SCS-Lua - Tutorial - Básico

Tecgraf
Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio)
scs-users@tecgraf.puc-rio.br

2011-07-13

1 Introdução

Este documento é um tutorial básico sobre a criação de componentes no modelo SCS v1.3.0, utilizando a versão Lua da implementação padrão. Não serão encontradas aqui explicações sobre o modelo, as quais encontram-se em documentos específicos. Também não será abordado o uso de serviços específicos desenvolvidos para o auxílio ao uso do modelo, como a infra-estrutura de execução. Essas informações também podem ser obtidas em outros documentos. A implementação SCS-Lua 1.3.0 baseia-se na versão 5.1 da máquina virtual Lua e em CORBA v2.3, representada pelo ORB OiL v0.5. Este documento assume que o leitor é familiarizado a conceitos de desenvolvimento de *software* baseado em componentes e à terminologia CORBA.

2 Inicialização do ORB e Carga da IDL do SCS

Para a criação e execução do código de um componente, é necessária a inicialização prévia de um ORB. A instância de ORB criada será passada por parâmetro posteriormente para o construtor de um componente SCS. O desenvolvedor da aplicação também é responsável por carregar a IDL do SCS. O procedimento deve ser feito de acordo com o código do Código 1.

Código 1: Criação do ORB

```
1  local oil = require "oil"
2  local orb = oil.init()
3  -- A linha abaixo assume que o arquivo scs.idl esteja acessível.
4  -- Pode ser necessário informar um caminho completo ou relativo.
5  orb:loadidlfile("scs.idl")
```

O método oil.init pode receber parâmetros, descritos na documentação do OiL.

3 Passos Necessários à Criação de um Componente

Aqui serão descritos os passos mínimos necessários para a criação de um componente SCS-Lua.

3.1 Definição do Identificador do Componente

O identificador do componente é uma estrutura definida em IDL (scs.idl) chamada ComponentId, e representada em Lua por uma tabela com os respectivos campos preenchidos. Um identificador de componente conta com os seguintes campos:

- `name`: Nome desejado para o componente.
- `major_version`: Número que define a versão principal do componente.
- `minor_version`: Número que define a versão secundária do componente, possivelmente relacionado a uma sub-versão da versão principal.
- `patch_version`: Número que define a versão de revisão do componente.
- `platform_spec`: *String* contendo quaisquer especificações de plataforma necessárias ao funcionamento do componente.

Os números de versão do componente, quando unificados, devem ser separados por pontos. Ou seja, um componente com versão principal 1, versão secundária 0 e versão de revisão 0 deve ser representado como a *String* "1.0.0".

3.2 Criação do Componente Básico

Todo componente SCS-Lua é representado por seu "contexto", que é a tabela Lua retornada após a criação de um novo componente. Essa tabela lua será também uma instância de classe LOOP, que chamamos de *ComponentContext*. Um Contexto de Componente atua como um envólucro local para as facetas e receptáculos de um componente SCS.

A classe *ComponentContext* é implementada pelo módulo Lua *scs.core.ComponentContext* e seu processo de instanciação engloba a criação das três facetas básicas, *IComponent*, *IReceptacles* e *IMetaInterface*. Caso o usuário deseje utilizar uma implementação diferente de alguma dessas facetas, existe no contexto um método para a atualização de facetas chamado *updateFacet*, descrito na Seção 3.3.

Como o contexto é quem cria os objetos CORBA, é necessário que tenha acesso ao ORB logo em sua construção, para que possa inserir as facetas básicas e também facetas adicionais, posteriormente. O ORB fornecido deve ter a IDL do SCS carregada, como mencionado na Seção 2, assim como as IDLs que definam quaisquer facetas adicionais. Outro parâmetro obrigatório é o Identificador do Componente (Seção 3.1).

Um exemplo de código para a criação de um componente básico pode ser visto no Código 2.

A classe ComponentContext aceita mais um parâmetro, opcional, em seu construtor: uma tabela contendo chaves para as facetas básicas. Essas chaves são utilizadas como a chave do objeto CORBA no ORB, para a criação de referências persistentes. Caso uma chave não seja fornecida, o ORB automaticamente gera uma aleatória, que geralmente não é do interesse da aplicação. A tabela de chaves deve ter como índices os nomes das facetas básicas, e como valores as chaves de tipo *string*. Não é necessário fornecer chaves para todas as facetas básicas.

