2
Componentes de Software

A proposta de componentes de software surgiu no final dos anos 60 como uma tentativa de suprir a necessidade de construir software de forma rápida e barata [12]. Entretanto, foi o modelo de orientação a objetos que teve maior aceitação na tentativa de fornecer mecanismos para tornar mais eficiente o desenvolvimento de software, em especial através de uma melhor modularização e possibilidade de reutilização. Porém, o modelo orientado a objetos ainda apresenta deficiências que dificultam a modularização e reutilização [7]. Dessa deficiência, a proposta de componentes de software ressurge, não para substituir, mas sim complementar o modelo orientado a objetos, assim como outros paradigmas de programação. Neste capítulo, serão abordados alguns dos principais aspectos da abstração de componentes de software, assim como suas vantagens e desvantagens. Além disso, é feita uma apresentação do modelo de componentes utilizado pelas ferramentas apresentadas nesse trabalho.

O conceito de componente de software é definido como uma abstração para construção de sistemas que pode ser adaptada sobre diversos paradigmas diferentes, como a orientação a objetos ou o paradigma procedural, entre outros. Entretanto, o modelo de componentes incorpora muitos dos conceitos do paradigma de objetos, como encapsulamento e separação de interface e implementação, mas além disso também torna explícitos conceitos como dependências e conexões entre componentes.

Componente de software pode ser definido como uma unidade de composição com interfaces bem definidas e dependências de contexto explícitas, além de poder ser independentemente implantado e estar sujeito a composição por terceiros [4]. Entretanto, as características de um componente de software não são bem consolidadas. Em especial, o tamanho ou a granularidade de um componente não é exata. Em alguns trabalhos o conceito de componente é recursivo através da idéia de subcomponentes que compõem componentes maiores [13, 14].

Tendo em vista a idéia de reuso, o tamanho do componente, medido pela quantidade de recursos implementados, tem influência primordial. Em relação às funcionalidades de um componente, em [15] é definido que um componente deve fornecer recursos suficientes para que valha a pena reutilizá-lo, entretanto, para que o componente possa ser amplamente aplicável, essas funcionalidades devem ser logicamente relacionadas de forma a compor um conjunto de funcionalidades coeso. Já em relação às suas dependências, quanto me-

nos dependências os componentes apresentarem mais robustos e pesados eles serão, gerando redundâncias de implementação e diminuindo a possibilidade de reutilização de outros componentes. Contudo, quanto mais funcionalidades forem delegadas a outros componentes maiores serão suas dependências, dificultando sua utilização. Portanto, apesar de não ter uma medida certa e clara, é necessário definir adequadamente as funcionalidades de cada componente, definindo assim toda a arquitetura do sistema. Caso essa arquitetura deva ser modificada posteriormente, pode ser necessário fazer a manutenção de grande parte do sistema.

Outro conceito importante do modelo de componentes é a separação entre a definição do componente (*i.e.*, suas interfaces, serviços e dependências) e sua implementação. Componentes são manipulados como caixas-pretas, ou seja, são manipulados com base exclusivamente na sua definição. A definição de um componente especifica um conjunto de *conectores* através dos quais é possível acessar os serviços do componente e fornecer os recursos esperados pelo componente, definidos como suas dependências. A construção de um sistema baseado em componentes é feita estabelecendo conexões entre componentes através da ligação de seus conectores, de forma que as dependências de um componente sejam supridas pelos serviços oferecidos por outro.

Há diversos modelos de componentes de software que são utilizados para definir e construir componentes. Cada modelo de componentes de software define os tipos de conectores que os componentes podem fornecer. Por exemplo, num modelo de componentes baseado em eventos, conectores são canais de eventos através dos quais é possível enviar eventos solicitando serviços ou receber eventos que fornecem os serviços solicitados. Inicialmente, a ênfase dos modelos de componentes estava apenas na definição dos serviços fornecidos por um componente e geralmente não havia mecanismos para definir as dependências de um componente, como é o caso do modelo de Java Beans e CORBA.

Mais recentemente, a OMG propôs um modelo de componentes baseado na arquitetura de comunicação de objetos distribuídos CORBA [16], que permite a interoperabilidade entre diferentes plataformas e linguagens de programação. Esse modelo, denominado Modelo de Componentes de CORBA (*CORBA Component Model* – CCM) [11], foi elaborado com base na experiência de outros modelos de componentes distribuídos, em particular o modelo EJB. Por essa razão, o modelo CCM é um dos modelos de componentes comerciais mais completos atualmente. Além disso, por ser baseado na arquitetura CORBA, o modelo CCM também permite a construção e a interoperabilidade de componentes em diferentes plataformas e linguagens de programação. Por essas razões, o modelo CCM é adotado como o modelo de componentes utilizado neste trabalho. A seguir é feita uma breve explanação das principais limitações da arquitetura CORBA que levaram ao surgimento do modelo CCM, assim como uma descrição do modelo CCM. O apêndice A apresenta uma breve descrição das principais características da arquitetura CORBA.

2.1
Limitações da Arquitetura CORBA

O uso do modelo de objetos de CORBA na construção de aplicações levou à identificação de limitações da arquitetura [17], que posteriormente deram origem a elaboração do modelo de componentes de CORBA, que será visto na seção 2.2. Muitas dessas limitações resultam no desenvolvimento de objetos altamente acoplados, com implementações pouco padronizadas, que muitas vezes são difíceis de projetar, reutilizar, implantar e estender. A seguir, são apresentadas algumas dessas limitações.

