5
Adaptação dinâmica no SCS Java

Os mecanismos de adaptação dinâmica implementados neste trabalho são fortemente inspirados nos implementados pelo *LuaCCM*, descritos na seção 2.5.2. Assim como no *LuaCCM*, as adaptações implementadas permitem efetuar alterações no conjunto de portas dos componentes e em sua implementação. Implementar adaptações de granularidade mais fina, que envolvam alterações nas operações das portas, representaria um grande desafio devido ao esquema estático de verificação de tipos de Java, além de que acarretariam em uma grande sobrecarga na API do sistema. Adaptações de granularidade mais baixa, envolvendo alterações nas conexões entre os componentes, já são permitidas pelo modelo de configuração do SCS, embora um melhor suporte a esse tipo de adaptação, que permita por exemplo criar conjuntos de adaptações que possam ser aplicados de maneira única ou aplicar uma mesma adaptação a um grupo de componentes, seja um tema promissor para trabalhos futuros.

Apesar das semelhanças, os mecanismos implementados sofreram algumas alterações, tanto para torná-los compatíveis com os recursos oferecidos pela linguagem Java, quanto para a incorporação de melhorias identificadas. A alteração mais importante se deu na forma como a implementação das funcionalidades substituídas ou adicionadas ao componente seriam fornecidas ao seu contêiner. A opção mais natural seria fornecê-la sob a forma de arquivos binários com classes Java compiladas, porém, após uma análise mais cuidadosa, optamos por privilegiar a flexibilidade do sistema, disponibilizando três formas diferentes de tráfego de implementação, descritas a seguir:

– Implementações trafegadas como *byte-code* Java: A abordagem mais imediata para esta questão foi o tráfego da implementação das funcionalidades sob a forma de arquivos binários contendo o *byte-code* de classes Java compiladas. Desta forma, para adicionar ou substituir uma faceta de uma aplicação, o desenvolvedor fornece um arquivo JAR (*Java Archive*) contendo as classes Java compiladas que a implementam, sendo que uma

delas deve atuar como fachada, implementando todas as operações declaradas na definição desta faceta. O contêiner se encarrega de carregar as novas classes, instanciá-las e associar as instâncias criadas à faceta informada pelo desenvolvedor.

Essa solução se mostra adequada para os casos onde a adaptação a ser aplicada tem uma complexidade razoável, pois a sua implementação deve ser compilada e testada pelo desenvolvedor antes de ser aplicada. Por outro lado, para adaptações de baixa complexidade, que envolvam pouco código e não tenham impacto significativo na aplicação, essa solução pode ser excessivamente trabalhosa.

– Implementações trafegadas como código Java: Para adaptações mais simples, que possam ser implementadas em apenas uma classe Java e não tenham impacto sobre a aplicação que justifique um processo de testes mais formal, seria ideal permitir que o desenvolvedor fornecesse diretamente o código fonte da alteração, ficando livre de compilá-lo e empacotá-lo. Para implementar esta solução, o contêiner compila o código Java fornecido utilizando a interface programática do compilador Java (*javac*), carrega e instancia as classes geradas e associa as instâncias criadas à porta informada pelo desenvolvedor. Assim como no caso anterior, a classe fornecida deve implementar todas as operações declaradas na definição da faceta alterada.
– Implementações trafegadas como código Lua: Um dos principais argumentos a favor da linguagem Lua, no que diz respeito à facilidade de uso pelo desenvolvedor, é a agilidade proporcionada pelo seu sistema de tipagem dinâmica. Sem a rigidez imposta pela declaração formal de tipos, o tempo de codificação pode ser significativamente reduzido. Em alterações de baixa complexidade, a agilidade e flexibilidade proporcionada por esta linguagem tem grande valor. Para permitir que o desenvolvedor forneça a implementação de adaptações também sob a forma de código Lua, o contêiner utiliza o *binding LuaJava* [42, 43] para carregar o trecho de código fornecido e efetuar chamadas às operações definidas nele. O trecho de código Lua fornecido, quando executado pelo contêiner, deve retornar uma tabela com todas as funções declaradas pela faceta alterada indexadas pelos seu nome, de forma a mimetizar um comportamento orientado a objeto.