Um exemplo de código para a criação de um componente com chaves definidas pelo usuário para as facetas básicas pode ser visto no Código 3.

Código 2: Instanciação de um Novo Componente

```
local oil = require "oil"
local ComponentContext    = require "scs.core.ComponentContext"

-- Criação do ORB e carga da IDL do SCS
local orb = oil.init()
orb:loadidlfile("scs.idl")

oil.main(function()
  -- cria uma thread para que o ORB passe a aguardar chamadas remotas
  oil.newthread(orb.run, orb)

  -- Criação do Identificador do Componente
  local componentId = {
    name = "MyComponent",
    major_version = 1,
    minor_version = 0,
    patch_version = 0,
    platform_spec = "lua"
  }

  -- Instanciação de um componente básico
  local context = ComponentContext(orb, componentId)
end)
```

3.3 Criação de Facetas

Facetas são interfaces CORBA, e devem ser implementadas pelo desenvolvedor da aplicação, como exigido pelas definições Lua desse padrão. No SCS-Lua, implementações de facetas podem ser tabelas simples. No entanto, é comum utilizar-se de orientação a objetos para a implementação de uma faceta. A biblioteca LOOP facilita o uso do paradigma de orientação a objetos em Lua.

Um exemplo de implementação de faceta com uso da biblioteca LOOP pode ser conferido no Código 5. Essa faceta precisa ter uma especificação em IDL. Para o nosso exemplo, utilizaremos a IDL contida no Código 4.

Essa implementação posteriormente poderá ser instanciada e inserida em um componente como uma nova faceta. Para adicionar uma nova faceta a um componente, o contexto fornece o método *addFacet*, que espera como parâmetros o nome, a interface e a implementação da faceta. Opcionalmente, também pode ser fornecida uma chave para a faceta, como explicado na Seção 3.2 para as facetas básicas. O uso desse método pode ser visto no Código 6.

No ato da adição de uma faceta a um componente, é realizada uma cópia da instância da faceta. A instância precisa ser alterada e assim a cópia é inserida no componente,

Código 3: Instanciação de um Novo Componente com Chaves

```
local oil = require "oil"
local ComponentContext    = require "scs.core.ComponentContext"

-- Criação do ORB e carga da IDL do SCS
local orb = oil.init()
orb:loadidlfile("scs.idl")

oil.main(function()
  -- cria uma thread para que o ORB passe a aguardar chamadas remotas
  oil.newthread(orb.run, orb)

  -- Criação do Identificador do Componente
  local componentId = {
    name = "MyComponent",
    major_version = 1,
    minor_version = 0,
    patch_version = 0,
    platform_spec = "lua"
  }

  -- Criação da tabela de chaves para duas das facetas básicas.
  local keys = {
    IComponent = "IC",
    IMetaInterface = "IM"
  }

  -- Instanciação de um componente com chaves definidas para as facetas básicas
  local context = ComponentContext(orb, componentId, keys)
end)
```

para evitar modificações no objeto fornecido pelo usuário. Será automaticamente inserido um campo "context"na faceta, com uma referência para o contexto do seu componente. O Contexto pode ser utilizado para acessar outras facetas e o identificador do componente, entre outros dados, como descrito na Seção 3.2. Além disso, é criada uma variável de mesmo nome da faceta dentro do contexto, que referencia diretamente o objeto CORBA da faceta. Por isso, não é possível utilizar o mesmo nome para uma faceta e um receptáculo.

O SCS-Lua exige ainda que facetas implementem o método *_component* de CORBA,

Código 4: Exemplo de IDL de uma Faceta

```
module mymodule{
  interface MyFacet {
    void myMethod();
  };
  interface AnotherFacet {
    void anotherMethod();
  };
};
```

Código 5: Implementação de uma Faceta MyFacet

```lua
local oo = require "loop.base"

-- Implementação do construtor
local MyFacet = oo.class{}
function MyFacet:__init()
  return oo.rawnew(self, {})
end

-- Implementação de um método
function MyFacet:myMethod()
  ...
end
```

Código 6: Adição de uma Faceta MyFacet a um Componente

```lua
-- Implementação da faceta
...

-- Criação do componente
...

-- Instanciação e adição da faceta ao componente
local facetInstance = MyFacet()
-- O último parâmetro é opcional
context:addFacet("MyFacetName", "IDL:mymodule/MyFacet:1.0", facetInstance, "MyKey")
```

definido pelo OiL, mas esse método já é inserido automaticamente em qualquer faceta adicionada a um componente. Esse método é o mesmo que o *_get_component* do ORB JacORB para Java e retorna o objeto CORBA da faceta IComponent. Em Lua deve-se sempre chamar *_component()*, independente da linguagem do objeto remoto.

