2
Estado da Arte na Implantação de Componentes

As tecnologias para componentes de software foram amplamente discutidas ao longo da última década e, assim, diversos modelos e sistemas foram propostos. É fundamental avaliar como esses modelos e sistemas suportam a implantação distribuída de seus componentes. Particularmente, é interessante observar se a metodologia de implantação é padronizada e, ao mesmo tempo, capaz de implantar artefatos inerentemente específicos de linguagem e plataforma. Com isso, espera-se que essas metodologias sejam aplicáveis a cenários de produção onde haja grande heterogeneidade de hardware, sistemas operacionais e software (eventualmente desenvolvidos em diferentes linguagens).

Este trabalho elegeu critérios para classificar os trabalhos relacionados à implantação distribuída. Consideramos trabalhos que ou suportam explicitamente o desenvolvimento baseado em componentes, ou, em caso contrário, que oferecem funcionalidades importantes à implantação distribuída, como controle programático e automatização. O processo de implantação considerado neste trabalho é detalhado na Seção 2.1. Os critérios para classificação são apresentados na Seção 2.2. A análise e a classificação dos trabalhos relacionados encontra-se a partir da Seção 2.3 e os comentários finais sobre as motivações identificadas a partir desse levantamento encontram-se na Seção 2.4.

2.1
Implantação Distribuída

A fim de orientar o estudo sobre os diferentes trabalhos, apresentados a partir da Seção 2.3, é importante aprofundar-se no conceito de implantação (do Inglês, *deployment*). Hall et al. trata a implantação de software como um conjunto de etapas[1] [21] que são aplicavéis à implantação de componentes distribuídos [4]. O uso de etapas de implantação bem definidas e com bom grau de separação de interesses permite um maior controle sobre a implantação. Por exemplo, viabiliza a mediação da implantação por planejadores automáticos que monitorem e atuem em casos de mudanças de estado observáveis [22].

As principais etapas do processo de implantação são [4]:

[1]Originalmente, Hall et al. usam o termo em Inglês *deployment activities*.

1. *liberação*: consta em disponibilizar todos os artefatos e descrições da configuração inicial necessários a instalação de um sistema;
2. *instalação*: consta em configurar e montar todos os artefatos necessários para que um sistema liberado entre em execução;
3. *ativação*: consta em colocar um sistema em execução levando-o a um estado pronto para uso;
4. *desativação*: consta em parar a execução de um sistema e desalocar os recursos utilizados por ele;
5. *reconfiguração*: consta em modificar um sistema instalado, ou ativo, para ajustá-lo às novas condições ambientais;
6. *atualização*: consta em modificar um sistema instalado através de uma nova instalação ou ativação de uma nova configuração desse sistema;
7. *remoção*: consta em remover todos os artefatos instalados;
8. *revogação*: consta em revogar os artefatos previamente liberados.

Alguns trabalhos [23] tornam explícita uma etapa de *montagem* (do Inglês, *assemble*) onde diferentes componentes ou aplicações que formarão o programa final devem ser montados (ligados ou compostos) de forma a se tornarem uma única unidade de implantação. De acordo com o processo de implantação considerado acima, a *montagem* deve ser posterior à *instalação* dos artefatos mais básicos.

Neste trabalho, entende-se por *empacotamento* (do Inglês, *packaging*) o processo que antecede a implantação e que é responsável por preparar os artefatos para que possam ser posteriormente liberados. O empacotamento engloba atividades como a compilação dos códigos-fonte e a criação de pacotes. Pacotes são arquivos (normalmente comprimidos) que agrupam binários, imagens e outros tipos de arquivos que façam parte do software.

Este trabalho considera os seguintes atores relacionados ao processo de implantação acima:

- *desenvolvedor*: quem cria os códigos e descreve os componentes.
- *empacotador*: quem usa as descrições para criar os pacotes e liberá-los.
- *administrador*: quem gerencia a localização e a configuração das infraestruturas (ou ferramentas) de implantação.
- *usuários*: quem usa as infraestruturas (ou ferramentas) para solicitar a carga dos componentes no ambiente de execução e, assim, executar uma aplicação.

2.2 Critérios para Classificação

Este trabalho elegeu um conjunto de critérios para classificar o suporte à implantação distribuída nas diferentes tecnologias relacionadas na Seção 2.3. A escolha desses critérios se baseia na visão que a implantação é um processo formado por etapas importantes que permeiam o ciclo de pós-desenvolvimento das aplicações, portanto é preciso investigar cada tecnologia segundo diferentes dimensões. Os critérios para classificação propostos por este trabalho são detalhados a seguir e são eles: *Controle da Implantação*, *Suporte à Composição*, *Repositórios de Implementações*, *Abstração da Distribuição*, *Modelo de Dependências* e *Abrangência Tecnológica*.

2.2.1 Controle da Implantação

É importante avaliar se as ferramentas para implantação respeitam e exploram essa visão mais completa sobre implantação ou se a consideram de forma simplificada como, por exemplo, só para lançar e parar aplicações. Um controle ampliado que permeie toda a execução da aplicação implantada é interessante para simplificar ações gerenciais e dinâmicas como monitoramento, adaptação, suspensão e pausa. A depender da estratégia de controle adotada pode-se promover o reuso dos planos de implantação e facilitar a gestão sobre as diferentes configurações de uma aplicação ao longo do tempo. Particularmente, isso pode ajudar os administradores a tomarem decisões de mais alto nível, como por exemplo, avaliar a eficácia dessas configurações.

De forma objetiva, este trabalho organiza as ferramentas estudadas em três grupos segundo o controle da implantação:

1. **Estático**: quando o controle não ocorre em tempo de execução da aplicação. Portanto, ou não há suporte às etapas como reconfiguração e atualização, ou essas só podem ser planejadas previamente em tempo de projeto. A maioria das abordagens nessa categoria usam descritores estáticos escritos em XML, em linguagens de descrição de arquitetura (ADL) ou em arquivos de configuração.

2. **Dinâmico por acesso direto à aplicação implantada**: quando o controle pode ocorrer em tempo de execução mas só através do acesso direto às entidades da aplicação implantada e às entidades do ambiente de execução. Nesses casos, entende-se que a implantação não é um conjunto de procedimentos padrão, já que algumas etapas como reconfiguração e atualização tendem a ser totalmente manuais e específicas por aplicação.

3. **Dinâmico por interface padrão**: quando o controle pode ocorrer em tempo de execução através de uma interface programática padronizada e que ofereça o controle das etapas da implantação. Essa interface programática normalmente existe com o objetivo de intermediar a implantação, simplificar atividades como monitoramento e adaptação, e evitar a necessidade do usuário agir manualmente para implantar seus artefatos. A interface programática pode ser visual ou não.

2.2.2 Suporte à Composição

Na área de Componentes de Software, entende-se por *composição* a fase que possibilita a integração de vários componentes em blocos de componentes mais complexos. Além das linguagens de descrição de interfaces (IDL) é comum o uso de linguagens de mais alto nível para descrever a arquitetura dos componentes em termos de suas dependências e composições, conhecidas como linguagens de descrição de arquiteturas (ADL). Cada tecnologia de componentes define uma forma própria para prover componentes compostos.

Lau e Wang apresentam uma classificação organizada pelos momentos do ciclo de vida dos componentes em que a composição pode ocorrer [24]. Esses mesmos autores apresentam um ciclo de vida ideal que potencializa a reusabilidade e que é ilustrado na Figura 2.1. Os tempos em que a composição pode ocorrer são:

1. **Projeto**: quando é possível definir componentes compostos em tempo de construção de código e, possivelmente, gerar uma única unidade de implantação.

2. **Implantação**: nesse tempo entende-se por composição a ligação (lógica na forma de *assemblies*) entre as unidades de implantação antes de disponibilizá-las para execução, seja em etapa específica para *montagem* ou durante a instalação.

3. **Execução**: quando compor significa criar conexões entre componentes que já estão em execução, permitindo, por exemplo, a troca de mensagens entre eles.

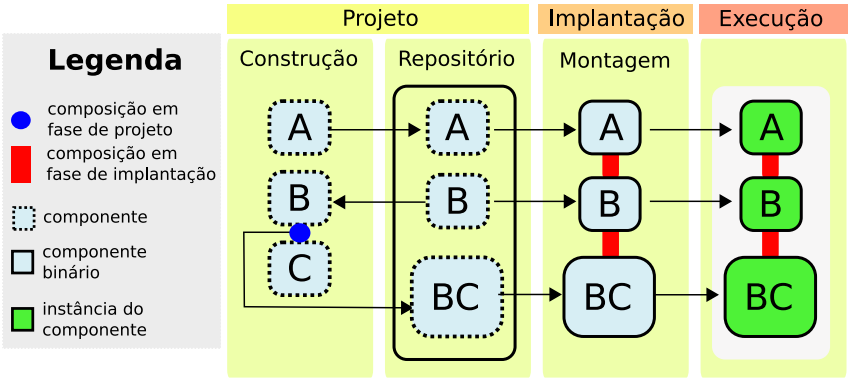


Figura 2.1: Um ciclo de vida com suporte ideal à composição de componentes

2.2.3 Repositórios de Implementações

Em cenários distribuídos é importante que se possa publicar e obter implementações de componentes de forma remota e distribuída. Portanto, a entidade do *repositório* de implementações de componentes se torna essencial. Ainda conforme Lau e Wang [24] um repositório pode ser uma entidade atuante no reuso ainda em tempo de projeto conforme visualizado na Figura 2.1. Portanto, este trabalho classifica o suporte a repositórios segundo três diferentes dimensões:

1. **Acesso**: dimensão relacionada a como se dá o acesso ao repositório:
   - **Local**: quando o repositório só pode ser usado localmente.
   - **Remoto**: quando também pode ser usado remotamente.
2. **Uso**: dimensão relacionada às funcionalidades disponíveis:
   - **Estático**: quando o repositório só armazena (e permite a obtenção de) um conjunto estático de componentes. Assim, não permite-se a publicação de novos componentes ao longo da execução do repositório. Lau e Wang [24] incluem esse tipo de repositório na classe *Design with Deposit-only Repository*.
   - **Dinâmico**: quando, ao longo da sua execução, o repositório permite (além da obtenção) a publicação de novos componentes e a revogação de componentes previamente publicados.
3. **Fase**: dimensão que indica em qual fase, do ciclo de vida da aplicação, o repositório pode estar presente:
   - **Projeto**: quando é possível usar o repositório em tempo de projeto.
   - **Implantação**: quando é possível usar o repositório em tempo de implantação.

