Composants logiciels

Evolution de la réutilisation:

1. Copie de code source
2. Bibliothèques de fonctions
3. Bibliothèques de classes (modules)
4. Composants logiciels

Architectures de composants logiciels

Plug-ins: composants additionnels pour des applications (Photoshop, Word, etc.)

Applets Java: parties actives dans des documents, pas de communication

LiveConnect (Netscape): communication entre applets, javascripts, plug-ins

OpenDoc/SOM: communications entre parties hétérogènes de documents

ActiveX: communications entre “contrôles” à travers DCOM

Java Beans: composants actifs dans l’outil de développement et dans l’application finale

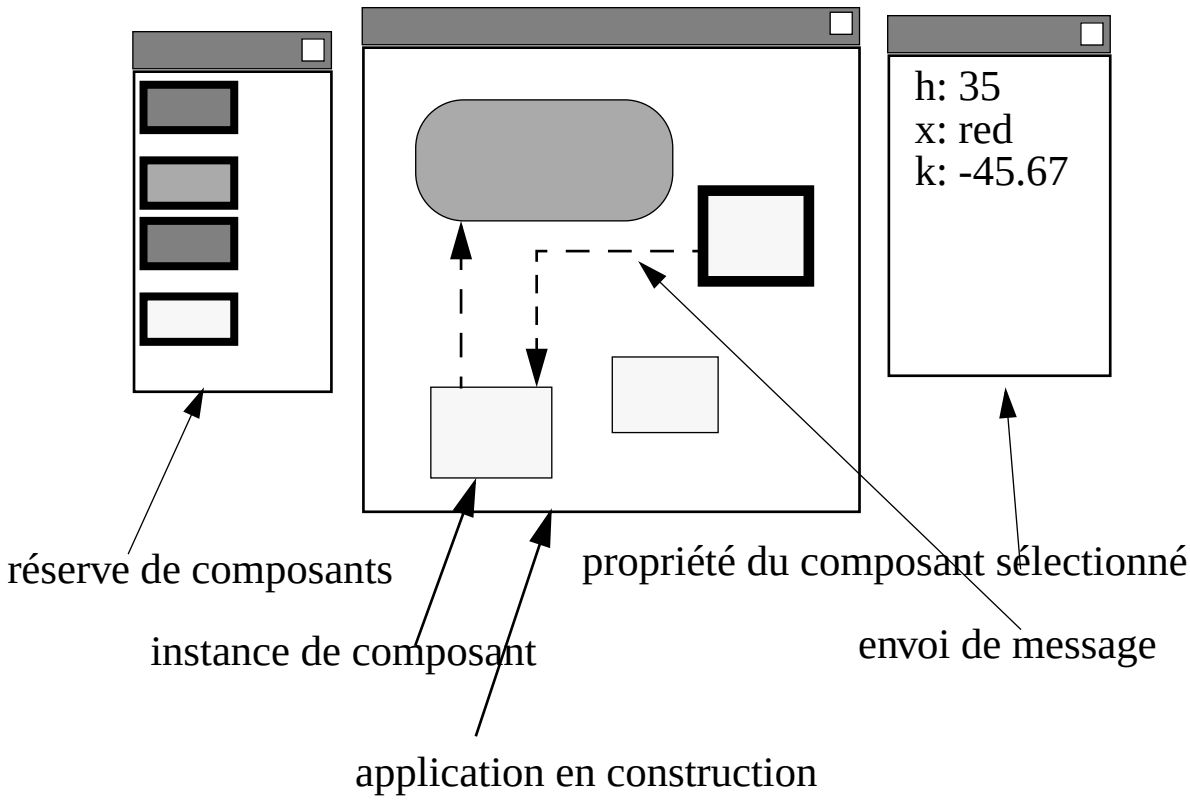
AppleScript: langage d’interconnexion de composants génériques, publication des capacités de chaque composant (terminologie), terminologies standardisées : BD, tableur, texte, comptabilité, etc.

