IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

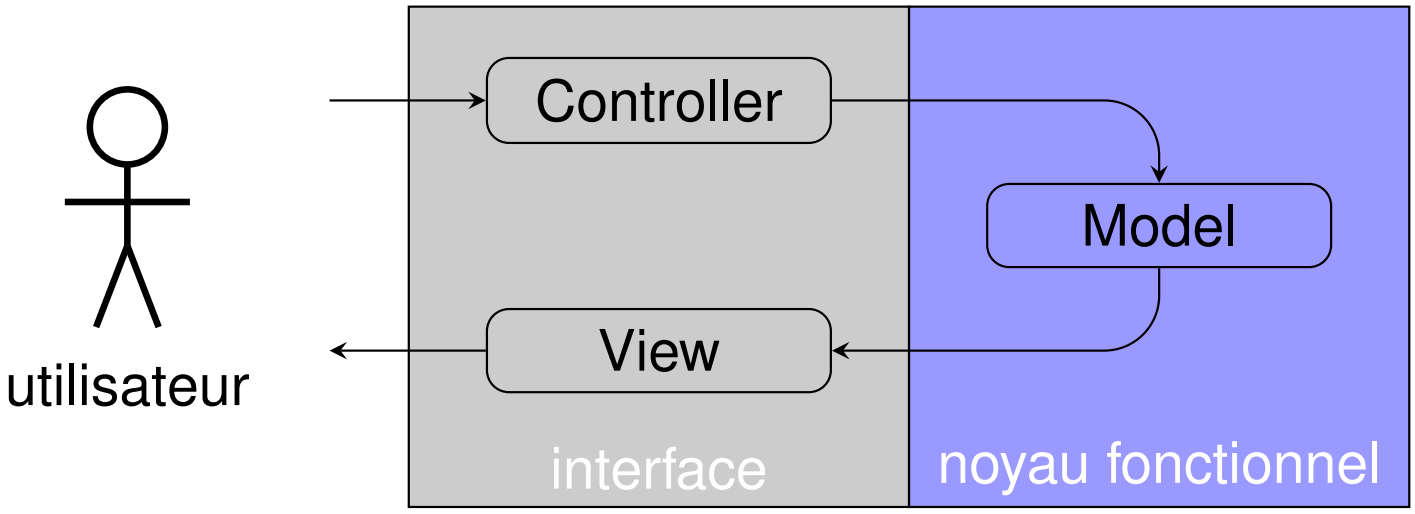
Conception d'IHM
Swing en scala

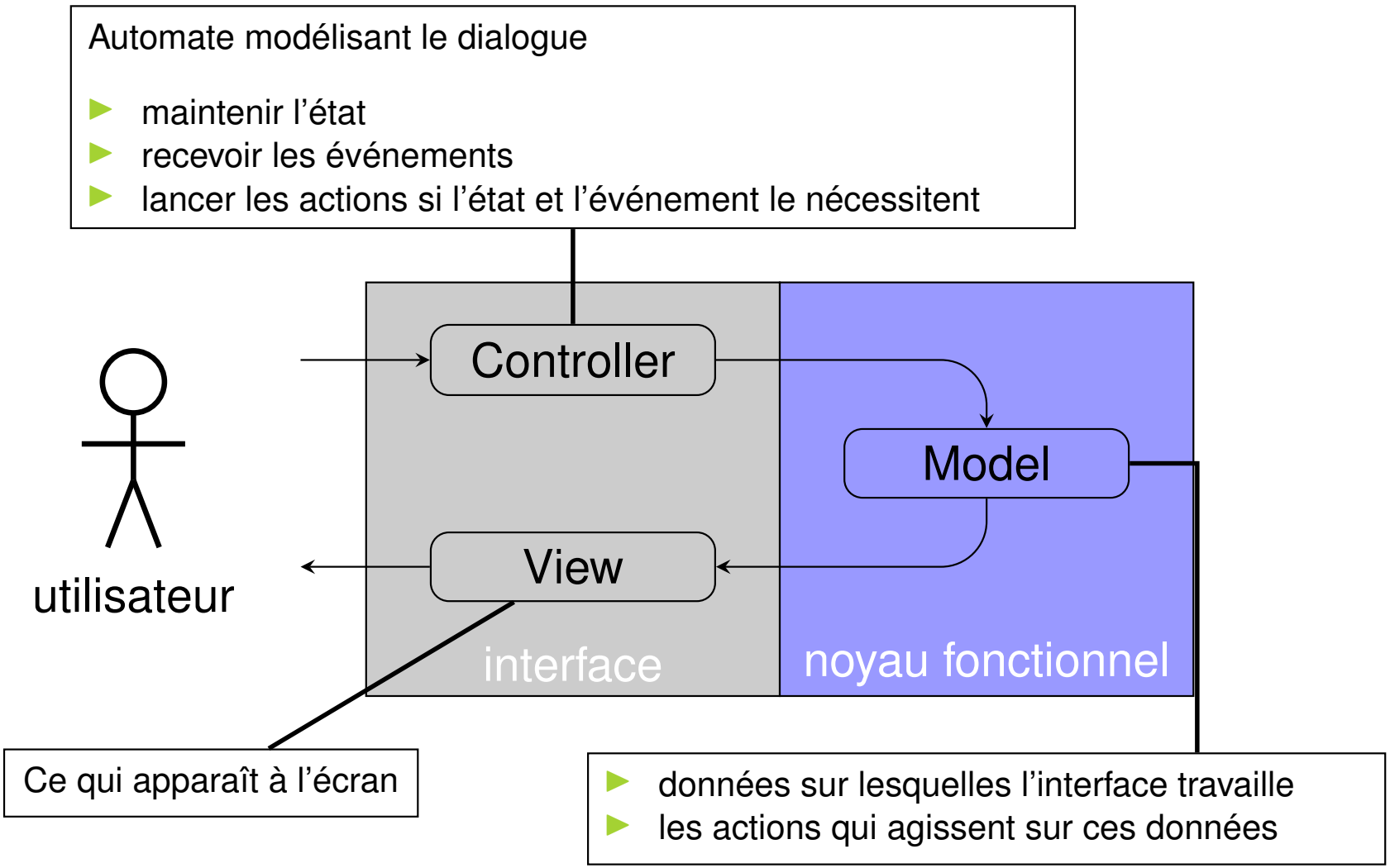
Fabien Dagnat

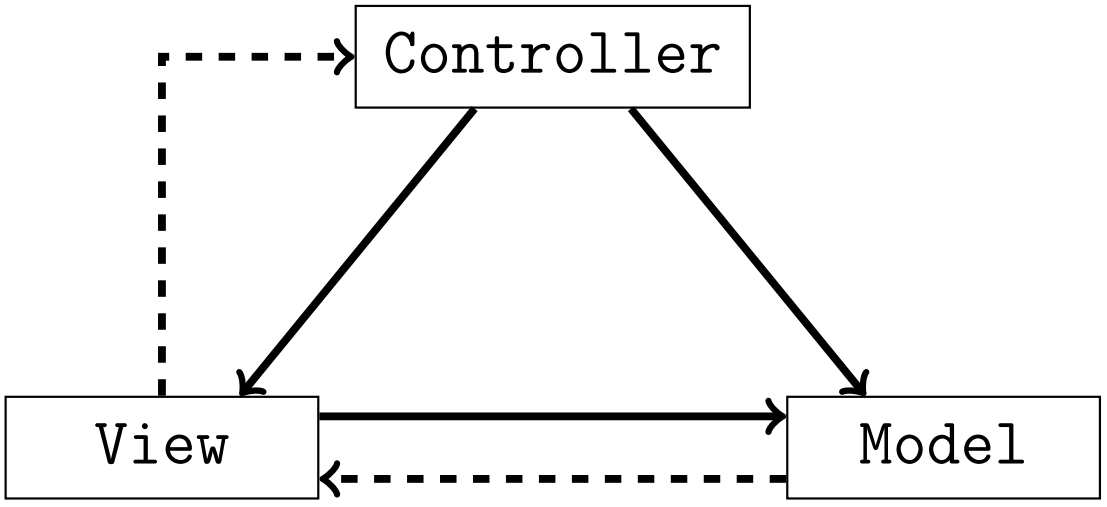
F2B304 – C7

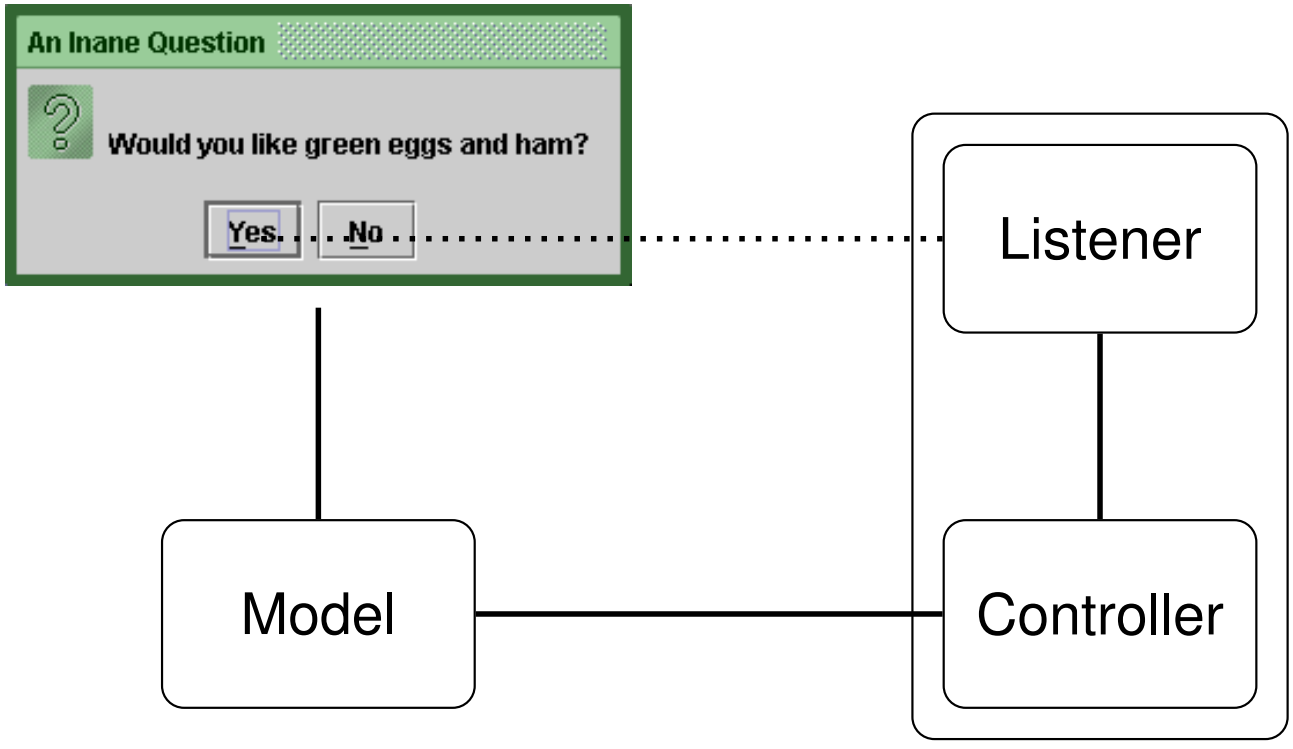
année 2018-2019

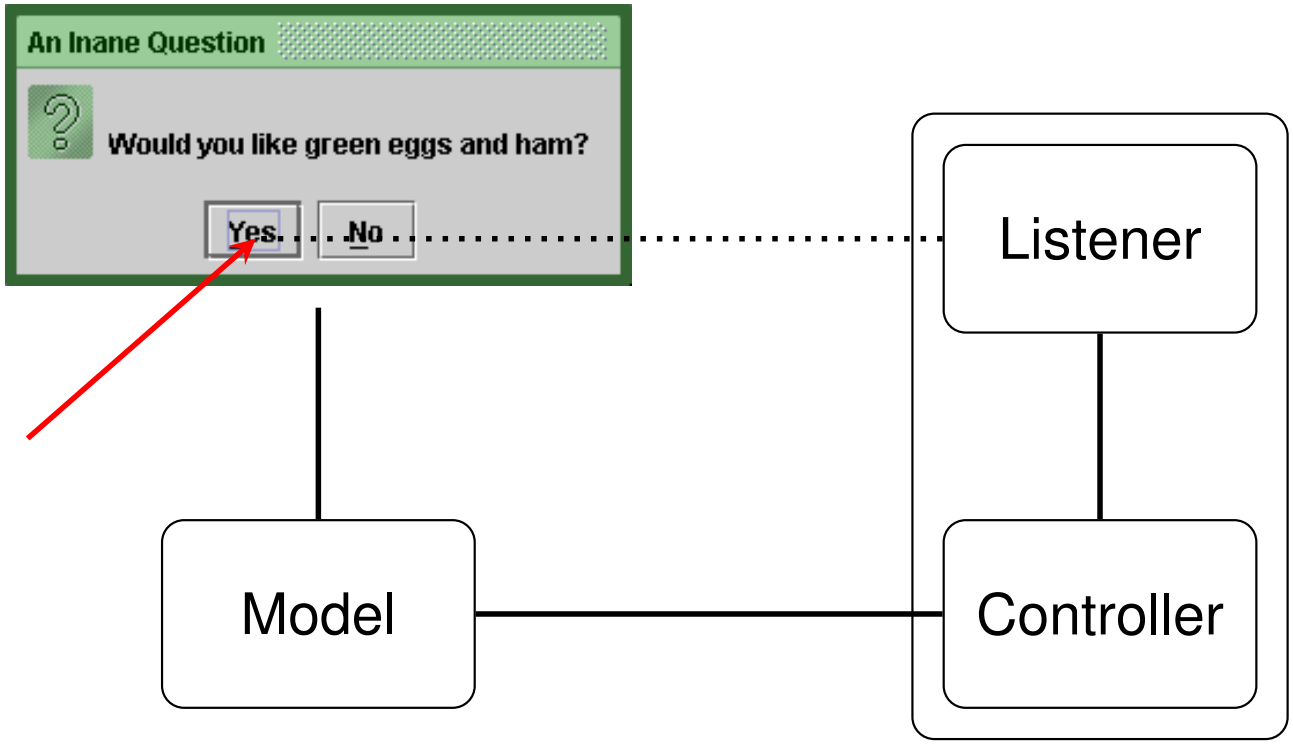
Plan

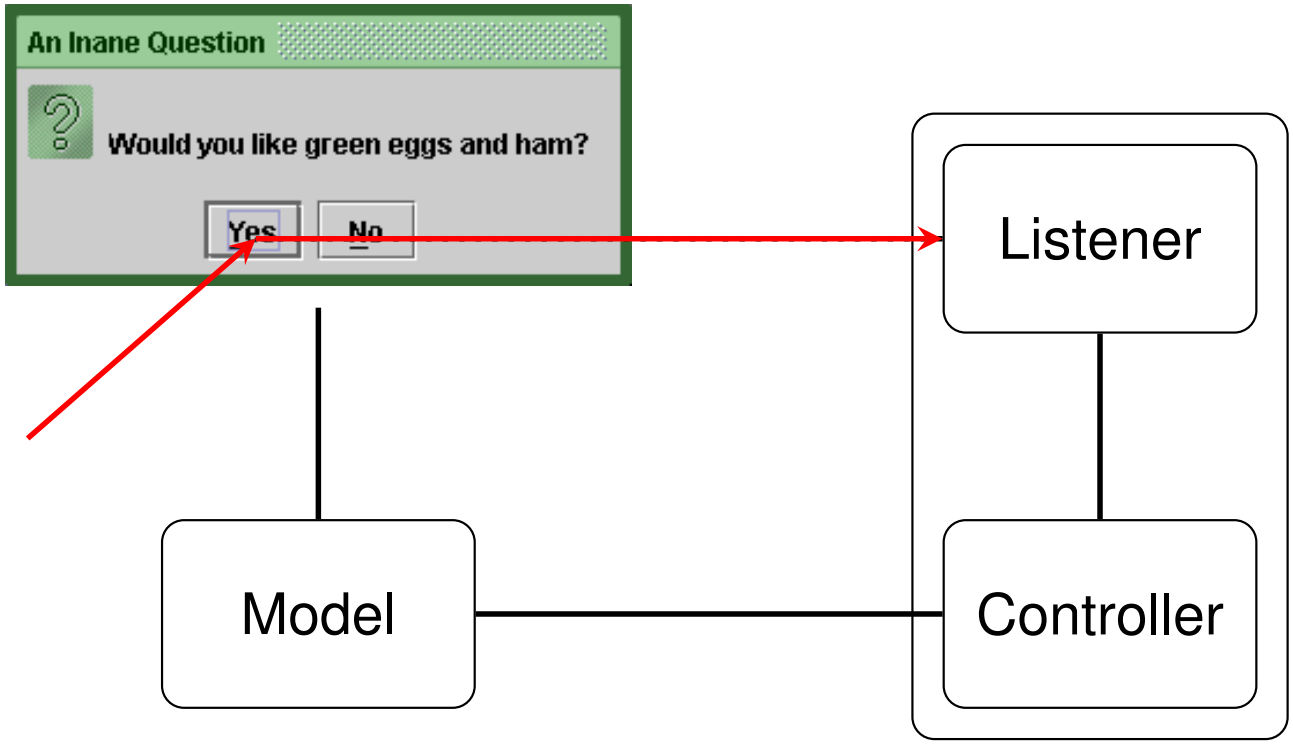

Controller
Model
View
utilisateur
interface
noyau fonctionnel

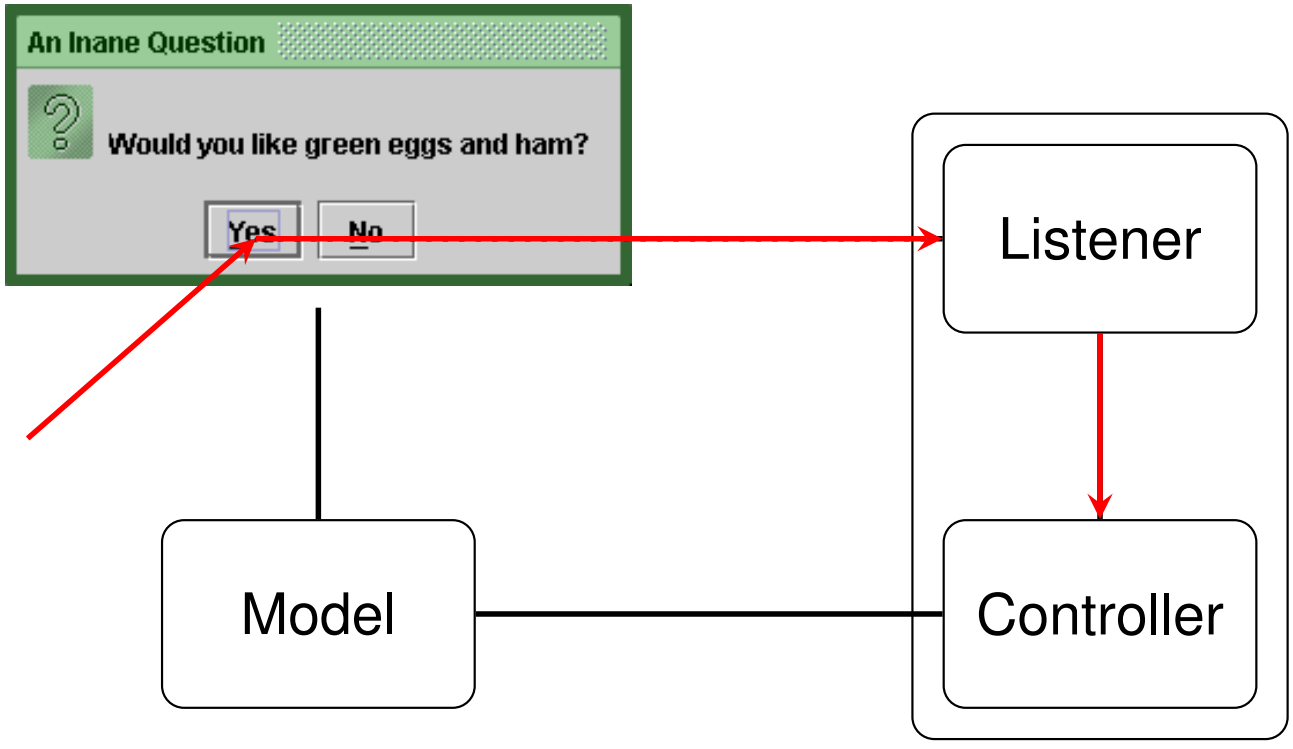

Automate modélisant le dialogue
maintenir l'état
recevoir les événements
lancer les actions si l'état et l'événement le nécessitent
Controller
Model
View
utilisateur
interface
noyau fonctionnel
Ce qui apparaît à l'écran
données sur lesquelles l'interface travaille
les actions qui agissent sur ces données

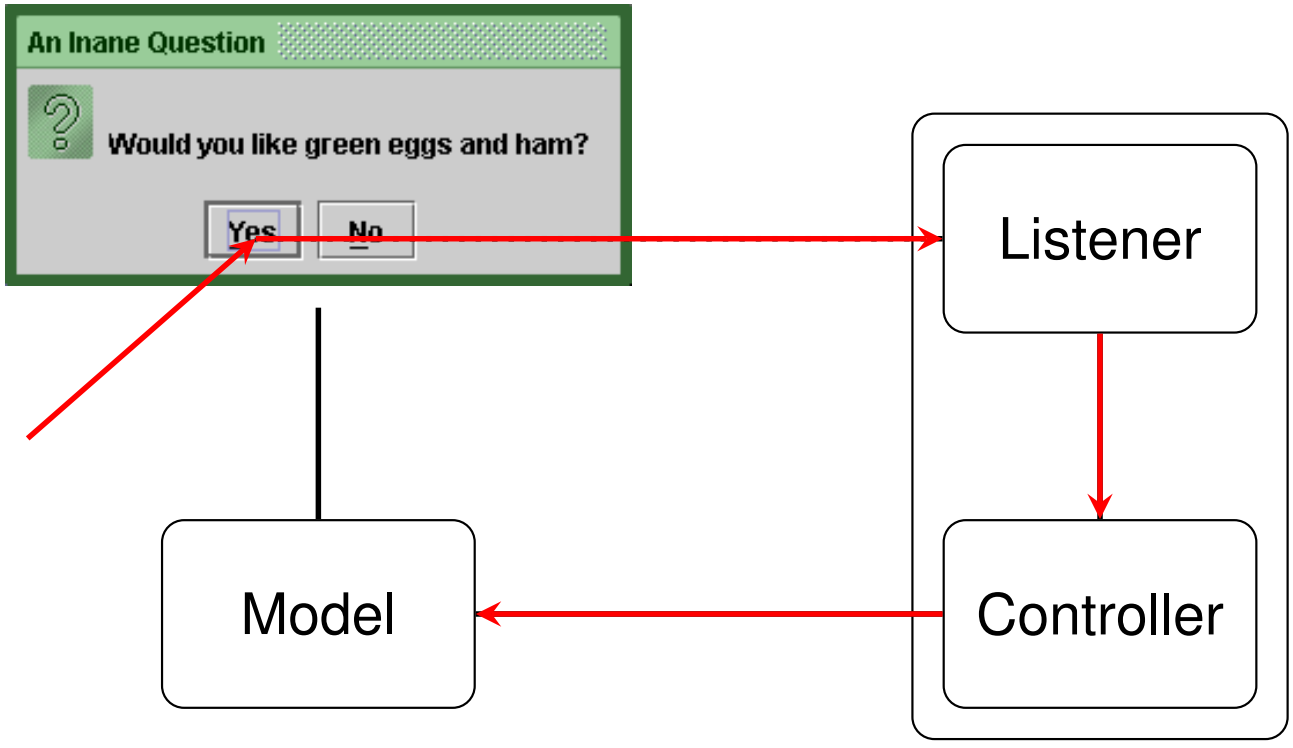

Controller
View
Model

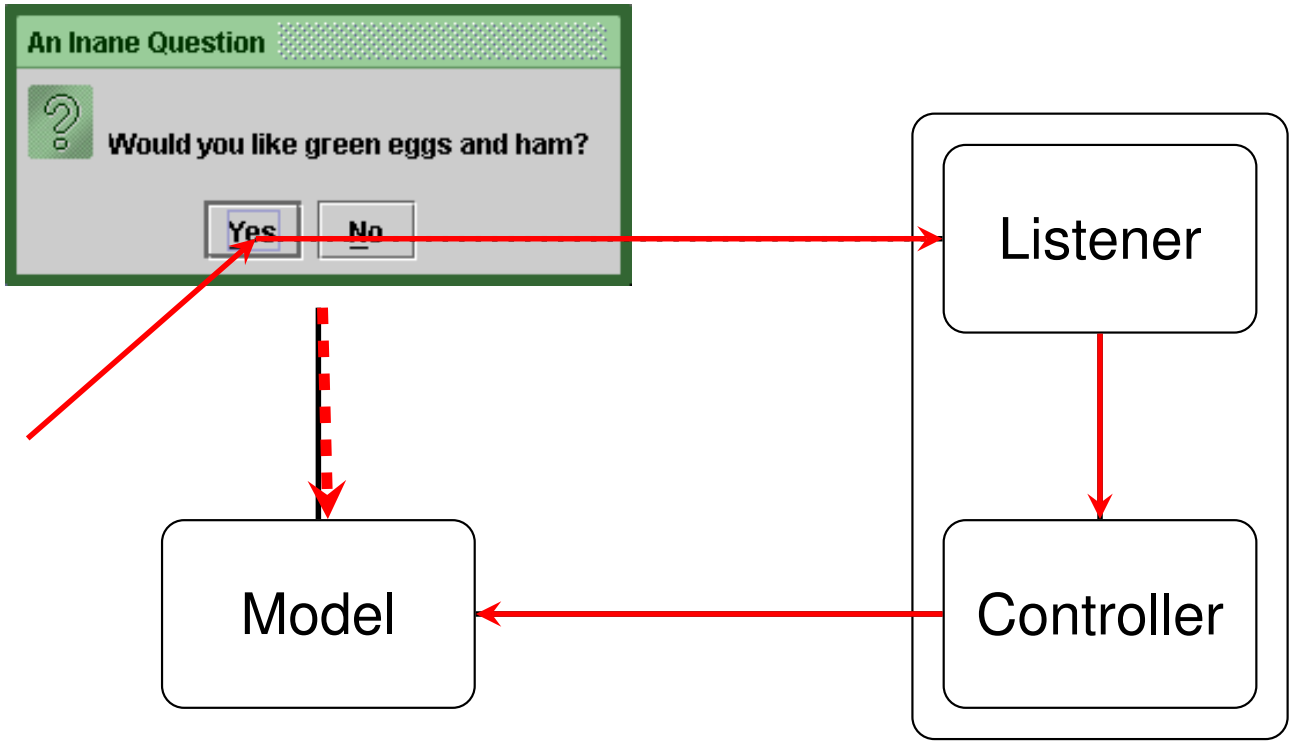

An Inane Question
Would you like green eggs and ham?
Yes
No
Listener
Model
Controller


An Inane Question
Would you like green eggs and ham?
Yes
No
Listener
Model
Controller


An Inane Question
Would you like green eggs and ham?
Yes
No
Listener
