CPS — Programmation par composants

© 2019- Jacques Malenfant

Master informatique, spécialité STL – UFR 919 Ingénierie
Sorbonne Université

Jacques.Malenfant@lip6.fr

Cours 2

Programmation par composants séquentialisée

Plan

Objectifs de la séquence

1. Objectifs pédagogiques
   - Comprendre le rôle des interfaces de composants.
   - Comprendre comment sont représentées les interfaces de composants en BCM4Java.
   - Comprendre les principaux types d'interfaces de composants définies dans BCM et leurs interrelations.
   - Comprendre comment les interfaces de composants doivent être utilisées en relation avec les autres entités BCM (composants, ports, connecteurs, ...).
2. Compétences à acquérir
   - Savoir définir des interfaces de composants requises et offertes.
   - Savoir utiliser les deux principaux patrons de conception sur les interfaces de composants en BCM :
     - Interfaces à la fois requises et offertes.
     - Définition partagée d'interfaces d'implantation des services et interfaces de composants offertes.

Comment sont représentées les interfaces de composants ?

- Les interfaces de composants définissent les signatures d'appel des services, que ce soient celles utilisées par le client (requises) ou celles acceptées par le fournisseur (offertes).
- Elles sont représentées par des interfaces Java, mais pour les différencier des simples interfaces Java, elles étendent toutes indirectement l'interface `ComponentInterface`, ce qui les étiquette comme interfaces de composants.
  - Aucune interface de composants définie par les utilisateurs ne doit étendre *directement* `ComponentInterface` !
  - Toutes les interfaces de composants sont soit offertes, soit requises soit les deux et ceci est marqué par le fait qu'elles étendent directement ou indirectement les interfaces `OfferedCI`, `RequiredCI` voire les deux.
  - `OfferedCI` et `RequiredCI` étendent `ComponentInterface`.

L'interface `OfferedCI` et les interfaces offertes

- Toutes les interfaces de composants offertes étendent directement ou indirectement `OfferedCI`.
- Elles définissent les signatures de services appelables sur des composants jouant alors le rôle de fournisseurs.
- Puisqu'elles s'appliquent à des objets potentiellement appelables en RMI :
  - `OfferedCI` étend `java.rmi.Remote` ;
  - toutes les méthodes doivent pouvoir lancer `RemoteException`.
- Les interfaces de composants offertes sont implantées (au sens Java) par les ports entrants (*inbound*) qui matérialisent les points d'entrée pour les appels vers les composants fournisseurs de services.
- Les connecteurs appellent les ports entrants selon les signatures définies par l'interface de composants offerte correspondante.

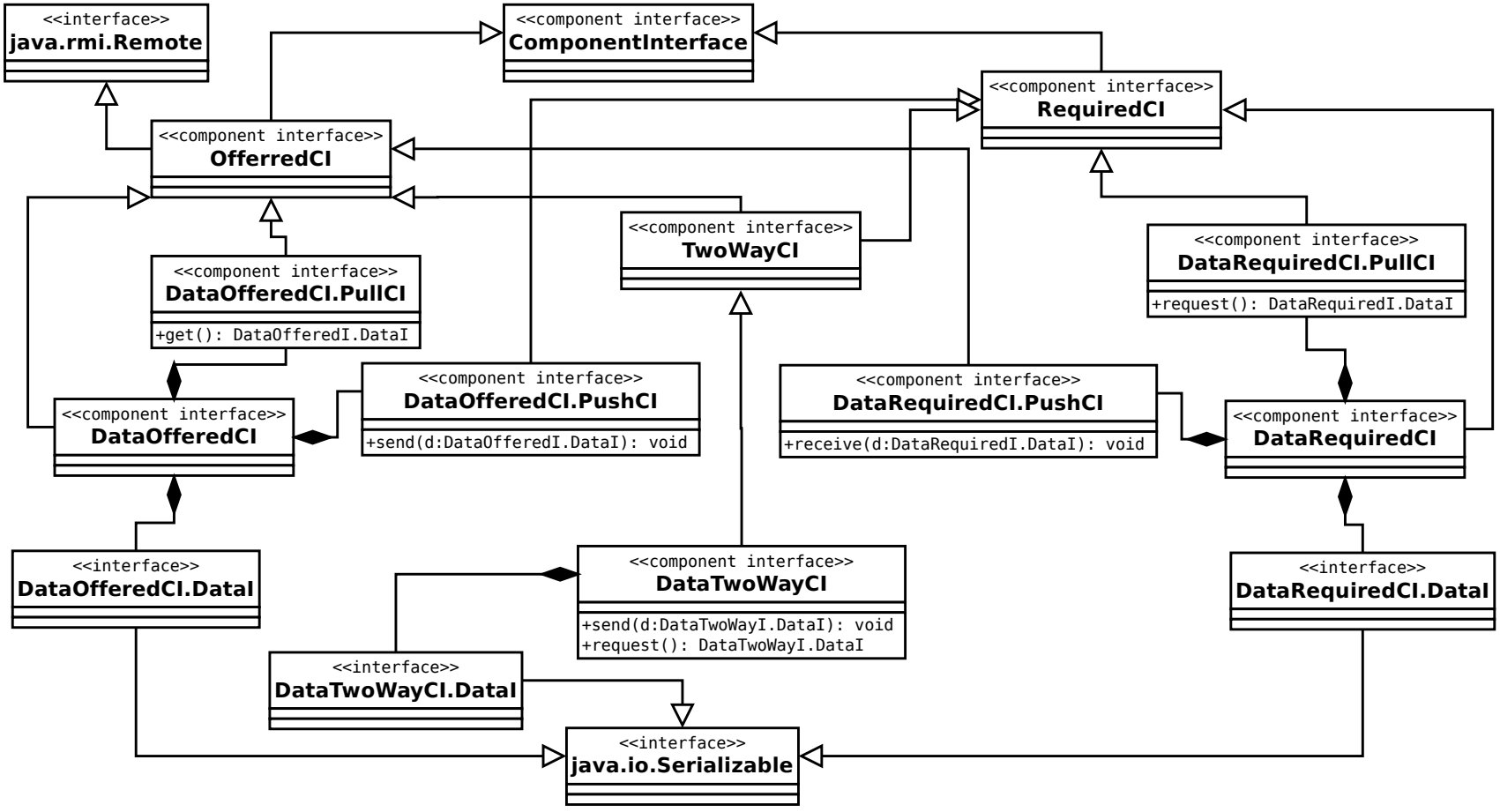
L'interface `RequiredCI` et les interfaces requises

- Toutes les interfaces de composants requises étendent directement ou indirectement `RequiredCI`.
- Elles définissent les signatures de services que les composants souhaitent appeler, ces composants jouant alors le rôle de clients.
  - Elles permettent d'écrire le code du composant appelant *sous l'hypothèse* de signatures de services attendues du fournisseur de services.
- Les interfaces de composants requises sont implantées (au sens Java) par les ports sortants (*outbound*) qui matérialisent les points de sortie pour les appels depuis ces composants clients de services.
- Elles sont aussi implantées (au sens Java) par les connecteurs qui sont appelés selon les signatures des services requis par le port sortant du client.

Conformité entre interfaces requises et offertes

- Informellement, une interface offerte est ***conforme*** à une interface requise R si elle déclare toutes les méthodes de R.
  - Plus formellement, pour toute méthode apparaissant dans l'interface requise, il existe une méthode de même nom, avec le même nombre de paramètres qui ont entre eux des types conformes et des types de résultats conformes au sens de Java.
  - L'interface offerte peut donc contenir plus de signatures que l'interface requise tout en restant conforme.
- Des interfaces requises et offertes conformes peuvent toujours mener à une connexion correcte.
  - Mais quand une interface offerte n'est pas conforme à une interface requise, il est parfois possible de les rendre conformes modulo quelques changements (noms de méthodes, ordre des paramètres, retrait ou ajout de paramètres réels par défaut, *etc.*).
  - Ce genre d'adaptations d'interfaces peuvent être programmées dans un connecteur.

