CPS — Programmation par composants

© 2019- Jacques Malenfant

Master informatique, spécialité STL – UFR 919 Ingénierie
Sorbonne Université

Jacques.Malenfant@lip6.fr

CC BY NC ND

Cours 3

Assemblage et exécution des programmes à composants

BY NC ND

Plan

cc BY NC ND

Objectifs de la séquence

1. Objectifs pédagogiques
   - Comprendre les notions d'assemblage de composants et de déploiements de composants en BCM.
   - Comprendre le lancement d'une application BCM et le cycle de vie d'une exécution de sa machine virtuelle à composants.
   - Comprendre l'imbrication entre le cycle de vie de la machine virtuelle à composants et les cycles de vie des composants.
2. Compétences à acquérir
   - Savoir programmer la création et l'interconnexion des composants dans le cas d'assemblages et déploiements statiques.
   - Savoir programmer la séquence de création, interconnexion, démarrage, finalisation puis extinction de la machine virtuelle BCM.

cc BY NC ND

Lancement d'une application BCM

- Comme pour Java, l'exécution d'une application BCM exige de lancer le programme à partir d'un point initial analogue à la méthode `main`.
- La machine virtuelle à composants de BCM introduit donc des moyens similaires à `main` pour ce faire, mais cela suppose plus de méthodes et une organisation un peu plus complexe.
  - Rappel : pour connecter des composants, il faut que les ports entrants aient été créés et publiés avant que l'on puisse les connecter avec les ports sortants via les connecteurs.
  - Cette contrainte va guider l'organisation du lancement des applications BCM à déploiements statiques pour garantir un ordre permettant des interconnexions sûres.
- Pour cela, la machine virtuelle BCM permet de créer, de connecter et d'enregistrer les composants pour les gérer selon leur cycle de vie.
- Ensuite, elle les *active* selon un ordre ***non déterministe*** en suivant ce cycle de vie (important pour le passage au réparti !).

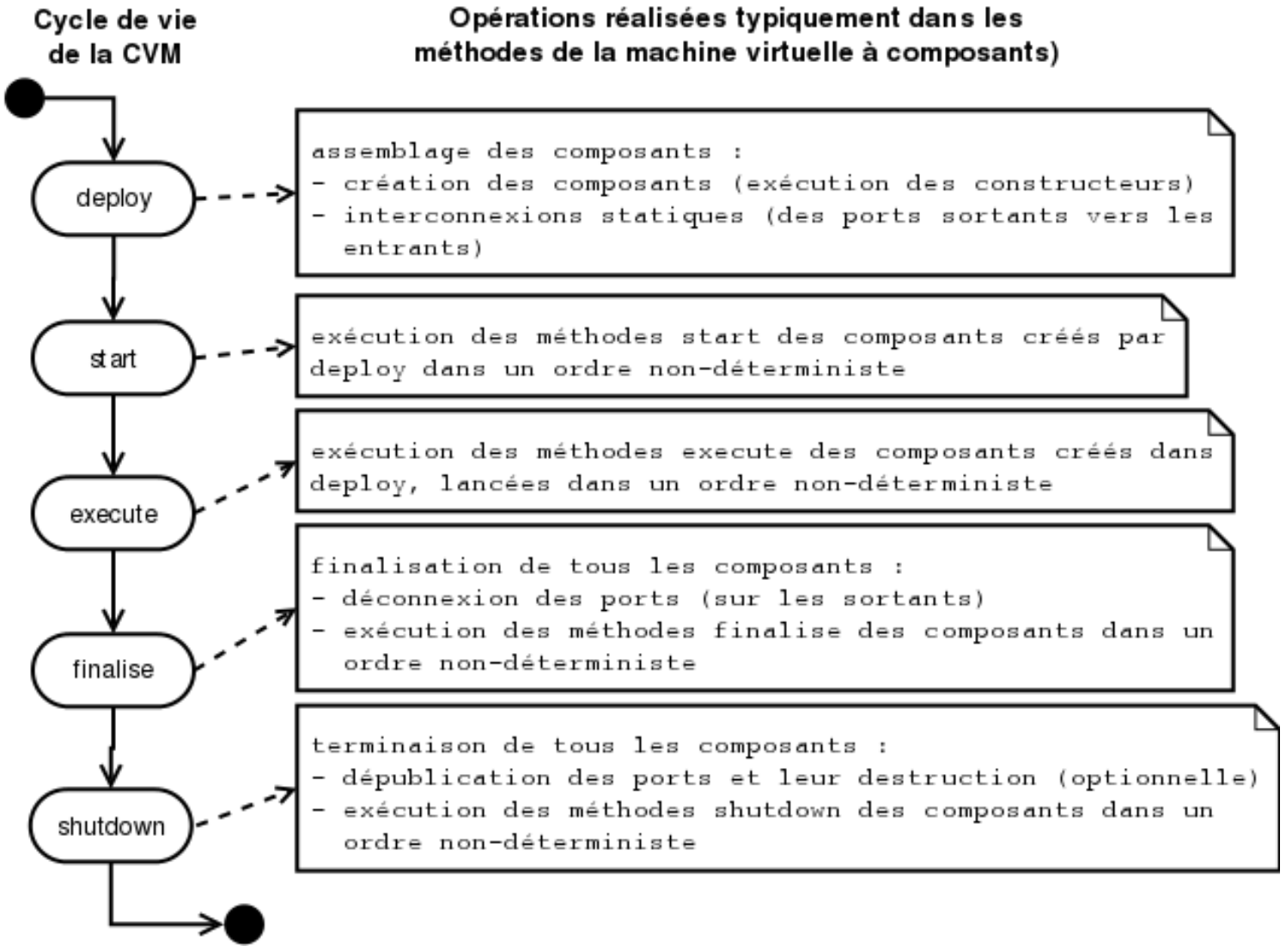
CC BY NC ND

Machine virtuelle à composants

- BCM propose deux manières de déployer des composants :
  - statiquement : tous les composants sont créés (et possiblement interconnectés) avant leur démarrage ;
  - dynamiquement : des composants sont (aussi) créés et interconnectés au fil de l'exécution, ce qui sera vu plus tard.
  - Puis, les composants suivent leur cycle de vie.
- En exécution mono-processus, un déploiement *statique* suit l'ordre suivant *i.e.*, son cycle de vie :
  1. Créer les composants, en supposant que ces derniers vont alors créer et publier leurs ports.
  2. Interconnecter statiquement les ports que les composants ont créés précédemment.
  3. Lancer le cycle de vie de chacun des composants (ils pourront alors prendre la main et faire des choses comme compléter leurs interconnexions, voir plus loin).
  4. Arrêter l'application en arrêtant la machine virtuelle (déconnecter les ports lors de la finalisation et les dépublier lors de l'arrêt).

