4
Java Objet Circuits Architecture

Java Object Circuits Architecture (JOCA) é uma implementação de circuitos de objetos utilizando a linguagem de programação Java (Arnold, 1996). Ela provê um ambiente de execução ou *runtime environment*, no qual um dado circuito de objetos é posto em funcionamento. O trabalho do programador se resume a descrever o comportamento dos dispositivos do circuito, e, em seguida, escrever um pequeno programa de teste que "monta" o circuito, instanciando os dispositivos e realizando as conexões apropriadas. No futuro, este programa de teste poderá ser substituído por ferramentas gráficas amigáveis.

No estágio atual do JOCA, não há suporte para composição: todos os dispositivos são atômicos. A computação que cada dispositivo realiza é trabalho de um número de *agentes de software*, internos ao dispositivo. O uso de agentes de software para descrição de dispositivos é perfeita, na medida em que ambos compartilham das mesmas características, como autonomia, encapsulamento e comunicabilidade.

A Ilustração 1 contém um diagrama parcial de classes do JOCA. Várias classes fazem parte da implementação interna da arquitetura e não ficam "visíveis" ao programador, tendo sido omitidas por este motivo.

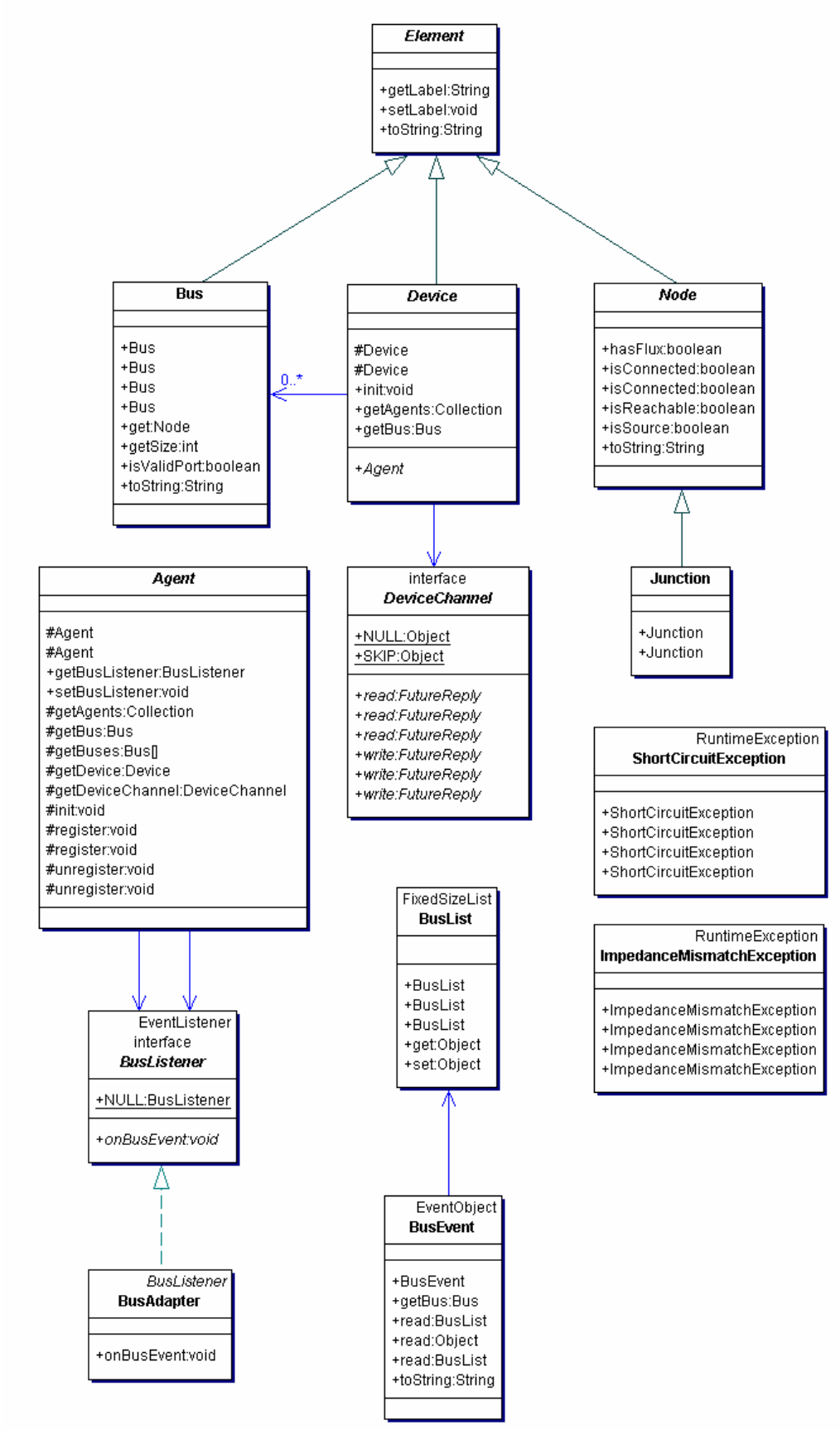

Ilustração 1 Diagrama de classes do JOCA. Algumas classes foram omitidas para facilitar a compreensão.

4.1.Nós e ramos

Um nó é uma instância da classe `Node`. Um ramo é uma instância da classe `Node.Branch`, interna à classe `Node`. A cada nó é associada uma única instância de `Node.Branch`, de tal forma que, se existir um caminho entre quaisquer dois nós, então ambos estão associados à mesma instância de ramo.

A representação explícita do ramo agiliza o processo de propagação do potencial por ocasião de uma escrita, que é feito em *O(1)*. Sem esta representação, a propagação teria que ser feita percorrendo cada nó e seus vizinhos, resultando numa complexidade de *O(n)*. Por outro lado, no caso de uma conexão ou desconexão de nós, o percurso individual de cada nó talvez seja necessário, pois, em ambos os casos, talvez seja preciso reconstruir os ramos. Como resultado, a complexidade destas mudanças é, em geral, *O(n)*[1].