Por fim, é possível substituir a implementação de uma faceta por uma diferente. Isso é feito através do método *updateFacet*. O método remove a faceta antiga e adiciona a nova, mas mantém o nome, interface e a chave definida pelo usuário (se não houver sido fornecida, uma nova é gerada aleatoriamente). O Código 7 mostra o uso do método.

Código 7: Atualização de Uma Faceta Básica

```lua
-- Nova implementação da faceta IComponent
local MyIComponent = oo.class{}
...

-- Criação do componente
...

-- Atualização da faceta IComponent
context:updateFacet("IComponent", MyIComponent())
```

Um exemplo mais detalhado de como criar uma classe LOOP que estenda uma outra classe será dado na Seção 5.1.

3.4 Criação de Receptáculos

Receptáculos representam dependências de interfaces (facetas), e devem ser descritos pelo desenvolvedor da aplicação, não implementados. Eles são manipulados pela faceta básica *IReceptacles*. Se a aplicação desejar manipular seus receptáculos de forma diferente, precisará substituir a implementação da faceta *IReceptacles* através do método *updateFacet* do contexto, como descrito na Seção 3.3.

A criação de receptáculos é análoga e muito parecida com a de facetas, descrita na Seção 3.3.

Para adicionar um receptáculo a um componente, o contexto fornece o método *addReceptacle*, que espera como parâmetros o nome, a interface esperada e um *boolean* indicando se o receptáculo deve aceitar múltiplas conexões ou somente uma. O uso desse método pode ser visto no Código 8.

Código 8: Adição de um Receptáculo MyReceptacle a um Componente

```
1 -- Criação do componente
2 ...
3
4 -- Instanciação e adição de um receptáculo que aceita múltiplas conexões de
5 -- facetas MyFacet ao componente
6 context:addReceptacle("MyReceptacleName", "IDL:mymodule/MyFacet:1.0", true)
```

3.5 Acesso a Facetas e Receptáculos

O contexto fornece métodos para o acesso às suas facetas e receptáculos. Esses métodos retornam uma tabela com metadados sobre a faceta ou receptáculo. Exemplos são fornecidos no Código 9.

A tabela de metadados de uma faceta contém os seguintes campos:

- `name`: Nome da faceta, fornecido pelo usuário. Atua como o identificador único da faceta dentro do componente.

Código 9: Métodos de Acesso a Facetas e Receptáculos

```
-- Criação do componente
...
-- Adição de facetas
...
-- Adição de receptáculos

-- Acesso à tabela de metadados da faceta MyFacet
local facet = context:getFacetByName("MyFacetName")
-- Acesso ao objeto CORBA da faceta MyFacet
local obj = context.MyFacetName
-- O objeto CORBA também pode ser acessado pela tabela de metadados
obj = facet.facet_ref

-- Acesso à tabela de metadados do receptáculo MyReceptacle
local receptacle = context:getReceptacleByName("MyReceptacleName")
```

- `interface_name`: A interface IDL da faceta, fornecida pelo usuário.
- `facet_ref`: O objeto CORBA que representa a faceta, criado pelo método *addFacet*.
- `key`: Chave opcional utilizada como a chave do objeto CORBA no ORB, para a criação de referências persistentes. Este campo somente é preenchido caso uma chave seja fornecida pelo usuário. Caso contrário, o ORB gerará automaticamente uma chave aleatória, mas este campo permanecerá nil.
- `implementation`: Instância da faceta, fornecida pelo usuário. Utilizada para a criação do objeto CORBA que fica armazenado em facet_ref.

Um exemplo de como acessar outras facetas de dentro da implementação de uma faceta pode ser visto no Código 10.

