Méthodes de Conception

(L3 Info – SINFL5A1)

Jour 5

- Présentation du sujet de devoir de CC
- Gestion de projet avec ANT
- Révisions sur MVC et Observer
- Révision sur Swing (interfaces graphiques en Java)

 yann.mathet@unicaen.fr

Gestion de projet avec ANT

- La mise en place d'applications s'accompagne :
  - d'un certain nombre d'opérations récurrentes (d'un projet à l'autre)
  - et répétitives (au sein d'un même projet)
- Exemples :
  - création d'un classpath (compilation, exécution)
  - compilation et détermination du chemin du code compilé
  - génération de la javadoc
  - Nettoyage de l'arborescence

- Ces opérations sont souvent liées :
  - on compile avec d'exécuter
  - on nettoie avant de compiler
- Il est donc fastidieux de mémoriser les chemins, l'enchaînement des tâches, etc.
- On peut donc automatiser avec un Makefile avec la commande make
- Ou utiliser l'équivalent plus souple « build.xml » avec « ant »

build.xml

- Il définit en XML un ensemble de « targets », utilisant différentes « tasks »
- Il définit des dépendances entre ces targets
- Ces targets peuvent être lancées avec la commande ant
- Exemple : « ant clean » va lancer la target « clean » du fichier build.xml

Aperçu général d'un build.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<project basedir="." default="compile"
    name="NomDuProjet">
      <!-- définition des targets -->
</project>
```

- Le répertoire de base du projet NomDuProjet est celui contenant le fichier build.xml
- Appelée sans argument, la commande ant invoquera la target par défaut : compile

Targets

- On définit un ensemble de targets
- Les noms sont arbitraires, mais certains sont conventionnels : compile, clean, dist, run
- On précise leurs relations de dépendance

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<project basedir="." default="compile" name="monApplication">
    <target name="compile"></target>
    <target name="dist" depends="compile"></target>
    <target name="clean"></target>
    <target name="run" depends="dist"></target>
</project>
```

Contenu d’une target : des tasks

- Exemple :

```
<target name="run" depends="dist">
        <java classname="Main" />
</target>
```

- ici, « java » est une task reconnue par ant
- ant run va ainsi lancer le main() sans argument

Définition de propriétés/valeurs

- Définition par :

```
<project basedir="." default="compile" name="monApplication">
    <property name="application.version" value="0.0.1"/>
</project>
```

- Utilisation dans le reste du build.xml via :

```
${application.version}
```

Définition d’un classpath

- Définition d’un path par :

```
<project basedir="." default="compile" name="monApplication">
    <path id="monAppli.classpath">
        <pathelement location="classes"/>
    </path>
 </project>
```

- Utilisation dans différentes tasks via :

```
<classpath refid="monAppli.classpath"/>
```

Principales tasks

- Compilation avec javac

```
<path id="monAppli.classpath">
    <pathelement location="classes"/>
</path>
<target name="compile">
    <javac srcdir="src" source="1.8" destdir="classes">
        <classpath refid="monAppli.classpath"/>
    </javac>
</target>
```

Principales tasks (2)

- Nettoyage avec delete

```
<target name="clean">
    <delete>
        <fileset dir="classes" includes="**/*.class"/>
        <fileset dir="src" includes="**/*~" defaultexcludes="no"/>
    </delete>
    <delete file="dist/application-${application.version}.jar"/>
</target>
```

Principales tasks (3)

- Création de jar avec jar :

```
<target name="dist" depends="compile">
    <delete file="dist/application-${application.version}.jar"/>
    <jar jarfile="dist/application-${application.version}.jar">
        <fileset dir="src" includes="**/*.gif,**/*.png,**/*.html"/>
        <fileset dir="classes" includes="**/*.class"/>
        <manifest>
            <attribute name="Main-Class" value="fr.unicaen.l3.Main"/>
            <attribute name="Class-Path" value="lib/mysql.jar"/>
        </manifest>
    </jar>
</target>
```

Principales tasks (4)

- Exécution avec java, lancement du jar :

```
<target name="run" depends="dist">
    <java jar="dist/application-${application.version}.jar" fork="true">
</target>
```