– **Execução**: quando é possível usar o repositório em tempo de execução.

2.2.4 Abstração da Distribuição

Em cenários com grande escala de máquinas e softwares, é imprescindível entender como as ferramentas suportam o mapeamento da aplicação no ambiente físico da execução. Por exemplo, é interessante observar se o mapeamento pode ser automático, desonerando o administrador de tarefas manuais. Esse critério organiza os trabalhos nas seguintes categorias:

1. **Abstrato**: quando for possível distribuir os componentes de uma forma mais alto nível, através de uma estratégia automática de mapeamento das aplicações nos recursos físicos. É interessante observar se a ferramenta aceita a parametrização da seleção automática por restrições.[2]

2. **Concreto**: quando o planejamento da distribuição dos componentes ocorre através do preenchimento de descritores que indicam explicitamente onde eles devem ser implantados.

3. **Manual**: quando não houver uma forma padronizada e sistemática para mapear a aplicação nos recursos físicos. Nessa categoria, o administrador é responsável por manualmente instalar e lançar os componentes em cada máquina do ambiente de execução.

2.2.5 Modelo de Dependências

Embora o processo de implantação possua uma etapa de instalação, não dita-se como instalar os componentes nas máquinas físicas. Ao mesmo tempo, é comum precisar instalar tanto o componente principal quanto os outros artefatos dependentes. Do ponto de vista do administrador de várias aplicações, não é prático considerar que toda aplicação carregará consigo tudo que seja necessário à sua execução. Portanto, é fundamental avaliar como as diferentes tecnologias distinguem suas dependências de contexto e como suportam a instalação dos artefatos dependentes.

Este trabalho considera a visão apresentada por Szyperski [3] para distinguir os seguintes tipos de dependências:

[2]Do Inglês, *constraints*. As mais comuns são restrições de localização.

1. **Paramétricas**: aquelas resolvidas pela relação entre os serviços providos e os serviços requeridos pelos componentes. No caso das tecnologias baseadas em componentes, a dependência representa outros componentes. Nas outras tecnologias, a dependência representa outros serviços disponíveis no ambiente de execução e também implantados pela mesma ferramenta de implantação.

2. **Estáticas**: aquelas que representam os artefatos, recursos específicos da plataforma ou módulos da linguagem que precisem estar previamente instalados para considerar o software pronto para execução.

Para facilitar o entendimento, as dependências **paramétricas** podem ser entendidas como dependências internas ao modelo, ou seja, que referem-se a outras entidades definidas conforme o mesmo modelo. Enquanto as dependências **estáticas** podem ser entendidas como dependências externas ao modelo, ou seja, que referem-se a outras entidades que não estão definidas conforme o mesmo modelo. Assim, as dependências estáticas não desfrutam das facilidades de instalação providas pelo modelo específico da tecnologia.

Uma discussão mais abrangente sobre a viabilidade do uso de dependências estáticas num sistema de componentes de software é parte da proposta deste trabalho e encontra-se no Capítulo 4.

2.2.6 Abrangência Tecnológica

Ao construir uma ferramenta para implantação é preciso entender algumas questões tecnológicas relacionadas ao tipo de software que pode ser implantado e como se dá a implantação. Este trabalho considera duas dimensões:

1. **Linguagem**: indica as linguagens em que as aplicações devem ser desenvolvidas para poderem ser implantadas.

2. **Acesso Remoto**: representa as formas de acesso a cada recurso físico para proceder às atividades de implantação remota. Podem ser:

   - **Submissão de Tarefas**: principalmente as ferramentas integradas a *middlewares* de computação em grade oferecem suporte a ferramentas padrão de submissão de tarefas que cuidam remotamente das políticas de acesso, permissão e lançamento das aplicações.
   - **Serviço Próprio**: caso não se use uma ferramenta bem conhecida para acesso remoto, é comum que a ferramenta de implantação

use um serviço próprio para viabilizar o acesso remoto. Entende-se nesse caso que esse serviço executa em todas as máquinas que sejam acessíveis.

2.3 Tecnologias de Implantação Distribuída

Ao longo desta Seção são apresentadas as principais tecnologias de implantação distribuída. Cada uma dessas tecnologias será analisada de acordo com os critérios propostos na Seção 2.2 deste Capítulo. Algumas dessas tecnologias são amplamente difundidas no uso comercial e outras são projetos acadêmicos com contribuições interessantes.

2.3.1 Enterprise JavaBeans

A tecnologia EJB [9] (*Enterprise JavaBeans*), proposta pela Sun Microsystems, define uma arquitetura para servidores baseados em componentes *JavaBeans* [25] para a plataforma Java. O modelo de componentes *JavaBeans* oferece introspecção, persistência e comunicação por eventos. A conexão entre componentes, originalmente, só era possível através da comunicação por eventos locais [25].

A arquitetura do EJB define uma entidade chamada *contêiner* a ser implantada num *servidor de aplicações* Java e que encapsula: gestão do ciclo de vida dos componentes, persistência, transações, tolerância a falhas e comunicação remota. Um contêiner EJB pode hospedar dois tipos de componentes: *sessão* e *dirigidos a mensagens*. Os *componentes de sessão* são objetos distribuídos com ou sem estado cujo acesso remoto é através de Java RMI. Caso possua estado, não terá concorrência de acesso e permitirá a persistência automática. Já os *componentes dirigidos a mensagens* são objetos distribuídos com comportamento assíncrono e orientado a eventos através do Serviço de Mensagens de Java (JMS). Nesse último caso, o cliente não pode usar as interfaces remotas diretamente e o servidor do componente receberá as mensagens através de uma fila do JMS para, então, processar as requisições.

A versão 3.0 da tecnologia EJB simplificou o desenvolvimento dos componentes através do uso de anotações em códigos. Nesse caso, as anotações reduzem o número de classes e interfaces que o programador precisava implementar, além de diminuir a quantidade de informações no descritor de implantação. Por exemplo, se um *Bean* possuir uma anotação *@Remote* ele será acessível remotamente e o contêiner encapsula toda parte da comunicação distribuída com o componente. Na falta dessa anotação, entende-se que o compo-

nente é acessível apenas localmente. Essa estratégia eliminou a necessidade de uso de interfaces *Home* e a extensão dos objetos *EJBObject* e *EJBLocalObject*, que existiam na versão 2.1.

A unidade de implantação em EJB é um pacote no formato JAR que contém as classes Java compiladas que implementam o componente, objetos serializados, imagens, arquivos de propriedades, descritores de implantação e um arquivo de manifesto [9]. Um pacote JAR pode conter um ou mais componentes *JavaBeans* que são implantados de uma só vez no *contêiner* EJB mas que podem ser carregados separadamente respeitando os descritores de implantação. A diferença para um pacote JAR comum - da especificação do Java - é a presença de descritores de implantação e de metadados adicionais no manifesto para indicar quais são os componentes *JavaBeans*.

Os descritores de implantação em EJB são arquivos XML com diferentes objetivos. Num pacote JAR compatível com o EJB, é preciso ter: descritor dos componentes (normalmente chamado ebj-jar.xml), descritor da implantação (exemplo: sun-ejb-jar.xml) e classes Java que implementam os componentes. Os elementos do descritor de componentes são preenchidos, em tempo de projeto e podem, a partir da versão 3.0, ser anotações no próprio código-fonte. Contudo, EJB não dita o formato do descritor de implantação no servidor de aplicações e, assim, diferentes implementadores de EJB podem ter descritores específicos e incompatíveis com aqueles usados na implementação de referência da Sun.[3]

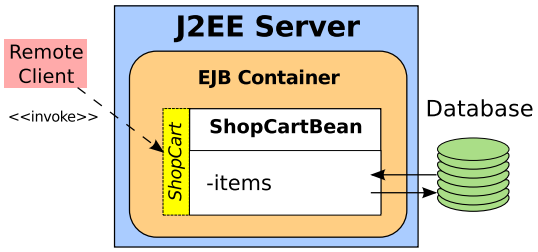


Figura 2.2: Arquitetura do Enterprise Java Beans 3.0

Um exemplo de um componente de sessão é ilustrado na Figura 2.2 e é definido conforme mostra o Código 2.1. Nesse exemplo, o componente *ShopCartBean* mantém estado sobre os itens de compra adicionados no carrinho e disponibiliza uma interface *ShopCart* que é acessível remotamente. Para implantar esse componente é preciso preencher um descritor de implantação com dados como: localização, usuário e senha tanto do servidor da aplicação quanto da base de dados a ser utilizada.

É importante salientar que EJB é proposta como uma arquitetura para desenvolvimento multi-camada e assume restrições de uso, detalhadas na seção

[3]Em `http://glassfish-samples.dev.java.net` existem alguns exemplos de descritores necessários para utilizar o servidor de aplicações da Sun.

Código 2.1: Exemplo do componente *ShopCartBean* de acordo com EJB 3.0

```
1 // Interface ShopCart (lógica de negócio)
2 import javax.ejb.Remote;
3 import javax.ejb.Stateful;
4 @Remote
5 public interface ShopCart {
6     public void addBook(String title);
7 }
8 // Classe para o componente (Bean) de sessão
9 @Stateful
10 public class ShopCartBean implements ShopCart {
11     List<String> items;
12     public void addBook(String title) { items.add(title); }
13 }
```

21.1.2 da especificação [9], podendo não se adequar a certos tipos de aplicações. Alguns exemplos dessas restrições são:

- Um componente não pode usar primitivas de sincronização de *threads* ou blocos para coordenar múltiplas instâncias, por considerar que só o contêiner EJB deve controlar o ciclo de vida e o modelo de concorrência.
- Um componente não pode tentar acessar arquivos ou diretórios no sistema de arquivos local, por considerar que as aplicações comerciais devem usar interfaces de programação para base de dados (como JDBC).
- Um componente não pode escutar um *socket*, aceitar conexões em um *socket* ou usar um *socket* para multicast. EJB evita que um componente atue como servidor em um *socket* diretamente, só é possível criar *sockets* como cliente. Desta forma, garante-se que o acesso remoto ao componente só é dado através do contêiner EJB.

