CPS — Programmation par composants

© 2019- Jacques Malenfant

Master informatique, spécialité STL – UFR 919 Ingénierie
Sorbonne Université

Jacques.Malenfant@lip6.fr

Cours 1

Introduction aux systèmes à base de composants

Plan

Généralités

- 10 séances de cours et 10 de TD/TME :

  Mardi, 16h00 – 18h00 : 2h cours magistraux.
  Lundi, 13h45 – 18h00 : 4h de travaux dirigés/encadrés sur machine.

- Évaluations (barême sur 100) :
  - Audit 1 (5/100) : TD/TME 4 du 12/02/2024.
  - Soutenance mi-semestre (35/100) : semaine du 4 au 8/03/2024 avec rendu préalable le 3/03/2024 à minuit au plus tard.
  - Audit 2 (5/100) : TD/TME 9 du 22/04/2024.
  - Soutenance finale (55/100) : période du 13 au 17/05/2024, et plus probablement les lundi 13/05/2024 et mardi 14/05/2024, avec rendu préalable le 12/05/2024 à minuit au plus tard.

Déroulement des évaluations

- Projet « Table de hachage répartie » en BCM4Java.
- Projet en quatre étapes.
  1. Audit 1 (5 à 10 minutes) : étape 1 complétée ; discussions/questions autour de votre code, démonstration.
  2. Soutenance à mi-parcours (20 minutes) : étape 2 complétée ; discussions/questions autour de votre code, démonstration, avec rendu de code préalable.
  3. Audit 2 (5 à 10 minutes) : étape 3 avec des scénarios de tests minimaux ; discussions/questions autour du code, démonstration.
  4. Soutenance finale (20 minutes) : projet complet exécuté sous scénarios de tests variés, avec rendu de code préalable ; discussions/questions autour du code, démonstration.
- Critères d'évaluation (cf. cahier des charges pour les détails) :
  - degré d'avancement et exécutabilité ;
  - qualité des solutions logicielles adoptées ;
  - qualité du code et de la documentation.

Contenu semaine par semaine

1. Introduction aux systèmes à base de composants
2. Programmation par composants séquentialisée
3. Assemblage et exécution des programmes à composants
4. Composants parallèles en BCM
5. Composants concurrents en BCM I
6. Composants concurrents en BCM II
7. Architectures logicielles dynamiques
8. Conception et programmation par contrats
9. Composants répartis en BCM
10. Systèmes répartis à grande échelle

Plan

Objectifs de la séquence

1. Objectifs pédagogiques
   - Comprendre les objectifs de l'approche par composants.
   - Comprendre les racines de cette approche.
   - Comprendre les principaux concepts de cette approche.
2. Compétences à acquérir
   - Savoir articuler les besoins industriels pour la conception, l'implantation et l'utilisation de composants avec les principaux concepts des modèles à composants logiciels.
   - Savoir décrire les rôles des différentes parties prenantes du développement logiciel à base de composants.
   - Savoir faire le lien entre un rôle dans le développement à base de composants (concepteur, développeur, utilisateur, ...) et les concepts et les artéfacts apparaissant dans les modèles à composants.

Vision globale (*the big picture*)

- Modèle : industrie comptemporaine.
  - Exemple de l'industrie automobile :
    1. équipementiers : conçoivent et produisent des pièces « standardisées », les *composants* ;
    2. concepteurs : conçoivent des produits intégrant des *composants standards* à partir de leurs *fiches techniques* ;
    3. fabricants : assemblent les produits à partir des *composants achetés* chez les équipementiers.
- Idée : répéter en informatique ce qui a si bien réussi dans d'autres industries.
  - Définir des « standards » permettant de concevoir et fournir des *composants sur étagères*.
  - Ouvrir un marché compétitif entre des composants de mêmes fonctionnalités.
  - Transformer les fournisseurs de logiciels en assembleurs de composants offerts sur étagère.
  - Augmenter significativement la réutilisation logicielle pour rendre l'industrie informatique plus sûre et plus efficace.

Qu'est-ce qu'un composant logiciel ?

Une ***entité logicielle*** se caractérisant comme suit :

- morceau de logiciel *encapsulé*, *directement déployable*, définissant explicitement tous ses *points d'interconnexion* qui exposent des *interfaces explicites* ;
- *assemblable* avec d'autres composants *par des tiers*, sans avoir à en examiner le contenu mais seulement en connaissant les points d'interconnexion et les interfaces qu'ils exposent ;
- dont l'assemblage se fait par *connexion* des *points de sortie* (appelants) avec des *points d'entrée* (appelés) réalisable depuis l'extérieur des composants ;
- dont tout le *cycle de vie*, depuis le chargement et l'initialisation jusqu'à la destruction en passant par l'activation (et désactivation), est *contrôlable de l'extérieur*, également sans avoir à connaître son contenu.

