5
Trabalhos Relacionados

Durante o trabalho desenvolvido nesta dissertação foram estudadas diversas tecnologias que têm objetivos semelhantes ao nosso. Os trabalhos estudados apresentam modelos de programação que adotam diferentes paradigmas para construção de aplicações baseadas em componentes de software que visam um maior desacoplamento entre o código funcional da aplicação e a infra-estrutura relacionada ao modelo de componentes

As principais tecnologias que foram estudadas, apresentadas neste capítulo, são o SCA [13], como um conjunto de especificações para a construção de uma arquitetura genérica baseada em componentes, o Fraclet [14], como um modelo de programação baseado em anotações que permite compatibilizar o código com outras tecnologias de componentes como o Fractal e o OpenCOM, o AOKell [31], como um framework baseado no modelo Fractal que abstrai a camada de controle do componente utilizando a tecnologia de orientação a aspectos [9, 8], e o CompJava [32] como implementação do modelo DisComp [32], que permite a criação de um componente independente de middleware e de plataforma utilizando uma linguagem própria para programação orientada a componentes.

5.1
SCA

O SCA (Service Component Architecture) [13] define um modelo de programação para construção de aplicações orientadas a serviços (SOA). O conjunto de especificações SCA trata dos principais aspectos relacionados a esse modelo de programação, como o empacotamento de componentes em serviços, a composição de componentes, e os modelos de implementação e acesso de componentes em diferentes linguagens de programação e protocolos de comunicação. Uma das principais características do SCA é a possibilidade de tornar compatível o código de um componente pré-existente inicialmente desenvolvido para outro middleware ou modelo de componentes, sem a necessidade de reprogramação.

A arquitetura SCA tem como objetivo permitir que aplicações dis-

tribuídas adotem diferentes tecnologias para implementar os componentes de serviço e as conexões entre eles. Os componentes podem estar implementados em diferentes linguagens de programação. Para conexão entre componentes, várias tecnologias de comunicação e acesso podem ser adotadas, como *Web-Services* e *RMI*. Um núcleo de execução SCA deve disponibilizar contêineres relativos às linguagens de programação e às tecnologias de comunicação que devem ser suportadas.

Componentes SCA são agrupados em composições cuja descrição é feita por meio de um arquivo de configuração XML escrito em SCDL (Service Component Definition Language). Um arquivo SCDL descreve os componentes da composição e a conexão entre eles.

As principais abstrações do modelo SCA são: seviços, referências, propriedades e ligações. A figura 5.1 ilustra tais abstrações. Dois componentes SCA se conectam através da ligação de um serviço com uma referência.

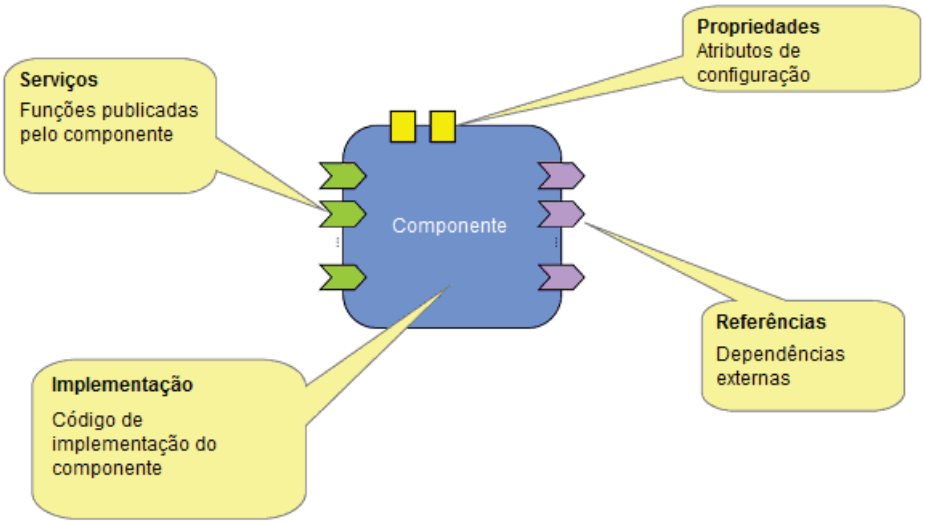

Figura 5.1: Modelo de Componentes SCA

O Tuscany [33] e o Fabric3 [34] são exemplos de tecnologias que implementam as especificações do SCA, e suportam a criação de componentes utilizando diferentes linguagens de programação e tecnologias de comunicação.

O modelo de programação Java do SCA é baseado em anotações [10] utilizadas no código-fonte de implementação do componente. Abaixo, as principais anotações que compõem o modelo de programação Java do SCA.

* @Service - Define as interfaces que compõe o serviço

* @Remotable - Indica que um serviço é remoto

* @Callback - Define uma interface de callback

* @Reference - Indica que um campo ou método da classe de implementação do componente é uma referência remota a um serviço de outro componente.

* @Property - Indica que um campo é uma propriedade de instância do componente.

* @Init - Indica que um determinado método deve ser chamado no momento da inicialização do componente.