O contêiner EJB só prevê a composição de componentes em tempo de projeto (e não em implantação ou execução). A fim de controlar o ciclo de vida dos componentes em tempo de execução, a Sun especificou o JMX [26] (*Java Management Extensions*) que permite o gerenciamento centralizado de componentes especiais conhecidos como *MBeans* (*Managed Beans*). Esses componentes encapsulam os componentes *JavaBeans* da aplicação ou recursos de rede, e são publicados em um servidor *MBeanServer*, expondo suas interfaces. É possível que um componente *MBeans* descubra outros do mesmo tipo através do mecanismo de notificação do servidor *MBeanServer*.

A arquitetura do JMX permite a instrumentação do contêiner EJB, pois: (i) os *MBeans* monitoram os componentes de usuário e os recursos de rede, e (ii) o gerenciamento remoto do *MBeanServer* é possível através de conectores (para comunicar-se via Java RMI, IIOP, JMS e Web Services) e adaptadores (para compatibilidade com outros protocolos de gerenciamento como SNMP). A combinação do EJB e JMX, embora permita a instalação, monitoramento e atualização (por notificações), apresenta alguns problemas:

– são dependentes de linguagem, pois os componentes precisam estar implementados em Java;

– não definem soluções para composição em tempo de implantação ou execução (limitação dada pelos contêineres EJB);

– não definem como resolver nem dependências entre componentes nem dependências estáticas.

Por fim, a Tabela 2.1 resume a classificação do EJB explorando o uso do JMX segundo os critérios propostos.

Critério Classificatório	**Avaliação**
Controle da Implantação	dinâmico por acesso direto
Suporte à Composição	projeto
Repositório de Implementações	–
Abstração da Distribuição	manual
Modelo de Dependências	–
Abrangência Tecnológica	**linguagem:** Java **acesso:** serviço próprio

Tabela 2.1: Classificação do EJB 3.0 incluindo o uso do JMX

2.3.2 OpenCom e Plastik

O OpenCom é um modelo de componentes minimalista com suporte a reflexão computacional destinado a construção de *middlewares* reconfiguráveis para cenários como sistemas embarcados, sistemas operacionais e roteadores programáveis [11]. O modelo de componentes do OpenCom define *interfaces* para provisão de serviços, *receptáculos* para requisição de serviços e *ligações* (do Inglês, *bindings*) entre interfaces e receptáculos. As ligações são representadas no modelo como um componente que associa dois outros. Esses elementos são definidos usando a OMG IDL [27] (do Inglês, *Interface Description Language*), entretanto, um componente OpenCom é definido usando uma extensão da OMG IDL para indicar a estrutura do componente (versão, interfaces e receptáculos) [11].

O compilador estendido de OMG IDL do OpenCom pode gerar código para as linguagens C, C++ e Java. O código gerado permite o acesso ao núcleo (*kernel*) do OpenCom que oferece suporte à introspeção sobre a estrutura do componente, à interceptação, à descoberta de informações sobre as interfaces e à exposição da topologia de composição. O Código 2.2 exemplifica a definição de três interfaces e dois componentes, um servidor e outro cliente. Após a compilação dessa IDL estendida, um exemplo da implementação desses

componentes em Java é exemplificado no Código 2.3. Nesse Código, o método *setup* busca a referência à interface *IServ1* do componente *Server* e a usa para criar uma ligação com o receptáculo *IServ1* do componente *Client*. Nas linhas 2 e 7 observa-se que as classes *_Server* e *_Client* são parte do código gerado pela compilação da IDL estendida e fornecem acesso à estrutura (identificadores, receptáculos e interfaces) dos componentes *Server* e *Client*, respectivamente.

Código 2.2: Exemplo da definição de interfaces e de componentes OpenCom usando a extensão da OMG IDL

```
1 // Interfaces em OMG IDL padrão
2 interface IServ1 {int op1(int i, int j);}
3 interface IServ2 {int op2(int i);}
4 interface IClient {int setup(); int call(char op, int arg1,int arg2);}
5 // Extensão da OMG IDL para descrever o tipo do componente OpenCom
6 componentType Server {version:int=1; provides IServ1; uses IServ2;}
7 componentType Client {provides IClient; uses IServ1;}
```

Código 2.3: Exemplo de código Java que mostra o uso do núcleo do OpenCom

```
1 // Implementação da interface IServ1
2 public class Server extends _Server implements IServ1 {
3   public Server() { super (); }
4   public int op1(int a, int b) {return a+b;}
5 }
6 // Exemplo da implementação do cliente que tem um receptáculo para IServ1
7 public class Client extends _Client implements IClient {
8   public Binding b;
9   public Client() { super(); }
10  public int setup() {
11    // obtendo a identificação do componente
12    Template t_serv = kernel.load("Server");
13    // instanciando o componente Server
14    Component serv = kernel.instantiate(t_serv.id);
15    // obtendo a referência para a interface IServ1 no componente Server
16    OCM_IRefList iList = (OCM_IRefList) kernel.getprop(serv.id,"I");
17    IServ1 i_IServ1 = (IServ1) iList.getIRef("IServ1");
18    // criando a ligação entre a interface e seu receptáculo
19    kernel.bind(r_IServ1.id, i_IServ1.id, b);
20  }
21  public int call(char op, int arg1, int arg2) {
22    // ...
23    // uso do receptáculo
24    int result = r_IServ1.op1(arg1, arg2);
25    return result;
26  }
27 }
```

O OpenCom possui uma infraestrutura de execução para componentes que usa o núcleo do OpenCom para permitir a gestão do ciclo de vida, a reconfiguração em tempo de execução e, portanto, é a base para se implantar componentes OpenCom. Essa infraestrutura oferece três componentes básicos, ilustrados na Figura 2.3, são eles:

– *Caplets* representam nós de implantação dependentes da implementação do modelo de componentes. A existência dos *caplets* motiva-se na separação dos componentes em diferentes domínios, com o devido isola-

mento e permitindo o suporte à diferentes definições sobre o conceito de componente (chamam de *component styles*). Por exemplo, pode haver um *caplet* para carregar componentes C++ e outro para componentes Java. O conjunto de componentes, primários ou compostos, dentro de um *caplet* define a idéia lógica de cápsula (*capsule*) que é semelhante ao contêiner EJB, pois provê um espaço de nomes e identificação única para interfaces e componentes;

– *Loaders* encapsulam a manutenção do ciclo de vida dos componentes. Os *Loaders* são especializados para um determinado tipo de *Caplet* facilitando a carga (*load*) e instanciação (*instantiate*) dos componentes em cenários heterogêneos;

– *Binders* permitem a gestão dos mecanismos de composição, através da criação (*bind*) ou da destruição (*destroy*) das ligações (*bindings*).

Além disso, o núcleo do OpenCom oferece um registro (*registry*) que armazena as meta-informações sobre os elementos do modelo (como informações sobre as facetas, receptáculos e conexões) e suporta notificações. O mecanismo de notificação permite, por exemplo, que um objeto de *callback* seja informado sobre as chamadas aos métodos *load* e *bind*.

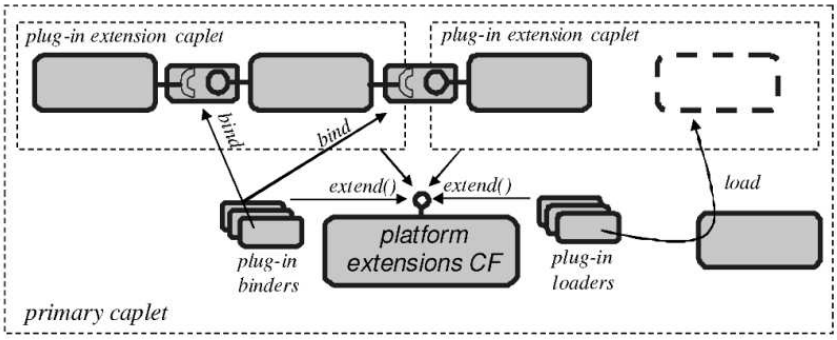


Figura 2.3: Componentes da infraestrutura de execução do OpenCom (fonte: [11])

O Plastik [20] é um *framework* que usa a infraestrutura de execução do OpenCom e é capaz de reconfigurar componentes OpenCom durante sua execução, desde que as reconfigurações sejam planejadas em tempo de projeto. O Plastik permite a definição de restrições (chamadas de *invariantes*) que não podem ser violadas por tais reconfigurações, se uma reconfiguração viola uma restrição então ela não é realizada. Para tanto, o Plastik usa uma abordagem baseada na extensão de uma linguagem de descrição de arquitetura (ADL), no caso a ACME [28], para descrever as ações de reconfiguração. A compilação das descrições arquiteturais geram *scripts* que são usados como entrada para

um configurador do ambiente de execução, que, por sua vez, realiza a carga e a instanciação da aplicação baseada em componentes [29].

Embora o OpenCom permita o uso remoto dos serviços da infraestrutura de execução, Coulson et al. [11] não comentam como resolvem questões fundamentais para a implantação distribuída, como por exemplo: (i) como se dá o acesso inicial aos *Caplets*; (ii) quais são os artefatos que compõem as unidades de implantação; (iii) como se instala as unidades de implantação nas diferentes máquinas para que, por fim, os componentes possam ser carregados. O Plastik também não resolve essas questões e concentra-se na manutenção da arquitetura dinamicamente reconfigurável.

Portanto, entendemos que o trabalho aqui proposto completa as lacunas existentes no OpenCom no que tange a implantação distribuída, podendo inclusive ser adaptado para implantar componentes OpenCom. Além disso, o OpenCom é estruturalmente semelhante ao modelo e aos serviços usados como base neste trabalho e detalhados no Capítulo 3.

Por fim, a Tabela 2.2 resume a classificação do OpenCom explorando o uso do Plastik segundo os critérios propostos.