Autres types d'interfaces prédéfinies de BCM



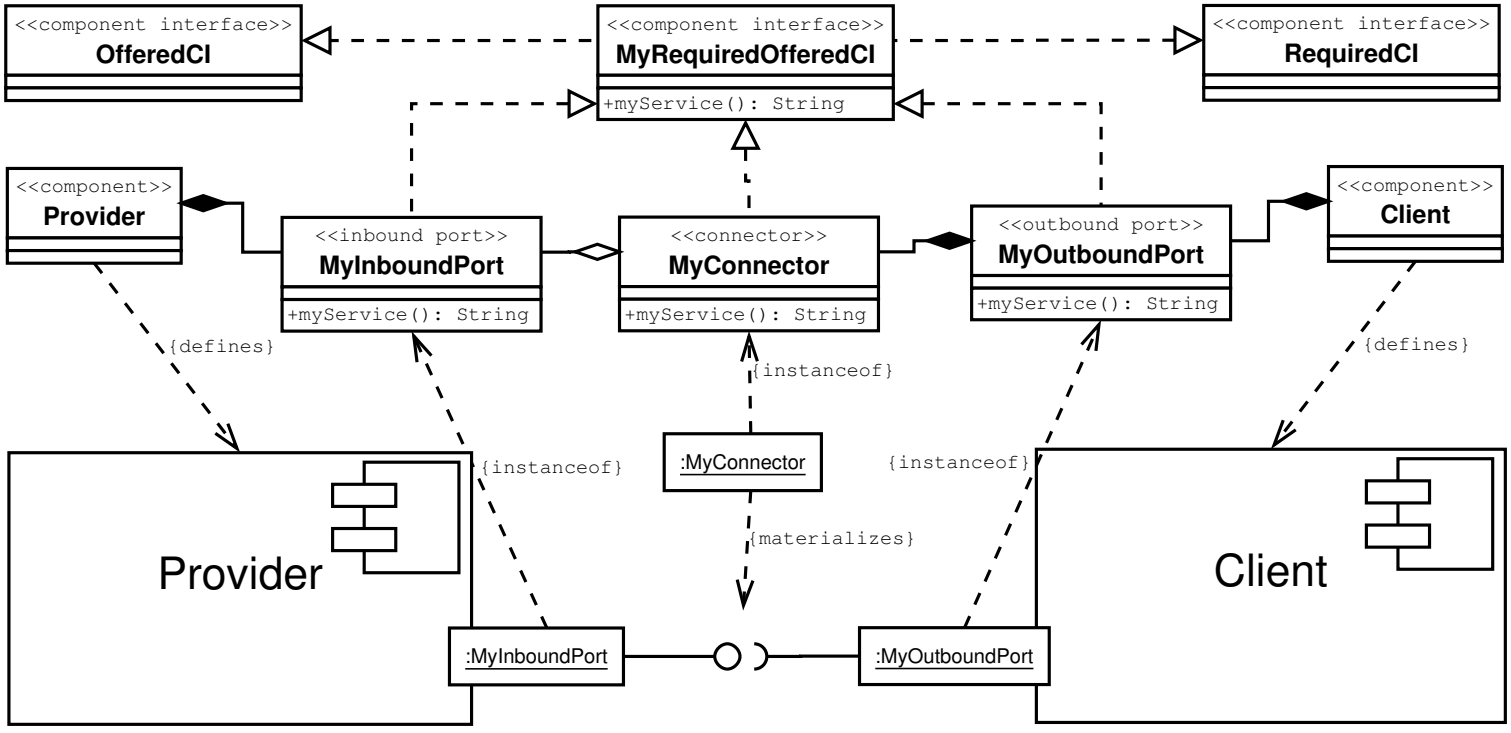
- Note : dans les sources de BCM, le répertoire `examples` contient un exemple appelé `pingpong` qui illustre l'utilisation de tous les types d'interfaces de BCM.

Exemples d'interfaces de composants

```
package fr.sorbonne_u.alasca.summing.vectorsummer.interfaces;
import fr.sorbonne_u.components.interfaces.RequiredCI;

public interface SummingServicesCI
extends RequiredCI
{
  public double sum(double x, double y) throws Exception;
}
```

```
package fr.sorbonne_u.alasca.summing.calculator.interfaces;
import fr.sorbonne_u.components.interfaces.OfferedCI;

public interface CalculatorServicesCI
extends OfferedCI
{
  public double add(double x, double y) throws Exception;
  public double subtract(double x, double y) throws Exception;
}
```

Patron d'interface à la fois offerte et requise



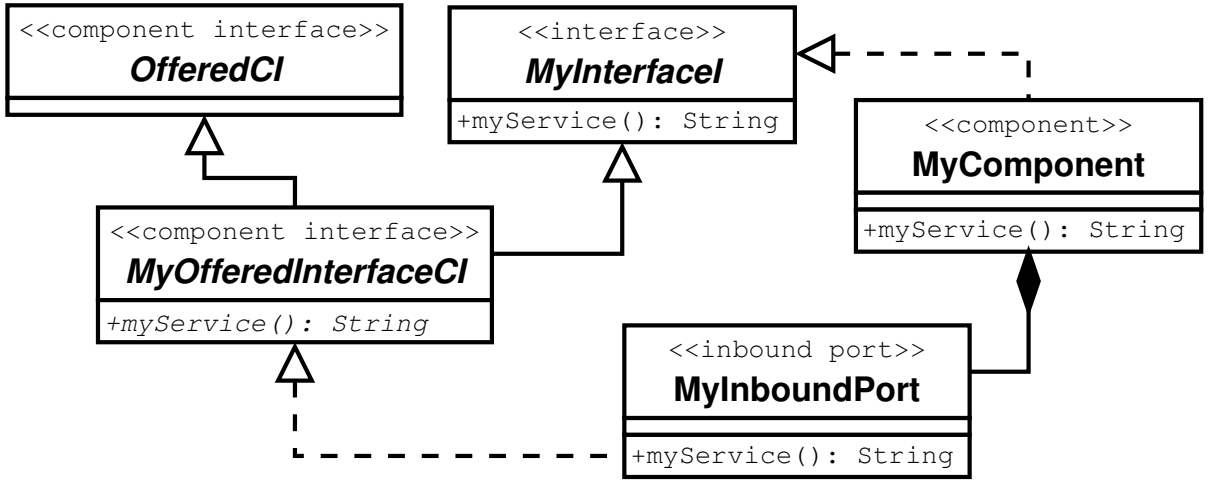
- Ici, l'interface `MyRequiredOfferedCI` étend à la fois `OfferedCI` et `RequiredCI`, ainsi elle est à la fois offerte et requise, et donc elle est implantée par les deux ports et le connecteur à l'identique.
- Utilisable lorsque le développeur maîtrise à la fois les composants fournisseurs et clients, pour leur imposer la même interface.

Exemple d'interface à la fois offerte et requise

```
package fr.sorbonne_u.alasca.summing.calculator.interfaces;
import fr.sorbonne_u.components.interfaces.OfferedCI;
import fr.sorbonne_u.components.interfaces.RequiredCI;

public interface CalculatorServicesCI
extends RequiredCI, OfferedCI {
 public double add(double x, double y) throws Exception;
 public double subtract(double x, double y) throws Exception;
}
```

Patron de définition partagée d'interface offerte et implantée

- Problème : un composant doit définir des services correspondant à l'interface offerte mais *ne doit jamais* l'implanter (au sens Java).
- Si on souhaite avoir les mêmes signatures de méthodes dans le composant que dans l'interface de composants offerte, il faut les factoriser dans une interface Java commune.



- **Rappel important** : la classe décrivant le composant *ne doit jamais* implanter l'interface `MyOfferedInterfaceCI`, sinon son instance sera considérée comme un port par le *framework* BCM.

