R3.04 : Qualité de développement

Patrons de conception – suite

Arnaud Lanoix Brauer
Arnaud.Lanoix@univ-nantes.fr

Département informatique

Sommaire

IUT Nantes

Patron Stratégie – Strategy Patterns

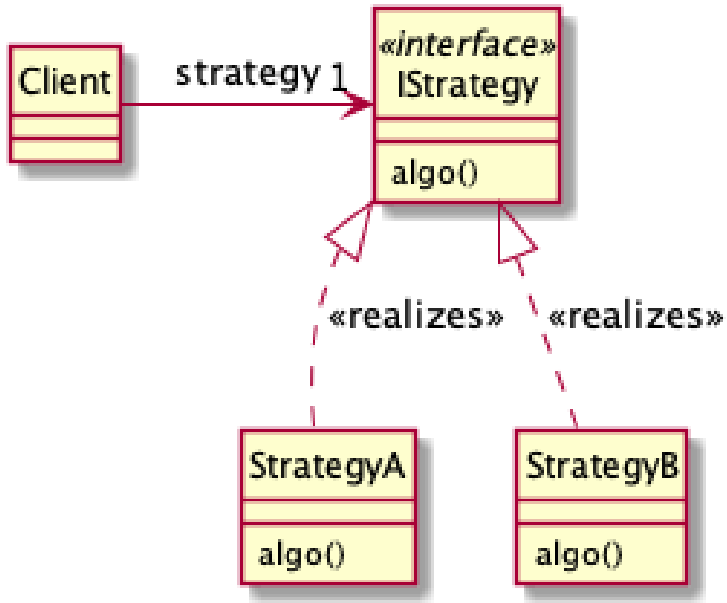
Patron Stratégie – Strategy Patterns

Problème

On a plusieurs algorithmes similaires ; comment changer facilement l'algorithme utilisé ?

Solution

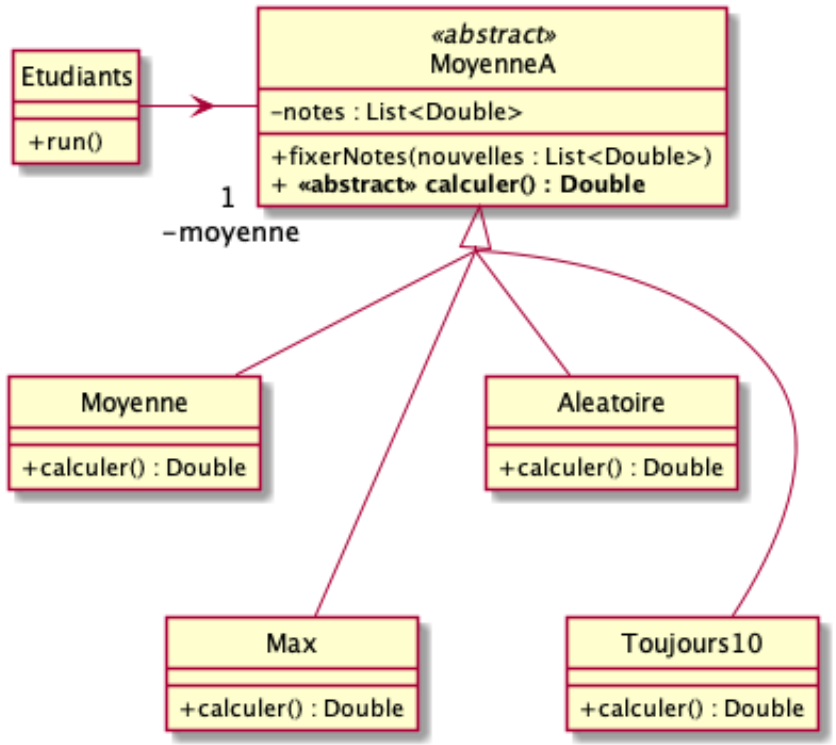
- Définir la famille d'algorithmes grâce à une interface/classe abstraite
- Définir chaque algo par une classe implémentant l'interface
- Le client utilise l'un des algorithmes à un moment donné via un attribut référençant la stratégie



- On peut facilement changer la classe concrète utilisée et donc l'algorithme
- proche du design pattern template method