Mecanismo de implantação de componentes A arquitetura CORBA não define nenhum mecanismo para distribuir, instalar, instanciar e ativar implementações de objetos. Em outras palavras, o método de implantação de objetos é feito de forma não padronizada, fazendo com que os métodos utilizados no desenvolvimento das aplicações sejam complicados e não portáveis, especialmente na implantação de sistemas compostos por diversos objetos.

Suporte para padrões de programação de servidores Diversos recursos tornam a arquitetura CORBA bastante flexível, mas muitos desses recursos se mostram complexos de serem utilizados. Apesar de existirem padrões recorrentes de configuração desses recursos, não há mecanismos que permitam simplificar o desenvolvimento através da utilização desses padrões de configuração. Como exemplo desses recursos, temos o Adaptador de Objetos Portátil (*Portable Object Adaptor* - POA) que define diversas políticas de como as chamadas aos objetos são entregues à sua implementação. Apesar de existirem certas configurações do POA muito comuns, não há facilitadores para realizar essas configurações.

Mecanismos de extensão de funcionalidade Uma limitação mais séria da arquitetura CORBA é a fragilidade do mecanismo de herança múltipla de interfaces sem sobrecarga de métodos de CORBA, que é o único mecanismo de extensão de funcionalidades disponível na arquitetura. Por exemplo, não é possível estender um servidor herdando de duas interfaces que definam operações com o mesmo nome, pois não há mecanismos para resolver conflitos de nomes entre duas interfaces herdadas. Adicionalmente, somente o mecanismo de herança múltipla não permite que um objeto implemente a mesma interface duas vezes, para fornecer implementações com características não funcionais diferentes. Além disso, não é possível que um objeto, que implemente várias interfaces diferentes através de herança, separe suas interfaces para que clientes diferentes tenham visões diferentes do mesmo serviço.

Disponibilidade de serviços de objetos A arquitetura CORBA define diversos serviços de objetos que auxiliam a construção de aplicações. Entretanto, a disponibilidade desses serviços no contexto de execução não é conhecida pela aplicação, nem há mecanismos padrões para ativar e

configurar esses serviços. Isso obriga as aplicações a utilizarem soluções próprias e pouco portáveis.

Mecanismos de gerência do ciclo de vida O ciclo de vida de objetos CORBA geralmente é controlado explicitamente pelos seus clientes seguindo soluções não padronizadas. Apesar da arquitetura CORBA definir um serviço de gerência de ciclo de vida, é necessário que o componente implemente um conjunto de interfaces auxiliares que permitam controlar o seu ciclo de vida.

2.2 O Modelo de Componentes de CORBA

O modelo de componentes de CORBA (CCM) [11] estende o seu modelo de objetos com o intuito de tratar alguns problemas deixados em aberto pela arquitetura CORBA. Para tanto, são definidos novos recursos e serviços que permitem que desenvolvedores de aplicação possam implementar, gerenciar, configurar e implantar componentes de forma padronizada, facilitando a reutilização e a manutenção do sistema.

No modelo CCM, componentes são os elementos básicos de construção de um sistema. Esses componentes são interconectados através de conexões orientadas a interface, onde a comunicação se dá através de chamadas de operações numa determinada interface; e de conexões orientadas a eventos, onde a comunicação se dá através da emissão e recebimento de eventos. Os componentes são armazenados num pacote contendo sua implementação e a descrição de suas características. Esse pacote é então utilizado para implantar o componente num *servidor de componentes*, onde esse pode ser então instanciando e utilizado por clientes.

2.2.1 Declaração de Componentes

No modelo CCM as conexões entre os componentes são feitas através de conectores denominados *portas*. Portas são pontos de comunicação do componente, que podem ser orientadas a interface ou a eventos. O modelo CCM define quatro tipos de portas: *facetas* que são conectadas a *receptáculos* para estabelecer conexões orientadas a interface; e *fontes de eventos* que são conectados a *receptores de eventos* para estabelecer conexões orientadas a eventos (*i.e.*, canais de eventos). As portas definem os serviços oferecidos pelos componentes, por exemplo através de interfaces (faceta) ou de canais de emissão de eventos (fonte de eventos). Da mesma forma, as portas definem as dependências do componente, por exemplo definindo uma interface que é exigida pelo componente (receptáculo) ou um canal de recebimento de eventos (receptor de eventos). As portas de componentes CCM são vistas com mais detalhes na seção 2.2.3. Além de portas, os componentes CCM também

podem oferecer interfaces e atributos, da mesma forma como objetos CORBA convencionais. As interfaces oferecidas e os atributos de um componente são destinados primordialmente à configuração do componente, através dos quais o componente é adaptado às necessidades da aplicação. A figura 2.1 ilustra uma representação da estrutura dos componentes CCM.

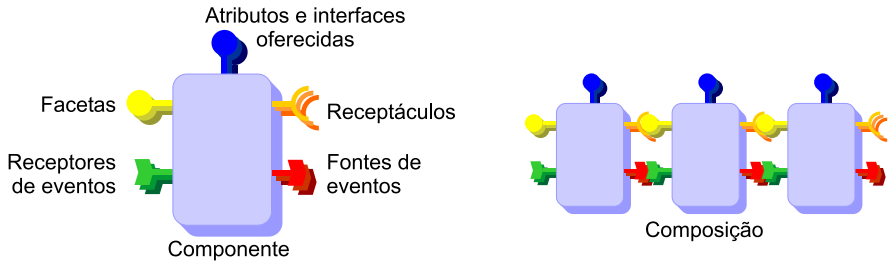

Figura 2.1: Estrutura de componentes CCM.