Model
Controller


An Inane Question
Would you like green eggs and ham?
Yes
No
Listener
Model
Controller


An Inane Question
Would you like green eggs and ham?
Yes
No
Listener
Model
Controller

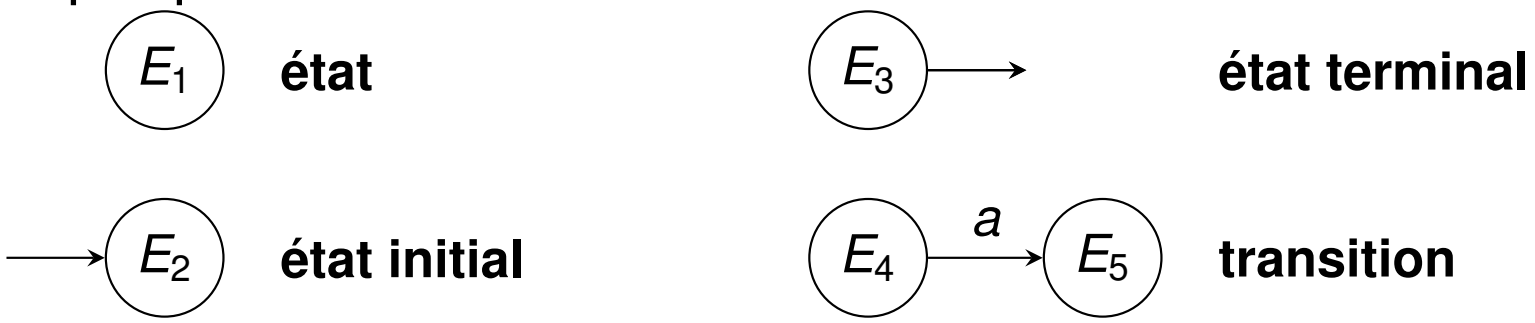

An Inane Question
Would you like green eggs and ham?
Yes
No
Listener
Model
Controller

- Automate d'états finis étendu pour décrire le dialogue

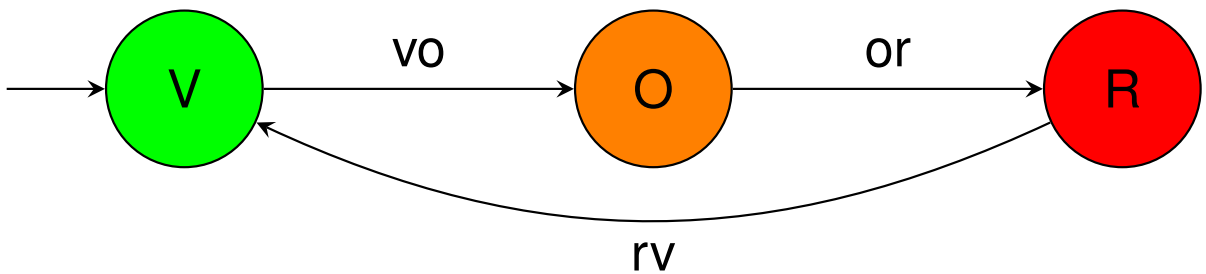
$\Rightarrow$ Construction de la matrice états/événements

$\Rightarrow$ Construction du gestionnaire d'événements, le contrôleur

- Représenter un système par des états et des transitions
- Graphe orienté dont
  - les nœuds sont les états
  - les arcs décrivent la fonction de transition et sont étiquetés par l'alphabet
- Graphiquement



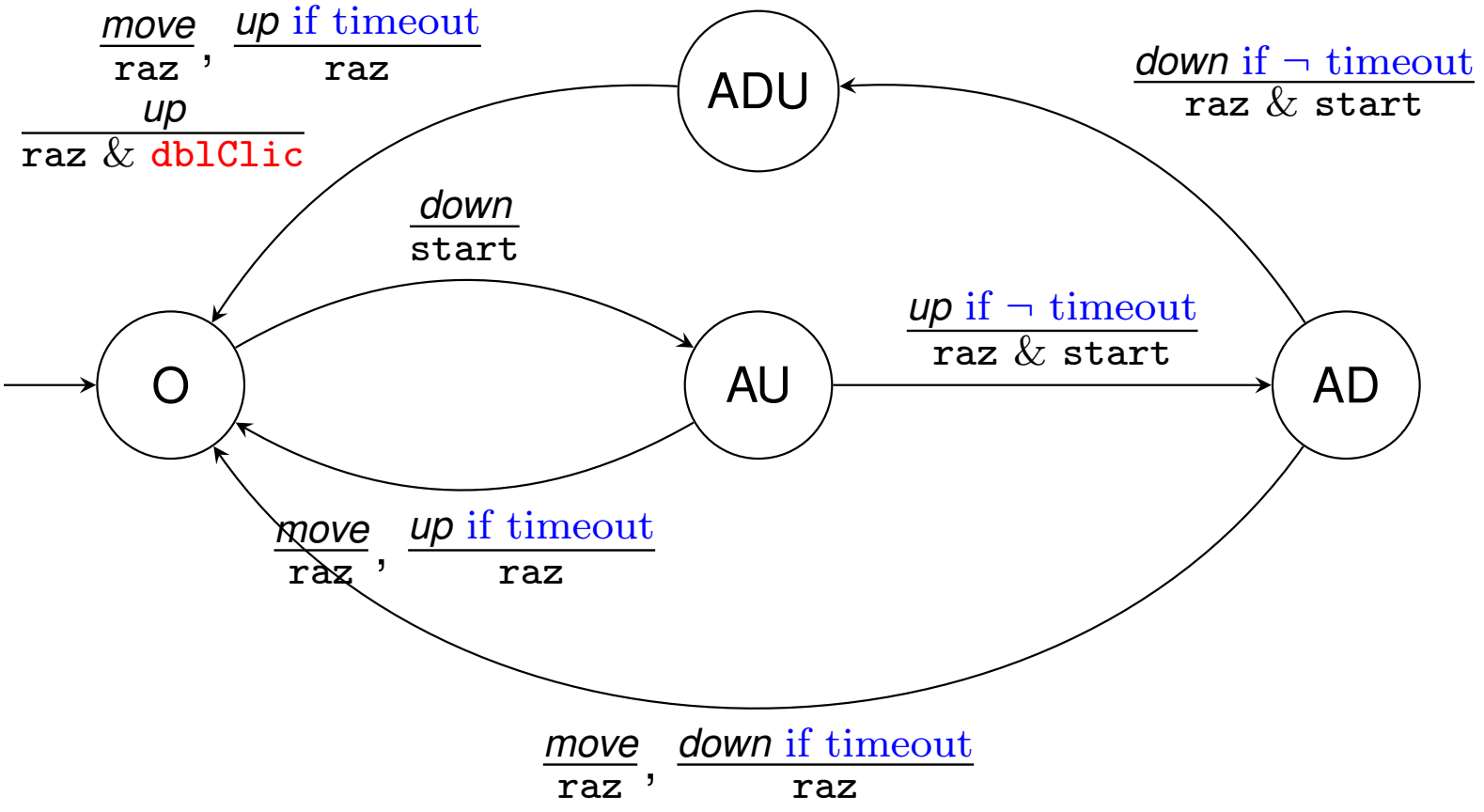
- Données : 3 lampes verte, orange et rouge
- États
  - V = (vert allumé, orange éteint, rouge éteint) – initial
  - O = (vert éteint, orange allumé, rouge éteint)
  - R = (vert éteint, orange éteint, rouge allumé)
- Actions
  - vo : Éteindre vert et allumer orange
  - or : Éteindre orange et allumer rouge
  - rv : Éteindre rouge et allumer vert



