R3.04 : Qualité de développement

Patrons de conception

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

IUT Nantes
Pôle Sciences et technologie

Nantes Université

Département informatique

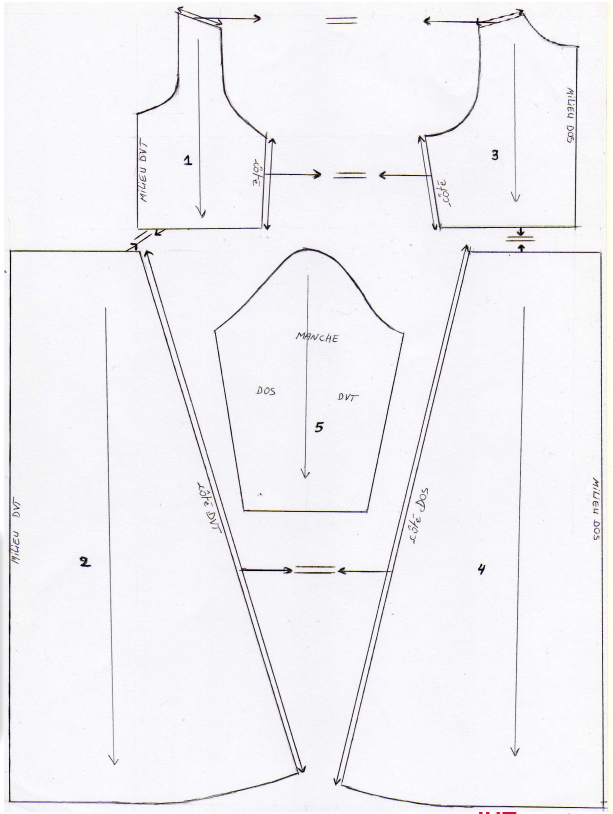
Sommaire

IUT Nantes

Patrons de conception

Un patron de conception (= Design Pattern) est une (la meilleure) *solution de conception* à un problème récurrent, qui pourra être réutilisée et adaptée indéfiniment

- Analogie = les patrons en couture : modèle/plan à réutiliser/adapter
- formalisent des bonnes pratiques :
  - "ne pas ré-inventer la roue"
  - bénéficier de l'expérience
  - faciliter la compréhension du code



IUT Nantes

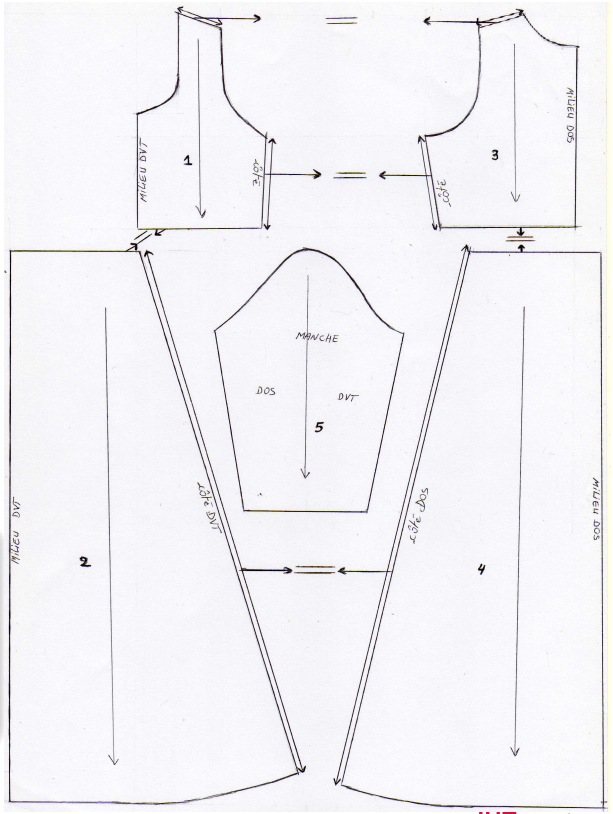
En pratique

Vous utilisez déjà (sans le savoir) de nombreux patrons de conception

Patrons de conception

Un patron de conception (= Design Pattern) est une (la meilleure) *solution de conception* à un problème récurrent, qui pourra être réutilisée et adaptée indéfiniment

- Analogie = les patrons en couture : modèle/plan à réutiliser/adapter
- formalisent des bonnes pratiques :
  - "ne pas ré-inventer la roue"
  - bénéficier de l'expérience
  - faciliter la compréhension du code



IUT Nantes

En pratique

Vous utilisez déjà (sans le savoir) de nombreux patrons de conception

Les patrons de conception du GoF

GoF : 23 patrons de conception

Design Patterns : Elements of Reusable Object-Oriented Software par E. Gamma, R. Helm, R. Johnson et J. Vlissides (= le gang des 4) – 1994.

	Rôle		
Domaine	**de construction**	**Structurel**	**Comportemental**
Classe	Fabrication		Interprète, patron de méthode
Objet	Fabrique, Monteur, Prototype, Singleton	Adaptateur, Pont, Composite, Décorateur, Façade, Poids-mouche, Procuration	Chaîne de responsabilité, Commande, Itérateur, Médiateur, Mémento, Observateur, Etat, Stratégie, Visiteur

IUT Nantes

Patrons (de conception)

Les patrons interviennent à différents niveaux :

- Patrons d'analyse
- Patrons de conception
- Patrons d'architecture
- Patrons d'implémentation
- ...
- Anti-patterns = erreurs courantes de conception/implémentation
  `https://fr.wikipedia.org/wiki/Antipattern`
- Green-patterns = patrons d'éco-conception / éco-développement
- ...

IUT Nantes

Patrons (de conception)

Les patrons interviennent à différents niveaux :

- Patrons d'analyse
- Patrons de conception
- Patrons d'architecture
- Patrons d'implémentation
- ...
- Anti-patterns = erreurs courantes de conception/implémentation
  `https://fr.wikipedia.org/wiki/Antipattern`
- Green-patterns = patrons d'éco-conception / éco-développement
- ...

IUT Nantes

Patrons (de conception)

Les patrons interviennent à différents niveaux :

- Patrons d'analyse
- Patrons de conception
- Patrons d'architecture
- Patrons d'implémentation
- ...
- Anti-patterns = erreurs courantes de conception/implémentation
  `https://fr.wikipedia.org/wiki/Antipattern`
- Green-patterns = patrons d'éco-conception / éco-développement
- ...

IUT Nantes

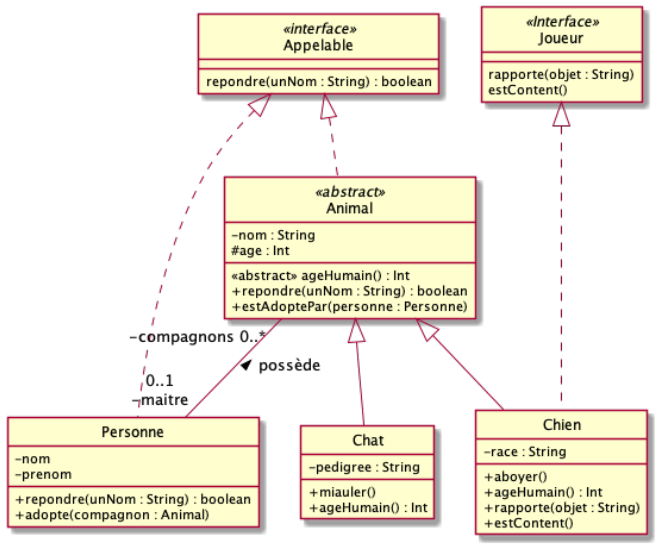
Patrons de conception vs. interfaces

POO vs. Design Patterns

Les design patterns reposent sur les concepts de la POO : en particulier héritage et polymorphisme

Interface

Pour mettre en oeuvre des design patterns, une des clefs est de **séparer** l'interface (quoi ?) de l'implémentation (comment ?)



IUT Nantes

Patrons de conception : langage(s)

Pattern vs. code

Les design patterns sont indépendants des langages de conception/programmation objet utilisés

Dans R3.04

- la conception se fera en **UML** (diagramme de classes, principalement)
- la mise ne oeuvre se fera en **Kotlin**

Autres ressources

Vous aurez l'occasion d'utiliser/implémenter des design patterns en PHP, JavaScript, etc.

IUT Nantes

Patrons de conception : langage(s)

Pattern vs. code

Les design patterns sont indépendants des langages de conception/programmation objet utilisés

Dans R3.04

- la conception se fera en **UML** (diagramme de classes, principalement)
- la mise ne oeuvre se fera en **Kotlin**

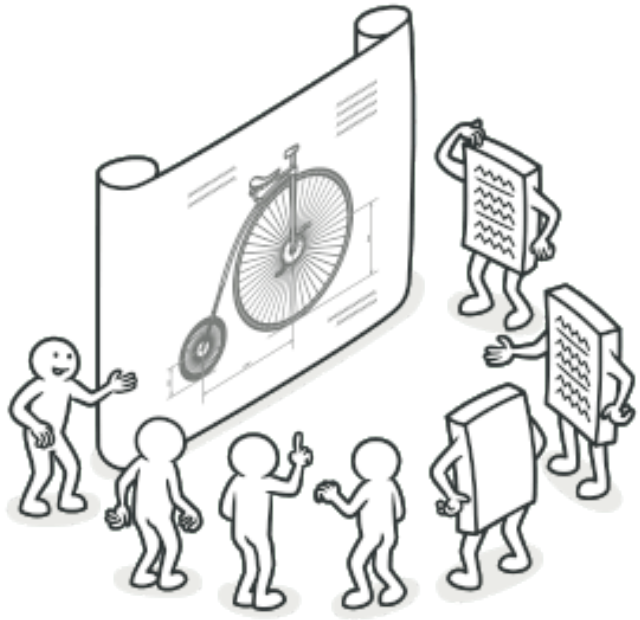
Autres ressources

Vous aurez l'occasion d'utiliser/implémenter des design patterns en PHP, JavaScript, etc.

IUT Nantes

Références / remerçiements

- E. Gamma, R. Helm, R. Johnson et J. Vlissides : *Design Patterns : Elements of Reusable Object-Oriented Software*
- A. Shalloway et J.R. Trott : *Design patterns par la pratique*
- A. Soshin : *Kotlin Design Patterns and Best Practices*
- D. Tamzalit pour son cours et ses TDs/TPs à propos des DPs
- T. Béziers La Fosse pour son cours et ses TDs/TPs à propos des DPs
- Wikipedia : `https://fr.wikipedia.org/wiki/Patron_de_conception`
- Refactoring Guru : `https://refactoring.guru/fr/design-patterns` [1]



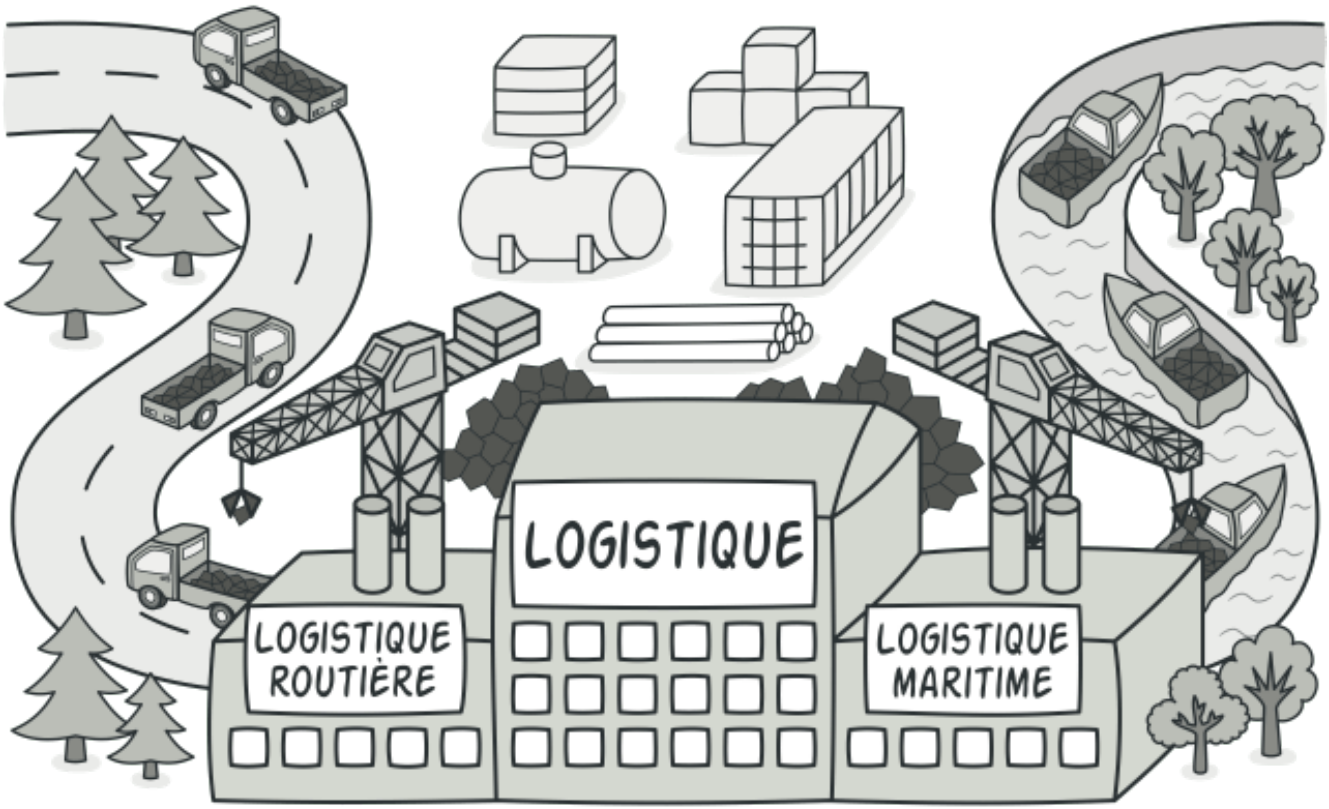
IUT Nantes

1. Merci également pour les images "humoristiques"

Sommaire

IUT Nantes

Patron Fabrique – Factory pattern



Patron Fabrique – Factory pattern

Problème

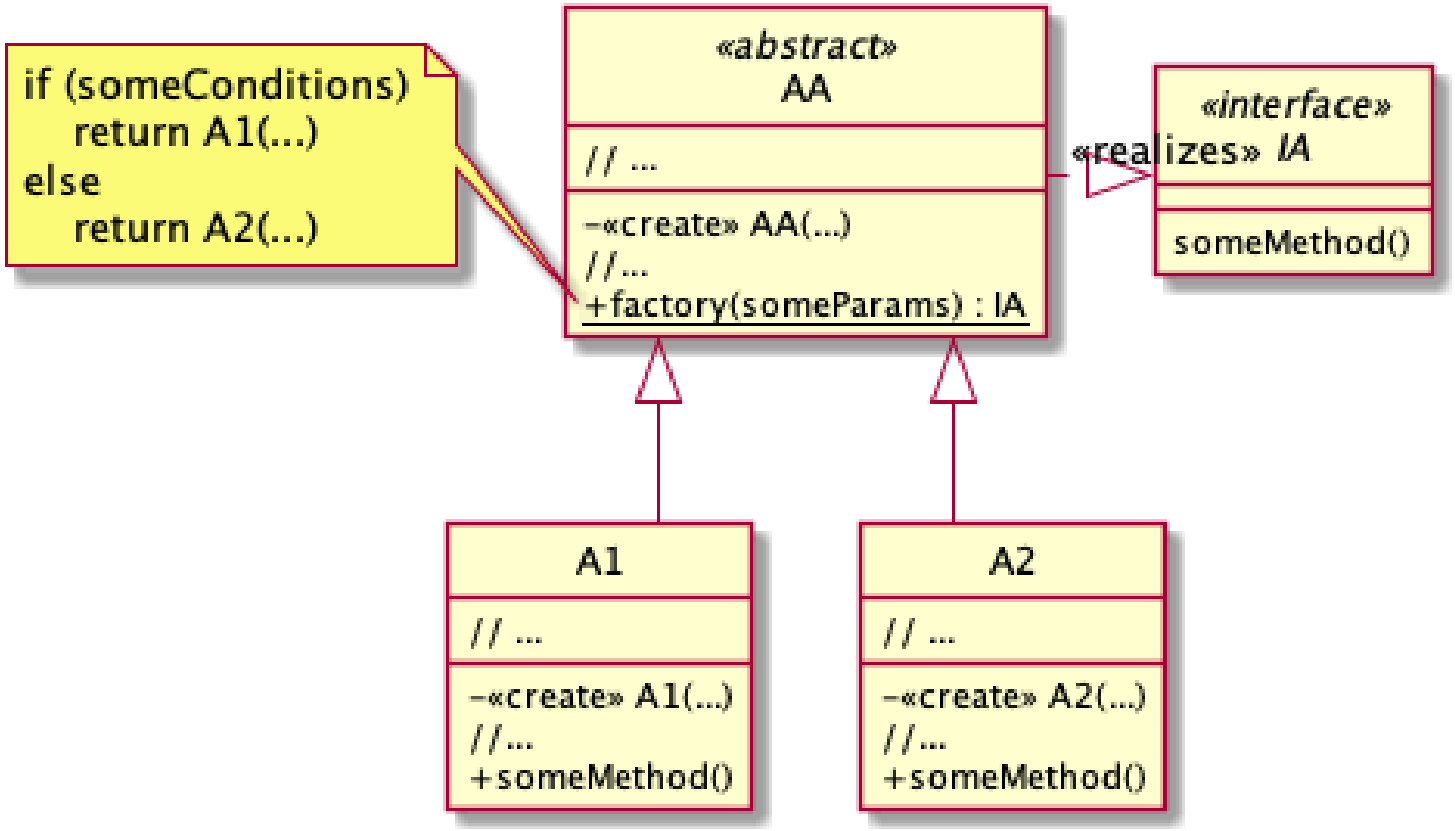
- Comment construire un objet à partir de paramètres complexes ?
- Comment vérifier les paramètres avant de construire l'objet ?
- Quelle(s) classe(s) instancier parmis une hiérarchie de classes ?

