# 1 Manual de Utilização

## 1.1 Instalação

O SCS apresenta alguns pré-requisitos e exige uma pequena configuração do sistema antes de poder ser utilizado.

### 1.1.1 Pré-Requisitos

Para instalar o SCS, são necessários os seguintes passos:

a) Instalar a linguagem Lua, versão 5.1 ( http://www.lua.org )
b) Instalar o ORB OiL (ORB in Lua), versão 0.4 beta ( http://oil.luaforge.net/ )

### 1.1.2 Instalação do SCS

A instalação do SCS em sí é bastante simples, e composta de apenas 2 passos:

a) Descompactar o pacote em um diretório de sua preferência
b) Assumindo o diretório escolhido como <install-dir>, adicionar à variável de ambiente LUA_PATH os caminhos:
    - < install-dir>/src/lua/?.lua
    - < install-dir>/src/lua/scs/container/?.lua

## 1.2 Escrevendo um Componente SCS

Para criar um novo componente que funcione na arquitetura SCS, basta seguir o passo-a-passo:

### 1.2.1 Carregando o Modelo de Componentes SCS

Para ter acesso à API do SCS, basta executar o comando "require", passando como parâmetro o módulo a ser carregado, "scs.core.base". Exemplo:

local scs = require "scs.core.base"

### 1.2.2 Definindo Classes no Modelo LOOP

Para definir uma classe LOOP, primeiro é necessário obter acesso à sua API de classes, da seguinte forma:

local oo = require "loop.base"

Neste exemplo utilizaremos o modelo de classes básico do LOOP. Para instruções sobre como utilizar modelos mais complexos, vá até http://loop.luaforge.net .

Com acesso à API do modelo de classes, podemos facilmente definir classes da seguinte forma:

```
1  local oo = require "loop.base"
2  local MyClass = oo.class( { myProperty = propValue } )
```

A tabela passada como parâmetro será a base da classe. Agora, podemos adicionar métodos:

```
1  function MyClass:myMethod()
2      -- do something here
3  end
```

Note que desta forma definimos valores padrão para a classe, que serão compartilhados por todas as instâncias. Sempre que uma instância da classe modificar o valor de um tipo básico, este novo valor será armazenado apenas nesta instância, e não afetará outras instâncias existentes. No entanto, caso o dado não seja um tipo básico, como uma tabela ou um *userdata*, devemos ter extremo cuidado.

Usaremos tabelas como exemplo. Podemos apenas modificar ou criar um novo campo dentro desta tabela, ao invés de atribuir um novo valor para a própria. Desta forma uma cópia não será criada, e a tabela da classe será modificada. Isto influenciará todas as instâncias. Exemplo de código com erro:

```
1  local oo = require "loop.base"
2  local MyClass = oo.class( { myTable = { myField = 2 } } )
3  MyClass.myTable.myField = 3
```

Para evitar este tipo de comportamento, devemos explicitamente atribuir uma nova tabela ao criar uma nova instância. Podemos garantir isto através do metamétodo __init(), que funciona como um construtor:

```
1  local oo = require "loop.base"
2  local MyClass = oo.class( { myTable = {}, myNumber = 1 } )
3  function MyClass:__init()
4      -- makes sure that table values are recreated for the instance.
5      -- if we do not take this care, base class values may be modified.
6      -- the line below creates a new instance of the class
7      local inst = oo.rawnew(self, {})
8      -- the line below IS NOT necessary
9      inst.myNumber = 1
10     -- the line below IS necessary if your code will use myTable
11     inst.myTable = {}
12     -- the new instance will contain it's own myTable field
13     return inst
14 end
```

Dentro dos métodos, podemos acessar os campos internos à classe através da variável “self”:

```
1  local oo = require "loop.base"
2  local MyClass = oo.class( { myTable = {}, myNumber = 1 } )
3  function MyClass:myPrintMethod()
4      print(self.myNumber)
5  end
```

### 1.2.3 Definindo Componentes LOOP-SCS

Pré-Requisitos:

a) Criar um novo arquivo Lua
b) Separar uma seção inicial para se realizar os “require” e guardar quaisquer variáveis globais necessárias
c) Criar um módulo

Exemplo:

```
1   local oil    = require "oil"
2   local oo     = require "loop.base"
3   local comp   = require "loop.component.base"
4   local port   = require "loop.component.base"
5   local scs    = require "scs.core.base"
6
7   local dofile = dofile
8   local string = string
9   local assert = assert
10  local print  = print
11
12  module "mymodule"
```

Como relatado anteriormente, o SCS se baseia no LOOP para criar classes e componentes. No entanto, o conjunto de funcionalidades que utilizamos da API de componentes do LOOP dá suporte apenas à criação de facetas e receptáculos, e espera um conjunto de classes que juntará em uma “caixa”.