- Permettre une spécification plus précise de la transition
  - ajout d'événements déclencheurs et de conditions
- Une transition peut donc être étiquetée par
  - un événement
  - une condition
  - une action
- Les conditions portent sur des données que peuvent manipuler les actions

- Données : un *timer*
- États
  - O : Oisif (état initial)
  - AU : Attend Up
  - AD : Attend Down
  - ADU : Attend deuxième Up
- Événements
  - move : la souris bouge
  - down : le bouton est enfoncé
  - up : le bouton est relaché
- Actions
  - `start` : Démarrer le timer
  - `raz` : Remettre à zéro le timer
  - `dblClic` : Gérer le double clic

Un exemple, le double clic souris



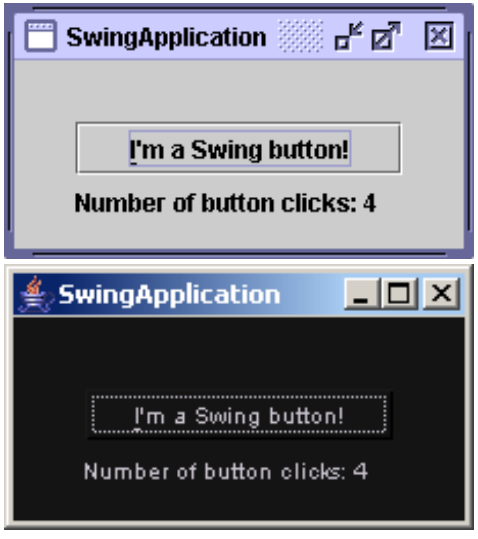
	move	up	down
O			AU
AU	O / `raz`	O / `raz` if to [1] AD / `raz` & `start` if ¬to	×
AD	O / `raz`	×	O / `raz` if to ADU / `raz` & `start` if ¬to
ADU	O / `raz`	O / `raz` if to O / `raz` & `DblClic` if ¬to	×

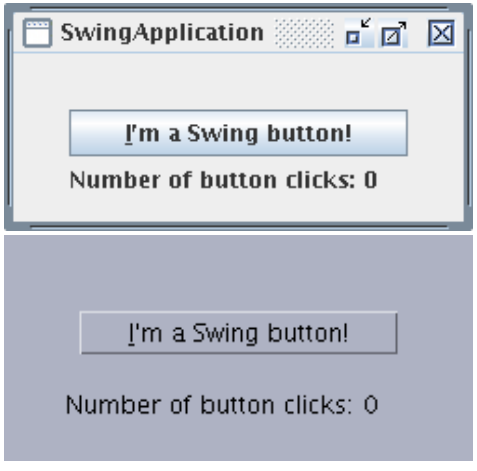
1. to=timeout

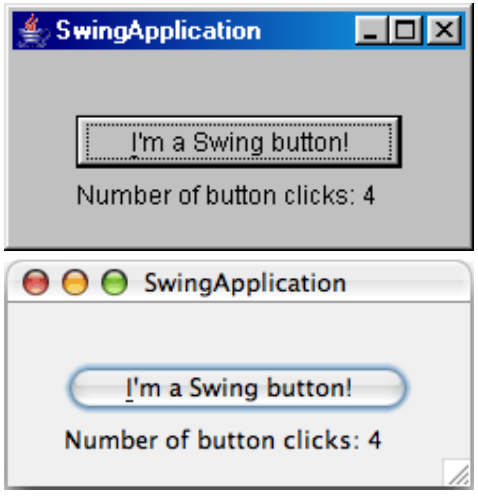
- Différentes bibliothèques
  - AWT : `java.awt`, gestion native d'interface simple
  - Swing : `javax.swing`, construit au-dessus de AWT (modèle MVC, composants sophistiqués...)
  - SWT : `org.eclipse.swt`, abandon (partiel) de la portabilité au profit de l'efficacité
  - ...
- Intérêts de Swing
  - portabilité
  - puissance des composants
  - modèle abstrait moderne, sophistiqué et réellement objet
- Tutorial détaillé de Oracle sur internet

  `https://docs.oracle.com/javase/tutorial/uiswing`

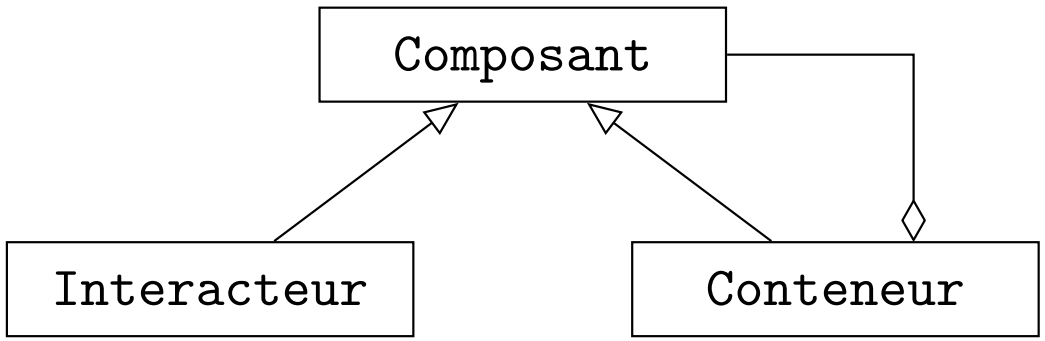
- Possible de choisir le LaF (dynamiquement)
- Nombreux LaF disponibles, on peut s'en faire un
- Par défaut, LaF identique sur tous les systèmes
- Pas toujours portable, car dépendant souvent du système



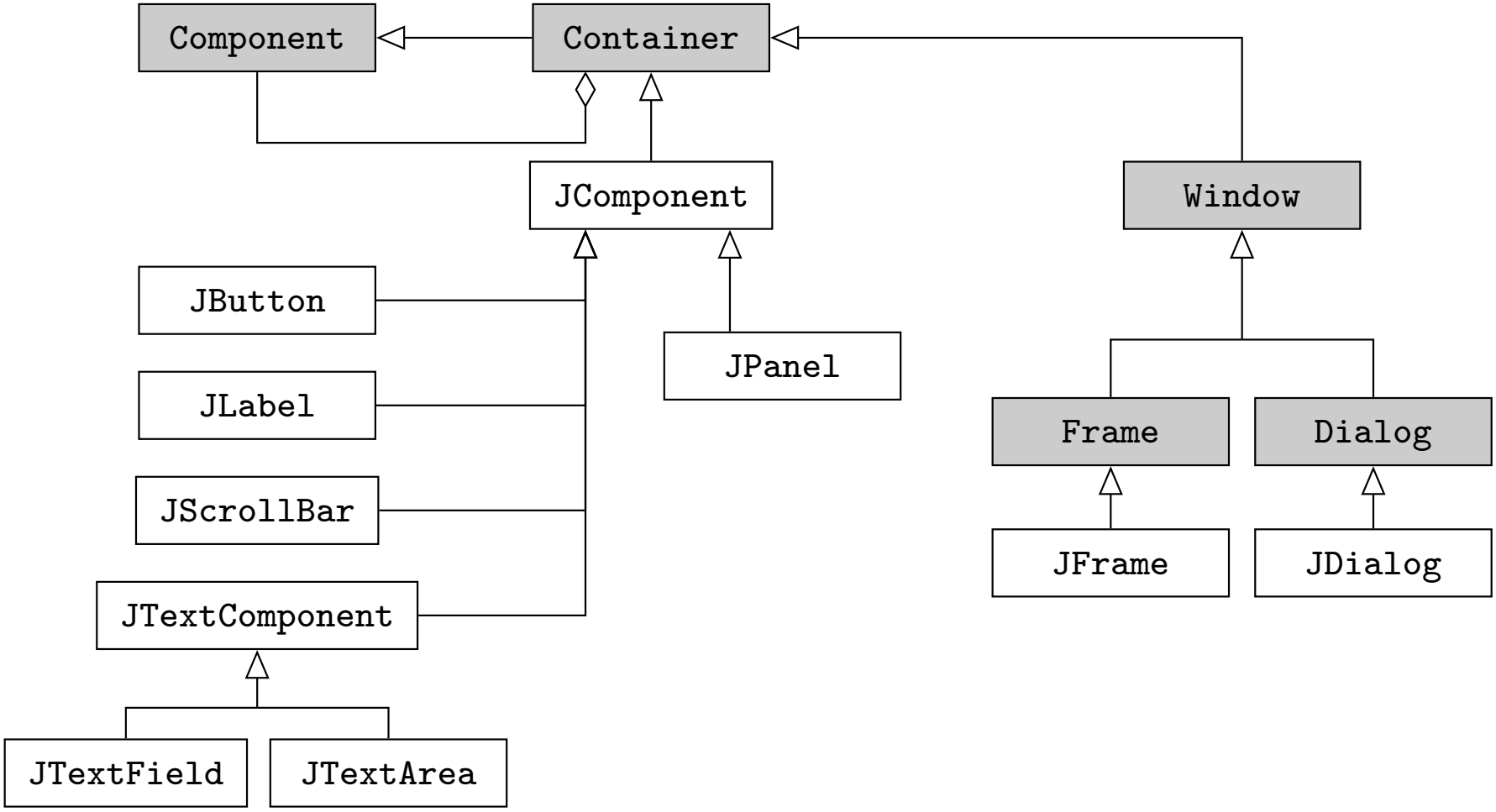




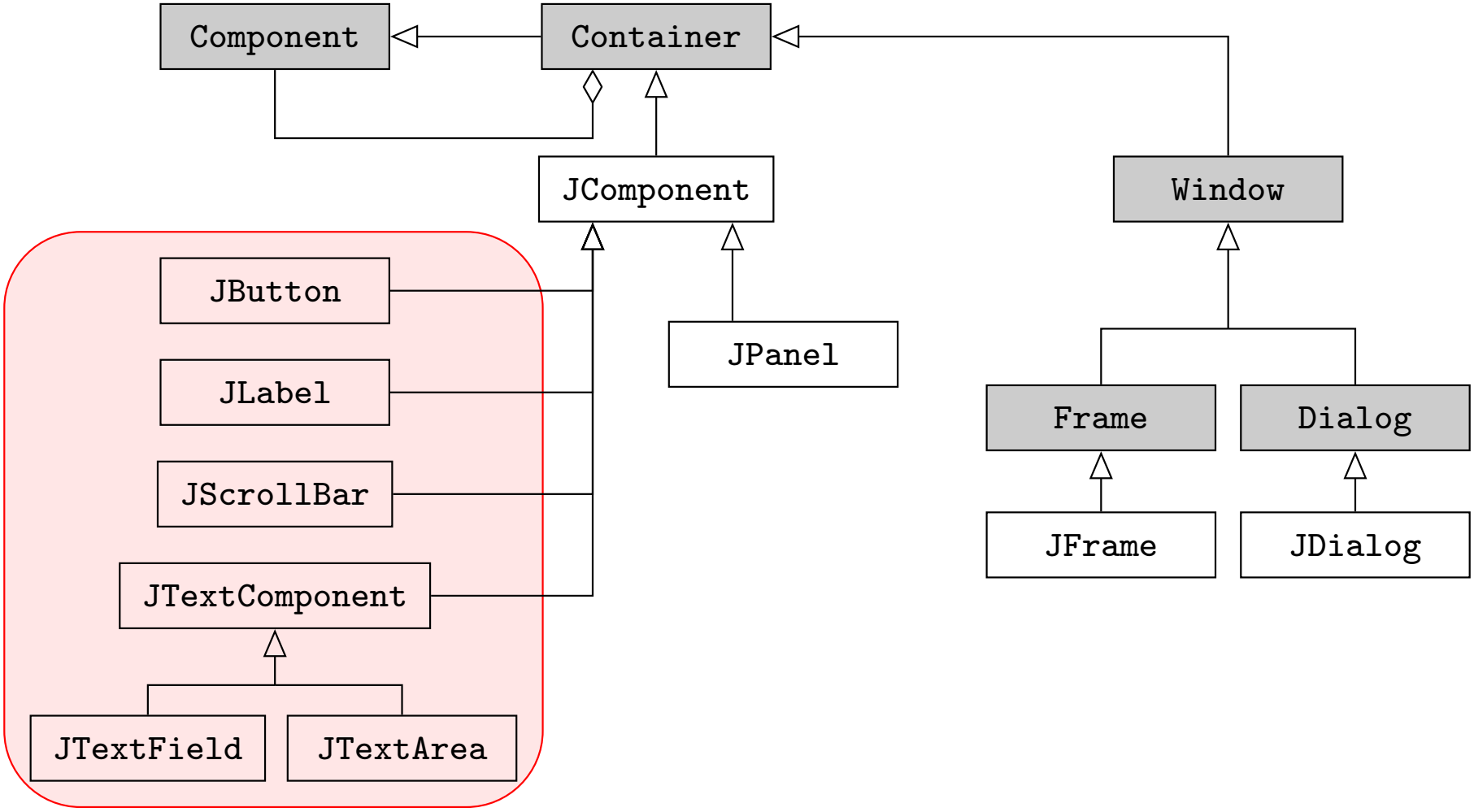
- La brique de base est le composant
  - il contient des propriétés pour la visualisation
  - il définit son service par des méthodes spécialisées
  - il est associé à des événements
- Certains composants sont des conteneurs
  - responsables de leurs sous-composants
  - peuvent être globaux (`JFrame`, ...) ou non



