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* en informatique, et en particulier dans les langages de programmation, 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 *introspecter* le contenu d'un programme à l'exécution et permettre de réaliser des opérations d'adaptation dynamique.
- Introduire la notion de *réflexivité comportementale* et sa déclinaison dans l'« écosystème » Java.
- Comprendre et apprendre à utiliser les *outils* de réflexivité comportementale de l'« écosystème » Java pour réaliser des *adaptations dynamiques*.
- Comprendre et apprendre à utiliser la *réflexivité architecturale* dans les modèles à composants comme BCM pour réaliser des opérations d'adaptation dynamique architecturales.

Plan

Réflexion et programmation

- Concept lié à la modification dynamique des programmes, une pratique aussi ancienne que l'informatique elle-même.
- Introduite autour de Lisp depuis les années '60, puis conceptualisée par Brian Smith (thèse, articles) au début des années 1980, et enfin développée par une large communauté depuis.
- Exemple : réflexion en Java
  - Java définit sous forme d'objets une grande partie des éléments constitutifs d'un programme comme les classes, les méthodes, etc., (`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 *i.e.*, les classes (ex.: BCEM, Javassist).
- Plusieurs autres langages de programmation, antérieurs ou postérieurs à Java, offrent des possibilités similaires (Smalltalk, Prolog, C#, Python, etc.).

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 l'achitecture de l'application.
- La réflexion dans une architecture à composants permet aussi :
  - de modifier cette architecture dynamiquement, en créant, ajoutant et retranchant des interfaces offertes et requises,
  - en connectant, déconnectant, reconnectant les ports,
  - en ajoutant, retranchant et remplaçant des composants,

  et ainsi l'adapter dynamiquement à son état d'exécution courant.
- En BCM, la réflexion de Java s'ajoute à celle du modèle à composants pour permettre d'examiner ou de modifier dynamiquement les composants existants et en ajouter de nouveaux.

Plan

Objectifs

- La réflexion structurelle s'intéresse à l'*examen* du contenu d'un programme par son propre code pendant l'exécution.
  On parle alors d'*introspection* du programme sur lui-même.
- Java possède une API pour cela fournie par les classes `Object` et `Class<T>` et 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, ...), etc.
- 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

- Réifier des entités consiste à en donner une représentation manipulable informatiquement qui en décrit ce qui les constitue.
- En programmation par objets, tout est objet, donc la représentation manipulable est un objet, avec ses variables, ses méthodes.
  Il faut décrire cet objet $\Rightarrow$ *c'est le rôle des classes !*
  Ex.: objet réprésentant une méthode et sa classe `Method`.
- Comme vu précédemment, la description des classes est aussi concernée car il faut pouvoir les manipuler dynamiquement pour être totalement réflexif. *Quelles conséquences ?*
  - $\Rightarrow$ Les classes sont représentées par des objets !
  - $\Rightarrow$ OK, mais alors, de quelle classe sont instances ces objets qui sont en réalité des classes ?
- *Métaclasse* : classe dont les instances 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 métaclasse `java.lang.Class<T>`

- Classe instantiant tous les objets représentant des classes.
  - `T` représente le type des instances de la classe représentée, donc la classe elle-même *i.e.*,
    `T newInstance() // méthode de Class<T>`
    Exemple : L'instance de `Class<Point>` est la métaclasse de la classe `Point`, les instances de celle-ci étant de type `Point`.
- `Class<T>` est *finale* (non-héritable) : tous les objets représentant des classes sont instances de cette unique métaclasse, d'où :
  - ⇒ pas de nouvelles formes de classes.
  - ⇒ c'est-à-dire, pas de modification de la réprésentation ou du comportement des classes.

  Exemple : classes singleton qui n'ont qu'une unique instance.
- Pour Java, on parle donc plutôt de *pseudo-métaclasse*[1].
- Suivant l'idée de Cointe et Briot, elle se décrit elle-même, ce qui clôt la régression potentiellement infinie : l'objet représentant la classe `Class<T>` est instance de `Class<Class>`.

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

Trois façons de récupérer un objet classe en Java

- 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 :

```
Method[] declared = Point.class.getDeclaredMethods();
```

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

```
<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`

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

```
Object get(Object obj)
<T extends Annotation> getAnnotation(Class<T> annotationClass)
Class<?> getDeclaringClass()
int getModifiers()
Type getGenericType()
String getName()
Class<?> getType()
boolean isEnumConstant()
void set(Object obj, Object value)
```

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

- Classe finale dont les instances représentent les méthodes.
- Ne permet pas de manipuler le code des méthodes, mais plutôt d'en découvrir la signature et les informations associées.
- Elle offre une méthode `invoke` pour appeler la méthode représentée sur un objet avec des paramètres réels donnés.