Contraintes de conception I

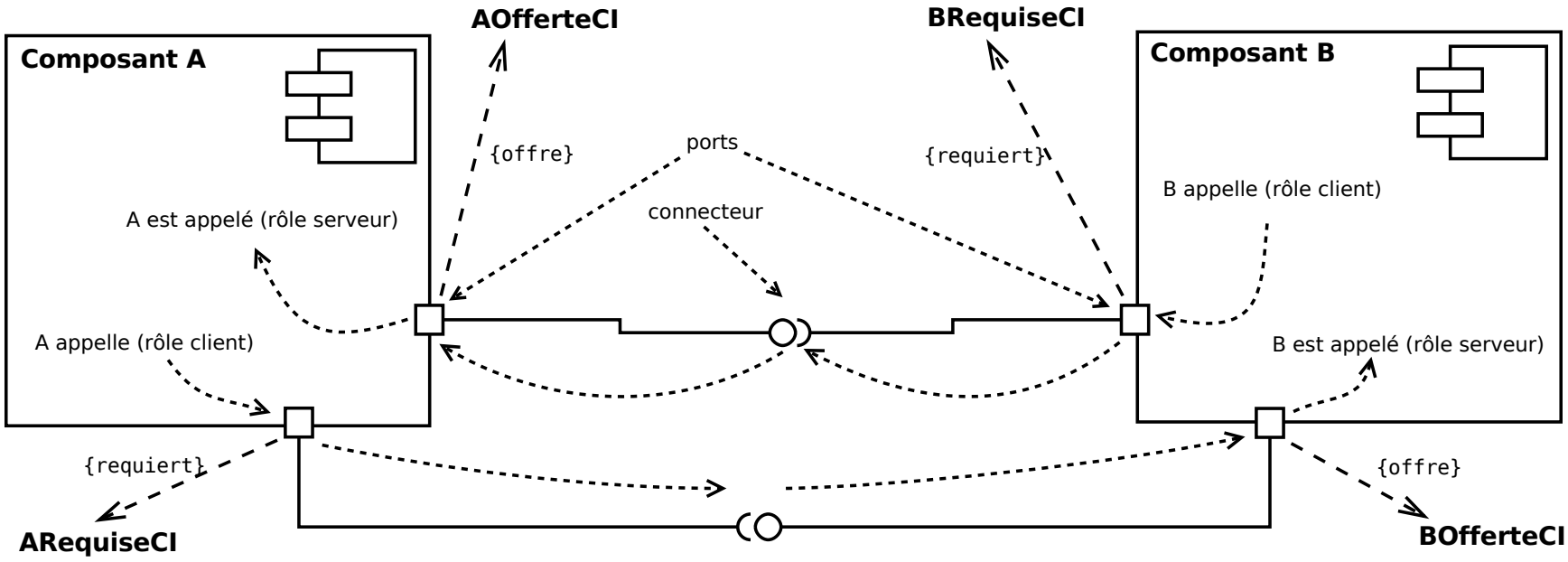
- Déployable ?
  - Une entité concrète, du *code exécutable*, chargeable et intégrable au sein d'une application (comparable à un objet ou à une bibliothèque chargeable dynamiquement plutôt qu'une classe).
  - Mais, aussi besoin d'une notion de *description* des composants (équivalente à la classe dans les langages à objets) qui ne doit pas être confondue avec les composants eux-mêmes.
    - *Attention*, le vocabulaire courant utilise généralement le même terme de *composant* dans les deux cas ; comparez :
      - *Tout* ordinateur possède un *composant* processeur.
      - *Le composant* processeur de *mon* ordinateur est en panne.
- Utilisable par des tiers ? (tiers $\neq$ ses développeurs)
  - Ses services offerts et requis sont décrits par des *interfaces* fournissant *toutes* les informations nécessaires pour connecter ces entités concrètes et les exécuter au sein d'une application.
  - Il possède un *cycle de vie* décrit par des *opérations standards*.

Contraintes de conception II

- Il forme donc une *unité d'encapsulation*, une *boîte noire* dont le contenu est *masqué* aux autres composants.

- Assemblable ou composable ?
  - Créer une application à base de composants se dit *assembler* des composants et lier deux composants se dit *connecter*.
  - Tous les *points de connexion* des composants sont visibles (référençables) *explicitement* de l'extérieur.
    - Implique que les composants définissent à la fois leurs points de connexion en entrée mais aussi leurs points de connexion *en sortie* (contrairement aux objets où ces derniers ne sont pas visibles de l'extérieur), c'est-à-dire les *services qu'ils requièrent et qu'ils appellent*.
    - Implique aussi que chaque point d'entrée ou de sortie possède un *identifiant unique* (URI) à l'échelle de l'architecture.
  - L'opération de connexion est réalisée explicitement *sur* (de l'extérieur) ou *par* chaque composant (de l'intérieur).
  - La connexion est (généralement) représentée explicitement par un *lien réifié*, référençable (connecteurs, *endpoints*).

Première vision générale



Comment définir un composant logiciel ? I

1. Services, activités et rôles
   - Les composants implantent des **services** dont l'exécution est déclenchée par les **appels** (requêtes) d'autres composants.
   - Les composants adoptent un **rôle**, client, serveur ou les deux, selon qu'ils sont appelés par ou appellent d'autres composants.
   - Les composants peuvent aussi avoir des **activités** propres (tâches) exécutées à leur initiative, indépendamment de l'exécution de leurs services (répondant eux à des appels extérieurs).
2. Interfaces
   - Les composants fournisseurs déclarent leurs services offerts à d'autres composants par leurs **interfaces offertes**.
   - Les composants clients déclarent leurs besoins des services d'autres composants par leurs **interfaces requises**.
3. Points de connexion (*end points*)
   - Les composants exposent leurs services offerts ou requis par des **points de connexion explicites**, chacun se décomposant en un point de sortie (*outbound*) côté client et un point d'entrée (*inbound*)) côté fournisseur.

Comment définir un composant logiciel ? II

- Les points de connexion représentent une connexion abstraite (dont la technologie n'est pas précisée : liaisons Java, liaison BCM, techno http/API REST, etc.).