Todos os recursos de adaptação disponibilizados pelo SCS são acessados através da faceta *IAdaptable*, que pode ser oferecida por todos os componentes da

mesma forma que as demais facetas de gerência e inspeção. As operações oferecidas por essa interface, exibida na listagem 5.1, são as seguintes:

- *addFacet*: Adiciona uma faceta ao componente. Recebe como parâmetros o nome da faceta (parâmetro *name*), um trecho de código IDL que contém a definição de sua interface (parâmetro *idlDef*), o nome da interface que deve ser utilizada (parâmetro *idlInterfaceName*), a implementação da faceta sob a forma de uma seqüência de bytes (parâmetro *impl*), um indicador que sinaliza que tipo de implementação está sendo fornecido (parâmetro *impl_type* - atualmente pode assumir os valores *java_code*, *java_bytecode* ou *lua_code*) e o nome da classe contida na implementação fornecida que contém as operações declaradas na faceta (parâmetro *className*). Lança exceções caso já exista alguma porta com o nome informado (exceção *PortNameAlreadyExists*), caso existam erros de compilação na implementação (exceção *CodeError*), erros de sintaxe na definição de interface fornecida (exceção *IDLDefError*), ou caso a implementação fornecida não seja estruturalmente compatível com a faceta (exceção *ImplError*).
- *removeFacet*: Remove uma faceta do componente. Recebe como parâmetro o nome da faceta a ser removida (parâmetro *name*) e lança uma exceção no caso de não haver uma faceta com o nome informado (exceção *InvalidPortName*).
- *updateFacet*: Atualiza a definição e a implementação de uma faceta. Assim com a operação *addFacet*, recebe como parâmetros o nome da faceta, a nova definição de sua interface, o nome da nova interface, a nova implementação da faceta, o indicador de tipo de implementação e o nome da classe da implementação. Lança exceções caso não exista faceta com o nome informado (exceção *InvalidPortName*), caso existam erros de compilação na implementação (exceção *CodeError*), erros de sintaxe na definição de interface fornecida (exceção *IDLDefError*), ou caso a implementação fornecida não seja estruturalmente compatível com a faceta (exceção *ImplError*).
- *addReceptacle*: Adiciona um receptáculo ao componente. Recebe como parâmetros o nome do receptáculo (parâmetro *name*), um trecho de código IDL que contém a definição de sua interface (parâmetro *idlDef*), o nome da interface que deve ser utilizada (parâmetro *idlInterfaceName*) e um indicador que sinaliza se o novo receptáculo deve aceitar conexões múltiplas (parâmetro *ismultiple*). Lança exceções caso já exista alguma

porta com o nome informado (exceção *PortNameAlreadyExists*) ou caso haja algum erro na definição de interface fornecida (exceção *IDLDefError*).