Os componentes CCM são descritos em IDL 3.0, que é uma extensão da linguagem de definição de interfaces de CORBA, que inclui novas estruturas para descrição de componentes, portas e outras estruturas relacionadas ao modelo de componentes CCM. A descrição de um componente também pode definir relações de herança, permitindo que o componente possa herdar interfaces oferecidas, atributos e portas de outro componente. Entretanto, cada componente só pode herdar de um único componente. A figura 2.2 mostra um exemplo de declaração de um componente em IDL 3.0.

```
1  component MeuComponente
2    supports InterfaceOferecida
3    : SuperComponente
4  {
5    attribute string                atributo    ;
6    provides  InterfaceDaFaceta     faceta      ;
7    uses      InterfaceDoReceptaculo receptaculo;
8    publishes EventoPublicado       publicador  ;
9    emits     EventoEmitido         emissor     ;
10   consumes  EventoConsumido       consumidor  ;
11 };
```

Figura 2.2: Um exemplo de definição de componente em IDL 3.0.

Componentes são declarados através da palavra component, de forma similar à definição de interfaces. A palavra supports define as interfaces oferecidas pelo componente. Herança de componentes é definida utilizando a mesma sintaxe da definição de interfaces, ou seja, através do operador : (dois pontos). Dentro da definição do componente, a declaração de atributos é feita através da palavra attribute, de forma similar à declaração de atributos em interfaces. As palavras provides, uses, publishes, emits e consumes são utilizados para declaração de portas e serão descritas com mais detalhes na seção 2.2.3.

Ao invés de definir regras para o mapeamento da definição de componentes para linguagens de programação, o modelo CCM define regras de mapeamento para IDL 2.3, gerando uma *IDL equivalente*. Essa nova IDL gerada é

constituída de um conjunto de interfaces que expõem todas as características e funcionalidades do componente (*i.e.*, interfaces oferecidas, atributos e portas) denominada *interface equivalente*. Dessa maneira, é possível que clientes que não utilizam o modelo CCM possam acessar componentes de forma transparente através de interfaces CORBA comuns.

Executores

Na terminologia CCM, a implementação de um componente é denominada *executor*. Os executores devem implementar interfaces locais definidas na IDL equivalente do componente, denominadas interfaces de *callback*. É através dessas interfaces que os serviços do componente são invocados. Cada definição de componente resulta na geração de duas interfaces de *callback* distintas, uma para cada um dos tipos de executores definidos no modelo CCM: executor monolítico e executor segmentado.

Executores denominados *monolíticos* são tratados como um único módulo, ou seja, como um único pedaço íntegro e completo. Isso implica que componentes com executores monolíticos são carregados e ativados como um todo. O modelo CCM também permite definir executores *segmentados*, que são executores divididos em segmentos que podem ser carregados e ativados independentemente. Por exemplo, quando é feita uma requisição através de uma das portas do componente, apenas o segmento que implementa essa porta é ativado para responder a requisição, enquanto o resto do componente pode permanecer inativo.

Contexto

O modelo CCM define um conjunto de interfaces que fornecem serviços ao executor do componente, denominado *interface de contexto*. Através dessas interfaces, o executor pode ter acesso às conexões em suas portas, assim como utilizar serviços de objetos da arquitetura CORBA, tais como localização, transação, eventos, persistência e segurança. A interface de contexto é implementada pelo *contêiner*, que é o elemento do modelo CCM que fornece um ambiente de execução para as implementações de componentes com serviços e recursos que simplificam o desenvolvimento de componentes. Os contêiners do modelo CCM serão vistos com mais detalhes na seção 2.2.5.

2.2.2

Eventos

Um novo recurso introduzido em IDL 3.0 é a definição de eventos através da palavra `eventtype`. Assim como a definição de componentes, esses eventos também são mapeados para IDL 2.3. Cada evento é mapeado para um *value*

type[1] e uma interface. O *value type* é utilizado para representar o evento propriamente dito. Por essa razão os eventos CCM são uma forma de *value type* mais restrito. Todos os *value types* que representam eventos CCM em IDL 2.3 devem ser uma especialização de um *value type* comum denominado ::Components::EventBase.

Além do mapeamento para *value type*, eventos também resultam na definição de uma interface em IDL 2.3 denominada <*nome do evento*>Consumer, que define a interface do consumidor do evento. Essa interface é constituída de uma única operação push_<*nome do evento*>, que recebe como parâmetro o *value type* representando o evento a ser consumido. Os eventos do modelo CCM são declarados em IDL 3.0 seguindo a seguinte forma:

```
eventtype <nome> : <base> {
    <membros de estado>
};
```

Caso a <*base*> do evento seja definida, ela pode tanto ser um outro evento ou um *value type*. A IDL equivalente resultante segue a seguinte forma:

```
valuetype <nome>
    : <base>, ::Components::EventBase {
    <membros de estado>
};

interface <nome>Consumer
    : ::Components::EventConsumerBase {
    void push_<nome> (in <nome> the_<nome>);
};
```


2.2.3 Portas

Como dito anteriormente, as portas de um componente definem os conectores através dos quais um componente pode ser interligado a outros componentes para formar um sistema. No modelo CCM são definidos dois tipos de conexões: conexões orientadas a interface, que são estabelecidas conectando uma porta que ofereça uma determinada interface, denominada *faceta*, e uma outra porta que utilize essa mesma interface, denominada *receptáculo*; assim como conexões orientadas a eventos, estabelecidas conectando uma porta que gere eventos de um determinado tipo, denominada *fonte de eventos*, e outra porta que consuma esses eventos, denominada *receptor de eventos*.