Critério Classificatório	**Avaliação**
Controle da Implantação	dinâmico por acesso direto
Suporte à Composição	implantação, execução
Repositório de Implementações	–
Abstração da Distribuição	manual
Modelo de Dependências	paramétricas
Abrangência Tecnológica	**linguagem:** C, C++ e Java **acesso:** serviço próprio

Tabela 2.2: Classificação do OpenCom incluindo o uso do Plastik

2.3.3 OMG CCM e OpenCCM

A OMG especifica o modelo de componentes distribuídos conhecido como CCM (*CORBA Component Model*) [10].[4] Os componentes CCM são definidos em termos da OMG IDL 3.0 [10] (do Inglês, *Interface Description Language*) que estende a OMG IDL existente na especificação CORBA 2.6 [27]. O Código 2.4 exemplifica a definição de dois componentes ilustrados na Figura 2.4, destacando as relações entre os diferentes tipos de portas. O modelo do CCM define quatro tipos de portas: (i) *facetas* para provisão de serviços; (ii) *receptáculos* para requisição de serviços; (iii) *fontes de eventos*

[4]Ao longo desta Seção referimo-nos à versão 3.0 do CCM, ainda que versão mais atual da especificação do CCM seja 4.0. A versão mais nova é avaliada na Seção seguinte.

(*event sources*) para publicação de eventos; e (iv) *consumidores de eventos* (*event sinks*) para recepção de eventos. Todas as portas de um componente CCM são implementadas como objetos CORBA [30]. Enquanto as facetas e os receptáculos possibilitam a interação síncrona, fontes e os consumidores de eventos permitem a interação assíncrona.

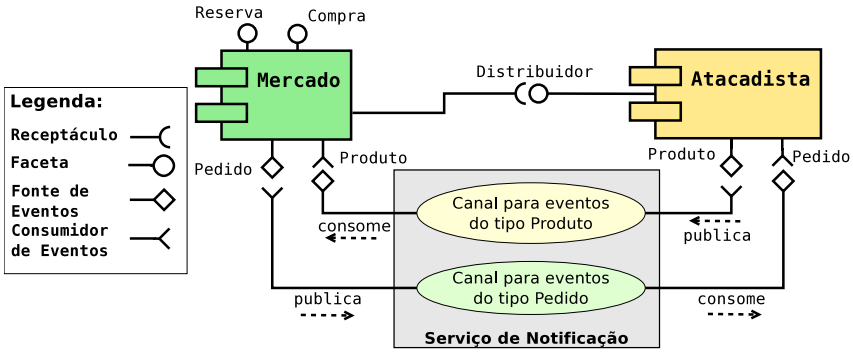


Figura 2.4: Modelo de componentes CCM

Código 2.4: Exemplo de componente definido conforme a OMG IDL 3.0

```
1  //Interfaces em IDL CORBA 2.6
2  interface Compra {};          //Extensões para eventos em IDL CORBA 3.0
3  interface Reserva {};         eventtype Produto { public string id; };
4  interface Distribuidor {};    eventtype Pedido { public string prod_id; };
5  //Extensões para definir componentes em IDL CORBA 3.0
6  component Mercado {
7    provides Compra compra;    //faceta
8    provides Reserva reserva; //faceta
9    uses Distribuidor distribuidor; //receptáculo
10   consumes Produto entrega; //consumidor de eventos
11   publishes Pedido pedido;  //fonte de eventos
12 }
13 component Atacadista {
14   provides Distribuidor distribuidor; //faceta
15   consumes Pedido pedido;       //consumidor de eventos
16   publishes Produto encomenda; //fonte de eventos
17 }
```

A maior contribuição do CCM é padronizar o ciclo de desenvolvimento de componentes através do uso de CORBA como *middleware* de comunicação distribuída [31]. Assim, os desenvolvedores podem *projetar* seus serviços em termos da OMG IDL 3.0, de forma que, seus componentes serão compatíveis com as ferramentas de diferentes implementadores do padrão CCM. O componente resultante pode ser *composto* e *empacotado*, por exemplo, em um arquivo DLL ou JAR. Após empacotados, os componentes podem ser *implantados* em vários servidores de componentes para futura *instanciação* e, a partir daí, entram em execução. CCM define que os *contêineres de componentes* devem ter uma interface conhecida como *CCMHome*, que será usada para criar e expor um único tipo de componente que pode ser carregado no contêiner.

A composição entre componentes CCM pode ser definida em tempo de projeto, ou mesmo em tempo de implantação, através do uso da linguagem OMG CIDL (do Inglês, *Component Implementation Description Language*) [10]. A OMG CIDL permite, por exemplo: (i) especificar conexões entre os diferentes tipos de portas; (ii) escolher a política do ciclo de vida do componente; e (iii) escolher a persistência de estados do componente.

A implantação de componentes CCM é detalhada no Capítulo *Packaging and Deployment* da especificação do CCM [10]. Ela se baseia na confecção de vários descritores no formato XML para descrever os componentes, suas implementações, as configurações iniciais das suas instâncias e seu mapeamento nas máquinas físicas. O processo de implantação é composto pelas etapas ilustradas na Figura 2.5:

1. Obtém-se do usuário as informações sobre locais de instalação, quantidade de instâncias por servidor, políticas de ativação dos objetos, etc.
2. Instala-se cada implementação de componente na plataforma de destino.
3. Cria-se a instância do servidor de componentes para então criar o contêiner, dentro do qual será carregado o *CCMHome* do componente.
4. Cria-se as instâncias dos componentes através da interface *CCMHome*.
5. Aplica-se um configurador a cada instância de componente.
6. Estabelece-se as conexões entre as portas dos componentes.
7. Indica-se o término da implantação pela chamada da operação *configuration_complete* em cada componente.

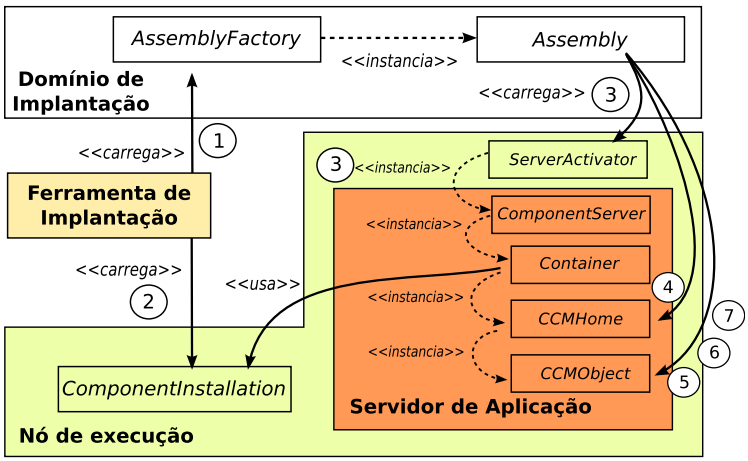


Figura 2.5: Relação entre objetos no processo de implantação (fonte: [32])

A unidade de implantação no CCM é um *pacote de componente* (formato ZIP) que agrupa: (i) um *descritor do pacote* em formato XML com as informações da plataforma (linguagem e sistema operacional suportados) e o ponto de entrada para a carga do componente; (ii) um *descritor do componente* em formato XML com as informações sobre a estrutura do componente (nome e tipo das portas) e os atributos necessários para implantação no contêiner; e (iii) a *implementação do componente* que contém os artefatos binários (.class, .exe, .bin), os arquivos de metadados gerados pelos compiladores de CIDL e IDL e outros arquivos constituintes da implementação (imagens, ícones, texto, etc.). Entretanto, para implantar um componente no ambiente de execução são necessários ainda outras informações. Essas informações formam um *pacote de montagem* (formato ZIP) que é específico por aplicação e agrupa: (i) os dados sobre o ambiente de execução como número de instâncias de componentes, localização do servidor de aplicação e máquina a ser utilizada; (ii) um *descritor de propriedades* em formato XML para cada instância de componente contendo os valores das propriedades do componente; (iii) um *descritor de montagem* em formato XML que indica a localização dos *pacotes de componentes* e dos *descritores de propriedades* desses componentes; (iv) os *pacotes de componentes* constituintes da aplicação. A organização dos pacotes e descritores de implantação é ilustrada na Figura 2.6.

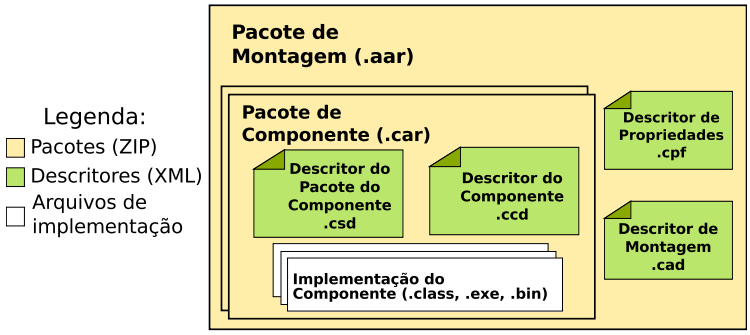


Figura 2.6: Organização dos descritores de componentes do CCM (fonte: [32])

A especificação de CCM prevê que a implantação pode ser conduzida por ferramentas de terceiros, mas não propõe uma interface padrão que tais ferramentas devam seguir. Essa especificação concentra-se apenas em definir as entidades que formam o ambiente de execução, a ordem de uso e o formato dos descritores de implantação. Portanto, cabe a cada implementador prover seu próprio suporte à implantação distribuída.

O OpenCCM é uma implementação CCM de código aberto [33]. Essa implementação disponibiliza várias ferramentas para o desenvolvimento de componentes CCM em Java. Destacam-se três ferramentas principais:

– *Open Production Tool*: agrupa ferramentas úteis em tempo de projeto para compilar os arquivos IDL, verificar a consistência das descrições XML e converter diagramas UML em arquivos IDL, em descritores de componentes e em código Java.

– *Open Packaging/Assembling Tool*: fornece uma interface gráfica, ilustrada na Figura 2.7, que auxilia o preenchimento dos descritores da implantação especificados no CCM.

– *Open Distributed Deployment Infrastructure*: fornece três serviços para concretizar a implantação [34]: (i) *NodeManager*, que precisa estar executando em todas as máquinas; (ii) *DCIManager*, que também precisa estar executando para instalar os componentes nos *NodeManagers* e interpretar os descritores da implantação a fim de instanciar o (iii) *AssemblyManager*, que por sua vez, instancia os componentes e ativa as conexões.