4. Ports et connecteurs
   - En BCM, les points de connexion se matérialisent principalement par des **ports explicites** qui exposent les services offerts ou requis par les composants ainsi que par des **connecteurs explicites** reliant un port sortant à un port entrant.
   - Les ports et connecteurs peuvent être associées en un point de connexion de type BCM ;
   - **tous** les appels se services entre composants passent obligatoirement par un point de connexion explicite ou directement par des ports et connecteurs.
   - Lorsque les interfaces requises et offertes ne sont pas identiques, les connecteurs peuvent faire la « *médiation* » entre appels aux services requis et offerts (*i.e.*, être des *adaptateurs*).

Comment définir un composant logiciel ? III

5. Assemblage
   - Les **assemblages de composants** peuvent être réalisés *statiquement* lors du démarrage de l'application, *dynamiquement* pendant l'exécution de l'application après le démarrage, ou les deux (partiellement au démarrage, partiellement après).
   - Un assemblage **statique** est entièrement réalisé avant de démarrer les composants :
     - les composants peuvent être *créés et connectés* par la **machine virtuelle des composants**, au démarrage ;
     - les composants peuvent aussi être créés par la machine virtuelle des composants mais ensuite *se connecter eux-mêmes*, des clients vers les serveurs, au moment de leur démarrage.
   - Des composants peuvent aussi être créés **dynamiquement** par d'autres composants puis être connectés ou se connecter eux-mêmes, pendant l'exécution.

Comment définir un composant logiciel ? IV

6. Sous-composants
   - Un composant peut inclure en son sein des **sous-composants** (récursivement), mais les sous-composants sont *invisibles* de l'extérieur à moins que leur composant composite n'expose explicitement leurs services via ses propres ports et interfaces.
7. Déploiement
   - Les composants assemblés peuvent être déployés dans un **unique processus** (au sens du système d'exploitation).
   - Ils peuvent également être déployés sur **plusieurs processus**, sur **un seul ou plusieurs ordinateurs** (donc répartis).

Plan

Objectifs de la séquence

1. Objectifs pédagogiques
   - Décliner les concepts de la programmation par composants en Java avec BCM.
   - Introduire les premiers éléments de programmation par composants en BCM.
2. Compétences à acquérir
   - Prendre en main BCM et savoir retrouver dans cette bibliothèque et sa documentation la réalisation des différents concepts de l'approche à base de composants.

Qu'est-ce que BCM ?

- Basic Component Model : modèle de composants.
  BCM4Java = BCM for Java : implantation de BCM en Java.
- Une bibliothèque définissant une machine virtuelle pour composants s'exécutant en Java.
- Permet d'implanter des applications réparties :
  - a servi à implanter une application avec 10.000 composants déployés sur 50 JVM s'exécutant sur 5 ordinateurs ;
  - est utilisé dans des cours à Sorbonne Université depuis 2014 (une quinzaine de projets de M1, 5 générations d'étudiants de M1 et 11 de M2 à ce jour).
- Technologies Java utilisées (volontairement minimales et stables) :
  - J2SE (robuste aux passages de versions).
  - packages standards uniquement, dont `java.util.concurrent`
  - appel de méthodes à distance avec Java RMI.

D'une exécution mono-processus au réparti

- BCM vise principalement à produire des applications s'exécutant sur *plusieurs processus* (JVM exécutant BCM) *répartis* sur *plusieurs hôtes*.
- Le grand principe général qui guide sa conception et la programmation en BCM est :

  Toute application BCM programmée selon ses préceptes (à expliciter plus loin) pourra passer d'une exécution mono-processus à une exécution multi-processus voire en réparti par simple redéploiement des composants sur plusieurs JVM et répartition de ces dernières sur plusieurs hôtes, sans changer le code des composants.

- Pour y arriver, il faut se contraindre à appliquer les règles de programmation imposées car l'implantation actuelle BCM4Java n'a pas toujours la possibilité de les imposer par la contrainte.

Comment BCM4Java réalise les principaux concepts ?

- C'est un *framework* (cadriciel) écrit en Java.
- Un composant est un objet Java décrit par une classe marquée comme telle, dont les méthodes définissent ses services.
- Les interfaces de composants sont des interfaces Java marquées comme telles qui déclarent les signatures appelées/appelables.
- Les ports et les connecteurs sont des objets Java créés et détenus par les composants.
- Les connexions entre ports via les connecteurs se font
  - par référence Java s'ils sont dans la même JVM ;
  - par référence RMI s'ils sont dans des JVM différentes.

  Mais le programmeur n'a pas à s'en soucier autrement qu'en faisant en sorte que ses appels aient la même sémantique en appel local qu'en appel RMI (on y reviendra).
- Un sous-composant est représenté par un objet dont la référence est détenue par son composite.