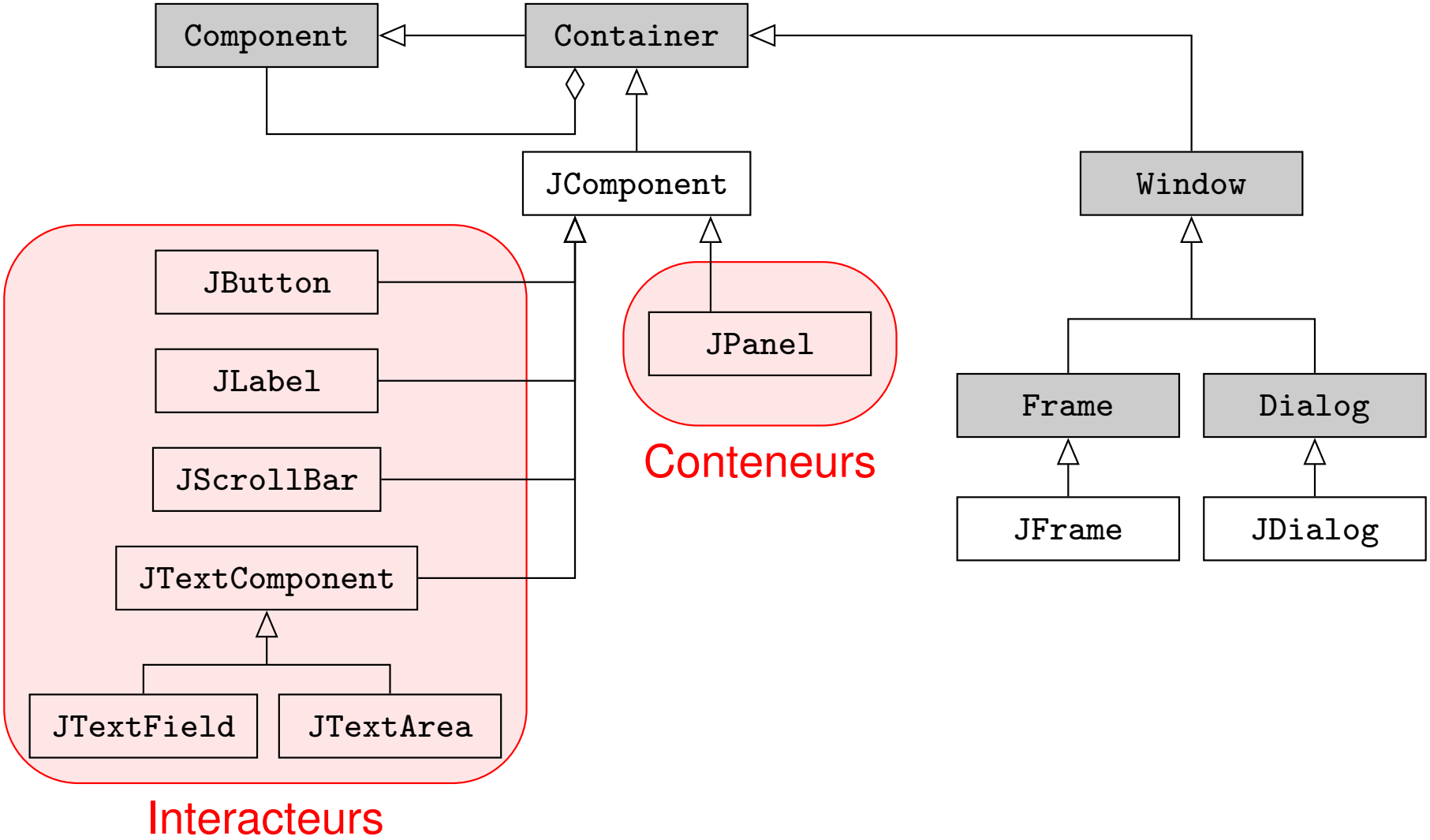
Hiérarchie Swing

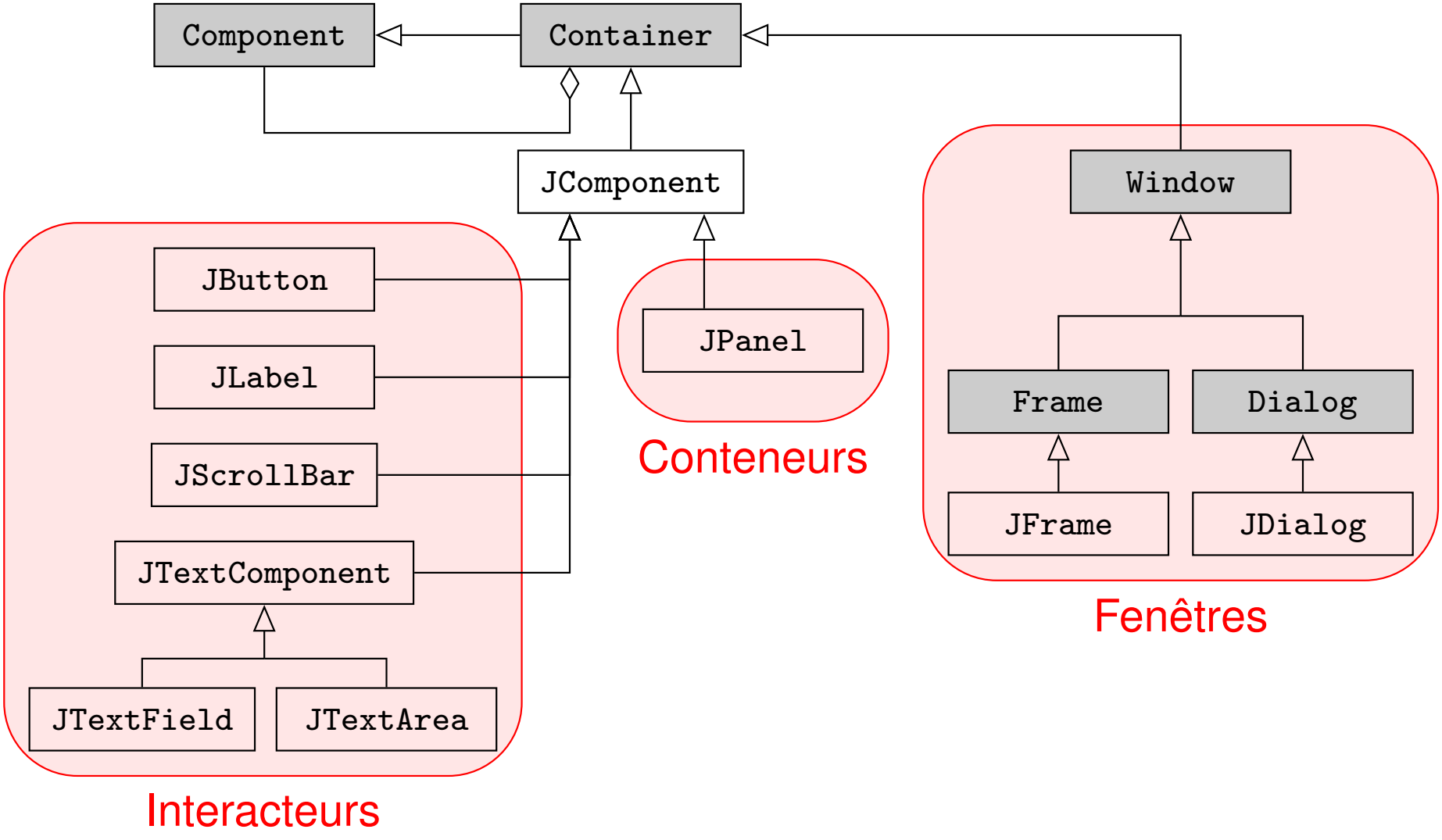


Hiérarchie Swing

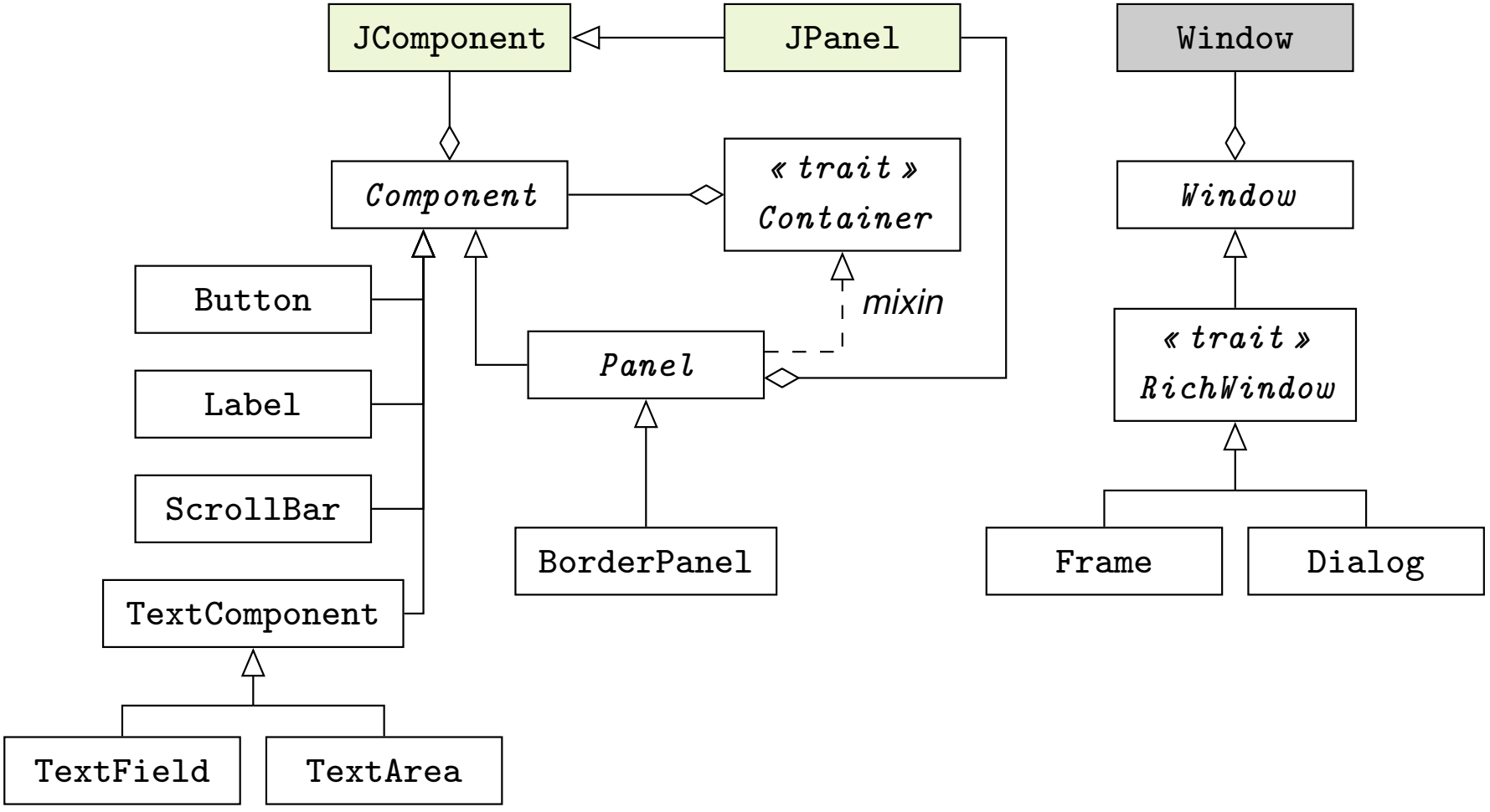


Hiérarchie Swing

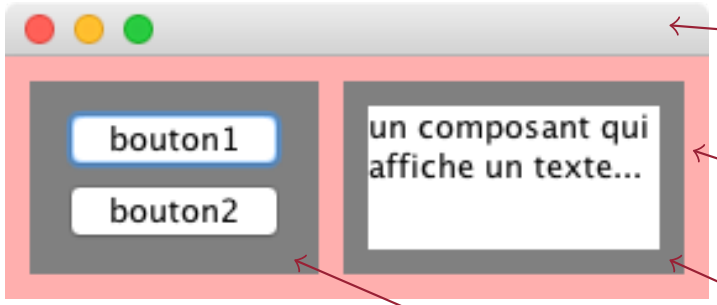



Component
Container
JComponent
JButton
JLabel
JScrollBar
JTextComponent
JTextField
JTextArea
Interacteurs
JPanel
Conteneurs
Window
Frame
Dialog
JFrame
JDialog
Fenêtres

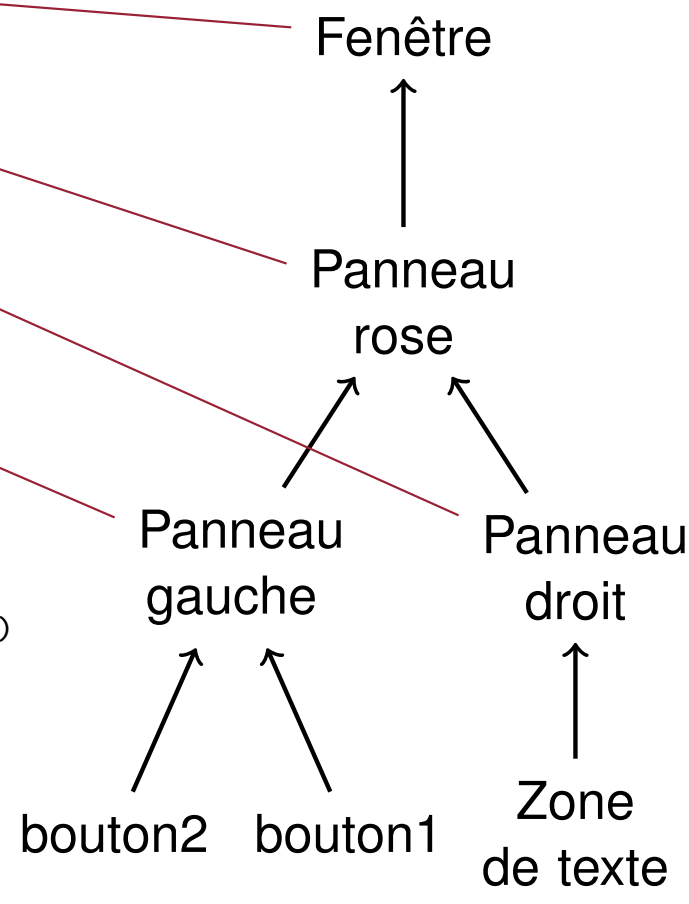
Hiérarchie Swing en scala



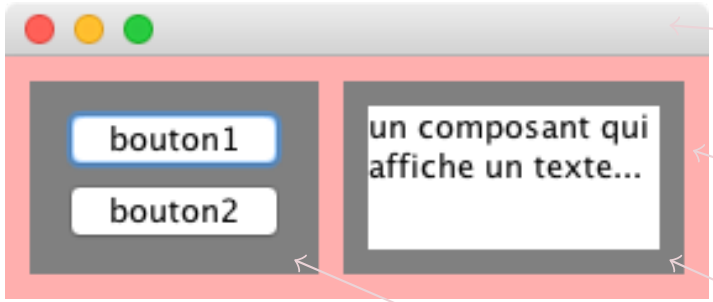
- Application Swing = arbre de composants



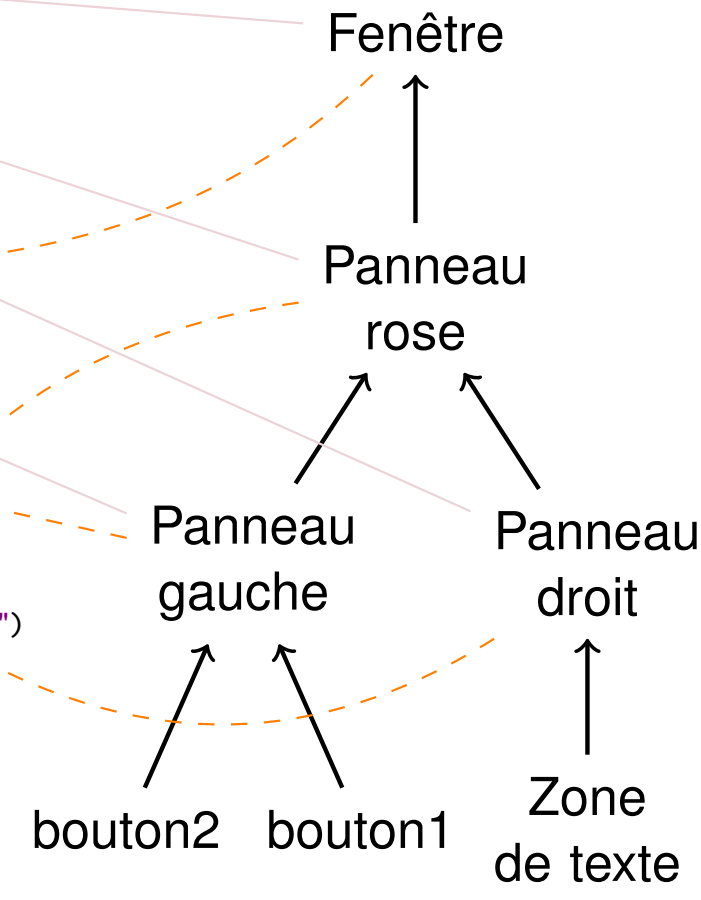

```
object Gui0 extends SimpleSwingApplication {
  def top = new MainFrame {
    val leftPanel = new BoxPanel(Orientation.Vertical) {
      border = Swing.EmptyBorder(10); background = Color.gray
      contents += new Button("button1")
      contents += new Button("button2")
    }
    val rightPanel = new BoxPanel(Orientation.Vertical) {
      border = Swing.EmptyBorder(10); background = Color.gray
      contents += new TextArea("a component that \ndisplays text...")
    }
    contents = new BoxPanel(Orientation.Horizontal) {
      border = Swing.EmptyBorder(10); background = Color.pink
      contents += leftPanel
      contents += Swing.HStrut(10)
      contents += rightPanel
    }
```