Exemple d'interface partagée implantée et offerte

```
package fr.sorbonne_u.alasca.summing.calculator.interfaces;
public interface CalculatorServicesImplementationI
{
 public double add(double x, double y) throws Exception;
 public double subtract(double x, double y) throws Exception;
}
```

```
package fr.sorbonne_u.alasca.summing.calculator.interfaces;
import fr.sorbonne_u.components.interfaces.OfferedCI;
import fr.sorbonne_u.components.interfaces.RequiredCI;
public interface CalculatorServicesCI
extends CalculatorServicesImplementationI, RequiredCI, OfferedCI
{
 // Répétition nécessaire pour respecter les règles RMI
 @Override
 public double add(double x, double y) throws Exception;
 @Override
 public double subtract(double x, double y) throws Exception;
}
```

Plan

Objectifs de la séquence

1. Objectifs pédagogiques
   - Comprendre le rôle des ports, des connecteurs et des registres.
   - Comprendre comment sont représentés les ports et les connecteurs.
   - Comprendre les principaux types de ports et de connecteurs définis dans BCM et leurs interrelations.
   - Comprendre comment les ports doivent être utilisées en relation avec les autres réalisations en BCM des concepts de composants et les registres.
2. Compétences à acquérir
   - Savoir définir des ports entrants et sortants.
   - Savoir utiliser les ports sortants dans des appels d'un composant client vers un composant fournisseur.
   - Savoir programmer des connecteurs simples et des connecteurs rendant des interfaces requises et offertes compatibles.
   - Savoir programmer des ports entrants pour relayer les appels vers un service implanté dans le composant, en gérant correctement les fils d'exécution de ce dernier.

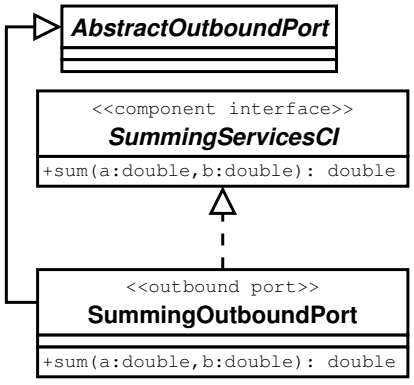
Les ports en BCM

- Les ports sont des objets Java, créés à partir de classes spécifiques, et détenus par les composants.
- Avant l'utilisation, un port :
  - doit avoir son interface de composants ajoutée à la liste des interfaces requises ou offertes du composant ;
  - est ensuite créé comme un objet Java, en lui passant (entre autres) la référence au composant qui le détient (son *owner*) ;
  - est publié dans les registres pour la connexion ;
  - et enfin connecté à un autre port.
- Après l'utilisation, le port doit être déconnecté puis dépublié (obligatoire avant la récupération du composant) puis détruit (optionnel, sinon cela se fait automatiquement lors de la destruction du composant).
- *Attention, le port et sa référence d'objet Java ne doivent jamais être utilisés à l'extérieur du composant qui le détient.*

Les ports sortants : côté composants clients

- Le principal rôle des ports sortants est de permettre de faire un appel de service selon l'interface requise, appel qui sera relayé au composant fournisseur grâce à la connexion.
- Ils sont créés par une expression `new`, comme n'importe quel objet Java.
- Ils sont mémorisés par le composant et on peut retrouver leur référence grâce à leur URI par le méthode `findPortFromURI` d'`AbstractComponent`.
  - Pour utiliser un port dans le composant, il est toutefois souvent plus simple de conserver la référence sur un port retournée par l'expression `new` dans une variable du composant.
- Leur connexion se fait par le composant qui les détient à l'aide de l'URI du port entrant avec lequel ils doivent se connecter grâce à la méthode `doPortConnection` (voir plus loin).

Exemple de classe définissant un port sortant




```
public class SummingOutboundPort
extends AbstractOutboundPort
implements SummingServicesCI
{
  public SummingOutboundPort(ComponentI owner) {
    // le constructeur d'AbstractOutboundPort attend
    // l'interface implantée par le port et la
    // référence au composant qui le détient.
    super(SummingServicesCI.class, owner);
  }

  @Override
  public double sum(double a, double b) {
    ((SummingServicesCI)this.getConnector()).sum(a, b);
  }
}
```

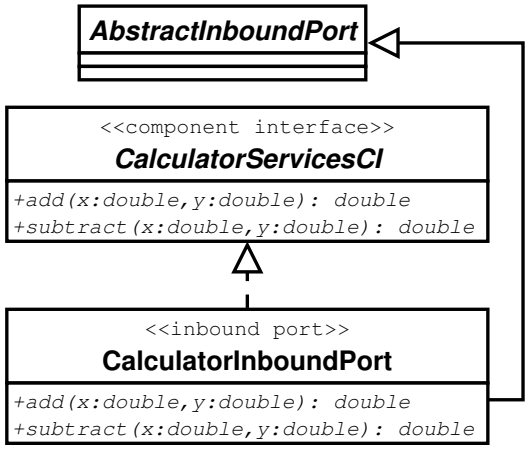
Exemple d'utilisation de port sortant

```
@RequiredInterfaces(required={SummingServicesCI.class})
public class VectorSummer extends AbstractComponent {
  protected SummingOutboundPort sop; // facilite l'utilisation dans le composant

    // ...
    this.sop = new SummingOutboundPort(this);           // création du port
    this.sop.publishPort();          // publication du port dans les registres
    // code réalisant la connexion (voir plus loin)
    // ...

  public double summingMyVector() {
    double result = 0.0;
    for(int i = 0 ; i < this.myVector.length ; i++) {
      result = this.sop.sum(result, this.myVector[i]); // utilisation
    }
    return result;
  }

    // généralement dans les méthodes finalise et shutdown:
    // déconnexion du port (voir plus loin)
    this.sop.unpublishPort();    // dépublication
    this.sop.destroyPort();      // destruction et retrait du composant
}
```

Les ports entrants : côté composants fournisseurs (serveurs)

- Leur principal rôle est de transmettre les appels de service selon l'interface offerte aux méthodes d'implantation du composant fournisseur pour être exécutés.
  - Ils sont créés par une expression `new`.
  - Pour la connexion, ils sont généralement passifs et connectés via le port sortant correspondant.
- Les composants ayant leurs propres fils d'exécution (*threads*), la transmission d'un appel au composant impose une rupture :
  - l'appel depuis le composant client jusqu'au port entrant (inclus) est exécuté par *un fil d'exécution du client* ;
  - lors du passage de l'appel au composant fournisseur, l'exécution du service est pris en charge par *les fils d'exécution du fournisseur*.
- Lorsqu'on utilise des interfaces requises et offertes simples, plusieurs ports sortants peuvent être connectés au même port entrant.
  - Un port entrant doit donc être *réentrant* (*i.e.*, code supportant sans risque d'être exécuté par plusieurs fils d'exécution à la fois).

Exemple de classe de port entrant

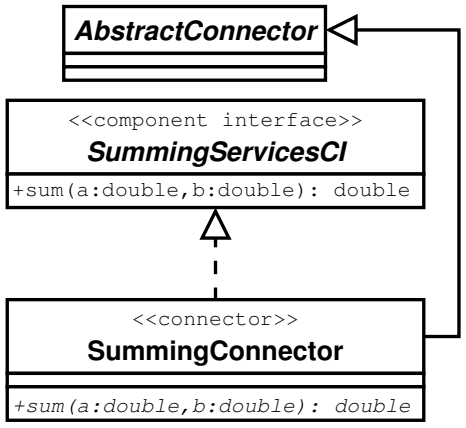



```
public class CalculatorInboundPort
extends AbstractInboundPort implements CalculatorServicesCI
{
  public CalculatorInboundPort(ComponentI owner) {
    super(CalculatorServicesCI.class, owner);
    assert owner instanceof Calculator;
  }
  @Override      // version avec création de classe anonyme
  public double add(double a, double b) {
    return this.getOwner().handleRequest(
      new AbstractComponent.AbstractService<Double>() {
        @Override
        public Double call() {
          return ((Calculator)this.getServiceOwner()).
                                              addService(x, y);
        }
      });
  }
  @Override                  // version avec lambda de Java 8
  public double subtract(double a, double b) {
    return this.getOwner().handleRequest(
              o -> ((Calculator)o).subtractService(x, y));
  }
}
```