Composants réutilisables: Java Beans

Bean: composant logiciel

– réutilisable

– manipulable visuellement dans un outil de développement (OD)



Beans

Un composant possède

- des propriétés (persistantes)
- des évènements reconnus
- des méthodes de traitement des évènements

Importance de l'introspection

Un outil de développement peut interroger un composant (quelles sont tes méthodes, tes variables, les évènements que tu traites, etc.) => pas de fichier de configuration.

Grâce à l'usage de modèles bien définis l'OD peut reconnaître les propriétés, évènements et méthodes du composant => utilisation dynamique du composant (le composant est "vivant") pendant le développement

L'outil peut créer des classes de connexion entre composants (associer des méthodes aux évènements)

Types de Beans

Component, Container, Invisible, Applet

La double vie des beans

Vie1: pendant la phase de configuration dans l'OD

- le bean peut posséder sa propre interface graphique de configuration

- à la fin le bean se sérialise

Vie2: pendant l'exécution de l'application finale

- le bean est déserialisé (rechargé)

- les méthodes de la phase exécution sont activées par les évènements

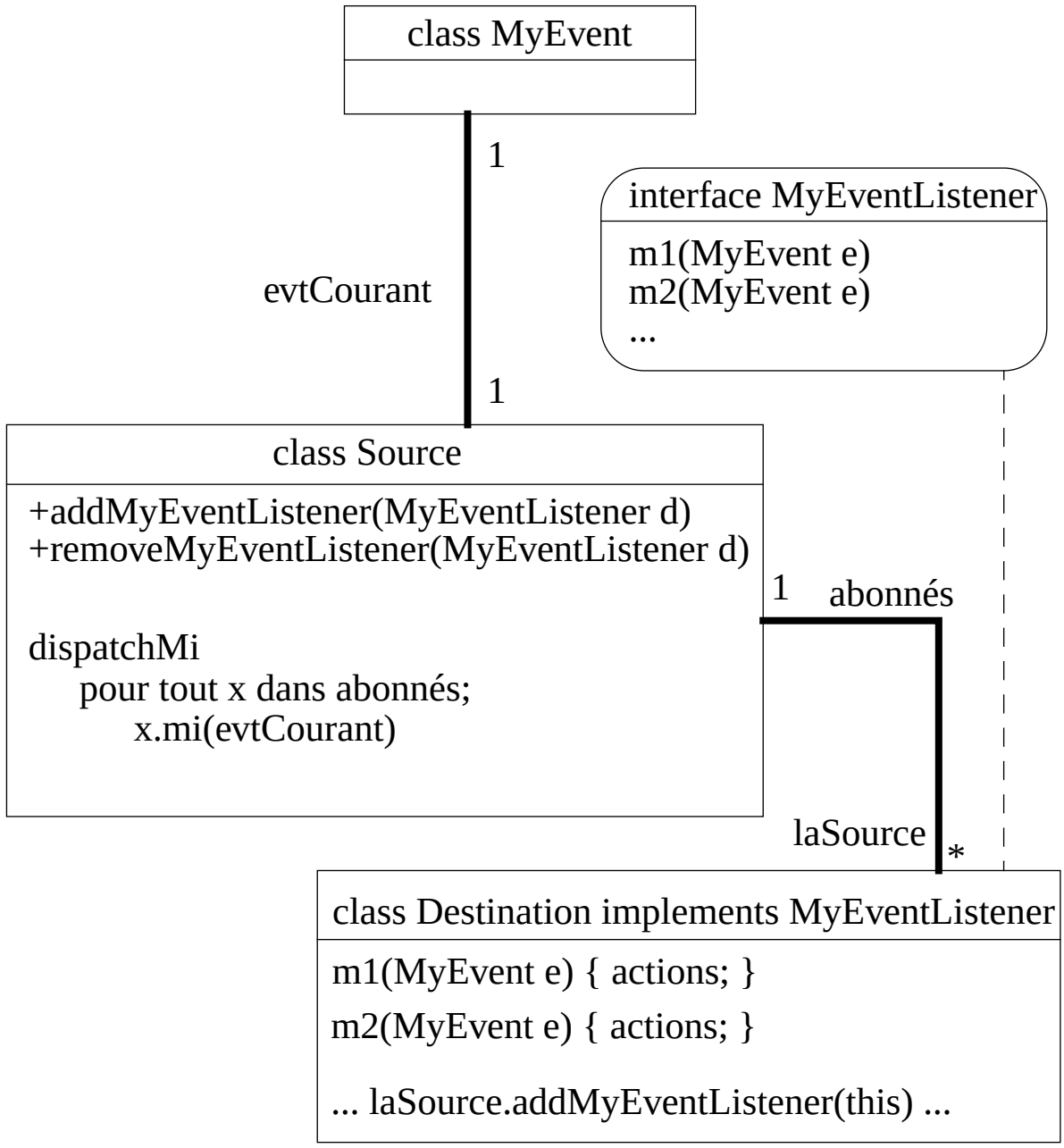
Modèle d'évènements

Modèle de délégation

Classe source d'évènements invoque des méthodes des classes destinations

Un objet Evenement est transmis

Les objets destination doivent s'inscrire auprès de l'objet source



Classes d’évènements prédéfinies

Evenements de bas niveau

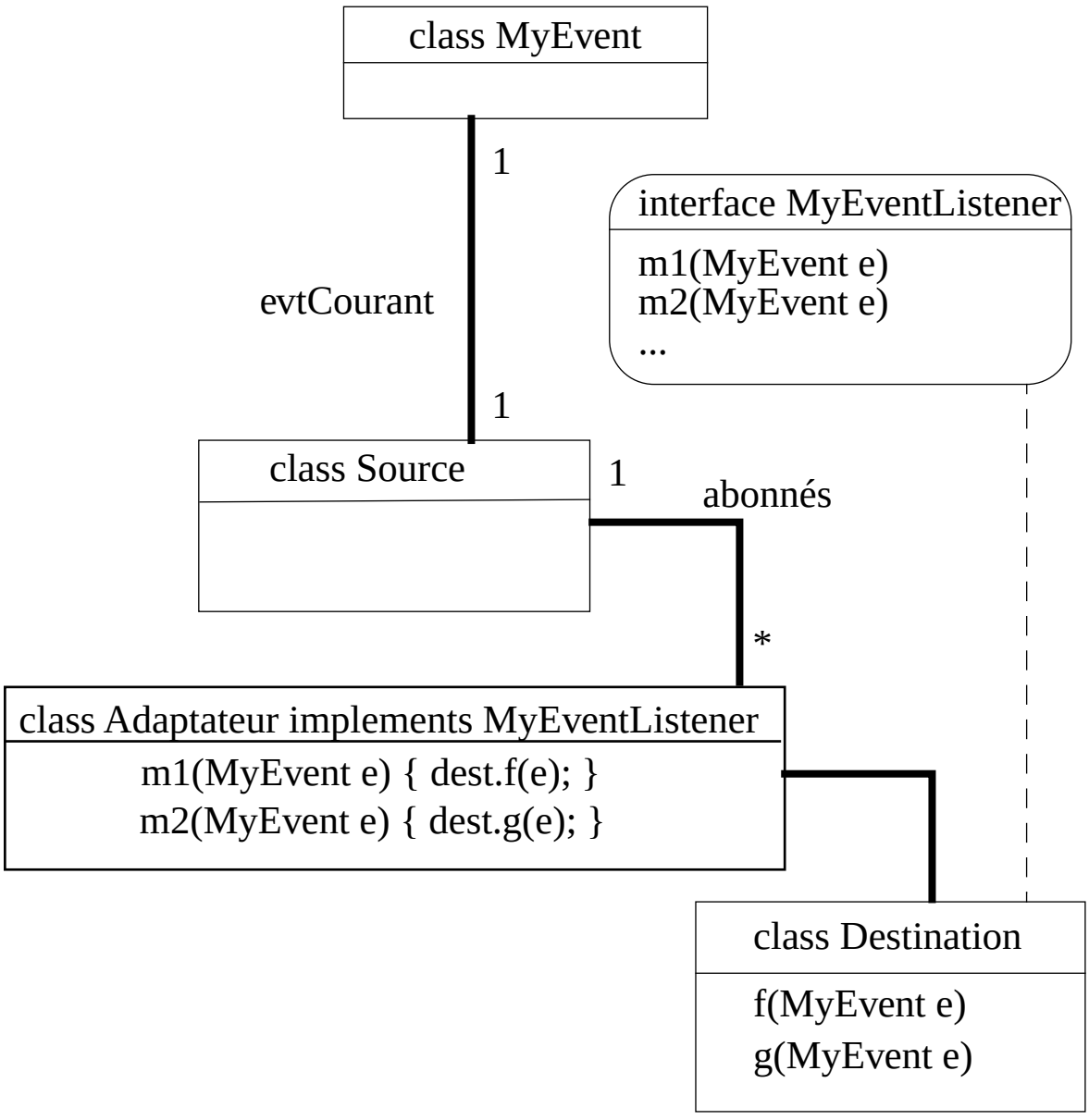
- java.util.EventObject
  - java.awt.AWTEvent
    - java.awt.event.ComponentEvent
    - java.awt.event.FocusEvent
    - java.awt.event.WindowEvent
    - java.awt.event.InputEvent
      - java.awt.event.KeyEvent
      - java.awt.event.MouseEvent