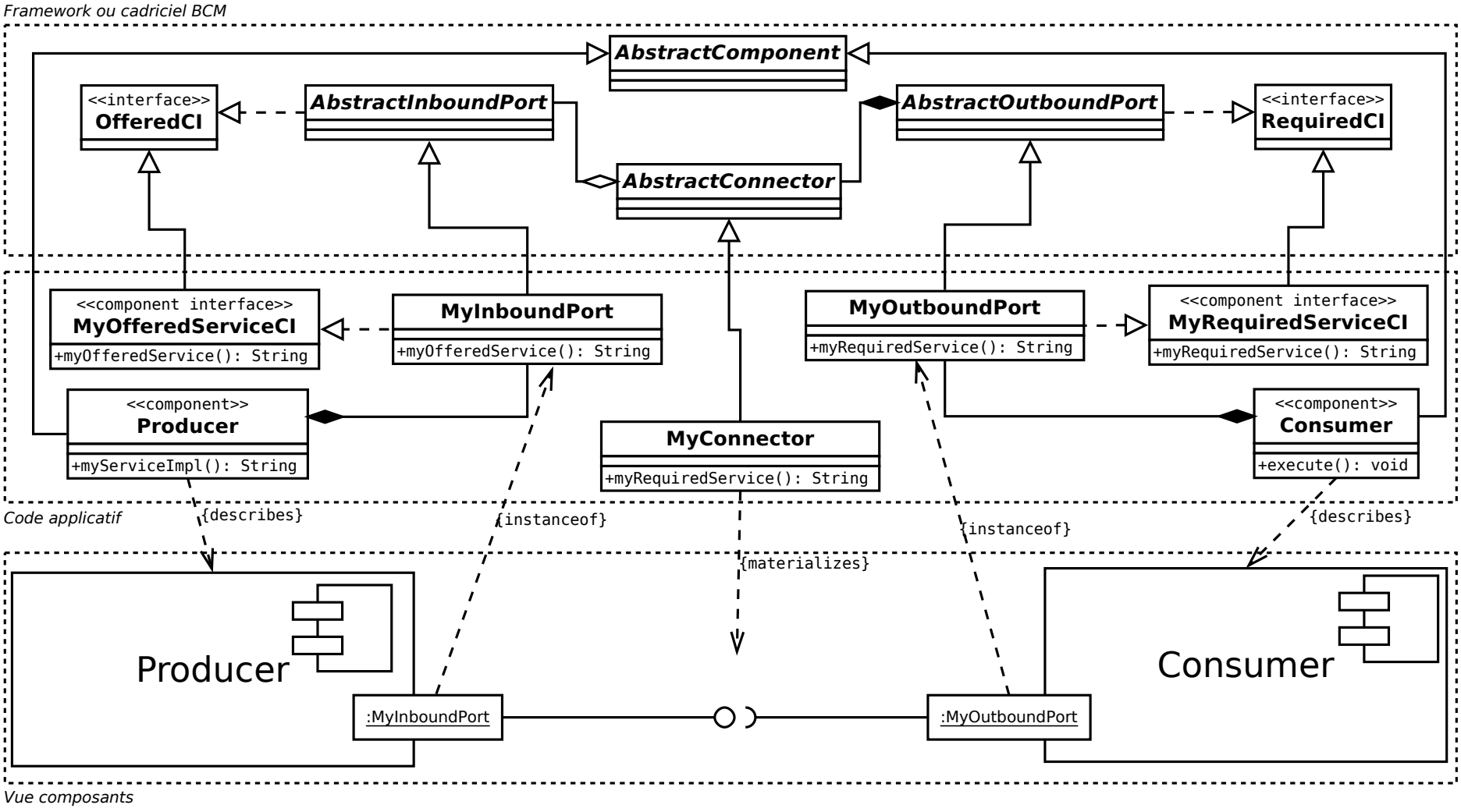
Comment réaliser les assemblages et le déploiement ?

- Les assemblages de composants *statiques* et leurs déploiements sont décrits dans des classes particulières définissant le « main » de la machine virtuelle pour composants.
- Le déploiement peut se faire au sein d'un processus, c'est-à-dire dans une seule JVM, auquel cas l'application s'exécute comme un programme Java classique.
- Le déploiement peut aussi se faire dans plusieurs processus, c'est-à-dire plusieurs JVM, auquel cas il faudra lancer autant de processus que de JVM plus des processus externes pour gérer les registres et la synchronisation des déploiements statiques.
  - Les détails seront abordés plus tard ;
  - une restriction : le nombre de processus participant à l'exécution d'une application est fixé définitivement au départ.

Le *framework* BCM

- La bibliothèque BCM4Java est en fait un *framework* ou cadriciel.
  - *framework* : ensemble d'interfaces, de classes abstraites et de classes concrètes définissant un noyau applicatif et que les utilisateurs doivent implanter (interfaces) et étendre (classes abstraites) pour développer leur application.
    Ex.: *framework* d'interfaces graphiques ou d'applications web.
- BCM fournit (entre autres) :
  - `AbstractComponent` que toute classe décrivant un composant doit étendre ; définit les opérations internes aux composants.
  - Des interfaces de composants `OfferedCI` et `RequiredCI` que toutes les interfaces de composants doivent étendre.
  - Différentes classes comme `AbstractInboundPort` et `AbstractOutboundPort` que tous les ports doivent étendre.
  - `AbstractConnector` que tous les connecteurs doivent étendre.
  - Une classe `AbstractCVM` qui doit être étendue pour définir les assemblages et le déploiement des composants pour exécution ; elle définit les opérations de la CVM sur les composants.

Illustration partielle du *framework* BCM



Programmation en BCM4Java

- Développer en BCM4Java consiste à programmer puis assembler des composants en suivant les étapes suivantes :
  1. Définir les interfaces de composants offertes et requises.
  2. Définir les classes et les signatures des services implantés pour chaque type de composants.
  3. Implanter les classes de ports entrants et sortants.
  4. Programmer les méthodes des services de chaque composant.
  5. Implanter les classes de connecteurs.
  6. Implanter la classe CVM pour assembler les composants puis exécuter l'application obtenue.
- Au cours 2, des *points de connexion* (*end points*) seront introduits ; cela modifiera un peu les étapes de réalisation des applications.