- *removeReceptacle*: Remove um receptáculo do componente. Recebe como parâmetro o nome o receptáculo a ser removido (parâmetro *name*) e lança uma exceção no caso de não haver um receptáculo com o nome informado (exceção *InvalidPortName*).
- *setReceptacleDef*: Altera a definição da interface de um receptáculo do componente. Assim como a operação de adição de receptáculos, recebe como parâmetros o nome do receptáculo (parâmetro *name*), um trecho de código IDL que contém a definição de sua interface (parâmetro *idlDef*), o nome da interface que deve ser utilizada (parâmetro *idlInterfaceName*) e um indicador que sinaliza se o novo receptáculo deve aceitar conexões múltiplas (parâmetro *ismultiple*). Lança exceções caso não haja um receptáculo com o nome informado (exceção *InvalidPortName*) e caso haja algum erro na definição de interface fornecida (exceção *IDLDefError*).
- *addInterceptor*: Adiciona um interceptador a uma porta do componente. Recebe como parâmetros o nome da porta que deve ser interceptada (parâmetro *portName*), a implementação do interceptador sob a forma de uma seqüência de bytes (parâmetro *impl*), um indicador que sinaliza que tipo de implementação está sendo fornecido (parâmetro *impl_type* - pode assumir os valores *java_code*, *java_bytecode* ou *lua_code*) e o nome da classe da implementação (parâmetro *className*). Retorna o identificador do interceptador adicionado (único para a porta) e lança exceções caso não haja uma porta com o nome informado (exceção *InvalidPortName*), caso existam erros de compilação (exceção *CodeError*) ou caso a implementação fornecida não declare as operações esperadas de um interceptador(exceção *ImplError*).
- *removeInterceptor*: Remove um interceptador de uma porta do componente. Recebe como parâmetros o nome da porta que contém o interceptador a ser removido (parâmetro *portName*) e o seu identificador (parâmetro *id*), e lança exceções caso não haja uma porta com o nome informado (exceção *InvalidPortName*) ou caso não exista interceptador com o código informado (exceção *InvalidInterceptor*).

```
1 module SCS {
2         ...
3         interface IAdaptable {
4                 void addFacet(in string name,
```

```
5                       in string idlDef,
6                       in string idlInterfaceName,
7                       in string className,
8                       in OctetSeq impl,
9                       in string impl_type)
10                  raises (PortNameAlreadyExists,
11                      ImplError,
12                      CodeError,
13                      IDLDefError);
14
15                  void removeFacet(in string name)
16                  raises (InvalidPortName);
17
18                  void updateFacet(in string name,
19                      in string idlDef,
20                      in string idlInterfaceName,
21                      in string className,
22                      in OctetSeq impl,
23                      in string impl_type)
24                  raises (InvalidPortName,
25                      ImplError,
26                      CodeError,
27                      IDLDefError);
28
29                  void addReceptacle(in string name,
30                      in string idlDef,
31                      in string idlInterfaceName,
32                      in boolean ismultiple)
33                  raises (PortNameAlreadyExists,
34                      IDLDefError);
35
36                  void removeReceptacle(in string name)
37                  raises (InvalidPortName);
38
39                  void setReceptacleDef(in string name,
40                      in string idlDef,
41                      in string idlInterfaceName,
42                      in boolean ismultiple)
43                  raises (InvalidPortName,
44                      IDLDefError);
45
46                  long addInterceptor(in string portName,
47                      in string className,
48                      in OctetSeq impl,
49                      in string impl_type)
50                  raises (ImplError,
51                      CodeError,
```

```
52                          InvalidPortName);
53
54                  void removeInterceptor(in string portName,
55                          in long id)
56                  raises (InvalidPortName,
57                          InvalidInterceptor);
58  };
```

Listing 5.1: Interface *SCS::IAdaptable*

A interface *IAdaptable* é derivada da interface do *LuaCCM* *LuaCCM::Adaptable*, que sofreu uma série de alterações para se adaptar aos recursos descritos neste trabalho:

- As operações de adição de emissores, publicadores e consumidores de eventos foram removidas, pois o SCS não oferece suporte a esses tipos de portas.
- As operações que recebiam como parâmetro o nome de uma interface, como *addFacet* ou *addReceptacle*, passaram a receber também a sua descrição em IDL, necessária para a instanciação das meta-classes.
- As operações que recebiam código Lua, como *addFacet* e *setImplementation*, passaram a receber um *array* de *bytes*, um identificador de tipo de implementação, de forma a contemplar os diversos tipos de tráfego de implementação, e o nome da classe que deve ser utilizada como ponto de entrada da implementação.
- A operação *setImplementation* foi renomeada para *updateFacet*, pois apenas as facetas podem ter sua implementação alterada, e teve seus argumentos alterados para permitir também a alteração da interface da faceta.
- As operações *intercept* e *unintercept* foram renomeadas para *addInterceptor* e *removeInterceptor*, sendo que a última passou a receber também o identificador do interceptador a ser removido. Essa alteração foi efetuada pois o mecanismo de interceptação passou a dar suporte a múltiplos interceptadores.