Solution

La fabrique fourni une (unique) méthode (statique) qui va retourner une instance de classes (parmi plusieurs sous-classes possibles) en fonction des paramètres

IUT Nantes

Patron Fabrique : schéma UML



En utilisant `factory(...)` la classe exacte instanciée n'est donc pas connue

Patron Fabrique – précisions

- Il est souvent nécessaire de rendre `private` le(s) constructeur(s) des classes considérées
- Fabrique permet de séparer la création des objets de leur utilisation.
- Fabrique fourni des noms plus lisibles / auto-documentées que les constructeurs.
- Plusieurs fabriques peuvent être regroupées en une fabrique abstraite.
- Fabrique est souvent utilisé conjointement à un autre patron de conception.

pour aller plus loin : Builder

Le patron de conception Monteur (Builder) est assez proche, mais dédié à la composition d'objets complexes

IUT Nantes

Patron Fabrique – précisions

- Il est souvent nécessaire de rendre `private` le(s) constructeur(s) des classes considérées
- Fabrique permet de séparer la création des objets de leur utilisation.
- Fabrique fourni des noms plus lisibles / auto-documentées que les constructeurs.
- Plusieurs fabriques peuvent être regroupées en une fabrique abstraite.
- Fabrique est souvent utilisé conjointement à un autre patron de conception.

pour aller plus loin : Builder

Le patron de conception Monteur (Builder) est assez proche, mais dédié à la composition d'objets complexes

IUT Nantes

Exemple de fabrique (1)

```
abstract class Animal(
  val nom : String,
  val age : Int = 1) {

  companion object {
      fun fabrique(sorteAnimal : String,
                   nom : String,
                   age : Int) : Animal {
          return when (sorteAnimal) {
              "chien" -> Chien(nom, age)
              "chat" -> Chat(nom, age)
              else -> throw AnimalException()
          }
```

```
  var monAnimal : Animal
  monAnimal = Animal.fabrique("chien", "rogue", 2)
  monAnimal = Animal.fabrique("chat", "totoro", 1)
  monAnimal = Animal.fabrique("tortue", "leonard", 80)
```

Gestion des erreurs

Utiliser une fabrique permet de gérer les erreurs de construction, ici une levée d'exception

Exemple de fabrique (2)

```
class Complex
private constructor(r : Double, i : Double) {
    private val real = r
    private val imag = i

companion object {
    fun fromCartesian(real : Double, imag : Double) {
        return Complex(real, imag)
    }
    fun fromPolar(rho : Double, theta : Double) {
        return Complex(rho * Math.cos(theta), rho * Math.sin(theta))
    }
}
```

- Le constructeur est `private`, ce qui oblige l'utilisation des fabriques

Kotlin

Les méthodes `listOf(...)`, `mutableListOf(...)`, etc. sont des fabriques

IUT Nantes

Exemple de fabrique (2)

```
class Complex
private constructor(r : Double, i : Double) {
    private val real = r
    private val imag = i

companion object {
    fun fromCartesian(real : Double, imag : Double) {
        return Complex(real, imag)
    }
    fun fromPolar(rho : Double, theta : Double) {
        return Complex(rho * Math.cos(theta), rho * Math.sin(theta))
    }
}
```

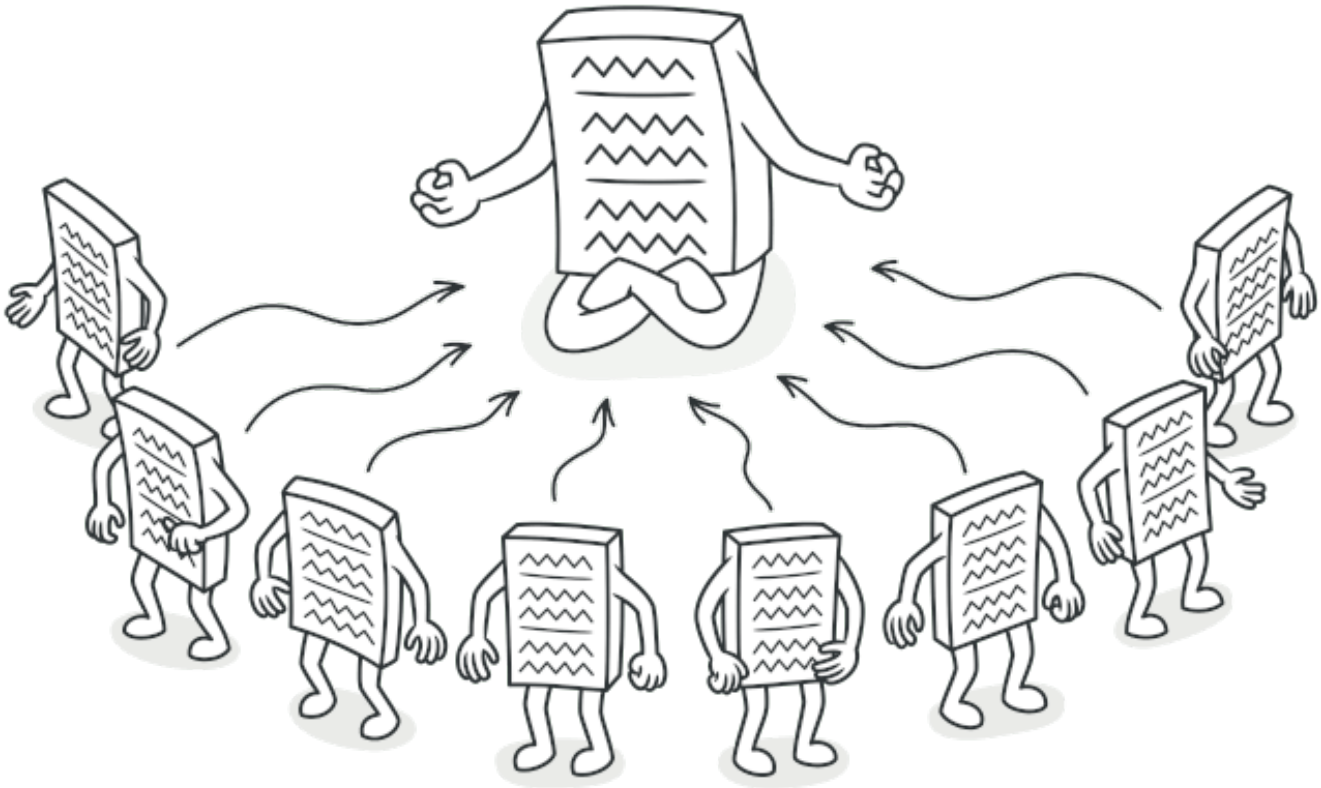
- Le constructeur est `private`, ce qui oblige l'utilisation des fabriques

Kotlin

Les méthodes `listOf(...)`, `mutableListOf(...)`, etc. sont des fabriques

IUT Nantes

Patron Singleton – Singleton pattern



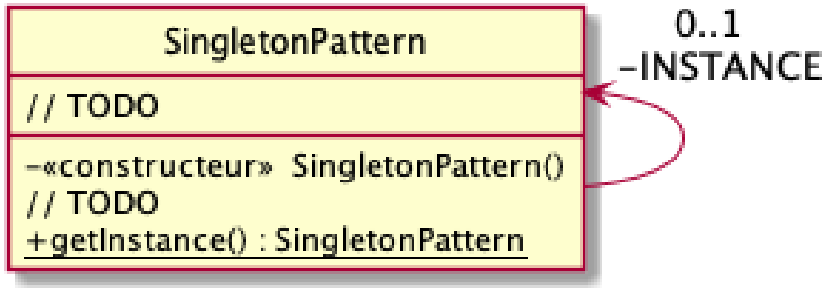
Patron Singleton – Singleton pattern

Problème

Garantir qu'une classe n'a qu'une seule et unique instance

Solution

La classe à créer va fournir un moyen d'accéder à l'instance unique



- Singleton est utilisé lorsqu'on a besoin d'exactement un objet pour coordonner des opérations dans un système
- Permet d'éviter des instanciations multiples, par exemple
  - accès à une BDD
  - flux en écriture vers un fichier
- utilise une fabrique

IUT Nantes

Implémentation(s) de singleton en Kotlin

Implémentation "classique"

```
class SingletonPattern
  private constructor() {

  companion object {

    private val INSTANCE
        = SingletonPattern()

    fun getInstance()
        : SingletonPattern {
      return INSTANCE
    }
  }
}
```

Initialisation "paresseuse"

```
class SingletonPattern
  private constructor() {

  companion object {

    private var INSTANCE :
          SingletonPattern?

    fun getInstance() :
        SingletonPattern {
      if (INSTANCE == null)
        INSTANCE =
          SingletonPattern()
      return INSTANCE!!
    }
  }
}
```

Le constructeur est privé.
La variable statique `INSTANCE` n'est initialisée qu'une fois.

IUT Nantes

Implémentation(s) de singleton en Kotlin

Implémentation "classique"

```
class SingletonPattern
  private constructor() {

  companion object {

    private val INSTANCE
        = SingletonPattern()

    fun getInstance()
        : SingletonPattern {
      return INSTANCE
    }
  }
}
```

Initialisation "paresseuse"

```
class SingletonPattern
  private constructor() {

  companion object {

    private var INSTANCE :
          SingletonPattern?

    fun getInstance() :
        SingletonPattern {
      if (INSTANCE == null)
        INSTANCE =
          SingletonPattern()
      return INSTANCE!!
    }
  }
}
```

Le constructeur est privé.
La variable statique `INSTANCE` n'est initialisée qu'une fois.

IUT Nantes

Implémentation(s) simplifiée(s) de Singleton en Kotlin

en utilisant "by lazy"

```
class SingletonPattern
private constructor() {

  companion object {

    private val INSTANCE :
      SingletonPattern
      by laszy {
        SingletonPattern()
      }

    fun getInstance() =
      INSTANCE
  }
}
```

en utilisant "object"

Kotlin propose une construction spécifique de Singleton : `object`

```
object SingletonPattern2 {
  // TODO
}
```

ATTENTION

Plus compliquer à assurer dans le cas de programmation multi-threadée.

Implémentation(s) simplifiée(s) de Singleton en Kotlin

en utilisant "by lazy"

```
class SingletonPattern
private constructor() {

  companion object {

    private val INSTANCE :
      SingletonPattern
      by laszy {
        SingletonPattern()
      }

    fun getInstance() =
      INSTANCE
  }
}
```

en utilisant "object"

Kotlin propose une construction spécifique de Singleton : `object`

```
object SingletonPattern2 {
  // TODO
}
```

ATTENTION

Plus compliquer à assurer dans le cas de programmation multi-threadée.

Implémentation(s) simplifiée(s) de Singleton en Kotlin

en utilisant "by lazy"

```
class SingletonPattern
private constructor() {

  companion object {

    private val INSTANCE :
      SingletonPattern
      by laszy {
        SingletonPattern()
      }

    fun getInstance() =
      INSTANCE
  }
}
```

en utilisant "object"

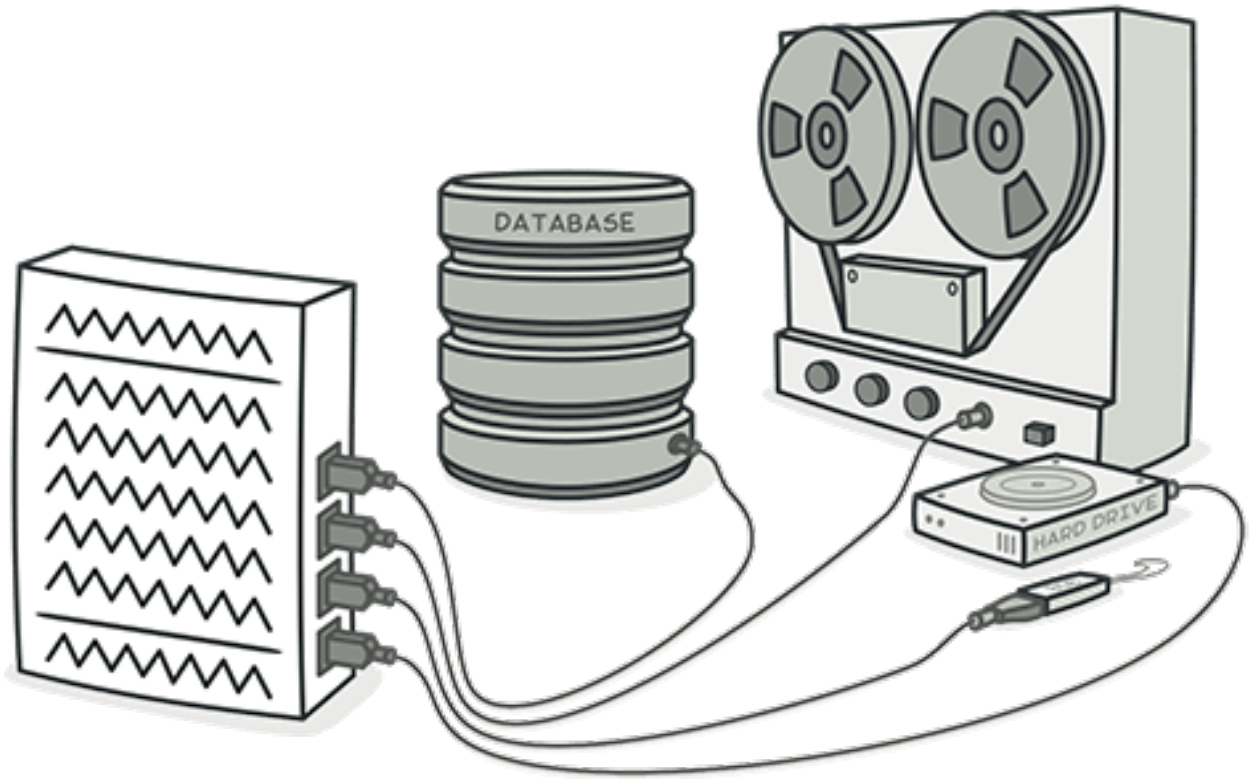
Kotlin propose une construction spécifique de Singleton : `object`

```
object SingletonPattern2 {
  // TODO
}
```

ATTENTION

Plus compliquer à assurer dans le cas de programmation multi-threadée.

Patron Objet d'accès aux données – DAO pattern



Patron Objet d'accès aux données – DAO pattern

Problème

Les objets métiers instanciés sont liés à des données persistentes (fichiers, BDD)

Solution

Le patron DAO (Data Acess Object) propose de séparer les classes métiers, de classes "techniques" réalisant la liaison avec le stockage persistent.

- Les classes métiers ne sont modifiées que si les règles métiers changent
- On peut changer la méthode d'accès aux données sans impacter les classes métiers

Pour aller plus loin : Repository

La différence est que **DAO** focuse sur la persistence des données alors que **Repository** focuse sur l'abstraction des données.