IUT Nantes

Exemple de Stratégie




```
abstract class MoyenneA {
  protected lateinit var notes : List<Double>

  fun fixerNotes(nouvelles : List<Double>) {
      notes = nouvelles
  }

  abstract fun calculer() : Double
```

```
class Toujours10() : MoyenneA()  {
  override fun calculer() = 10.0
```

```
class Aleatoire() : MoyenneA() {
  override fun calculer() = notes.random()
```

```
class Max() : MoyenneA() {
  override fun calculer() = notes.maxOf {it}
```

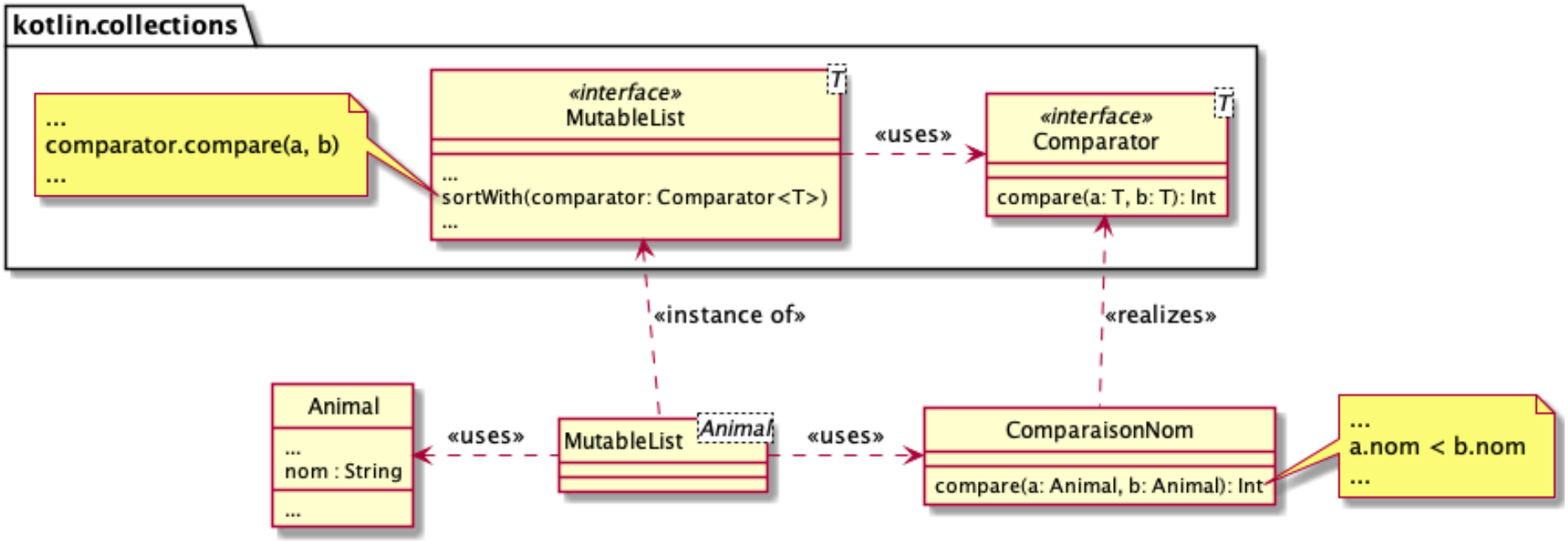
```
class Moyenne() : MoyenneA() {
  override fun calculer() = notes.average()
```

```
when (enseignants.random()) {
    "AL" -> moyenne = Toujours10()
    "JFB" -> moyenne = Aleatoire()
    "JFR" -> moyenne = Max()
    "JMM" -> moyenne = Moyenne()
}
for (notes in etudiants) {
    moyenne.fixerNotes(notes)
    println(" ${moyenne.calculer()} ")
}
```