Exemple d'utilisation de port entrant

```
@OfferedInterfaces(offered={CalculatorServicesCI.class})
public class Calculator extends AbstractComponent {
  CalculatorInboundPort cip;      // facilite son utilisation dans le composant

    // ...
    this.cip = new CalculatorInboundPort(this);           // création du port
    this.cip.publishPort();         // publication du port dans les registres
    // ...

  public double addService(double x, double y) {
    return x + y;
  }
  public double subtractService(double x, double y) {
    return x - y;
  }

    // dans les methodes shutdown et shutdownNow...
    this.cip.unpublishPort();   // dépublication
    this.cip.destroyPort();     // destruction optionnelle
    // ...
}
```

Exemple de connecteur et de connexion/déconnexion



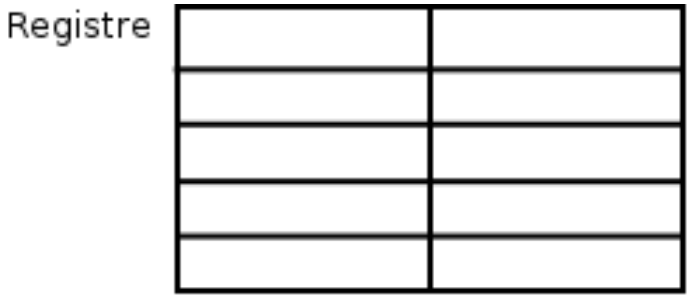

```
public class SummingConnector
extends AbstractConnector
implements SummingServicesCI
{
  @Override
  public double sum(double a, double b) {
    return ((CalculatorServicesCI)this.offering).add(a, b);
  }
}
```

- Pour connecter un port entrant à un port sortant, il faut connaître leurs URIs respectives et fournir la classe de connecteur :

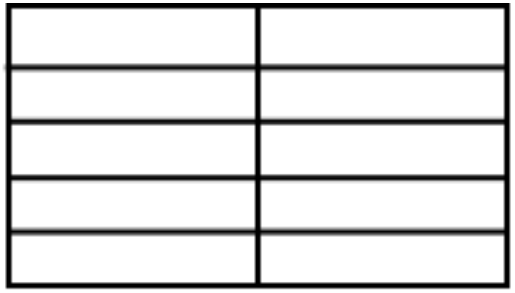
```
// Dans la classe définissant le composant VectorSummer
this.doPortConnexion(
   this.sop.getPortURI(), // URI du port sortant
   /* ici URI du port entrant, voir plus loin */,
   SummingConnector.class.getCanonicalName()); // nom de la classe
// ...
this.doPortDisconnection(this.sop.getPortURI();
```

- Notez la forme utilisée pour obtenir la nom de la classe du connecteur : elle est robuste !

Déroulement des opérations de connexion (mono-JVM)



Déroulement des opérations de connexion (mono-JVM)



Inbound Port

```
// Dans le composant serveur
public static String INBOUNDPORT_URI = "XXX";
ibp = new CalculatorServicesInboundPort(
                              INBOUNDPORT_URI, this);
```

Déroulement des opérations de connexion (mono-JVM)

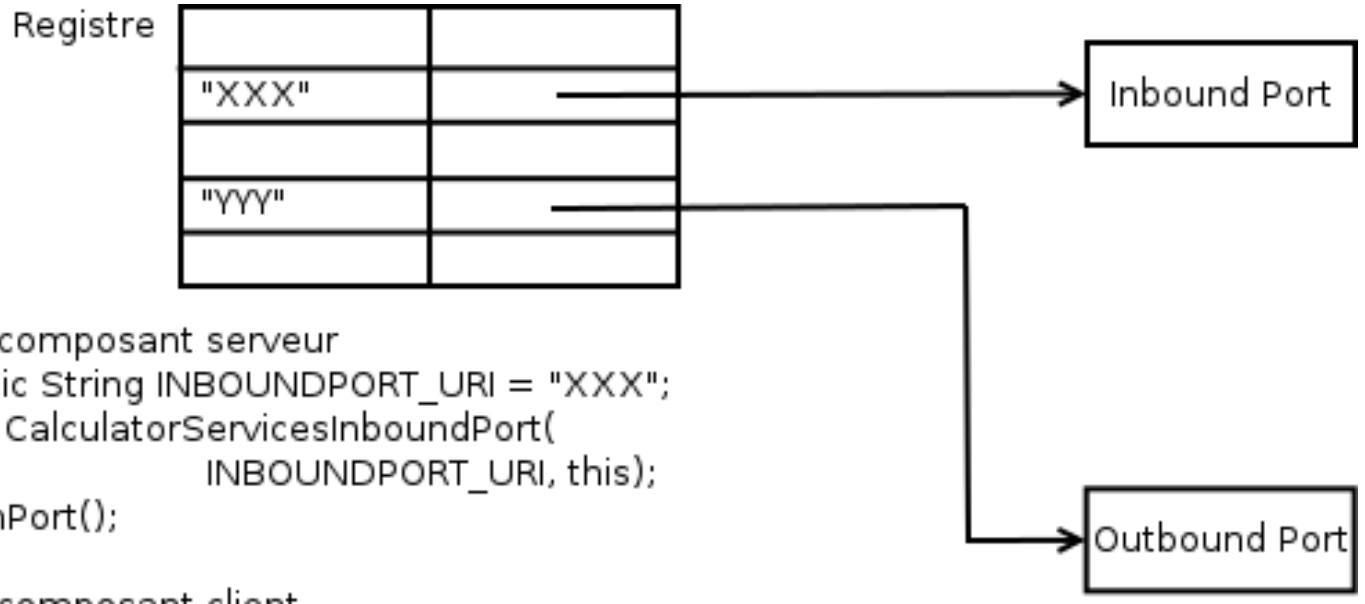

```
// Dans le composant serveur
public static String INBOUNDPORT_URI = "XXX";
ibp = new CalculatorServicesInboundPort(
                        INBOUNDPORT_URI, this);
ibp.publishPort();
```

Déroulement des opérations de connexion (mono-JVM)


```
// Dans le composant serveur
public static String INBOUNDPORT_URI = "XXX";
ibp = new CalculatorServicesInboundPort(
                        INBOUNDPORT_URI, this);
ibp.publishPort();

// Dans le composant client
protected static final OUTBOUNDPORT_URI = "YYY";
obp = new CalculatorServicesOutboundPort(
                        OUTBOUNDPORT_URI, this);
```