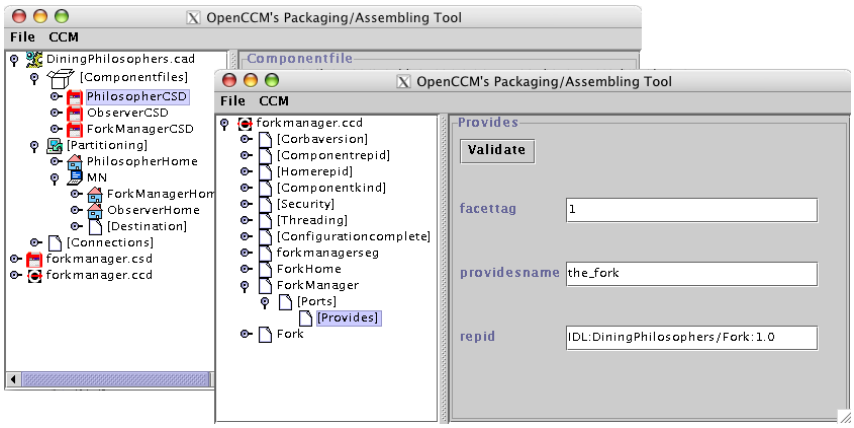


Figura 2.7: Capturas de tela da ferramenta de empacotamento do OpenCCM

Por outro lado, Briclet et al. [34] não comentam se o OpenCCM permite a implantação de componentes desenvolvidos em outras linguagens (como C e C++) ou usando outras implementações do CCM como o CIAO [35].

Do ponto de vista da implantação, a especificação do CCM apresenta algumas limitações:

– Não especifica o uso de repositórios para obter remotamente os pacotes de componentes.[5]

– Não especifica como instalar os pacotes de componentes. Por exemplo, no OpenCCM o *DCIManager* provê um mecanismo próprio para isso.

[5]É importante não confundir os repositórios de interfaces existentes no CCM com repositórios de pacotes de componentes (que agrupam as implementações).

– Obriga o ator da implantação a preencher uma grande quantidade de descrições, não apenas sobre a estrutura do componente mas também sobre sua configuração (quantidade de servidores, quantidade de instâncias, mapeamento físico nas máquinas, etc.).

– Não facilita o reuso dos descritores de implantação, já que em cenários um pouco diferentes, a maioria dessas descrições precisam ser recriadas.

– Não especifica como adaptar as conexões entre os componentes pois o pacote de montagem só define a configuração inicial [36]. Em casos de reconfiguração da aplicação, faz-se necessário que outras ferramentas atualizem os descritores.

CCM permite descrever três tipos de dependências estáticas (DLL, JAR e binários) que podem ser específicas por plataforma e linguagem mas que precisam estar dentro do pacote de implementação ou previamente instaladas na máquina. Mesmo assim, CCM não dita como instalar as dependências estáticas. Logo, ou o programador é obrigado a incluir no pacote todas as dependências ou o administrador precisa certificar-se que as dependências estejam previamente instaladas nas máquinas. Em ambos os casos, essa abordagem mostra-se manual, propensa a erros e não incentiva o reuso de artefatos binários. Este trabalho propõe uma solução para esse problema no Capítulo 4.

Por fim, a Tabela 2.3 resume a classificação do CCM considerando o uso do OpenCCM segundo os critérios propostos. Como sugestão, indicamos aos leitores iniciantes na tecnologia CCM 3.0 que leiam o trabalho de Nardi [32] pois trata-se de uma abordagem mais didática para explicar o funcionamento do CCM, além de ser um dos poucos textos em língua portuguesa.

Critério Classificatório	**Avaliação**
Controle da Implantação	dinâmico por interface padrão
Suporte à Composição	projeto, implantação
Repositório de Implementações	–
Abstração da Distribuição	concreto
Modelo de Dependências	paramétricas, estáticas
Abrangência Tecnológica	**linguagem:** Java **acesso:** serviço próprio

Tabela 2.3: Classificação do CCM incluindo o uso do OpenCCM

2.3.4 OMG D&C e DAnCE

Recentemente, a OMG definiu uma especificação exclusiva para implantação de componentes CCM, conhecida como D&C (*Deployment and Configuration*) [14]. Essa especificação substitui o Capítulo *Packaging and Deployment* da especificação CCM 3.0 [10] e foi lançada concomitantemente com a versão 4.0 da especificação CCM [37]. Mesmo assim, a definição de componentes ainda é a mesma utilizada na versão 3.0 da especificação do CCM. A especificação OMG D&C revisa os formatos dos descritores e define interfaces para a gestão distribuída dos processos de implantação e de configuração de aplicações baseadas em componentes CCM.

A unidade de implantação na OMG D&C também é um *pacote de componente*, semelhante ao apresentado na Seção 2.3.3 deste trabalho. Contudo, a OMG D&C organiza os constituintes de um pacote conforme um novo modelo de dados que favorece a separação dos atores durante o desenvolvimento:

- O **projetista** (do Inglês, *specifier*) define as portas e a estrutura do componente e produz descritores do tipo *ComponentInterfaceDescription*.
- O **desenvolvedor** cria um ou mais artefatos que implementam as portas especificadas pelo projetista e, então, produz descritores dos tipos: (i) *ImplementationArtifactDescription* para descrever seus artefatos e outros dependentes; e (ii) *MonolithicImplementationDescription* e *ComponentImplementationDescription* para o componente como um todo.
- O **montador** (do Inglês, *assembler*) cria composições (do Inglês, *assembly*) entre os componentes implementados por diferentes desenvolvedores. O montador pode configurar e interconectar os subcomponentes para, então, produzir descritores dos tipos: (i) *ComponentAssemblyDescription* que descreve a composição; e (ii) *ComponentImplementationDescription* que define o novo componente composto.
- O **empacotador** (do Inglês, *packager*) agrupa as várias implementações de um mesmo componente em um *pacote de componente* baseado nos descritores dos tipos *ComponentInterfaceDescription* e *ComponentImplementationDescription*. O empacotador certifica-se que as implementações são válidas conforme as interfaces projetadas e produz descritores do tipo *ComponentPackageDescription*.

A partir da análise das unidades de implantação da OMG D&C é possível notar que, ao contrário de CCM 3.0, a OMG D&C não inclui nos pacotes de componentes as informações sobre o ambiente de execução a ser usado.

Entretanto, essas informações ainda são necessárias e devem ser providas pelos atores da implantação. A interação entre os atores da implantação é organizada pela existência de modelos de dados e modelos de gerenciamento da infraestrutura de execução. Os modelos de dados permitem descrever: (i) os recursos físicos (*target data model*); e (ii) as instâncias, o mapeamento nos recursos e as configurações dos componentes (*execution data model*). Já os modelos de gerenciamento definem interfaces CORBA para coordenar tanto os recursos físicos (*target management model*) quanto as entidades internas da infraestrutura de execução (*execution management model*).

Uma implementação da OMG D&C é o *framework* DAnCE [13]. Em particular, o DAnCE foi desenvolvido para oferecer suporte a qualidade de serviço (QoS) em tempo real baseado na motivação de que *middlewares* como J2EE e .NET não se adequam bem a aplicações distribuídas embarcadas e de tempo real (do Inglês, *DRE - Distributed Real-Time and Embedded*). O DAnCE utiliza a implementação CCM provida pelo CIAO [35] que combina a flexibilidade de um *middleware* baseado em componentes com a previsibilidade oferecida pelo RT-CORBA [38].

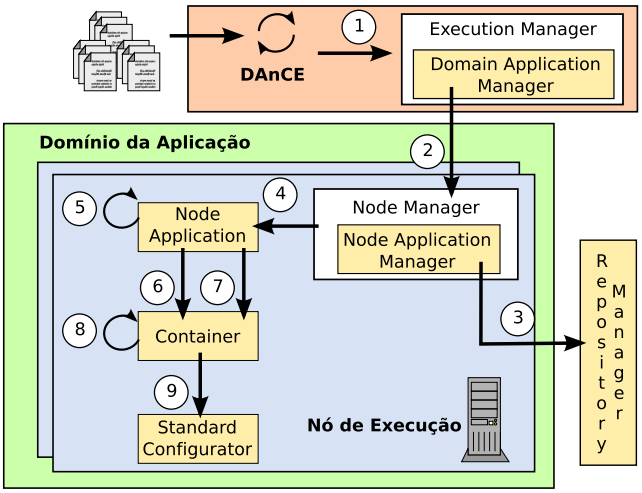


Figura 2.8: Arquitetura interna e etapas da implantação no DAnCE (fonte: [13])

A implantação coordenada pelo DAnCE é ilustrada na Figura 2.8 e segue as seguintes etapas:

1. A partir dos descritores dos componentes e do ambiente de execução, o serviço *ExecutionManager* é usado para iniciar a implantação em vários domínios (*DomainApplication*) que são compostos por *nodes, interconnects, bridges* e *resources*, definindo um *TargetEnvironment*.

2. O serviço *DomainApplicationManager* (co-alocado no *ExecutionManager*) instancia cada nó do domínio e solicita a implantação dos componentes nos nós de execução que formam o domínio.

3. O controle passa ao *NodeManager* que executa em cada máquina remota e concretizará a implantação dos componentes remotamente. Para isso ele cria um *NodeApplicationManager* que obtém as referências para os componentes instalados no repositório.

4. Em seguida, o *NodeApplicationManager* inicia os diversos *NodeApplications*, que agem como servidores de componentes.

5. Os servidores dos *NodeApplications* aplicam as configurações requisitadas pelas implementações dos componentes.

6. Cada *NodeApplication* inicia os serviços do *middleware* que serão usados pelas instâncias dos componentes, como serviços de notificação e eventos.

7. Os *NodeApplications* criam os contêineres.

8. Cada *Container* carrega as instâncias dos componentes.

9. Finalmente, cada *Container* aplica as configurações iniciais nas instâncias dos componentes.

No DAnCE, em especial, o repositório ()*RepositoryManager*) é dedicado para cada domínio de aplicação (*DomainApplication*) e as ligações entre componentes são automaticamente efetivadas se houver *interconnects e bridges* entre os nós de execução associados ao domínio de aplicação.