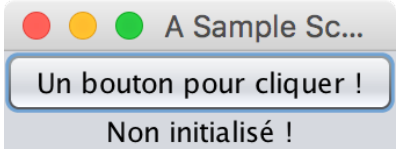
- Application Swing = arbre de composants




```
object Gui0 extends SimpleSwingApplication {
  def top = new MainFrame {
    val leftPanel = new BoxPanel(Orientation.Vertical) {
      border = Swing.EmptyBorder(10); background = Color.gray
      contents += new Button("button1")
      contents += new Button("button2")
    }
    val rightPanel = new BoxPanel(Orientation.Vertical) {
      border = Swing.EmptyBorder(10); background = Color.gray
      contents += new TextArea("a component that \ndisplays text...")
    }
    contents = new BoxPanel(Orientation.Horizontal) {
      border = Swing.EmptyBorder(10); background = Color.pink
      contents += leftPanel
      contents += Swing.HStrut(10)
      contents += rightPanel
    }
```

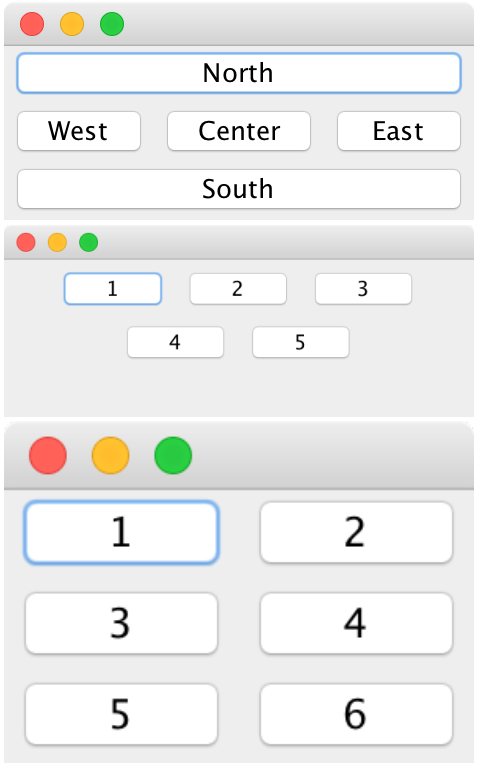


- Conteneur global qui fournit une fenêtre
- Gère des décorations
  - des bords, un titre, des boutons pour fermer, iconifier ..., une barre de menu

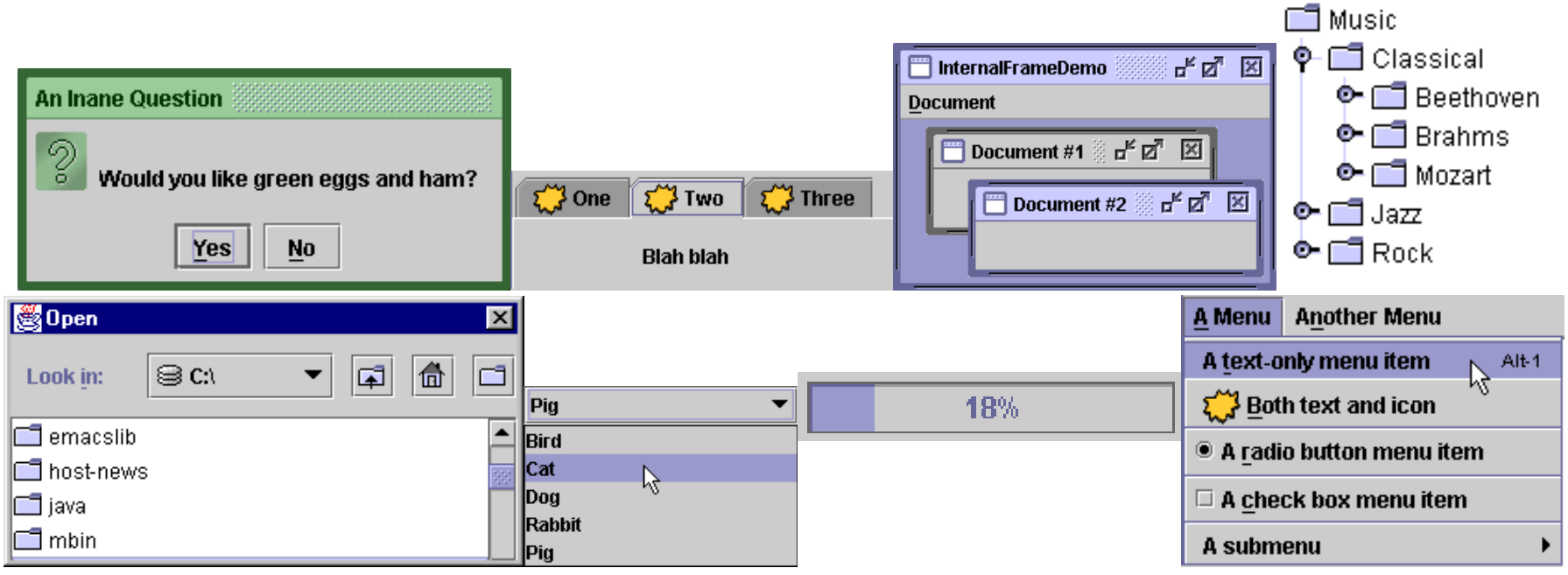



```
object Gui1 extends SimpleSwingApplication {
  def top = new MainFrame {
    UIManager.setLookAndFeel(new NimbusLookAndFeel)
    title = "A Sample Scala Swing GUI"
    val label = new Label("Uninitialized!")
    val button = new Button("A button to click!")
    contents = new BorderPanel {
      layout(button) = Center
      layout(label) = South
    }
  }
}
```

- En scala swing, les panneaux sont typés en fonction du placement
- BorderPanel placement par position
  - `contents =new BorderPanel {`
    `layout(new Button("Center"))=Center}`
- FlowPanel positions séquentielles
  - `contents =new FlowPanel {`
    `contents+=new Button("1")}`
- idem sauf création GridPanel
  - `new GridPanel(3,2)`

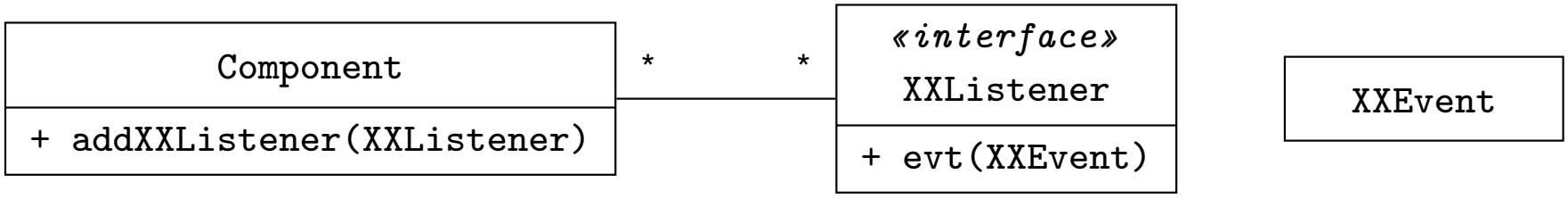


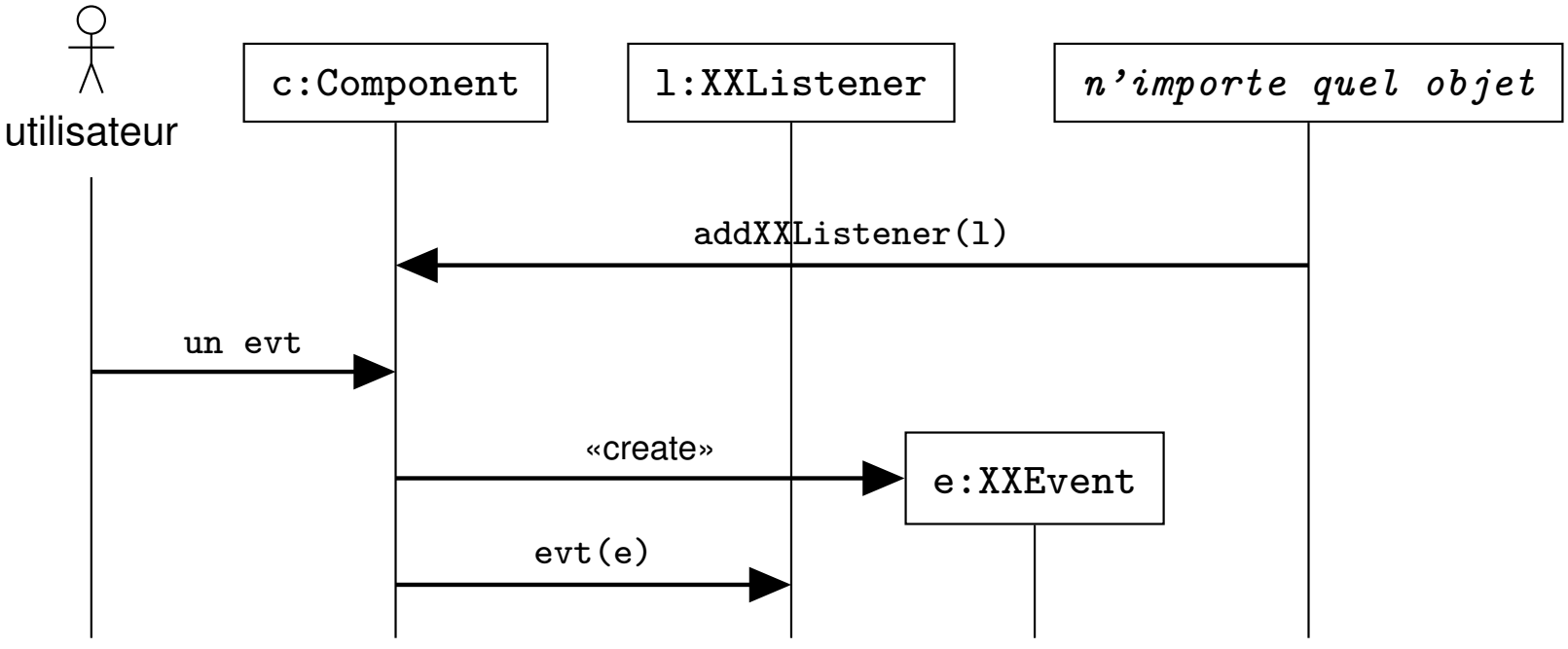
Des composants...

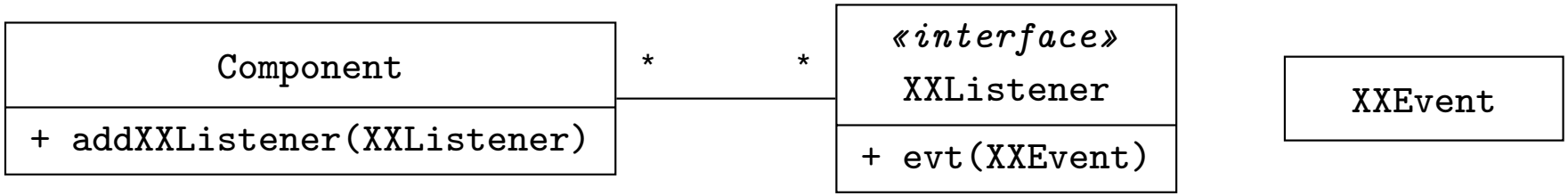


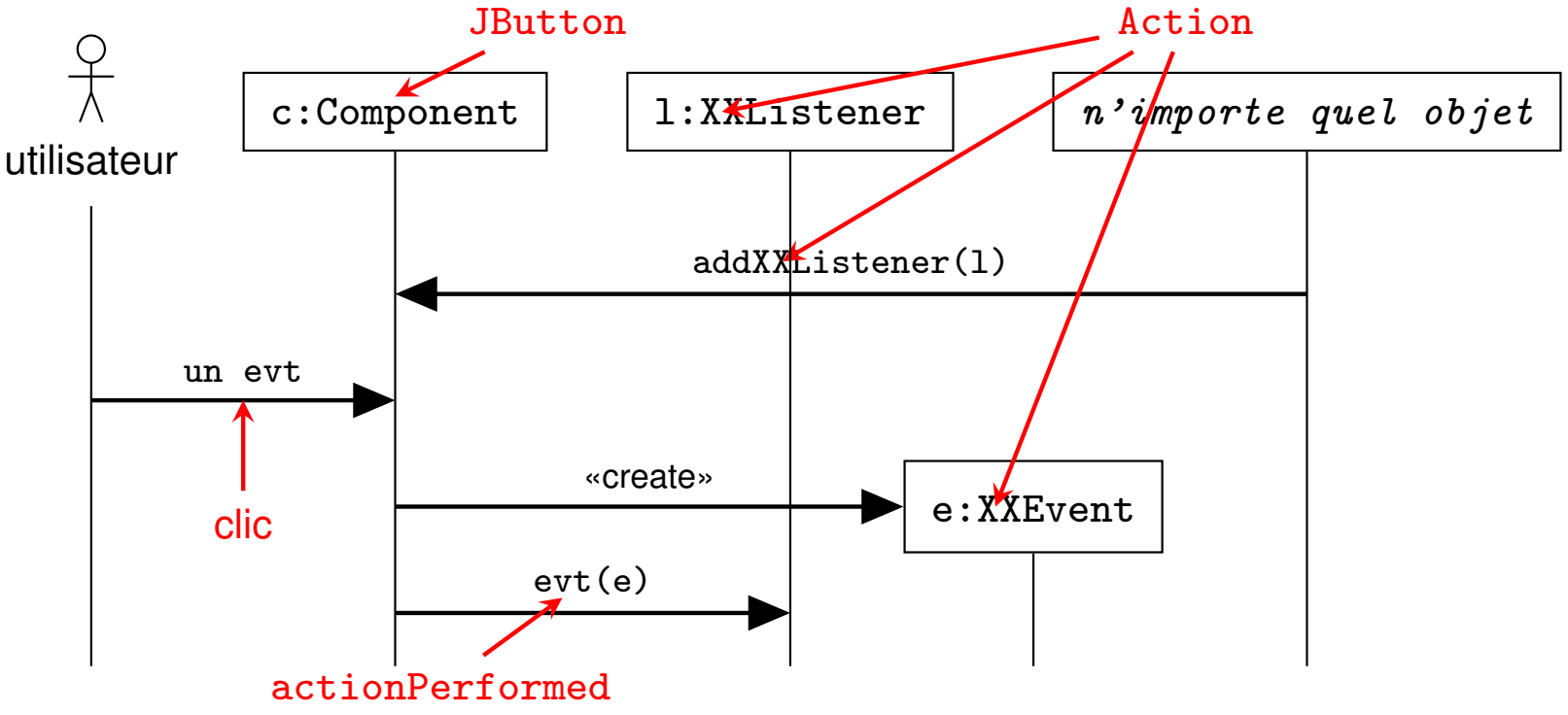
⇒ Consulter les API et le tutorial Oracle

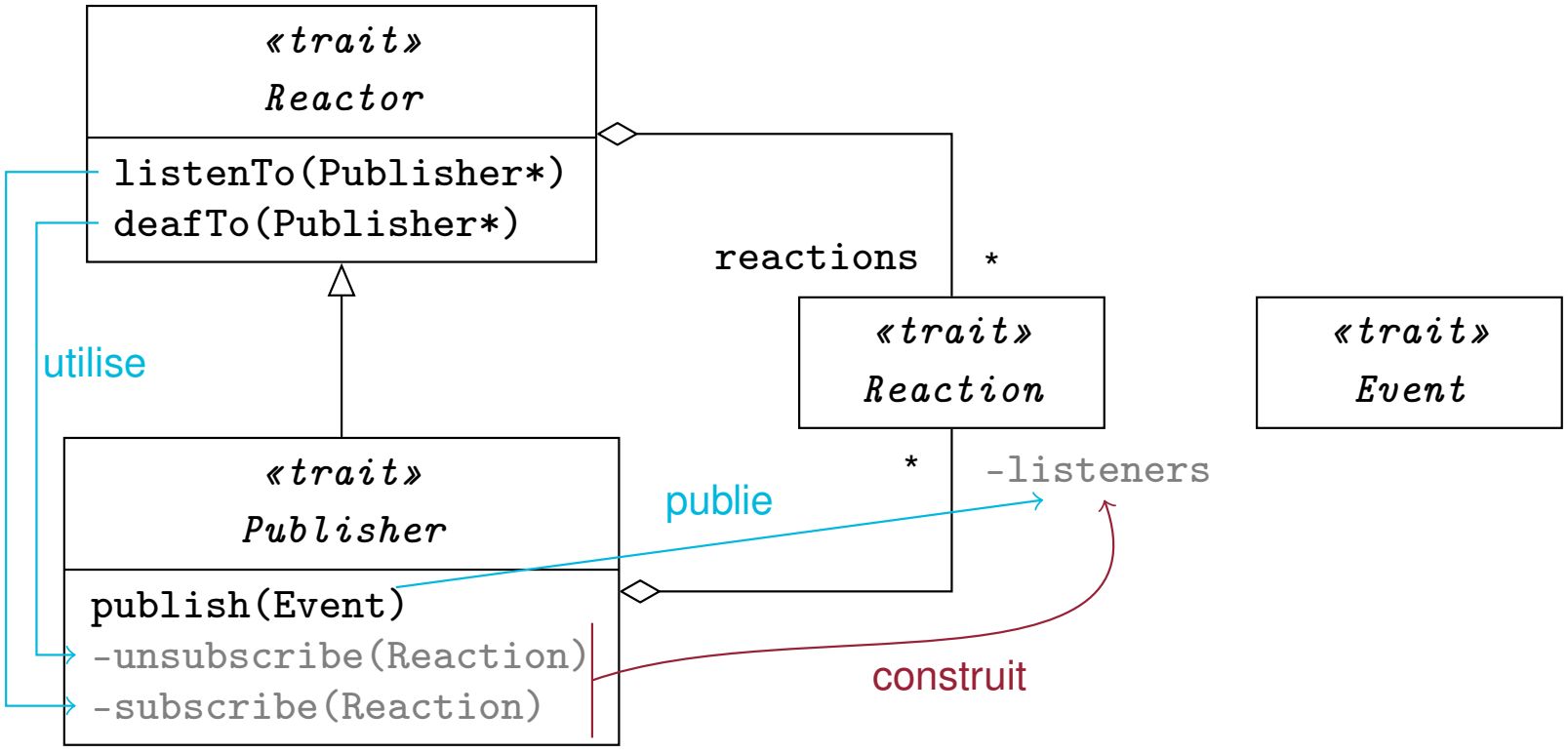
- En général d'AWT (`java.awt.event`) `XXXEvent`
- Des objets (les écouteurs) peuvent s'abonner pour recevoir les événements d'un composant
- Pour cela, doivent réaliser interface `XXXListener`
- `XXX` :
  - Component (taille, position, visibilité)
  - Focus
  - Key (clavier)
  - Mouse (clic, ...), MouseMotion (mouvement dans)
  - Action – seulement certains composant, par ex. bouton, ...
  - ...
  - Des événements définis par l'application