```
Method m = Point.class.getMethod("toString", new Class<?>[]);
System.out.println(m.invoke(new Point(1, 2), new Object[]));
```

⇒ *on touche ici à la frontière entre réflexion de structure et de comportement ! (introspection et intro-action)*

⇒ *c'est-à-dire, la limite entre ce qui est visible et possible de faire à propos des méthodes en Java et ce qui ne l'est pas...*

- En fait, cela montre la puissance, mais aussi la limite de l'API réflexion de Java : le code des méthodes peut être invoqué réflexivement, mais dont il ne peut pas être examiné ni modifié.

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

- Principales méthodes :

```
<T extends Annotation> getAnnotation(Class<T> annotationClass)
Class<?> getDeclaringClass()
Class<?>[] getExceptionTypes()
Type[] getGenericExceptionTypes()
Type[] getGenericParameterTypes()
Type getGenericReturnType()
int getModifiers()
String getName()
Annotation[][] getParameterAnnotations()
Class<?>[] getParameterTypes()
Class<?> getReturnType()
Object invoke(Object obj, Object... args)
boolean isVarArgs()
String toGenericString()
```

Forme de réflexion offerte par Java

En résumé :

- De la réflexion structurelle limitée à l'introspection.
- Une des limitations importantes de l'API Reflection est de ne pas fournir un accès au *code des méthodes*, ce qui pourtant est une partie intégrante de la définition de réflexion structurelle.

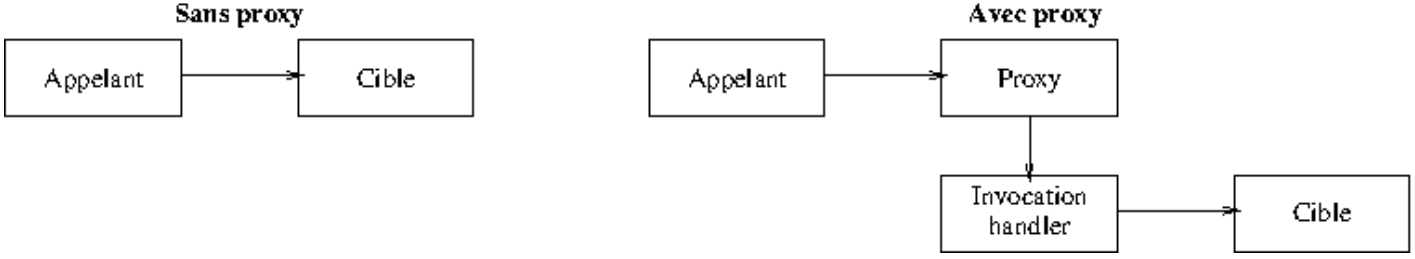
Pour pallier à ces limitations, Java offre des mécanismes *plus restreints*, comme l'interception des appels de méthodes par des *proxys*.

- Un *proxy* permet, pour un objet ciblé, de remplacer ou d'étendre le code des méthodes plutôt que de le remplacer.
- Il faut ensuite l'interposer *manuellement* entre l'appelant et l'objet ciblé pour bénéficier de sa finalité.
  Cette interposition manuelle, contraignante en termes de programmation, est l'un des principaux défauts de l'approche proxy.

Plan

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 :



  1. faire créer par Java un *proxy* à partir des interfaces implantées par la cible et l'imposer à l'appelant ;
  2. fournir à cet intercepteur un objet dit « *invocation handler* » qui va effectivement traiter tous les appels reçus par le *proxy* via une interface prédéfinie et une méthode `invoke` ;
  3. l'*invocation handler* peut être lié à la cible si nécessaire.
- 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).

Connecteurs *proxys* : 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 *requise* et appelant le port entrant sur l'interface *offerte*.
  $\Rightarrow$ 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 automatiquement* les connecteurs à partir des interfaces requise 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 `CalculatorServicesCI` et appelé par un composant `VectorSummer` via une interface requise `SummingServiceCI`.

Interface offerte et requise

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

public interface SummingServiceCI
extends RequiredCI
{
  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 SummingServiceCI
{
  @Override
  public double sum(double x, double y) throws Exception
  {
    return ((CalculatorServicesCI)this.offering).add(x, y);
  }
}
```

Comment procéder ?