cc BY NC ND

Cycle de vie de la machine virtuelle en mono-JVM



BY NC ND

Création et interconnexion statique des composants I

- Pour définir un assemblage de composants ou une application, il faut créer une sous-classe, disons `CVM`, d'`AbstractCVM`, cette dernière fournissant les méthodes implantant les opérations de la machine virtuelle à composants.
- La création et l'interconnexion statique des composants se fait en redéfinissant la méthode `deploy`.
- Les composants sont créés par la méthode `AbstractComponent#createComponent`, comme on l'a déjà vu :

```
String uri = AbstractComponent.createComponent(
                MonComposant.class.getCanonicalName(), new Object[]{1, 0});
```

- On peut ensuite les interconnecter statiquement par la méthode `AbstractCVM#doPortConnection`, en utilisant l'URI produite :

```
this.doPortConnection(
      uri,                              // URI retournée par createComponent
      OUTBOUND_PORT_URI,                // URI du port sortant du client "uri"
      INBOUND_PORT_URI,                 // URI du port entrant du fournisseur
      C.class.getCanonicalName());      // nom de la classe de connecteur C
```

CC BY NC ND

Création et interconnexion statique des composants II

- La machine virtuelle à composants lance alors toutes les méthodes `execute` des composants créés dans un ordre non-déterministe comme tâches sur leurs fils d'exécution.
- Lorsque la durée d'exécution prévue est atteinte, la machine virtuelle exécute sa méthode `finalise` qui lance les méthodes `finalise` de tous les composants créés.
  On doit alors redéfinir la methode `finalise` dans `CVM` pour déconnecter les ports sortants des ports entrants préalablement connectés :

```
this.doPortDisconnection(
      uri,                    // URI retournée par createComponent
      OUTBOUND_PORT_URI);     // URI du port sortant du client "uri"
```

- Ensuite, la machine virtuelle lance les méthodes `shutdown` de tous les composants créés dans lesquelles les ports seront dépubliés.

cc BY NC ND

Connexions/déconnexions internes et externes

- Nous avons maintenant introduit deux manières de connecter et déconnecter les ports :
  - `AbstractCVM#doPortConnection` dans la méthode `deploy` de la machine virtuelle (externe aux composants).
  - `AbstractComponent#doPortConnection` dans les composants.

  Les déconnexions correspondantes sont faites par `AbstractCVM#doPortDisonnection` et `AbstractComponent#doPortDisconnection`
- Chacune de ces approches *statiques* a ses avantages et inconvénients :
  1. *Externe* : *i.e.*, dans la machine virtuelle (méthode `deploy`), qui permet d'exposer en un seul endroit toute l'architecture de composants de l'application mais est plutôt adapté à des architectures statiques.
  2. *Interne* : *i.e.*, dans les composants (méthode `start`), qui permet d'encapsuler les connexions nécessaires au sein du composant et de passer à des architectures dynamiques (à voir plus tard) mais qui rend plus opaque l'architecture de composants parce qu'elle la dissémine à l'intérieur des composants.

BY NC ND

Lancement de l'application

- Le lancement du cycle de vie se fait en appelant la méthode `main` définie dans une classe `CVM`.
- Dans `main`, il faut créer une instance de `CVM` et appeler sur cette dernière la méthode `startStandardLifeCycle`.
- L'arrêt d'une application Java utilisant des groupes de fils d'exécution n'est pas simple et cela reste un défaut de l'implantation actuelle de BCM4Java. Pour simplifier les choses :
  - on fixe une durée d'exécution (en millisecondes) qui est donnée en paramètre à `startStandardLifeCycle`,
  - puis on force l'arrêt par un appel à `System.exit(0)`.

```
CVM c = new CVM();
c.startStandardLifeCycle(10000L);   // durée de 10 secondes
System.exit(0);
```

- Cette méthode appelle les méthodes du cycle de vie de la machine virtuelle vues précédemment (`deploy`, `start`, `execute` et, après la durée fixée, `finalise` puis `shutdown`).

BY NC ND

Plan

BY NC ND

Objectifs de la séquence

1. Objectifs pédagogiques
   - Comprendre les difficultés de la mise au point (*debugging*) des programmes utilisant plusieurs *threads*.
   - Comprendre les principes des aides à la mise au point des programmes de BCM : modes de *debugging*, traces et journalisation.
2. Compétences à acquérir
   - Savoir configurer et utiliser les fonctions de trace des composants BCM.
   - Savoir configurer et utiliser les fonctions de journalisation (*logging*) de BCM.
   - Savoir utiliser et étendre les modes de *debugging* de BCM.

cc BY NC ND

Mise au point des applications concurrentes

- La mise au point des applications concurrentes en général est notoirement difficile.
  - Contrairement aux application séquentielles, il est très difficile d'utiliser les outils comme le *debugger* d'Eclipse car si on arrête l'exécution d'un *thread* par un *breakpoint*, les autres *threads* vont poursuivre leur exécution ce qui change totalement l'ordre d'exécution des instructions du programme.
- C'est donc aussi le cas des applications BCM !
- De fait, pour comprendre une exécution fautive, le moyen le plus courant est d'introduire des instructions d'impression (`System.out.println` ...) dans les programmes pour tracer l'évolution de l'état, cerner la faute depuis l'endroit où elle se manifeste.
- Il demeure toutefois difficile de connaître l'ordre précis selon lequel les instructions des différents *threads* s'exécutent, et c'est pire encore en exécution multi-processus.

BY NC ND

Mécanisme de journalisation de BCM I

- Pour faciliter la mise au point, BCM offre un mécanisme de journalisation simple (*log*) attaché à chaque composant.
  - Ce mécanisme est activé et désactivé, composant par composant, par la méthode `toggleLogging`.
  - Les entrées de journalisation sont produites par la méthode `logMessage` d'`AbstractComponent` qui prend une entrée (chaîne) et l'ajoute au journal avec l'heure système :
    `1612948041565|starting provider component`
  - Les entrées sont arrangées de telle façon que, si on les voit comme formant un fichier, elles ont un format `csv` qui pourra être lu par un tableur.
    Ce sera encore plus utile pour les exécutions multi-processus.
- `AbstractComponent` propose les méthodes :
  - `setLogger` qui prend en paramètre une instance de la classe `Logger` (fournie par BCM4Java mais personnalisable), laquelle est créée avec les informations sur le répertoire à utiliser, etc.

CC BY NC ND

Mécanisme de journalisation de BCM II

- `printExecutionLog` prenant un nom de fichier en paramètre et écrit toutes les entrées dedans, en tenant compte du contexte initialisé dans le « *logger* ».

- Il faut donc que le composant définisse son « *logger* » puis qu'on active la journalisation et qu'à la fin (dans `finalise`), on appelle `printExecutionLog`.
  - Nota : les entrées sont accumulées en mémoire pour gêner le moins possible l'exécution du programme, ce qui pourrait causer des problèmes de consommation mémoire si trop d'entrées sont produites.
  - De nos jours, la taille des mémoires est cependant très importante.