O modelo CCM não impõe nenhuma restrição quanto à cardinalidade das conexões, ou seja, é possível que uma faceta seja conectada a nenhum ou

[1] Recurso da arquitetura CORBA para permitir uma forma de passagem de objetos por valor.

a vários receptáculos simultaneamente e vice-versa. O mesmo se aplica aos conectores orientados a eventos (*i.e.*, fontes e receptores de eventos); ou seja, um mesmo receptor de eventos pode ser registrado em diversas fontes diferentes de um mesmo evento. Adicionalmente, o modelo CCM também permite que um componente possa ter mais de uma porta do mesmo tipo (*e.g.*, duas fontes de eventos que emitem o mesmo tipo de evento), desde que cada porta seja identificada por um nome diferente dentro de um mesmo componente. Note que essa flexibilidade dada pelas portas do modelo CCM funciona como um mecanismo de extensão, pois permitem que o componente possa incorporar novas implementações de uma mesma interface, ou fornecer novas interfaces sem interferir nas demais interações através de outras interfaces (*i.e.*, portas).

Facetas

Facetas são portas que oferecem uma determinada interface. Comumente, os serviços oferecidos pelo componente são disponibilizados através de um conjunto de facetas com interfaces distintas. Além disso, também é possível que um componente ofereça uma mesma interface através de portas distintas, mas com diferentes implementações e comportamentos. Facetas são implementadas por objetos que expõem interfaces CORBA comuns, podendo ser acessadas por clientes que não sigam o modelo CCM.

As facetas de um componente são definidas através da palavra provides dentro da definição do componente, definindo a interface oferecida pela faceta e o nome da faceta, da seguinte forma:

```
provides <interface> <nome>;
```

Cada faceta declarada na definição de um componente resulta numa nova operação na interface equivalente do componente na seguinte forma:

```
<interface> provide_<nome>();
```

A declaração de uma faceta também resulta na definição de uma nova operação na interface de *callback* do componente. Essa nova operação é utilizada para obtenção da implementação da faceta, ou seja, o executor da faceta. A nova operação adicionada a interface de *callback* do componente possui a seguinte forma:

```
CCM_<interface> get_<nome>();
```

Receptáculos

Conexões orientadas a interface são estabelecidas conectando uma faceta a um receptáculo, que é uma porta através da qual o executor do componente pode utilizar uma determinada interface. Os receptáculos de um componente representam suas dependências com demais componentes. Como os receptáculos utilizam interfaces CORBA comuns, é possível conectar objetos CORBA aos receptáculos de um componente CCM.

Os receptáculos de um componente são declarados através da palavra uses dentro na definição do componente, definindo a interface esperada e o seu nome. A menos que seja especificado o contrário, apenas um objeto pode ser acoplado a cada receptáculo. Entretanto é permitido declarar receptáculos que possam acoplar mais de um objeto simultaneamente, através da palavra multiple. A declaração de um receptáculo possui a seguinte forma:

```
uses [multiple] <interface> <nome>;
```

Cada receptáculo declarado na definição de um componente resulta em novas operações na interface equivalente do componente. Essas operações permitem conectar e desconectar interfaces ao receptáculo, assim como obter as referências das interfaces conectadas. As operações geradas dependem do tipo do receptáculo, ou seja, se o receptáculo permite conexão com um ou mais objetos. Para receptáculos simples (*i.e.*, que permitem apenas uma única conexão) as operações geradas apresentam a seguinte forma (por simplificação, as exceções lançadas pelas operações não são listadas.):

```
void connect_<nome>(in <interface> conxn);
<interface> disconnect_<nome>();
<interface> get_connection_<nome>();
```

A operação connect_<*nome*> permite conectar o objeto passado por parâmetro ao receptáculo. A operação disconnect_<*nome*> permite desconectar um objeto associado ao receptáculo, sendo que a referência desse é dada como valor de retorno da operação. A operação get_connection_<*nome*> permite obter uma referência ao objeto conectado ao receptáculo.

No caso de receptáculos que permitem mais de uma conexão, as operações geradas permitem desfazer uma conexão específica, assim como listar as conexões atuais. As operações geradas apresentam a seguinte forma:

```
Components::Cookie connect_<nome>(in <interface> conxn);
<interface> disconnect_<nome>(in Components::Cookie ck);
<nome>Connections get_connections_<nome>();
```

No caso de receptáculos múltiplos, ao conectar uma interface ao receptáculo através da operação connect_<*nome*>, é fornecido um identificador da conexão (*cookie*) como valor de retorno da função. Esse identificador pode então ser utilizado para desfazer a conexão. A operação disconnect_<*nome*>,

recebe como parâmetro o identificador da conexão que deve ser desfeita. Como no caso de receptáculos simples, o valor de retorno da operação de desconexão é uma referência ao objeto sendo desconectado. Através da operação get_connections_<*nome*>, é possível obter uma seqüência de descritores de conexão. Um descritor de conexão é uma estrutura contendo uma referência ao objeto conectado e um identificador da conexão.