Ant avec Netbeans

- Netbeans permet d'utiliser ant (choisir « java with ant »)
- Par défaut, il crée lui-même son propre build.xml qui fait appel à build-impl.xml placé dans le répertoire nbproject
- Mais on peut créer un « java free-form project » plutôt qu'une « java application », et lui indiquer alors le répertoire de notre application où l'on a **préalablement** placé notre propre build.xml

Ant avec Netbeans (2)

- Netbeans reconnaît automatiquement les targets de votre build.xml en fonction de leur nom (run, build, etc.)
- On peut malgré tout corriger le mapping qu'il propose en faisant un clic droit sur le projet, puis dans « générer et exécuter », modifier manuellement le mapping
- Il est possible d'utiliser ant en ligne de commande ou dans netbeans, ou encore eclipse, avec le même fichier build.xml

Architecture possible

- créer un répertoire portant le nom de l'application, et y placer :
  - votre fichier build.xml
  - les répertoires suivants :
    - src : votre code source (.java)
    - lib : les éventuelles librairies (.jar)
  - Seront ajoutés par votre script build.xml :
    - build : le code compilé (.class)
    - dist : la distribution = .jar + copie du répertoire lib
    - doc : la javadoc
  - Nous verrons plus tard l'ajout de répertoires de tests

Librairie + Application

- Prenons l'exemple du TP, où l'on crée 2 projets :
  - personnagesJeu, un projet permettant de créer des personnages de jeu pour différents jeux possibles.
  - jeuCombat, un jeu au tour par tour exploitant le projet personnageJeu en tant que librairie.
- On génère un .jar de l'application personnagesJeu, qui fait alors office de librairie pour l'application jeuCombat :
  - placer personnagesJeu.jar dans le repertoire lib de jeuCombat.
  - faire un « add jar » de personnagesJeu.jar dans « classpath des sources java » des propriétés du projet dans NetBeans

Librairie + Application (2)

- Vous pourrez utiliser les deux fichiers build.xml fournis sur ecampus pour ces deux applications
- Difficultés :
  - Vous risquez de modifier la librairie personnagesJeu durant le développement de jeuCombat.
  - Il faut alors mettre à jour le personnagesJeu.jar situé dans le répertoire lib de jeuCombat.
  - Ceci est automatisé avec une instruction de recopie dans le build.xml de personnagesJeu (à ne garder que pendant la phase de développement simultané des deux projets !)
  - Ceci est dû au fait inhabituel que vous développez les deux vous même, alors que souvent, la librairie utilisée préexiste.

Librairie + Application (3)

- Attention, pour générer le .jar de jeuAssemblage, il est nécessaire d'ajouter la dépendance dans le classpath :

```
<jar jarfile="${dist.home}/${app.fullname}.jar"
        basedir="${build.home}">
       <manifest>
          <attribute name="Main-Class" value="jeuCombat.MainClass"/>
          <attribute name="Class-Path" value="lib/personnagesJeu-0.1.jar" />
       </manifest>
</jar>
```

Révisions et compléments : Le (méta) pattern MVC

Modèle

Vue

Contrôleur

Objectif

- Dissocier les trois "composantes". Ainsi :
- La conception du modèle n'est pas entachée par celles de la vue et du contrôleur
- On peut disposer de plusieurs vues (éventuellement différentes) sur un même modèle
- Réutilisabilité, en particulier du modèle