Plan

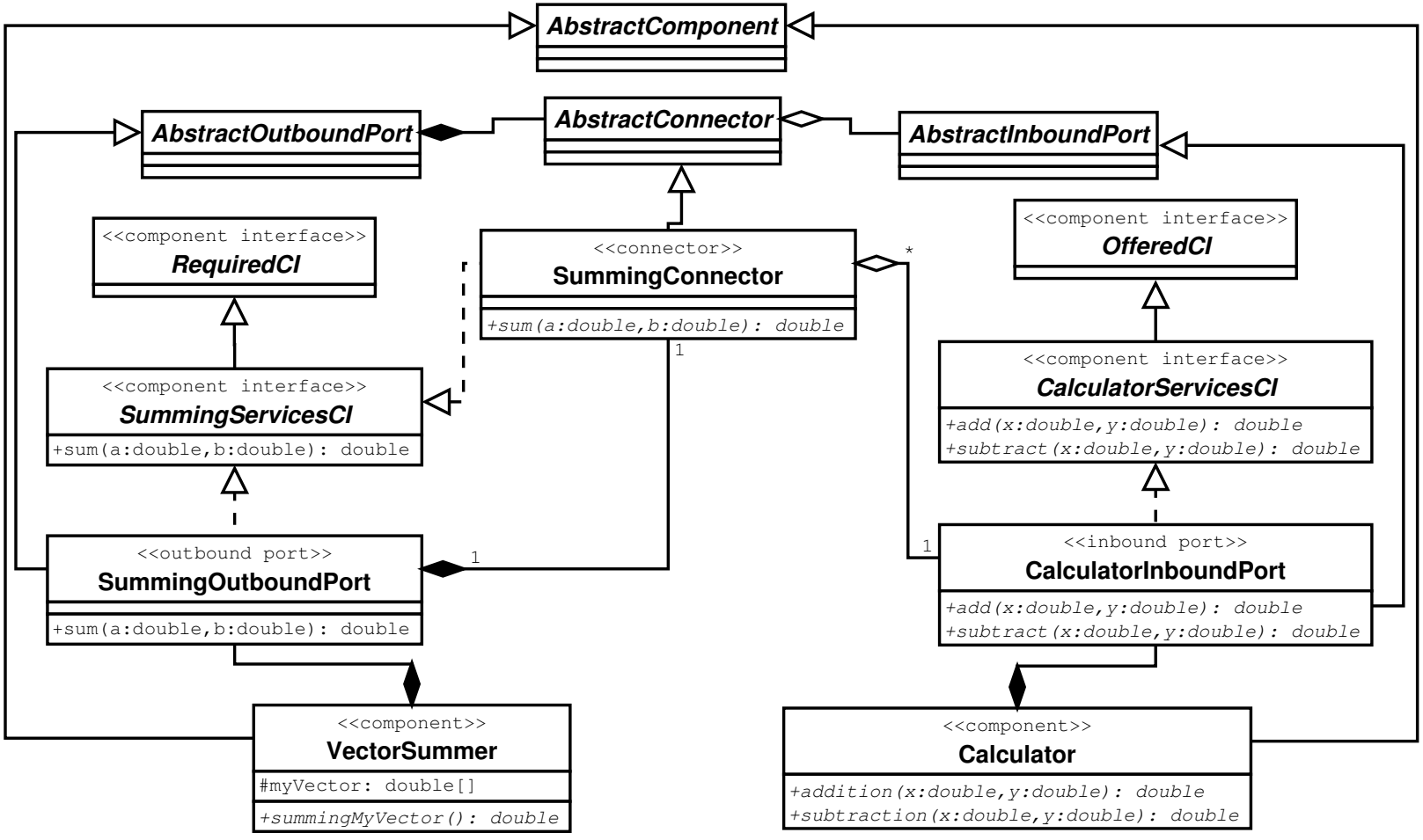
Objectifs de la séquence

1. Objectifs pédagogiques
   - Apprendre les rudiments de programmation en BCM grâce à un premier exemple complet.
2. Compétences à acquérir
   - Être en mesure de décrire et de suivre les grandes étapes de la mise en œuvre d'une application simple avec BCM du départ jusqu'à une exécution sur une (unique) machine virtuelle Java.
   - Savoir programmer les différents éléments d'une application BCM dans une première version simple : interfaces de composants, composants, ports, connecteurs, interconnexion et déploiement sur une (unique) machine virtuelle Java.

Premier exemple complet : somme d'un vecteur

- Pour comprendre comment définir une application en BCM, nous allons développer sous Eclipse un premier exemple complet.
- Cet exemple est volontairement simple pour mettre l'accent sur les mécanismes des composants de BCM.
- Cahier des charges :
  - Un composant offre un ensemble de services de calcul (addition, soustraction, multiplication et division).
  - Un composant client requiert ces services de calcul pour réaliser la somme des valeurs contenues dans un vecteur.

Diagramme de l'exemple



Mise en œuvre

- Étapes de réalisation de cet exemple :
  1. Définir l'interface de composants `CalculatorServicesCI`.
  2. Implanter le port entrant `CalculatorServicesInboundPort`.
  3. Implanter la classe de composant `Calculator`.
  4. Définir l'interface de composants `SummingServicesCI`.
  5. Implanter le port sortrant `SummingServicesOutboundPort`.
  6. Implanter la classe de composant `VectorSumer`.
  7. Implanter la classe de connecteur `SummingConnector`.
  8. Implanter la classe de déploiement `CVM`.
- L'implantation de cet exemple vous est présenté dans une vidéo déroulant toutes les étapes en détails.