Evènements sémantiques

- java.util.EventObject
  - java.awt.AWTEvent
    - java.awt.event.ActionEvent
    - java.awt.event.AdjustmentEvent
    - java.awt.event.ItemEvent
    - java.awt.event.TextEvent

Adaptateurs

Intermédiaire entre un ou des objets écouteur qui n'implémentent pas l'interface *MyEventListener* et une source d'évènements du type *MyEvent*.

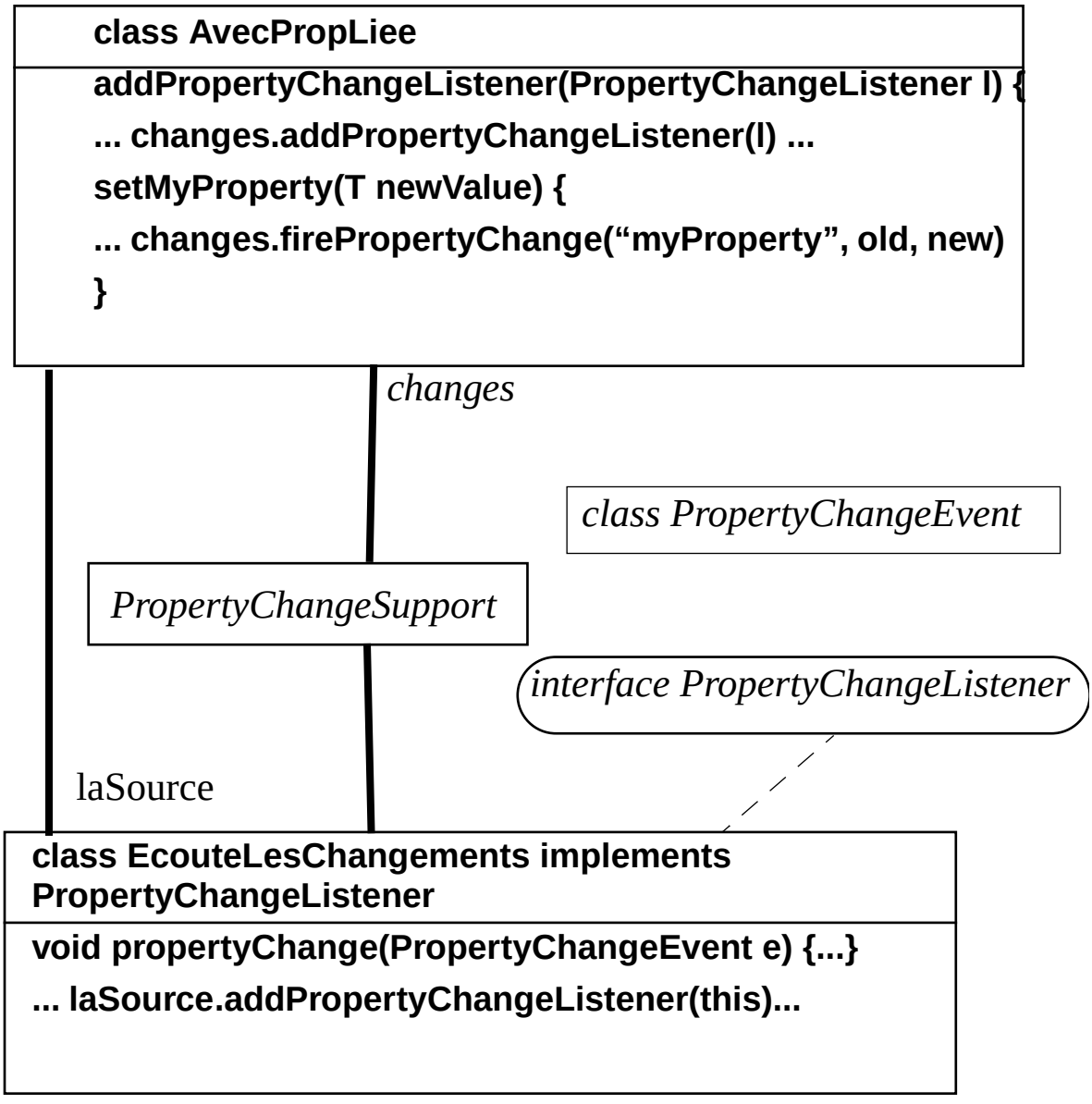


P.ex. pour démultiplexer des sources d'évènements.

Notification des changements de valeurs de propriétés et propriétés contraintes

Propriétés liées

But: transmettre à d'autre objets tout changement de valeur d'une propriété donnée.



Signatures de design des beans

Pour que l'introspection permette de reconnaître les propriétés, évènements, etc. il faut respecter des normes de dénomination.

Ces normes ne sont pas vérifiées par les compilateurs Java !

Quelques exemples:

Propriétés:

```
public void set<propertyName>
            (<PropertyType> <propertyName>)
public <PropertyType> get<propertyName>()
```

Propriétés indexées:

```
public void set<propertyName>
    (int <indexName> <PropertyType> <propertyName>)
public <PropertyType> get<propertyName>(int <indexName>)
```

Propriétés contraintes:

```
public void set<propertyName>
(<PropertyType> <propertyName>) throws
PropertyVetoException
```

Source d'évènements

```
public void add<EventListenerType>
        ( <EventListenerType> <eventListenerName>)
public void remove<EventListenerType>
        ( <EventListenerType> <eventListenerName>)
```

Sérialisation des beans

Nécessaire pour conserver l'état final de configuration