Component
+ addXXListener(XXListener)
*
*
«interface»
XXListener
+ evt(XXEvent)
XXEvent


utilisateur
c:Component
l:XXListener
n'importe quel objet
addXXListener(l)
un evt
«create»
e:XXEvent
evt(e)

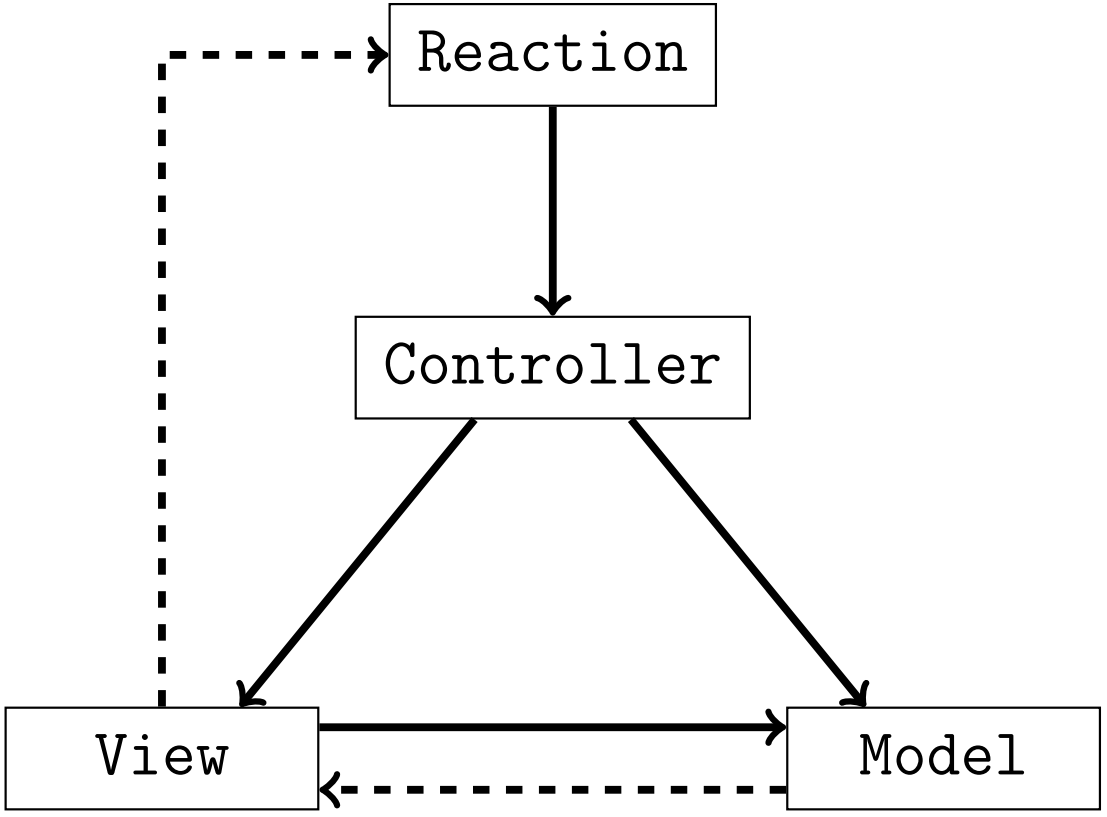

Component
+ addXXListener(XXListener)
*
*
«interface»
XXListener
+ evt(XXEvent)
XXEvent

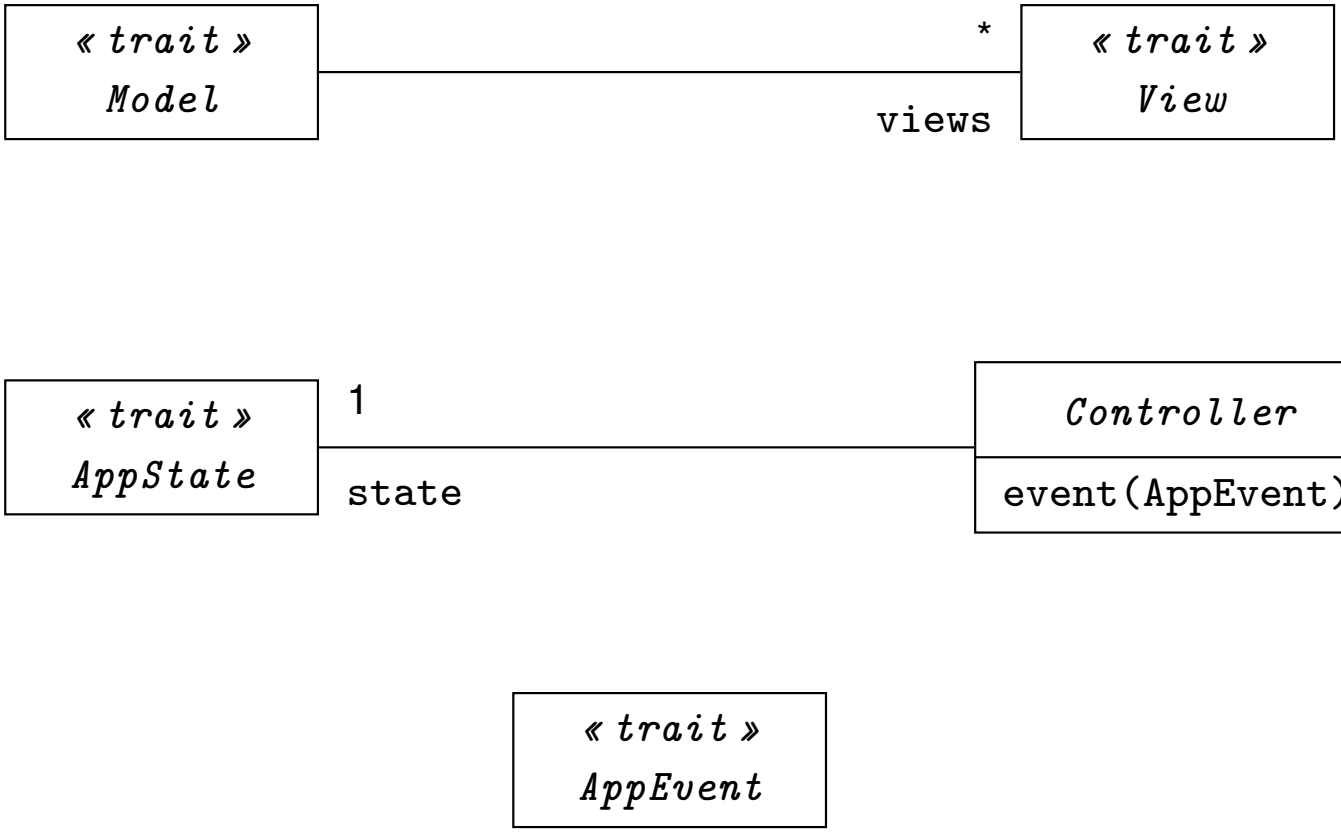

JButton
Action
utilisateur
c:Component
l:XXListener
n'importe quel objet
addXXListener(l)
un evt
clic
«create»
e:XXEvent
evt(e)
actionPerformed



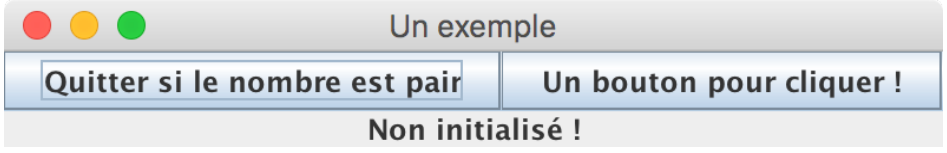
- trait `Publisher` injecté dans les interacteurs
- ajout de réactions puis `listenTo`

```
object Gui1 extends SimpleSwingApplication {
  def top = new MainFrame {
    UIManager.setLookAndFeel(new NimbusLookAndFeel)
    title = "A Sample Scala Swing GUI"
    val label = new Label("Uninitialized!")
    val button = new Button("A button to click!")
    contents = new BorderPanel {
      layout(button) = Center
      layout(label) = South
    }
    listenTo(button)
    var numClicks = 0
    reactions += {
      case ButtonClicked(component) if component == button =>
        numClicks += 1
        label.text = " Click number: " + numClicks + " "
    }
  }
}
```


Reaction
Controller
View
Model

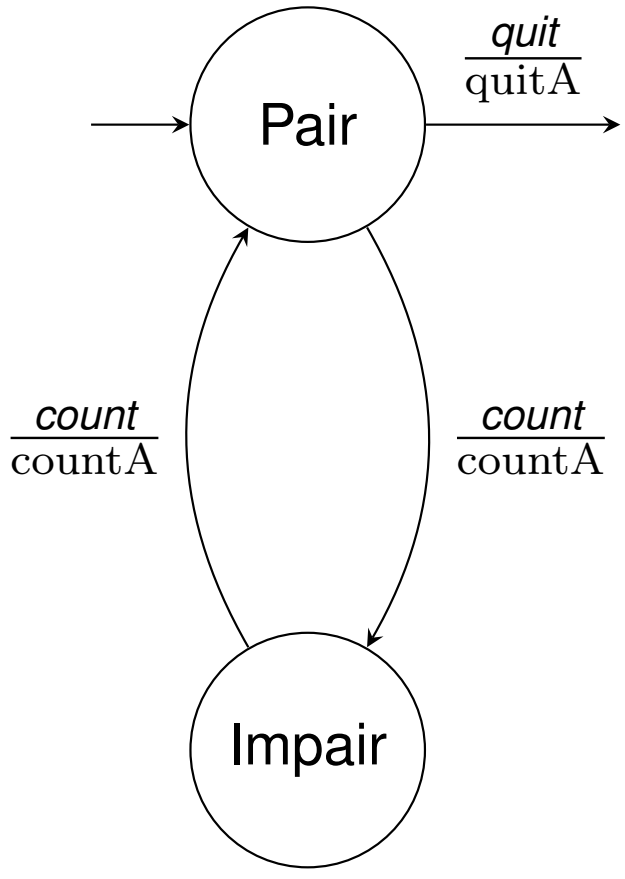

« trait »
Model
*
views
« trait »
View
« trait »
AppState
1
state
Controller
event(AppEvent)
« trait »
AppEvent

- On reprend l'exemple précédent
- On veut ajouter un bouton pour quitter qui quitte si le nombre de clics est pair



```
object Main extends SimpleSwingApplication {
  def top = new MainFrame {
    title = "Un exemple"
    val model = new ClicModel()
    val label = new ClicView(model)
    model.addView(label)
    val controller = new Controller(model)
    val compter = new Button("Un bouton pour cliquer !")
    val quitter = new Button("Quitter si le nombre est pair")
    contents = new BorderPanel {
      layout(compter) = East; layout(quitter) = Center; layout(label) = South
    }
    listenTo(quitter)
    listenTo(compter)
    reactions += {
      case ButtonClicked(c) if c == quitter => controller.event(QUIT)
      case ButtonClicked(c) if c == compter => controller.event(COUNT)
    }
  }
}
```

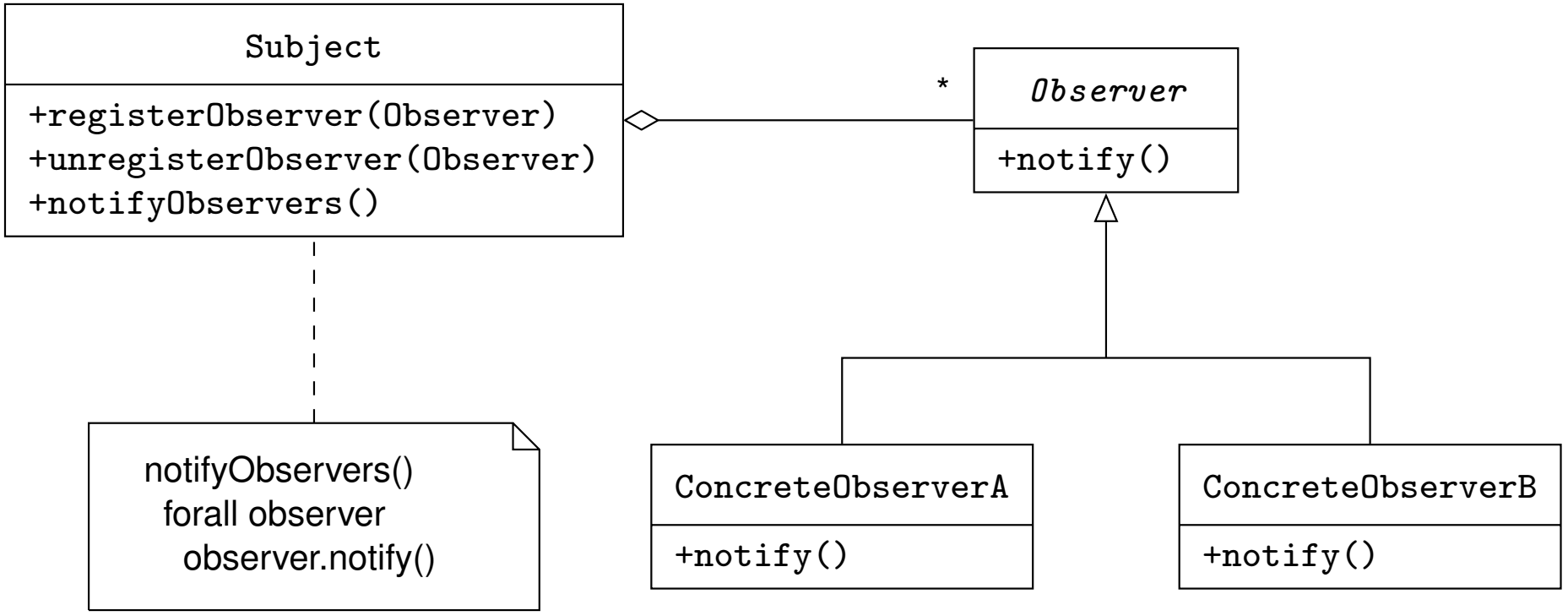
- Données : nombres de clics
- États :
  - Pair : nombre clics pair
  - Impair : nombre clics impair
- Événements :
  - count : clic sur Compter
  - quit : clic sur Quitter
- Actions :
  - quitA : quitter
  - countA : incrémenter le nombre de clics