Principaux concepts à approfondir

- **Composant** : objet Java matérialisant les éléments d'une application à base de composants en BCM.
  - Classes Java étendant directement ou indirectement `AbstractComponent`.
- **Interface de composants** : interface Java marquée comme telle représentant les services offerts ou requis par des composants.
  - Interfaces étendant indirectement `ComponentInterface`.
- **Port** : objet Java matérialisant les points d'entrée ou de sortie des appels entre les composants (clients $\Rightarrow$ sortant ; fournisseurs $\Rightarrow$ entrant).
  - Classes étendent directement ou indirectement `AbstractPort` et implantant (au sens Java) une interface de composants.
- **Connecteur** : objet Java matérialisant la connection entre un port sortant et un port entrant.
  - Classes étendant directement ou indirectement `AbstractConnector` et implantant (au sens Java) une interface de composants.
- **Point de connexion** : objet Java matérialisant une connexion abstraite entre deux entités logicielles.
  - Classes étendant directement ou indirectement `EndPoint`.

Plan

Objectifs de la séquence

1. Objectifs pédagogiques
   - Comprendre la différence de niveaux entre langage d'implantation de BCM et langage de programmation des applications en BCM.
   - Comprendre comment sont représentés les composants en BCM4Java.
   - Comprendre les principales parties d'un composant.
   - Comprendre comment les services sont implantés et comment les appels de services faits à un composant sont exécutées.
2. Compétences à acquérir
   - Savoir développer un composant simple en BCM4Java.
   - Savoir distinguer le code de l'application du code qui gère les composants, et savoir où il est admis d'utiliser ce dernier.
   - Savoir implanter les services et activités propres à un composant.
   - Savoir lier les appels de services via un port entrant à son implantation Java dans le composant et à une exécution sur un fil d'exécution du composant.
   - Savoir utiliser les fils d'exécution d'un composant pour faire exécuter des services et des activités propres.

Représentation d'un composant

- En BCM4Java, un composant est défini par une classe et il est créé par la méthode statique `AbstractComponent#createComponent`.
  - C'est un objet particulier qu'on ne manipule qu'à travers les opérations fournies par le *framework* BCM4Java.
- La *marque* indiquant qu'un objet représente un composant est que sa classe de définition étend `AbstractComponent` qui elle-même implante l'interface `ComponentI` :

```
// définition de composants
public class MonComposant extends AbstractComponent {
  protected MonComposant(int ndT, int nbST) { super(nbT, nbST); }
  ... // définition internes au composant
}

// creation d'un composant dans le code framework
String uri = AbstractComponent.createComponent(
                MonComposant.class.getCanonicalName(), new Object[]{1, 0});
```

  - Les paramètres du constructeur (par défaut) indiquent le nombre de fils d'exécution que le composant aura à sa disposition pour exécuter ses services (nous y reviendrons plus tard).

Principales parties d'un composant

- La définition d'un composant comporte tout ce qu'une classe Java peut contenir, mais avec caractéristiques spécifiques :
  - **Partie classes et types internes** : comme pour les objets Java, la classe de définition d'un composant peut déclarer des classes et types internes (*inner*) pour ses composants (instances).
  - **Partie variables et constantes** : elle peut aussi déclarer des constantes et des variables.
  - **Partie constructeurs** : `AbstractComponent` définit des constructeurs par défaut mais les composants peuvent définir leurs propres constructeurs (qui appellent les précédents).
  - **Partie cycle de vie** : méthodes redéfinies d'`AbstractComponent` précisant le comportement du composant au fil de sa vie (démarrage, exécution, finalisation, arrêt).
  - **Partie implantation des services** : méthodes implantant les différents services du composant.
  - **Partie méthodes auxiliaires** : méthodes internes, utilisées par les autres méthodes et qui ne correspondent pas directement à des implantation de services.

Interfaces requises versus offertes

- Un composant requiert et offre des services en requérant et offrant des interfaces de composants.
- Les interfaces de composants déclarent les signatures d'appels de services qui indiquent comment un composant client doit appeler les services du composant serveur.
  - L'interface requise donne les signatures d'appels qui doivent être utilisées dans le code du composant client pour appeler les services via son port sortant.
  - L'interface offerte donne les signatures utilisées pour appeler les services du composant serveur via son port entrant.
- C'est le rôle du connecteur de faire la liaison entre les deux, comme on l'a vu dans notre premier exemple complet : l'appel sur la signature `sum` de l'interface requise `SummingServicesCI` devient dans le connecteur `SummingConnector` un appel sur la signature `add` de l'interface offerte `CalculatorServicesCI`.

Implantation des services offerts par des méthodes

- Les interfaces de composants offertes ne donnent qu'une *vision externe* des services proposés par le composant :
  - Le composant ***n'implante pas (au sens de Java) ses interfaces de composants offertes*** et il n'est pas forcé d'implanter des méthodes de même signature que celles qui y sont proposées.
  - Il peut définir des méthodes d'implantation de services de signatures différentes, ou même d'implanter un service grâce à plusieurs méthodes.
- C'est le rôle du port entrant de faire la liaison entre les deux, comme on l'a vu dans notre premier exemple complet : l'appel sur la signature `add` de l'interface offerte `CalculatorServicesCI` devient dans le port entrant `CalculatorServicesInboundPort` un appel à la méthode d'implantation `addService` du composant `Calculator`.

Exécution d'un service ou d'une tâche I

- Les composants BCM ont leurs propres fils d'exécution (*threads*) qui seront utilisés pour exécuter :
  - des requêtes correspondant à des appels de service faits par des clients et passant par les ports entrants,
  - des tâches correspondant à des activités que le composant peut lancer de lui-même.
- Les fils d'exécution (*threads*) des composants sont gérés en groupes (*pools*) définis par les classes d'`ExecutorService` de Java. Ces classes implantent deux méthodes principales pour leur soumettre du code à exécuter :
  - `<T> Future<T> submit(Callable<T> request)`
  - `Future<?> submit(Runnable task)`
- Qu'est qu'un `Callable` ? un `Runnable` ?
  - `Callable<T>` est une interface exigeant une implantation de la méthode `<T> call()` $\Rightarrow$ une *requête* avec résultat.

