

ALASCA — Architecture logicielles avancées pour les systèmes cyber-physiques autonomiques

© Jacques Malenfant

Master informatique, spécialité STL – UFR 919 Ingénierie

Sorbonne Université
Jacques.Malenfant@lip6.fr

Cours 2

Architectures logicielles dynamiquement adaptables

Objectifs pédagogiques du cours 2

- Introduire la notion de réflexivité logicielle pour répondre à une partie du cahier des charges de la fonction d'auto-adaptabilité des systèmes autonomiques.
- Comprendre et apprendre à utiliser la réflexivité structurelle de Java pour réaliser des opérations d'adaptation dynamique.
- Introduire la notion de réflexivité comportementale et sa déclinaison en Java.
- Comprendre et apprendre à utiliser la réflexivité comportementale de Java pour réaliser des opérations d'adaptation dynamique.
- Comprendre et apprendre à utiliser la réflexivité dans les modèles à composants comme BCM pour réaliser des opérations d'adaptation dynamique.

Plan

- 1 Réflexion et adaptabilité logicielle
- 2 Réflexivité structurelle en Java
- 3 Exemple : connecteur de composants par proxy
- 4 Réflexivité comportementale en Java/Javassist
- 5 Exemple : connecteur généré avec Javassist
- 6 Réflexivité en BCM

Architectures réflexives et programmation

- Concept lié à la modification dynamique des programmes, une pratique aussi ancienne que l'informatique elle-même.
- Conceptualisée en tant que discipline par Brian Smith dans sa thèse et ses articles au début des années 1980.
- Exemple : réflexion en Java
 - Java définit une grande partie des éléments constitutifs d'un programme comme les classes, les méthodes, etc., sous la forme d'objets (`java.lang.reflect`).
 - Java prévoit le chargement dynamique de code ainsi que des moyens pour remplacer le code (classes) chargé dans sa machine virtuelle (API JPDA).
 - Des bibliothèques externes permettent d'ajouter et de modifier dynamiquement du code compilé *i.e.*, les classes (ex.: Javassist).

Plan

- 1 Réflexion et adaptabilité logicielle
- 2 Réflexivité structurelle en Java
- 3 Exemple : connecteur de composants par proxy
- 4 Réflexivité comportementale en Java/Javassist
- 5 Exemple : connecteur généré avec Javassist
- 6 Réflexivité en BCM

Réflexion et architectures à base de composants

- Des modèles de composants, comme BCM et Fractal, prévoient des interfaces d'introspection et d'intro-action permettant
 - de découvrir et modifier les interfaces requises et offertes par les composants,
 - les ports et leurs interconnexions entre composants, ainsi que
 - d'autres informations sur le contenu et l'architecture de l'application.
- La réflexion dans une architecture à composants permet aussi :
 - de modifier cette architecture dynamiquement, en connectant, déconnectant, reconnectant,
 - en ajoutant, retranchant et remplaçant des composants pendant l'exécution,

et ainsi l'adapter à son état d'exécution courant.
- En BCM, la réflexion de Java et celle du modèle à composants permettent de modifier des composants et en ajouter de nouveaux dynamiquement.

Objectifs

- Le langage Java possède une API pour la réflexion structurelle fournie par les classes `Object` et `Class<T>` de même que le paquetage `java.lang.reflect`.
- Pour les classes, chaque classe est représentée par une instance de `Class<T>` permettant au programme d'examiner :
 - les champs et les méthodes définis par la classe,
 - la classe dont elle hérite,
 - les interfaces implantées,
 - ses propriétés (modificateurs, ...).
- Le paquetage `java.lang.reflect` contient des classes permettant de représenter chacun des constituants d'une classe (champs, méthodes, etc.) par des objets permettant d'examiner :
 - les noms, les types et les modificateurs ;
 - pour les méthodes, les paramètres et leurs types.
- C'est-à-dire que Java offre leur *réification* sous forme d'objets.

Réifier = décrire + accéder

- Pour réifier des entités, il faut d'abord les décrire, ou en donner une représentation (objet) manipulable.
- En programmation par objets, tout est objet, donc la représentation manipulable est un objet, avec ses variables, ses méthodes, qu'il faut décrire.
⇒ *c'est le rôle des classes !*
- Cela pose donc la question de l'accès à la description des classes elles-mêmes. Il faut aussi pouvoir les manipuler dynamiquement. *Comment faire ?*
 - ⇒ Représenter les classes comme des objets !
 - ⇒ OK, mais alors, quelle sera la classe qui va décrire ces objets qui sont en réalité des classes ?
- **Métaclass** : classe dont les instances (objets) sont des classes.
 - Concept introduit en Smalltalk (1976), puis étudié par Cointe et Briot (1984-87) (dont le problème de clôture de la régression potentiellement infinie de métaclasses).

