4
SCS Java com binding dinâmico

Durante a avaliação da implementação Java do sistema SCS encontramos alguns problemas que dificultariam bastante a implementação dos mecanismos propostos neste trabalho. Um dos principais problemas encontrados é a dependência imposta entre as classes que implementam os componentes e classes fornecidas pelo SCS para disponibilizar esses componentes para acesso remoto. Uma vez que Java não suporta herança múltipla, o fato de a implementação de um componente ter que estender uma classe do sistema impede que ela estenda outras classes, muitas vezes impossibilitando a reutilização de classes existentes para este propósito. Outra desvantagem identificada nesta implementação é a necessidade de gerar código para a implementação dos componentes sempre que a interface do componente sofrer alterações. Como alterações nas interfaces dos componentes são freqüentes durante o desenvolvimento de uma aplicação e a cada alteração é necessário gerar e substituir o código do esqueleto dos componentes alterados, o processo de desenvolvimento acaba se tornando lento e cansativo.

Para superar essas dificuldades, implementamos uma versão alternativa do sistema SCS com o intuito de reduzir ao máximo o acoplamento entre a implementação dos componentes e o SCS, reduzindo consequentemente as exigências sobre as classes que contêm essa implementação, e criar um nível de indireção entre os componentes e o *middleware* que permitisse a introdução de mecanismos de adaptação que pudessem interceptar e manipular as mensagens trocadas entre eles.

Esta versão do SCS foi baseada no conceito de *esqueletos dinâmicos*, que são um tipo especial de contêiner que encapsula a implementação das facetas do componente, efetuando a sua ligação com os mecanismos de comunicação apenas em tempo de execução. A única exigência imposta às implementações de componentes fornecidas ao *esqueleto dinâmico* é que elas sejam estruturalmente compatíveis com as facetas que elas implementam, isto é, para cada operação

declarada pela faceta em IDL, a sua implementação deve definir um método com assinatura equivalente. Mesmo esta exigência é apenas teórica, uma vez que para permitir implementações parciais de componentes, o esqueleto verifica a compatibilidade entre a interface do componente e sua implementação apenas em tempo de execução. Como a ligação entre a implementação e os mecanismos de comunicação dos componentes é feita de maneira dinâmica, a necessidade da geração de código foi completamente eliminada.

O *esqueleto dinâmico* armazena internamente descritores que mantêm uma estrutura de meta classes refletindo a interface das portas oferecidas. Ao receber uma chamada externa em uma faceta, o esqueleto consulta o seu descritor para decidir se está apto a respondê-la. Esta verificação é feita comparando a assinatura da chamada recebida com a assinatura das chamadas oferecidas pela faceta. Em caso afirmativo, o esqueleto constrói dinamicamente uma chamada local equivalente à recebida e a encaminha à implementação da faceta. O retorno da chamada local ou eventuais exceções lançadas são encaminhados para o objeto externo, encerrando assim a chamada. O suporte a facetas do *esqueleto dinâmico* foi implementado utilizando o mecanismo de *Dynamic Skeleton Interface* (DSI) de CORBA, que permite instanciar um objeto servidor CORBA sem especificar a priori qual interface ele irá implementar. Para construir as chamadas locais às implementações das facetas de forma dinâmica, foi utilizada a API reflexiva de Java. Embora atualmente a compatibilidade entre a implementação da faceta e o descritor fornecido se encontre sob total responsabilidade do desenvolvedor, é possível implementar um esquema de verificação que rejeite implementações que não contenham todos os métodos declarados no descritor, evitando assim eventuais erros em tempo de execução.

De maneira análoga, o *esqueleto dinâmico* utiliza os descritores dos receptáculos do componente para gerar *proxies* locais para os componentes externos conectados a eles. Quando a implementação do componente necessita efetuar alguma chamada a um componente externo, ela solicita ao *esqueleto dinâmico* o *proxy* de um componente conectado ao receptáculo desejado e efetua a chamada localmente. O *esqueleto dinâmico* recebe a chamada local, consulta o descritor do receptáculo para obter informações adicionais, incluindo uma referência ao componente externo, constrói dinamicamente uma chamada remota equivalente e a encaminha ao componente externo. O retorno da chamada ou eventuais exceções lançadas são encaminhados à implementação local encerrando a chamada. Para construir chamadas remotas a objetos CORBA de maneira dinâmica é utilizado o mecanismo *Dynamic Invocation*

Interface (DII) de CORBA, que permite construir chamadas dinâmicas de maneira similar à API reflexiva de Java.