A balança, no entanto, pende a favor desta abordagem, visto que, a menos que seja implementada evolução dinâmica, não é possível conectar ou desconectar nós de um sistema em execução. De qualquer forma, mesmo em um ambiente que permita evolução, presume-se que operações de escrita sejam mais comuns que alterações na estrutura do circuito.

4.2.Dispositivos

Dispositivos são entidades com poder computacional, instâncias da classe `Device`. Cada dispositivo possui uma lista finita e imutável de barramentos, especificados no momento de sua criação.

4.2.1.Canais de Dispositivo

As funcionalidades de leitura e escrita estão descritas na interface `DeviceChannel`, que modela o canal de comunicação de um dispositivo com seus barramentos. A cada dispositivo é associada uma única instância desta classe. O acesso a ela, entretanto, dá-se pela classe interna `Device.Agent`, descrita mais adiante.

---

[1] A reconstrução do ramo não é necessária numa conexão de nós se eles já pertencerem ao mesmo ramo. Neste caso, a complexidade é *O(1)*. A desconexão ocorre sempre em *O(n)*.

Por questões de conveniência, `DeviceChannel` disponibiliza uma variedade de métodos de leitura e escrita. Por exemplo, o método `FutureReply read( int index, int port )` inicia a operação de leitura do potencial de uma porta específica do barramento indicado. Já o método `FutureReply write( int index, int port, List values )` é a chamada da operação de escrita de uma lista de potenciais no barramento associado ao índice especificado, iniciando na porta dada com o primeiro elemento da lista, e prosseguindo até o fim da lista.

A assincronia da comunicação é visível, dado que todos os métodos de `DeviceChannel` retornam instâncias da classe `FutureReply`. Esta classe modela *respostas futuras* a uma mensagem: somente após ter sido concluída a respectiva operação – executada por outra linha de execução – será associada uma resposta à mensagem original. Esta resposta fica então disponível em `FutureReply.getReply()`. No caso de uma leitura, a resposta contém os potenciais das portas envolvidas na operação. No caso de uma escrita, a resposta é nula se a operação foi bem sucedida; porém, se ela de algum modo falha, uma exceção do tipo `MessageException` é levantada. Uma `MessageException` contém uma referência para a exceção que causou o erro da operação, que pode ser obtida através de `getCause()`. Uma causa possível é a ocorrência de um curto-circuito, representada pela classe `ShortCircuitException`.