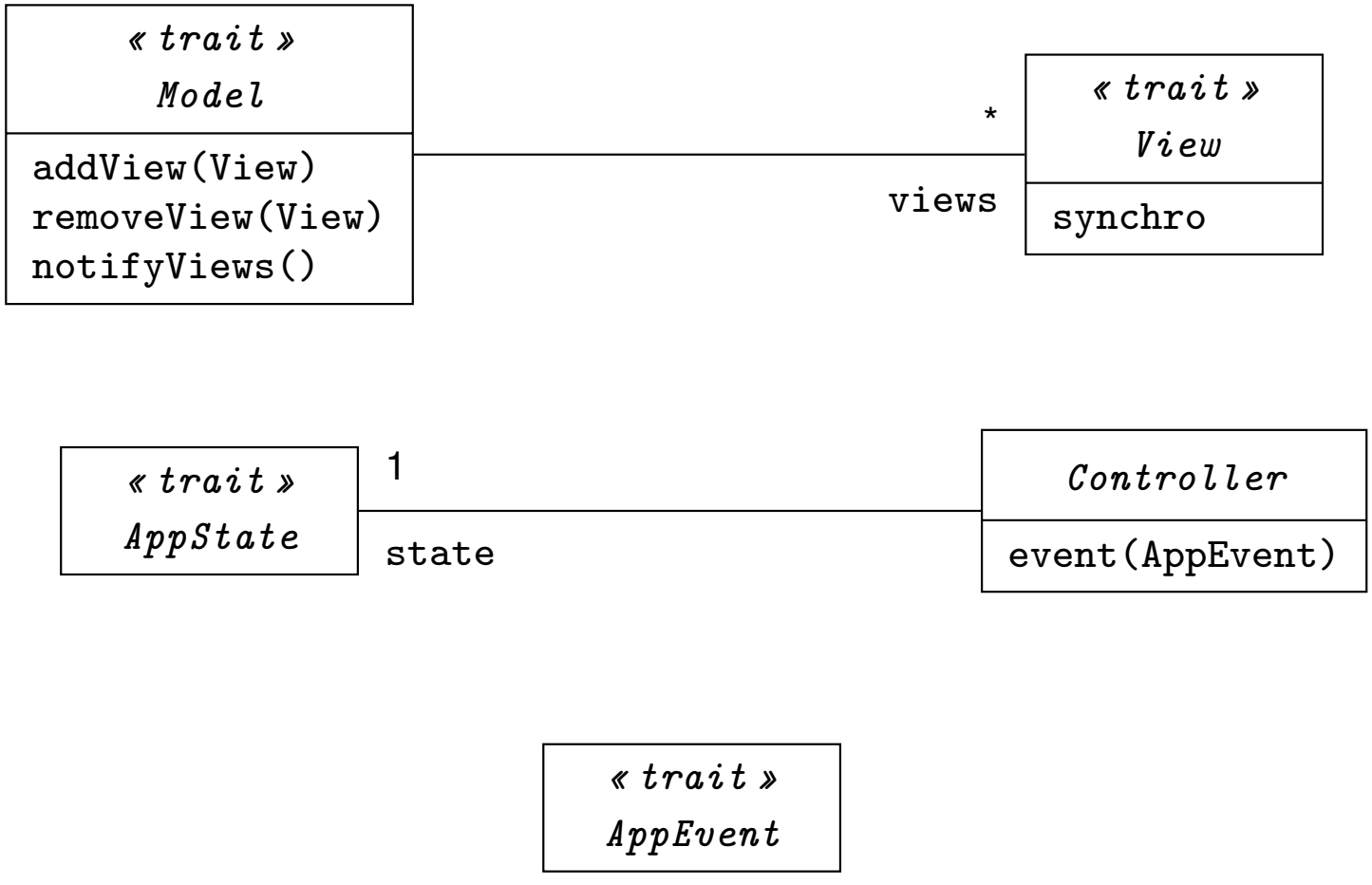


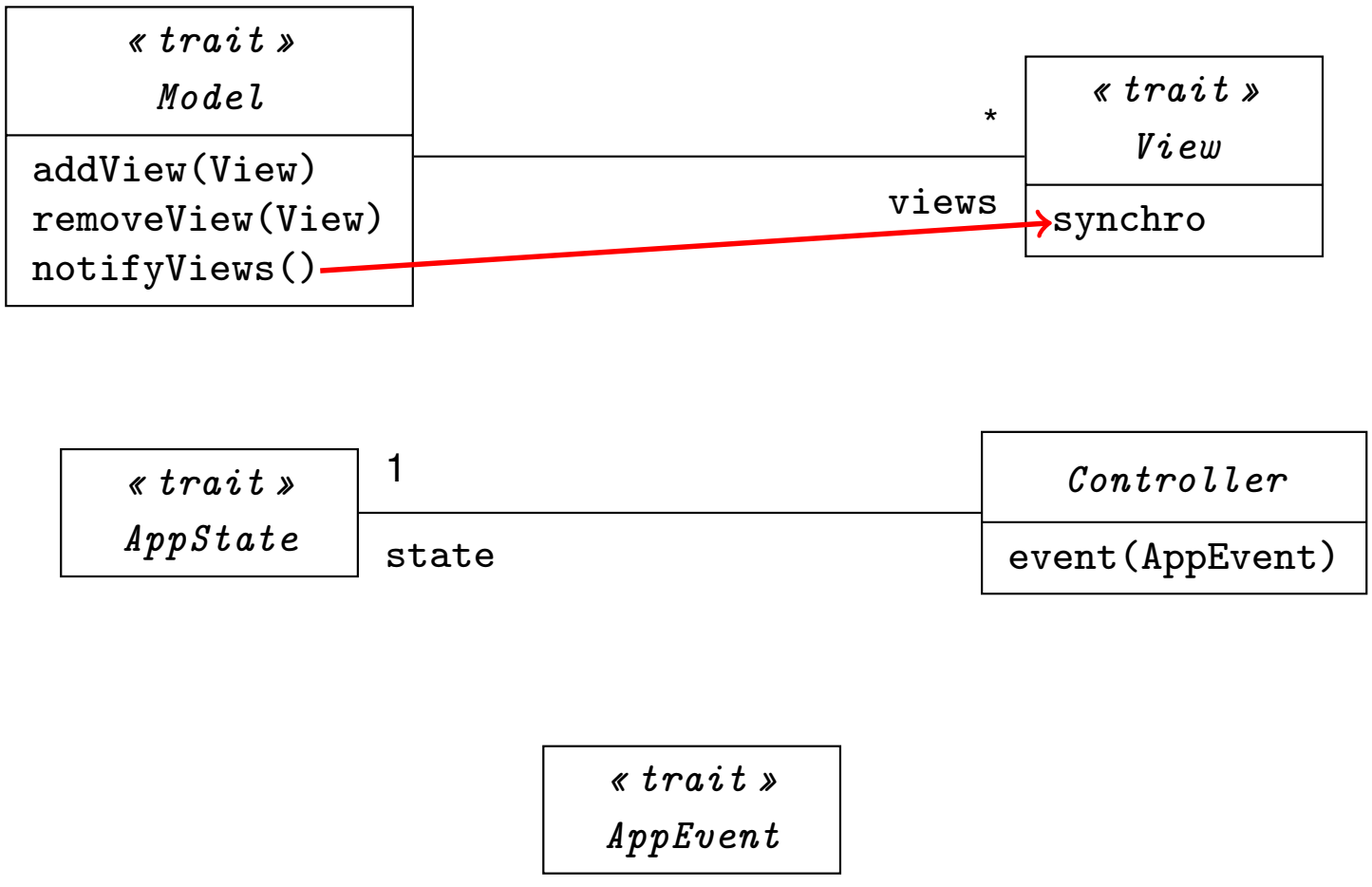
```
class Controller(val controlledModel: ClicModel) extends mvc.Controller(PAIR) {
  // Les actions
  def countAction = controlledModel.incr
  def quitAction = sys.exit(0)
  // Les évenements
  override def event(evt: mvc.AppEvent):Unit = {
    println("event [" + evt + "]")
    evt match {
      case QUIT if (state == PAIR) => quitAction
      case COUNT =>
        state = State.inv(state)
        countAction
      case _ => ()
    }
  }
}
```

- Utilisé pour envoyer un signal à des modules jouant le rôle d'observateur
- En cas de notification, observateurs : action adéquate en fonction des informations disponibles dans « observable »
- Permet de coupler des modules pour réduire les dépendances aux seuls phénomènes observés

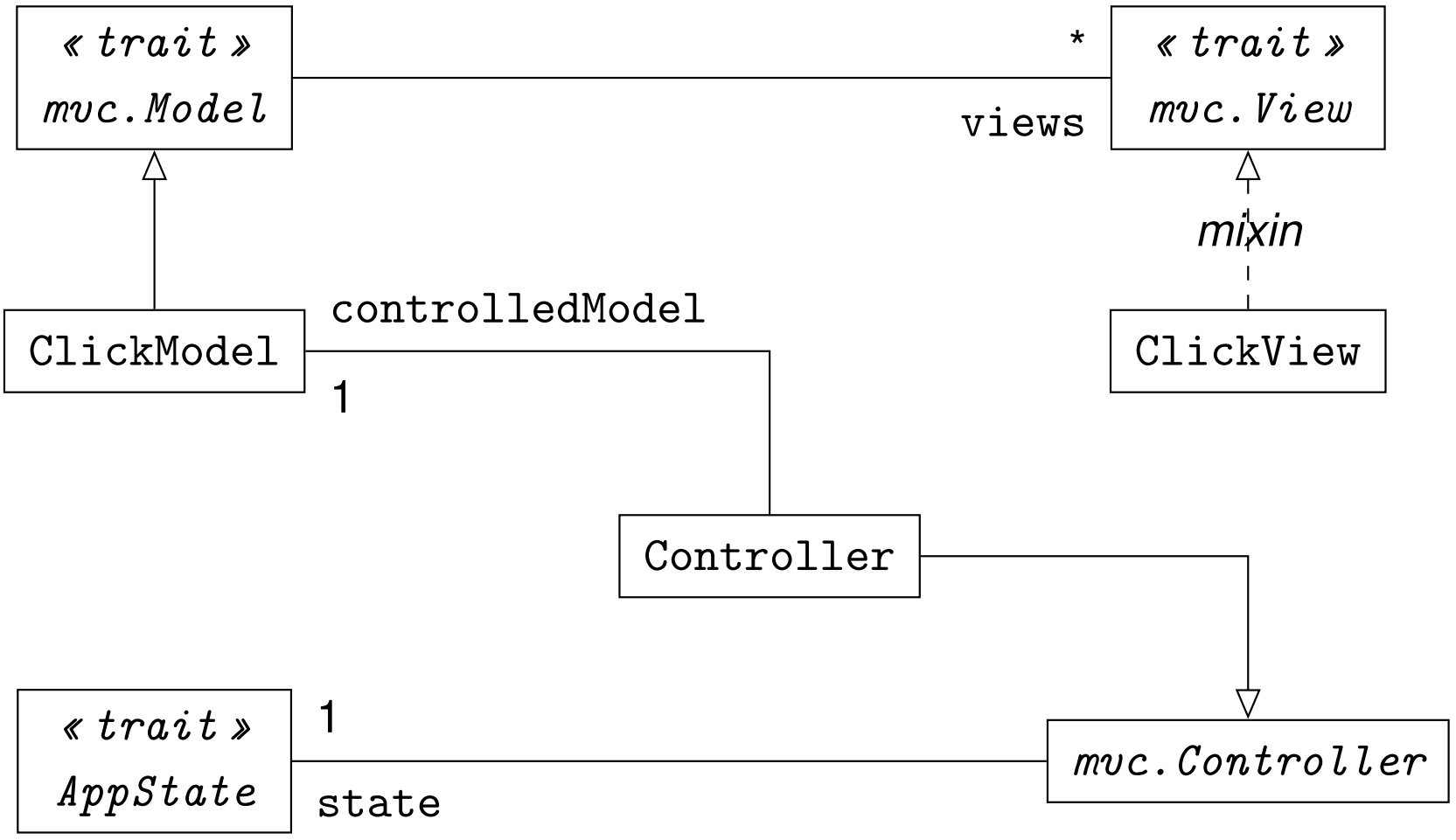
Pattern observateur/observable (suite)