IUT Nantes

Patron Objet d'accès aux données – DAO pattern

Problème

Les objets métiers instanciés sont liés à des données persistentes (fichiers, BDD)

Solution

Le patron DAO (Data Acess Object) propose de séparer les classes métiers, de classes "techniques" réalisant la liaison avec le stockage persistent.

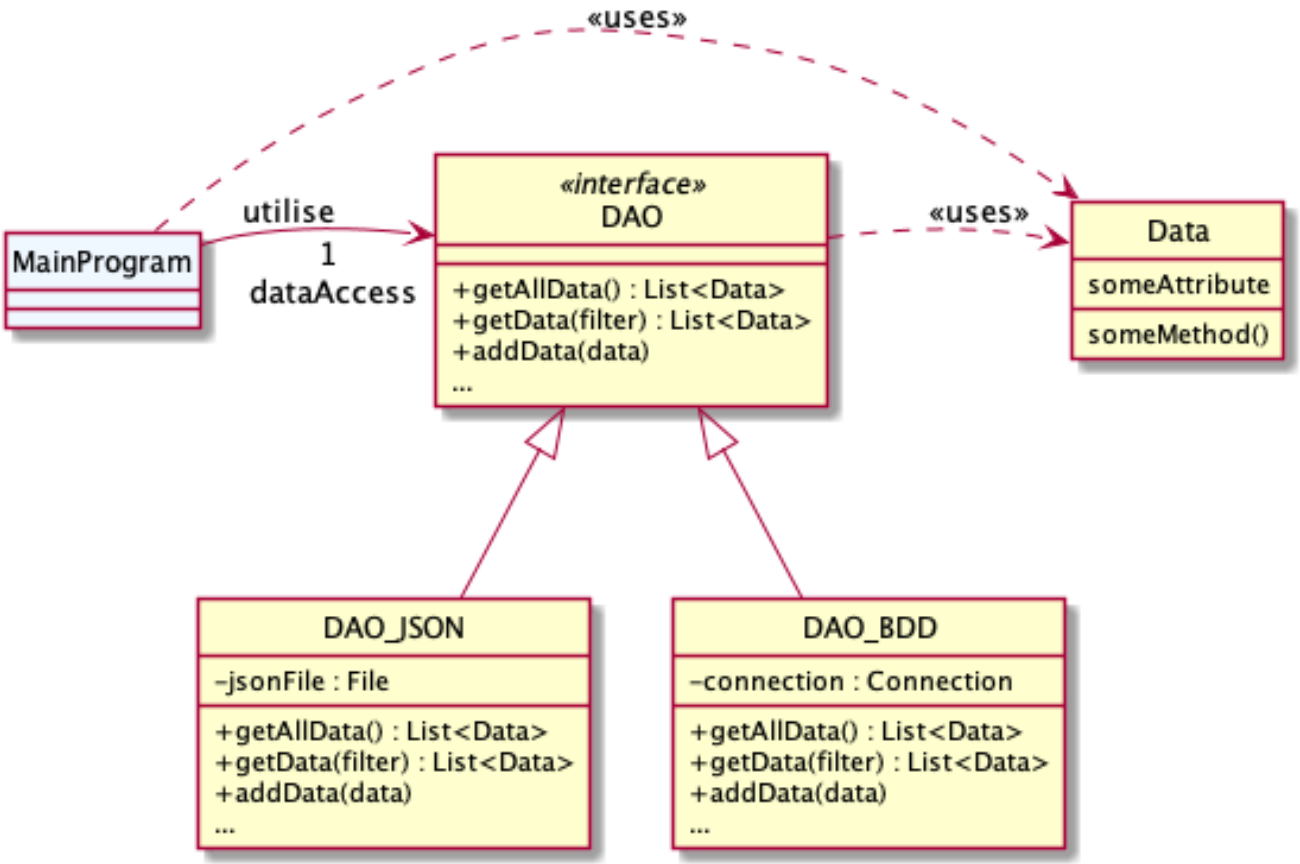
- Les classes métiers ne sont modifiées que si les règles métiers changent
- On peut changer la méthode d'accès aux données sans impacter les classes métiers

Pour aller plus loin : Repository

La différence est que **DAO** focuse sur la persistence des données alors que **Repository** focuse sur l'abstraction des données.

IUT Nantes

Patron DAO : schéma UML



IUT Nantes

Exemple de DAO

```
[
  { "nom" : "Lego",
    "prix": 10.0,
    "quantite" : 30 },
  { "nom" : "Playmobil",
  "prix": 20.0,
  "quantite" : 50 },
  ...
]
```

```
@Serializable
data class Produit(
    val nom : String,
    var prix : Double,
    var quantite : Int
)
```

```
class ProduitDAO (val json : File) {

  fun donneTousLesProduits() : List<Produit> {
      return Json
          .decodeFromStream<List<Produit>>(json.inputStream())
  }

  fun donneProduit(nom : String) : Produit {
      return Json
              .decodeFromStream<List<Produit>>(json.inputStream())
              .filter {it.nom == nom} .get(0)
  }
}
```

IUT Nantes

Sommaire

IUT Nantes

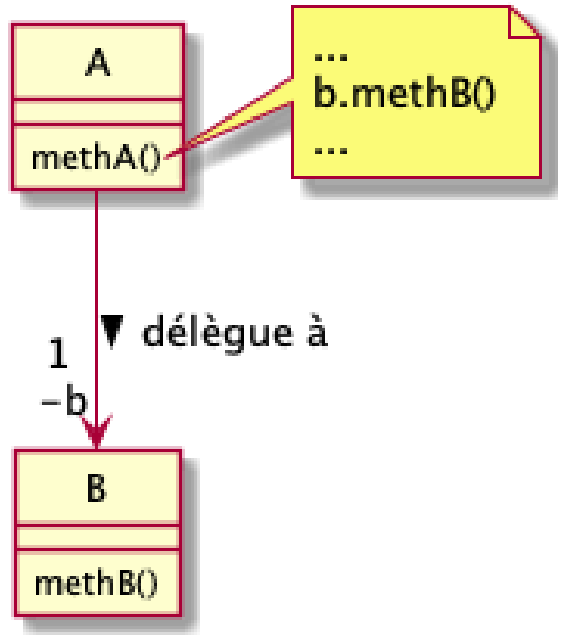
Patron Délégation – Delegate pattern

Problème

Implémenter dans une classe `A` un (des) traitement(s) complexe(s) alors qu'on connaît une autre classe `B` qui sait déjà faire ce(s) traitement(s)

Solution

Déléguer à la classe `B` les traitements à réaliser, en ajoutant à `A` un attribut de type B et en appelant les méthodes de `B` dans `A`
$\Rightarrow$ on parle de composition



- On doit pouvoir remplacer la classe `B` sans impacter l'usage de `A`

NB : vous utilisez tout le temps des délégations, dès que vous utilisez une classe fournie par Kotlin.

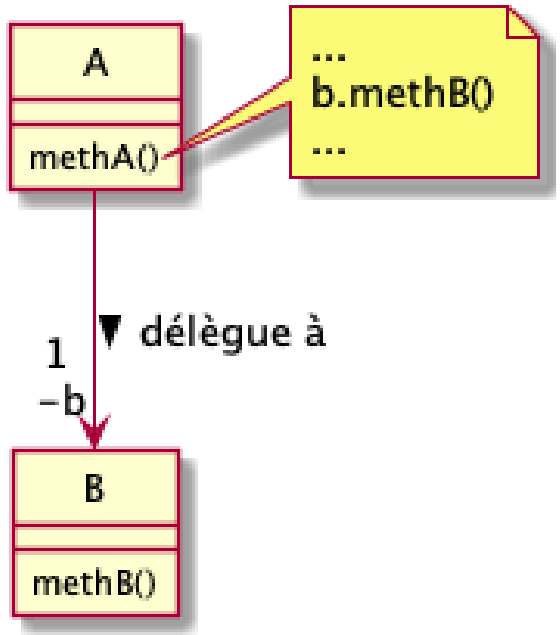
Patron Délégation – Delegate pattern

Problème

Implémenter dans une classe `A` un (des) traitement(s) complexe(s) alors qu'on connaît une autre classe `B` qui sait déjà faire ce(s) traitement(s)

Solution

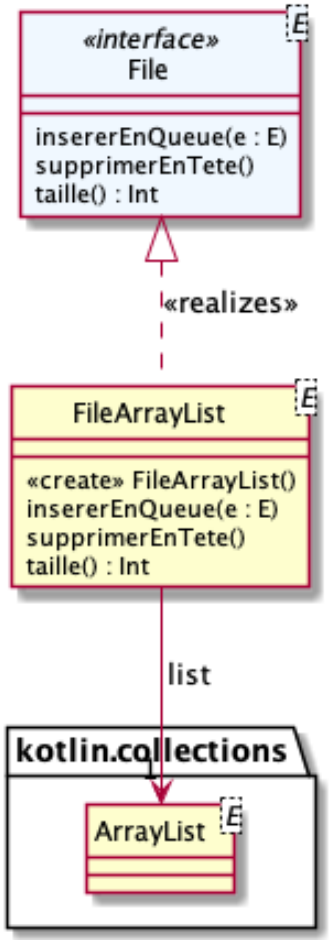
Déléguer à la classe `B` les traitements à réaliser, en ajoutant à `A` un attribut de type B et en appelant les méthodes de `B` dans `A`
$\Rightarrow$ on parle de composition



- On doit pouvoir remplacer la classe `B` sans impacter l'usage de `A`

NB : vous utilisez tout le temps des délégations, dès que vous utilisez une classe fournie par Kotlin.

Exemple de délégation

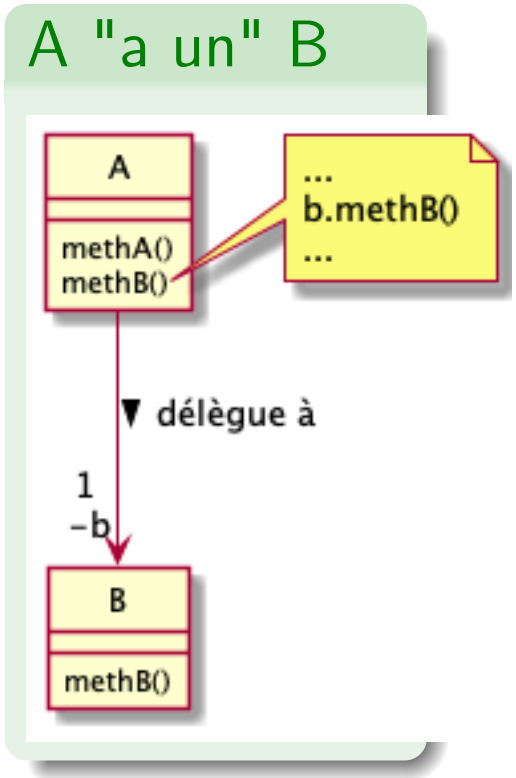



```kotlin
class FileArrayList<E> : File<E> {
  val list = ArrayList<E>()

  override fun insererEnQueue(element: E) {
      list.add(element)
  }

  override fun supprimerEnTete() {
      list.removeFirst()
  }

  override fun taille(): Int {
      return list.size
  }
  ...
```

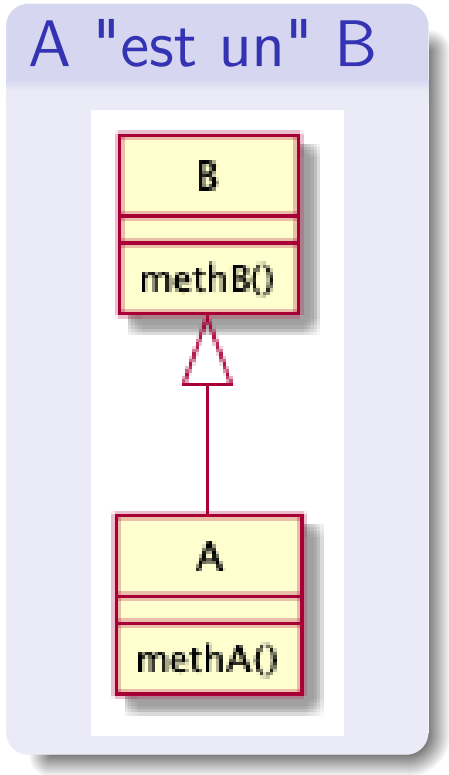
IUT Nantes

Délégation (directe) vs. héritage

Souvent, on peut hésiter entre délégation ou héritage



vs.

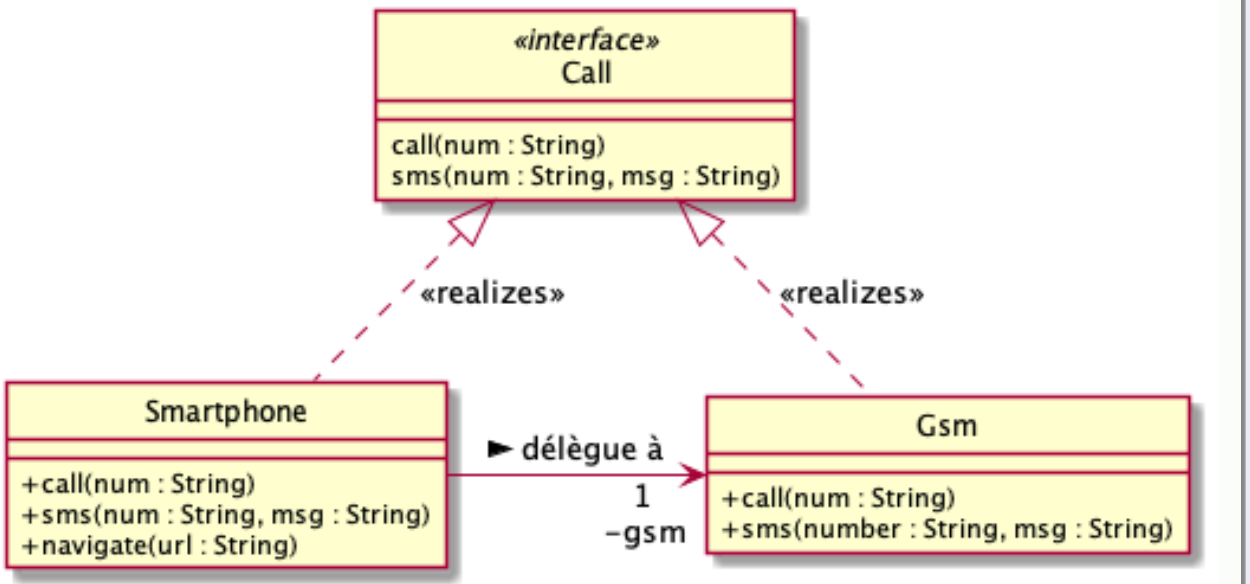


- L'objet "B" peut être réutilisé dans plusieurs classes différentes
- La délégation permet l'injection de dépendances