Na classe `DeviceChannel` é definida uma constante, `SKIP`, que, ao ser utilizada como um dos parâmetros de uma escrita, aborta a operação na porta correspondente, e *somente nela*. Sua utilidade é visível, por exemplo, quando um certo dispositivo *d*, detentor de uma porta ativa *p*, pertencente a um barramento *b*, deseja atualizar *d* sem alterar o estado de *p*: basta que `SKIP` seja utilizado como parâmetro de escrita em *p*. De outra forma, o potencial atual de *p* teria que ser armazenado internamente por *d* e só então passado como parâmetro da operação, causando um inconveniente de programação.

Finalmente, é em `DeviceChannel` que é especificado o potencial nulo *N*, dado pela constante `NULL`. Se usado como um dos parâmetros de uma escrita, e a porta correspondente estiver ativa, então a porta é desativada, *desde que o dispositivo seja o detentor da porta*. Não sendo, haverá um curto-circuito.

Finalmente, se a porta correspondente estiver inativa, o uso de `NULL` é inócuo, sendo preferível a utilização de `SKIP`.

4.2.2.Agentes

O "comando" do dispositivo, por assim dizer, fica a cargo de entidades conhecidas como *agentes*. Um dispositivo, denominado *pai*, contém uma coleção destes agentes, que, por sua vez, pertencem a um único pai. Todo agente é instância da classe interna `Device.Agent`, onde estão definidos os métodos de acesso às funcionalidades do dispositivo. É importante ressaltar que todos métodos da classe são protegidos (possuem o modificador `protected`). A intenção é justamente restringir a operação do dispositivo aos agentes que o compõem.

Através da classe `Device.Agent` é disponibilizado o acesso ao canal do dispositivo pai, invocando-se `getDeviceChannel()`. Embora o acesso ao canal seja protegido, o canal em si é instância de uma interface pública, de forma que o agente pode passá-lo adiante, permitindo que operações de leitura e escrita sejam iniciadas por outras classes.

Um agente é capaz ainda de receber eventos de barramentos do dispositivo. Para isto, é necessário, inicialmente, cadastrar junto ao agente um *ouvinte*, instância da interface `BusListener`. Isto é feito invocando-se o método `setBusListener( BusListener busListener )`, com o ouvinte apropriado como argumento. Em seguida, o agente deve solicitar o serviço de envio de notificações de eventos de um determinado barramento, através do método `register( int index )`, passando como parâmetro o índice do barramento. A partir daí, sempre que o barramento em questão gerar um novo evento, ele será tratado diretamente pelo ouvinte do agente. Finalmente, a cessão do envio de eventos pode ser solicitada invocando-se o método `unregister( int index )`.

Note que o mecanismo de eventos possibilita a existência de agentes puramente reativos, isto é, aqueles que apenas reagem à ocorrência de um evento. Não fosse este mecanismo, um agente teria que periodicamente inspecionar um dado barramento (utilizando os meios convencionais de leitura da classe `DeviceChannel`) para detectar mudanças de potencial.

Num cenário genérico, temos dispositivos formados por comunidades de agentes que se comunicam entre si – diretamente – ou com os agentes de outros dispositivos – indiretamente – através de mensagens trocadas nos barramentos. A Ilustração 2 mostra esta situação.