BY NC ND

Mécanisme de trace de BCM

- Le mécanisme de journalisation est plutôt orienté vers l'examen *post-mortem* en cas d'erreur ; mais pour suivre l'exécution, BCM offre un *mécanisme de trace*.
  - Ce mécanisme s'active également composant par composant par la méthode *toggleTracing* et un message est produit par `traceMessage` prenant une chaîne en paramètre.
  - Si le mécanisme de journalisation et le mécanisme de trace sont tous les deux activés, les entrées de journalisation sont à la fois tracées et conservées pour être récupérées dans une fichier.
- BCM4Java offre deux classes de « traceurs » :
  - `TracerConsole` dont les traces seront affichées dans la console active du processus.
  - `TracerWindow` dont les traces sont affichées dans une nouvelle fenêtre associée au composant.
  - L'association d'un traceur à un composant se fait par `setTracer`.
  - Des méthodes de `TracerWindow` permettent de configurer la fenêtre, sa taille, sa position relative dans l'écran, etc.

BY NC ND

Bonnes pratiques I

- Introduire des instructions de journalisation ou de trace dans un programme est un travail long, fastidieux et répétitif.
- Plutôt que de les ajouter et retirer au besoin, on définit des modes de *debugging* et on met ces instructions dans des alternatives (`if`) pour ne les exécuter qu'en fonction du mode de *debugging* courant.
- Pour cela, BCM introduit dans `AbstractCVM` une variable globale `DEBUG_MODE` de type `Set<CVMDebugModesI>` et un type énumération `CVMDebugModes` implantant `CVMDebugModesI`.
  - Ce mécanisme est utilisé dans BCM pour produire des entrées de journalisation et des traces sélectives.
  - `AbstractCVM` propose aussi une méthode `logDebug` et un *logger* spécifique pour faire la journalisation des entrées de *debugging* qui est utilisé dans son propre code (hors de tout composant).

BY NC ND

Bonnes pratiques II

- Parmi les modes définis par `CVMDebugModes`, il y a par exemple le mode `PUBLISHING` qui active la journalisation des opérations de publication des ports dans le *framework*.
- Un autre mode est `CALLING` prévu pour activer les opérations de transmission des appels dans les ports et les connecteurs. Pour introduire une journalisation conditionnelle dans ses ports, l'utilisateur peut écrire dans les méthodes de ces derniers :

```
// activation du mode de debugging CALLING
AbstractCVM.DEBUG_MODE.add(CVMDebugModes.CALLING);
...
if (AbstractCVM.DEBUG_MODE.contains(CVMDebugModes.CALLING) {
  // utilisation de la journalisation si CALLING est activé
  AbstractCVM.logDebug(CVMDebugModes.CALLING, message);
}
```

- Pour constater l'effet de ces mécanismes à l'exécution voir `fr.sorbonne_u.components.examples.basic_cs.CVM` et suivre les indications dans les commentaires.

BY NC ND

Ajout de nouveaux modes

- Il est possible pour un développeur d'étendre ces types pour introduire ses propres modes pour son application et ainsi capitaliser sur l'introduction d'instructions de journalisation conditionnelles dans son code activables à volonté lorsqu'une faute se manifeste.
- Il faut d'abord définir une nouvelle énumération qui implante l'interface `CVMDebugModesI`, par exemple :

```
public enum MyDebugModes implements CVMDebugModesI {
    MY_MODE
}
```

Ensuite, on peut ajouter ce mode à la variable globale :

```
AbstractCVM.DEBUG_MODE.add(MyDebugModes.MY_MODE);
```

Et enfin, l'utiliser dans une instruction de journalisation conditionnelle :

```
if (AbstractCVM.DEBUG_MODE.contains(MyDebugModes.MY_MODE) {
    this.logMessage(...);

}
```

CC BY NC ND

Plan

CC BY NC ND

Objectifs de la séquence

1. Objectifs pédagogiques
   - Comprendre le besoin de réutilisation de code en programmation.
   - Comprendre la difficulté posée par BCM pour la réutilisation.
   - Comprendre les principes du mécanisme de greffons (*plug-ins*) de BCM.
   - Comprendre comment modéliser des rôles réutilisables pour les composants comme des greffons.
2. Compétences à acquérir
   - Savoir programmer des greffons en BCM et utiliser correctement leur cycle de vie.
   - Savoir utiliser des greffons sur des composants.

cc BY NC ND

Réutilisation et composants

- La réutilisation du code au niveau source est une préoccupation importante en informatique qui a été particulièrement mis en avant par la programmation par objets.
  - Bien sûr, la réutilisation par la simple récupération d'un morceau de code appelé dans un programme, que ce soit une bibliothèque ou une classe est aussi offerte en programmation par composant par récupération et utilisation d'un composant déjà programmé.
- On recherche plutôt un mécanisme de réutilisation de code source existant comparable à l'héritage entre les classes.
- Mais les modèles à composants n'ont pas de manière standard pour les définir (pas de description ou de « classe » de composants ni *a fortiori* de mécanisme d'héritage).
  - Pour BCM, s'ajoute une difficulté technique : on a bien une classe Java définissant un composant mais elle doit hériter d'`AbstractComponent` or, Java étant à héritage simple, une classe ne peut hériter que d'une seule superclasse, ce qui en limite l'utilisation.

BY NC ND

Patron de conception « Délégation » I

[1] *héritage simple : une seule superclasse ; héritage multiple : plusieurs superclasses.*

- Le fait que Java ne propose que l'héritage simple[1] a amené les développeurs à réfléchir à des moyens d'obtenir une forme de réutilisation par des mécanismes alternatifs à l'héritage.
- Le mécanisme souvent utilisé pour ce faire est la *délégation* [2].
  - L'idée de la délégation est de partager le travail entre l'objet o que l'on souhaite créer et des objets délégués $p_1, \ldots, p_n$ qui vont représenter différents *fragments*, avec leurs propres méthodes, auxquels o va déléguer les appels qui les concernent.
- Exemple : voiture volante
  - Supposons qu'on veuille créer une classe « voiture volante » pouvant rouler ou voler alors que nous avons déjà programmé deux classes `Voiture` qui roulent et `Avion` qui volent.
  - Il serait intéressant de réutiliser le code de ces deux classes pour créer la classe `VoitureVolante` mais en Java cette classe ne peut hériter à la fois de `Voiture` et `Avion`.