1. Le *proxy* connecteur est créé sur les interfaces `ConnectorI` (interne à BCM4Java) et `SummingServiceCI`.

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

2. Pour traiter les appels via `SummingServiceCI`, l'*invocation handler* doit savoir comment faire correspondre ses méthodes et celles de `CalculatorServiceCI`.
   - 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++ ) {
      // limited equality test, should take parameters into account...
      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 =      // get the method object
          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 :

```
// creation of a mapping between required and offered methods
HashMap<String, String> methodNamesMap = new HashMap<String, String>();
methodNamesMap.put("sum", "add");
// creation of the connector object
ConnectorI connector =
    (ConnectorI) Proxy.newProxyInstance(
                    this.getClass().getClassLoader(),
                    new Class<?>[]{ConnectorI.class, SummingServiceCI.class},
                    new ConnectorIH(methodNamesMap));
// connecting using the doPortConnection method that takes a connector
// object as last parameter
this.getOwner.doPortConnection(clientPortURI, serverPortURI, connector);
```

Plan

De la réflexion de structure à la réflexion de comportement

- La limite de la réflexion en Java apparaît avec `Method#invoke` qui permet d'exécuter une méthode sans que son code soit réifié.
- La réflexion comportementale s'intéresse à l'accès et à la *modification dynamique* du code par le programme lui-même. On parle alors d'*intro-action* (« *intercession* »).
- 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* »).
  - Ainsi, plusieurs outils ont été conçus et implantés pour donner un accès au code des classes via les fichiers `.class`.
  - Ces outils s'interposent généralement entre la machine virtuelle et les fichiers `.class` et permettent ainsi 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 traiter sous la forme de code source Java.[1]

[1] **Attention** : pour des raisons pragmatiques, Javassist est resté à du source Java 1.4 !

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<T>` ⇒ `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 via la méthode `toClass()` dont l'appel provoque le chargement de la classe dans la JVM et le gel de la « ct-classe ».
- 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.*;
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` (qui procède au chargement) 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.

Remarque : dans le projet de gestionnaire d'énergie, c'est plutôt cette façon d'utiliser Javassist qui permettra de générer les classes de connecteurs, à ceci près qu'il n'y aura pas de classe de connecteur préexistante, elles seront entièrement générées dynamiquement.

Interception au chargement I

- Une seconde facette importante 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 leur mise en place dans la machine virtuelle.
- Ce mécanisme permet de réaliser des transformations au chargement par un objet *transformateur*.
- Un transformateur est une instance d'une classe implantant l'interface `Translator`.

```
public interface Translator {
    // appelée au début, pour initialiser
    public void start(ClassPool pool)
        throws NotFoundException, CannotCompileException;
    // appelée à chaque chargement de classe
    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.
- La méthode `onLoad` peut :
  - ne rien faire, auquel cas la classe sera chargée sans modification dans la JVM
  - ou alors elle peut charger la classe sous forme de « ct-classe » dans le *pool* Javassist, la modifier puis rendre le contrôle et alors c'est cette *ct-classe* qui sera chargée dans la JVM (par un appel interne à `toClass`).

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 la JVM à charger toutes les classes dans un chargeur spécifique désigné pour préserver l'unicité des classes chargées. Il est possible de désigner un chargeur de classes spécifique lors du lancement de la JVM.

Modification du code des méthodes en Javassist

- 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'exceptions sur l'ensemble du corps de la méthode, et
  - La méthode `insertAt()` permet d'introduire du code à une ligne donnée (source) 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.
- Javassist propose aussi un façon de modifier le code existant des méthodes via un parcours du code selon un patron observateur, mais nous ne le détaillerons pas ici.

Exemple

- Soit la classe `Point` suivante :

```
class Point {
  int x, y;
  void move(int dx, int dy) { x += dx; y += dy; }
}
```

- On ajoute une trace de `move` par :

```
ClassPool pool = ClassPool.getDefault();
CtClass cc = pool.get("Point");
CtMethod m = cc.getDeclaredMethod("move");
m.insertBefore("{ System.out.println(dx); System.out.println(dy); }");
```

- Pour donner l'équivalent de :