Código 10: Acesso a Outras Facetas de Dentro de Um Método de Faceta

```
...
-- Implementação de um método
function MyFacet:myMethod()
  -- como acessar o contexto da instância de componente ao qual essa
  -- faceta pertence
  local context = self.context
  -- como acessar e usar outras facetas da mesma instância de componente
  local anotherFacet = context.AnotherFacet
  anotherFacet:anotherMethod()
end
...
```

A tabela de metadados de um receptáculo contém os seguintes campos:

- `name`: Nome do receptáculo. Atua como o identificador único do receptáculo dentro do componente.
- `interface_name`: A interface IDL esperada pelo receptáculo.
- `is_multiplex`: *Boolean* indicando se o receptáculo aceita múltiplas conexões.
- `connections`: Lista de conexões realizadas nesse receptáculo.

4 Exemplo Completo

Demonstraremos aqui o uso mais simples de um componente: apenas uma faceta além das três facetas básicas. Não será criado nenhum receptáculo, apesar da existência da faceta *IReceptacles*. Esta demonstração será baseada na *demo Hello*, e exemplos mais complexos poderão ser encontrados nas outras *demos* do projeto.

O componente *Hello* oferece quatro interfaces: *IComponent*, *IReceptacles*, *IMetaInterface* e apenas uma interface própria, de nome *IHello*. Sua IDL está disponível no Código 11.

Código 11: IDL do Componente Hello

```
1 module scs{
2   module demos{
3     module helloworld {
4       interface Hello {
5         void sayHello();
6       };
7     };
8   };
9 };
```

O Código 12 implementa a faceta *IHello*, que conta com apenas um método, *sayHello*. Além disso, realiza a criação do componente. O código é bastante similar ao apresentado na Seção 3.

Por fim, temos o código "cliente", que acessa o componente. Note que esse código pode ser CORBA puro, não é necessária a criação de um componente para acessar outro componente. Um exemplo desse tipo de código pode ser visto no Código 13.

Neste exemplo, a mensagem "Hello User!"será exibida somente na máquina servidor. O código cliente apenas terá a chamada *sayHello()* completada corretamente e será finalizado sem erros.

5 Elementos Adicionais da API do SCS

As seções anteriores descreveram o uso mais comum do SCS para o desenvolvimento de aplicações baseadas em componentes. No entanto, alguns tópicos e funcionalidades adicionais merecem destaque. Nesta seção descreveremos os mais importantes, que podem ser necessários em aplicações ligeiramente mais complexas que o código apresentado anteriormente.

5.1 Extensão de Classes no LOOP

Como mencionado na Seção 3.3, facetas SCS geralmente são implementadas como classes LOOP. Além disso, a representação local do componente também é uma classe, a classe *ComponentContext*. Com o uso do paradigma de orientação a objetos, pode ser necessário estender algumas dessas classes.

5.1.1 Extensão do Contexto

Em particular, o contexto pode ser usado para guardar o estado do componente como um todo, armazenando informações que sejam úteis para mais de uma faceta. A classe *ComponentContext* já faz isso, guardando metadados sobre as facetas e receptáculos. Se o usuário desejar inserir novos dados nessa classe, o ideal é estendê-la. É importante notar que o construtor deve receber todos os parâmetros que o construtor da classe *ComponentContext* recebe, para que possa repassá-los. A documentação do LOOP provê informações mais completas, mas o Código 14 mostra como estender a classe *ComponentContext* com herança simples.

5.1.2 Extensão de Facetas

Além do exemplo do contexto, é comum também encontrarmos a necessidade de estender classes que implementam facetas. Por exemplo, a classe *Component*, que implementa a faceta *IComponent*, contém métodos para gerenciar o ciclo de vida do componente, chamados *startup* e *shutdown*. Como a lógica desses métodos deve ficar a cargo do desenvolvedor da aplicação, suas implementações não fazem nada. Eles precisam ser sobrescritos com uma nova implementação. Lua permite que simplesmente mudemos a função no objeto diretamente, mas o Código 15 segue no uso do paradigma de orientação de objetos, estendendo a classe e sobrescrevendo o método, para exemplificar o procedimento.

5.2 Builders

Em todos os exemplos anteriores, a definição e "montagem"do componente (adição de facetas e receptáculos) é feita dentro do código fonte. Isso significa que, caso seja necessária alguma mudança nessa configuração, o código-fonte precisa ser alterado. Lua é uma linguagem interpretada, mas o código pode ser pré-compilado e nesse caso uma alteração desse tipo levaria a uma recompilação. É fácil perceber que essa configuração do componente pode ser definida externamente, permitindo alterações sem a necessidade de mudanças no código-fonte.