CC BY NC ND

Patron de conception « Délégation » II

- La délégation propose, par exemple, de choisir l'une des deux classes dont on va hériter et d'utiliser une instance de l'autre classe pour lui déléguer les appels qu'elle sait exécuter.
- Par exemple, on peut créer `VoitureVolante extends Voiture` et déclarer une variable `avionDelegue` de type `Avion`.

```
public class VoitureVolante extends Voiture  // héritage
{
  protected Avion avionDelegue;               // le délégué

  // délégation de voler à l'instance d'avion
  public void voler()  { this.avionDelegue.voler(); }

  // traitement de rouler par la superclasse Voiture
  // (définition inutile, juste là pour comprendre)
  public void rouler() { super.rouler(); }
}
```

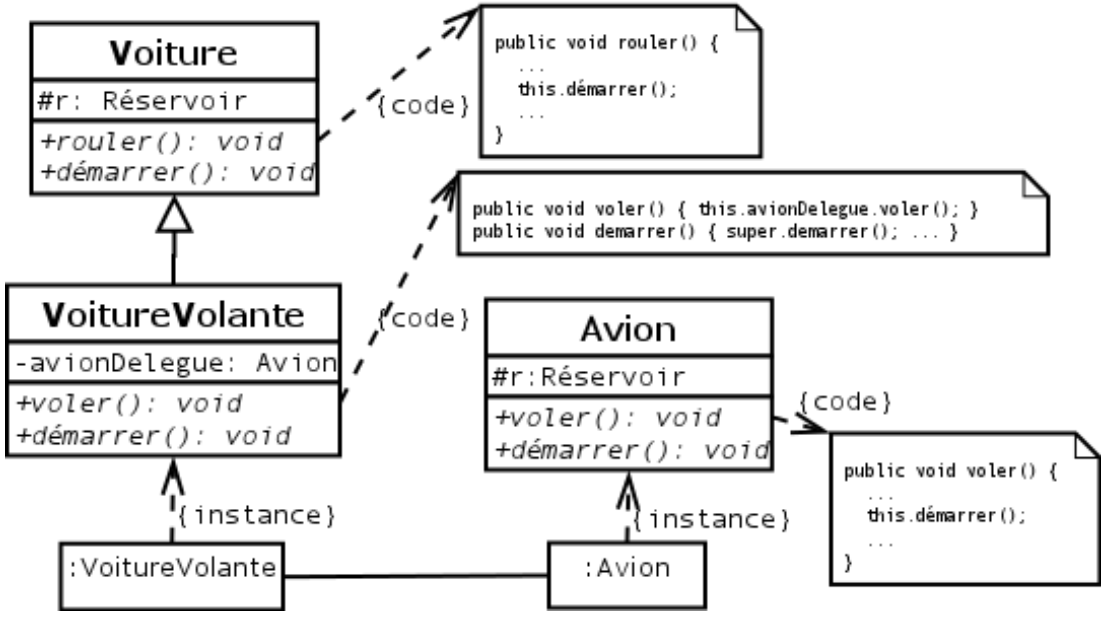
[2] Cette acception du terme *délégation* ne doit pas être confondue avec le mécanisme homonyme des *langages à prorotypes* dans lesquels il existe effectivement l'équivalent d'un `this`, ce qui n'est pas le cas ici en Java ni dans les différents patrons de conception fondés sur la délégation (*Adapter*, *Decorator*, etc.). Personnellement, je considère que ces patrons doivent être utilisés qu'avec une extrême circonspection car ils produisent inévitablement du code *spaghetti*.

BY NC ND

Limites du mécanisme de délégation I

- Pas d'équivalent du `this` : une méthode d'*Avion*, appelée par `voler` (par exemple), ne peut pas être redéfinie dans `VoitureAvion` contrairement à une méthode de `Voiture`.

La méthode `démarrer` de `VoitureVolante` est appelée par la méthode `rouler` héritée de `Voiture` par `VoitureVolante` mais pas par la méthode `voler` d'`Avion`.



- Il vaut mieux hériter de la classe dont on doit redéfinir des méthodes mais si les deux classes exigent des redéfinitions, *la délégation ne fonctionne pas* (du moins pas directement).

CC BY NC ND

Limites du mécanisme de délégation II

- Perte de l'unicité de l'objet : l'objet voiture volante est *fragmenté* en deux objets, l'instance de `VoitureVolante` et celle d'`Avion`.
  - Peut poser de sérieux problèmes de conception par exemple, a-t-on deux réservoirs de carburant ?
    Voir le diagramme de la page précédente et notez l'existence de deux variables `r`, l'une dans `:VoitureVolante` et l'autre dans `:Avion`.
  - Il faudra faire attention à ne jamais séparer ces deux objets, par exemple lors d'une mise en base de données.
  - Il ne faudra jamais laisser fuiter la référence à d'`:Avion` hors de `:VoitureVolante` sinon, elle pourrait être utilisée pour casser les invariants qui existent entre ces deux instances représentant en fait une unique entité du domaine d'application.

CC BY NC ND

Introduction aux greffons de BCM

- Le mécanisme de greffons de BCM utilise des objets délégués, conçus spécifiquement, à l'*intérieur* des composants pour réutiliser le code qui y est programmé.
  - Les instances de greffons connaissent leur composant et ne peuvent être attachés qu'à un seul composant.
  - Ainsi, on délègue des opérations du composant à des greffons et les greffons délèguent à leur composant les opérations internes comme l'ajout d'interfaces et de ports aux composants.
- Un composant pourra avoir plusieurs greffons, y compris plusieurs greffons du même type.
  - Un greffon d'un composant y sera désigné par une URI qui va ensuite permettre d'y accéder par la méthode `getPlugin(String)`.
- Un greffon est défini par une classe Java qui hérite (directement ou indirectement) de la classe abstraite `AbstractPlugin`.
  - Cette classe définit les méthodes propres aux greffons et à leur gestion, y compris leur *cycle de vie*.

BY NC ND

Création d'un greffon et attachement à un composant

- Définir un greffon demande de définir une classe, par exemple `G`, héritant d'`AbstractPlugin` (qui elle-même implante `PluginI`).
- À partir de `G`, une instance de ce greffon est créée par `new` mais il requiert encore des initialisations pour être utilisable.
  - Son URI doit être fixée : elle est attribuée par appel de la méthode `AbstractPlugin#setPluginURI` ; cette URI pourra ensuite être utilisée pour récupérer le greffon dans le composant.
  - Son composant de rattachement n'est pas encore connu : on doit l'installer sur un composant en appelant la méthode `AbstractComponent#installPlugin`. Cette méthode appelle les méthodes du cycle de vie du greffon pour compléter son intialisation et l'inscrit parmi ses greffons.
- Une méthode `AbstractComponent#uninstallPlugin` permet de désinstaller un greffon dynamiquement, sinon il sera désinstallé lors de l'extinction et de la destruction du composant.

cc BY NC ND

Cycle de vie d'un greffon

- Le cycle de vie d'un greffon est défini par quatre méthodes d'`AbstractPlugin` qui sont appelées dans l'ordre suivant par le composant (les deux premières par `installPlugin`, les deux dernières par `finalise` et `shutdown` respectivement)[1] :
  1. `installOn(ComponentI)` : attribue au greffon son composant de rattachement ; c'est le bon endroit pour ajouter des interfaces de composants requises ou offertes.
  2. `initialise()` : initialise le greffon ; c'est le bon endroit pour créer des ports et les connecter ainsi qu'initialiser ses variables.
  3. `finalise()` : appelée avant la désinstallation, c'est souvent le bon endroit où déconnecter les ports et libérer des ressources accaparées par le greffon.
  4. `uninstall()` : appelée pour désintaller le greffon ; c'est souvent le bon endroit pour dépublier et détruire les ports puis se retirer les interfaces de composants.