La métaclass `java.lang.Class<T>`

- Classe instantiant tous les objets représentant des classes.
 - La variable de type `T` représente le type des objets instances de cette classe, donc la classe elle-même.
Exemple : L'instance de `Class<Point>` est la métaclass de la classe `Point`, ses instances étant de type `Point`.
 - C'est une classe *finale*, donc non-héritable : tous les objets représentant des classes étant instance de l'unique `java.lang.Class<T>`, il en découle
 - ⇒ pas de nouvelles formes de classes.
 - ⇒ c'est-à-dire, pas de modification de la représentation ou du comportement des classes.
- Exemple : classes singleton qui connaissent leur unique instance.*
- C'est pour cela qu'on parle plutôt de *pseudo-métaclass*¹.
 - Suivant l'idée de Cointe et Briot, elle est sa propre description, ce qui clos la régression potentiellement infinie.

¹ C'est-à-dire, pas une métaclass de plein droit car non-remplaçable et non-extensible.

Trois façons de récupérer un objet classe

- Par la méthode `getClass` définie par la classe `Object` :

```
Point p = new Point(0.0, 0.0) ;
Class<Point> pointClass = p.getClass() ;
```
- Par la méthode statique `forName` définie par la classe `Class` :

```
String suffix = "int" ;
try {
    Class<?> pointClass = Class.forName("Po" + suffix) ;
} catch (ClassNotFoundException e) {
    System.out.println("La classe Po" + suffix + " n'existe pas.") ;
    throw e ;
}
```
- Par la variable statique `class` associée à chaque classe :

```
Class<Point> pointClass2 = Point.class ;
```
- Exemple d'utilisation : récupérer les méthodes déclarées :

```
Point.class.getDeclaredMethods()
```

Principales méthodes de `java.lang.Class<T>`

```
<A extends Annotation> getAnnotation(Class<A> annotationClass)
Class[] getClasses()
ClassLoader getClassLoader()
Class getComponentType()
Constructor<T> getConstructor(Class... parameterTypes)
Constructor[] getConstructors()
Class[] getDeclaredClasses()
Constructor getDeclaredConstructor(Class... parameterTypes)
Constructor[] getDeclaredConstructors()
Field getDeclaredField(String name)
Field[] getDeclaredFields()
Method getDeclaredMethod(String name, Class... parameterTypes)
Method[] getDeclaredMethods()
Class<?> getDeclaringClass()
Field getField(String name)
Field[] getFields()
```

Principales méthodes de `java.lang.Class<T>` II

```
Class[] getInterfaces()
Method getMethod(String name, Class... parameterTypes)
Method[] getMethods()
int getModifiers()
String getName()
Package getPackage()
ProtectionDomain getProtectionDomain()
Class<? super T> getSuperclass()
boolean isArray()
boolean isAssignableFrom(Class<?> cls)
boolean isInstance(Object obj)
boolean isInterface()
boolean isPrimitive()
T newInstance()
```

Le « package » `java.lang.reflect`

```
java.lang.Object
java.lang.reflect.AccessibleObject (implements java.lang.reflect.AnnotatedElement)
java.lang.reflect.Constructor<T> (implements java.lang.reflect.GenericDeclaration,
    java.lang.reflect.Member)
java.lang.reflect.Field (implements java.lang.reflect.Member)
java.lang.reflect.Method (implements java.lang.reflect.GenericDeclaration,
    java.lang.reflect.Member)
java.lang.reflect.Array
java.lang.reflect.Modifier
java.security.Permission (implements java.security.Guard, java.io.Serializable)
java.security.BasicPermission (implements java.io.Serializable)
java.lang.reflect.ReflectPermission
java.lang.reflect.Proxy (implements java.io.Serializable)
java.lang.Throwable (implements java.io.Serializable)
java.lang.Error
    java.lang.LinkageError
        java.lang.ClassFormatError
            java.lang.reflect.GenericSignatureFormatError
java.lang.Exception
    java.lang.reflect.InvocationTargetException
    java.lang.RuntimeException
        java.lang.reflect.MalformedParameterizedTypeException
    java.lang.reflect.UndeclaredThrowableException
```

