3
ASCS

O ASCS (Annotated Software Components System) é o modelo de programação Java, desenvolvido neste trabalho, para a infra-estrutura SCS. Este novo modelo foi desenvolvido para auxiliar na investigação da adoção da técnica de programação orientada a atributos juntamente com uma linguagem orientada a objetos para construção de aplicações baseadas em componentes.

Através de uma API baseada em anotações [10], o ASCS permite que o usuário, que no nosso caso é o desenvolvedor de componentes, concentre as informações relativas ao modelo de componentes e ao middleware na camada de anotações. Desta forma, o código funcional pode ficar concentrado na camada de código orientado a objetos do componente.

Esta separação de conceitos no código, proporcionada pela API baseada em anotações, permite uma maior modularização no desenvolvimento de um componente. A confecção do código funcional e das anotações podem ser feitas em momentos diferentes e de forma independente. Ou seja, o programador que escreve o código funcional do componente não precisa conhecer detalhes do middleware ou da API de anotações.

O ASCS é equipado com um mecanismo de validação em tempo de compilação do código de implementação do componente. Com este mecanismo de validação é possível identificar de forma antecipada (no momento da compilação do código) possíveis erros estruturais do componente. Assim, o tempo gasto com a identificação e correção de erros estruturais é reduzido.

O objetivo deste capítulo é apresentar o modelo de componentes utilizado pelo ASCS, que é uma extensão do modelo de componentes original do SCS, assim como seu modelo de programação baseado em anotações. No final deste capítulo, é apresentado um passo-a-passo de como construir um componente utilizando o modelo de programação ASCS e como utilizá-lo em uma aplicação.

3.1
Modelo de Componentes Estendido

O ASCS utiliza uma extensão do modelo de componentes original do SCS apresentado no capítulo anterior. Além da faceta nativa *IComponent*, o

modelo de componentes do ASCS possui outras duas facetas nativas: *Properties* e *LyfeCycle*. Estas novas facetas permitem, respectivamente, o cadastro de propriedades para um componente, e o controle do ciclo de vida do componente. A figura 3.1 ilustra as entidades do modelo de componentes ASCS.

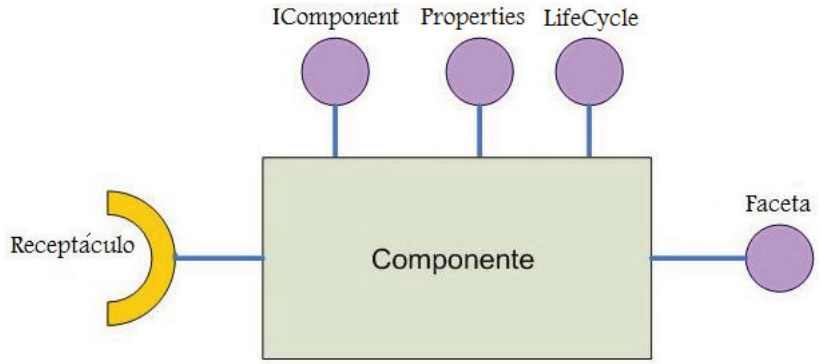

Figura 3.1: Modelo de Componentes ASCS

Além das novas facetas, o contêiner SCS também foi estendido através do desenvolvimento de uma fábrica genérica de componentes. Esta seção detalha as novas facetas assim como esta nova fábrica de componentes.

3.1.1 Contêiner Estendido

Como vimos no capítulo anterior, juntamente com o componente, o desenvolvedor precisa criar uma fábrica responsável pela instanciação do componente. Esta fábrica é um dos conceitos que fazem parte do contêiner SCS.

No ASCS, o contêiner foi estendido através do desenvolvimento de uma fábrica genérica de componentes. Esta fábrica genérica tem a capacidade de instanciar qualquer componente ASCS, eliminando assim a necessidade de o desenvolvedor criar uma fábrica específica para cada componente. Além disso, as informações sobre as facetas e receptáculos que compõem o componente, desta forma, ficam concentradas no próprio código do componente, o que facilita o desenvolvimento e melhora a legibilidade do código.