O mecanismo de interceptadores do *LuaCCM* foi estendido para permitir a adição de múltiplos interceptadores em uma mesma porta. Desta forma, ao adicionar um interceptador a uma porta, o desenvolvedor recebe um identificador, único para aquela porta, que deve ser usado posteriormente para removê-lo da porta. Os interceptadores são executados em ordem FIFO (*First*

In, First Out), isto é, os primeiros a serem adicionados serão os primeiros a serem executados. A implementação fornecida para um interceptador deve conter as operações *before* e *after*, que serão invocadas antes e depois da operação interceptada, respectivamente. Esses métodos recebem como parâmetro uma referência à faceta *IComponent* do componente cuja porta está sendo interceptada, o nome da porta e da operação interceptada e uma lista com os parâmetros passados a esta operação. Implementações de interceptadores em Lua precisam apenas definir as duas funções enquanto implementações Java também precisam implementar a interface *scsimpl.shared.IInterceptor*, exibida na listagem 5.2.

```
1  package scsimpl.shared;
2
3  import SCS.IComponentOperations;
4
5  /**
6   * Define os métodos que devem ser implementados por um
7   * interceptador
8   */
9  public interface IInterceptor {
10         /**
11          * Método invocado ANTES da invocação da operação
12          * @param component IComponent que disponibiliza a porta
13          *  interceptada
14          * @param portName Nome da porta interceptada
15          * @param operation Nome da operação invocada
16          * @param args Argumentos passados à operação invocada
17          */
18         public void before(
19                 IComponentOperations component,
20                 String portName,
21                 String operation,
22                 org.omg.CORBA.NVList args);
23
24         /**
25          * Método invocado ANTES da invocação da operação
26          * @param component IComponent que disponibiliza a porta
27          *  interceptada
28          * @param portName Nome da porta interceptada
29          * @param operation Nome da operação invocada
30          * @param args Argumentos passados à operação invocada
31          */
32         public void after(
33                 IComponentOperations component,
34                 String portName,
35                 String operation,
```

```
36                  org.omg.CORBA.NVList args);
37 }
```

Listing 5.2: Interface *IInterceptor*

5.1 Implementação

A implementação dos mecanismos de adaptação dinâmica introduziu uma série de novos desafios à implementação do contêiner baseado em esqueletos dinâmicos. O primeiro deles surgiu a partir da necessidade de oferecer uma forma mais simples de informar ao contêiner a definição das facetas e receptáculos expostos pelo componente. Na implementação original, essas definições eram fornecidas através de meta-classes que descreviam as interfaces dessas portas. Porém, além desta solução ter se mostrado muito trabalhosa, com a possibilidade de alterar remotamente as definições das portas em tempo de execução, surgiu a necessidade de trafegar essas definições (e conseqüentemente toda a estrutura de tipos das meta-classes) entre os componentes, o que acarretaria em um aumento significativo na complexidade da API da solução. Para contornar essa dificuldade e tornar a solução mais amigável ao desenvolvedor, um *parser* IDL foi incorporado ao SCS Java para permitir que as representações das portas passassem a ser informadas sob a forma de trechos de código contendo as definições escritas nesta linguagem. O *parser* IDL utilizado foi o *LuaIDL* [44], que recebe como entrada uma definição em IDL e retorna um conjunto estruturado de tabelas Lua equivalente. Para acoplar este *parser* ao SCS Java, utilizamos novamente o *binding LuaJava*. A partir dos objetos criados pelo *LuaJava* foi possível instanciar as meta classes do SCS Java de modo a refletir as interfaces fornecidas.

Outro obstáculo que surgiu em decorrência da implementação dos mecanismos de adaptação dinâmica é relacionado à facilidade de geração de *proxies* de componentes externos do contêiner dinâmico. Na implementação original, o contêiner dinâmico recebe para cada receptáculo declarado um conjunto de meta-classes e uma interface Java que refletem a sua definição. Quando um componente externo se conecta a algum receptáculo, o contêiner utiliza o mecanismo de *proxies* dinâmicos da API reflexiva de Java para gerar para ele um *proxy* que implementa a interface Java do receptáculo. O problema surge com a adição de novos receptáculos em tempo de execução, pois para gerar *proxies* para eles, é necessário conhecer as interfaces Java que eles implementam. Exigir