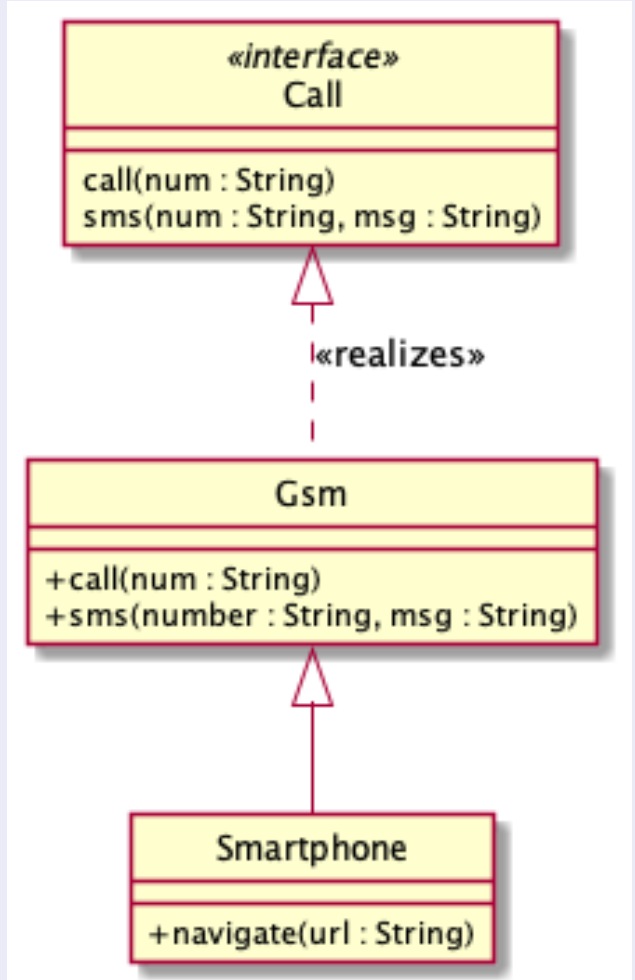
IUT Nantes

Délégation vs . héritage : exemple

Smartphone "a un" Gsm

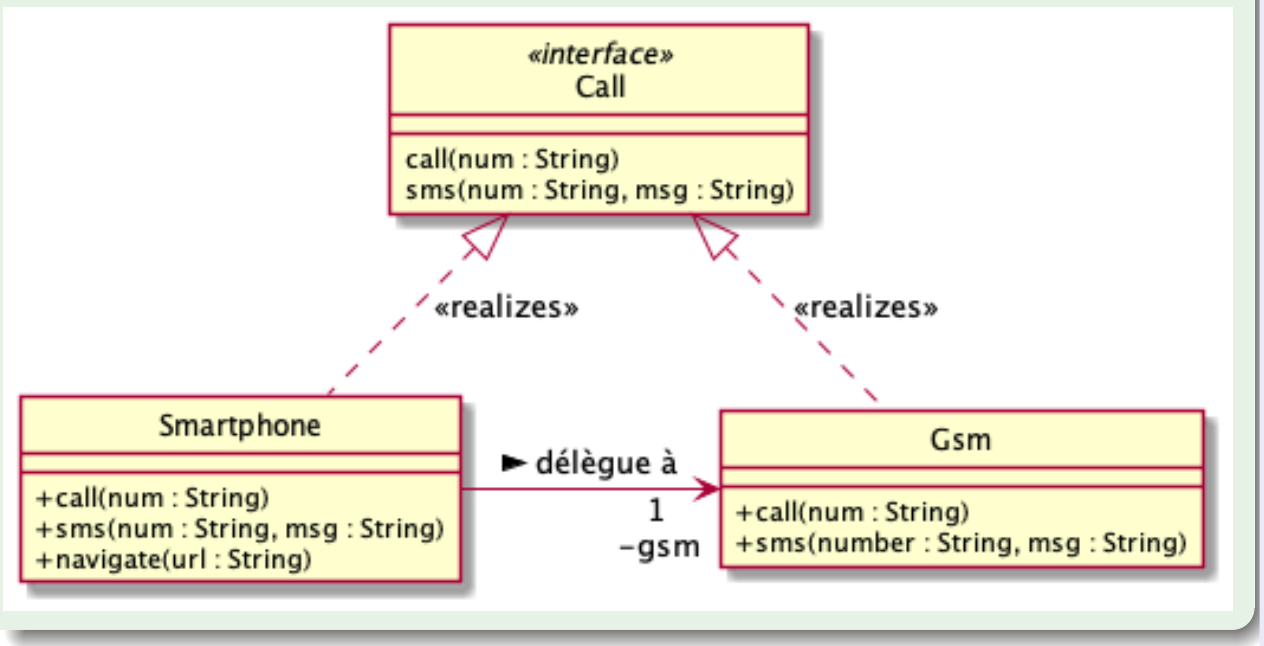


Smartphone "est un" Gsm

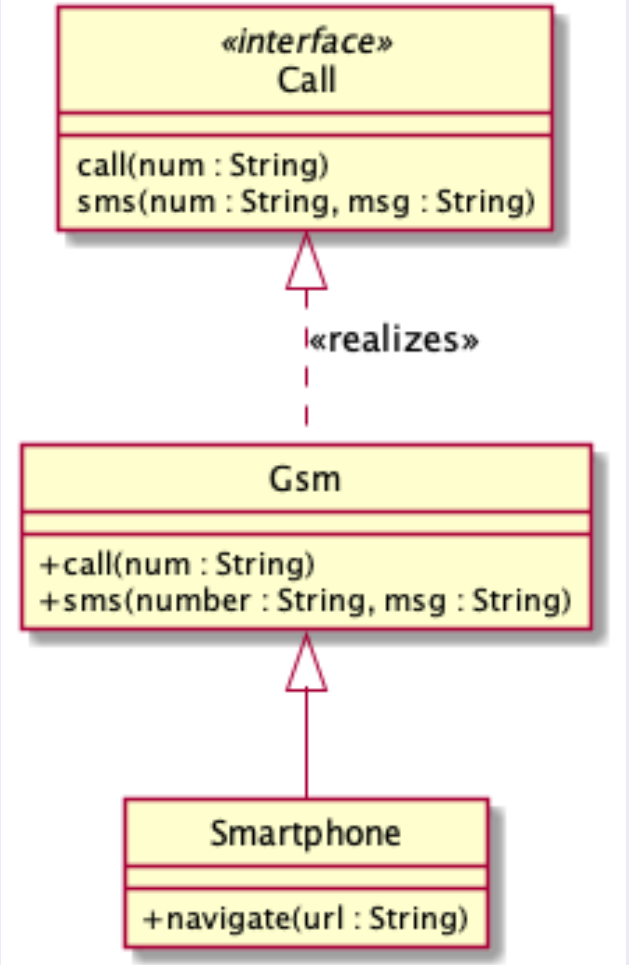


Délégation vs . héritage : exemple

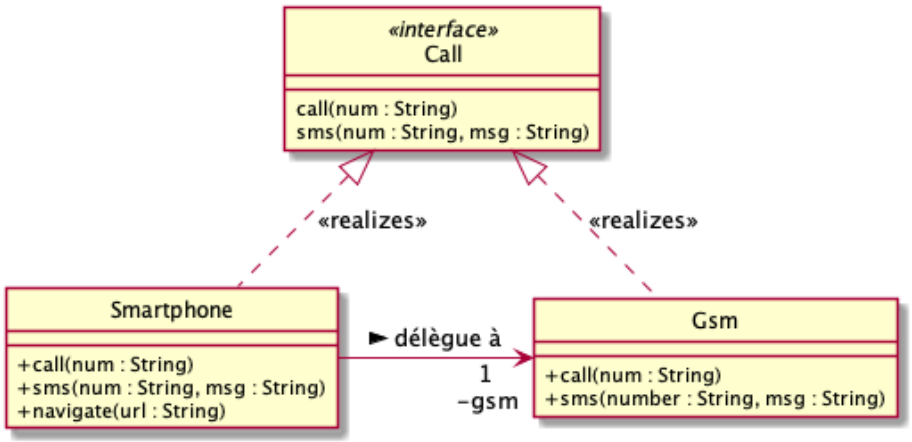
Smartphone "a un" Gsm



Smartphone "est un" Gsm



Délégation directe en Kotlin grâce à `by`



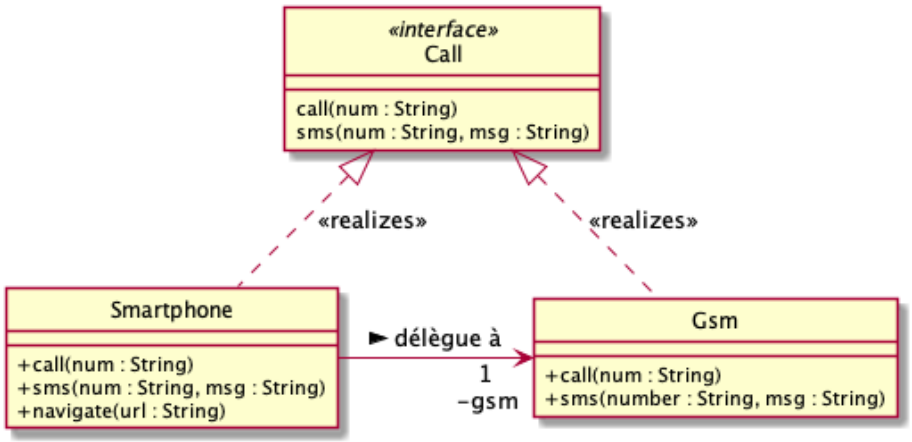

```kotlin
class Gsm : Call {
  override fun call(num: String) {
      println("call $num")
  }

  override fun sms(num: String,
                   msg: String) {
      println("send $msg to $num")
  }
}
```

```kotlin
class Smartphone(val gsm : Gsm)
                        : Call {
  override fun call(num: String) {
      gsm.call(num)
  }
  override fun sms(num: String,
                   msg: String) {
      gsm.sms(num, msg)
  }
  fun navigate(url : String) {
      println("go to $url")
  }
```

Version simplifiée utilisant `by`

```kotlin
class Smartphone(val gsm : Gsm)
                : Call by gsm {
  fun navigate(url : String) {
      println("go to $url")
  }
}
```

Délégation directe en Kotlin grâce à `by`



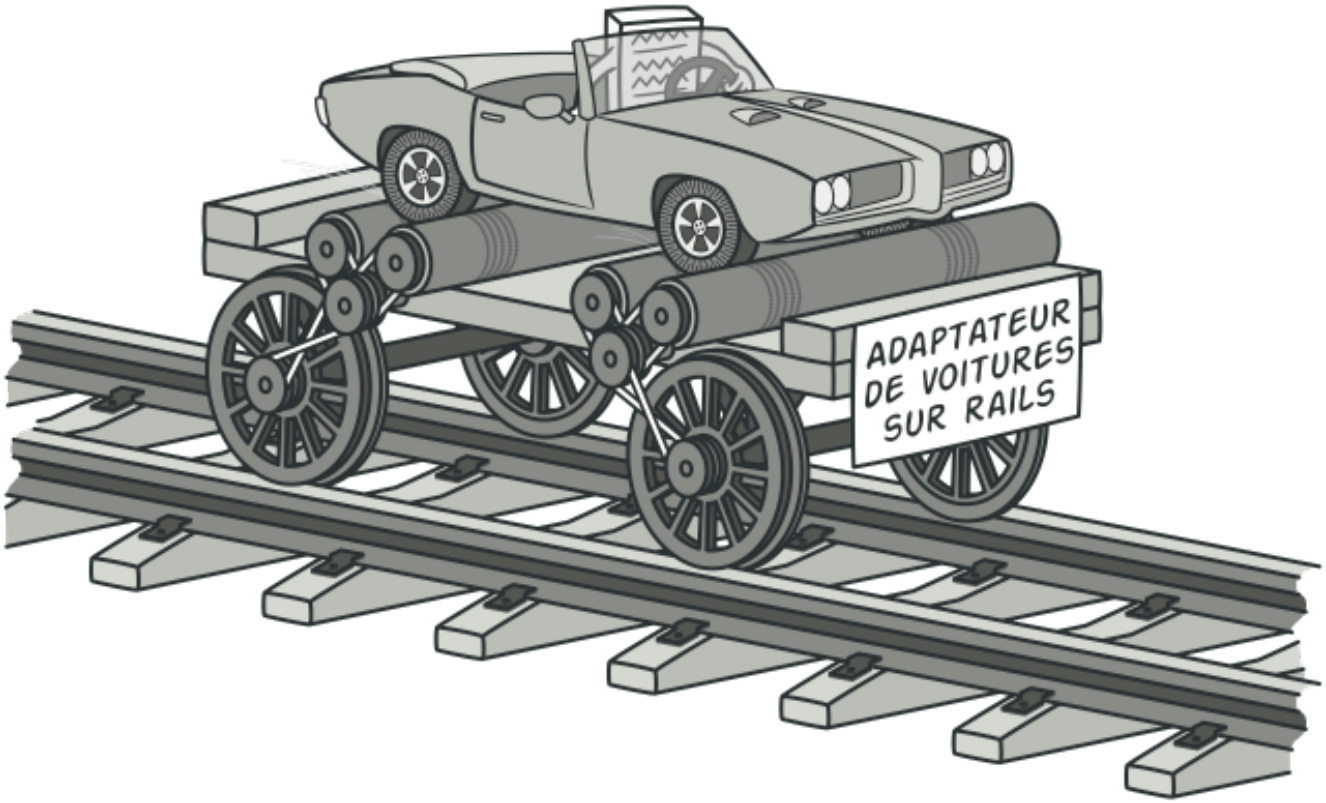

```kotlin
class Gsm : Call {
  override fun call(num: String) {
      println("call $num")
  }

  override fun sms(num: String,
                   msg: String) {
      println("send $msg to $num")
  }
}
```

```kotlin
class Smartphone(val gsm : Gsm)
                        : Call {
  override fun call(num: String) {
      gsm.call(num)
  }
  override fun sms(num: String,
                   msg: String) {
      gsm.sms(num, msg)
  }
  fun navigate(url : String) {
      println("go to $url")
  }
```

Version simplifiée utilisant `by`

```kotlin
class Smartphone(val gsm : Gsm)
                 : Call by gsm {
  fun navigate(url : String) {
      println("go to $url")
  }
}
```

Patron Adaptateur – Adapter pattern



Patron Adaptateur – Adapter pattern

Problème

Comment continuer à utiliser une bibliothèque dont l'interface à été modifiée sans toucher au reste du programme ?

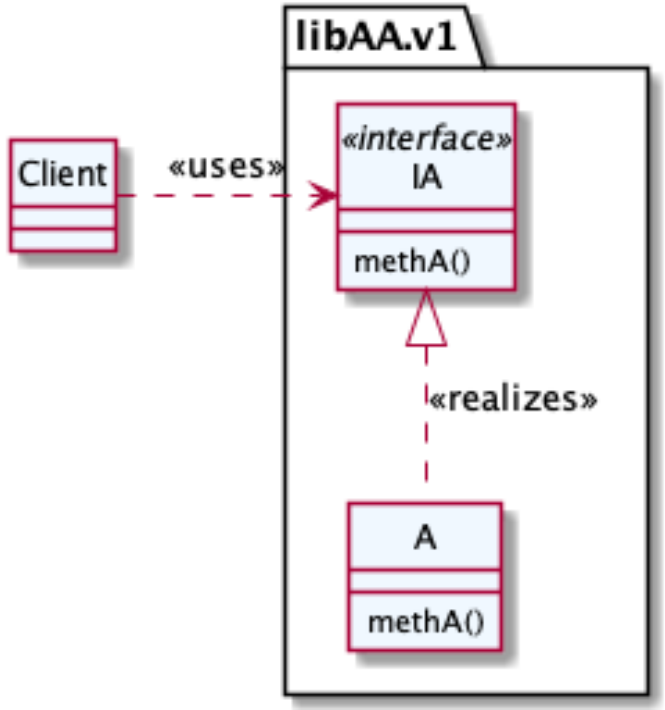
Solution

Une nouvelle classe va exposer l'ancienne interface et utiliser les méthodes de la nouvelle pour réaliser l'ancienne interface