Nesta fábrica é realizado um processamento baseado em reflexão computacional, percorrendo todas as anotações suportadas pelo ASCS presentes no componente e realizando um tratamento específico para cada uma delas. A fábrica genérica irá então, por exemplo, para cada anotação @Facet e @Receptacle presentes no componente, criar internamente os objetos SCS referentes às facetas e às descrições dos receptáculos, e por fim, criar o contexto contendo o objeto SCS que representa a instância do componente.

A existência de uma fábrica genérica não impede que o desenvolvedor crie uma fábrica específica para instanciar seu componente. Em casos particulares, o desenvolvedor pode precisar acrescentar processamentos específicos na criação do componente. Para fazer isso, o desenvolvedor pode criar a sua fábrica desde que estenda a fábrica genérica de componentes ASCS ou atribua processamentos para todas as anotações ASCS utilizadas em seu componente.

3.1.2 LifeCycle

A faceta *LifeCycle* foi criada para fornecer ao desenvolvedor do componente um controle em relação ao ciclo de vida do componente. Através desta faceta, o desenvolvedor pode, opcionalmente, interferir nas etapas de inicialização, carga, descarga, e desligamento do componente.

Para a etapa de inicialização, o desenvolvedor do componente poderá escrever um trecho de código que será executado no momento em que o componente for inicializado. Através deste controle, o desenvolvedor pode, por exemplo, inicializar algum tipo de interface gráfica, ou iniciar a execução de um serviço. O código de controle referente à etapa de inicialização será automaticamente executado em uma *thread* distinta.

A etapa de carga consiste no momento em que o componente é carregado no contêiner. Com o controle sobre a etapa de carga do compnente, o desenvolvedor pode ter acesso a valores atribuídos no momento de implantação do componente, e então, realizar um processamento específico, como por exemplo, inicializar o valor de um campo do componente que será atribuído apenas no momento de implantação.

As etapas de desligamento e descarga do componente são análogas às etapas de inicialização e carga, respectivamente.

As operações referentes a controle de ciclo de vida, hoje, ainda não são implementadas através de uma faceta SCS de fato. Ou seja, não existe uma interface definida em um arquivo IDL para agregar as operações de ciclo de vida. Por conta da arquitetura atual do SCS, as operações de ciclo de vida estão divididas nas interfaces *IComponent* e *ComponentLoader*. Porém, consideramos a existência da faceta conceitual *LifeCycle* já que estas operações, mesmo estando em interfaces distintas, estão ligadas ao conceito de controle de ciclo de vida do componente.

3.1.3 Properties

A faceta *Properties* permite que pares chave-valor sejam cadastrados para um determinado componente. O valor de uma propriedade normalmente é atribuído pelo desenvolvedor da aplicação, e consumido na implementação do próprio componente.

As operações da faceta *Properties* estão definidas na interface IDL mostrada na figura 3.1. Através das operações *setProperty()* e *getProperty()*, o desenvolvedor da aplicação pode atribuir e consultar valores de propriedades de um componente. A operação *getKeys()* retorna uma coleção das chaves de todas as propriedades cadastradas de um componente,

Código 3.1: Properties - Interface IDL

```
1 interface ComponentProperties {
2   void setProperty (in string key, in string value);
3   string getProperty (in string key);
4   string_seq getkeys ();
5 };
```

Através de uma propriedade referente à faceta *Properties*, um componente pode fazer referência a um campo cujo valor poderá ser atribuído apenas após a carga ou durante a execução do componente.

3.2 Modelo de Programação

O modelo de programação do ASCS é composto por anotações Java que são utilizadas para relacionar elementos do código do componente com conceitos do modelo de componentes SCS. As anotações do modelo de programação do ASCS foram definidas de forma que cada uma delas represente de forma clara e direta um conceito presente no modelo de componentes.

A seguir, as anotações que compõem o modelo de programação ASCS são detalhadas. Para cada uma delas é apresentado um exemplo simples de uso.