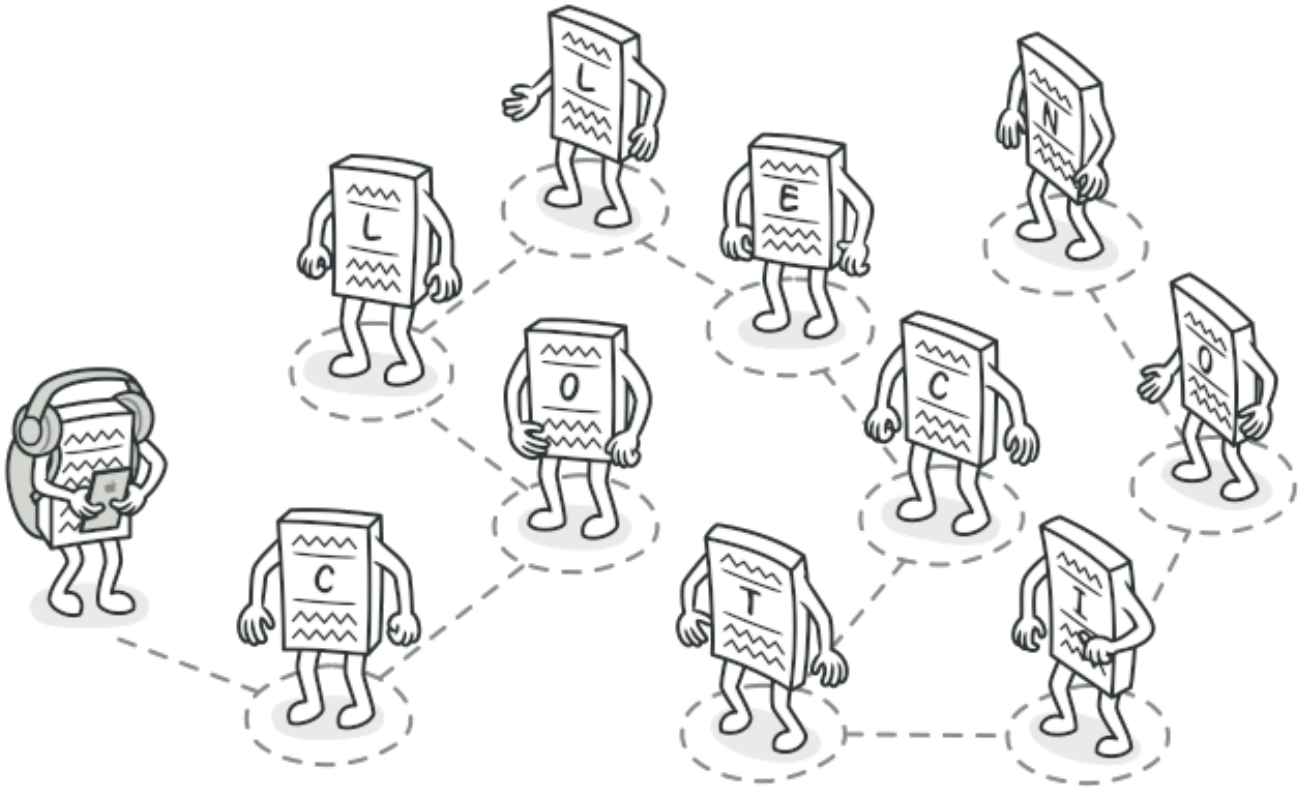
IUT Nantes

L'interface Kotlin `Comparator<T>`

`Comparator<T>` propose de définir une stratégie d'ordonnancement des éléments `T`.
Cette stratégie est ensuite utilisée, par exemple pour trier une liste :
MutableList<T>.sortWith(comparator : Comparator<T>)



Patron Itérateur – Iterator Pattern



Patron Itérateur – Iterator Pattern

Problème

Comment parcourir séquentiellement un objet conteneur (liste, arbre, etc.) sans en dévoiler la structure interne (potentiellement complexe) ?