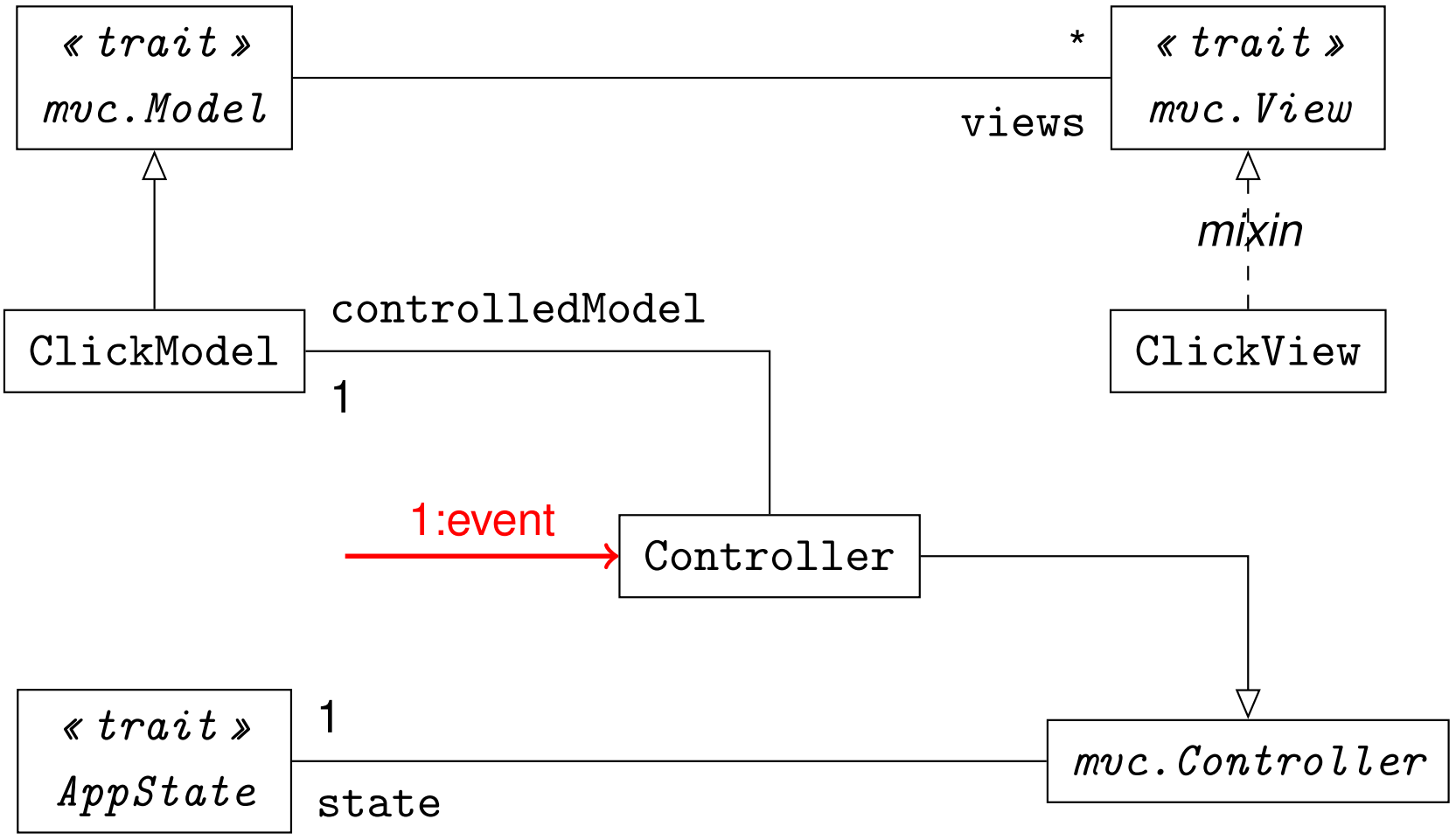


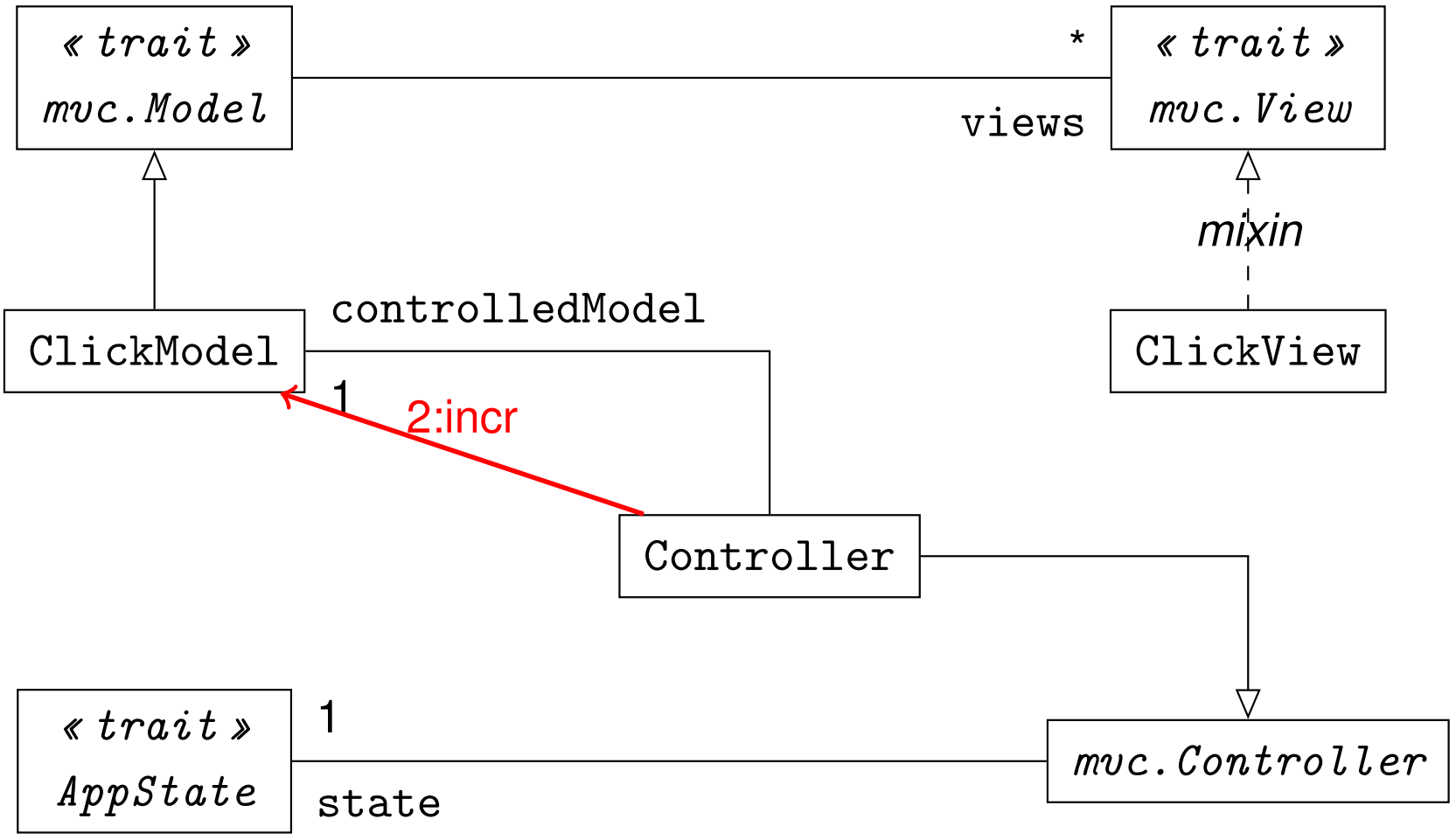

« trait »
Model
addView(View)
removeView(View)
notifyViews()
*
views
« trait »
View
synchro
« trait »
AppState
1
state
Controller
event(AppEvent)
« trait »
AppEvent

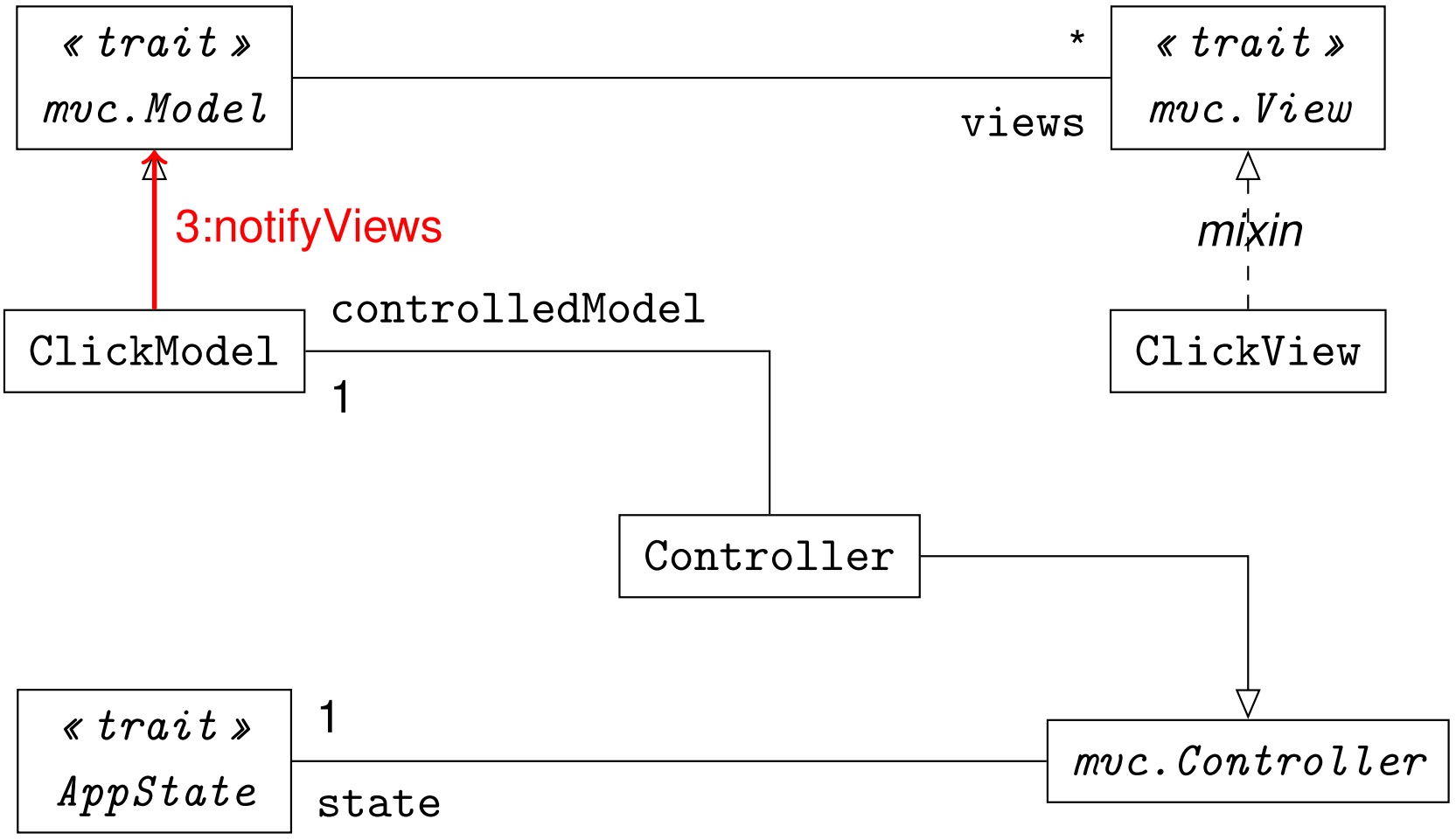

« trait »
Model
addView(View)
removeView(View)
notifyViews()
*
views
« trait »
View
synchro
« trait »
AppState
1
state
Controller
event(AppEvent)
« trait »
AppEvent

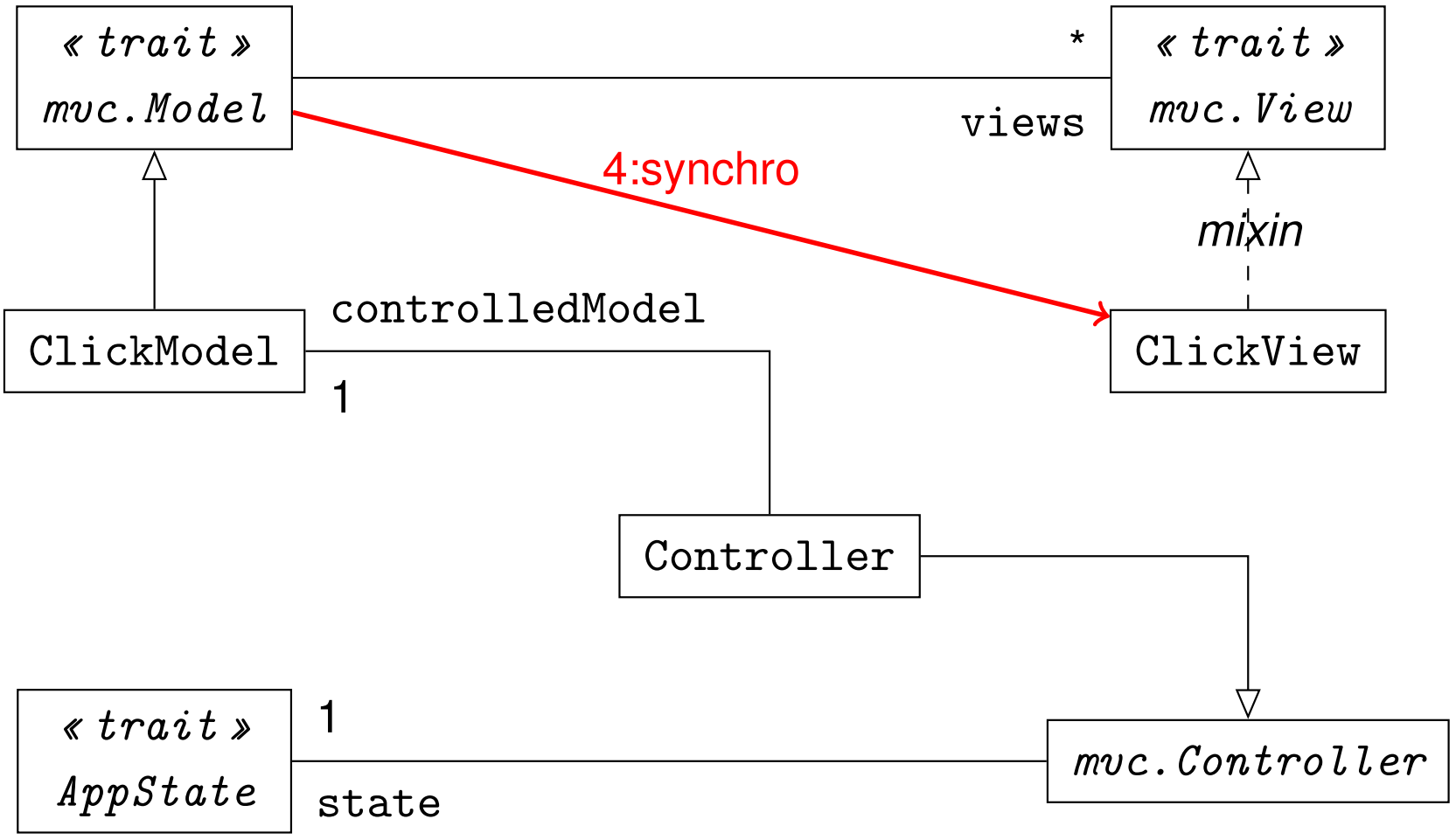
```
class ClicView(val model:ClicModel) extends Label("Non initialisé !") with mvc.View {
  def synchro = text = " Nombre de clics : " + model.numClics + " "
}

class ClicModel(var numClics:Int = 0) extends mvc.Model {
  def incr = {
    numClics += 1
    this.notifyViews
  }
}
```

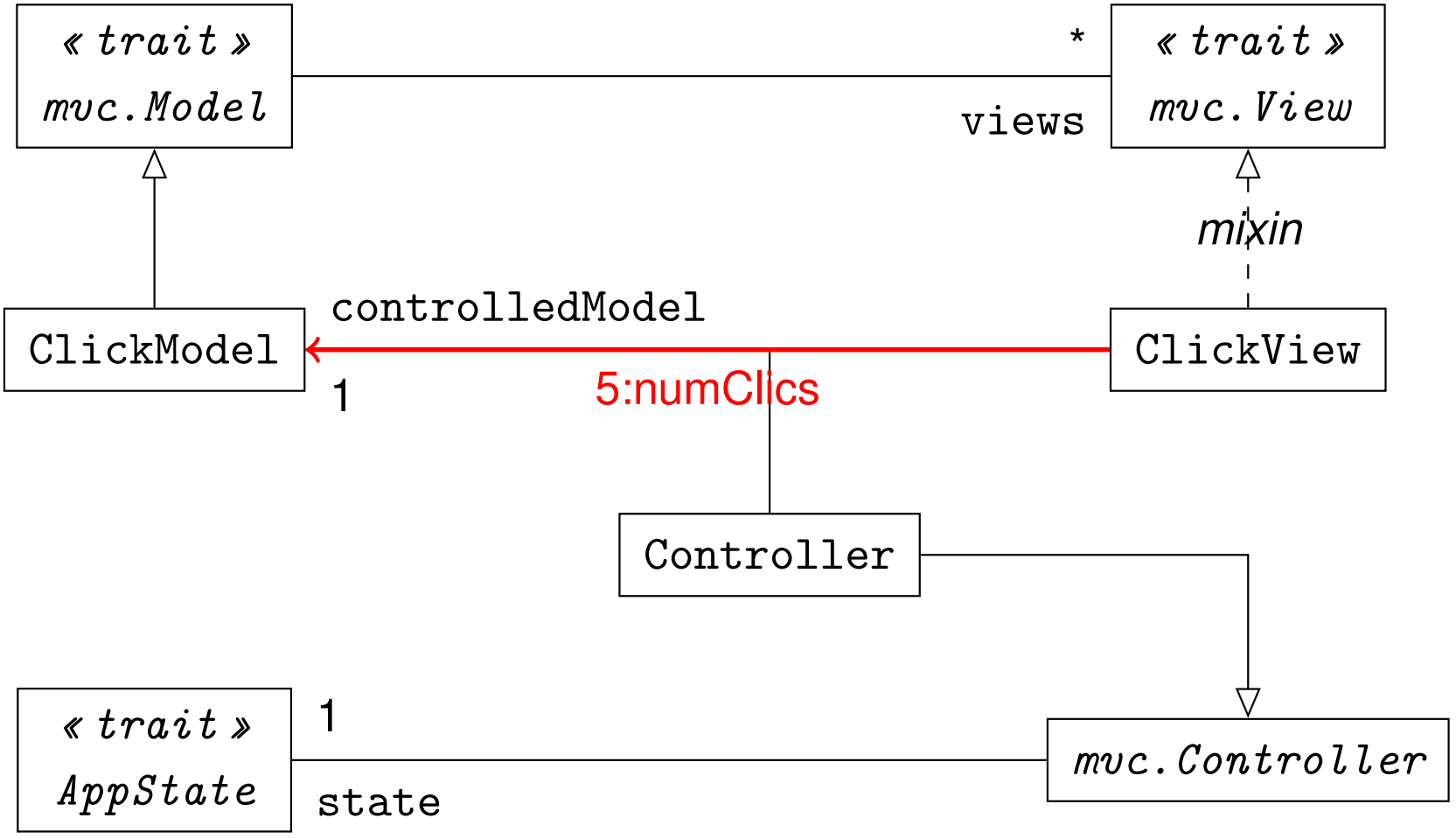

« trait »
mvc.Model
*
views
« trait »
mvc.View
mixin
ClickModel
controlledModel
1
ClickView
Controller
« trait »
AppState
1
state
mvc.Controller


« trait »
mvc.Model
*
views
« trait »
mvc.View
mixin
ClickModel
controlledModel
1
ClickView
1:event
Controller
« trait »
AppState
1
state
mvc.Controller


« trait »
mvc.Model
*
views
« trait »
mvc.View
mixin
ClickModel
controlledModel
1
2:incr
ClickView
Controller
« trait »
AppState
1
state
mvc.Controller


« trait »
mvc.Model
*
views
« trait »
mvc.View
3:notifyViews
mixin
controlledModel
ClickModel
1
ClickView
Controller
« trait »
AppState
1
state
mvc.Controller


« trait »
mvc.Model
*
views
« trait »
mvc.View
4:synchro
mixin
ClickModel
controlledModel
1
ClickView
Controller
« trait »
AppState
1
state
mvc.Controller


« trait »
mvc.Model
*
views
« trait »
mvc.View
mixin
ClickModel
controlledModel
1
5:numClics
ClickView
Controller
« trait »
AppState
1
state
mvc.Controller

- Cours très partiel centré sur les bases et une méthode
- N'hésitez pas à lire
- Programmation qui devient très complexe dès que l'interface devient importante (en taille)
- Ne jamais négliger la phase de conception dans la partie IHM

1. Concevoir la vue statique (le *look*) :
   - 1.1 modèle de utilisateurs
   - 1.2 hiérarchie des tâches
   - 1.3 réalisation de la présentation
   - 1.4 validation par les futurs utilisateurs
2. Concevoir la partie dynamique :
   - 2.1 modélisation du dialogue (données, états, événements, actions)
   - 2.2 automate de contrôle
   - 2.3 réalisation du contrôleur et de sa représentation
   - 2.4 ajout des réactions à la partie statique