```
class Point {
    int x, y;
    void move(int dx, int dy) {
        { System.out.println(dx); System.out.println(dy); }
        x += dx; y += dy;
    }
}
```

Ajout de méthodes et de champs

Ajout d'une méthode :

```
CtClass point = ClassPool.getDefault().get("Point");
CtMethod m =
  CtNewMethod.make("public int xmove(int dx) { x += dx; }", point);
point.addMethod(m);
```

Ajout d'un champ :

- Préparation :

```
CtClass point = ClassPool.getDefault().get("Point");
CtField f = new CtField(CtClass.intType, "z", point);
```

- puis, ajout sans initialisation :

```
point.addField(f);
```

- ou encore avec initialisation :

```
point.addField(f, "0");     // la valeur initiale est 0.
```

Plan

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 génére le code suivant pour la méthode `sum` :

```
public double sum(double aaa0, double aaa1) throws java.lang.Exception
{
  return ((fr.upmc.alasca.summing.calculator.interfaces.CalculatorServicesI)
                                        this.offering).add(aaa0, aaa1) ;
}
```

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) ;
this.getOwner().doPortConnection(
                  clientPortURI,
                  serverPortURI,
                  connectorClass.getCanonicalName()) ;
```

Plan

Réflexivité dans un modèle à composants

- Objectif : donner la capacité d'introspecter et d'intro-agir sur les composants et leur assemblage à l'exécution.
- Qu'est-ce qu'on vise à gérer ?
  - des informations contribuant à définir le « type » de composants (interfaces offertes et requises, héritage entre classes de définition, etc.),
  - 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 services et des constructeurs, les greffons installés, les fonctions de trace et de journalisation, ...).

Les interfaces de réflexion en BCM

- La réflexion en BCM est encore en développement.
  - L'intégration avec la réflexion de Java est toujours embryonnaire.
- Deux interfaces principales peuvent être offertes et requises :
  - `IntrospectionCI` : regroupe toutes les méthodes permettant d'accéder à des informations sur le composant.
  - `IntercessionCI` : regroupe toutes les méthodes permettant de modifier des informations sur le composant ou plus généralement de modifier l'état du composant.
- Une interface `ReflectionCI` regroupe les deux en héritant d'`IntrospectionCI` et d'`IntercessionCI`, permettant aux composant d'offrir facilement les deux en même temps.
- BCM définit les ports entrants et sortants ainsi que les connecteurs correspondants sur cette interface `ReflectionCI`.

Intégration des capacités réflexives dans les composants

- `ReflectionCI` est ajoutée automatiquement aux interfaces offertes de tous les composants ; un port entrant est créé, publié puis son URI retournée lors de la création du composant.
- Un appel sur l'interface `IntercessionCI` 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, d'abord expérimentée par un PSTL, en cours d'évaluation et donc pas encore intégrée à la version courante de BCM.*

Court exemple : ajout de code dans une méthode I

```
public class ReflectionServer extends AbstractComponent {
  ...
  public void   myService(String message) {
    System.out.println("-----------------------------------------\n" + message);
    this.test();
    System.out.println("-----------------------------------------");
  }
  public void   test() { System.out.println("Code already there!"); }
}

public class ReflectionClient extends AbstractComponent {
  ...
  public void execute() throws Exception {
    // connexion des ports...
    this.servicePort.myService("Before change:");
    this.rObp.insertBeforeService("test", new String[]{},
                                  "System.out.println(\"Yes!\");");
    this.rObp.insertAfterService("test", new String[]{},
                                 "System.out.println(\"Again!\");");
    this.servicePort.myService("After change:");
  }
}
```

Court exemple : ajout de code dans une méthode II

Ce qui donne le résultat suivant côté composant serveur :

```
$ ./start-dcvm server
starting...
executing...
-----------------------------------------
Before change:
Code already there!
-----------------------------------------
-----------------------------------------
After change:
Yes!
Code already there!
Again!
-----------------------------------------
finalising...
shutting down...
ending...
```

Cet exemple est fourni avec la bibliothèque BCM. Attention, pour le lancer, il y a des paramètres spécifiques à ajouter à la ligne de commande (le « javaagent » spécifique à Javassist). Les scripts de lancement indiquent comment faire.

Récapitulons...

1. L'auto-adaptabilité logicielle peut se décomposer en quatres 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 le tutoriel suivant sur le WWW : `http://tutorials.jenkov.com/java-reflection/index.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.