelo *Gerenciador de Definição* daquele componente. Ou seja, as alterações são feitas no nível da definição do componente e são refletidas sobre as instâncias de todas as suas implementações.