[1] En fait, d'une certaine manière, `installOn` est similaire à un constructeur pour un composant, `initialise` est similaire à `start`, `finalise` est similaire à la méthode homonyme et `uninstall` est similaire à `shutdown`.

BY NC ND

Un exemple complet : utilisation d'une table de hachage

1. Considérons un exemple de composant table de hachage `MapComponent` qui offre une interface `MapCI` avec les méthodes usuelles `put`, `get`, `containsKey` et `remove`.
2. Ce composant doit :
   - implanter les méthodes de service correspondant à `put`, `get`, `containsKey` et `remove`,
   - ajouter l'interface `MapCI` à ses interfaces offertes,
   - créer, publier et s'ajouter un port entrant `MapInboundPort`,
   - puis, après utilisation, dépublier et détruire son port entrant ainsi que retirer l'interface offerte `MapCI`
3. Pour utiliser `MapComponent`, un composant client doit :
   - ajouter l'interface `MapCI` à ses interfaces requises,
   - créer, publier et s'ajouter un port sortant `MapOutboundPort`,
   - connecter son port sortant au port entrant de `MapComponent`,
   - puis, après utilisation, déconnecter, dépublier et détruire son port sortant avant de se retirer l'interface requise `MapCI`.

- Ces opérations devraient être abstraites et réutilisées.

BY NC ND

Un exemple complet : passage à des greffons

- Deux *rôles* : le rôle de serveur et celui de client, ce qui sera souvent le cas pour des greffons entre deux composants.
  - Le rôle serveur n'est pas réellement utile dans cet exemple, mais cela nous permet ici d'illustrer aussi les greffons côté serveur.
- On va donc créer deux greffons :
  1. `MapPlugin` qui implante le rôle de serveur (côté `MapComponent`),
  2. `MapClientPlugin` qui implante le rôle côté client.
- La passage aux greffons consiste à :
  - déplacer le code ajoutant les interfaces dans les méthodes `installOn` des deux greffons,
  - déplacer le code créant les ports dans leurs méthodes `initialise` et, côté client, y connecter le client au serveur,
  - déplacer le code de déconnexion dans la méthode `finalise` du greffon client,
  - déplacer le code dépubliant et détruisant les ports et retirant les interfaces dans les méthodes `uninstall` des deux greffons, et
  - ajouter le code créant les greffons dans les constructeurs des composants, selon leur rôle.

CC BY NC ND

Diagramme de classes de l'exemple

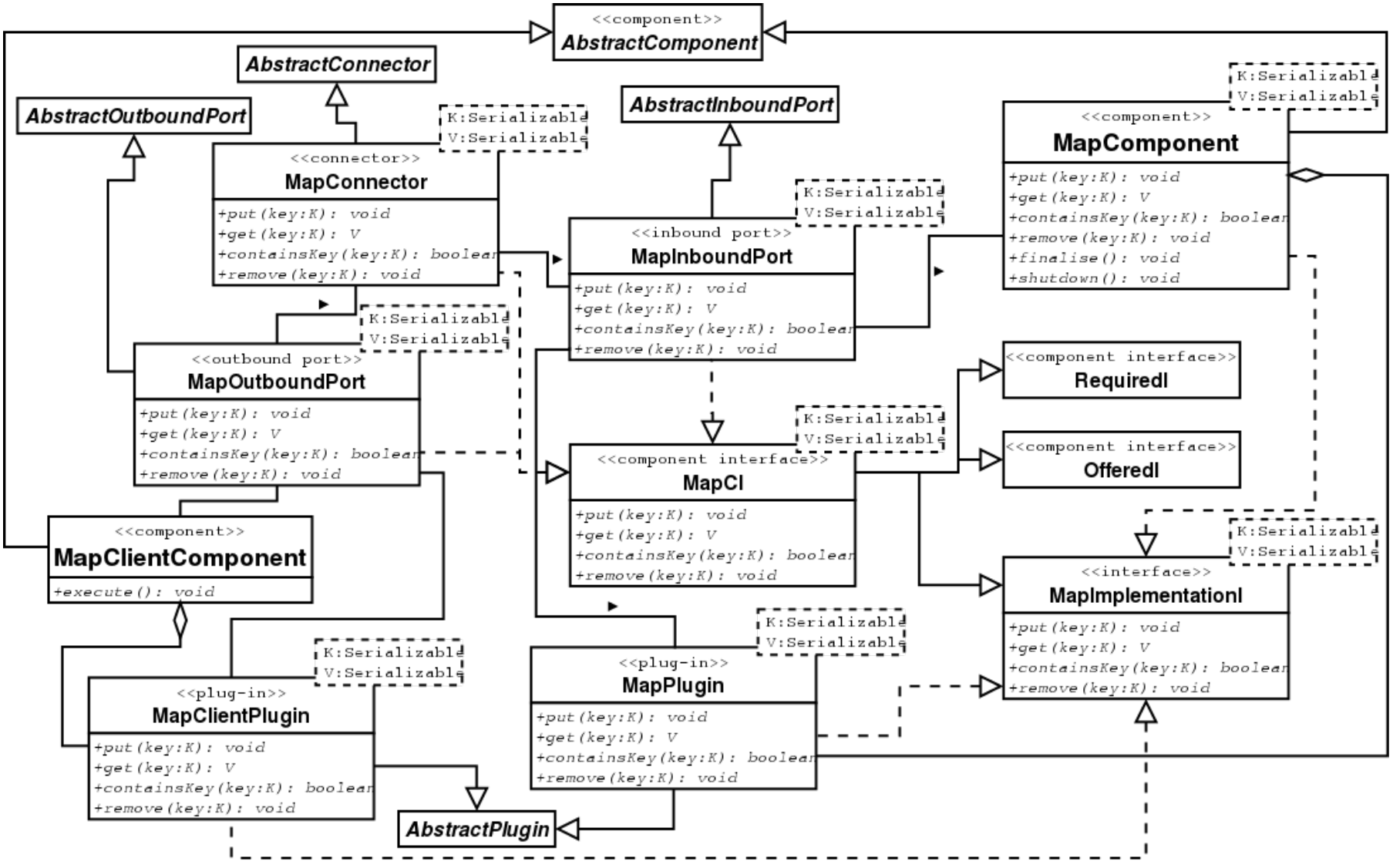
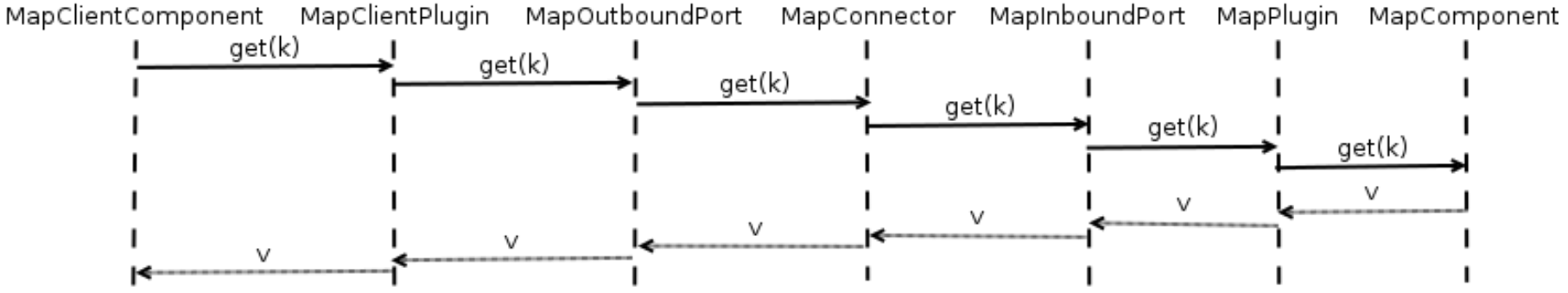