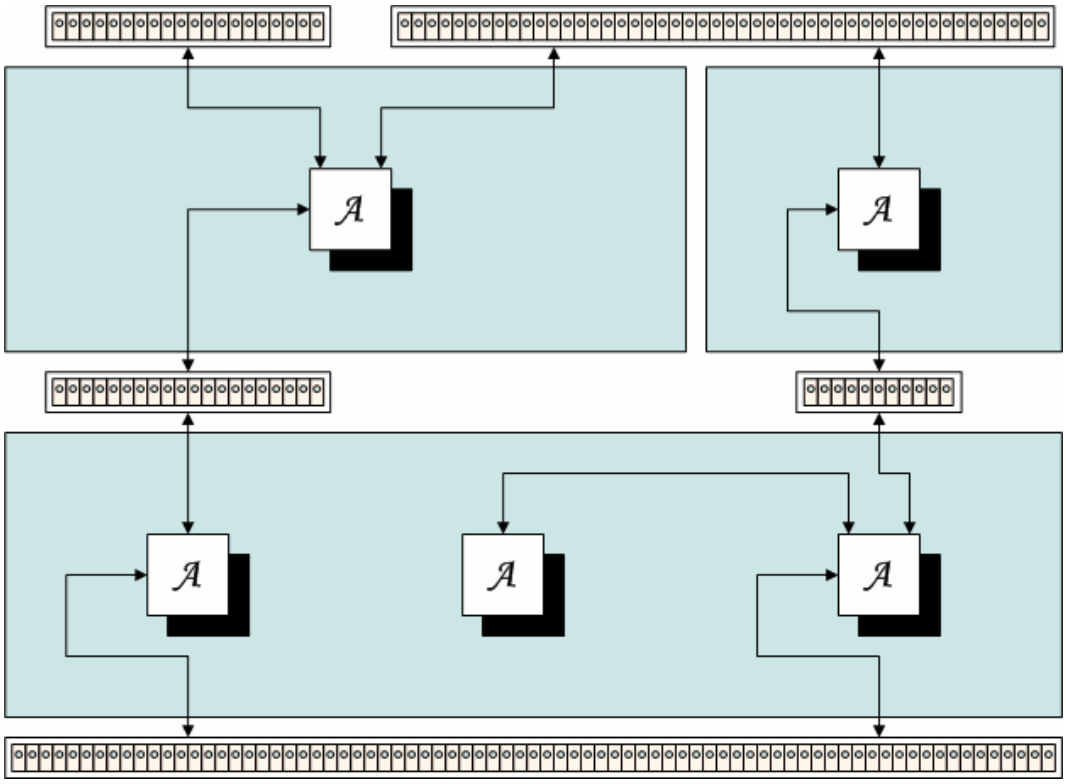

Ilustração 2 Representação interna de dispositivos e seus agentes.

4.3.Barramentos

Barramentos são estruturas semelhantes a listas indexadas, onde a cada posição da lista é associada uma porta. Barramentos possuem *largura* – isto é, o número de portas que contém. Este atributo é definido durante a sua criação e é imutável; caso não seja especificado, é arbitrada largura "infinita" (na prática, `Integer.MAX_VALUE`, o maior valor inteiro representável).

Barramentos são instâncias de `Bus`. Eles podem ser usados por dispositivos para a transmissão e recepção de dados, através de operações de leitura e escrita. Neste caso, a entidade ativa da comunicação é o dispositivo, o que nem sempre é conveniente. Por exemplo, para perceber uma mudança de potencial em alguma porta do barramento, o dispositivo teria que se valer de algum mecanismo de

polling, realizando continuamente operações de leitura no barramento, o que certamente é dispendioso computacionalmente.

Por este motivo, barramentos podem atuar como entidades ativas da comunicação, ao serem utilizados como geradores de eventos. Sempre que uma mudança de potencial ocorre em uma de suas portas, um evento é criado e despachado para todos os ouvintes previamente cadastrados. Um evento é uma instância de `BusEvent`, e armazena o estado completo do barramento tal como era no instante exato em que o evento ocorreu. Já um ouvinte é instância de `BusListener`, e implementa o método `onBusEvent( BusEvent busEvent )`, chamado sempre que um barramento lhe notifica sobre um novo evento.

4.4.Gerenciamento de recursos

Uma das tarefas cruciais do JOCA é tornar simples ao programador a gerência das linhas de execução que dão vida aos dispositivos. Para isso, um ou mais dispositivos estão associados a um *servidor de eventos*, instância da classe `Server` (Ilustração 3). Um servidor concentra um *pool* de linhas de execução, que são compartilhadas por todos os dispositivos do servidor.