3.2.1 @FacetSet

Esta anotação é utilizada para definir o conjunto de facetas de um componente SCS. No caso do ASCS, uma faceta é representada por uma interface Java. A classe marcada com a anotação @FacetSet precisa ser concreta e disponibilizar um construtor padrão (sem argumentos) para que ela possa ser instanciada de forma reflexiva pelo núcleo do ASCS.

Para que as facetas possam ser criadas em tempo de execução, as classes geradas a partir do arquivo IDL referentes ao tipo da faceta devem estar

acessíveis no *class-path*. Este passo de compilação do arquivo IDL é detalhado na seção 4.4 deste capítulo.

Esta anotação é composta por anotações do tipo *@Facet*.

3.2.2 @Facet

Esta anotação é utilizada para descrever cada faceta do componente e deve ser configurada com as seguintes propriedades:

* name - É o nome dado a faceta e que o desenvolvedor da aplicação usa para procurar por uma faceta de um componente através da API de introspecção.

* type - Interface Java que define o tipo da faceta. A classe que representa o componente SCS deve obrigatoriamente implementar todas as interfaces Java definidas como suas facetas. O tipo da faceta obrigatoriamente deve ser do tipo MyFacetOperations onde MyFacet é uma interface definida no arquivo IDL correspondente.

O trecho de código abaixo exemplifica o uso das anotações @FacetSet e @Facet.

Código 3.2: @FacetSet - Exemplo de uso

```
1 /**
2  * Componente que implementa o serviço de exemplo
3  * retornando "Hello World!".
4  */
5 @FacetSet({
6   @Facet(name="HelloFacet", type=HelloOperations.class)
7 })
8 public class HelloComponent implements HelloOperations {
9
10   @Override
11   public String sayHello() {
12     return "Hello World!";
13   }
14 }
```

3.2.3 @Receptacle

Esta anotação é utilizada para marcar um campo do componente como um receptáculo. Um receptáculo é uma referência externa para a faceta de um outro componente. Um receptáculo precisa ter seu tipo compatível com o tipo da faceta que será posteriormente conectada a ele. A definição de métodos *setter* e *getter* públicos na classe em que está definido o receptáculo

é imprescindível para que a referência da faceta a ser conectada possa ser atribuída ou obtida em tempo de execução. A classe que define receptáculos precisa disponibilizar um construtor padrão (sem argumentos) para que ela possa ser instanciada em tempo de execução pelo núcleo do ASCS.

De forma similar às facetas, a anotação *@Receptacle*, possui o atributo "name"com o nome do receptáculo. É com este nome que o desenvolvedor da aplicação realiza a amarração de um receptáculo com uma determinada faceta daquele mesmo tipo.

O trecho de código abaixo exemplifica o uso da anotação @Receptacle.

Código 3.3: @Receptacle - Exemplo de uso

```
1 /**
2  * Componente que implementa o serviço de exemplo
3  * consultando a faceta do componente HelloComponent
4  * e imprimindo HelloWorld! no console.
5  */
6 @FacetSet({
7   @Facet(name="PrinterFacet", type=HelloPrinterOperations.class)
8 })
9 public class HelloPrinterComponent implements HelloPrinterOperations {
10
11   @Receptacle(name="HelloReceptacle")
12   private HelloOperations hello;
13
14   @Override
15   public void printHello() {
16     System.out.println(hello.sayHello());
17   }
18
19   public void setHello(HelloOperations hello) {
20     this.hello = hello;
21   }
22
23   public HelloOperations getHello() {
24     return this.hello;
25   }
26
27 }
```

Um receptáculo multiplex pode receber múltiplas conexões. Um receptáculo é definido como multiplex simplesmente declarando-o como um vetor. Desta forma, o núcleo do ASCS irá considerar automaticamente que o desenvolvedor da aplicação poderá conectar várias facetas neste mesmo receptáculo.

O desenvolvedor do componente pode assumir que a instância do vetor que representa o receptáculo multiplex será automaticamente atribuída, e então iterar normalmente sobre o vetor. Sempre que uma nova conexão for realizada, uma nova instância do vetor é criada pelo núcleo ASCS, com o tamanho atualizado.