Diagramme de séquence pour l'exemple

- Les greffons utilisant un patron délégation, ils ajoutent des indirections dans l'appel entre deux composants.
- Dans notre exemple, comme il y a des greffons à la fois côté client et côté serveur, c'est d'autant plus visible.
- Voici le diagramme de séquence (simplifié) d'un appel à `get` depuis le code côté client jusqu'à la méthode d'implantation côté serveur puis le rétour du résultat :



- Mais cela donne la flexibilité permettant la réutilisation et masquant aux composants les détails de la connexion.
  En informatique, *indirection* $\Rightarrow$ *+flexibilité*, mais *+complexité*.

CC BY NC ND

Code de l'exemple

- Lire le code de l'exemple fourni sur le site.
- Observez :
  - Comment les composants programmées de manière classique `fr.sorbonne_u.cps.map.components.MapComponent` et `fr.sorbonne_u.cps.map.components.MapComponentClient` créent directement leurs ports et déclarent les interfaces requises et offerts.
  - Comment ces créations sont factorisées dans les greffons `fr.sorbonne_u.cps.map.withplugin.plugins.MapPlugin` et `fr.sorbonne_u.cps.map.withplugin.plugins.MapPluginClient` récupèrent ce code en passant à la version avec greffons.
  - Et comment dans cette version avec greffons tout ce code a été remplacé dans les composants `fr.sorbonne_u.cps.map.withplugin.components.MapComponent` et `fr.sorbonne_u.cps.map.withplugin.components.MapComponentClient` par la création et l'installation des greffons.

BY NC ND

Utilisation du greffon côté client

- Le code du greffon `MapClientPlugin` peut être examiné dans l'exemple fourni. Pour créer l'instance de greffon et l'installer dans le composant client, on procède comme suit :

```
MapClientPlugin<String,Integer> plugin = new MapClientPlugin<>();
plugin.setPluginURI(MY_PLUGIN_URI);
this.installPlugin(plugin);
```

- Pour appeler le composant map via son greffon, il suffit de conserver une référence sur l'objet représentant le greffon et l'utiliser comme n'importe quel objet Java dans le composant :

```
plugin.put("a", 1);
System.out.println("" + plugin.containsKey("a"));
```

- Pour désinstaller explicitement le greffon, le composant appelle `uninstallPlugin` :

```
this.uninstallPlugin(plugin.getPluginURI());
```

BY NC ND

Utilisation du greffon côté fournisseur

- Le greffon côté fournisseur est `MapPlugin` et son instance est créée et installée dans le composant `MapComponent` :

```
MapPlugin<String,Integer> plugin = new MapPlugin<>();
plugin.setPluginURI(MAP_PLUGIN_URI);
this.installPlugin(plugin);
```

- D'autre part, le port entrant doit diriger ses appels vers le greffon et non le composant. Pour cela, le greffon lui passe son URI lorsqu'il le crée :

```
this.mip = new MapInboundPortForPlugin<K,V>(
                    this.getOwner(), this.getPluginURI());
this.mip.publishPort();
```

- Et dans le port entrant, voici l'appel au greffon :

```
this.getOwner().handleRequest(
  new AbstractComponent.AbstractService<Void>(this.getPluginURI()) {
    @Override
    public Void call() throws Exception {
      ((MapPlugin<K,V>) this.getServiceProviderReference()).put(key, value);
      return null;
    }});
```

CC BY NC ND

Greffons standards fournis dans BCM

- BCM offre une bibliothèque (encore limitée) de greffons standards :
  - Connexion dynamique : permet de connecter dynamiquement deux composants via n'importe quelle interface offerte et requise sans connaître à l'avance les URIs des ports concernés.
    Les greffons définis dans `fr.sorbonne_u.components.plugins.dconnection` et un exemple dans `fr.sorbonne_u.components.plugins.dconnection.example`.
  - Contrôle du mode *push* des ports d'échanges de données.
    Les greffons définis dans `fr.sorbonne_u.components.plugins.dipc` et un exemple dans `fr.sorbonne_u.components.plugins.dipc.example`.
- Cette bibliothèque offre aussi une classe de greffon abstraite facilitant les opérations de connexion dynamique côté client qui sont en fait un peu toujours les mêmes ; cette classe peut être héritée par celle de tout greffon de rôle côté client (`fr.sorbonne_u.components.plugins.helpers.AbstractClientSidePlugin`).

BY NC ND

Plan

cc BY NC ND

Objectifs de la séquence

1. Objectifs pédagogiques
   - Comprendre la notion de sous-composants et la relation avec les composants « composites ».
   - Comprendre la réalisation du concept de sous-composants et les contraintes imposées par BCM à leur créationainsi que les raisons d'être de ces dernières.
2. Compétences à acquérir
   - Savoir créer un sous-composant dans un composant composite.
   - Savoir définir le cycle de vie d'un sous-composant et sa relation avec le cycle de vie de son composite.
   - Savoir comment appeler un sous-composant à partir du composite.
   - Savoir récupérer dans le composant composite un appel fait via un port sortant par un de ses sous-composants.

BY NC ND

La notion de sous-composant

- L'assemblage des composants nous amène naturellement à imaginer qu'un composant complexe puisse résulter de l'assemblage de *sous-composants*.
  - Exemple : un composant table résultant de l'assemblage d'un composant plateau et de quatre composants pieds.