A figura 4.1 ilustra o funcionamento do *esqueleto dinâmico*. Na figura, o esqueleto hospeda um componente que declara uma faceta e um receptáculo e duas operações são representadas: Na primeira, o componente externo inferior invoca uma operação definida na faceta do componente hospedado, que está conectada a seu único receptáculo (1). Ao receber esta chamada remota, o esqueleto consulta o descritor da faceta, constrói uma chamada local equivalente e a encaminha à implementação do componente (2). Em seguida, a implementação do componente hospedado efetua uma chamada ao componente externo superior, que está conectado ao receptáculo declarado pelo esqueleto. Para isso, ela obtém um *proxy* equivalente junto ao esqueleto e efetua uma chamada local a ele (3). O esqueleto recebe essa chamada, constrói uma chamada remota equivalente e a encaminha ao componente adequado (4).

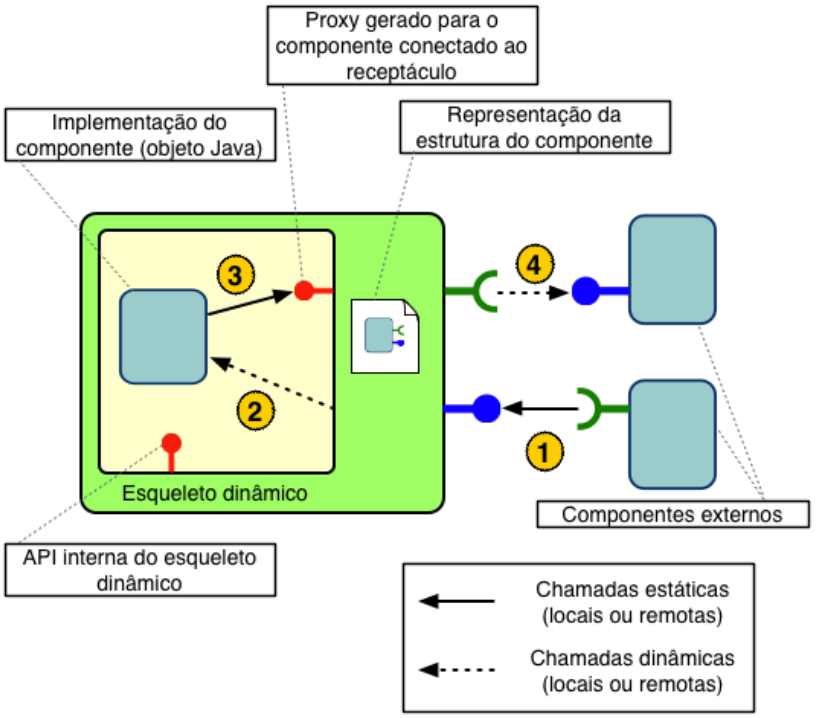

Figura 4.1: Esqueleto dinâmico em Java

Para criar um novo componente utilizando esta versão do sistema SCS devemos inicialmente obter instâncias das meta classes que representam as interfaces das facetas e receptáculos dos componentes. Esse processo, que inicialmente envolvia instanciar objetos para cada entidade da interface (tipos, métodos, parâmetro, exceções, entre outros), foi significativamente simplificado pela incorporação de um *parser* IDL, utilizado para gerar a estrutura de meta

classes a partir da descri¸cao das interface nesta linguagem. Para gerar um *proxy* em Java é necessário informar a interface que será implementada por ele. Como o *esqueleto dinâmico* gera *proxies* para os receptáculos do componente, é necessário anexar à estrutura de meta classes um mapa relacionando o nome de todos os tipo IDL utilizados em receptáculos às interfaces Java que os *proxies* desses tipos devem implementar. Com a estrutura de meta classes e o mapa de tipos, resta instanciar descritores para as facetas e receptáculos desejados e adicionar essas portas ao esqueleto passando como parâmetro o descritor e a implementação, caso a porta seja uma faceta. A seção 4.1 ilustra esse processo com um exemplo.