O executor do componente obtém as referências das interfaces conectadas aos seus receptáculos através de um objeto de contexto, que é fornecido ao componente no momento de sua ativação. A interface oferecida pelo objeto de contexto é denominada *interface de contexto*. Para cada receptáculo declarado na definição do componente é gerada uma operação na interface de contexto, que permite obter as referências das interfaces conectadas ao receptáculo. O formato da operação na interface de contexto segue a mesma assinatura das operações get_connection_<*nome*> e get_connections_<*nome*> geradas na interface equivalente, de acordo com o tipo do receptáculo (*i.e.*, simples ou múltiplo).

Fontes de Eventos

O modelo CCM define dois tipos de portas para emissão de eventos. As portas denominadas *publicadores de evento* são portas através das quais eventos são enviados para um ou mais consumidores de eventos. Já as portas denominadas *emissores de evento* são portas através das quais eventos enviados são entregues a apenas um único consumidor de eventos.

Os publicadores de um componente são declarados através da palavra publishes dentro na definição do componente, definindo o tipo do evento publicado e o nome do publicador. A declaração de um publicador possui a seguinte forma:

publishes <*evento*> <*nome*>;

Da mesma forma, os emissores de um componente são declarados através da palavra emits dentro na definição do componente, também definindo o tipo do evento emitido e o nome do emissor. A declaração de um emissor possui a seguinte forma:

emits <*evento*> <*nome*>;

Cada fonte de evento declarada na definição do componente (*i.e.*, publicador ou emissor) acarreta a geração de novas operações na interface equivalente do componente. Essas operações permitem conectar e desconectar consumidores da fonte de evento. Cada consumidor de evento a ser associado às fontes de eventos deve fornecer a interface apropriada, ou seja, a interface de consumidor do tipo do evento emitido pela fonte de eventos, como descrito na seção 2.2.2. As operações geradas dependem se a fonte é um publicador ou um emissor de eventos. Para emissores de eventos as operações geradas permitem conectar e desconectar apenas um consumidor por vez e apresentam a seguinte forma:

```
void connect_<nome>(in <evento>Consumer consumer);
<evento>Consumer disconnect_<nome>();
```

No caso de publicadores, as operações geradas permitem registrar um consumidor na fonte de evento e obter um identificador da conexão, da mesma forma como as conexões em receptáculos múltiplos. Além disso, as operações também permitem remover um consumidor especificado pelo identificador de conexão. As operações geradas para um publicador de eventos apresentam a seguinte forma:

```
Components::Cookie subscribe_<nome>(in <evento>Consumer consumer);
<evento>Consumer unsubscribe_<nome>(in Components::Cookie ck);
```

A definição de fontes de eventos também resulta na geração de uma nova operação na interface de contexto que é fornecida ao executor do componente. Essa operação permite que o executor envie eventos através de suas fontes de eventos. Isso é feito independentemente do tipo da fonte, ou seja, se é um publicador ou emissor de eventos. A operação adicionada à interface de contexto do componente segue a seguinte forma:

```
void push_<nome>(in <evento> event);
```

Receptores de Eventos

Componentes CCM recebem eventos através de portas denominadas *receptores de eventos*. Através dessas portas, eventos são recebidos e então repassados ao executor do componente. Um receptor de eventos pode ser conectado a uma ou mais fontes de eventos de mesmo tipo de evento, estabelecendo canais de eventos entre componentes.

Os receptores de eventos de um componente são declarados através da palavra consumes dentro da definição do componente, definindo o tipo do evento consumido e o nome do receptor de eventos. A declaração de um receptor de eventos possui a seguinte forma:

```
consumes <evento> <nome>;
```

Cada receptor de evento declarado na definição do componente acarreta a geração de uma nova operação na interface equivalente do componente. Essa operação permite obter uma referência para um objeto consumidor, que representa o receptor de evento. Esse objeto fornece a interface do consumidor do evento de mesmo tipo do evento consumido pelo receptor, como descrito na seção 2.2.2. A operação gerada na interface equivalente do componente apresenta a seguinte forma:

```
<evento>Consumer get_consumer_<nome>();
```

A definição de receptores de eventos também resulta na geração de uma nova operação na interface de *callback*, que deve ser fornecida pelo executor do componente. Através dessa operação, os eventos recebidos através do receptor de evento são repassados ao executor do componente. As operações adicionadas à interface de contexto do componente seguem a seguinte forma:

```
void push_<nome>(in <evento> event);
```

Manipulação de Portas

Além das operações da interface equivalente, que permitem manipular as portas de um componente, o modelo CCM define um conjunto de interfaces que permitem manipular as portas de um componente de forma genérica. Através dessas interfaces é possível obter a lista de portas de um componente, assim como estabelecer conexões entre essas portas. Todos os componentes CCM oferecem essas interfaces de manipulação de portas, que são listadas a seguir:

Components::Nagivation Interface que fornece operações que permitem navegar a partir da referência de um componente para todas as suas facetas, ou seja, obter referências para suas facetas.

Components::Receptacles Interface que fornece operações que permitem manipular as conexões estabelecidas através dos receptáculos de um componente. Por exemplo, é possível obter a lista de receptáculos do componente, assim como conectar e desconectar objetos desses receptáculos. Além disso, através dessa interface, também é possível obter todos os objetos conectados aos receptáculos do componente.