  On parle alors de *composant composite* ou simplement de *composite*.
- Cette possibilité crée une relation « *est sous-composant de* » récursive (un sous-composant lui-même obtenu par assemblage de sous-sous-composants, et ainsi de suite).
  - Cette relation est tout à fait similaire à la relation « *est partie de* » et la hiérarchie d'objets qui en résulte en programmation par objets aussi appelée *hiérarchie de parties*.
  - Elle est aussi similaire à la relation de composition d'UML (dite aussi *aggrégation forte*).

cc BY NC ND

Questions soulevées par ce concept

- Plus concrètement, l'introduction de sous-composants dans un modèle à composants soulève quelques questions :
  - Comment sont liés les cycles de vie des composites et ceux de leurs sous-composants ?
  - Les sous-composants sont-ils visibles à l'extérieur de leurs composites ? Est-il possible pour un composant tiers de se connecter directement à un port d'un sous-composant ?
  - Quelle relation existe-t-il entre les *threads* du composite et ceux de ses sous-composants ?
- BCM choisit l'intégration forte entre composites et sous-composants ainsi que l'abstraction (composite = boîte noire) :
  - les sous-composants demeurent dans le même espace mémoire que leur composite (*i.e.*, même JVM) ;
  - qui contrôle aussi leurs cycles de vie (synchonisés) ;

  mais :
  - leurs ports ne font pas partie des ports de leur composite ;
  - et la gestion de leurs *threads* est indépendante de celle du composite.

cc BY NC ND

Création d'un sous-composant en BCM

- La création d'un sous-composant se fait par la méthode `AbstractComponent#createSubcomponent`, similaire à `createComponent` :

```
String uri = this.createSubcomponent(
                MySubcomponent.class.getCanonicalName(),
                new Object[]{...});
```

- Cette opération ajoute le sous-composant à son composite, ce qui implique :
  - que le composite connaît ses sous-composants directs et que ces derniers connaissent leur composite immédiat ;
  - que le composite peut faire des opérations sur ses sous-composants en utilisant l'URI retournée ;
  - que les méthodes du cycle de vie du sous-composant sont appelées lorsque la méthode de même nom est appelée sur le composite (automatique dans `AbstractComponent`).
- Les sous-composants d'un composite peuvent s'interconnecter entre eux via ports et connecteurs.

cc BY NC ND

Appels entre composite et ses sous-composants

- Comment faire en sorte que le composite et un de ses sous-composants puissent s'appeler l'un l'autre ?
  - Comme pour les composants, les points de connexion des sous-composants sont des ports entrants et sortants.
  - Pour les utiliser, il faudrait que le composite se connecte à ses sous-composants en utilisant des ports :
    - des ports sortants du composite qui se connecteraient à des ports entrants de ses sous-composants ;
    - des ports sortants de ses sous-composants qui se connecteraient à des ports entrants du composite.
- Mais cette approche mènerait à exposer des ports sur le composite qui ne devraient pas être visibles de l'extérieur, ce qui entre en contradiction avec la notion de port.
- BCM utilise plutôt la relation particulière existant entre composite et sous-composant pour autoriser des « connexions » particulières, spécifiques au cas composite/sous-composant.

cc BY NC ND

Appel d'un sous-composant via un de ses ports entrants

- Pour appeler un de ses sous-composants, le composite doit toujours passer par un port entrant de ce dernier pour respecter la politique d'utilisation des *threads* de ce dernier.
- Ceci demande à pouvoir récupérer la *référence* au port entrant à partir de son URI.
  - normalement, cela n'est pas autorisé en BCM : le cas composite vers sous-composant devient donc un cas particulier ;
  - heureusement, la référence en question est nécessairement une référence Java locale à la JVM où le composite s'exécute car les sous-composants sont toujours déployés dans cette dernière.
- Pour récupérer cette référence, BCM offre la méthode

`AbstractComponent#findSubcomponentInboundPortFromURI` :

```
ServicesCI toSubcomponentInPort =
  (ServicesCI)  // "cast" interface offerte du sous-composant
      this.findSubcomponentInboundPortFromURI(
         subcomponentURI,   // URI du sous-composant
         inboundPortURI);   // URI du port du sous-composant
```

BY NC ND

Appel d'un composite depuis un de ses sous-composants

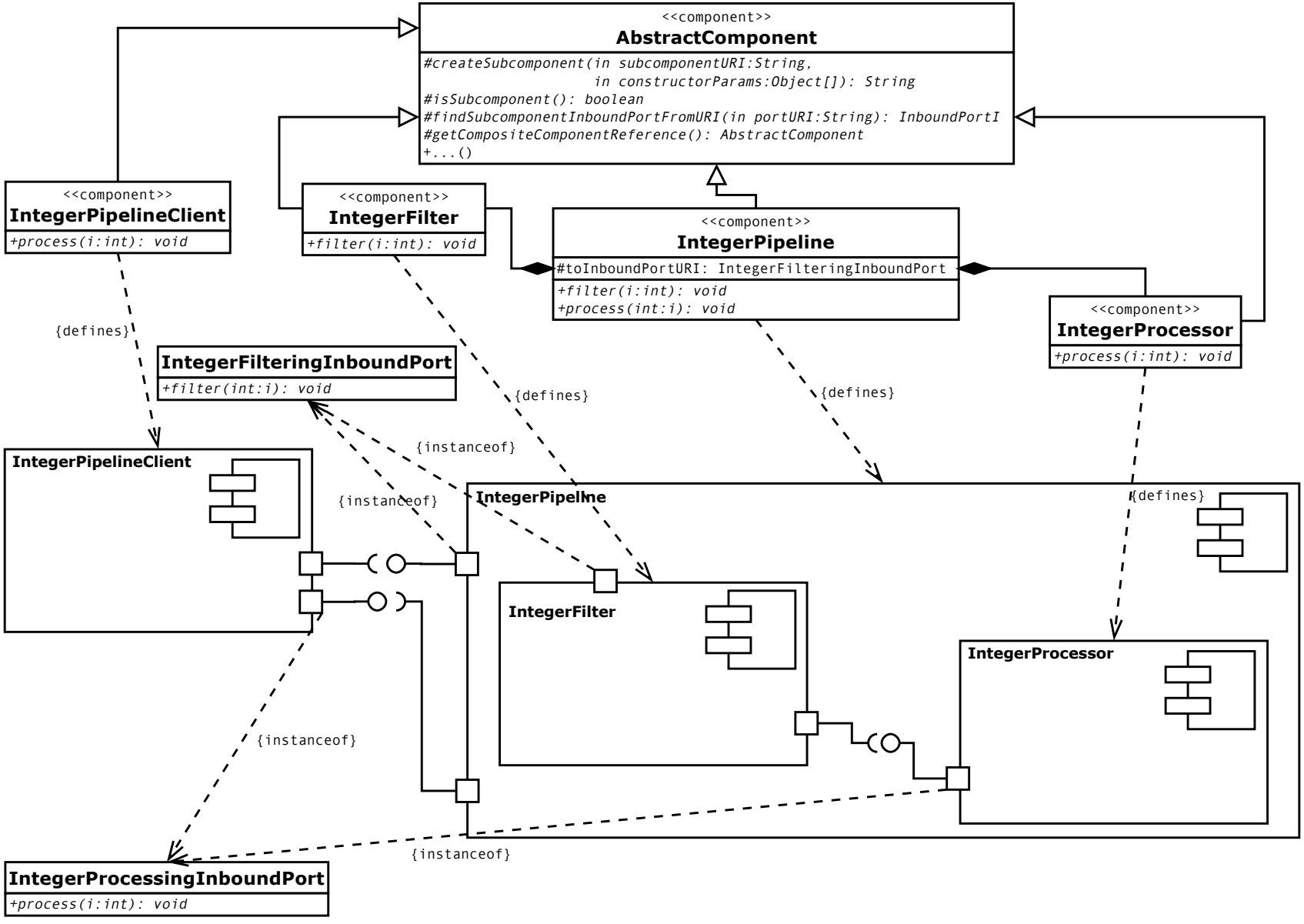
- Plaçons-nous maintenant du côté d'un sous-composant voulant appeler une méthode (service) de son composite ; ceci requiert :
  - d'avoir une référence sur l'objet représentant le composant, ce qui n'est pas autorisé normalement en BCM mais cela devient un cas particulier pour le composite direct d'un sous-composant ;
  - de lui passer une requête via `handleRequest` par exemple pour que l'exécution se fasse par les *threads* du composite.
- Pour récupérer la référence à l'objet représentant le composite, BCM offre la méthode `AbstractComponent#getCompositeComponentReference` :

  ```
  this.getCompositeComponentReference().handleRequest(...);
  ```