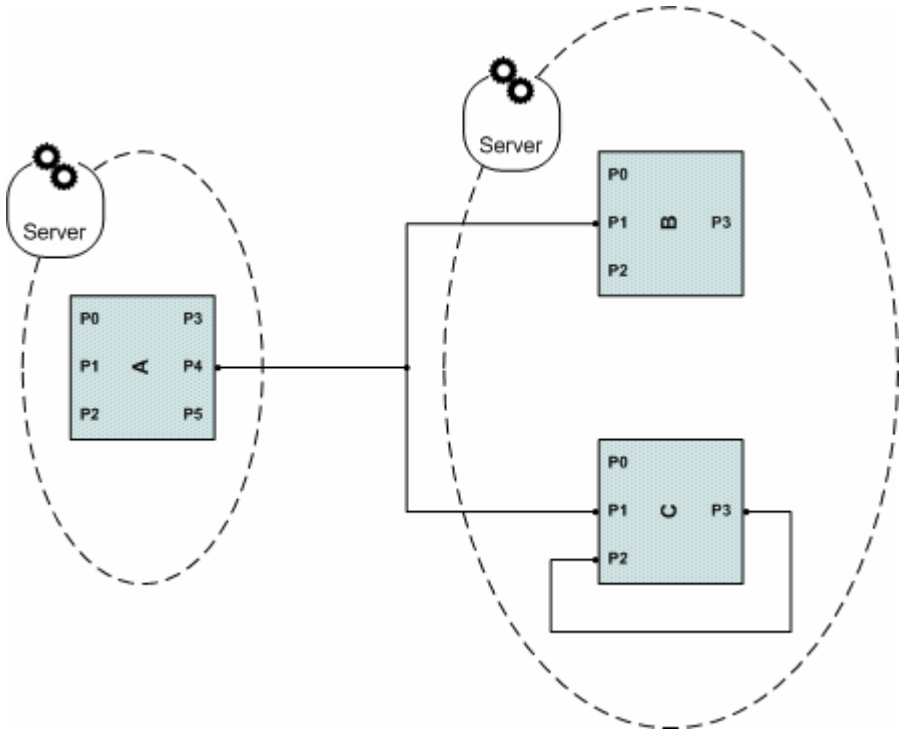

Ilustração 3 Associação entre dispositivos e servidores de eventos.

Tipicamente, a ocorrência de um evento num dado barramento deve ser tratada por todos os ouvintes de dispositivo associados a este barramento. Para cada ouvinte, então, é criada uma *tarefa*, que é colocada numa fila de espera do servidor. Assim que houver uma linha de execução livre no pool, ela é alocada à primeira tarefa da fila, e o evento é finalmente tratado.

Esta abordagem permite que o programador tenha algum controle sobre o gerenciamento de recursos, dando prioridade de execução a certos dispositivos em detrimento dos outros. O balanceamento ocorre de duas formas: variando o número de dispositivos associados a um dado servidor, é alterada somente a performance local; por outro lado, uma mudança no tamanho do pool modifica a performance de todos os servidores, visto que existirão mais linhas de execução em competição num nível global.

De uma forma geral, a associação explícita de dispositivos a servidores não é necessária. Caso o programador não especifique o servidor de um dado dispositivo, ele será associado a um servidor *default* do sistema.

O compartilhamento de linhas de execução é necessário na medida em que este é um recurso escasso no sistema; seria inviável alocar uma linha a cada dispositivo exceto nos circuitos mais elementares. Entretanto, o compartilhamento gera um problema adicional: não há garantia alguma de que o algoritmo em execução termine, e, sem término, não há liberação da linha de execução. Como conseqüência, numa situação excepcional, alguns dispositivos podem nunca receber uma linha de execução, provavelmente devido a uma falha de outro dispositivo. Em princípio, isto fere uma das características de dispositivos – sua autonomia – mas consideramos ser este um *tradeoff* aceitável nesta implementação. Futuras implementações podem utilizar outras técnicas – como, por exemplo, pools de tamanho auto-ajustável – para contornar este problema.