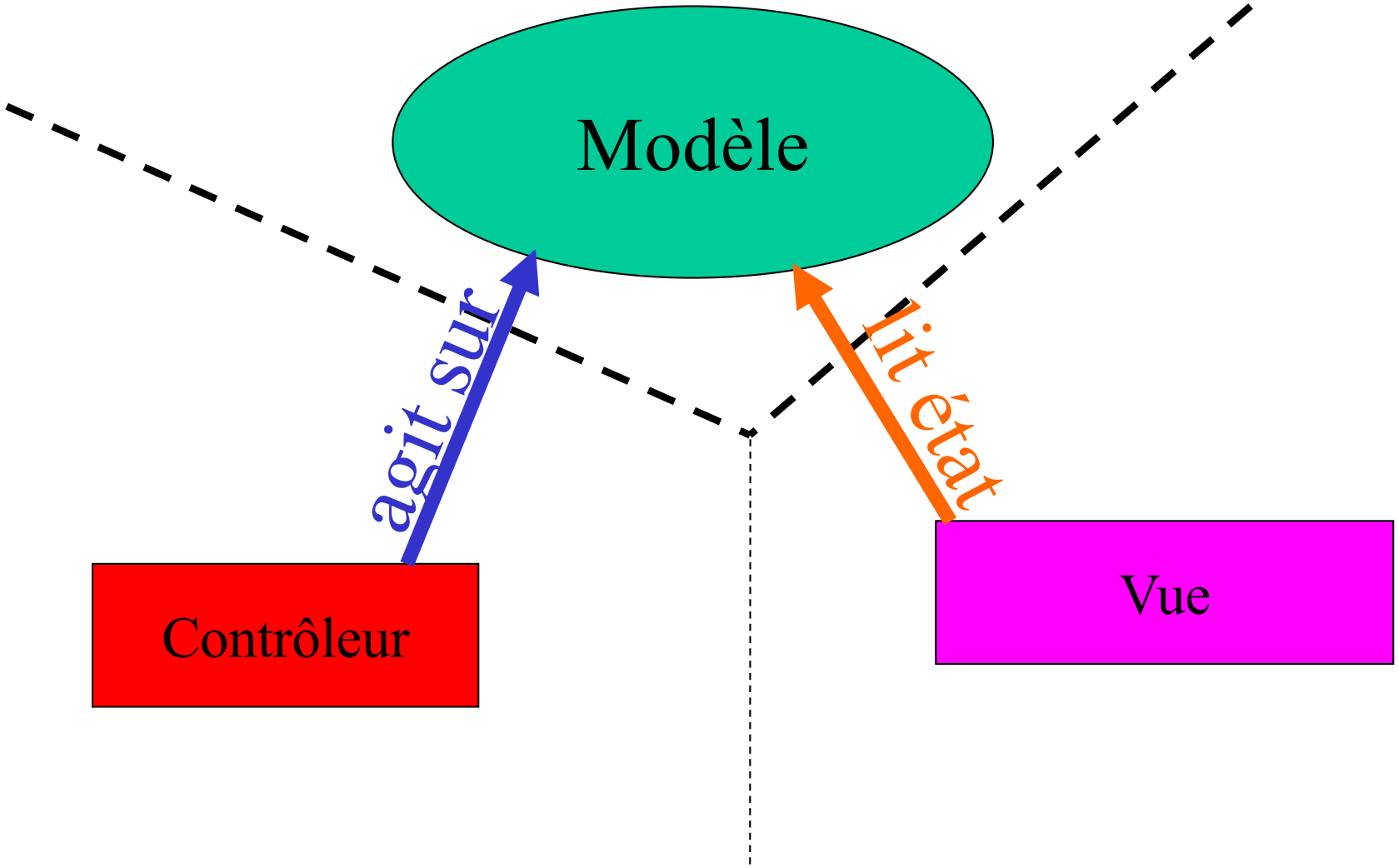
Principe

- Modèle : état et comportement
- Contrôleur : des listeners (de clavier, de souris), et des composants graphiques générant des événements.
- Vue : différents éléments Swing.

Principe (2)

- Le Modèle doit tout ignorer de V et de C
- Le Contrôleur doit posséder une référence sur le Modèle, pour agir dessus (ex. modifier une donnée, lancer un comportement)
- La vue doit aussi posséder une référence sur le Modèle, mais uniquement en lecture d'information (jamais en modification)