La classe `java.lang.reflect.Field` I

- Classe finale dont les instances représentent les champs.
- Champs : variables, constantes.
- Peuvent être de classe (statiques) ou d'instance.
- Principales méthodes :

```
Object get(Object obj)
"Type" get"Type"(Object obj) avec "Type" = Boolean, Byte, ...
<T extends Annotation> getAnnotation(Class<T> annotationClass)
Class<?> getDeclaringClass()
int getModifiers()
Type getGenericType()
String getName()
Class<?> getType()
```

La classe `java.lang.reflect.Field` II

```
boolean isEnumConstant()
void set(Object obj, Object value)
void set"Type"(Object obj, "Type" z) avec "Type" = Boolean, Byte, ...
String toGenericString()
```


Plan

- 1 Réflexion et adaptabilité logicielle
- 2 Réflexivité structurelle en Java
- 3 Exemple : connecteur de composants par proxy
- 4 Réflexivité comportementale en Java/Javassist
- 5 Exemple : connecteur généré avec Javassist
- 6 Réflexivité en BCM

Mécanisme de *proxy* dynamique de Java

- Mécanisme général d'interception des appels : introduire un objet entre un client et un fournisseur qui va intercepter « tous » les appels du premier au second.
Exemple d'utilisation : talons (*stub*) et squelettes RMI.
- En Java, il s'agit :
 - 1 de créer un intercepteur (*proxy*) à partir des interfaces implantées par le fournisseur ;
 - 2 de lier à cet intercepteur un objet dit « *invocation handler* » qui va effectivement traiter tous les appels reçus par l'intercepteur ;
 - 3 pour ce faire, cet « *invocation handler* » implante une interface prédéfinie contenant une méthode *invoke*.
- La classe `java.lang.Proxy`, superclasse de toutes les classes de *proxy* créées dynamiquement, propose des méthodes statiques pour créer des classes de *proxy* ou directement des objets *proxy* (dont la classe est créée implicitement).

Énoncé du problème

- Dans le modèle à composants BCM, les composants exposent et requièrent des services par leurs ports entrants et sortants reliés par des connecteurs implantant l'interface *requis* et appelant le port entrant sur l'interface *offerte*.
⇒ C'est une forme de *proxy* !
- La solution la plus évidente et la plus flexible consiste à programmer chaque connecteur manuellement.
- Lorsqu'il s'agit simplement de relayer les appels, une solution plus satisfaisante serait de générer les connecteurs automatiquement à partir des interfaces *requis* et *offerte*.
- Dans certains cas, il est possible de le faire en utilisant les *proxys* dynamiques de Java.
- Illustrons cela sur un exemple d'un composant *Calculator* offrant une interface *CalculatorServicesI* et appelé par un composant *VectorSummer* via une interface *requis* *SummingServiceI*.

Interface offerte et requise

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

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

Le service requis *sum* est réalisée par le service offert *add*, ce qui donne le connecteur « *manuel* » suivant :

```
public class ManualConnector
extends AbstractConnector
implements SummingServiceI
{
    @Override
    public double sum(double x, double y) throws Exception
    {
        return ((CalculatorServicesI)this.offering).add(x, y) ;
    }
}
```

Comment procéder ?

- 1 Le *proxy* connecteur est créé sur les interfaces `ConnectorI` et `SummingServiceI`.

```
ConnectorI proxy =
    (ConnectorI) Proxy.newProxyInstance(
        this.getClass().getClassLoader(),
        new Class<?>[]{ConnectorI.class, SummingServiceI.class},
        new ConnectorIH(methodNamesMap) );
```

- 2 Pour traiter les appels via `SummingServiceI`, l'*invocation handler* doit savoir comment faire correspondre ses méthodes et celles de `CalculatorServiceI`.
 - Par simplicité, faisons cette correspondance uniquement sur les noms (nombre, types et ordre des paramètres étant les mêmes).
- 3 Puisqu'il doit implanter les méthodes de l'interface *ConnectorI*, l'*invocation handler* hérite d'`AbstractConnector` et exécute ces méthodes sur lui-même (`this`).