IUT Nantes

Adaptateur : schéma UML

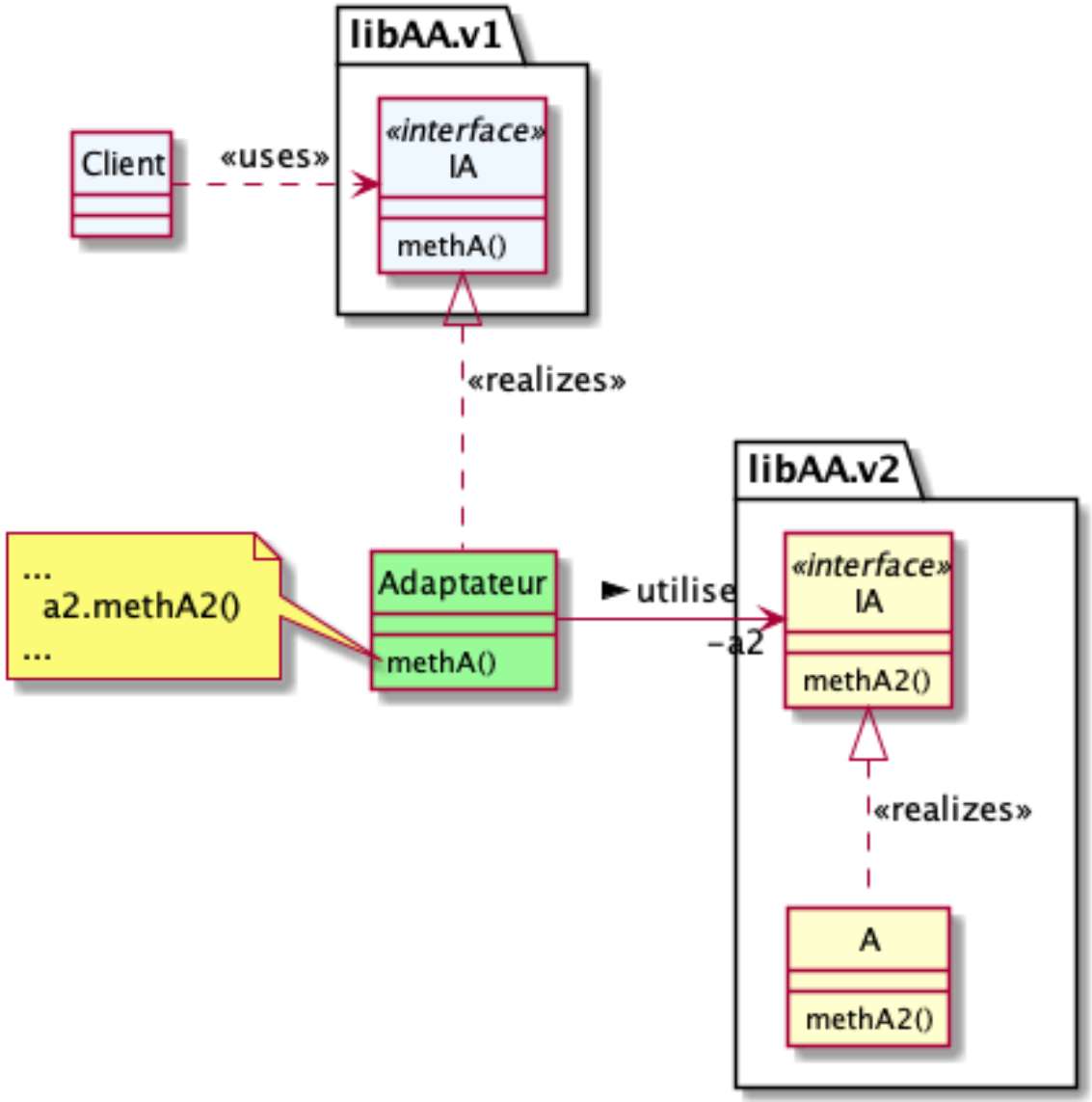
La librairie `libAA` est mise à jour : l'interface d'utilisation a changée



IUT Nantes

Adaptateur : schéma UML

La librairie `libAA` est mise à jour : l'interface d'utilisation a changée



Adapter – précisions

- Adaptateur converti l'interface d'une classe en une autre forme **conforme** à l'attente du client.
- Permet d'interconnecter des classes qui sans cela seraient incompatibles.
- Adaptateur peut également être utiliser pour remplacer une bibliotèque par une autre
- Le reste du programme utilisera l'adaptateur de manière transparente.
- L'adaptateur peut utiliser plusieurs classes pour réaliser l'interface attendue.

Pour aller plus loin : patron Proxy

Variante d'Adapter avec une restriction de l'interface réalisée

IUT Nantes

Adapter – précisions

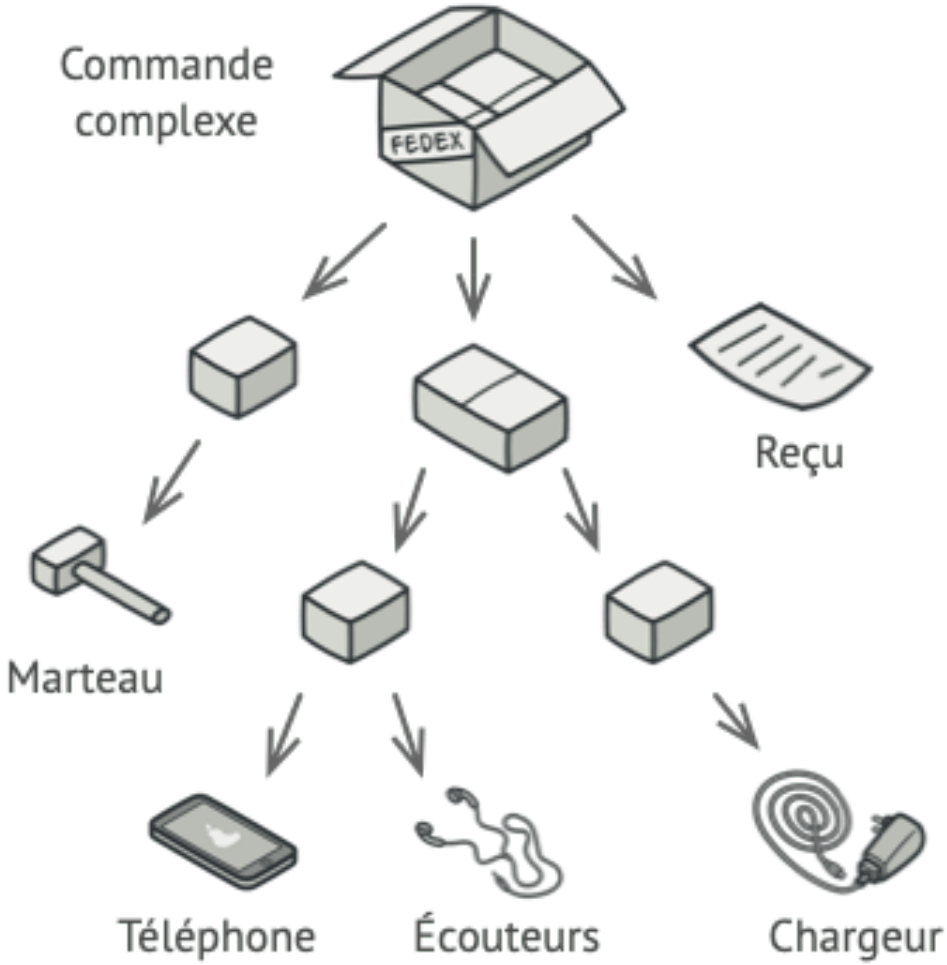
- Adaptateur converti l'interface d'une classe en une autre forme **conforme** à l'attente du client.
- Permet d'interconnecter des classes qui sans cela seraient incompatibles.
- Adaptateur peut également être utiliser pour remplacer une bibliotèque par une autre
- Le reste du programme utilisera l'adaptateur de manière transparente.
- L'adaptateur peut utiliser plusieurs classes pour réaliser l'interface attendue.

Pour aller plus loin : patron Proxy

Variante d'Adapter avec une restriction de l'interface réalisée

IUT Nantes

Patron Composite – Composite pattern



IUT Nantes

Patron Composite – Composite pattern

Problème

Comment représenter une structure arborescente ?

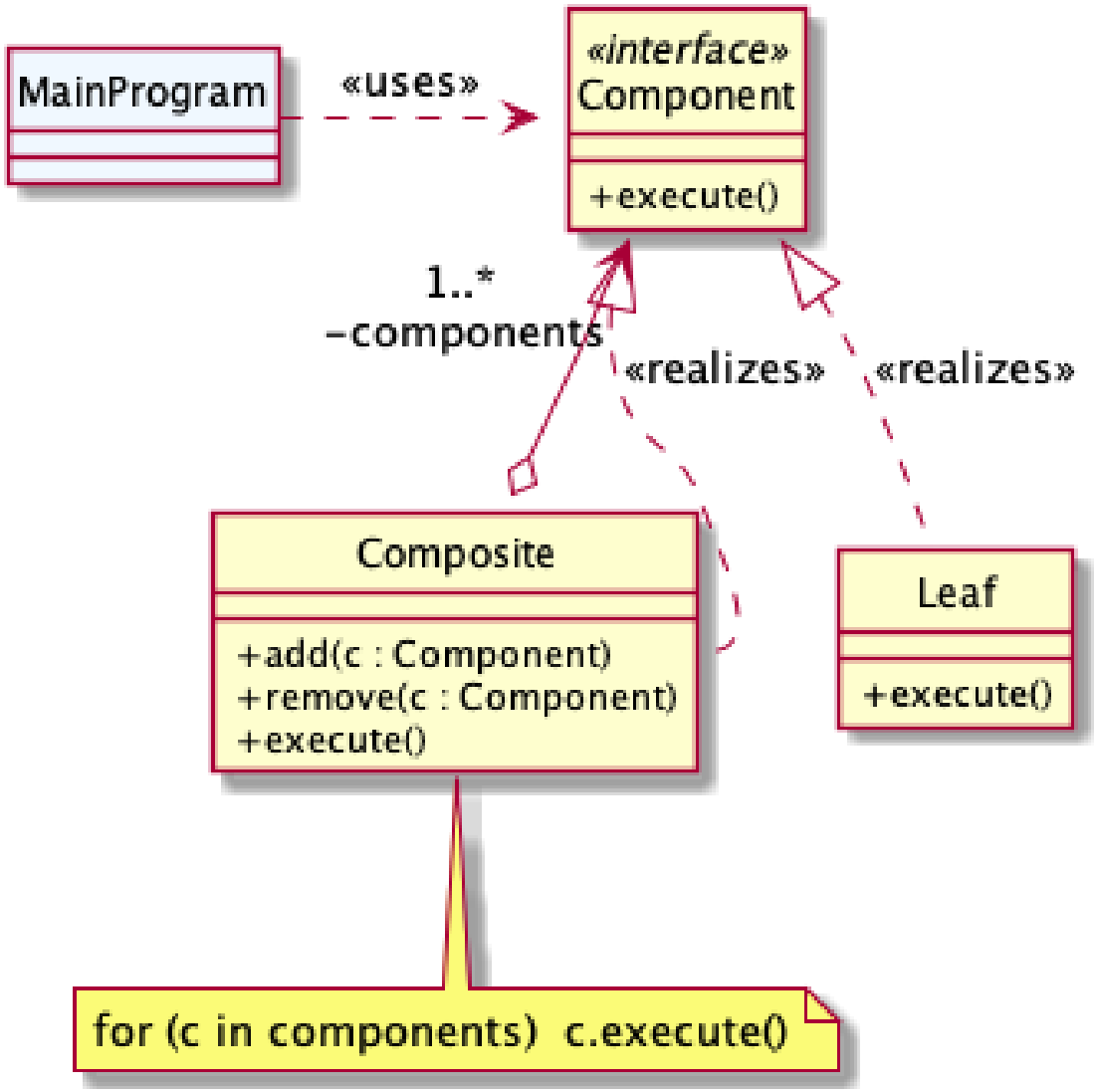
Solution

Les composants-feuilles et les composite-conteneurs implémentent une même interface

- En pratique, l'utilisateur n'aura pas à distinguer entre les objets primitifs et les conteneurs.

IUT Nantes

Patron Composite : schéma UML

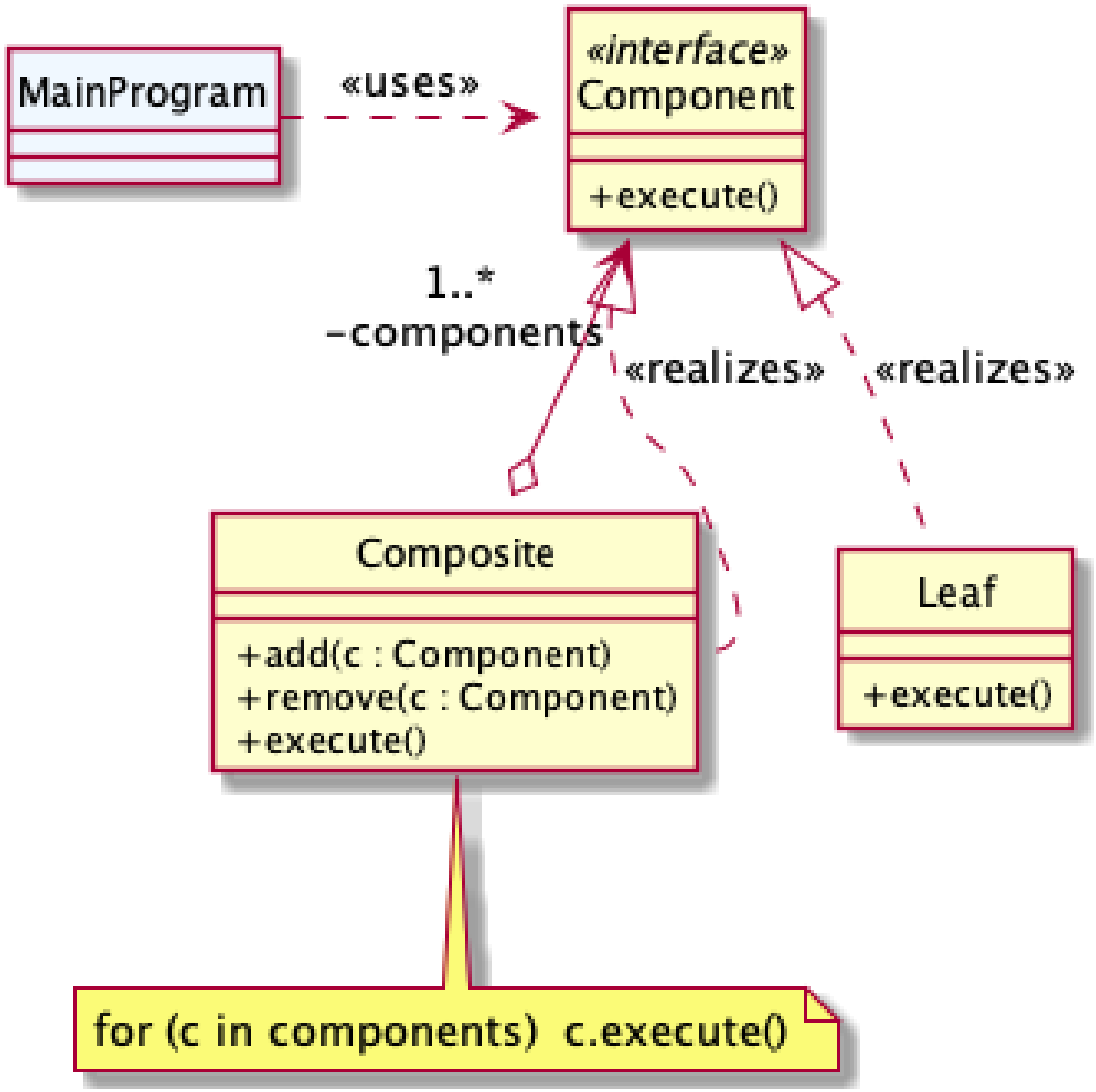


Notations UML

Notez l'utilisation du "◊" sur l'association, indiquant une aggrégation UML "♦" possible, pour indiquer une composition UML

IUT Nantes

Patron Composite : schéma UML

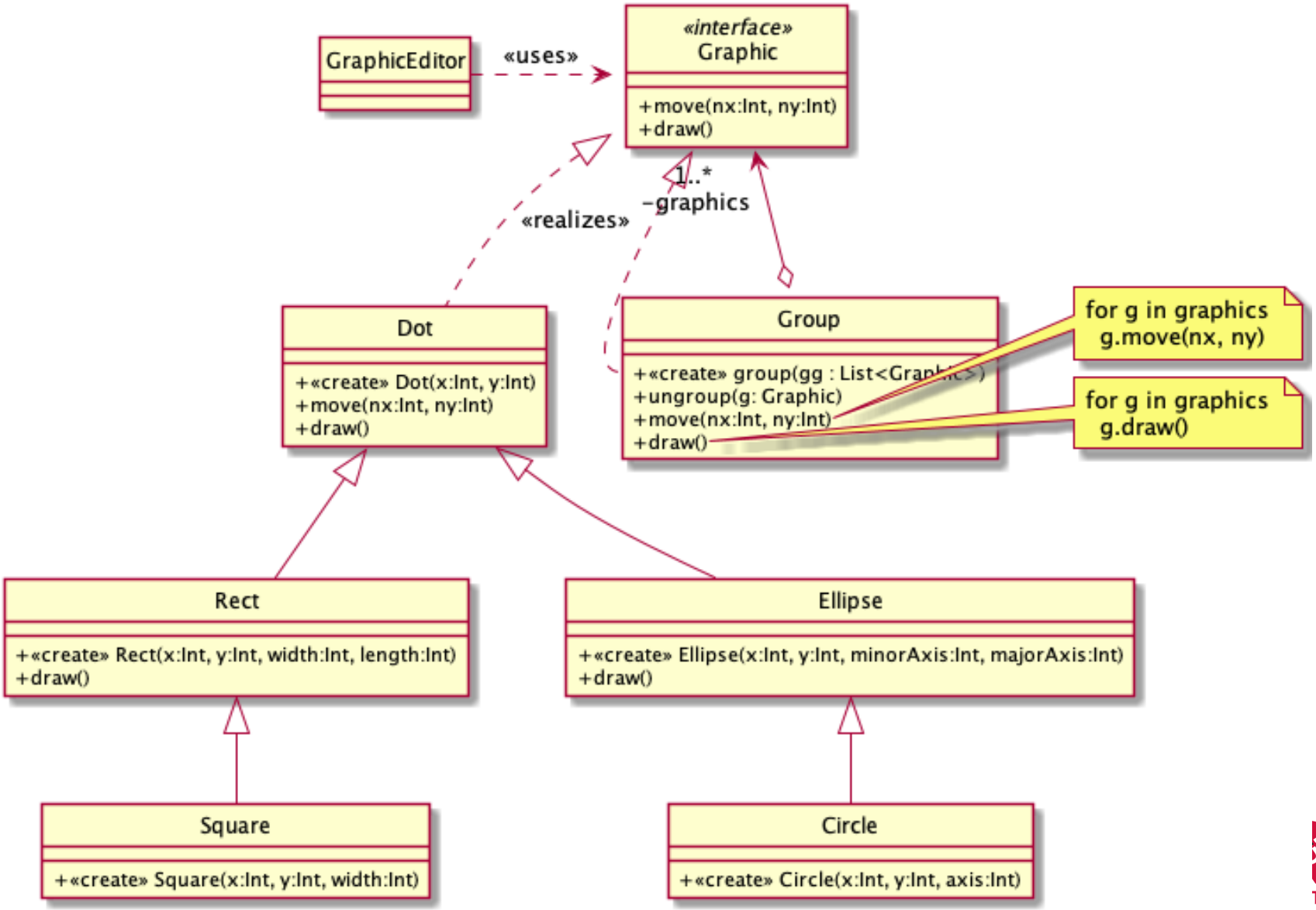


Notations UML

Notez l'utilisation du "◊" sur l'association, indiquant une aggrégation UML "♦" possible, pour indiquer une composition UML