un modèle indépendant



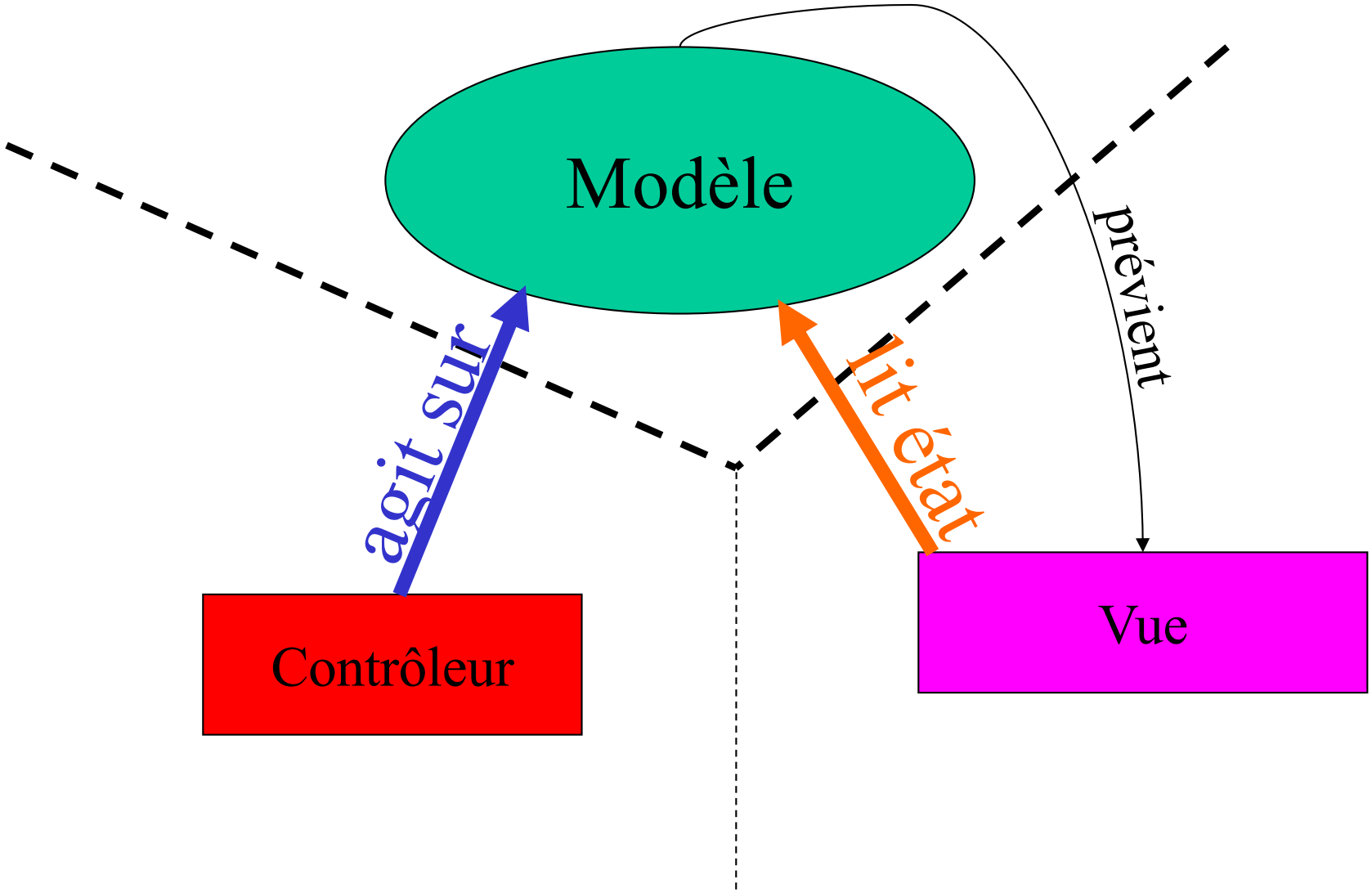
Indépendance du Modèle ?

- Si le modèle est construit en totale indépendance, comment la vue peut-elle savoir que son état a changé (il faut alors actualiser la vue) ?
- 1. On peut imaginer que le contrôleur, en même temps qu'il agit sur le modèle, le signale à la vue… Le modèle reste donc indépendant.
- 2. Le modèle peut aussi être conçu pour avoir des Listeners de ses modifications. Lorsque son état change, il prévient ses listeners, en lançant un événement (fire).