Além disso, serviços de mais alto nível podem se beneficiar de descrições em uma linguagem declarativa qualquer, para realizar a implantação automática de componentes num domínio. Administradores de sistema, sem um conhecimento maior sobre o desenvolvimento de componentes de *software*, podem alterar a configuração de aplicações sem a necessidade da intervenção de um programador.

Para facilitar esse processo de externalização da configuração do componente, o SCS utiliza o conceito de *builders*. *Builders* são pequenas bibliotecas que lêem uma descrição de um componente em uma linguagem específica e então interpretam os dados para criar um componente de acordo com a configuração desejada. O SCS-Lua já fornece um *builder* para a linguagem XML.

5.2.1 XMLComponentBuilder

O *XMLComponentBuilder* interpreta um arquivo XML com a descrição de um componente e retorna um componente pronto com a configuração especificada nesse arquivo. Na versão atual não é possível especificar parâmetros para os construtores das facetas. É possível especificar facetas, receptáculos, o Identificador do Componente e a implementação do contexto a ser usada. O Código 16 mostra um XML de exemplo, enquanto que o Código 17 demonstra como utilizar o *XMLComponentBuilder*.

Para obter a implementação de facetas e contexto, o *XMLComponentBuilder* realiza um *require* no valor fornecido na *tag* respectiva. Se o nome de uma faceta já existir, a faceta anterior será substituída pela nova.

O SCS fornece em seu pacote de distribuição um arquivo chamado *ComponentDescription.xsd* que contém o *schema* XML utilizado pelo XMLComponentBuilder em qualquer linguagem suportada pelo SCS. A versão Lua atual, no entanto, não verifica o XML fornecido pelo usuário contra o *schema*.

Código 12: Criação do Componente Hello

```lua
local oo  = require "loop.base"
local oil = require "oil"
local ComponentContext = require "scs.core.ComponentContext"

-- inicialização do ORB
-- porta e host apenas para fins do exemplo
local orb = oil.init({host = "localhost", port = 1050})

-- carga das IDLs no ORB
orb:loadidlfile("scs.idl")
orb:loadidlfile("hello.idl")

-- implementação da faceta IHello
local Hello = oo.class{name = "World"}
function Hello:sayHello()
  print("Hello "..self.name.."!")
end

-- função main
oil.main(function()
  -- instrução ao ORB para que aguarde por chamadas remotas (em uma nova "thread")
  oil.newthread(orb.run, orb)

  -- criação do ComponentId
  local cpId = {
    name = "Hello",
    major_version = 1,
    minor_version = 0,
    patch_version = 0,
    platform_spec = "lua"
  }

  -- cria o componente
  local instance = ComponentContext(orb, cpId)

  -- adiciona a faceta Hello
  instance:addFacet("IHello", "IDL:scs/demos/helloworld/IHello:1.0", Hello())

  -- modificação do nome a ser exibido na mensagem da faceta Hello
  instance.IHello.name = "User"

  -- publicação do IOR para que a faceta IHello do componente possa ser
  -- encontrada. Observação: podemos exportar qualquer faceta, pois temos
  -- o método _component para obter a faceta IComponent e, com ela,
  -- pode-se obter outras facetas (esse passo pode ser substituído por outras
  -- formas de publicação, como a publicação em um serviço de nomes, por
  -- exemplo).
  oil.writeto("hello.ior", orb:tostring(instance.IHello))
end)
```

Código 13: Utilização do Componente Hello

```lua
local oil = require "oil"

-- inicialização do ORB
local orb = oil.init()

-- carga das IDLs no ORB
orb:loadidlfile("scs.idl")
orb:loadidlfile("hello.idl")

-- função main
oil.main(function()
  -- assume-se que o arquivo que contém o IOR (publicado pelo código
  -- anterior) esteja disponível. O arquivo pode ter sido criado em
  -- outra máquina e, nesse caso, tem de ser copiado manualmente
  -- (pode-se também utilizar um método diferente de publicação,
  -- como um serviço de nomes).
  local iHelloIOR = oil.readfrom("hello.ior")

  -- obtenção da faceta IHello e IComponent
  local iHelloFacet = orb:newproxy(iHelloIOR, "synchronous",
    "IDL:scs/demos/helloworld/IHello:1.0")
  -- obtenção da faceta principal do componente, IComponent
  local icFacet = iHelloFacet:_component()
  -- precisamos utilizar o método narrow pois estamos recebendo um
  -- org.omg.CORBA.Object
  icFacet = orb:narrow(icFacet)

  -- inicialização do componente (opcional, não fará nada nesse caso pois não
  -- modificamos o método startup no servidor. O método padrão não faz nada)
  icFacet:startup()

  -- com o componente inicializado, podemos utilizá-lo à vontade.
  -- note que não é possível modificar o campo "name" da classe Hello
  -- remotamente, pois o campo não está definido em IDL (nem há um
  -- método "setter").
  iHelloFacet:sayHello()
end)
```