A OMG D&C apresenta um processo de implantação no qual algumas etapas são definidas diferentemente daquelas etapas consideradas na introdução deste Capítulo. As principais diferenças são:

– A instalação é considerada o ato de levar um pacote de componente disponível até um repositório de implementações.[6] Neste trabalho, entretanto, considera-se que a instalação é o ato de expandir um pacote de componente numa máquina alvo que tenha sido escolhida para fazer parte do ambiente de execução. Portanto, o ato de instalar é executado diretamente no sistema de arquivos de uma máquina alvo e não em um repositório. Este trabalho nomeia a ação de disponibilizar um pacote de componente em um repositório (remoto ou não) como *liberação* ou ainda *publicação*. A ação inversa chamamos de *revogação*.

[6] A pré-condição é que ou ator da implantação tenha em mãos os pacotes de componentes ou eles já estão instalados no repositório.

– A localização do repositório não implica necessariamente no local de instalação onde o componente deve executar. Este trabalho também compartilha dessa visão, mas considera que durante o processo de instalação nos sistemas de arquivos de uma máquina alvo, o repositório (geralmente remoto) é invocado para fornecer as implementações de componentes previamente publicadas. Portanto, o ator da implantação não é obrigado a possuir os pacotes de componentes previamente. Esses, por sua vez, deveriam ser publicados diretamente nos repositórios pelos desenvolvedores, promovendo o reuso dos repositórios em tempo de projeto, conforme discutido na Seção 2.2.3.

– O planejamento faz parte da implantação e é a etapa que decide como e onde os componentes devem executar no ambiente de execução, sem necessariamente ter qualquer efeito no ambiente de execução. Este trabalho também compartilha dessa visão, mas ressalta que o planejamento: (i) não deve obrigar o desenvolvedor a cuidar do mapeamento físico; e (ii) os planos devem permitir alterações em tempo de execução.

O DAnCE possui quatro principais contribuições para a implantação de aplicações baseadas em componentes [13]:

1. O repositório no DAnCE pode atuar como um cliente HTTP para se relacionar com outros repositórios e a obtenção dos componentes versionados é otimizada pelo uso de cache local no *NodeApplicationManager*.

2. O gerente de domínio (*DomainApplicationManager*) coordena o ciclo de vida dos componentes montados (do Inglês, *assemblies*) considerando os estados pré-ativo, ativo, passivo ou inativo para cada componente. Assim, é possível garantir só conectar e ativar os componentes quando todos os outros estiverem no estado pré-ativo. Além disso, garante-se que as invocações remotas se concretizem antes do componente entrar num estado passivo ou inativo.[7]

3. Estende-se as descrições da OMG D&C para viabilizar que o servidor de componentes (*NodeApplication*) seja configurado com: (i) as opções de linha de comando do ORB e (ii) os parâmetros de tempo real para o ORB, como o modelo concorrência e as prioridades de conexões e despacho.

4. Implementa-se um *serviço configurador* em cada servidor (*NodeApplication*) que utiliza as interfaces de cada serviço CORBA através de fachadas

[7]Essa garantia é dada através de recursos presentes nos ORB CORBA, como a tabela de despacho do adaptador de objetos (POA) e os interceptadores de chamadas.

do TAO [39] para o CIAO, por compatibilidade. Esse *configurador* usa as extensões dos descritores da OMG D&C (comentadas no item anterior).

Por fim, a especificação OMG D&C possui, dentre as tecnologias aqui avaliadas, o processo de implantação distribuída mais semelhante ao que este trabalho adota. Entretanto, a OMG D&C não define como instalar as dependências estáticas e não determina a semântica das dependências específicas de linguagem e plataforma. Assim, cada implementador da especificação pode instalar as dependências estáticas de uma forma própria (e possivelmente incompatível com outras implementações). Os documentos sobre o DAnCE, por exemplo, não explicam a abordagem usada nesse caso, o que torna possível o preenchimento das descrições XML com valores sintaticamente válidos mas sem a semântica necessária. Até o momento da escrita deste trabalho, o DAnCE é a única implementação da OMG D&C disponível publicamente.

A Tabela 2.4 resume a classificação da OMG D&C considerando o uso do DAnCE segundo os critérios propostos.

Critério Classificatório	**Avaliação**
Controle da Implantação	dinâmico por interface padrão
Suporte à Composição	projeto, implantação
Repositório de Implementações	**acesso:** remoto **uso:** estático **fase:** implantação, execução
Abstração da Distribuição	concreto
Modelo de Dependências	paramétricas, estáticas
Abrangência Tecnológica	**linguagem:** C++ **acesso:** serviço próprio

Tabela 2.4: Classificação da OMG D&C incluindo o uso do DAnCE

2.3.5 ADAGe

Na Computação em Grade, as dificuldades de reconhecer os tipos de recursos, localizar, selecionar e mapear as tarefas do usuário normalmente são tratadas pelo *middleware* da grade. Entretanto, é cada vez mais complexo prover tais funcionalidades de forma transparente [17], uma vez que certas aplicações dependem de uma combinação de várias tecnologias como MPI, CCM e Globus. Assim, torna-se ainda mais complexo implantar essas aplicações em cenários heterogêneos e amplamente distribuídos.

Lacour et al. propõem um modelo genérico para descrição de aplicações conhecido como GADe (do Inglês, *Generic Application Description*) [17].

Os descritores genéricos são um conjunto de hierarquias de "entidades de computação" que possuem conexões entre si. São elas:

- *entidades de sistema* que representam a máquina física distribuída. Exemplos de valores dos descritores: linux, windows, i386, amd64.
- *entidades de processo* que compartilham a mesma *entidade de sistema* representando as instâncias das aplicações com espaços de endereçamento privados. Exemplos de valores dos descritores: jvm, mpiexec.
- *códigos* que são carregados numa mesma *entidade de processo* compartilhando o mesmo espaço de endereçamento virtual. Exemplos de valores dos descritores: nomes de arquivos .jar, .dll ou .so.

As conexões entre as entidades de sistema indicam com quais outros grupos de sistemas, processos e códigos cada sistema pode interagir, respeitando a semântica de cada tecnologia. Por exemplo, se duas entidades de sistemas estão conectadas e a aplicação é baseada em MPI então isso significa comunicar processos MPI, ou se é baseada em CCM significa conectar componentes CCM.

Os descritores genéricos, entretanto, são gerados automaticamente pela conversão de outros descritores específicos da tecnologia considerada na aplicação. Essa conversão é realizada através do uso da ferramenta de implantação ADAGE [17]. O ADAGE é extensível por *plugins* e cada *plugin* pode definir seu próprio formato de descrição e deve implementar a conversão para descritores genéricos no formato GADE. Atualmente existem *plugins* disponíveis para diversas tecnologias, entre elas: CCM, MPI, JXTA e redes P2P *Overlay* (iChord, Chimera, khashmir). O ADAGE vem sendo usado pelo *middleware* de grade conhecido como Grid'5000[8] que integra 5000 processadores distribuídos em 9 instituições dispersas geograficamente na França.

A unidade de implantação no ADAGE é um conjunto de descritores específicos e de arquivos com a implementação do software. Os arquivos com a implementação podem ser obtidos remotamente caso sejam acessíveis por um dos protocolos suportados no ADAGE, entre eles: HTTP, FTP e SFTP.

O ADAGE implementa uma arquitetura de implantação que se baseia em etapas de descoberta e seleção dos recursos computacionais, instalação remota e lançamento da aplicação distribuída [40]. Essa arquitetura é ilustrada na Figura 2.9. Essa arquitetura considera uma fase de planejamento automático da implantação que escalona um conjunto de recursos computacionais, mapeia cada elemento descrito em GADE para os recursos selecionados satisfazendo adequadamente as restrições impostas (sistema operacional, uso do processador, espaço em memória, etc.) e armazena os parâmetros de configuração que

[8] `http://www.grid5000.fr`

serão usados em tempo da inicialização da aplicação. O produto da fase de planejamento é um plano de implantação que será escalonado conforme os *escalonadores da implantação*, os quais são implementados como *plugins*. Por fim, o plano de implantação é executado e para isso as unidades de implantação são instaladas nas máquinas remotas, os processos do sistema operacional são criados e configurados. Dessa forma, completa-se o início da aplicação e gera-se um relatório final. É interessante observar que o ADAGE define o planejamento em termos das descrições genéricas e, portanto, elimina a complexidade relacionada a implementar planejadores específicos para cada tecnologia.

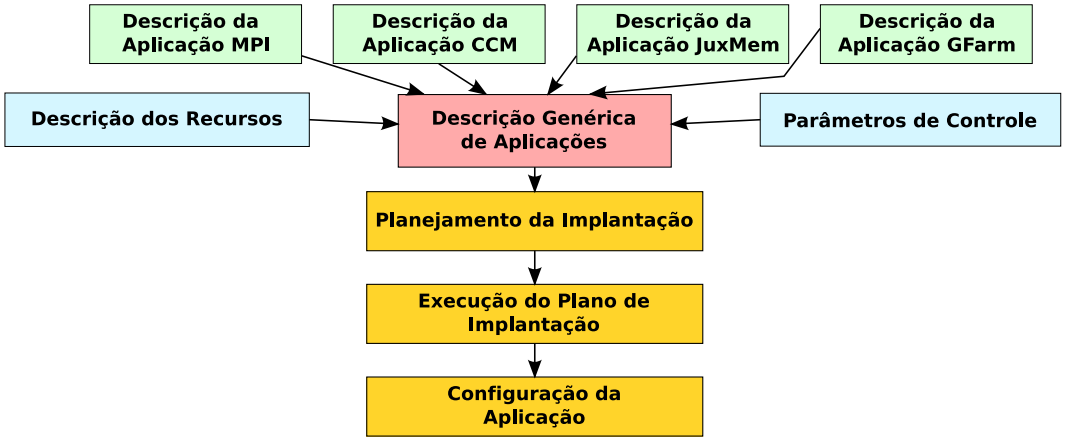


Figura 2.9: Arquitetura do ADAGe (fonte: [17])

Por outro lado, o ADAGE apresenta algumas limitações. O planejamento automático da implantação é tal que a implantação ocorre de forma estática, ou seja, uma vez gerado o plano de implantação ele não pode ser modificado até que a aplicação pare de executar. O ADAGE não define em sua arquitetura a noção de repositório onde as unidades de implantação pudessem estar armazenadas para facilitar o reuso por outros desenvolvedores. Além disso, o ADAGE não provê uma solução para implantar dependências estáticas das aplicações. Ainda que o ADAGE não incorpore um modelo de componentes de software, este trabalho considera que ele gerencia dependências paramétricas das aplicações, pois, o ADAGE permite que uma instância de aplicação estabeleça uma conexão a outra instância (normalmente em uma máquina diferente) de uma segunda aplicação durante a implantação. Para isso, basta que a tecnologia em questão possua essa semântica e que o *plugin* da tecnologia indique como se estabelece essa conexão.