Solution

Le conteneur propose un itérateur qui va permettre un parcours séquentiel, c-à-d

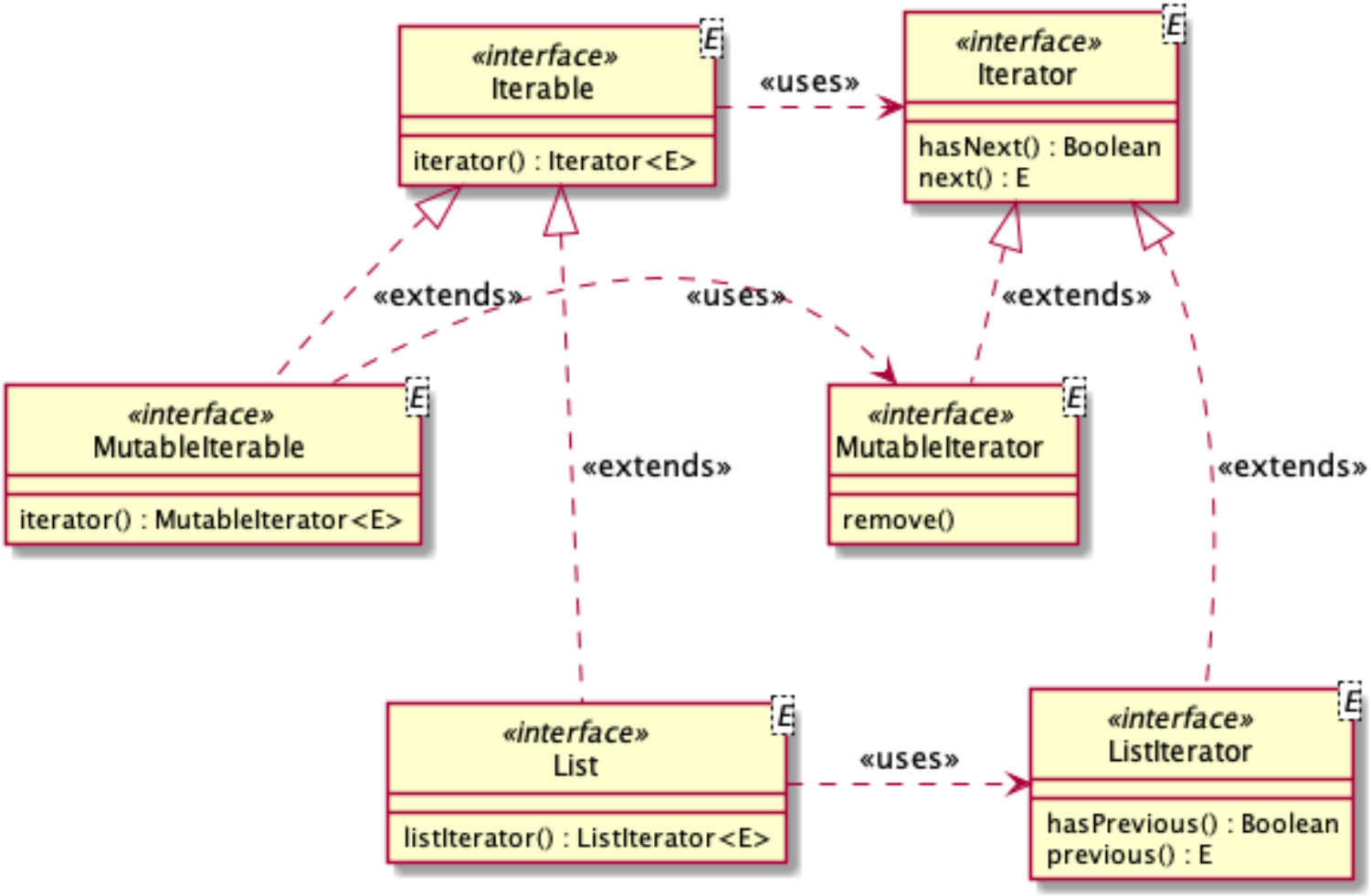
- d'accéder à l'élément courant
- de passer à l'élément "suivant"
- de déterminer si on a tout parcouru

- Un itérateur permet l'accès séquentiel de structures non-indexées/non-ordonnées : ensembles, arbres, etc.
- Certains itérateurs proposent de modifier le conteneur pendant le parcours, d'itérer dans /alertl'autre sens

IUT Nantes

L'interface `Iterable<E>` en Kotlin

En Kotlin, toutes les collections standard de `kotlin.collections` héritent de `Iterable<E>`