O trecho de código abaixo exemplifica o uso de receptáculos multiplex.

Código 3.4: @Receptacle (Multiplex)

```
1 /**
2  * Componente que implementa o serviço de exemplo
3  * consultando a faceta do componente HelloComponent
4  * e imprimindo HelloWorld! no console.
5  */
6 @FacetSet({
7   @Facet(name="PrinterFacet", type=HelloPrinterOperations.class)
8 })
9 public class HelloPrinterComponent implements HelloPrinterOperations {
10
11   @Receptacle(name="HelloReceptacle")
12   private HelloOperations[] helloArray;
13
14   @Override
15   public void printHello() {
16     for(HelloOperations hello : helloArray) {
17       System.out.println(hello.sayHello());
18     }
19   }
20
21   public void setHelloArray(HelloOperations[] helloArray) {
22     this.helloArray = helloArray;
23   }
24
25   public HelloOperations[] getHelloArray() {
26     return this.helloArray;
27   }
28
29 }
```

3.2.4 @LifeCycle

Esta anotação especifica que um determinado método da implementação de um componente ASCS trata um determinado evento de ciclo de vida. Um método marcado com esta anotação será chamado pelo núcleo de execução do ASCS toda vez que o evento do tipo informado na definição da anotação ocorrer. Os possíveis tipos de evento são:

* RUN

  Um método de ciclo de vida marcado com este tipo de evento representa o método que será executado na inicialização do componente. A inicialização de um componente é referente à chamada do método *startup()* definido na faceta *IComponent*. O método *RUN* definido pelo desenvolvedor do componente será associado automaticamente a uma *thread* distinta.

* SUSPEND

  Um método de ciclo de vida marcado com este tipo de evento representa o método que será executado na interrupção da execução do componente.

A interrupção de um componente é referente à chamada do método *shutdown()* definido na faceta *IComponent*. A *thread* associada ao método *RUN* também será interrompida.

* LOADED

Evento que informa que o componente foi carregado no contêiner. A carga do componente está associada à chamada do método *load()* definido na faceta *ComponentLoader*. Um método marcado com este tipo de evento pode, opcionalmente, receber como parâmetro um vetor de strings. Através desse parâmetro o desenvolvedor do componente pode ter acesso a valores que são atribuídos apenas na etapa de implantação do componente.

* UNLOADED

Evento que informa que o componente foi descarregado do contêiner. A descarga do componente está associada à chamada do método *unload()* definido na faceta *ComponentLoader*.

O trecho de código abaixo exemplifica o uso da anotação *@LifeCycle*.

Código 3.5: @LifeCycle - Exemplo de uso

```
1  /**
2   * Método de inicialização do componente.
3   * Imprime uma notificação de inicialização no console.
4   */
5  @LifeCycle(eventType=EventType.RUN)
6  public void init() {
7    System.out.println("The Component has been initialized.");
8  }
9
10 /**
11  * Método de instalação do componente.
12  * Imprime uma notificação que o componente
13  * foi instalado.
14  * @param args argumentos passados pela aplicação
15  */
16 @LifeCycle(eventType=EventType.LOADED)
17 public void load(String[] args) {
18   System.out.println("The Component has been loaded in container.");
19   System.out.print("Arguments received in load: ");
20   for(String s : args) {
21     System.out.print(s);
22   }
23   System.out.println();
24 }
```

3.2.5 @Property

Um campo marcado com esta anotação representa uma propriedade de instância do componente. A propriedade é identificada pelo valor configurado através de sua chave.

O valor do campo será atribuído, utilizando a chave da propriedade, através do acesso à faceta *ComponentProperties*. O campo deve obrigatoriamente, ser do tipo *String*, seguindo a definição da interface associada à faceta *Properties* demonstrada anteriormente na figura 3.1.

Seguindo a mesma idéia do receptáculo, a classe que contém um campo marcado com a anotação *@Property* precisa disponibiilzar métodos *setter* e *getter* para este campo.