Por fim, a Tabela 2.5 resume a classificação do ADAGE segundo os critérios propostos.

Critério Classificatório	Avaliação
Controle da Implantação	estático
Suporte à Composição	projeto
Repositório de Implementações	–
Abstração da Distribuição	abstrato
Modelo de Dependências	paramétricas
Abrangência Tecnológica	**linguagem:** qualquer **acesso:** submissão de tarefas

Tabela 2.5: Classificação do ADAGe

2.3.6 CoRDAGe

Em tempo de projeto ou implantação é difícil prever as reais necessidades de recursos físicos para executar aplicações científicas (numéricas de longa execução) ou serviços de compartilhamento de dados, num contexto de computação em grade [41]. Nesses cenários torna-se interessante dispor de facilidades para *expandir* e *retrair* a topologia da rede em tempo de execução. Isso traz implicações diretas nas metodologias de implantação distribuída, uma vez que estratégias estáticas[9] de implantação, como aquelas usadas no ADAGe [17], não conseguem prover tais facilidades. Nessa condição, os usuários são obrigados a reconfigurar o plano manualmente e reiniciar toda execução da aplicação.

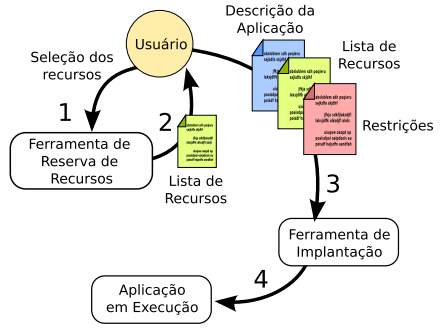
Cudennec et al. introduzem os conceitos de re-implantação (do Inglês, *re-deployment*) e co-implantação (do Inglês, *co-deployment*) e disponibilizam a ferramenta CoRDAGe [41], que implementa tais conceitos e reusa o ADAGe para implantar as aplicações. Re-implantação significa que a ferramenta de implantação deve estar preparado para concretizar mudanças (expansão ou retração) no plano de implantação assim que a aplicação esteja implantada e em execução. Co-implantação significa que a ferramenta de implantação precisa implantar e gerenciar aplicações que, mesmo sendo de tecnologias diferentes, imponham restrições[10] de funcionamento conjunto. O modelo de restrições no CoRDAGe permite restrições temporais - qual tecnologia deve ser implantada antes - ou de co-alocação - em quais máquinas ambas aplicações devem estar executando simultaneamente.

A unidade de implantação no CoRDAGe é do mesmo tipo que aquela utilizada no ADAGe, graças ao reuso dessa ferramenta. Entretanto, existe uma mudança na interação dos usuários com a ferramenta de implantação que é ilustrada na Figura 2.10. Diante dessa nova forma de interação, o usuário não lida diretamente com a ferramenta real de implantação, no caso o ADAGe.

[9]Na bibliografia usa-se o termo em Inglês *one-shot deployments*.
[10]No ADAGe já havia o conceito de restrições (do Inglês, *constraints*), embora estáticas.

Agora o usuário usa o CoRDAGe que vai se responsabilizar por selecionar os recursos (da *reservation tool*) e por atualizar os planos de implantação (na *deployment tool*). O CoRDAGe usa o protocolo XMLRPC[11] para interagir diretamente com as aplicações e concretizar as operações relacionadas à re-implantação e co-implantação.

2.9(a): Interação original apenas com ADAGe



2.9(b): Nova interação usando o CoRDAGe

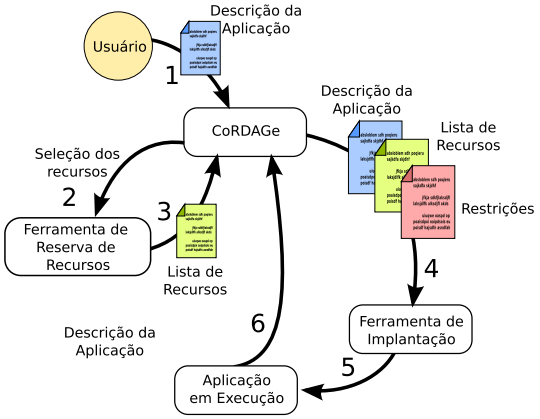


Figura 2.10: Alteração da perspectiva do usuário com o CoRDAGe (fonte: [41])

Internamente, o CoRDAGe adota uma representação própria da aplicação em forma de árvores lógicas (do Inglês, *logical tree*) para gerir a aplicação, expandindo ou retraindo seu uso de recursos. Cada tipo de aplicação deve disponibilizar um tratador capaz de construir tais árvores. Esse deve ser escrito em C++, estendendo o *framework* do CoRDAGe. Assim, o núcleo do CoRDAGe consegue realizar certas operações da implantação como: *register, deploy, add_sub_app, discard, terminate*. Os recursos da grade também possuem uma representação hierárquica baseada em proximidade na rede e nas informações oriundas dos serviços de informação da grade (do Inglês, *Grid Information Service*). Exemplos de informações acessíveis pelo *middleware* da grade são: carga da máquina, quantidade de memória disponível, capacidade de processamento e banda de rede.

Dessa forma, a solução para implantação distribuída do CoRDAGe é uma das mais interessantes dentre as analisadas neste trabalho, pois considera a implantação como um processo continuado que não termina quando a aplicação entra em execução. O CoRDAGe permite um gerenciamento dinâmico da implantação ao longo da sua execução. Por outro lado, as mesmas críticas apresentadas na Seção 2.3.5 sobre o ADAGe com relação às dependências estáticas e inexistência de repositório são aplicáveis ao CoRDAGe pois ele

[11]http://www.xmlrpc.com/spec

delega ao ADAGE a concretização da implantação. Por fim, a Tabela 2.6 resume a classificação do CORDAGE segundo os critérios propostos.

Critério Classificatório	**Avaliação**
Controle da Implantação	dinâmico por interface padrão
Suporte à Composição	projeto,implantação,execução
Repositório de Implementações	–
Abstração da Distribuição	abstrato
Modelo de Dependências	paramétricas
Abrangência Tecnológica	**linguagem:** qualquer **acesso:** submissão de tarefas

Tabela 2.6: Classificação do CoRDAGe

2.3.7 Fractal e DeployWare

FRACTAL é um modelo de componentes hierárquico e reflexivo [12]. Por ser hierárquico, o modelo de FRACTAL permite componentes compostos (*composite*) e componentes compartilhados (*shared*). Por ser reflexivo, FRACTAL permite que a estrutura interna possa ser explícita e controlada por interfaces bem definidas. FRACTAL permite diferentes níveis de controle sobre suas capacidades reflexivas.

Um componente FRACTAL é uma entidade de execução com uma identidade e pode oferecer uma ou mais interfaces. As interfaces são o ponto de entrada no componente. Podem existir interfaces de servidor (semelhante a *facetas*) para disponibilizar métodos a terceiros, ou de cliente (semelhante a *receptáculos*) para invocar métodos em outros componentes.

Para facilitar o entendimento, normalmente o componente FRACTAL é definido como uma membrana que oferece interfaces para inspecionar e reconfigurar seus elementos internos. Além da membrana, um componente FRACTAL contém um conjunto finito de outros componentes (chamados subcomponentes). A membrana possui as interfaces internas e externas, sendo que as internas só podem ser utilizadas por subcomponentes que façam parte do conteúdo da membrana. A membrana oferece interfaces de *controladores* para gerenciar o ciclo de vida do componente, fornecer acesso à introspecção e a composição entre componentes. Os controladores podem ser usados para, por exemplo, suspender, salvar o estado e resumir as operações dos subcomponentes. Os controladores ainda podem agir como interceptadores sobre os métodos requisitados no componente ou solicitados por ele.

A composição entre componentes FRACTAL pode ser feita de forma explícita através das *ligações* (do Inglês, *bindings*) que podem ser estabelecidas

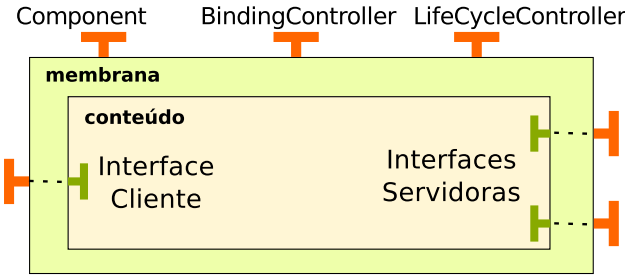


Figura 2.11: Modelo de componentes Fractal

em tempo de implantação. A comunicação de componentes Fractal depende das interfaces estarem interligadas. Portanto, existem dois tipos de ligações: (i) *primitivas*, que representam a ligação entre interfaces clientes e servidoras num mesmo espaço de endereçamento através da simples troca de referências entre as entidades da linguagem (ponteiros em C++ ou referências em Java); (ii) e *compostas*, que são componentes Fractal responsáveis por mediar a comunicação seja local ou remota.

A implantação de componentes Fractal pode ser conduzida pelo *framework* DeployWare [6]. O DeployWare é construído usando componentes Fractal e segue motivações semelhantes ao ADAGe, propondo-se a implantar aplicações desenvolvidas em diferentes tipos de tecnologias. Nesse sentido, o DeployWare implementa um *metamodelo* que representa conceitos da implantação distribuída, na tentativa de mascarar a heterogeneidade do software para o usuário. O DeployWare simplifica a descrição de dependências entre as aplicações, adota uma etapa de validação estática (com base nas entidades do metamodelo) e disponibiliza um conjunto de componentes Fractal que implantam aplicações baseadas em MPI, OpenCCM, J2EE (Glassfish, JOnAS, JBoss) e SCA.