IUT Nantes

Utilisation d'un itérateur en Kotlin

```
val teachers = mutableListOf<String>("Arnaud", "JFB",
                                     "JFR", "JM", "Olivier")
val ite = teachers.iterator()
while(ite.hasNext()) {
        val teacher = ite.next()
        print("$teacher ")
}
```

est équivalent à

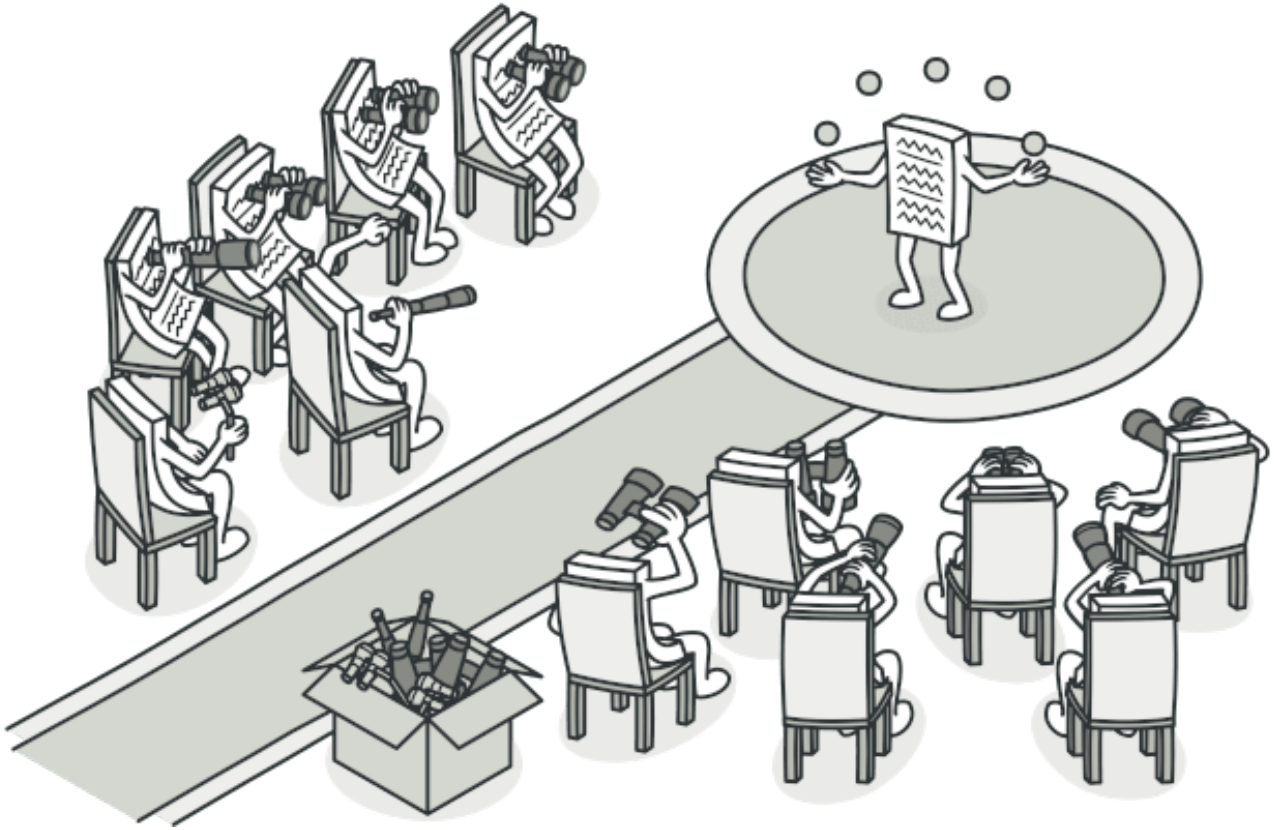
```
for (teacher in teachers) {
  print("$teacher ")
}
```

On peut modifier la collection à travers l'itérateur :

```
while(ite.hasNext()) {
  val teacher = ite.next()
  if (teacher == "JFB")
          ite.remove()
}
```