O trecho de código a seguir exemplifica a implementação de um componente SCA.

Código 5.1: Componente SCA

```
1  public class ClientImpl implements Runnable {
2
3   private Service s;
4
5   public ClientImpl() {
6     Console.println("Client Created.");
7   }
8
9   @Init
10  public void init() {
11    Console.println("Client Initialized.");
12  }
13
14  public void run() {
15    s.print("Hello World!");
16  }
17
18  @Reference
19  public void setPrintService(Service s} {
20    this.s = s;
21  }
22 }
```

A definição da anotação *@Reference* indica que o campo "s"é uma referência externa para um serviço do tipo "Service", e deve ser atribuída através do método "setPrintService".

A definição da anotação *@Init* indica que o método "init"será executado no momento da inicialização do componente.

5.2 Fraclet

O Fraclet [14], é um modelo de programação Java baseado em anotações que seguem um modelo de componentes genérico. O Fraclet apresenta uma abordagem baseada em geração de código permitindo a compatibilização do código marcado com as anotações do Fraclet com outras tecnologias de componentes de software como o Fractal e o OpenCOM.

O modelo de componentes definido pelo Fraclet é um modelo genérico, baseado em abstrações normalmente encontradas em outros modelos de componentes. A figura 5.2 ilustra as abstrações que compõem este modelo.

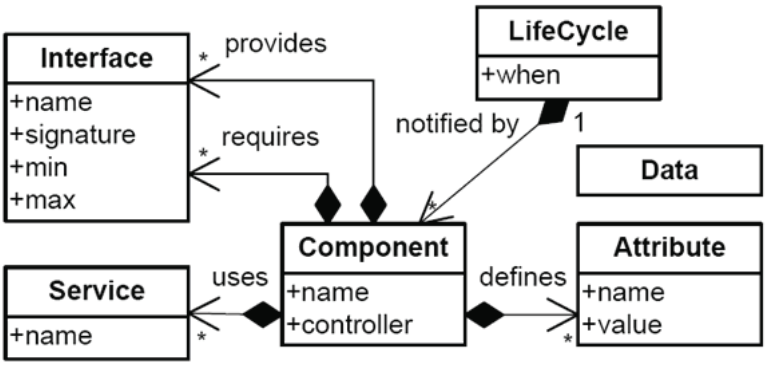

Figura 5.2: Modelo de Componentes Fraclet

As anotações do modelo de programação do Fraclet refletem diretamente as abstrações definidas no modelo de componentes. São elas:

* @Data - Estrutura de dados usada na comunicação entre componentes.

* @Component - Definição do componente.

* @Provides - Interface publicada como um serviço pelo componente.

* @Requires - Serviço necessário para o funcionamento do componente.

* @Attribute - Propriedade do componente.

* @Service - Interface utilizada para controle e configuração do componente.

* @LifeCycle - Método que controla o estado de ciclo de vida do componente.

O trecho de código mostrado na figura 5.2 exemplifica a implementação de um componente Fraclet.

Código 5.2: Componente Fraclet

```
1 @Data public class Request {
2  public Socket s;
3  public Reader is;
4  public PrintStream out;
5  public String url;
6 }
7
8 @Provides(name="a") public interface RequestHandler {
9  void handleRequest(Request r) throws java.io.IOException
10 }
11
12 public class Analyzer implements RequestHandler {
13  @Requires private RequestHandler rh;
```

```
14  @Requires private Logger l;
15  @Attribute(value="GET ") private String filter;
16
17  public void handleRequest(Request r) throws IOException {
18   r.in = new InputStreamReader(r.s.getInputStream());
19   r.out = new PrintStream(r.s.getOutputStream());
20   String rq = new LineNumberReader(r.in).readLine();
21   this.l.log(rq);
22   if(rq.startsWith(this.filter)) {
23    r.url = rq.substring(this.filter.length+1, rq.indexOf(' ', this.filter.
         length));
24    this.rh.handleRequest(r);
25   }
26   r.out.close();
27   r.s.close();
28  }
29 }
```

5.3 AOKell

O AOKell [31] é uma implementação do modelo Fractal [5] que visa extrair da implementação do componente o código não-funcional relativo ao controle e a configuração do componente. Diferentemente do SCA [13] e do Fraclet [14], ele não define seu próprio modelo de componentes.

Para atingir tais objetivos, o AOKell utiliza a técnica de programação orientada a aspectos [8], e considera as atribuições de controle e configuração do componente como interesses não-funcionais, isoladas em uma camada de aspectos. Esta camada será implementada com os próprios conceitos de componente do Fractal, onde uma entidade de controle como o LifeCycle é representada também por um componente.

A figura 5.3 mostra como o AOKell procura isolar na camada de aspectos o código relativo à controle e configuração do componente.

5.4 CompJava

O CompJava [32] é uma implementação do modelo DisComp [32]. Diferentemente dos trabalhos apresentados anteriormente, o CompJava especifica uma linguagem própria para construção dos componentes. Esta abordagem permite o desenvolvimento de um componente de forma independente de middleware, de forma que o código não-funcional relativo a conceitos de componentes é implementado pela própria linguagem.