Components::Events Interface que fornece operações que permitem manipular as portas de eventos de um componente, ou seja, fontes e receptores de eventos. Nessa interface estão definidas operações para obter a lista das portas de eventos, além de estabelecer e desfazer conexões nas portas. Ou seja, é possível obter consumidores que representam receptores de eventos, além de registrar consumidores em fontes de eventos.

Além das interfaces descritas acima, foi adicionada na versão 3.0 da especificação da arquitetura CORBA a operação get_component na interface ::CORBA:Object, que é herdada por todas as interfaces CORBA. Essa operação permite obter a referência do componente ao qual a interface pertence. No caso de interfaces de objetos comuns (*i.e.*, que não representem um componente CCM), essa operação sempre retorna uma referência nula. Através dessa operação, é possível navegar de uma faceta à referência do componente e então utilizar as interfaces de manipulação de portas para navegar a todas as demais facetas, assim como manipular as demais portas do componente.

2.2.4

Homes de Componentes

O modelo de componentes CCM define o conceito de *home de componente*. Através de um *home* é possível criar e recuperar componentes. Cada componente é associado a um único *home* a partir do qual o componente foi criado ou recuperado. Além disso, cada *home* é capaz de manipular apenas um tipo de componente.

O propósito dos *homes* é fornecer um mecanismo para gerenciar componentes. O *home* é um tipo especial e limitado de componente, cujo objetivo é gerenciar um conjunto de componentes de um mesmo tipo. Em especial, um *home* representa uma determinada implementação de um componente. Tipicamente, quando um pacote contendo uma implementação de componente é instalado num servidor de componentes, um *home* é criado a partir desse pacote para representar a implementação do componente, fornecendo mecanismos para criar e recuperar instâncias de componentes daquela implementação.

Homes de componentes são declarados através da palavra home, de forma similar à declaração de componentes, podendo inclusive definir interfaces oferecidas através da palavra supports. O tipo do componente gerenciado pelo *home* é definido através da palavra manages. Opcionalmente a declaração do *home* pode definir uma chave primária que será utilizada para identificar componentes persistentes. As chaves primárias são uma especialização do *value type* ::Components::PrimaryKeyBase com algumas restrições.

O *home* pode estender outro *home* base, do qual herdará suas operações. Nesse caso, a herança só é permitida se o componente gerenciado herda, direta ou indiretamente, do componente manipulado pelo *home* base. O corpo da declaração de um *home* de componentes pode definir operações, que são chamadas de operações explícitas. A declaração de um *home* de componente é dada pela seguinte forma:

```
home <nome> : <home base>
     supports <interfaces oferecidas>
     manages <nome do componente gerenciado>
     primarykey <tipo da chave primária>
{
     <operações explícitas>
};
```

Além das operações explicitamente definidas na declaração do *home*, outras operações são implicitamente definidas no *home* do componente. Caso o *home* não defina nenhuma chave primária, então apenas uma única operação é implicitamente definida, através da qual é possível criar novos componentes. A operação implicitamente definida no *home* apresenta a seguinte forma:

```
<componente> create()
```

No caso de *homes* de componentes com chave primária, outras operações são implicitamente definidas, além da operação para criação de componentes. As demais operações permitem consultar o valor da chave primária de um dado componente, além da obtenção ou exclusão de componentes já existentes identificados através da sua chave primária. As operações implicitamente definidas no *home* apresentam a seguinte forma:

```
<componente> create()
<componente> create(in <tipo da chave> key)
<componente> find_by_primary_key(in <tipo da chave> key)
void remove (in <tipo da chave> key)
<tipo da chave> get_primary_key(in <componente> comp);
```

À exceção das operações explícitas, toda implementação de *homes* de componentes é gerada automaticamente a partir da sua descrição em IDL, como será descrito mais adiante.

2.2.5 Contêiners

O contêiner é uma abstração que define um ambiente de execução protegido onde implementações de componentes são implantadas, criadas e executadas. Todas as interações entre a implementação de um componente e o mundo externo são intermediadas pelo contêiner. Dessa maneira, o contêiner é capaz de fornecer serviços e funcionalidades a implementações de componentes, de forma a minimizar ao máximo seu esforço de implementação. Além disso, através do conceito do contêiner é possível definir maneiras padronizadas para implantar, criar, ativar e fornecer serviços a implementações de componentes. As responsabilidades de um contêiner podem ser listadas como segue:

Criação de componentes O contêiner deve ser capaz de construir uma instância inteiramente funcional de um componente, a partir da sua implementação e da sua descrição fornecidas nos moldes definidos pela especificação CCM.

Ativação de componentes O contêiner deve ativar e desativar um componente de acordo com as políticas definidas em sua descrição.

Gerenciar conexões O contêiner deve controlar o estado das conexões do componente, assim como fornecer mecanismos para que conexões sejam estabelecidas e desfeitas. Esses mecanismos são fornecidos pela implementação das interfaces equivalentes dos componentes.

Prover acesso a serviços de objetos O contêiner deve prover acesso aos serviços de objetos — como eventos, transações, persistência, segurança, etc. — às implementações de componentes, fornecendo assim um mecanismo padrão de obtenção desses serviços.

A combinação das diversas políticas que definem os usos dos recursos fornecidos pelo contêiner definem as características de diferentes implementações de componentes. Cada conjunto de características distintas define uma categoria de componente. A especificação do modelo CCM 3.0 define quatro categorias de componentes, que conseqüentemente definem quatro tipos de contêiners, que são capazes de acomodar a maioria das implementações de componentes existentes atualmente.