A API do SCS provê uma forma de se criar um componente SCS, com tudo o que for necessário. Para isto, ela pede como entrada uma fábrica LOOP, e as descrições de todas as facetas e receptáculos.

Para definir uma fábrica loop, são necessários dois passos:

a) Definir um molde, onde se explicitam as portas do componente, e seus tipos. Uma porta pode ser uma faceta ou um receptáculo. As opções são:

    1. Facet - Faceta.

2. Receptacle – Receptáculo simples, suporta apenas uma conexão.
3. ListReceptacle – Receptáculo para mútilplas conexões, em forma de lista.
4. HashReceptacle – Receptáculo para múltiplas conexões, em forma de tabela *hash*.
5. SetReceptacle – Receptáculo para múltiplas conexões, em forma de conjunto.

Não é necessário saber muito mais que isto sobre os tipos de receptáculo, pois o tratamento é feito internamente pela arquitetura do SCS. No entanto, caso queira se aprofundar, consulte o manual LOOP em seu site.

Um exemplo de molde seria:

```
1  local comp   = require "loop.component.base"
2  local port   = require "loop.component.base"
3  local MyTemplate = comp.Template ( {
4      IComponent      = port.Facet,
5      IReceptacles    = port.Facet,
6      IMetaInterface  = port.Facet,
7      MyFacet         = port.Facet,
8      MyReceptacle    = port.Receptacle,
9  } )
```

b) Definir uma fábrica LOOP, através do molde e de um construtor. O construtor nada mais é que uma tabela onde se atribuem as implementações das facetas a nomes equivalentes aos nomes das portas. O primeiro item desta tabela servirá de base para o componente, então é uma boa idéia colocar uma nova tabela ou classe neste ponto. Esta tabela base poderá conter variáveis globais do componente, que serão visíveis apenas por suas classes internas. Segue um exemplo:

```lua
local oo      = require "loop.base"
local comp    = require "loop.component.base"
local port    = require "loop.component.base"

local MyBase = oo.class( { myTable = {}, myNumber = 1 } )
function MyBase:myPrintMethod()
    print(self.myNumber)
end

local MyFacetImplementation = oo.class( { myProperty = propValue } )
function MyFacetImplementation:myMethod()
    -- do something here
end

local MyTemplate = comp.Template ( {
    IComponent      = port.Facet,
    IReceptacles    = port.Facet,
    IMetaInterface  = port.Facet,
    MyFacet         = port.Facet,
    MyReceptacle    = port.Receptacle,
} )

local MyFactory = MyTemplate ( {
    MyBase,
    IComponent      = scs.Component,
    IReceptacles    = scs.Receptacles,
    IMetaInterface  = scs.MetaInterface,
    MyFacet         = MyFacetImplementation,
} )
```

Aqui, inclui-se uma observação. Devido ao funcionamento interno do pacote LOOP, caso o desenvolvedor deseje que uma classe tenha acesso à tabela base do componente, deverá criar na classe um campo “context”, com qualquer valor diferente de *nil*. Desta forma, o LOOP se encarregará de transformar este campo “context” numa referência à tabela base, que poderá ser acessada de dentro da classe através do uso de “self.context”. A função de criação de componentes da API SCS (newComponent() no módulo Lua scs.core.base), que veremos mais à frente, já se encarrega de realizar este processo para todas as classes de facetas fornecidas. Assim, dentro do método myMethod() do exemplo acima, poderíamos acessar myProperty da seguinte forma:

self.myProperty

E, no mesmo método, poderíamos acessar myNumber (da classe MyBase que serve de base para o componente como um todo) da seguinte forma:

self.context.myNumber

Criada a fábrica, precisamos definir as descrições de facetas e receptáculos, seguindo os arquivos de definição IDL utilizados. Exemplo:

```
1  local descriptions = {}
2  descriptions.IComponent = { name = "IComponent",
3  interface_name = "IDL:scs/core/IComponent:1.0" }
4  descriptions.MyClassInterface = { name = "MyClassInterface",
5  interface_name = "IDL:mymodule/MyClassInterface:MyVersion" }
```