A implementação de uma faceta pode ser um objeto Java qualquer que implemente as operações declaradas. Caso a implementação de uma faceta dependa de componentes externos ou de outras implementações locais de facetas, o desenvolvedor deve passar a ela uma referência da API interna do *esqueleto dinâmico*. A listagem 4.1 mostra a interface da API interna, que disponibiliza os seguintes métodos:

- *getFacetImpl*: Obtém uma referência a uma implementação local de uma faceta, a partir do identificador de sua interface.
- *getFacetImplByName*: Obtém uma referência de uma implementação local de uma faceta, a partir do seu nome.
- *getReceptacleProxies*: Obtém um conjunto de *proxies* dos componentes conectados a um receptáculo a partir do nome deste receptáculo.
- *getReceptacleProxy*: Obtém o *proxy* de um componente específico conectado a um receptáculo a partir do identificador da conexão.

```
1  package scsimpl.shared;
2  import SCS.CodeError;
3  import SCS.ImplError;
4  import SCS.NoConnection;
5
6  public interface ComponentInternalAPI {
7
8          public Object getFacetImpl(
9                  String facetInterface);
10
11         public Object getFacetImplByName(
12                 String facetName);
13
14         public Object[] getReceptacleProxies(
```

```
15                  String receptacleName);
16
17          public Object getReceptacleProxy(
18                  int id)
19          throws InvalidId;
20
21  }
```

Listing 4.1: API interna do contêiner dinâmico

4.1
Exemplo de uso

O SCS Java foi projetado para tornar o processo de desenvolvimento de aplicações o mais simples possível, reduzindo ao máximo o esforço de programação por parte do desenvolvedor de componentes. A seguir, utilizamos o exemplo *FooBar*, implementado na seção 3.1 com a versão convencional da SCS, para ilustrar o processo de desenvolvimento.

A listagem 4.2 exibe um trecho de código que ilustra a criação de um componente. Conforme visto na seção anterior, o primeiro passo é a instanciação das meta classes que representam as interfaces utilizadas pelo componente. Nas linhas de 1 a 18, declaramos a interface IDL das facetas sob a forma de uma *String* e invocamos o *parser*, que recebe essa descrição e retorna um mapa com as estruturas de meta classes, do qual são extraídos os tipos IDL *Foo* e *Bar*. A seguir, os descritores das facetas são criados (linhas 24 e 25), o componente é instanciado (linha 27), as facetas são adicionadas (linhas 28 e 29) e finalmente o componente é iniciado (linha 32). É importante notar que as implementações das facetas são fornecidas na criação dos seus descritores (linhas 24 e 25). A implementação das facetas do componente é exibida na listagens 4.3 e 4.4. Como essas implementações não dependem de nenhum receptáculo ou recurso do esqueleto, elas não têm nenhuma dependência de classes geradas ou fornecidas pelo sistema.

```
1  String foobarIDL =
2          " module foobar {"+
3          "        interface Foo {"+
4          "                string foo();"+
5          "        };"+
6          "        interface Bar {"+
7          "                string bar();"+
```

```
8           "         };"+
9           "  };";
10
11  HashMap<String, IDLDefinition > idlTypeMap = null;
12  try {
13          idlTypeMap = LuaIDLHelper.getInstance().
14                  parseAllIDLDefinitions(foobarIDL);
15  } catch (LuaException e) {
16          e.printStackTrace();
17  }
18
19  IDLInterfaceType fooType = (IDLInterfaceType)
20          idlTypeMap.get("IDL:foobar/Foo:1.0");
21  IDLInterfaceType barType = (IDLInterfaceType)
22          idlTypeMap.get("IDL:foobar/Bar:1.0");
23
24  FacetDesc fooFacet = new FacetDesc(fooType, "Foo");
25  FacetDesc barFacet = new FacetDesc(barType, "Bar");
26
27  Component fooBarComponent = ComponentFactory.create("FooBar");
28  fooBarComponent.addFacet(fooFacet, new Foo());
29  fooBarComponent.addFacet(barFacet, new Bar());
30
31  try {
32          fooBarComponent.startup();
33  } catch (Exception e) {
34          e.printStackTrace();
35  }
```

Listing 4.2: Criação do componente *FooBar*

```
1  public class Foo {
2          public Foo() {}
3
4          public String foo() {
5                  return "foo";
6          }
7  }
```

Listing 4.3: Implementação da faceta *Foo*

```
1  public class Bar {
2          public Bar() {}
3
4          public String bar() {
5                  return "bar";
6          }
7  }
```