O trecho de código abaixo exemplifica o uso da anotação *@Property*.

Código 3.6: @Property - Exemplo de uso

```
1  /**
2   * Componente que implementa o serviço de
3   * exemplo que consulta uma propriedade
4   * e imprime seu valor no console.
5   */
6  @FacetSet({
7    @Facet(name="PrinterFacet", type=MessagePrinterOperations.class)
8  })
9  public class MessagePrinterComponent implements MessagePrinterOperations {
10
11   @Property(key="msg")
12   private String message;
13
14   @Override
15   public void printMessage() {
16     System.out.println(message);
17   }
18
19   public void setMessage(String message) {
20     this.message = message;
21   }
22
23   public String getMessage() {
24     return this.message;
25   }
26
27 }
```

3.2.6 @MetaInfo

Anotação que permite que o desenvolvedor de um componente ASCS acesse meta-informações como o nome e a versão do componente. Tais informações são configuradas na hora da implantação do componente, através

do arquivo de descrições. Um campo marcado com esta anotação deve ser declarado como *public static String*, por ser uma propriedade referente à classe, e não a uma instância do componente. Sendo público, não há necessidade da definição de métodos *getter* e *setter*.

Uma anotação *@MetaInfo* deve ser detalhada com um dos seguintes tipos:

* NAME - Nome do componente.

* MAJOR_VERSION - Versão "major"do componente.

* MINOR_VERSION - Versão "minor"do componente.

* PATCH_VERSION - Versão "patch"do componente.

* PLATFORM_SPEC - Plataforma utilizada para o desenvolvimento do componente.

* FULL_DESCRIPTION - Texto completo que informa todas as meta-informações do componente. Este tipo será assumido automaticamente caso nenhum tipo seja informado.

O trecho de código abaixo exemplifica o uso da anotação *@MetaInfo*.

Código 3.7: @MetaInfo - Exemplo de uso

```
1  /**
2   * Componente que implementa o serviço de exemplo
3   * retornando "Hello World!".
4   */
5  @FacetSet({
6    @Facet(name="HelloFacet", type=HelloOperations.class)
7  })
8  public class HelloComponent implements HelloOperations {
9
10   @MetaInfo(type=Type.FULL_DESCRIPTION)
11   public static String DESC;
12
13   @LifeCycle(eventType=EventType.RUN)
14   public void init() {
15     System.out.println("The Component " + DESC + " has been initialized.");
16   }
17
18   @Override
19   public void sayHello() {
20     return "Hello World!";
21   }
22
23 }
```

3.3 Validação

O APT(Annotations Processing Tool) [20] é uma ferramenta Java de processamento de anotações. Com o APT, é possível desenvolver extensões para o compilador Java que são chamadas em pontos específicos do processo de compilação e permitem, por exemplo, emitir erros e *warnings* em um ponto específico do código-fonte.

O APT [20] já é suportado de forma nativa pela maioria das plataformas de desenvolvimento Java. No Eclipse [28], por exemplo, é possível habilitar o uso de processamento de anotações, de forma que o desenvolvedor será informado de um possível erro assim que acabar de escrever a linha de código. Também é possível a compilação do código utilizando o APT por linha de comando.

O ASCS é equipado com um mecanismo de validação baseado na ferramenta APT. Desta forma, o desenvolvedor do componente pode identificar de forma antecipada um possível erro estrutural do componente, antes mesmo do término de sua codificação, caso esteja utilizando um ambiente de desenvolvimento integrado com o API. Assim, o tempo gasto com a identificação e correção de erros estruturais é reduzido.

O framework do ASCS é acompanhado do arquivo *annotation-processors.jar* que contém os processadores responsáveis por realizar a validação do código-fonte do componente. Este arquivo é requisitado pelo APT no momento da compilação do código fonte.

A figura 3.2 mostra como que, na ferramenta Eclipse, o desenvolvedor do componente é notificado de um erro estrutural em seu código ASCS.