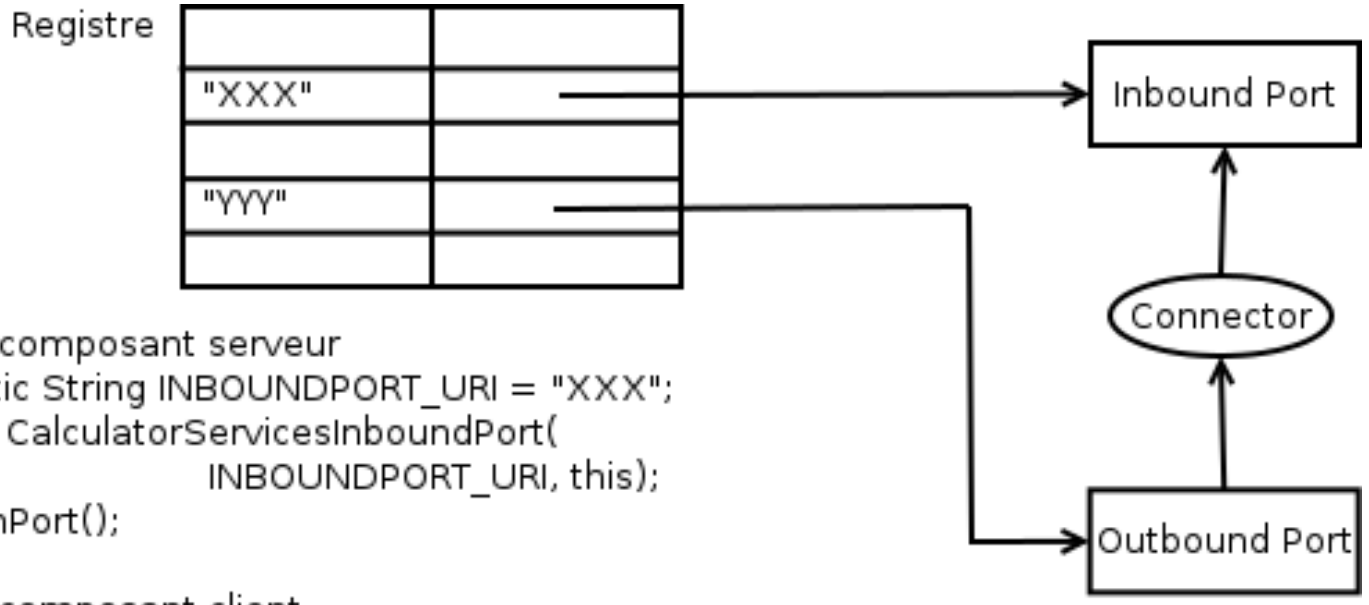
Outbound Port

Déroulement des opérations de connexion (mono-JVM)




```
// Dans le composant serveur
public static String INBOUNDPORT_URI = "XXX";
ibp = new CalculatorServicesInboundPort(
                        INBOUNDPORT_URI, this);
ibp.publishPort();

// Dans le composant client
protected static final OUTBOUNDPORT_URI = "YYY";
obp = new CalculatorServicesOutboundPort(
                        OUTBOUNDPORT_URI, this);
obp.publishPort();
```

Déroulement des opérations de connexion (mono-JVM)




```
// Dans le composant serveur
public static String INBOUNDPORT_URI = "XXX";
ibp = new CalculatorServicesInboundPort(
                        INBOUNDPORT_URI, this);
ibp.publishPort();

// Dans le composant client
protected static final OUTBOUNDPORT_URI = "YYY";
obp = new CalculatorServicesOutboundPort(
                        OUTBOUNDPORT_URI, this);
obp.publishPort();

// Toujours dans le composant client
this.doPortConnection(
        obp.getPortURI(),
        INBOUNDPORT_URI,
        CalculatorServicesConnector.class.getCanonicalName());
```

Relations de connexion

- Les connexions sont les seuls moyens qui permettent à des composants de s'appeler les uns les autres.
- Pour les ports et les connecteurs simples (d'interfaces dérivées directement de `RequiredCI` et `OfferedCI`) :
  - Un port sortant n'est connecté qu'à un seul port entrant.
  - Un port entrant peut être la cible de plusieurs connexions ; ceci correspond au fait qu'un composant fournisseur peut servir plusieurs composants clients, comme il se doit.
  - Cette asymétrie force les composants clients à avoir autant de ports sortants que connexions sortantes.
  - Il y a également un connecteur par connexion.
- Pour les autres types de ports et de connecteurs (autres interfaces), les connexions sont de type 1:1, un port sortant est connecté à un seul port entrant et *vice versa*.
  - En effet, pour tous ces cas, il y a possibilité d'appel inversé de l'entrant vers le sortant (comme le mode *push* pour les port d'échange de données), ce qui force une relation 1:1.
  - Voir l'exemple `pingpong` de la bibliothèque BCM4Java.

Plan

Objectifs de la séquence

1. Objectifs pédagogiques
   - Comprendre la notion d'URI appliquée aux ports en BCM.
   - Comprendre le lien entre les URIs de ports et la nécessité d'avoir des registres.
   - Comprendre les différentes façons de récupérer les URIs des ports d'un composant.
2. Compétences à acquérir
   - Savoir créer un port avec une URI imposée ou avec une URI générée automatiquement.
   - Savoir connecter des ports avec des URIs imposées par le biais de partage de variables globales.
   - Savoir le faire avec un passage des URIs au moment de la création des composants.
   - Savoir connecter deux composants créés dynamiquement grâce aux opérations réflexives pour récupérer les URIs.

Qu'est-ce qu'une URI et leur rôle en BCM

- Définition : **U**nique **R**esource **I**dentifier.
- Généralisation de la notion de pointeur ou d'identifiant d'objet.
  - Quand on crée un objet Java par `new`, on récupère un identifiant d'objet qui sert à appeler cet objet.

```
Toto t = new Toto(); // crée et retourne l'identifiant affecté à t
t.m();               // l'identifiant dans t sert à appeler l'objet
```

- Mais cette référence n'a de sens que dans l'espace mémoire de la JVM où l'objet a été créé (cache une adresse mémoire).
- En programmation répartie, il faut avoir un moyen de faire référence à un objet situé dans l'espace mémoire d'une autre JVM.
- Dans BCM, les objets qu'on veut pouvoir identifier ainsi sont les ports ; ce seront les URI qui vont nous servir à cela.
- Les registres vont permettre de les connecter en faisant la correspondance entre URIs par définition globales et références d'objet locales.

Connexion et registres de publication des ports

- Tous les ports possèdent une URI (de type `String`) qui va servir à les identifier globalement pour les connexions.
  - Les URIs des ports peuvent être engendrées automatiquement à la création (cas utilisé pour les exemples précédents).
  - On peut aussi imposer une URI à un port lors de sa création en la passant à son constructeur.
  - Les ports possèdent une méthode `getPortURI()` permettant de récupérer leur URI.
- Pour lier deux ports, il faut récupérer leurs références Java, dont celle du port entrant :
  - en execution mono-JVM, on pourrait directement utiliser la référence Java obtenue lors de la création de l'objet port ;
  - mais en exécution multi-JVM, le protocole RMI va fournir une référence à un *proxy* local qui va appeler l'objet port via RMI.
- Pour avoir le même code dans les deux cas, toutes les connexions nécessitent une publication des ports dans un registre :
  - en mono-JVM, un registre local à la JVM suffit ;
  - en multi-JVM, nous verrons plus tard les registres impliqués.

Problématique de gestion des URIs

- Pour établir une connexion, le composant client doit connaître l'URI du port entrant de son composant serveur.
- Question : comment un composant peut-il connaître les URIs des ports *des autres composants* avec lesquels il veut se connecter ?
- Problème d'oeuf et de poule...
  - Les ports sont créés dans les composants, donc lorsqu'on est dans le composant client, on ne peut pas connaître *a priori* les URIs des ports entrants du fournisseur et *vice versa*.
- BCM fait en sorte que les ports puissent être créés avec une URI imposée ou de la générer automatiquement à la création.
- BCM fait également en sorte de conserver les informations sur les URIs de ses ports de manière à pouvoir répondre à des requêtes comme :
  - Quelles sont les URI des ports offrant ou requérant telle interface de composants ?
  - Quelles sont les interfaces offertes ou requises par un composant ?

Accès aux URIs

- Première idée : par partage de constantes globales.
  - Définir les URIs nécessaires comme des constantes globales et les imposer lors de la création des ports afin de les utiliser ensuite pour les connexions.
- Deuxième idée : passer des URI prédéterminées lors de la création des composants par les constructeurs.
  - Permet d'éviter la variable globale.
  - En mono-JVM, il est facile de générer une URI robuste.
- Solutions bien adaptées :
  - aux architectures statiques où le nombre de composants et les ports qu'ils vont créer sont connus à la conception de l'application ;
  - à certaines architectures régulières de taille fixée au déploiement en calculant les URIs et en les passant en paramètres lors de la création des composants.
- Nous verrons plus tard des techniques plus souples et plus dynamiques.