IUT Nantes

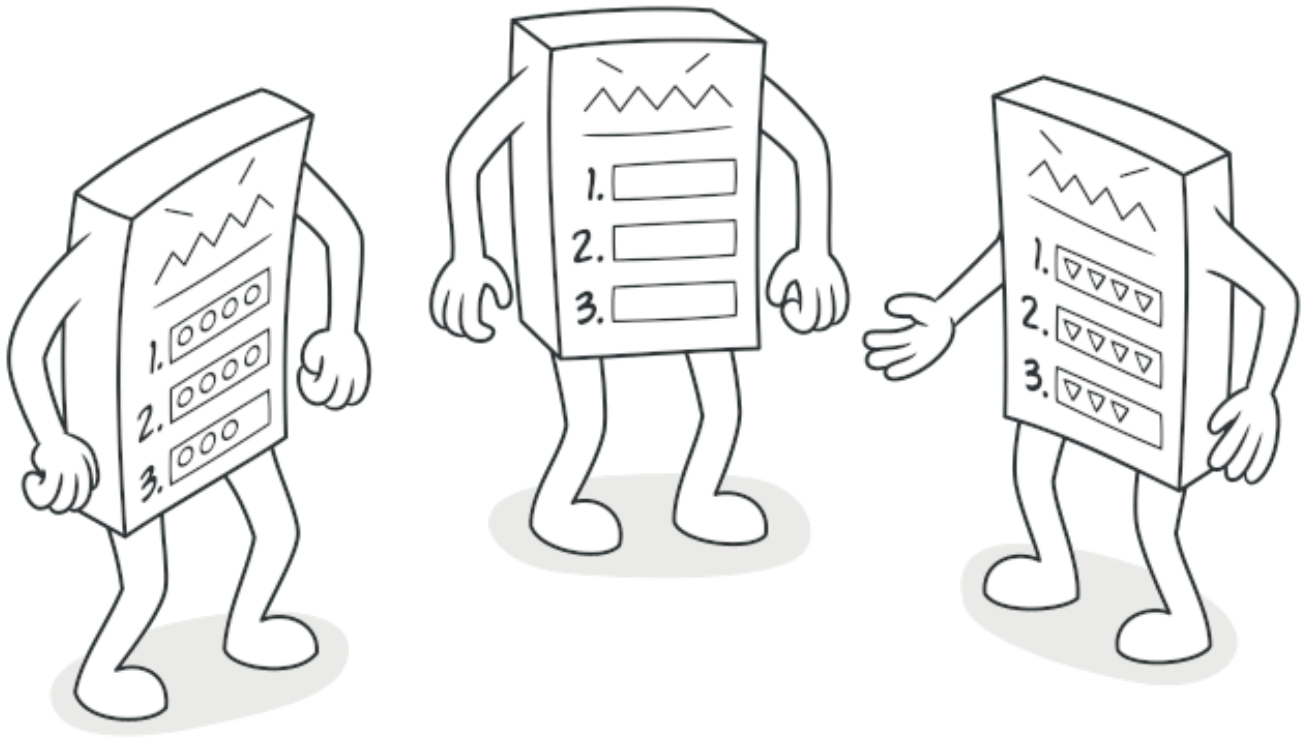
Exemple de structure Composite



Sommaire

IUT Nantes

Patron de méthode – Template method pattern



Patron de méthode – Template method pattern

Problème

Comment généraliser un algorithme, dont uniquement certaines étapes sont spécifiques ? La spécificité dépend de la sous-classe sur laquelle s'applique l'algorithme

Solution

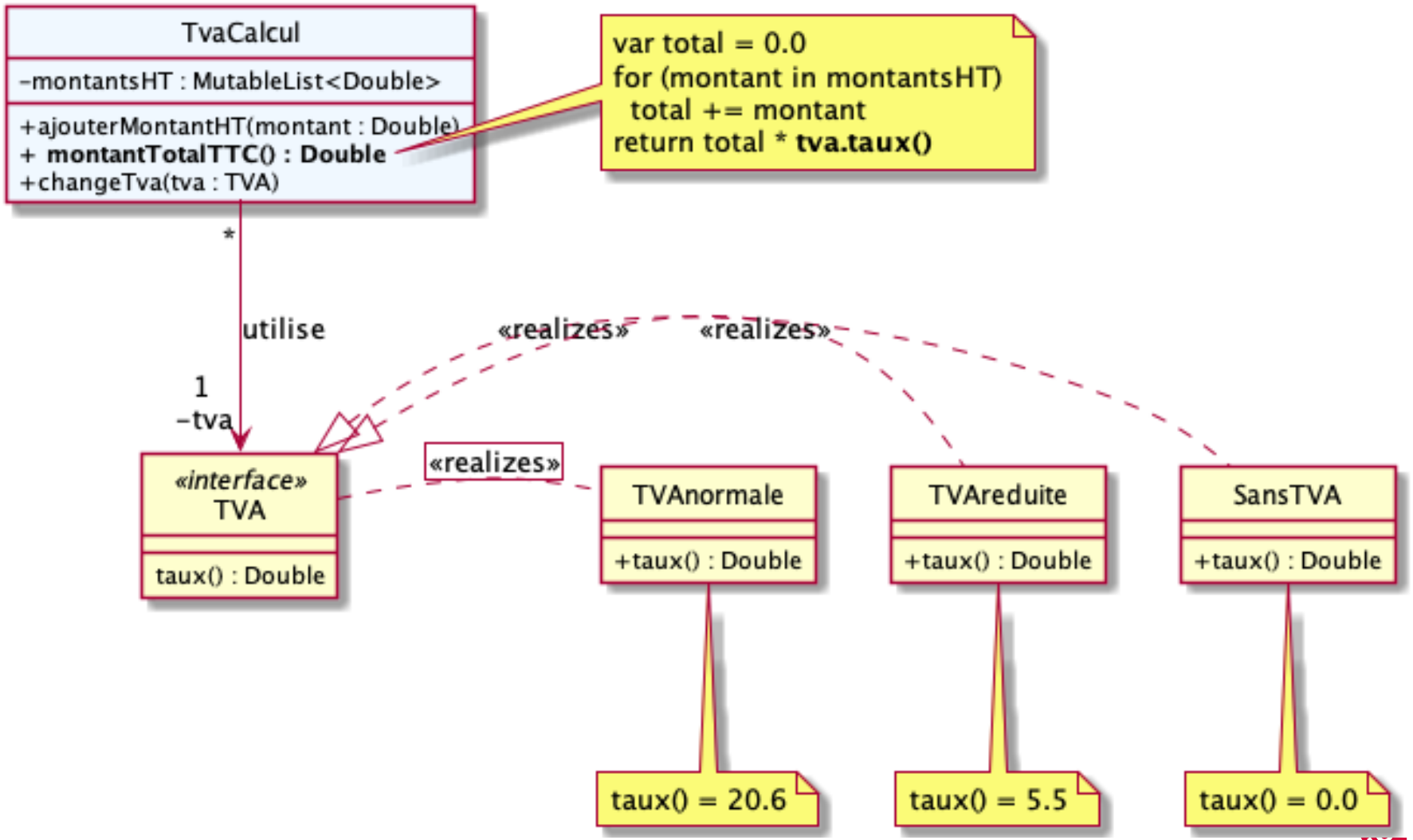
Le patron de méthode propose de

1. définir les parties spécifiques comme des méthodes d'une interface
2. implémenter le squelette de l'algorithme en utilisant les méthodes définies par l'interface

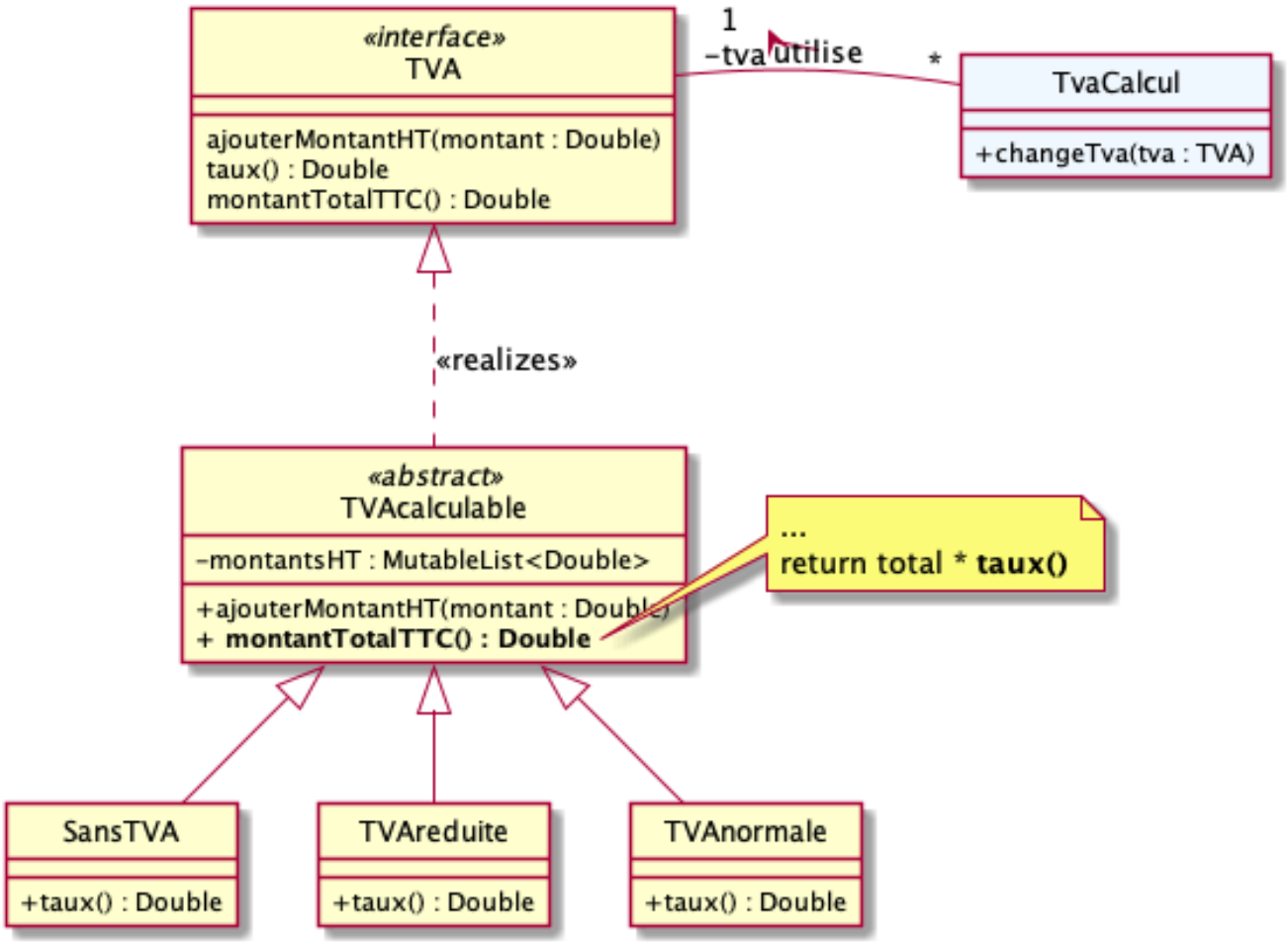
- permet de factoriser du code qui serait redondant
- l'algorithme peut être défini dans une classe abstraite parente ou dans une autre classe

IUT Nantes

Exemple de Patron de méthode

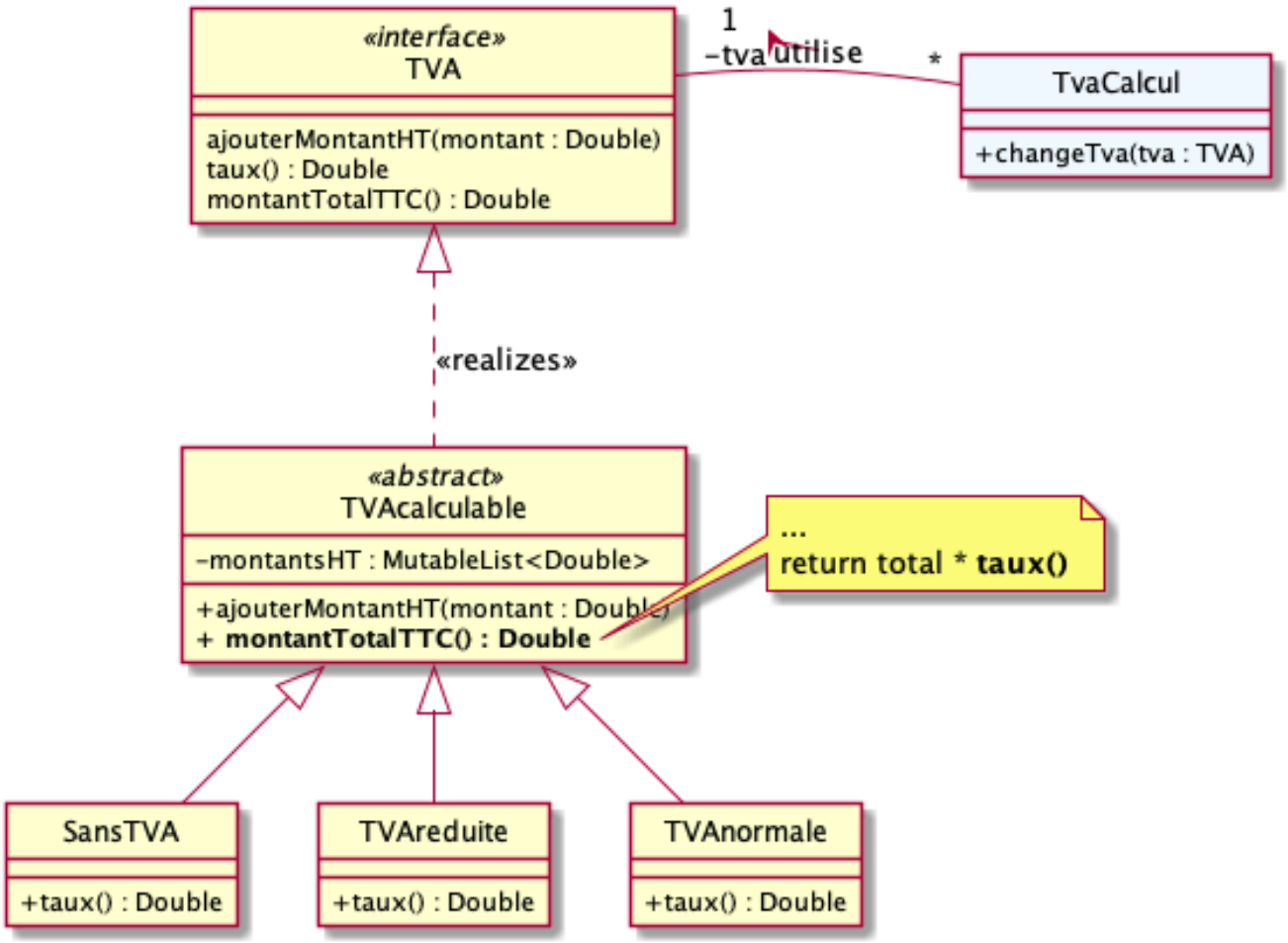


Autre exemple de Patron de méthode (2)