Exécution d'un service ou d'une tâche II

- `Runnable` est une interface exigeant une implantation de la méthode `void run()` $\Rightarrow$ une *tâche* sans résultat.

- Pour s'abstraire de l'implantation précise des fils d'exécution, les composants BCM offrent une méthode principale pour soumettre des requêtes à leurs fils d'exécution :
  - `<T> T handleRequest(ComponentService<T> request)` : appel *synchrone* qui soumet une requête exécutée dès qu'un fil d'exécution se libère et où le fil d'exécution du client est bloqué en attente du résultat jusqu'au retour de ce dernier.

  et une méthode principale pour leur soumettre une tâche :
  - `void runTask(ComponentTask task)` : appel *asynchrone* qui soumet une tâche exécutée dès qu'un fil d'exécution se libère et où le fil d'exécution du client poursuit immédiatement son exécution sans attendre de résultat.

Les requêtes et les tâches des composants

- BCM étend les `Callable<T>` de Java par l'interface suivante définie dans `ComponentI` :

```
public interface ComponentService<V> extends Callable<V> {
  public void setOwnerReference(ComponentI owner);
  public ComponentI getServiceOwner();
  public Object getServiceProviderReference();
}
```

`AbstractService` propose une implantation à étendre ; les détails seront examinés plus loin, mais le détenteur (*owner*) est le composant exécutant la requête et il est automatiquement affecté par ce dernier lors de la soumission de la requête.

- De même pour les tâches, BCM étend les `Runnable` de Java par l'interface suivante :

```
public interface ComponentTask extends Runnable {
  public void setOwnerReference(ComponentI owner);
  public ComponentI getTaskOwner();
  public Object getTaskProviderReference();
}
```

`AbstractTask` propose une implantation à étendre ; les détails seront également examinés plus loin (idem).

Appels passés à un composant serveur et exceptions I

- Les modes d'appels aux composants modifient le comportement vis-à-vis des exceptions selon la sémantique des méthodes `submit` et la manière de récupérer les résultats.
- Avant même toute exécution de requête ou de tâche, les méthodes `submit` peuvent lancer deux exceptions :
  - `RejectedExecutionException` : si la tâche n'a pas pu être démarrée (politique de gestion de la surcharge).
  - `NullPointerException` : si le paramètre requête est `null`.
- Les exceptions `e` lancées durant l'exécution d'une requête par `handleRequest` sont relayées par cette dernière et peuvent être rattrapées comme `ExecutionException` ayant pour cause `e`.
  - Pour faire exécuter une requête sur un service `r` de type de retour `int`, dans un port entrant d'un composant de type `C`, voici les idiomes usuels :

Appels passés à un composant serveur et exceptions II

```
return this.getOwner().handleRequest(
        new AbstractComponent.AbstractService<Integer>() {
          public Integer call() {
            return ((C)this.getServiceOwner()).r(...);
        });
// avec lambda-expressions depuis Java 8
return this.getOwner().handleRequest(o -> ((C)o).r(...));
```

- pour faire exécuter une requête `t` avec type de retour `void` :

```
return this.getOwner().handleRequest(
        new AbstractComponent.AbstractService<Void>() {
          public Void call() {
            ((C)this.getServiceOwner()).t(...);
            return null;
        });
// avec lambda-expressions depuis Java 8
return this.getOwner().handleRequest(
                o -> {((C)o).t(...); return null;});
```