que um usuário remoto forneça uma interface Java no momento da adição de um receptáculo não é uma alternativa razoável, pois além de sobrecarregar a assinatura do método, essa interface a princípio seria conhecida apenas por este usuário. A solução encontrada foi alterar a API interna do componente para permitir que ao solicitar um *proxy* de receptáculo, as classes que implementam o componente forneçam uma interface Java equivalente à do receptáculo desejado. Ao adicionar um novo receptáculo, consideramos aceitável supor que nenhuma implementação previamente carregada no contêiner fará uso dele e que as novas implementações carregadas depois de sua adição conheçam a sua interface.

A possibilidade de fornecer a implementação de facetas sob a forma de código Lua também levantou algumas questões que merecem ser mencionadas. O contêiner efetua as chamadas a implementações Java de componentes através da API reflexiva da linguagem Java. Desta forma, a abordagem que geraria menos impacto sobre a implementação anterior seria criar um "adaptador"que fosse visto pelo contêiner como uma implementação Java comum e traduzisse as chamadas recebidas em chamadas ao código Lua. A maneira que encontramos de implementar tal mecanismo foi a utilização do recurso de geração de *proxies* Java. Infelizmente, esse mecanismo necessita que seja fornecida uma interface Java equivalente ao *proxy* gerado, e o contêiner não recebe esta interface junto com as implementações das facetas. Assim como no caso dos receptáculos, alterar a API não seria uma alternativa razoável. Uma saída trabalhosa seria gerar uma interface Java equivalente à interface da faceta sempre que fosse fornecida uma implementação Lua. Outra solução alternativa seria tornar o mecanismo de invocação do contêiner dinâmico uma entidade abstrata com extensões que efetuassem invocações em implementações Lua e Java. Optamos por adotar essa última solução por ela ser uma alternativa mais simples e elegante, no sentido de evitar a geração desnecessária de código, e por ser uma solução extensível, que possibilita a incorporação futura de novos mecanismos de invocação. Para implementar essa alternativa, a classe *Facet*, que implementava o método de invocação das facetas, passou a ser uma classe abstrata que apenas declara esse método, e duas sub-classes que o implementam foram criadas: a classe *JavaFacet*, que trata invocações a implementações Java e a classe *LuaFacet*, que trata invocações a implementações Lua. Ao adicionar uma nova faceta, o contêiner decide qual sub-classe instanciar baseado no tipo de implementação fornecida.

Em alguns dos experimentos descritos no capítulo seguinte surgiu a necessidade de carregar novas classes no contêiner em tempo de execução. No

exemplo de depuração distribuída, por exemplo, a funcionalidade de adição de inspetores permite a instalação de classes que inspecionam o estado interno do componente. A implementação dos inspetores é passada como de costume, como código Java, código Lua ou bytecode Java, e deve ser carregada no ClassLoader do componente para que possa ser invocada posteriormente através da faceta de inspeção. Para satisfazer a esta demanda, adicionamos à API interna do contêiner a operação loadImpl, que recebe uma implementação Java ou Lua de forma similar às operações de adição de facetas e interceptadores, carrega ela no ClassLoader do contêiner e retorna uma instância dela, que pode ser um proxy caso a implementação fornecida seja em Lua.

Uma questão que não foi abordada na implementação do SCS adaptável diz respeito à consistência das aplicações durante sua adaptação, citada na seção de considerações finais do capítulo de trabalhos relacionados (seção 2.6). Como não foram implementados mecanismos que permitam que um conjunto de adaptações seja aplicado de maneira atômica, é possível que o sistema apresente erros inesperados caso receba requisições durante a aplicação de uma adaptação ou entre a aplicação de adaptações dependentes. Algumas praticas podem ser adotadas no intuito de reduzir o tempo de inconsistência dos componentes. Para criar uma nova versão de um componente por exemplo, é preferível adicionar novas portas e efetuar reconexões, ao invés de substituir a implementação das portas existentes. Entretanto, a criação de mecanismos que garantam a consistência das aplicações esta prevista na seção de trabalhos futuros (seção 7.1).