Ici la méthode `montantTotalTTC()` n'est pas `open` pour éviter toute modification de l'algorithme

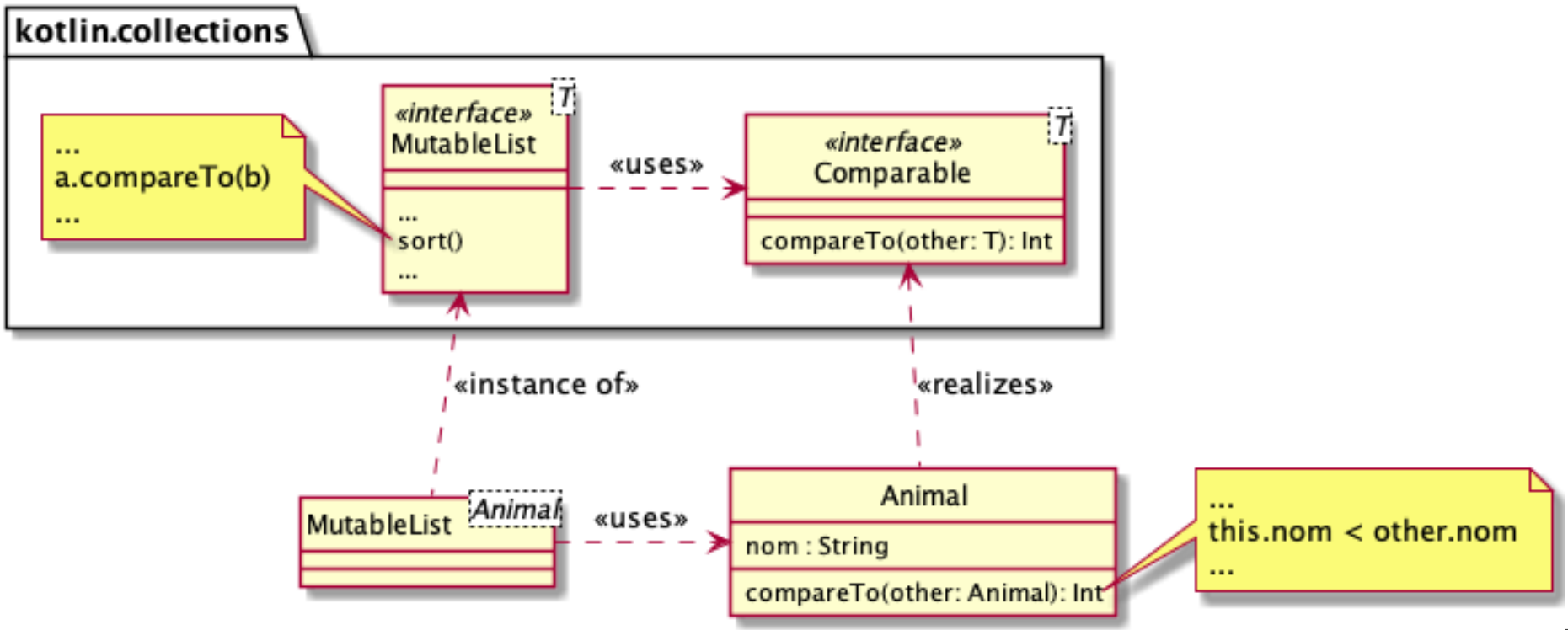
Autre exemple de Patron de méthode (2)



Ici la méthode `montantTotalTTC()` n'est pas `open` pour éviter toute modification de l'algorithme

La méthode Kotlin `sort()`

La méthode `<T : Comparable<T>> MutableList<T>.sort()` utilise le patron de méthode :
Les éléments `T` doivent implémenter `Comparable<T>` pour pouvoir être triés



Patron Stratégie – Strategy Patterns

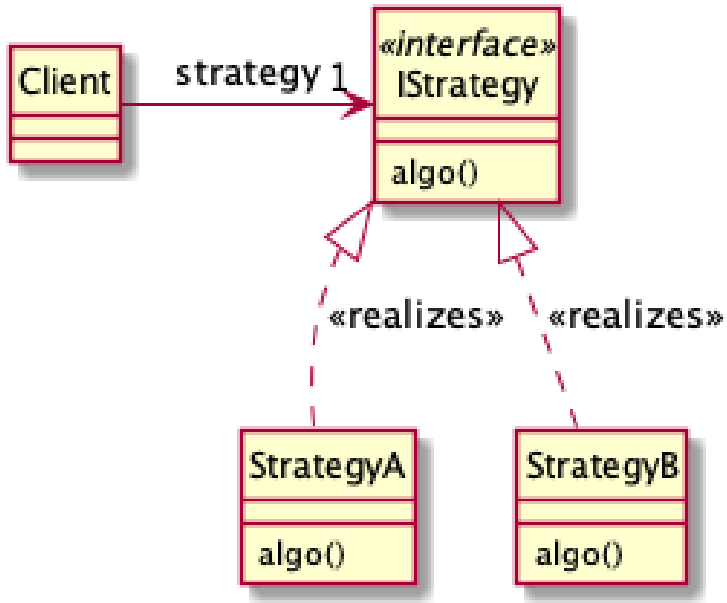
Patron Stratégie – Strategy Patterns

Problème

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

Solution

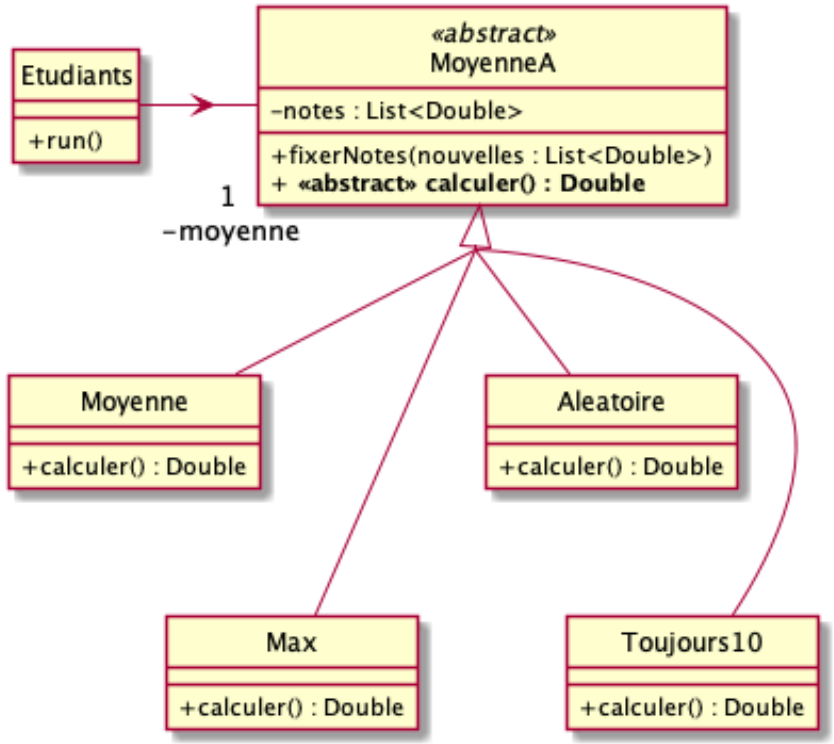
- Définir la famille d'algorithmes grâce à une interface/classe abstraite
- Définir chaque algo par une classe implémentant l'interface
- Le client utilise l'un des algorithmes à un moment donné via un attribut référençant la stratégie



- 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}
```

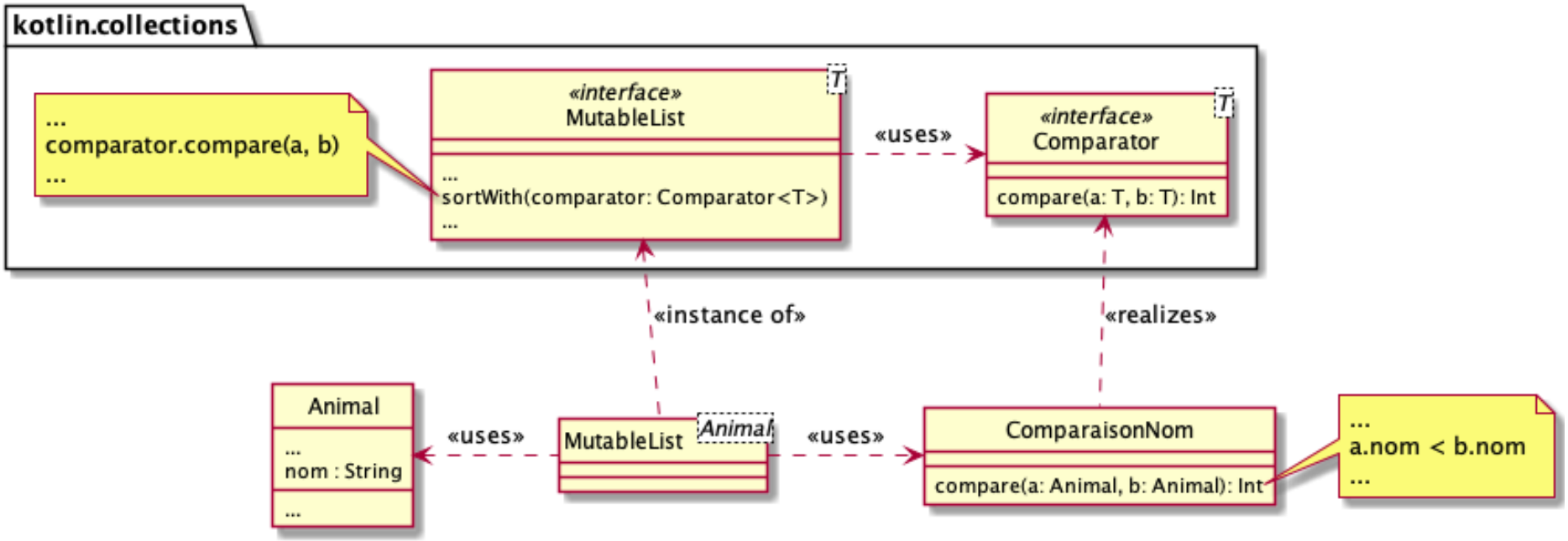
```
class Moyenne() : MoyenneA() {
  override fun calculer() = notes.average()
```