L'invocation handler I

```
public class ConnectorIH extends AbstractConnector implements InvocationHandler
{
    protected HashMap<String, String> methodNamesMap ;

    public ConnectorIH(HashMap<String, String> methodNamesMap) {
        this.methodNamesMap = methodNamesMap ;
    }

    protected boolean isConnectorMethod(String methodName) {
        Method[] connectorMethods = ConnectorI.class.getMethods() ;
        boolean ret = false ;
        for(int i = 0 ; !ret && i < connectorMethods.length ; i++ ) {
            ret = connectorMethods[i].getName().equals(methodName) ;
        }
        return ret ;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable
    {
        if (this.isConnectorMethod(method.getName())) {
            // Invoke the method on the invocation handler as connector object.
            return method.invoke(this, args) ;
        }
    }
}
```

L'invocation handler II

```
} else {
    // Get the name of the offered method for the required one.
    String offeredMethodName = this.methodNamesMap.get(method.getName()) ;

    // When no correspondance is given, keep the same name.
    if (offeredMethodName == null) { offeredMethodName = method.getName() ; }

    // Find the method implementation in the inbound port.
    // First, compute the types of the arguments.
    Class<?>[] pt = null ;
    if (args != null) {
        pt = new Class<?>[args.length] ;
        for (int i = 0 ; i < args.length ; i++) {
            pt[i] = args[i].getClass() ;
        }
    }
    Method offeredMethod =
        this.offering.getClass().getMethod(offeredMethodName, pt) ;

    // Invoke the found method on the inbound port.
    return offeredMethod.invoke(this.offering, args) ;
}
```

Création du proxy et séquence de connexion

Exemple de séquence d'exécution :

```
HashMap<String, String> methodNamesMap = new HashMap<String, String>() ;
methodNamesMap.put("sum", "add") ;
ConnectorI connector =
    (ConnectorI) Proxy.newProxyInstance(
        this.getClass().getClassLoader(),
        new Class<?>[]{ConnectorI.class, SummingServiceI.class},
        new ConnectorIH(methodNamesMap)) ;
p.doConnection(ServerPortURI, connector) ;
```

Plan

- 1 Réflexion et adaptabilité logicielle
- 2 Réflexivité structurelle en Java
- 3 Exemple : connecteur de composants par proxy
- 4 Réflexivité comportementale en Java/Javassist
- 5 Exemple : connecteur généré avec Javassist
- 6 Réflexivité en BCM

De la réflexion de structure à la réflexion de comportement

- La limite de la réflexion en Java apparaît au niveau de la méthode `invoke` de la classe `Method` qui permet d'exécuter un objet méthode, mais comme une boîte noire dont le code n'est pas réifié (donc inaccessible).
- Comment donner accès au code ?
 - La machine virtuelle Java accède au code par le contenu des fichiers `.class` chargé dans sa mémoire par des chargeurs de classes (« *class loaders* »).
 - Plusieurs outils ont été conçus et implantés pour donner un accès au code des classes via son fichier `.class`.
 - Ces outils s'interposent entre la machine virtuelle et les fichiers `.class` et permettent aux programmes de modifier le code des classes lors de leur chargement.
 - BCEL et ASM réifient le code octal de la JVM alors que Javassist permet de le voir sous la forme de code source Java.

Principes généraux de Javassist

- Javassist réifie le contenu des classes sous forme d'objets d'une bibliothèque ressemblant autant que possible à l'API Reflection de Java.

```

Class<?>  =>  CtClass
Method    =>  CtMethod
etc.

```

- Une classe est représentée par une unique instance de `CtClass` (pour « *compile-time class* ») en la chargeant dans le *pool* de classe de Javassist grâce à son chargeur de classe (*class loader*).
- Les manipulations sont possibles tant que cette « ct-classe » n'est pas mise à disposition de la machine virtuelle en appelant explicitement la méthode `toClass()` sur cette ct-classe qui est alors chargée dans la JVM puis gelée.
- Alternativement, Javassist permet d'intervenir au chargement des classes par Java en interposant un filtre agissant selon un patron de conception *Observateur* ; le chargement dans le *pool* Javassist demeure manuel mais le chargement dans la JVM après exécution du filtre est alors fait automatiquement.
- Normalement, une classe ne peut être chargée deux fois dans une JVM, mais l'API JPDA permet de modifier une classe déjà chargée par la technique du « *hot swap* » utilisée par les *debuggers*.