IUT Nantes

Patron Observateur – Observer pattern



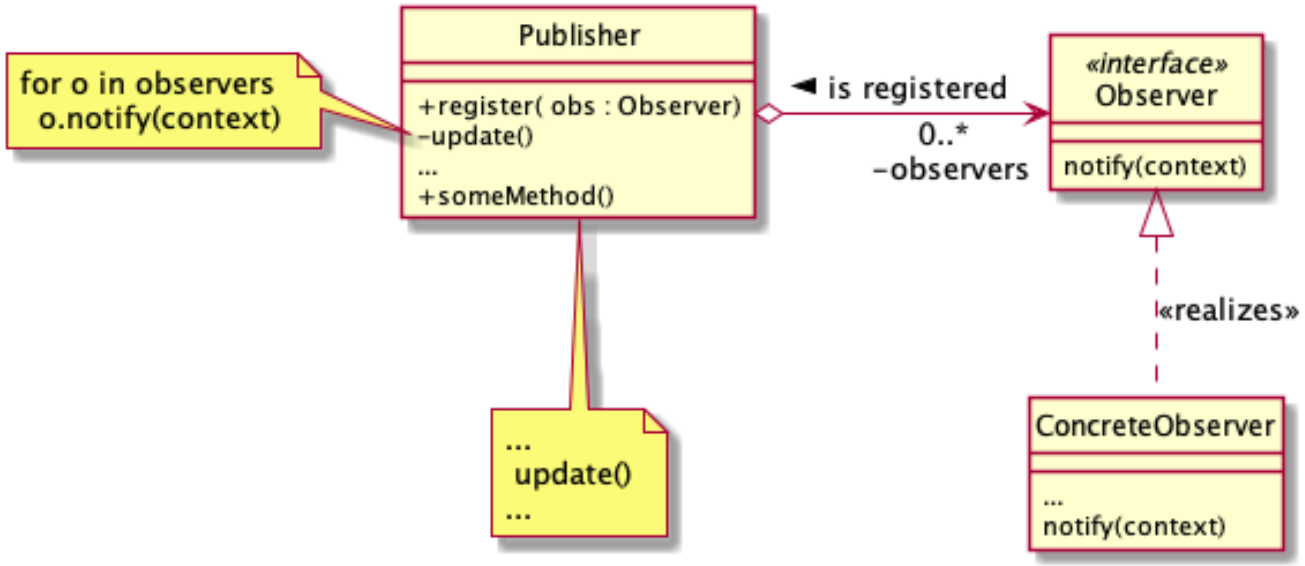
Patron Observateur – Observer pattern

Problème

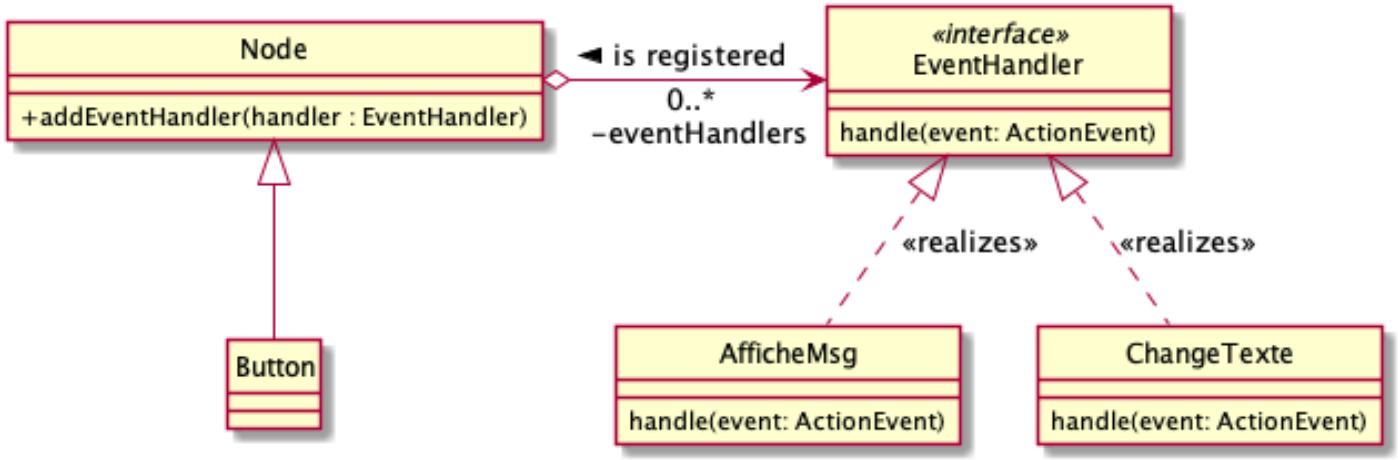
Informer automatiquement d'un changement d'état d'un objet donné (= l'observable), un ou plusieurs autres objets (= les observateurs)