```
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()} ")
}
```

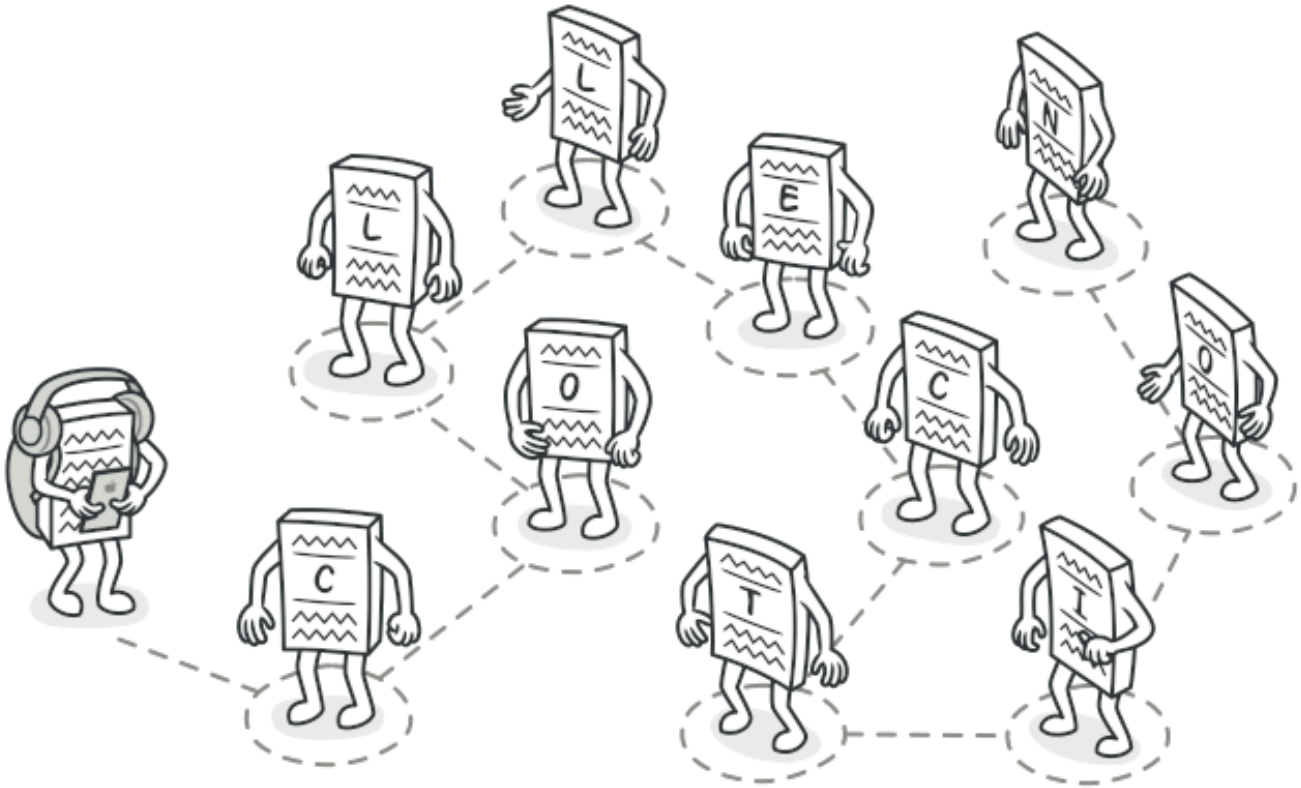
IUT Nantes

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

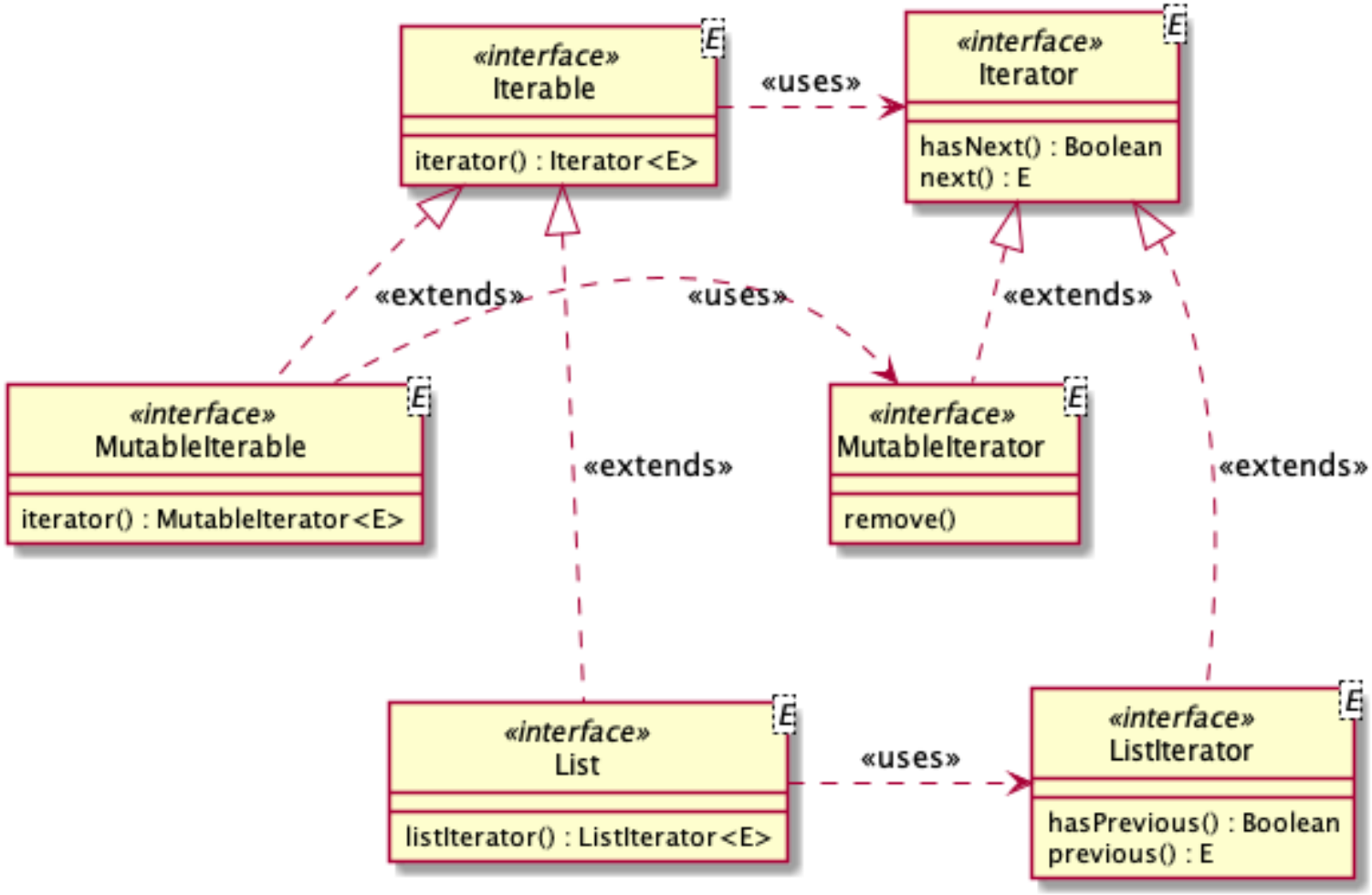
- d'accéder à l'élément courant
- de passer à l'élément "suivant"
- de déterminer si on a tout parcouru

- 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

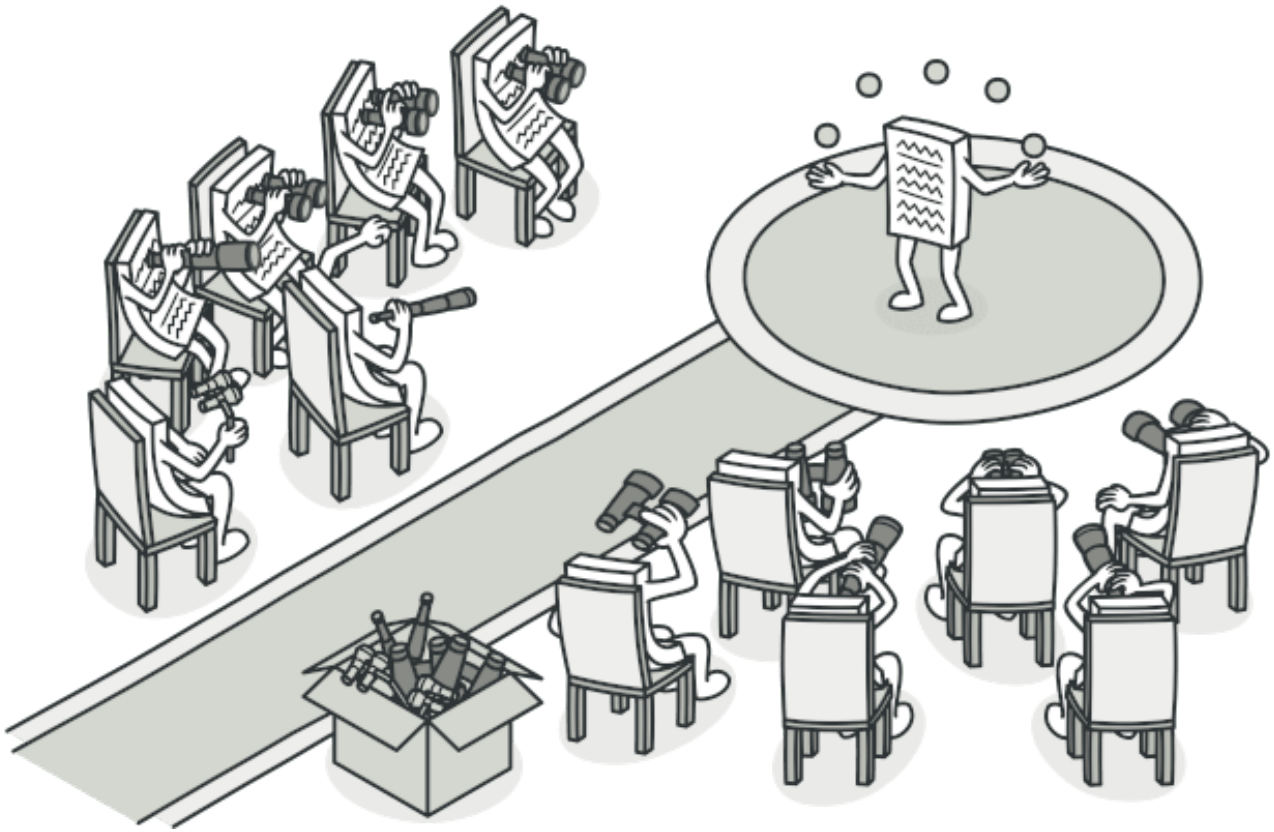
```
for (teacher in teachers) {
  print("$teacher ")
}
```

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



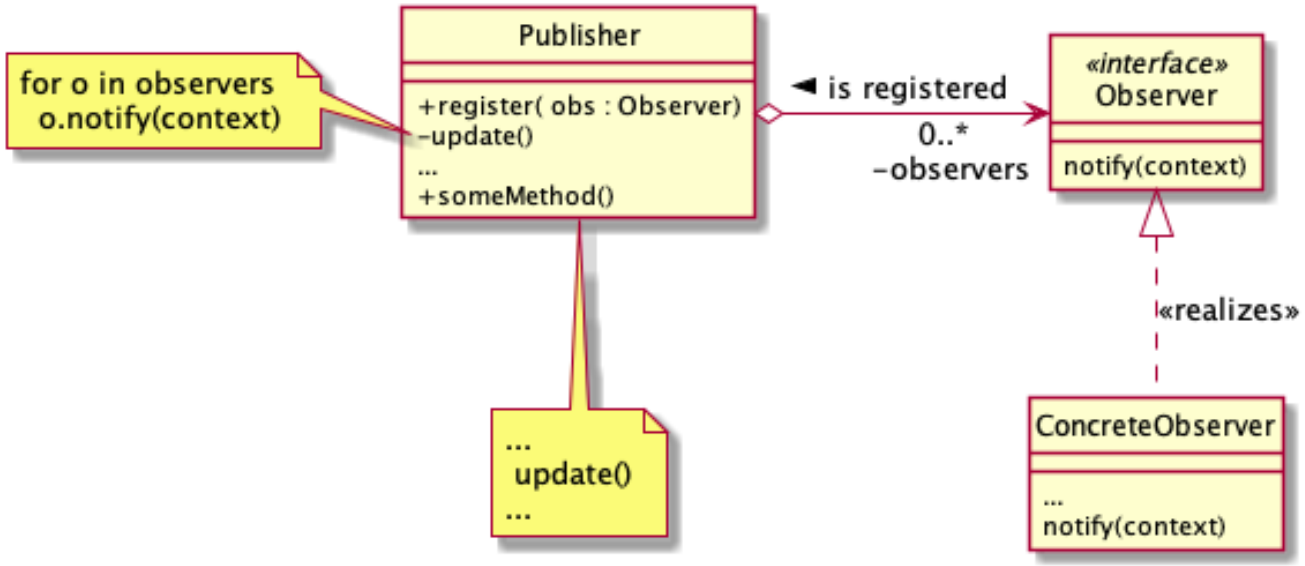
Patron Observateur – Observer pattern

Problème

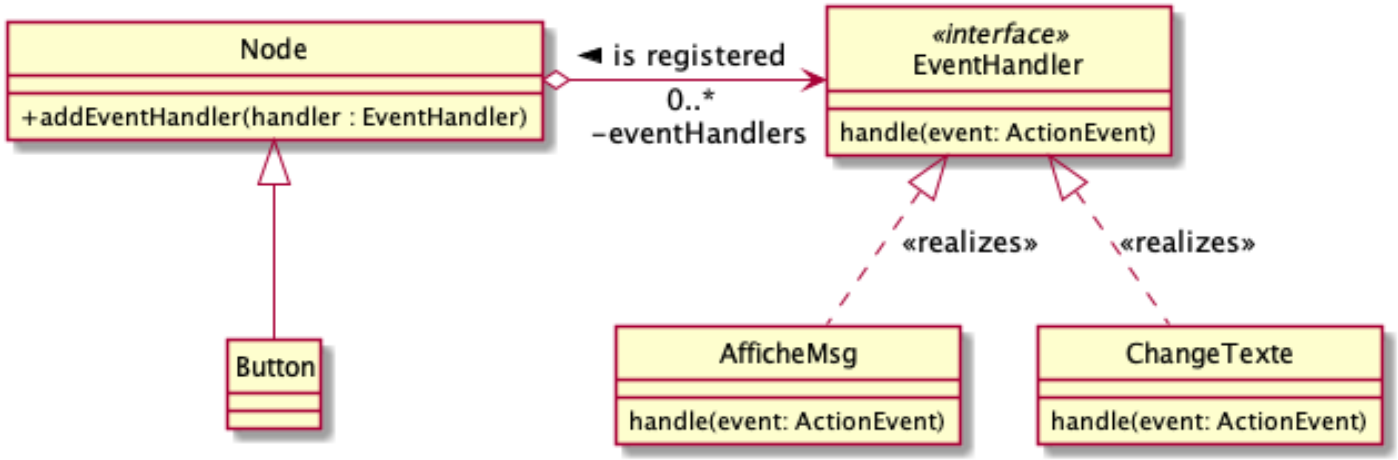
Informer automatiquement d'un changement d'état d'un objet donné (= l'observable), un ou plusieurs autres objets (= les observateurs)

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



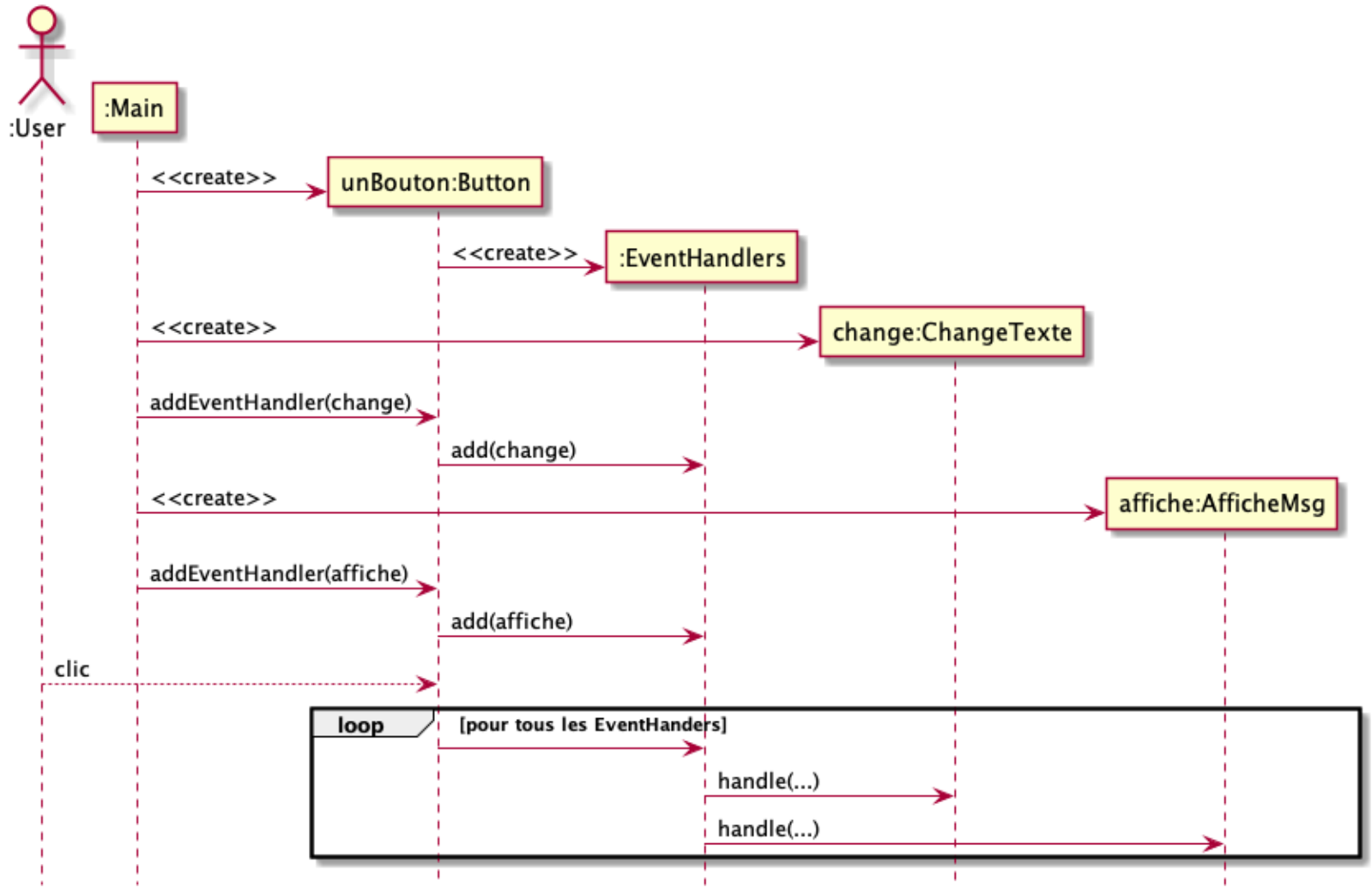

```
...
val unBouton = Button("Go")
val texte = TextField()
unBouton.addEventHandler(ActionEvent.ACTION,
                         ChangeTexte(texte))
unBouton.addEventHandler(ActionEvent.ACTION,
                          AfficheMsg())
...
```

```
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

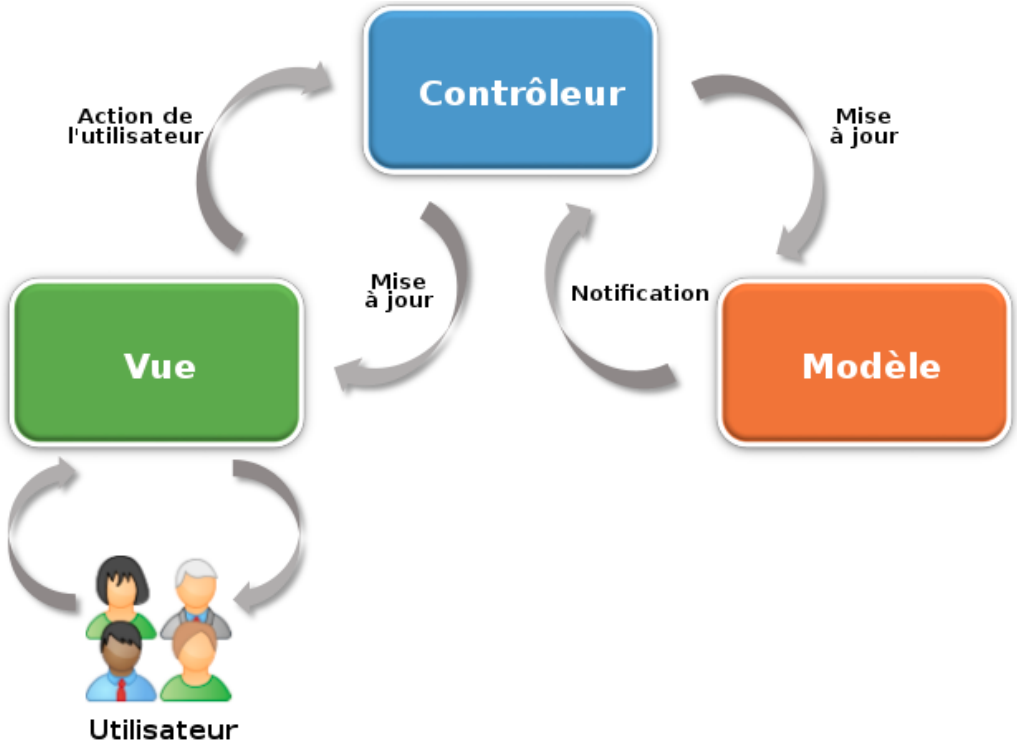
Sommaire

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