Exemple

- Alternative variable globale :

```
// dans une classe XYZ accessible globalement
public static final String URI_PORT_ENTRANT = "mon-URI";

// dans le composant fournisseur
MonInboundPort mip = new MonInboundPort(XYZ.URI_PORT_ENTRANT, this);
mip.publishPort();

// dans le composant client
MonOutboundPort mop = new MonOutboundPort(this);
mop.publishPort();
this.doPortConnection(mop.getPortURI(), XYZ.URI_PORT_ENTRANT,
                      MonConnecteur.class.getCanonicalName());
```

- Alternative par les constructeurs créant les ports :

```
String uri_port_entrant = AbstractPort.generatePortURI();
// création du composant fournisseur qui crée le port avec l'URI
AbstractComponent.createComponent(MonFournisseur.class.getCanonicalName(),
                                  new Object[]{uri_port_entrant, ...});
// création du composant fournisseur qui sauvegarde l'URI puis
// l'utilise pour la connexion
AbstractComponent.createComponent(MonClient.class.getCanonicalName(),
                                  new Object[]{uri_port_entrant, ...});
```

Plan

Objectifs de la séquence

1. Objectifs pédagogiques
   - Comprendre la notion de point de connexion entre deux entités logicielles.
   - Comprendre sa déclinaison en BCM4Java.
   - Comprendre le protocole de création et d'utilisation des points de connexions en BCM4Java.
   - Comprendre la notion de points de connexion *composite*.
2. Compétences à acquérir
   - Savoir créer un point de connexion à partir de ports et connecteurs BCM4Java.
   - Savoir utiliser les points de connexion pour connecter deux composants Java et appeler les services entre un composant client et un composant fournisseur.
   - Savoir créer et utiliser des points de connexion composites.

Qu'est-ce qu'un point de connexion en BCM ?

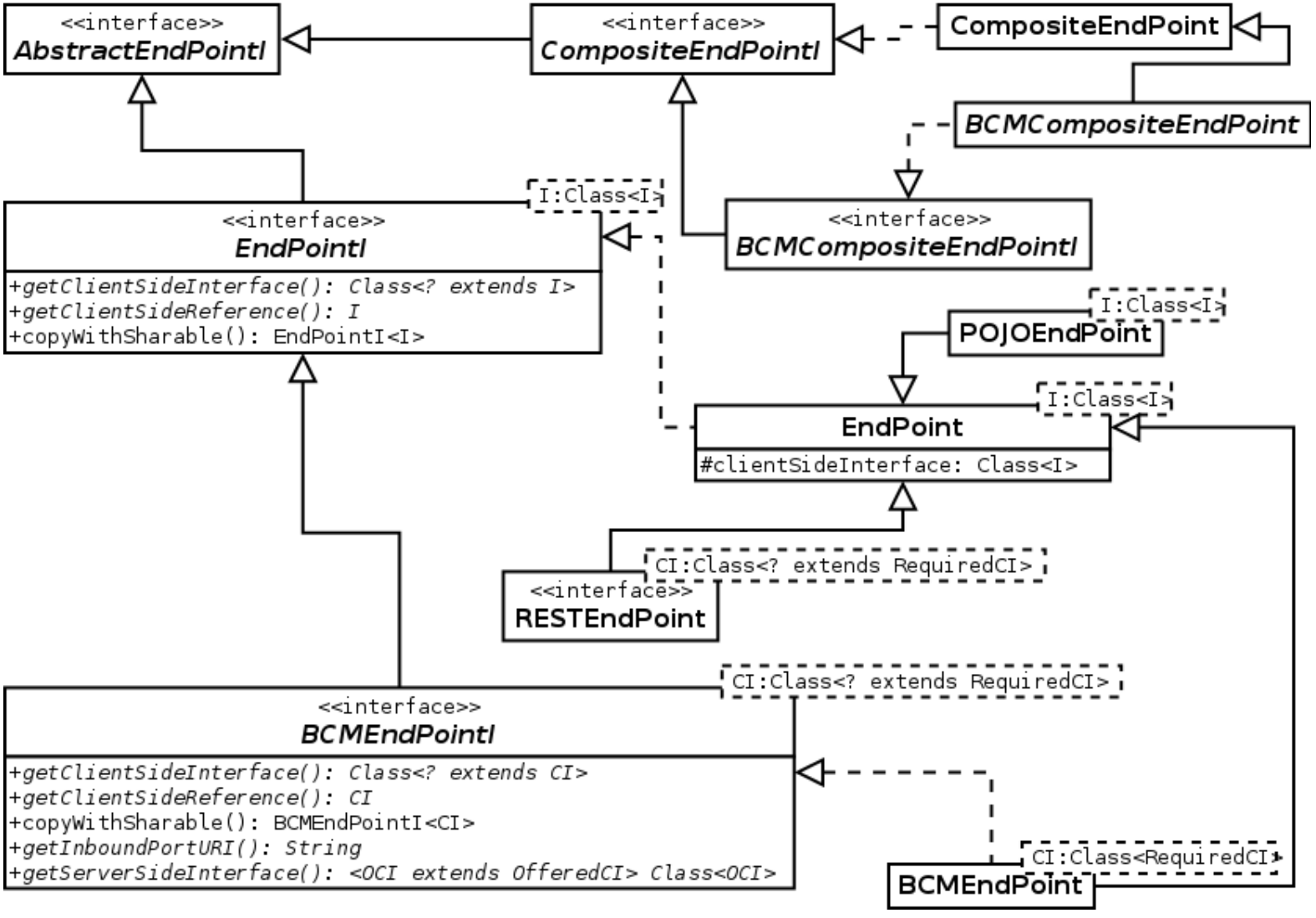
- Dans une architecture logicielle, un point de connexion (*endpoint*) contient les informations nécessaires et réalise la connexion entre deux entités logicielles communiquant à travers une API.
  - Côté client, il permet d'obtenir une référence via laquelle les services de l'API sont appelés.
  - Côté serveur, il définit la localisation de réception des appels.
    - Par exemple, en API REST, une URL représentant le chemin d'accès aux services de l'API.
  - Entre les deux, il établit le lien de transmission des appels (avec passage de paramètres) et de retour des résultats.
- En BCM, un point de connexion se définit par :
  - Une interface requise, proposée par la référence côté client.
  - Une URI de point d'entrée, équivalent de l'URL en REST, identifiant le chemin d'accès aux services offerts côté serveur.
  - Une interface offerte côté serveur.
  - Un adapteur assurant la correspondance entre l'interface requise et l'interface offerte.

Il se matérialise normalement par les ports, URI et connecteurs.

Qu'est-ce qu'un point de connexion en BCM4Java ?

- En BCM4Java, les points de connexion représentent une liaison *abstraite* entre deux composants.
  - Vu des composants BCM, les points de connexion présentent les liaisons entre clients et serveurs qui se comportent comme les ports et connecteurs habituels.
  - Toutefois, le caractère abstrait permet de mettre en œuvre la connexion en utilisant différentes technologies : ports et connecteurs BCM sous Java RMI, API REST en https ou autre.
  - Pour illustrer la capacité d'abstraction du concept et pour des raisons pédagogiques, la bibliothèque BCM4Java propose aussi une abstraction des liaisons entre objets Java standards sous forme de points de connexion.
- BCM4Java propose également le concept de points de connexion *composite*.
  - Un point de connexion composite regroupe plusieurs points de connexion simples reliant deux composants, l'un étant client et l'autre serveur pour l'ensemble des interfaces requises et offertes.

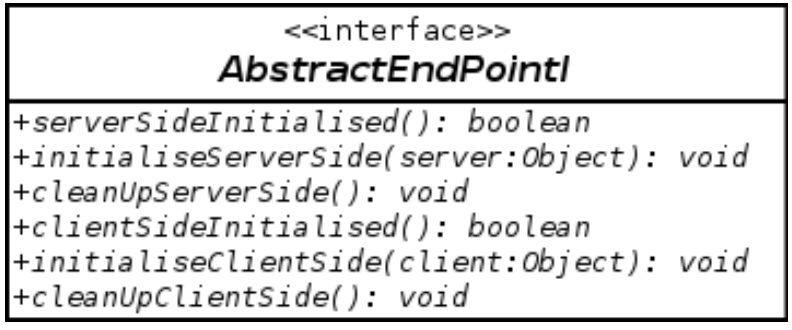
Points de connexion BCM4Java (vue d'ensemble)