Solution

- Les observateurs réalisent tous une certaine interface
- L'objet observable possède une liste d'observateurs
- A chaque changement, il notifie tous ses observateurs



Exemple d'observateurs : les événements JavaFX



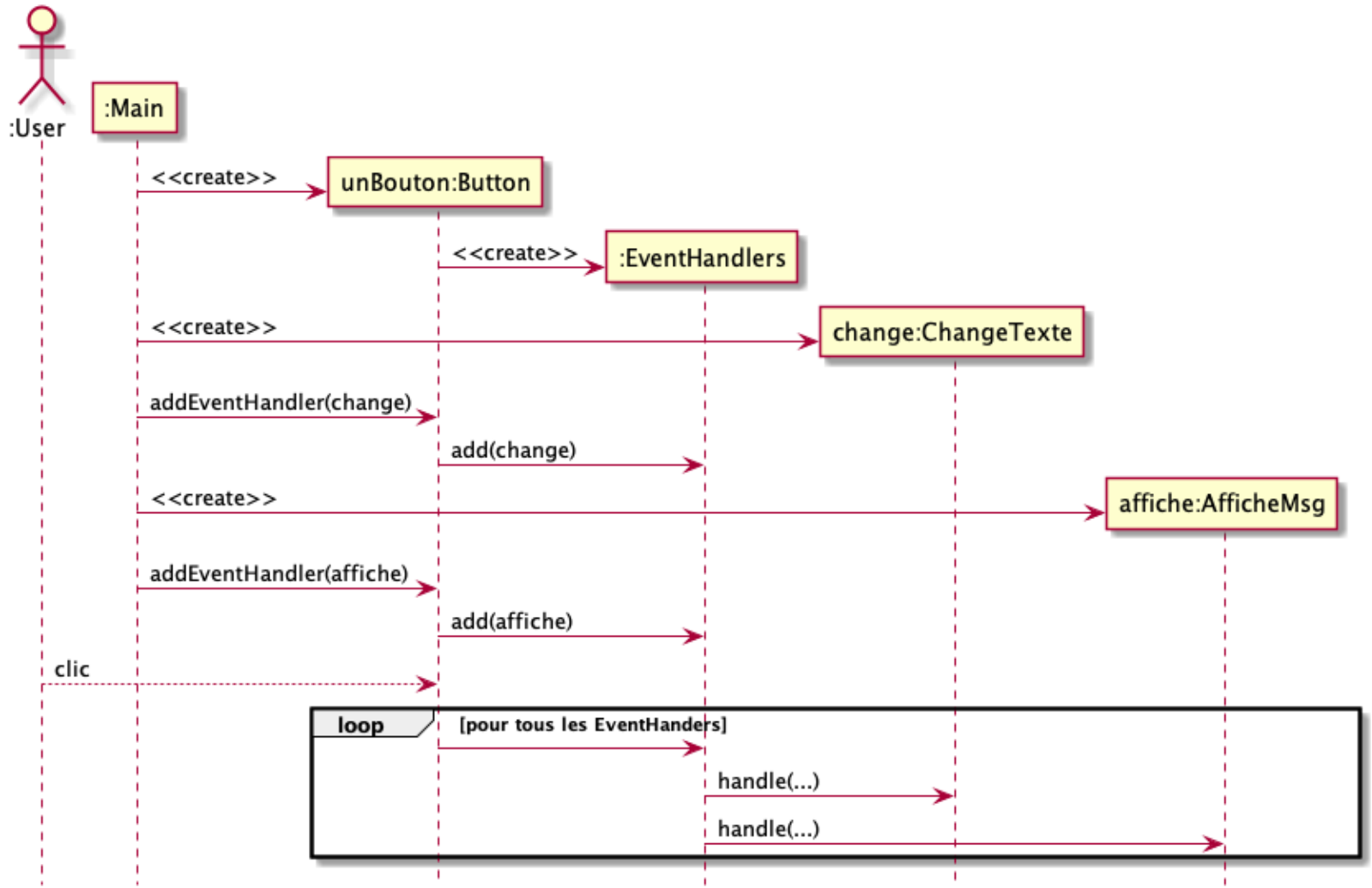

```
...
val unBouton = Button("Go")
val texte = TextField()
unBouton.addEventHandler(ActionEvent.ACTION,
                         ChangeTexte(texte))
unBouton.addEventHandler(ActionEvent.ACTION,
                          AfficheMsg())
...
```

```
class ChangeTexte(val texte : TextField)
            : EventHandler<ActionEvent> {
  override fun handle(event: ActionEvent?) {
      texte.text = "OK"
  }
```

```
class AfficheMsg : EventHandler<ActionEvent> {
  override fun handle(event: ActionEvent?) {
      println("Bouton clique")
  }
```