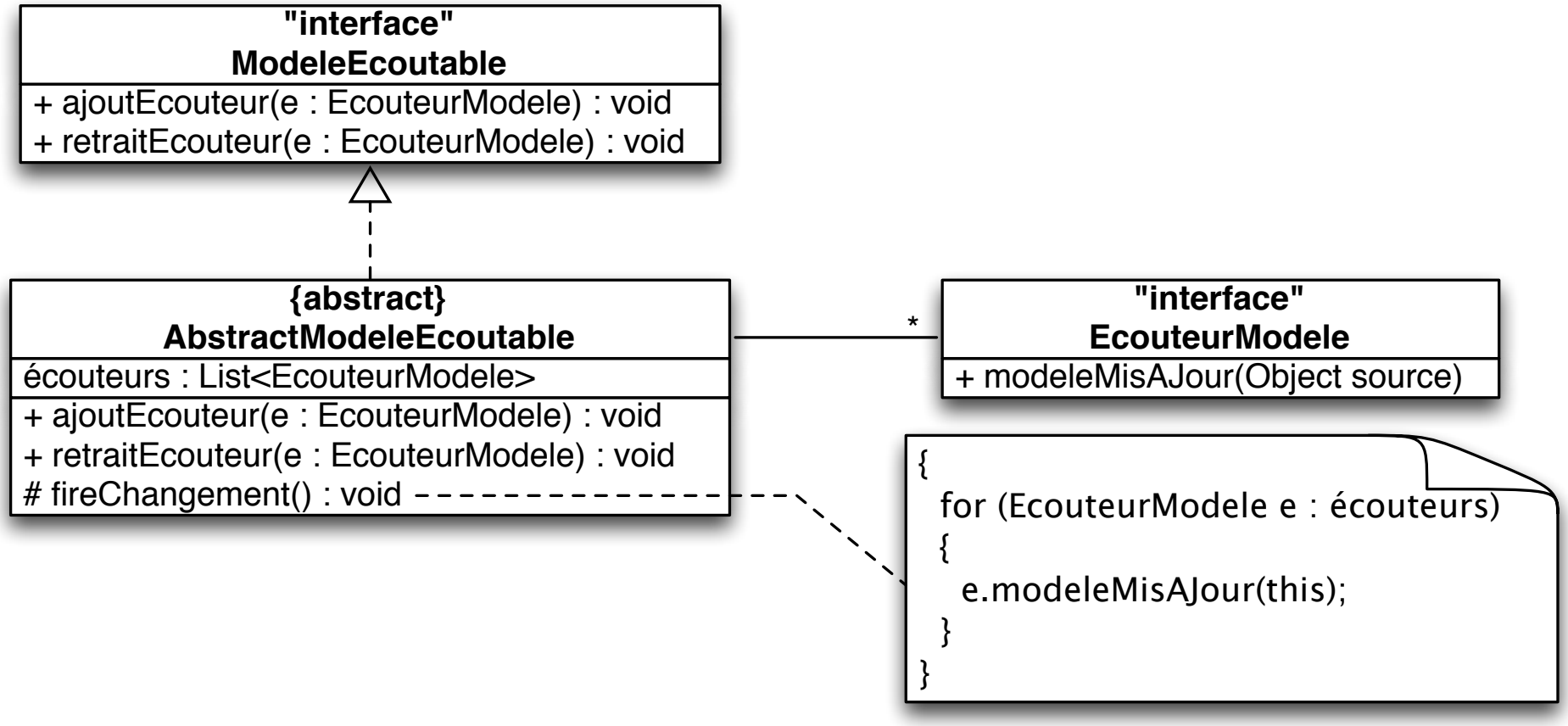
Quelle solution ?

- La solution 1 a le mérite de laisser le modèle totalement indépendant. Mais le contrôleur outrepasse son rôle. Mais surtout, l'état du modèle peut changer pour une raison autre que l'action du contrôleur !
- La solution 2 est donc préférable : c'est le modèle qui sait quand il change, quelle que soit l'origine de ce changement. De plus, le modèle ignore toujours tout du contrôleur et de la vue : il possède juste des "listeners" de ses éventuels changements.