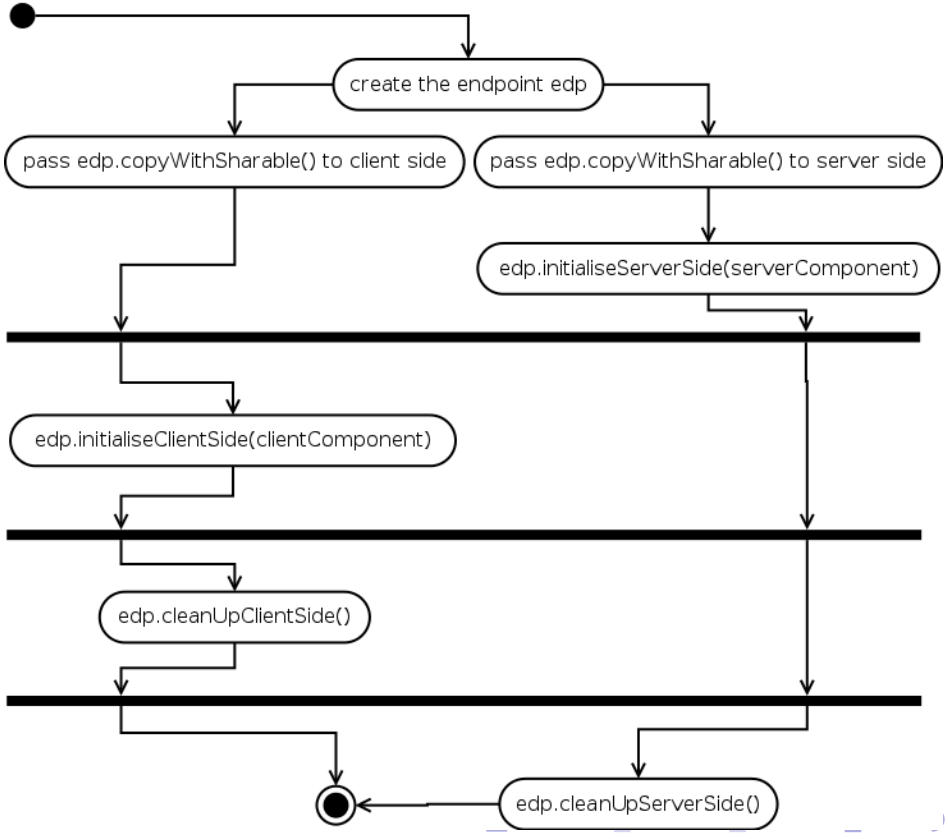
Protocole de création et d'utilisation

- Les points de connexion ont un cycle de vie avec des opérations qui doivent être appelées selon un ordre précis à assurer par des synchronisations implicites ou explicites.

Opérations :



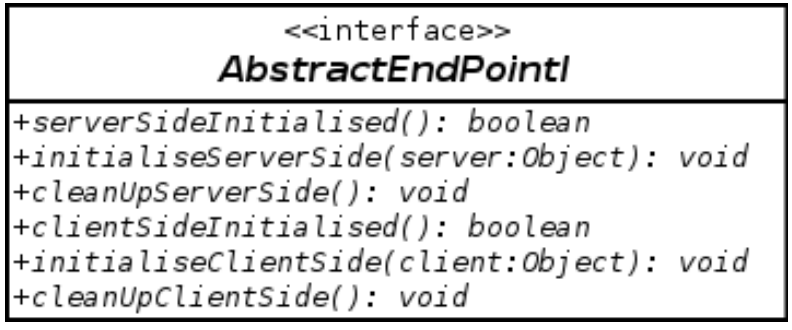
Protocole :



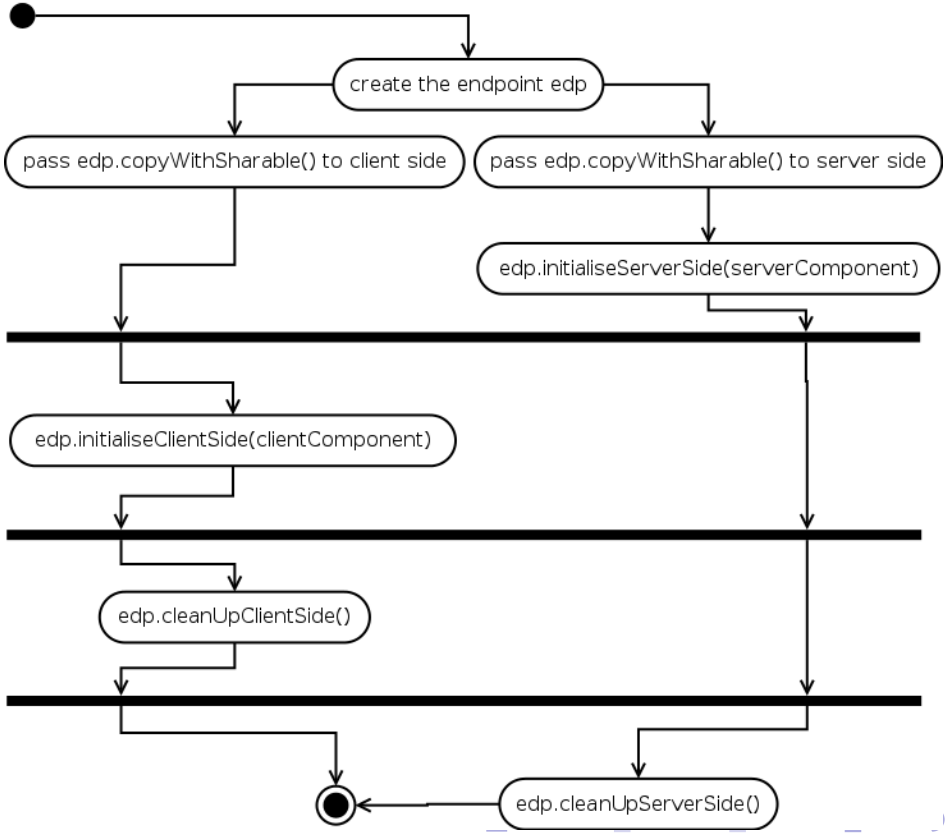
Protocole de création et d'utilisation

- Les points de connexion ont un cycle de vie avec des opérations qui doivent être appelées selon un ordre précis à assurer par des synchronisations implicites ou explicites.

Opérations :

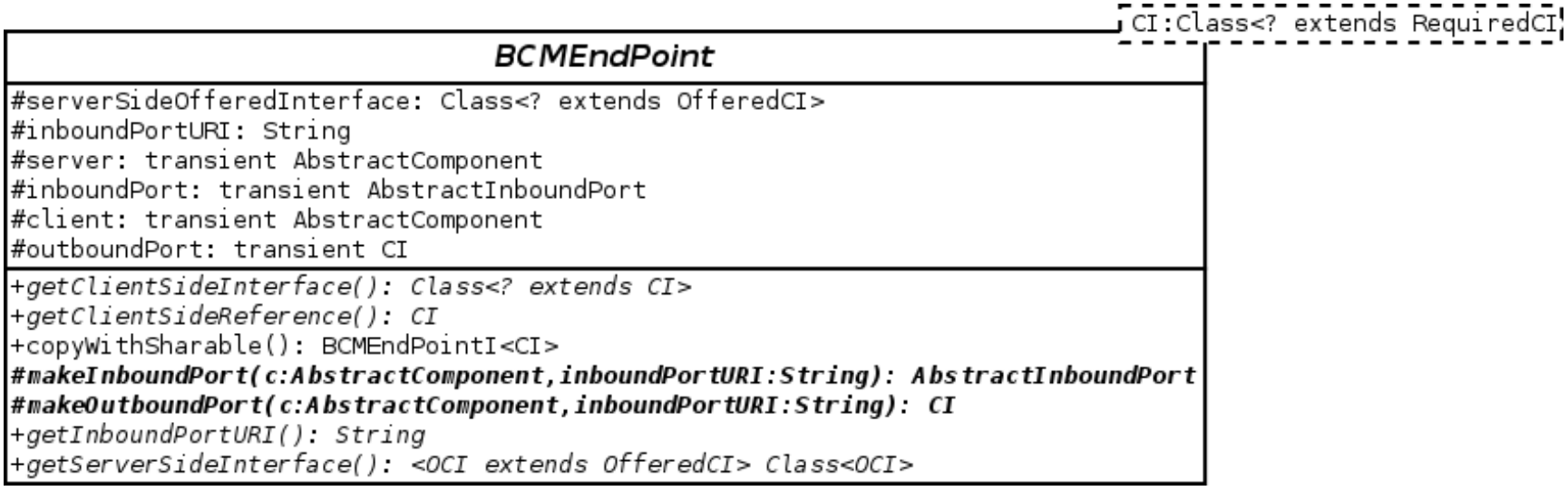


Protocole :