Categorias de Componentes

A categoria do componente define basicamente os serviços oferecidos pelo contêiner através de uma interface de programação (*Application Program Interface* - API) e a maneira de manipular a implementação do componente. Essa maneira de manipulação da implementação do componente é definida pelo modelo de uso de CORBA (*CORBA Usage Model*) e uma política de gerência de ciclo de vida definidas pelo modelo CCM, que serão descritas adiante.

A categoria de um componente é definida de acordo com o tipo de estado apresentado pelo componente e como esse é visto pelo cliente. Um componente pode apresentar estado ou não (*stateless* ou *statefull*). Além disso, caso o componente apresente estado, esse estado pode ou não ser persistente. O mesmo ocorre com a identidade do componente, que pode distingüí-lo dos demais perante seus clientes, além de poder estar associada a um estado persistente que é utilizado para recuperar uma instância de componente. A partir dessas possibilidades, são definidas quatro categorias de componentes no modelo CCM, como descritas a seguir:

service Componente sem estado ou identidade. São componentes que fornecem comandos cuja duração não excede o tempo de processamento de uma requisição do cliente, que pode ser uma chamada de operação ou recebimento de um evento.

session Componente com estado e identidade, porém sem persistência. Esses componentes são distinguíveis dos demais e podem guardar estado, porém não perduram por mais do que uma transação do usuário. São componentes utilizados para modelar iteradores ou interações do cliente sem estado persistente.

process Componente com estado persistente, mas identidade não persistente. Esse tipo de componente não apresenta identificação através do qual o cliente possa recuperá-lo, entretanto apresenta um estado persistente. Na realidade, esse tipo de componente possui identidade persistente, mas que não é externalizada aos seus clientes (entretanto, a identidade pode ser externalizada através de operações específicas da aplicação). Esses componentes são utilizados para modelar processos de negócio (*e.g.*, aplicação de um empréstimo, criação de um pedido, etc.).

entity Componente com estado e identidade persistentes. A identidade persistente dessa categoria de componentes é externalizada através da as-

sociação do componente a uma chave primária. Esses componentes são utilizados para modelar entidades do negócio (*e.g.*, uma instutuição, um produto, uma conta bancária, etc.).

Modelo de Uso de CORBA

O modelo de uso de CORBA define basicamente a maneira de criação e utilização da implementação do componente. Cada modelo de uso define a forma que as requisições feitas a um componente são entregues à sua implementação. Por exemplo, no caso de um componente sem estado (*i.e.*, Serviço), as requisições feitas a um componente podem ser entregues a um único executor ou a um repositório de executores daquele componente. Por outro lado, no caso de um componente com identidade persistente (*i.e.*, Entidade), as requisições feitas a um componente devem ser entregues a um executor específico, que encarna o estado persistente do componente.

A partir dessas possibilidades são definidos três modelos de uso na especificação CCM, como descrito a seguir:

stateless São criadas referências de componentes transientes e um grupo de referências são mapeadas a um único executor. Adicionalmente, o contêiner pode instanciar um único executor ou manter um repositório de executores para atender todas as requisições de componentes. Esse repositório pode ser aumentado de acordo com o número de requisições, sendo que tal política é definida pelo contêiner.

conversational São criadas referências de componentes transientes e cada referência é mapeada a um único executor. O contêiner é responsável por criar uma instância do executor para cada componente criado e direcionar todas as requisições às referências do componente (*i.e.*, referências de interfaces oferecidas e portas) ao executor correspondente.

durable São criadas referências de componentes persistentes e cada referência é mapeada a um único executor que encarna o estado do componente associado a referência persistente. A recuperação do estado do componente associado a uma referência persistente pode ser feita pelo contêiner ou pelo próprio componente, denominadas respectivamente de persistência gerenciada pelo contêiner ou auto-gerenciada.

Políticas de Gerência de Ciclo de Vida

Apesar do modelo de uso definir a maneira de criação e utilização das implementações de um componente, a política de ativação das implementações é definida separadamente, através das políticas de gerência de ciclo de vida. São definidas quatro políticas de gerência de ciclo de vida, conforme descritas abaixo:

method Define que o executor do componente deve ser ativado antes do recebimento de cada requisição e desativado após o processamento da requisição. Nesse caso, os recursos do componente só permanecem alocados durante o processamento de uma requisição, entretanto isso acarreta em ativações e desativações muito freqüentes.

transaction Define que o executor do componente deve ser ativado antes do recebimento da primeira requisição dentro de uma transação e ser mantido ativado até o fim da transação. Nesse caso, os recursos do componente só permanecem alocados durante o contexto de uma transação.

component Define que o executor do componente deva ser ativado antes do recebimento da primeira requisição e seja mantido ativado até que o próprio componente solicite a desativação, que é feita após o processamento da requisição em que a solicitação foi feita. Nesse caso, os recursos do componente permanecem alocados até que o componente decida liberá-los.

container Define que o executor do componente deva ser ativado antes do recebimento da primeira requisição e seja mantido ativado até que o contêiner determine que o componente deve ser desativado, o que é feito fora do contexto de processamento de uma requisição. Nesse caso, os recursos do componente permanecem alocados até que o contêiner perceba a necessidade de desalocá-los.