Exemple de manipulation manuelle en Javassist

```
import javassist.*;
import test.Rectangle;

public class Main {
    public static void main(String[] args) throws Throwable {
        ClassPool pool = ClassPool.getDefault();
        Loader cl = new Loader(pool);           // chargeur Javassist
        CtClass ct = pool.get("test.Rectangle"); // chargement manuel
        ct.setSuperclass(pool.get("test.Point")); // modification via Javassist
        Class c = ct.toClass();                 // chargement dans la JVM
        Object rect = c.newInstance();           // utilisation en Java
        ...
    }
}
```

Note : on ne doit pas typer la variable `rect` par `Rectangle`, sinon Java chargerait la classe `Rectangle` avant d'appeler la méthode `main` et l'appel à `toClass` lèverait une exception puisque la classe aurait déjà été chargée ; on devrait alors faire du *hot swap* pour la charger, ce qui est plus complexe.

Interception au chargement I

- L'intérêt du chargeur Javassist est de fournir un mécanisme d'interception basé sur un patron Observateur pour s'interposer entre la lecture des fichiers `.class` et la mise en place dans la machine virtuelle de manière à réaliser des transformations au chargement.
- Un transformateur est une instance d'une classe implantant l'interface `Translator`.

```
public interface Translator {
    public void start(ClassPool pool)
        throws NotFoundException, CannotCompileException;
    public void onLoad(ClassPool pool, String classname)
        throws NotFoundException, CannotCompileException;
}
```

Interception au chargement II

- La méthode `onLoad` est appelée par le chargeur de classe Javassist avec le collecteur et le nom de la classe *avant* de lire le fichier `.class`.
- Il faut donc définir :
 - une classe de transformation qui réalise les modifications désirées à la classe dans sa méthode `onLoad`, puis
 - attacher une instance de cette classe au chargeur de classes de Javassist.

À partir de là, cette méthode `onLoad` sera appelée à chaque fois qu'une classe doit être chargée.

Une transformation simple : rendre publique

```
public class MakePublicTranslator implements Translator {
    void start(ClassPool pool) throws ... {}
    void onLoad(ClassPool pool, String classname)
        throws NotFoundException, CannotCompileException {
        CtClass cc = pool.get(classname);
        cc.setModifiers(Modifier.PUBLIC);
    }
}

public class Main {
    public static void main(String[] args) throws Throwable {
        Translator t = new MakePublicTranslator();
        ClassPool pool = ClassPool.getDefault();
        Loader cl = new Loader();
        cl.addTranslator(pool, t);
        cl.run("MyApp", args);
    }
}
```

Notez ici l'utilisation de deux chargeurs (le chargeur initial implicite et celui donné par `cl`). L'existence de ces deux chargeurs peut faire qu'une même classe soit chargée dans les deux et alors considérées comme différentes. Dans ce cas, on peut devoir forcer le chargement de classes à se faire ensuite dans un chargeur spécifique pour préserver l'unicité de la classe chargée.

Les méthodes

- Les méthodes sont représentées par des instances de `CtMethod`.
 - Les constructeurs sont représentés par des instances de `CtConstructor`.
 - Ces classes définissent des méthodes :
 - `insertBefore()` and `insertAfter()` pour ajouter du code source dans le corps des méthodes,
 - `addCatch()` pour ajouter des traitements d'exception sur l'ensemble du corps de la méthode, et
 - La méthode `insertAt()` permet d'introduire du code à une ligne donnée de la méthode.
- Condition : la méthode doit avoir été compilée avec l'option -g (« *debugging* » ou déverminage) laissant plus d'information dans le fichier `.class` sur les numéros de lignes et les noms de variables locales.

Énoncé du problème

- 1 Reprenons l'exemple précédent : comment éviter de programmer manuellement le connecteur entre deux ports de composants ?
- 2 Ici, la solution sera conceptuellement plus simple, puisqu'il s'agit de créer dynamiquement la classe définissant le connecteur, mais avec le *même code* que la classe créée manuellement.
- 3 Pour créer une classe de connecteur, il faut avoir :
 - le nom de la classe à créer,
 - la superclasse de connecteur à utiliser,
 - l'interface requise devant être implantée par le connecteur,
 - l'interface offerte et implantée par le port entrant, et
 - la correspondance entre les méthodes de l'interface requise et celles de l'interface offerte.
- 4 Les classes et les interfaces sont représentées par des instances de la classe `java.lang.Class`.