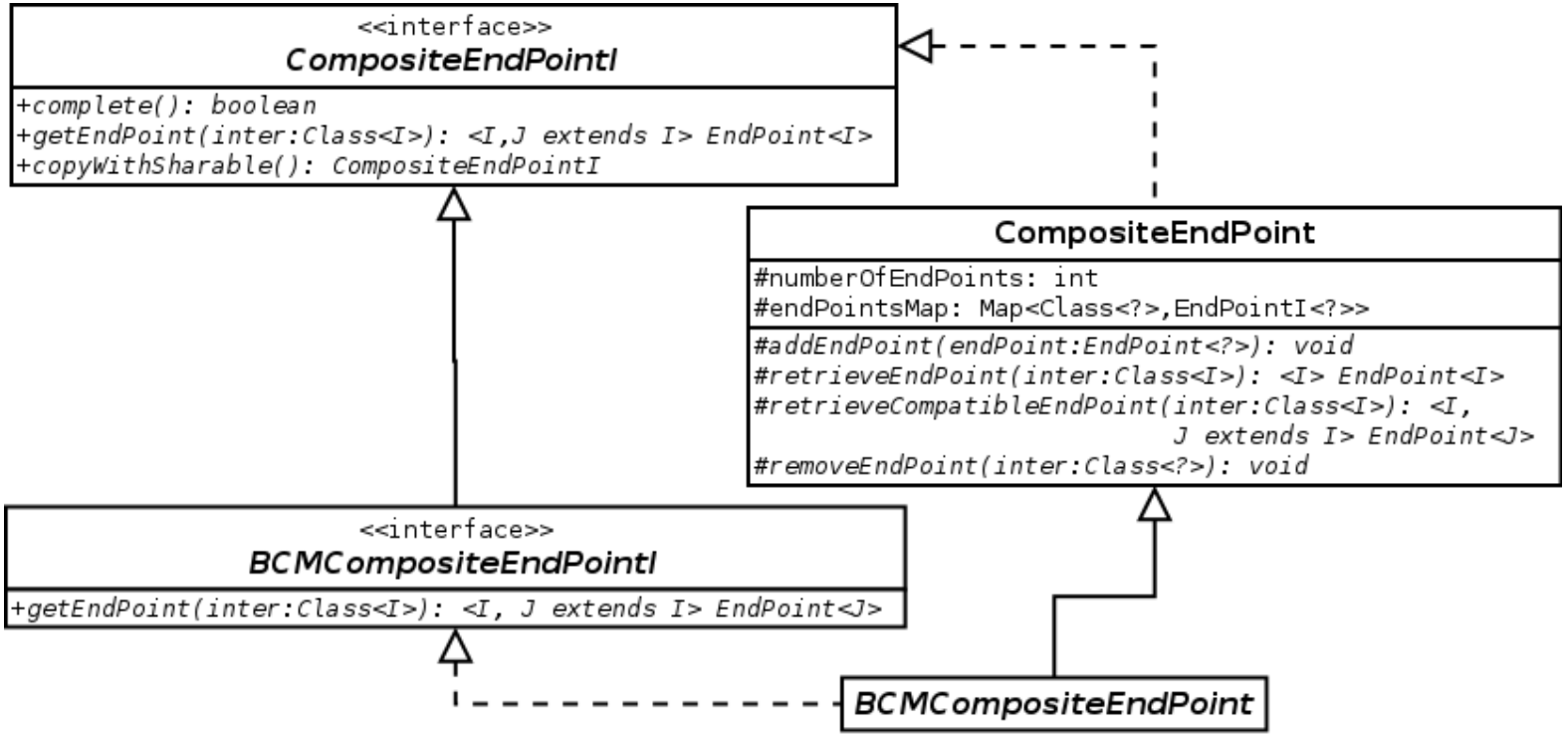
- En BCM, la synchronisation peut s'appuyer sur le cycle de vie des composants :
  1. Constructeurs
  2. méthode `start`
  3. méthode `finalise`
  4. méthode `shutdown`

Points de connexion standards BCM



- Les méthodes `makeInboundPort` et `makeOutboundPort` sont abstraites. Il faut donc céer une sous-classe pour les implanter.
- Ces méthodes sont appelées par les méthodes `initialise-ServerSide` et `initialiseClientSide` respectivement.
- Selon le protocole de BCM, le port entrant doit être créé et publié, le port sortant créé et publié et seulement ensuite connecté, d'où le protocole des points de connexion.

Points de connexion composites

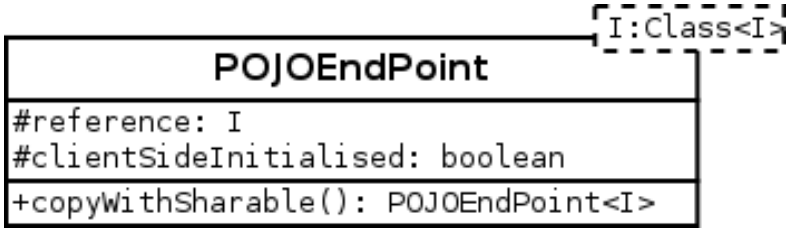


- Un point de connexion composite regroupe un nombre fixé de points de connexion.
- La méthode interne `CompositeEndPoint#addEndPoint` permet d'ajouter des points de connexion dans le composite.
- La méthode `CompositeEndPointI#getEndPoint` permet de récupérer un point de connexion par son interface côté client.

Exemples

- Dans la bibliothèque des exemples de BCM4Java, l'exemple client/serveur basique, déjà développé directement avec des ports, est proposé dans une variante avec points de connexion dans le package `fr.sorbonne_u.components.examples.edp_cs` et ses sous-packages.
- Le même exemple est repris pour illustrer les points de connexion composites. Les interfaces de composant `URIProviderCI` et `URIConsumerCI` sont divisés en deux pour créer deux points de connexion qui sont ensuite regroupés dans une points de connexion composite utilisé pour connecter les deux composants `URIProvider` et `URIConsumer`. Cette variante est proposée dans le package `fr.sorbonne_u.components.examples.cedp_cs` et ses sous-packages.

Points de connexion Java




```
public interface IncrementI {
  public int incrementAndGet();
}

public class Counter
implements IncrementI
{
  protected int counter = 0;
  public Counter(
    POJOEndPoint<IncrementI> edp
    )
  {
    edp.initialiseServerSide(this);
  }
  @Override
  public int incrementAndGet() {
    return ++this.counter;
  }
```

```
public class Client
{
  protected POJOEndPoint<IncrementI> edp;
  public Client(
    POJOEndPoint<IncrementI> edp
    )
  {
    this.edp = edp;
    edp.initialiseClientSide(edp);
  }

  public void go() {
    System.out.println(
      this.edp.getClientSideReference()
                    .incrementAndGet());
  }
}

// Séquence d'utilisation
POJOEndPoint<IncrementI> edp =
  new POJOEndPoint<>(IncrementI.class);
new Counter(edp);
(new Client(edp)).go();
```

Activités à réaliser avant le prochain TME

1. Examiner la variante `fr.sorbonne_u.components.examples.internal_cs` de l'exemple client/serveur basique et le comparer avec la version originale du même exemple dans `fr.sorbonne_u.components.examples.basic_cs`.
2. Examiner l'exemple `fr.sorbonne_u.components.examples.pingpong` et notez les variantes dans les échanges selon les types de ports utilisés.
3. Examiner le *package* `fr.sorbonne_u.components.endpoints` et les variantes de l'exemple client/serveur basique qui utilisent les points de connexion :
   - `fr.sorbonne_u.components.examples.edp_cs` pour les points de connexion simples ;
   - `fr.sorbonne_u.components.examples.cedp_cs` pour les points de connexion composites.

Tout ce code se trouver dans dans l'archive BCM4Java.