Para que o componente possa ser carregado pelo contêiner de componentes Lua, devemos ainda devolver uma fábrica SCS no final de nosso arquivo do componente. Esta nada mais é do que uma função que usa a API SCS para criar uma nova instância do componente, e retorna esta instância. Veja:

```
1  local MyFactory = ...        -- same as previous example
2  local descriptions = ...     -- same as previous example
3  local SCSFactory = oo.class{ factory = MyFactory,
4                               descriptions = descriptions }
5  function SCSFactory:create( args )
6      local instance = scs.newComponent( self.factory, self.descriptions )
7      -- args may be a table with "command line" arguments.
8      -- It's not mandatory. In this case, we assume only one,
9      -- which is the name. If the argument is not nil, it'll be used.
10     -- If it is, then the name will remain "Bob", which is the
11     -- class default value.
12     instance.componentName = arg[1] or instance.componentName
13     return instance.IComponent
14 end
15 return SCSFactory
```

Como o método create() da classe SCSFactory atua como um construtor para o componente como um todo, pode-se receber parâmetros de entrada neste ponto se

necessário (representados no exemplo pela tabela args). O método create() da fábrica deve retornar a faceta IComponent da nova instância.

Finalmente o componente está pronto. O passo-a-passo pode parecer trabalhoso inicialmente, mas a dificuldade de entendimento é realmente pequena e deve desaparecer após o desenvolvimento de um ou dois componentes.

### 1.2.4 Redefinindo Métodos

Na arquitetura SCS Lua, qualquer método interno pode ser redefinido. O mais comum é redefinir os métodos startup() e shutdown() da faceta IComponent, que inicialmente não fazem nada. Redefinir um método é extremamente simples. Basta criar a nova implementação em uma função qualquer e, dentro do código da fábrica, trocar o método da nova instância pelo novo. Por exemplo:

```
function newImplementation()

-- do something here

end

local SCSFactory = ...

function SCSFactory:create( args )

        local instance = scs.newComponent( self.factory, self.descriptions )

        instance.myMethod = newImplementation

end
```

## 1.3 Implantação

A implantação da arquitetura do SCS-Lua constitui-se, no mínimo, de um nó de execução. Este pode ser utilizado em conjunto a um contêiner de componentes Lua e um repositório de componentes Lua, e/ou a um contêiner de componentes Java. Apesar de todos serem componentes SCS, a carga do nó de execução e do contêiner diferem

um pouco da dos demais componentes. Uma explicação mais detalhada das versões Lua será apresentada a seguir.

### 1.3.1 Nó de Execução

Para executar este componente, vá até <install-dir>/src/lua/scs/execution_node e execute o arquivo ExecutionNode.lua. Se houver um arquivo de inicialização no diretório (definido no arquivo de propriedades Properties.txt), este será executado logo em seguida, na inicialização do nó de execução. Este arquivo serve para realizar tarefas iniciais, geralmente de configuração, e pode-se modificá-lo caso seja necessário. Além disso, deve-se configurar o arquivo Properties.txt com a plataforma correta.

Este é o único componente que deve ser iniciado de fora do sistema. Apenas um nó de execução é necessário por máquina, apesar disto não ser uma regra. Ao terminar sua inicialização, será gerado um arquivo execution_node.ior no diretório, que poderá ser usado como referência por outro componente.

### 1.3.2 Contêiner de Componentes

Para carregar este componente, é necessário que um nó de execução esteja em funcionamento. Deve-se obter uma referência para este (através do arquivo execution_node.ior gerado automaticamente na inicialização do nó, por exemplo). De posse desta referência, pode-se iniciar um container de acordo com o seguinte exemplo:

```
local oil = require “oil”

local orb = oil.init()

-- obtaining Execution Node’s IComponent facet

local enCmp = orb:newproxy(assert(oil.readfrom(“execution_node.ior”)))

-- obtaining Execution Node’s Execution Node facet

local en = enCmp:getFacetByName(“ExecutionNode”):_narrow()

-- creating properties that will be passed to the container
```

local containerPropertySeq = { { name = “myProperty”, value = “propValue” } }

-- starting container and obtaining its IComponent reference

local containerCmp = en:startContainer(“MyContainer”, containerPropertySeq)

Se houver um arquivo de inicialização no diretório <install-dir>/src/lua/scs/container (definido no arquivo de propriedades Properties.txt), este será executado logo em seguida, na inicialização do contêiner. Este arquivo serve para realizar tarefas iniciais, geralmente de configuração, e pode-se modificá-lo caso seja necessário. Além disso, deve-se configurar o arquivo Properties.txt com a plataforma correta.