Génération de la classe de connecteur I

```
public Class<?> makeConnectorClassJavassist(String connectorCanonicalClassName,
                                           Class<?> connectorSuperclass,
                                           Class<?> connectorImplementedInterface,
                                           Class<?> offeredInterface,
                                           HashMap<String,String> methodNamesMap
                                           ) throws Exception
{
    ClassPool pool = ClassPool.getDefault();
    CtClass cs = pool.get(connectorSuperclass.getCanonicalName());
    CtClass cii = pool.get(connectorImplementedInterface.getCanonicalName());
    CtClass oi = pool.get(offeredInterface.getCanonicalName());
    CtClass connectorCtClass = pool.makeClass(connectorCanonicalClassName);
    connectorCtClass.setSuperclass(cs);
    Method[] methodsToImplement = connectorImplementedInterface.getDeclaredMethods();
    for (int i = 0; i < methodsToImplement.length; i++) {
        String source = "public ";
        source += methodsToImplement[i].getReturnType().getName() + " ";
        source += methodsToImplement[i].getName() + "(";
        Class<?>[] pt = methodsToImplement[i].getParameterTypes();
        String callParam = "";
        for (int j = 0; j < pt.length; j++) {
            String pName = "aaa" + j;
            source += pt[j].getCanonicalName() + " " + pName;
            callParam += pName;
            if (j < pt.length - 1) {
                source += ", ";
                callParam += ", ";
            }
        }
        source += ")";
        Class<?>[] et = methodsToImplement[i].getExceptionTypes();
        if (et != null && et.length > 0) {
            source += " throws ";
        }
    }
}
```

Génération de la classe de connecteur II

```
for (int z = 0; z < et.length; z++) {
    source += et[z].getCanonicalName();
    if (z < et.length - 1) {
        source += ", ";
    }
}
source += "\n    return (";
source += offeredInterface.getCanonicalName() + ")this.offering);";
source += methodNamesMap.get(methodsToImplement[i].getName());
source += "(" + callParam + ");\n";
CtMethod theCtMethod = CtMethod.make(source, connectorCtClass);
connectorCtClass.addMethod(theCtMethod);
}
connectorCtClass.setInterfaces(new CtClass[]{cii});
cii.detach(); cs.detach(); oi.detach();
Class<?> ret = connectorCtClass.toClass();
connectorCtClass.detach();
return ret;
}
```

Ce qui génère le code suivant pour la méthode `sum` :

```
public double sum(double aaa0, double aaal) throws java.lang.Exception
{
    return ((fr.upmc.alasca.summing.calculator.interfaces.CalculatorServicesI)
            this.offering).add(aaa0, aaal);
}
```

Séquence de création de la classe et de connexion

```
HashMap<String, String> methodNamesMap = new HashMap<String, String>();
methodNamesMap.put("sum", "add");
Class<?> connectorClass =
    this.makeConnectorClassJavassist(
        "fr.upmc.alasca.summing.assembly.GeneratedConnector",
        AbstractConnector.class,
        SummingServiceI.class,
        CalculatorServicesI.class,
        methodNamesMap);
p.doConnection(ServerPortURI,
               connectorClass.getCanonicalName());
```

- 1 Réflexion et adaptabilité logicielle
- 2 Réflexivité structurelle en Java
- 3 Exemple : connecteur de composants par proxy
- 4 Réflexivité comportementale en Java/Javassist
- 5 Exemple : connecteur généré avec Javassist
- 6 **Réflexivité en BCM**

- La réflexion en BCM est encore en développement.
- Deux interfaces principales peuvent être offertes et requises :
 - `IntrospectionI` : regroupe toutes les méthodes permettant d'accéder à des informations sur le composant.
 - `IntercessionI` : regroupe toutes les méthodes permettant de modifier des informations sur le composant ou plus généralement de modifier l'état du composant.
- Une interface `ReflectionI` regroupe les deux en héritant d'`IntrospectionI` et d'`IntercessionI`, permettant aux composant d'offrir les deux en même temps.
- BCM définit les ports entrants et sortants ainsi que les connecteurs correspondants.