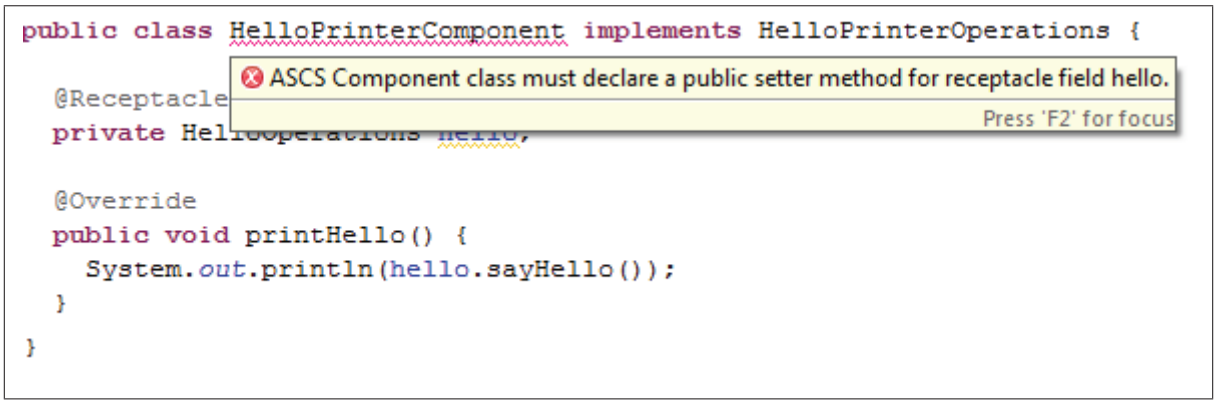

Figura 3.2: Validação em tempo de compilação

A seguir são detalhadas, para cada anotação, todas as restrições existentes que serão validadas pelo processador de anotações em tempo de compilação.

* @FacetSet - Uma classe marcada com a anotação *@FacetSet*, para ser válida, deve:

  - Apresentar apenas descrições de facetas que tenham a interface correspondente implementada pela classe.
  - Ser declarada como pública.
  - Ser uma classe concreta.
  - Apresentar um construtor sem parâmetros.
  - Apresentar todas as facetas referentes a tipos MyFacetOperations onde MyFacet é uma interface definida no arquivo IDL correspondente.
  - Estar em um ambiente onde as classes geradas a partir do arquivo IDL, necessárias para o funcionamento do ASCS, existam e são acessíveis.

* @LifeCycle - Um método marcado com a anotação *@LifeCycle*, para ser válido, deve:

  - Não apresentar qualquer tipo de parâmetro. O método *LOAD* é a única exceção, que pode, opcionalmente, receber um vetor de strings.
  - Ser declarado como público.
  - Ser um método concreto.

* @MetaInfo - Um campo marcado com a anotação *@MetaInfo*, para ser válido, deve:

  - Ser declarado como *public* e *static*.

- Ser do tipo *String*.

* @Property - Um campo marcado com a anotação *@Property*, para ser válido, deve:

  - Apresentar métodos *setter* e *getter* correspondentes.
  - Ser do tipo *String*.

* @Receptacle - Um campo marcado com a anotação *@Receptacle*, para ser válido, deve:

  - Apresentar métodos *setter* e *getter* correspondentes.

3.4 Construindo uma Aplicação ASCS

Esta seção apresenta um passo-a-passo para a construção de uma aplicação ASCS. O exemplo apresentado nesta seção é a mesma aplicação *HelloWorld* apresentada no capítulo anterior, compostas pelos componentes *Hello* e *HelloPrinter*.

A etapa de definição do componente através da criação de um arquivo IDL é a mesma demonstrada no capítulo anterior. A figura 2.1 apresenta o arquivo IDL utilizado no exemplo, e a figura 2.2 exemplifica a linha de comando utilizada para compilar o arquivo, gerando as classes Java necessárias para o desenvolvimento dos componentes.