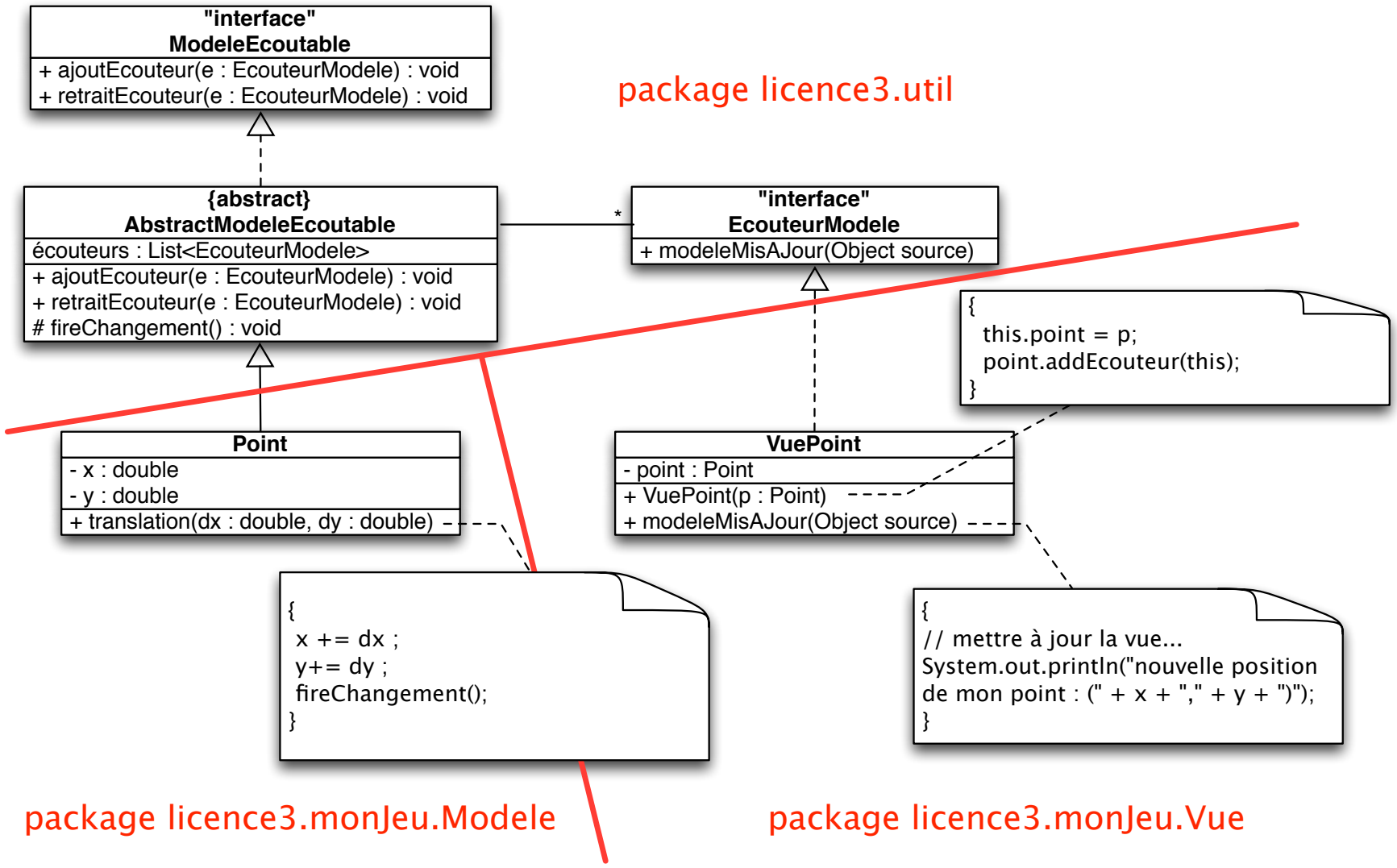
Un modèle écouté.



Implémentons Observer…



Application à MVC



Une subtilité…

L'interface ModeleEcoutable est utile lorsque l'on veut définir un nouveau type d'objets (écoutables) via une interface plutôt que par une classe. Dans ce cas là, l'interface de ce nouveau type hérite de ModeleEcoutable, et ses implémentations peuvent hériter (si elles n'héritent pas déjà d'une autre classe) de la classe AbstractModeleEcoutable (pour ne pas avoir à réécrire la partie événementielle…)

MVC ou M-VC

- Quand on le peut : MVC (exemple : calculatrice classique)
- Souvent : M-VC, car il n'est pas toujours aisé voire possible de distinguer le contrôleur de la vue (exemple : calculatrice autorisant la modification directe de la valeur affichée sur son écran JTextField).

MVC et Swing

- Les composants Swing sont bâtis selon le principe.
- La séparation n'est pas totalement explicite : une certaine classe de composant propose une "Vue-Contrôleur" sur un certain modèle.
- La vue peut être paramétrée par l'utilisation de "renderers"
- Ex :

- JTable (vue-contrôleur), TableModel (le modèle).
- JTree (vue-contrôleur), TreeModel (le modèle).

getModel() → référence vers le modèle

Exemple d'une calculatrice