- Objectif : donner la capacité d'inspecter et d'intro-agir sur les composants à l'exécution.
- Qu'est-ce qu'on vise à gérer ?
 - des informations définissant le « type » de composants (interfaces offertes et requises,
 - des informations statiques définissant les capacités du composant (passif ou actif, capable d'ordonnancer des tâches ou non, ...);
 - des informations plus dynamiques sur l'état courant du composant dans son cycle de vie (initialisé, démarré, arrêté, ...);
 - des informations de nature plus architecturales (les ports, leurs états, leurs connexions, ...).
 - des informations de nature plus comportementale et fonctionnelle, comme les implantations des services (signatures des méthodes d'implantation des serces et des constructeurs, les greffons installés, les fonctions de trace et de journalisation, ...).

```
public interface IntrospectionI extends OfferedI, RequiredI
{
    // Plug-ins facilities
    public boolean hasInstalledPlugins() throws Exception ;
    public boolean isInstalled(String pluginId) throws Exception ;
    public PluginI getPlugin(String pluginURI) throws Exception ;
    public boolean isInitialised(String pluginURI) throws Exception ;

    // Logging facilities
    public boolean isLogging() throws Exception ;
    public boolean isTracing() throws Exception ;

    // Internal behaviour requests
    public boolean isInStateAmong(ComponentStateI[] states) throws Exception ;
    public boolean notInStateAmong(ComponentStateI[] states) throws Exception ;
    public boolean hasItsOwnThreads() throws Exception ;
    public int getTotalNumberOfThreads() throws Exception ;
    public boolean hasSerialisedExecution() throws Exception ;
    public boolean canScheduleTasks() throws Exception ;

    // Implemented interfaces management
    public Class<?>[] getInterfaces() throws Exception ;
    public Class<?> getInterface(Class<?> inter) throws Exception ;
    public Class<?>[] getRequiredInterfaces() throws Exception ;
    public Class<?> getRequiredInterface(Class<?> inter) throws Exception ;
    public Class<?>[] getOfferedInterfaces() throws Exception ;
    public Class<?> getOfferedInterface(Class<?> inter) throws Exception ;
    public boolean isInterface(Class<?> inter) throws Exception ;
    public boolean isRequiredInterface(Class<?> inter) throws Exception ;
    public boolean isOfferedInterface(Class<?> inter) throws Exception ;

    // Port management
    public String[] findPortURIsFromInterface(Class<?> inter) throws Exception ;
```

L'interface IntrospectionI II

```
public String[] findInboundPortURIsFromInterface(Class<?> inter) throws Exception ;
public String[] findOutboundPortURIsFromInterface(Class<?> inter) throws Exception ;
public boolean isPortExisting(String portURI) throws Exception ;
public Class<?> getPortImplementedInterface(String portURI) throws Exception ;
public boolean isPortConnected(String portURI) throws Exception ;

// Reflection facility
public String getComponentDefinitionClassName() throws Exception ;
public Annotation[] getComponentAnnotations() throws Exception ;
public ClassLoader getComponentLoader() throws Exception ;
public ServiceSignature[] getComponentServiceSignatures() throws Exception ;
public ConstructorSignature[] getComponentConstructorSignatures() throws Exception ;
}
```

Intégration des capacités réflexives dans les composants

- ReflectionI est ajoutée automatiquement aux interfaces offertes de tous les composants et un port entrant créé et publié.
- Le constructeur (int, int) génère une URI pour ce port, mais celui (String, int, int) permet de fournir une URI prédéfinie.
 - Son URI peut donc jouer le rôle d'URI du composant.
- Un appel sur l'interface IntercessionI pose un problème épineux : il modifie l'état du composant d'une manière qui peut engendrer des erreurs sur les autres appels à ce composant.
 - le composant doit d'abord suspendre son fonctionnement normal, exécuter l'appel d'intro-action puis reprendre son exécution ;
 - les appels reçus pendant ce temps par le composant sont mis en attente et l'appelant est bloqué si l'appel est synchrone.

La suspension peut avoir des conséquences sur les appels en cours, donc s'il s'avère impossible de suspendre, l'appel d'intro-action échoue et lève une exception.

Fonctionnalité, expérimentée par un PSTL, en cours d'intégration et de test.

L'interface IntercessionI

```
public interface IntercessionI extends OfferedI, RequiredI
{
    // Plug-ins facilities
    public void installPlugin(PluginI plugin) throws Exception ;
    public void finalisePlugin(String pluginURI) throws Exception ;
    public void uninstallPlugin(String pluginId) throws Exception ;
    public void initialisePlugin(String pluginURI) throws Exception ;

    // Logging facilities
    public void setLogger(Logger logger) throws Exception ;
    public void toggleLogging() throws Exception ;
    public void logMessage(String message) throws Exception ;
    public void printExecutionLog() throws Exception ;
    public void printExecutionLogOnFile(String fileName) throws Exception ;
    public void setTracer(TracerOnConsole tracer) throws Exception ;
    public void toggleTracing() throws Exception ;
    public void traceMessage(String message) throws Exception ;

    // Implemented interfaces management
    public void addRequiredInterface(Class<?> inter) throws Exception ;
    public void removeRequiredInterface(Class<?> inter) throws Exception ;
    public void addOfferedInterface(Class<?> inter) throws Exception ;
    public void removeOfferedInterface(Class<?> inter) throws Exception ;

    // Port management
    public void doPortConnection(String portURI, String otherPortURI, String ccname) throws Exception ;
    public void doPortDisconnection(String portURI) throws Exception ;

    // Reflection facility
    public ComponentI newInstance(Object[] parameters) throws Exception ;
    public Object invokeService(String name, Object[] params) throws Exception ;
    public Object invokeServiceSync(String name, Object[] params) throws Exception ;
    public Object invokeServiceAsync(String name, Object[] params) throws Exception ;
}
```