Listing 4.4: Implementação da faceta *Bar*

Caso a implementação das facetas dependesse de algum receptáculo, seria necessário passar ao *parser* um mapa associando o nome do tipo IDL do receptáculo à uma interface Java. A implementação do componente teria ainda que oferecer um construtor que recebesse uma referência para a API interna do componente. A listagem 4.5 ilustra a utilização deste mapa e a adição do receptáculo *FooRecpt*, do mesmo tipo da faceta *Foo*. A listagem 4.6 mostra a implementação da faceta Bar alterada para utilizar o receptáculo *FooRecpt*.

```
1  HashMap<String, IDLDefinition > idlTypeMap = null;
2  HashMap<String, Class> idlJavaMapping =
3          new HashMap<String, Class >();
4  idlJavaMapping.put("IDL:foobar/Foo:1.0:1.0", IFoo.class);
5  try {
6          idlTypeMap = LuaIDLHelper.getInstance().
7                  parseAllIDLDefinitions(foobarIDL,
8                          idlJavaMapping);
9  } catch (LuaException e) {
10         e.printStackTrace();
11 }
12 ...
13 ReceptacleDesc fooRecpt = new ReceptacleDesc(fooType,
14         "FooRecpt",
15         false);
16 fooBarComponent.addReceptacle(fooRecpt);
```

Listing 4.5: Adição de um receptáculo ao componente *FooBar*

```
1  public class Bar {
2          private ComponentInternalAPI component;
3
4          public Bar(ComponentInternalAPI component) {
5                  this.component = component;
6          }
7
8          public String bar() {
9                  try {
10                         IFoo fooRecpt = (IFoo) component.
11                                 getReceptacleProxies(
12                                         "FooRecpt")[0];
13                         return fooRecpt.foo() + "bar";
14                 } catch (NoConnection e) {
15                         e.printStackTrace();
16                 }
```

```
17                  return "bar";
18          }
19  }
```

Listing 4.6: Implementação da faceta *Bar* alterada para utilizar o receptáculo *FooRecpt*

O cliente que utiliza as facetas do componente *FooBar* pode ser idêntico ao descrito na seção 3.1, dependendo inclusive da geração de *stubs* a partir da IDL do componente. Uma versão de um cliente com *stubs* dinâmicos que dispensa completamente a geração de código foi vislumbrada e é mencionada no capítulo de trabalhos futuros.

4.2 Avaliação

Alguns dos recursos de adaptação propostos no próximo capítulo, como a adição de facetas, receptáculos e interceptadores, poderiam ser implementados sem o mecanismo de *binding* dinâmico, utilizando ao invés dele um esquema de geração de código de esqueleto que disponibilizasse pontos de extensão. Esta saída porém, acarretaria em uma grande perda de flexibilidade pela solução. Em primeiro lugar, não seria possível alterar as interfaces das portas em tempo de execução, pois elas estariam ligadas estaticamente ao código gerado. Além disso, a ligação estática dificultaria a implementação dos mecanismos de verificação de tipos baseados em compatibilidade estrutural.

Um dos problemas da versão convencional do SCS que motivou o desenvolvimento da versão com *binding* dinâmico é fato de que as implementações de facetas precisam estender classes geradas a partir da IDL do componente. Como Java não suporta herança múltipla, não é possível utilizar classes existentes que herdem de alguma outra classe como implementação de uma faceta. Na versão dinâmica, a única restrição imposta para a utilização de uma classe como implementação de uma faceta é que ela seja estruturalmente compatível com essa faceta, isto é, para cada operação declarada pela faceta, a sua implementação deve definir um método com assinatura equivalente. O uso de um mecanismo de verificação de tipos baseado em compatibilidade estrutural aumenta significativamente a flexibilidade da solução, pois permite o uso de qualquer objeto compatível na implementação de facetas de componentes.

Outra desvantagem identificada durante o uso da versão convencional do SCS é que o processo de desenvolvimento de componentes se torna cansativo por causa da necessidade de geração de código. Durante o desenvolvimento de uma aplicação, mudanças na interfaces dos componentes são freqüentes, e com essa versão, a cada mudança é necessário gerar o código dos esqueletos novamente. Na versão dinâmica conseguimos simplificar o processo, eliminando a necessidade de geração de código. Uma última vantagem da versão dinâmica frente a versão convencional do SCS é que o nível de indireção introduzido pelo esqueleto dinâmico facilitou a implementação dos mecanismos de adaptação dinâmica discutidos no capítulo seguinte deste trabalho.

Como na versão dinâmica as mensagens entre componentes são interceptadas, inspecionadas e reconstruídas pelo esqueleto dinâmico em tempo de execução, fica claro que o desempenho das aplicações desenvolvidas com essa versão será inferior ao de aplicações idênticas desenvolvidas com a versão convencional. A seção 6.1 descreve uma série de testes que foram realizados para tentar medir o impacto que o uso da versão dinâmica do SCS tem sobre o desempenho das aplicações.