IUT Nantes

Exemple d'observateurs : les événements JavaFX (2)

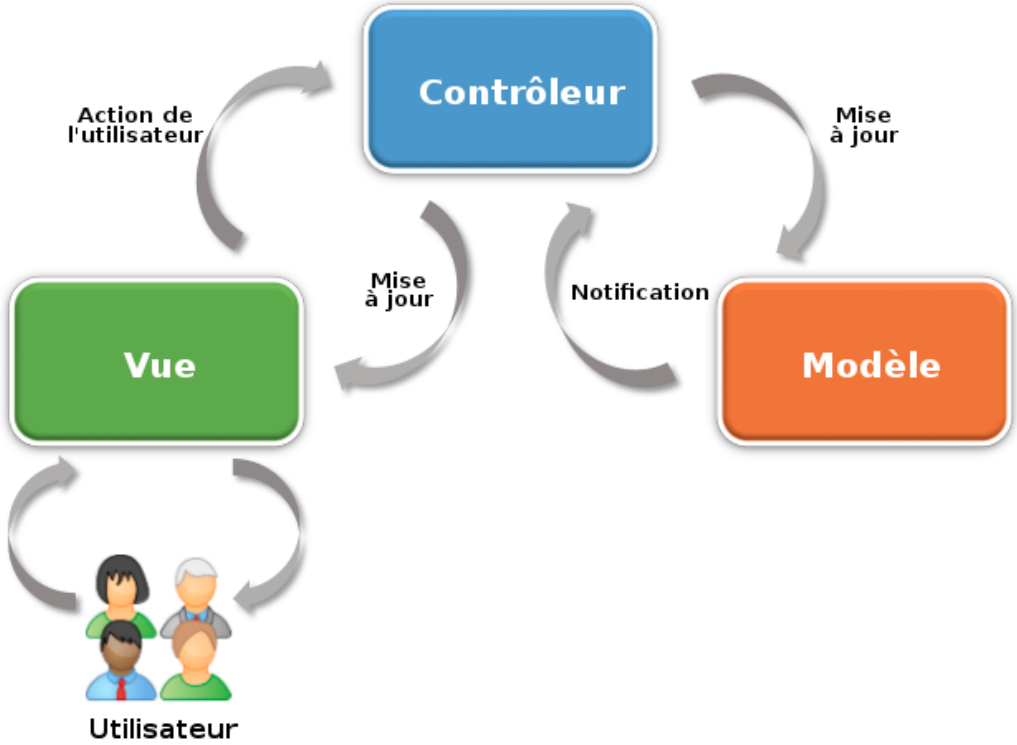


IUT Nantes

Patron d'architecture Modèle-Vue-Contrôleur

Le patron MVC est un patron architectural pour les applications graphiques proposant de séparer les "préoccupations" :

1. Modèle = données à afficher + règles métier
2. Vue = interface graphique de présentation des données
3. Contrôleur = logique concernant les actions des utilisateurs ; modifie le modèle et la vue



- MVC combine les patrons de conception Observateur, Stratégie et Composite
- Variantes : modèle-vue-vue modèle (MVVM), modèle-vue-présentation (MVP)

IUT Nantes