Court exemple d'utilisation I

```
public static class Client extends AbstractComponent
{
    protected ReflectionOutboundPort rObp ;
    public Client() throws Exception {
        super(1, 0) ;
        this.addRequiredInterface(ReflectionI.class) ;
        this.rObp = new ReflectionOutboundPort(this) ;
        this.addPort(rObp) ;
        rObp.publishPort() ;
    }

    @Override
    public void start() throws ComponentStartException
    {
        super.start() ;
        try {
            String[] uris = this.rObp.findInboundPortURIsFromInterface(ReflectionI.class) ;
            System.out.println(uris[0]) ;
            uris = this.rObp.findInboundPortURIsFromInterface(ComponentPluginI.class) ;
            System.out.println(uris[0]) ;
        } catch (Exception e) { throw new RuntimeException(e) ; }
    }
}
```

Court exemple d'utilisation II

```
@Override
public void deploy() throws Exception
{
    AbstractComponent server = new AbstractComponent(1, 0) {};
    this.addDeployedComponent(server);
    String[] rpIbpURI =
        server.findInboundPortURIsFromInterface(ReflectionI.class);
    Client client = new Client();
    this.addDeployedComponent(client);
    String[] rpObpURI =
        client.findOutboundPortURIsFromInterface(ReflectionI.class);
    client.doPortConnection(rpObpURI[0], rpIbpURI[0],
        ReflectionConnector.class.getCanonicalName());
    super.deploy();
}
```

Ce qui donne le résultat suivant :

```
starting...
fr.upmc.components.pre.reflection.interfaces.ReflectionI-4e9a2511-2a24-49fb-af39-7e
fr.upmc.components.pre.plugins.interfaces.ComponentPluginI-7e8694a5-2eaf-417a-b260-
shutting down...
ending...
```

Récapitulons...

- 1 L'auto-adaptabilité logicielle peut se décomposer en quatre grands types : **paramétrique, de ressources, fonctionnelle et architecturale**.
- 2 La **réflexivité logicielle** est une approche permettant de mettre en œuvre des opérations d'adaptation fonctionnelle et architecturale des entités logicielles.
- 3 **Java** est un langage offrant une forme de **réflexion structurelle**, en particulier par son *package* `java.lang.reflect` mais également une forme limitée de **réflexion de comportement** par son mécanisme de *proxy* dynamique.
- 4 Pour obtenir une forme de **réflexion de comportement plus complète** en Java, il faut faire appel à des **bibliothèques externes**, comme **Javassist**, permettant de créer du code dynamiquement et de l'intégrer à l'application.
- 5 Les principes de la réflexion s'appliquent également à la programmation par composants, où les réflexions structurelle et de comportement sont complétées par de la **réflexion architecturale**.

Pour aller plus loin : sélection de lectures recommandées

- Lire l'article « *Reflection in logic, functional and object-oriented programming : a Short Comparative Study* » disponible sur le site de l'UE.
- Examiner la Javadoc l'API Reflection de Java (dans la documentation standard de J2SE).
- Lire les tutoriaux suivants sur le WWW :
<http://tutorials.jenkov.com/java-reflection/index.html>
<http://onjava.com/pub/a/onjava/2007/03/15/reflections-on-java-reflection.html>
- Lire le tutoriel Javassist disponible avec la distribution du logiciel et la page Javassist du projet JBoss à l'URL <http://www.jboss.org/javassist> et les articles qui sont pointés par cette dernière.
- Regardez la fonctionnalité réflexion en BCM.
- Lire la description du modèle de composants Fractal dans *The Fractal Component Model*, E. Bruneton, T. Coupaye et J.-B. Stefani, version 2.0-3, 2004, disponible sur le site de l'UE.