Código 14: Extensão do Contexto

```
local ComponentContext    = require "scs.core.ComponentContext"

-- note que ao invés de utilizar loop.base, utilizamos loop.simple, para herança
-- simples. Para o suporte a herança múltipla, consulte o manual do LOOP.
local oo = require "loop.simple"

-- carga do orb e das IDLs
...

-- a linha abaixo especifica a classe ComponentContext como classe a ser herdada
local MyComponentContext = oop.class({}, ComponentContext)
function MyComponentContext:__init(orb, componentId, basicKeys)
  -- chama o construtor da superclasse passando o self da classe atual
  self = ComponentContext.__init(self, orb, componentId, basicKeys)
  -- não podemos juntar a linha abaixo com a linha acima - a definição de "self"
  -- na linha acima não é uma definição de variável local temporária
  return self
end

-- definição de um método da classe MyComponentContext
function MyComponentContext:aMethod()
  -- é possível chamar métodos da superclasse ComponentContext pelo self
  local componentId = self:getComponentId()
  ...
end

-- criação do componente
...
local instance = MyComponentContext(orb, componentId)
```

Código 15: Extensão da Faceta IComponent

```
local Component    = require "scs.core.Component"

-- note que ao invés de utilizar loop.base, utilizamos loop.simple, para herança
-- simples. Para o suporte a herança múltipla, consulte o manual do LOOP.
local oo = require "loop.simple"

-- carga do orb, das IDLs e outros requires
...

-- a linha abaixo especifica a classe Component como classe a ser herdada
local MyIComponent = oop.class({}, Component)
function MyIComponent:__init()
  -- chama o construtor da superclasse passando o self da classe atual
  self = Component.__init(self)
  -- não podemos juntar a linha abaixo com a linha acima - a definição de "self"
  -- na linha acima não é uma definição de variável local temporária
  return self
end

-- sobrescrevendo o método startup
function MyIComponent:startup()
  ...
end

-- criação do componente
...
-- O instância de ComponentContext já vem com a faceta IComponent padrão
local instance = ComponentContext(orb, componentId)
-- substituição da faceta IComponent padrão pela nova implementação
instance:updateFacet("IComponent", MyIComponent())
```

Código 16: Arquivo XML Definindo um Componente

```xml
<?xml version="1.0" encoding="iso-8859-1" ?>
<component xmlns="tecgraf.scs">
  <id>
    <name>ExemploArquivoXML</name>
    <version>1.0.0</version>
    <platformSpec>Lua</platformSpec>
  </id>
  <context>
    <type>MyComponentContext</type>
  </context>
  <facets>
    <facet>
      <name>MyFacetName</name>
      <interfaceName>IDL:mymodule/MyFacet:1.0</interfaceName>
      <facetImpl>MyFacet</facetImpl>
    </facet>
    <facet>
      <name>AnotherFacet</name>
      <interfaceName>IDL:mymodule/AnotherFacet:1.0</interfaceName>
      <facetImpl>AnotherFacet</facetImpl>
    </facet>
  </facets>
  <receptacles>
    <receptacle>
      <name>MyReceptacleName</name>
      <interfaceName>IDL:mymodule/MyFacet:1.0</interfaceName>
      <isMultiplex>true</isMultiplex>
    </receptacle>
  </receptacles>
</component>
```

Código 17: Exemplo de uso do XMLComponentBuilder

```lua
local builder = require "scs.core.builder.XMLComponentBuilder"

-- carga do orb, das IDLs e outros requires
...

-- criação do componente. 'instance' será um MyComponentContext, já com todas as
-- facetas e receptáculos especificados no arquivo XML
local instance = builder:build(orb, "exemplo.xml")
```