Lorsque l'application est entièrement configurée, l'outil de développement sérialise chaque bean.

Classes sérialisables

implements java.io.Serializable;

possède un constructeur sans paramètres

ne fait référence qu'à des classes sérialisables (sauf les transient)

Problème : gestion des versions

Que se passe-t-il si la définition d'une classe change entre la sérialisatin et la désérialisation d'un objet ?

Certaines modifications sont autorisées

- ajout d'une variable d'instance

- ajout d'une méthode

- etc.

D'autre sont interdites

- suppression d'une variable d'instance

- changement de nom de la classe

- etc.

Introspection et reflexion

Un système peut se regarder lui-même

Les objets de la classe Class représentent des classes du système

Les objets de la classe Method représentent des méthodes

Exemples:

Recherche une classe d'après son nom

```
Class classButton = Class.forName("Button");
```

Crée une nouvelle instance de cette classe

```
Object myButton = classButton.newInstance();
```

Récupère toutes les méthodes de cette classe

```
Method [] bmethods = classButton.getDeclaredMethods();
```

S'il existe une méthode clear avec un paramètre de type Rectangle, on l'appelle

```
Class [] argTypes = { Class.forName("Rectangle") }
try {
    Method clear = classButton.getDeclaredMethod
        ("clear", argTypes)
    Objects[] args = {monRectangle};
    clear.invoke(myButton, args);
catch (NoSuchMethodException e) {
```

On peut donc contrôler complètement un système (p.ex. écrire un moniteur d'exécution de tâches).

Un outil de développement utilise les beans de cette manière.