O DeployWare combina (i) uma linguagem específica a domínio (do Inglês, *DSL*) para associar as entidades concretas da implantação ao metamodelo; (ii) uma máquina virtual que interpreta o metamodelo; e (iii) uma ferramenta gráfica para monitorar e gerenciar, em tempo de execução, as aplicações implantadas. As principais entidades do metamodelo usado no DeployWare estão ilustradas na Figura 2.12. As *personalities* é a representação do software conforme a tecnologia específica e são formadas por *software types* que, por sua vez, representam os artefatos a serem implantados, podem indicar várias *procedures* e várias *dependencies*. Uma *procedure* representa um comando que é executado durante a implantação. As *dependencies* representam outros *software types* dependentes.

O usuário da implantação usa as diversas *personalities* disponibilizadas pela máquina virtual do DeployWare para descrever um plano de im-

plantação. Um plano é exemplificado no Código 2.5 onde representa-se o descritor da implantação de um componente *SC*, ilustrado na Figura 2.13. Na prática, as *personalities* são entidades em nível de metamodelo que representam componentes FRACTAL pré-existentes constituintes do DEPLOYWARE. Flissi e Merle apresentam uma máquina virtual, conhecida como FDF (*Fractal Deployment Framework*), que instancia os componentes FRACTAL, semelhante ao da Figura 2.13, e executa o plano de implantação [15].

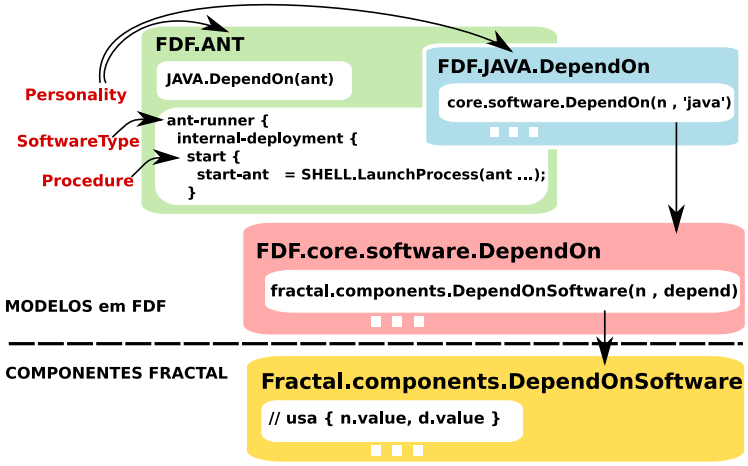


Figura 2.12: Relações entre metamodelos, modelos FDF e Fractal

Código 2.5: Descrição da implantação de um componente FRACTAL

```
1  MyDESC.SC = FDF.Software { // descrição de um componente SC
2    properties = {
3      file = PARAM.ARCHIVE;   // PARAM é uma 'personality' interna do FDF
4      home = PARAM.HOME;
5      port = HTTP.PORT(9000);// HTTP também é outro 'personality' interna
6    }
7    install = {               // SHELL e TRANSFER também são 'personalities'
8      upload = TRANSFER.Upload(#[file]); // uso da propriedade 'file'
9      make = SHELL.Execute(make);
10   }
11   start = {
12     ping = SHELL.Execute(ping,#[host]);// uso da propriedade 'host'
13     ...
14   }
15   dependencies = {
16     MyDESC.SA; MyDESC.SB; // depende de dois outros componentes SA e SB
17   } }
```

Originalmente, a especificação de FRACTAL não define o conceito de unidades de implantação [12]. O processo de implantação proposto no DEPLOYWARE se inicia a partir dos descritores de implantação da aplicação. Os descritores de implantação (ou plano de implantação) são responsáveis por indicar como acontece a obtenção da implementação da aplicação e como ela entra em execução. Em princípio, o preenchimento desses descritores pode ser facilitado se as *personalities* da tecnologia fornecerem facilidades adicionais para automatizar a instalação e o lançamento.

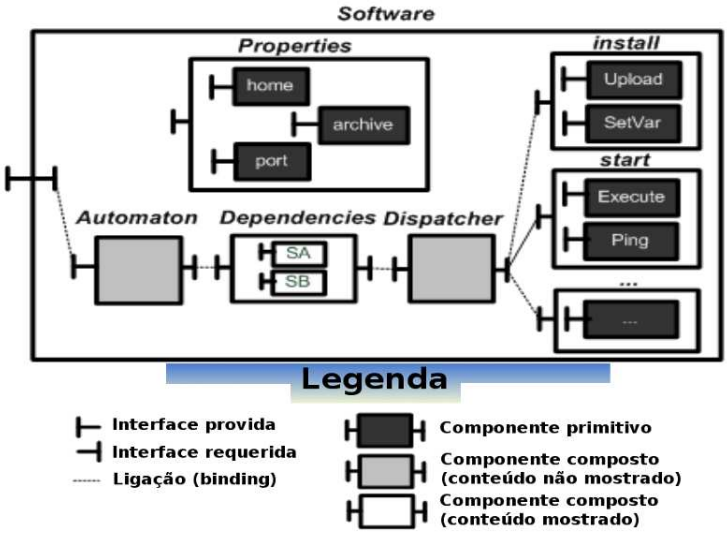


Figura 2.13: Componente Fractal gerado pela descrição FDF (fonte: [6])

A máquina virtual do FDF pode precisar gerenciar uma enorme quantidade de conexões de rede para implantar uma aplicação. Portanto, o DeployWare permite o lançamento de várias máquinas virtuais do FDF na tentativa de escalar o próprio procedimento de implantação. Essa preocupação com a escalabilidade da própria implantação é fundamental e não foi comentada nos outros trabalhos estudados, até então.

Por outro lado, ainda que haja implementações de Fractal para C++, o DeployWare só implanta componentes Fractal implementados em Java. Um dos motivos para isso é que Fractal não provê um mecanismo de comunicação entre componentes escritos em diferentes linguagens, como acontece, por exemplo, em CCM. O DeployWare também não permite descrever e implantar dependências estáticas, pois, como todo componente Fractal precisa ser programado em Java, considera-se que o pacote de um componente é, na verdade, um arquivo JAR. Os autores do DeployWare indicam que é possível implantar diferentes tecnologias como OpenCCM e MPI. Para isso, devem existir *personalities* em FDF que as representem e, consequentemente, componentes Fractal implementados em Java que implementem as ações de lançamento e execução. Ainda outro problema, é que não se usa repositórios de implementações de componentes e, assim, as *personalities* de uma dada tecnologia precisam estar previamente disponíveis em todas as máquinas físicas gerenciadas pelo FDF.

Por fim, a Tabela 2.7 resume a classificação do Fractal considerando o uso do DeployWare segundo os critérios propostos.

Critério Classificatório	**Avaliação**
Controle da Implantação	dinâmico por acesso direto
Suporte à Composição	projeto,implantação
Repositório de Implementações	–
Abstração da Distribuição	concreto
Modelo de Dependências	paramétricas
Abrangência Tecnológica	**linguagem:** Java **acesso:** serviço próprio

Tabela 2.7: Classificação do Fractal com uso do DeployWare

2.4 Considerações finais

Após esse estudo aprofundado dos diversos trabalhos relacionados à implantação distribuída, é possível afirmar que nenhum trabalho além da especificação do CCM [10] e D&C [14] tratam o uso e instalação de dependências estáticas. Contudo, mesmo nesses não é disponibilizada uma forma sistemática e automatizada de instalar tais dependências nas diversas máquinas remotas. Considera-se que o programador precisa agrupar todas as dependências no pacote do componente ou que o administrador deve garantir manualmente que elas estejam instaladas. Nem o CCM, nem a D&C e nem suas implementações (OpenCCM e DAnCE) possuem estratégias de implantação automática tão robustas como aquelas apresentadas no ADAGe [17] e no CoRDAGe [41]. Além disso, apenas a D&C e o DAnCE usam repositórios de componentes e, mesmo assim, sem permitir a publicação ou a revogação dinâmica dos componentes publicados. Uma visão geral da classificação entre os trabalhos relacionados é apresentada na Tabela 2.8.

Este trabalho baseia-se nessas observações para propor uma infraestrutura completa para implantação distribuída, com suporte a dependências estáticas para componentes multi-versão, multi-linguagem e multi-plataforma. Essa infraestrutura promove o reuso de repositórios de implementações em todas as fases do ciclo de vida (projeto, implantação e execução), como é o ideal [24]. Essa infraestrutura permite o planejamento da implantação através de níveis de detalhamento diferenciados que possibilitam o uso de estratégias automáticas, garantindo, ao mesmo tempo, maior controle da distribuição quando o administrador considerar necessário.

O Capítulo 4 explica o suporte a dependências estáticas e o Capítulo 5 detalha todas as funcionalidades da infraestrutura proposta.

Critério Classificatório	EJB	OpenCom	OpenCCM	DAnCE	ADAGe	CoRDAGe	DeployWare
Controle da Implantação	dinâmico por acesso direto	dinâmico por acesso direto	dinâmico por interface padrão	dinâmico por interface padrão	estático	dinâmico por interface padrão	dinâmico por acesso direto
Suporte à Composição	projeto	implantação, execução	projeto, implantação	projeto, implantação	projeto	projeto, implantação, execução	projeto, implantação
Repositório de Implementações	–	–	–	**acesso:** remoto **uso:** estático **fase:** implantação, execução	–	–	–
Abstração da Distribuição	manual	manual	concreto	concreto	abstrato	abstrato	concreto
Modelo de Dependências	–	paramétricas	paramétricas, estáticas	paramétricas, estáticas	paramétricas	paramétricas	paramétricas
Abrangência Tecnológica	**linguagem:** Java **acesso:** serviço próprio	**linguagem:** C, C++ e Java **acesso:** serviço próprio	**linguagem:** Java **acesso:** serviço próprio	**linguagem:** C++ **acesso:** serviço próprio	**linguagem:** qualquer **acesso:** submissão de tarefas	**linguagem:** qualquer **acesso:** submissão de tarefas	**linguagem:** Java **acesso:** serviço próprio

Tabela 2.8: Classificação de todas tecnologias segundo os critérios propostos