As entidades do modelo DisComp são :

* Porta p = (n, k, I)

  n = Nome da porta

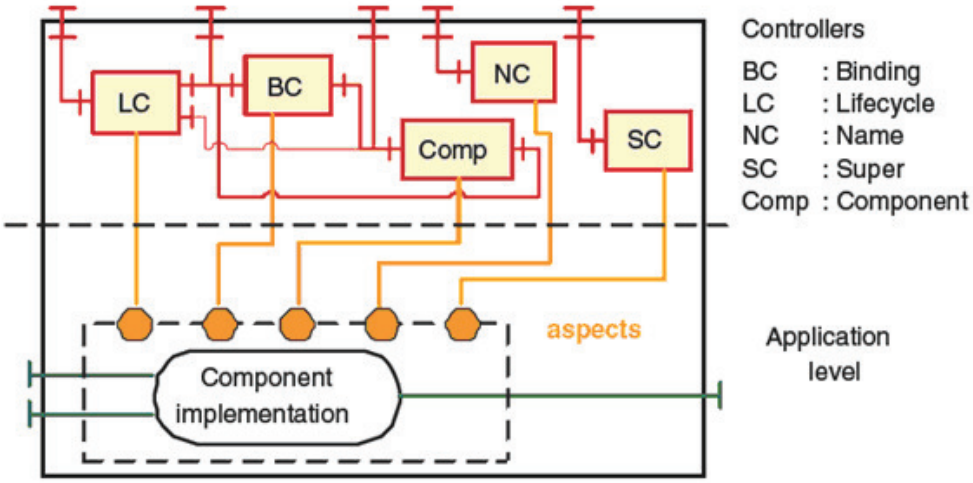

Figura 5.3: AOKell

k = Tipo da porta (requer uma interface ou disponibiliza uma interface)

I = A interface associada

* Tipo do componente T = (x, R)

x = Conjunto de portas

R = Acessível remotamente (booleano)

* Component c = (T, D, C)

T = Tipo do componente

D = Indica se o componente é distribuído

C = Conjunto de subcomponentes

Como ilustrado na figura 5.4, para programar um componente DisComp usamos uma linguagem específica, que estende a linguagem Java, onde temos palavras reservadas pra criar, configurar e conectar os componentes. Neste exemplo, o tipo do componente define como portas os serviços publicados e requeridos pelo componente, assim como as conexões com os outros serviços. O componente conterá a implementação de fato, escrita em Java.

```
component type CustomerAndAccountAdministrationType { }
component type CustomerHandlingType {
  port in provides InputEvents;
  port out1 requires CustomerData;
  port out2 requires CustomerIF;
}

component CustomerHandling ofType CustomerHandlingType{
  //port in provides InputEvents;
  //port out1 requires CustomerData;
  //port out2 requires CustomerIF;
  ...                    //implementation with e.g. an inner class
}


component CustomerAndAccountAdministration
            ofType CustomerAndAccountAdministrationType {
  CustomerViewType theCustomerView = new CustomerView();
  CustomerHandlingType theCustomerHandling =
                                    new CustomerHandling();
  CustomerType theCustomer = new Customer();
  connect theCustomerHandling.out1 to theCustomerView.in;
  connect theCustomerView.out to theCustomerHandling.in;
  connect theCustomerHandling.out2 to theCustomer.in;
  //initialization, main method etc.
}
```

Figura 5.4: CompJava

5.5 Comparando com o ASCS

Como vimos ao longo do capítulo, o SCA e o FRACLET foram os trabalhos estudados que mais se relacionam com o ASCS e com o tema principal do estudo desta dissertação por apresentarem um modelo de programação baseado em anotações. Esta seção visa comparar as anotações do modelo de programação ASCS com as anotações dos modelos de programação do SCA e do FRACLET.

A tabela a seguir associa as principais anotações que compõem os modelos de programação estudados:

Como vimos, as anotações @MetaInfo e @Data não possuem um mapeamento direto entre os três modelos de programação. A primeira foi criada para atender um caso específico ligado à conceitos do SCS, por isso não se aplica às outras tecnologias. A segunda foi criada para abstrair uma estrutura compartilhada entre dois componentes. O ASCS atualmente utiliza as próprias estruturas geradas pelo compilador de IDL para este tipo de caso. A inclusão

ASCS	**SCA**	**FRACLET**
@Facet	@Service	@Provides
@Receptacle	@Reference	@Requires
@LifeCycle	@Scope, @Init, @Destroy	@LifeCycle
@Property	@Property	@Attribute
@MetaInfo	-	-
-	-	@Data

Tabela 5.1: Comparando anotações

de uma anotação semelhante à @Data no ASCS é uma alternativa para que futuramente as dependências com o middleware CORBA possam ser retiradas do modelo de programação.

O modelo de programação do SCA é mais complexo e abrangente que os demais e possui diversas outras anotações para fins mais específicos que foram omitidas da comparação.