Ao término da chamada startContainer() do nó de execução, será retornada uma referência para o contêiner recém-criado.

### 1.3.3 Repositório de Componentes

A carga de um repositório de componentes ocorre ao efetuar-se uma chamada load() em um contêiner de componentes. Assumindo que um contêiner já esteja em execução, e que uma referência à faceta IComponent deste seja conhecida através de uma variável chamada containerRef, podemos construir o exemplo a seguir:

```
1  -- obtaining Component Container's Component Loader Facet
2  local clFacet = containerRef:getFacetByName("ComponentLoader"):_narrow()
3  -- loading Component Repository. Will receive it's handle
4  local componentId = { name = "ComponentRepository", version = 1.0 }
5  local cpnRepHandle = clFacet:load(componentId, {})
6  -- obtaining the IComponent facet
7  local repositoryCmp = cpnRepHandle.cmp
```

Se houver um arquivo de inicialização no diretório <install-dir>/src/lua/scs/repository (definido no arquivo de propriedades Properties.txt), este será executado logo em seguida, na inicialização do repositório. Este arquivo serve para realizar tarefas iniciais, geralmente de configuração, e pode-se modificá-lo caso seja necessário.

Ao término da chamada load() do contêiner, será retornada uma referência para o repositório recém-criado. Neste ponto, provavelmente será desejável conectar este repositório ao contêiner, o que pode ser feito da seguinte forma:

```
1  -- connecting component repository to container's receptacle
2  local recepFacet = containerRef:getFacetByName("IReceptacles"):_narrow()
3  local connectionId = recepFacet:connect("ComponentRepository", repositoryCmp)
```

### 1.3.4 Demais Componentes

A carga de qualquer outro componente SCS ocorre de forma similar à carga do repositório. Assumindo que um contêiner e um repositório munido do componente já estejam em execução e conectados, e que uma referência ao contêiner já seja conhecida através de uma variável chamada containerRef, podemos construir o exemplo a seguir:

```
1  -- obtaining Component Container's Component Loader Facet
2  local clFacet = containerRef:getFacetByName("ComponentLoader"):_narrow()
3  -- creating arguments table
4  local myArgsSeq = {}
5  table.insert(myArgsSeq, { name = "myArgument" , value = "argValue" })
6  -- loading Component MyComponent. Will receive it's handle
7  local componentId = { name = "MyComponent", version = 1.0 }
8  local cpnHandle = clFacet:load(componentId, myArgsSeq)
9  -- obtaining the IComponent facet
10 local myComponentCmp = cpnHandle.cmp
```

Ao término da chamada load() do contêiner, será retornada uma referência para o componente recém-criado.

## 1.4 Exemplo – PingPong

Um exemplo completo de componente pode ser visto em <install-dir>/src/lua/scs/demos/pingpong/PingPong.lua .

Componentes PingPong nada mais fazem que chamar ping() e pong() um no outro. Deve-se instanciar dois desses componentes, conectá-los, iniciá-los com startup() e chamar o método start() em um deles (no caso da implementação Java, este método deve ser chamado em todos).

Três exemplos completos do uso da infra-estrutura SCS para a execução de componentes PingPong podem ser encontrados em <install-dir>/src/lua/scs/demos/pingpong . Todos utilizam um nó de execução Lua, mas contam com configurações diferentes:

- pingpongluaconfig.lua: Utiliza um contêiner Lua para carregar dois PingPong's Lua.
- pingpongjavaconfig.lua: Utiliza um contêiner Java para carregar dois PingPong's Java.
- pingpongluajavaconfig.lua: Utiliza um contêiner Lua e um Java para carregar um PingPong em cada.

Para executar um destes exemplos, execute primeiro o nó de execução Lua.