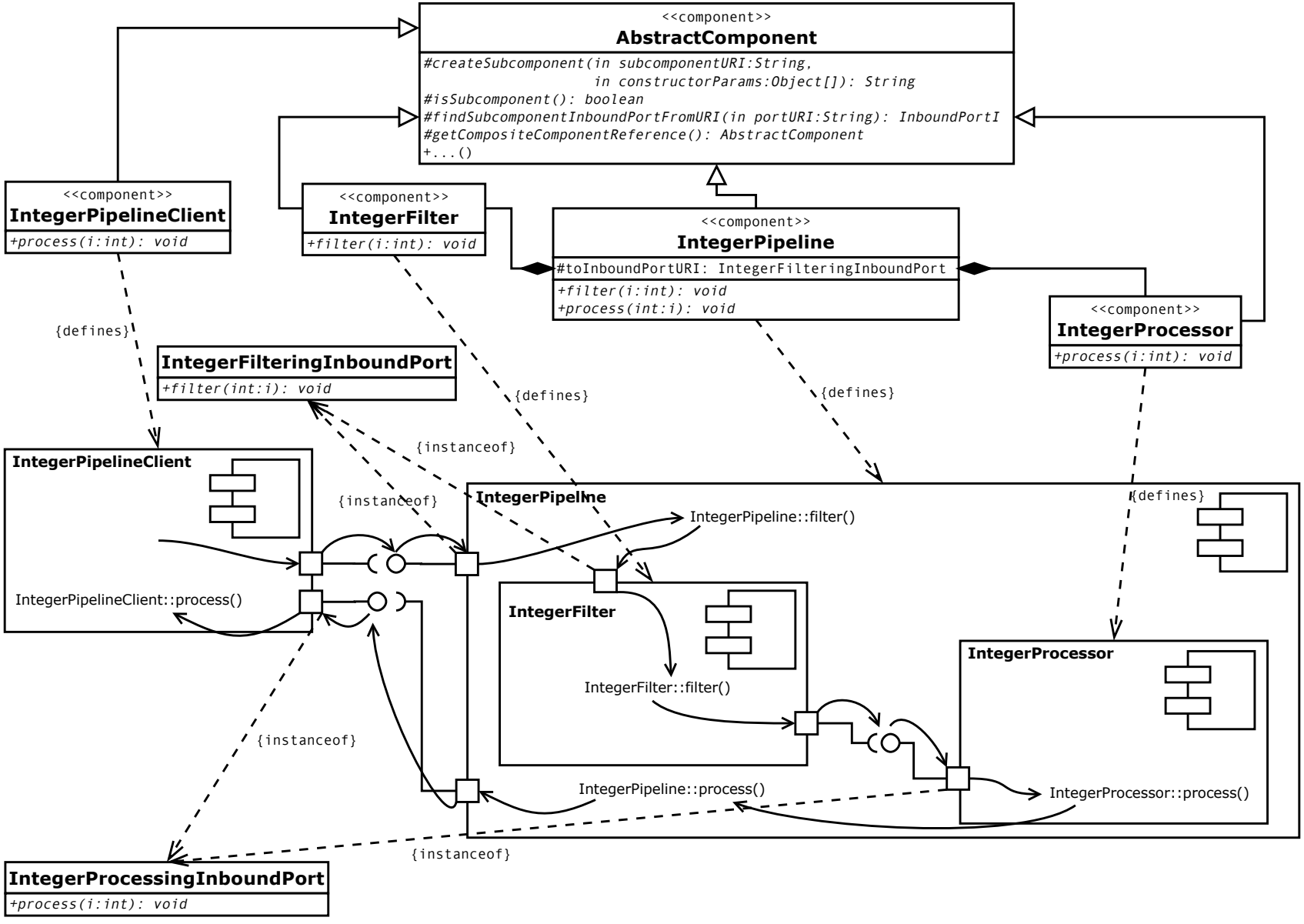
  avec un code qui ressemble à celui utilisé dans un port entrant pour passer la requête au composite.

CC BY NC ND

Exemple de pipeline de traitement de flôts d'entiers



Exemple de pipeline de traitement de flôts d'entiers



Précautions et réutilisabilité des sous-composants

- Cet exemple montre (comme le cas précédent) que la gestion des *threads* entre sous-composants et composite doit être faite de manière soigneuse.
- Dans cet exemple, le code suppose que les composants de types `IntegerFilter` et `IntegerProcessor` sont toujours des sous-composants du type de composite `IntegerPipeline`.
- En fait, cela n'est pas obligatoire, on pourrait utiliser le même type de composants tantôt dans un contexte de sous-composant, tantôt dans un contexte de composant.
  - Il s'agit alors surtout de bien gérer les appels faits au composite.
  - `AbstractComponent` propose une méthode `isSubcomponent` qui retourne vrai si le composant est dans un contexte où il est sous-composant.
  - Un composant peut donc être rendu compatible avec les deux contextes en testant le résultat de cette méthode pour choisir entre un appel au composite et un appel via un port sortant connecté à un composant tiers, par exemple.

cc BY NC ND

Activités à réaliser avant le prochain TME

Remarque importante

La lecture de code en CPS est une activité aussi importante pour acquérir les concepts et les techniques de programmation présentées que la préparation des examens dans d'autres UE. N'oubliez pas que l'évaluation du projet porte entre autres choses sur la qualité de votre code.

1. Examiner l'utilisation des logs et des techniques d'aide au *debugging* dans les exemples fournis avec BCM4Java.
2. Récupérer l'exemple de table de hachage avec greffons et examiner côte à côte cet exemple sans et avec greffons pour bien comprendre le passage de l'un à l'autre.
3. Examiner les exemples de greffons dans l'archive BCM4Java : connexion dynamique et contrôle dynamique du mode push dans les interfaces d'échange de données.
4. Examiner l'exemple `subcomp` dans le répertoire des sources `examples` de l'archive BCM4Java.

cc BY NC ND