Appels passés à un composant serveur et exceptions III

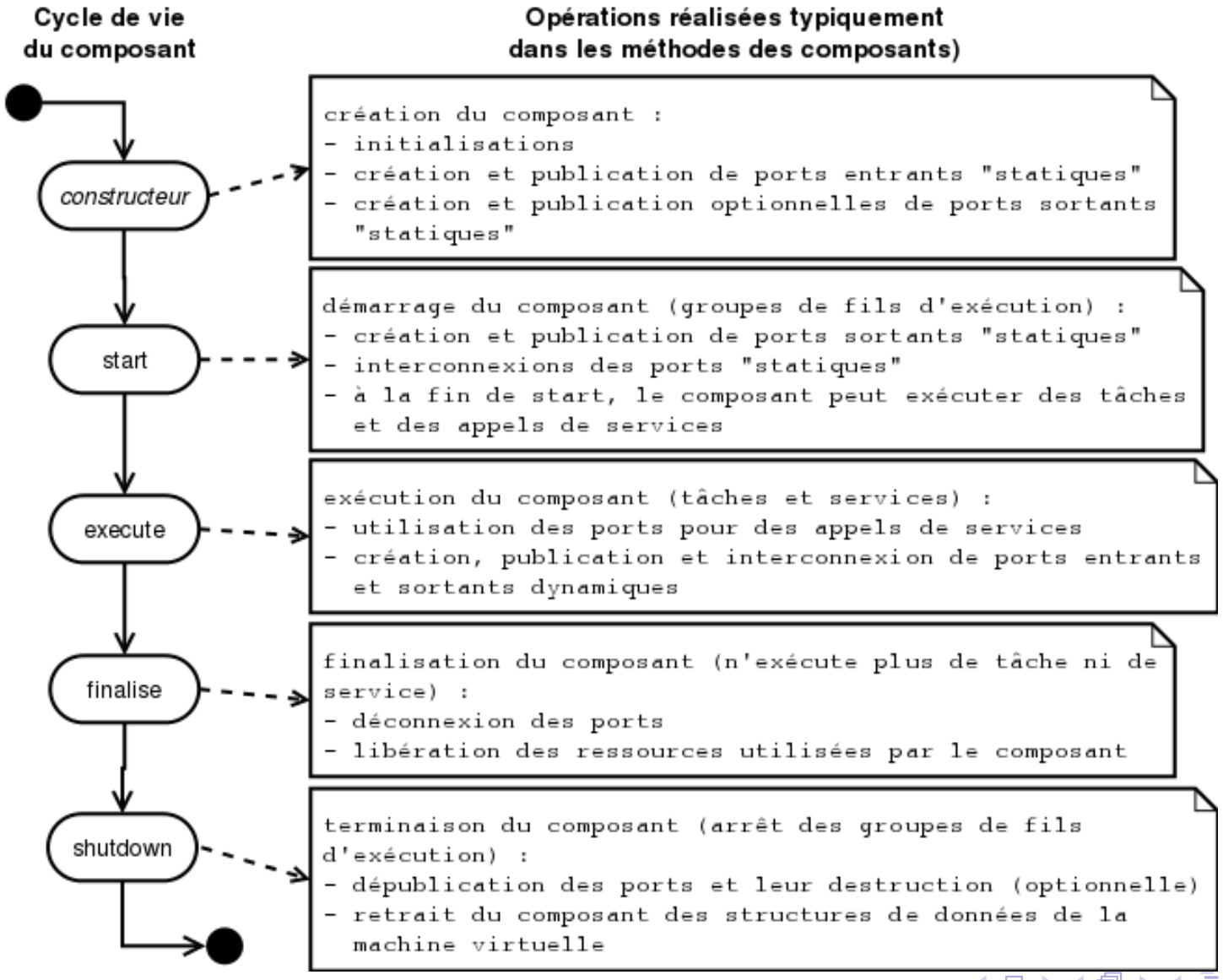
- Pour les tâches exécutées de manière asynchrone par `runTask`, impossible de propager les exceptions à l'appelant.

  Les exceptions sont alors silencieuses à moins de provoquer explicitement l'impression de la pile d'exécution.

  - Pour faire exécuter un service comme une tâche `a` de type de retour `void` tout en provoquant cette impression, voici les idiomes à adopter la plupart du temps :

```
this.getOwner().runTask(
        new AbstractComponent.AbstractTask() {
          public void run() {
            try { ((C)this.getTaskOwner()).a(...);
            } catch(Exception e) { e.printStackTrace(); }});
// avec lambda-expressions depuis Java 8
this.getOwner().runTask(
        o -> { try { ((C)o).a(...);
              } catch(Exception e) { e.printStackTrace(); }});
```

Cycle de vie des composants



Création statique de composants

- Pour éviter la confusion entre appels d'opérations de la machine virtuelle BCM et code de l'application à base de composants, BCM4Java masque autant que possible la référence aux objets Java qui représentent les composants.
- Pour arriver à cela, tous les constructeurs des classes définissant des composants *doivent être déclarés* `protected`, ce qui empêche de les appeler ailleurs que dans le composant lui-même.
- La création d'une instance de composant se fait par l'appel à la methode statique `AbstractComponent#createComponent` :

```
AbstractComponent.createComponent(
    URIProvider.class.getCanonicalName(), // nom de la classe à instancier
    new Object[]{...})                    // tableau des paramètres du constructeur
```

  Cette méthode retourne une URI propre au nouveau composant, qui peut être utilisée pour faire des opérations sur ce composant ; nous y reviendrons.

BCM et BCM4Java

- BCM pourrait être implanté dans différents langages à objets mais la seule implantation actuelle est en Java.
- BCM4Java utilise Java à la fois comme langage d'implantation de BCM et comme langage d'écriture des programmes en BCM.
- Plus précisément, BCM4Java se présente comme un *framework* implantant en Java les composants et le langage dans lequel ces composants sont programmés (ses services) est aussi Java.
- Il faut donc bien distinguer les opérations sur les entités de BCM, où on utilise le *framework*, du code des applications où on utilise simplement Java pour manipuler des données de base :

```
doPortDisconnection(myPort.getPortURI()); // ⇒ framework
ArrayList<String> l; l.add("un"); // code Java d'un composant
```

- Les opérations du *framework* ne doivent être utilisées que dans des contextes *légitimes* :
  - dans le code d'*assemblage* et de *déploiement* des composants
  - et dans chaque composant pour les *opérations sur lui-même*.

Activités à réaliser avant le prochain TME

1. Récupérer la vidéo de l'exemple somme d'un vecteur et visionner-la.
2. Récupérer la bibliothèque BCM4Java.
   - Le jar de BCM4Java à utiliser comme tel dans vos projets sous Eclipse pour séparer nettement votre code du code de BCM4Java.
   - L'archive de sources de BCM4Java à installer comme projet séparé dans votre workspace pour consulter facilement le code source de BCM4Java, sa documentation et les exemples fournis.
3. Examiner le code de l'exemple `basic_cs` et sa documentation.
4. Bien lire le cahier des charges du projet et préparer vos questions en vue de la prochaine séance de TD/TME.
5. Réfléchir à former vos équipes et envoyer un message à `Jacques.Malenfant@lip6.fr` comportant un nom d'équipe (voir le cahier des charges) et les noms de ses membres.