Serviços de Objetos

Além de gerenciar as implementações dos componentes, através das diversas políticas e modelos definidos pela especificação CCM, o contêiner também é responsável por fornecer um meio padronizado de obter acesso a serviços de objetos descritos na arquitetura CORBA, como eventos, transação, persistência e segurança. O acesso a esses serviços pode ser feito de formas diferentes, tanto através da definição de políticas, como através da interface de programação do contêiner, que é fornecida às implementações dos componentes, ou até mesmo através dos próprios serviços CORBA, na maneira definida pela especificação.

Por questões de simplificação, neste trabalho não é considerado o suporte aos serviços de objeto fornecido pelo contêiner. Entretanto, todo o trabalho é conduzido prevendo uma futura inclusão desses recursos. Além disso, algum suporte restrito ao serviço de eventos é fornecido através das portas de eventos do modelo CCM.

Estrutura

Como dito anteriormente, o contêiner é um ambiente de execução, através do qual implementações de componentes interagem com o ambiente externo, comunicando-se com seus clientes e obtendo serviços. A estrutura do contêiner

pode ser descrita como um conjunto de interfaces que regem essas interações. Essas interfaces podem ser classificadas em três categorias:

Externa Interfaces através das quais os clientes interagem com o componente. São interfaces exportadas por objetos CORBA criados pelo contêiner e que representam o componente diante dos seus clientes. Nessa categoria estão as interfaces equivalentes dos componentes e dos *homes*.

Interna Interfaces através das quais o contêiner provê serviços e funcionalidades à implementação do componente. Como essas interfaces só são utilizadas pela implementação do componente, elas são interfaces locais comuns da linguagem de programação. Pela mesma razão, essas interfaces são implementadas por elementos da própria linguagem (*e.g.*, objetos), que são criados pelo contêiner. Nessa categoria estão as interfaces de contexto que são geradas de acordo com a definição de um componente e as interfaces através das quais o contêiner provê acesso aos serviços de objetos de CORBA.

Callback Interfaces através das quais o contêiner solicita serviços da implementação do componente, ou seja, solicita tratamento de requisições (*e.g.*, chamada de operação, recebimento de evento, etc.), ativação ou desativação, além de obter segmentos do executor ou inclusive solicitar à implementação do *home* uma nova instância da implementação do componente. Nessa categoria estão as interfaces de *callback* dos componentes e dos *homes*.

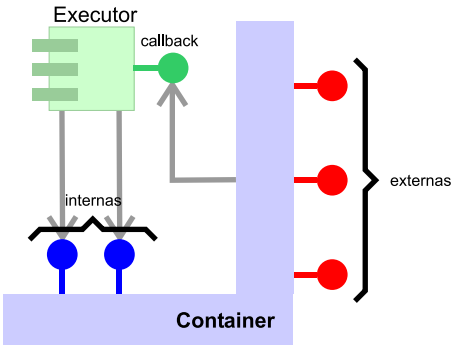

Figura 2.3: Interfaces do contêiner CCM.

Dentro de um único contêiner, podem ser instaladas diversas implementações de um mesmo componente, assim como de componentes diferentes. Por essa razão, cada contêiner deve ser capaz de se adequar à estrutura de cada componente (*e.g.*, interfaces oferecidas, portas, etc.). Entretanto, isso é difícil de ser feito em linguagens estáticas como C ou C++. Para contornar esse problema, parte das responsabilidades do contêiner é adicionada na implementação do componente através de código gerado pelo compilador de IDL 3.0.

Esse código gerado pelo compilador de IDL funciona como um adaptador ou cola do código escrito pelo desenvolvedor do componente, de forma que ele

possa ser inserido num determinado contêiner de uma implementação CCM. Como o código é automaticamente gerado e compilado junto com o código do desenvolvedor, o fato de parte das responsabilidades do contêiner estar inclusa na implementação do componente se torna transparente ao desenvolvedor do componente.

2.2.6

Implantação de Componentes

A especificação CCM também define mecanismos para implantação de componentes. Esses mecanismos consistem basicamente de um conjunto de descritores de pacotes de componentes e um *framework* de ferramentas e interfaces. Os descritores são utilizados para especificar as características dos componentes a serem implantados, como sua estrutura, categoria, políticas, entre outras. Ou seja, os descritores definem uma forma através da qual os componentes podem ser empacotados e distribuídos. Já o *framework* de ferramentas e interfaces define um método de implantação de pacotes de componentes.

Um pacote de software CCM é uma unidade de implantação de um componente ou sistema de componentes. Esse pacote é um arquivo compactado (formato ZIP) que contém uma descrição do pacote e um conjunto de arquivos. A especificação CCM define um conjunto de arquivos XML (*eXtensible Modeling Language*) que são utilizados para descrever pacotes, componentes e sistemas de componentes do modelo CCM. Esses arquivos são denominados *descritores*.

O modelo CCM define uma arquitetura para implantação de componentes constituída por um conjunto de interfaces, que são utilizadas por uma ferramenta de implantação utilizada pelo configurador do sistema para implantar componentes. O processo de implantação de componentes proposto consiste dos seguintes passos:

1. Através de interações com um usuário, identificar em que máquina cada *home* e cada componente será implantado;

2. instalar a implementação dos componentes nas máquinas correspondentes, de acordo com as definições do passo anterior;

3. instanciar homes e componentes seguindo o mapeamento definido no passo 1

4. por fim, se for a implantação de um pacote de componentes, as conexões entre eles devem ser feitas de acordo com o descritor de pacotes associado.