Após a definição e compilação do arquivo IDL, o próximo passo é a implementação dos componentes. As figuras 3.2 e 3.3, apresentadas na seção 4.2 deste capítulo, exemplificam a implementação dos componentes definidos no arquivo IDL utilizando o modelo de programação ASCS.

Após a codificação dos dois componentes, o próximo passo, já relativo à implantação do componente, é criar os arquivos de descrição. A figura 3.8 mostra como deve ser o arquivo de descrição para o componente *HelloPrinter*.

Código 3.8: Arquivo de descrição do componente HelloPrinter

```
1 name = aHelloPrinter
2 major_version = 1
3 minor_version = 0
4 patch_version = 0
5 platform_spec = none
6 entry_point = scs.ascs.factory.AComponentFactory
7 extension = class
8 factory_arg = scs.demos.ascs.helloworld.HelloPrinterComponent
```

O campo *entry_point* corresponde à fábrica que será utilizada para criar o componente. No caso do ASCS, esse campo deve ser configurado exatamente

como mostrado no exemplo, sendo que a *AComponentFactory* é a classe que interpreta as anotações do componente e cria um componente SCS.

O campo *factory_arg* deve ser configurado com o caminho completo da classe de implementação do componente.

O nome dos arquivos de descrição deve seguir o mesmo padrão de nomes informado no capítulo anterior. Neste caso, o arquivo exemplificado deve ser nomeado *aHelloPrinter100.desc*.

O último passo relativo à construção da classe principal da aplicação é o mesmo demonstrado no capítulo anterior, na figura 2.6.

3.5 Implementação do ASCS

Esta seção apresenta uma visão geral do funcionamento interno do ASCS.

O pacote *scs.ascs.annotations* contém a definição de todas as anotações do ASCS. O conteúdo deste pacote forma o modelo de programação do ASCS, com o qual o desenvolvedor de um componente terá que interagir.

A implementação do ASCS se baseia nos mecanismos de reflexão da linguagem Java. O pacote *scs.ascs.core* contém as classes do núcleo do ASCS, nas quais ocorre o processamento reflexivo das anotações. A seguir, um resumo das classes que compõem este pacote.

* AComponentFactory

  Classe central da implementação do ASCS. Instanciada no contêiner SCS, é responsável por criar um *IComponent* através de um componente ASCS marcado com anotações. É no método *create()* desta classe que ocorre o processamento da anotação *@Facet* e a criação das facetas correspondentes no componente SCS. Esta é a classe correspondente à extensão do contêiner SCS que foi introduzida na seção 4.1 deste capítulo.

* AReceptaclesServant

  Implementação ASCS da faceta responsável por conectar os receptáculos do componente à facetas externas. É na implementação do método *connect()* definido nesta classe que a anotação *@Receptacle* é processada, e que uma conexão entre faceta e receptáculo é de fato realizada.

* AComponentContext

  Contexto de um componente ASCS. Armazena a instância do componente ASCS utilizada na hora de fazer a conexão dos receptáculos com as facetas pela classe *AReceptaclesServant*.

* AComponentLoaderServant

  Implementação do *ComponentLoader* utilizada pelo ASCS. Esta implementação delega os métodos *load()* e *unload()* para métodos específicos de tratamento de ciclo de vida do componente ASCS correspondente.

* AComponentPropertyServant

  Implementação da faceta *ComponentProperty* utilizada pelo ASCS. Esta implementação atribui o valor da propriedade informada no método *setProperty()* no campo do componente ASCS marcado com a anotação *@Property* com a chave correspondente.

* AComponentServant

  Implementação do *IComponent* utilizada pelo ASCS. Esta implementação delega os métodos *startup()* e *shutdown()* para métodos específicos de tratamento de ciclo de vida do componente ASCS correspondente.

Por fim, o pacote *scs.ascs.annotations.processor* contém as classes responsáveis pela validação em tempo de compilação das anotações do código-fonte do componente. Dentro deste pacote encontram-se classes relativas à validação de cada anotação, onde por exemplo a classe *FacetAnnotationProcessor* valida as restrições impostas pelo uso da anotação *@Facet*.