R3.04 : Qualité de développement

Patrons de conception

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

Département informatique

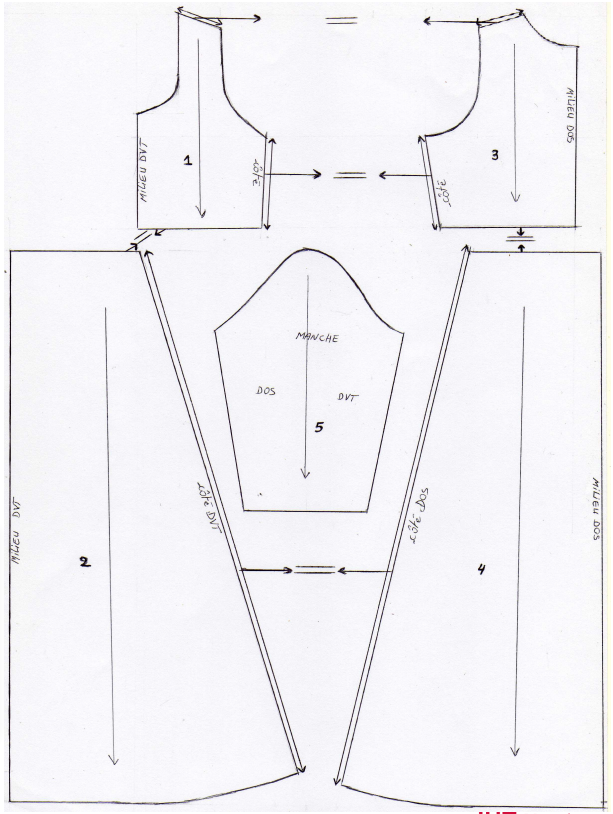
Sommaire

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

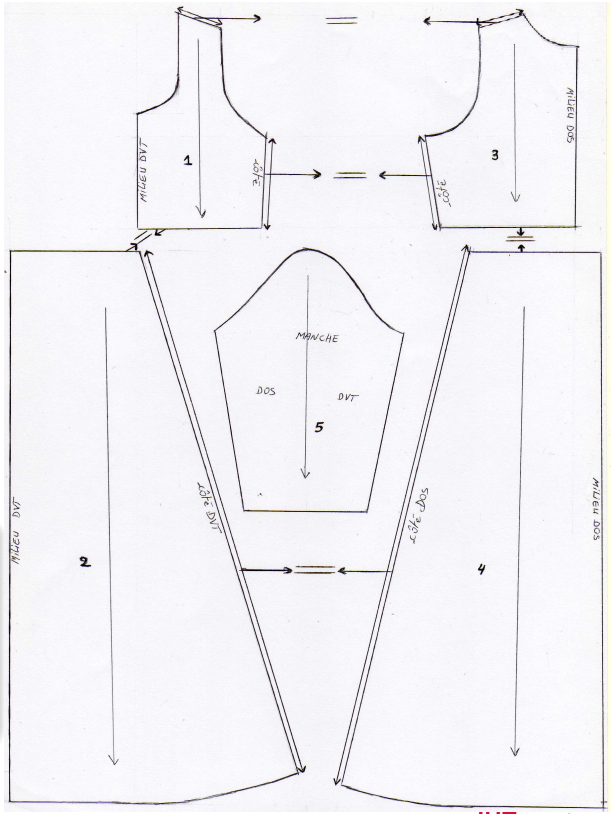
En pratique

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
- Green-patterns = patrons d'éco-conception

Interface + classes

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

Pattern vs. code

Les design patterns sont indépendants du langage de programmation objet utilisé

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
- Green-patterns = patrons d'éco-conception

Interface + classes

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

Pattern vs. code

Les design patterns sont indépendants du langage de programmation objet utilisé

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
- Green-patterns = patrons d'éco-conception

Interface + classes

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

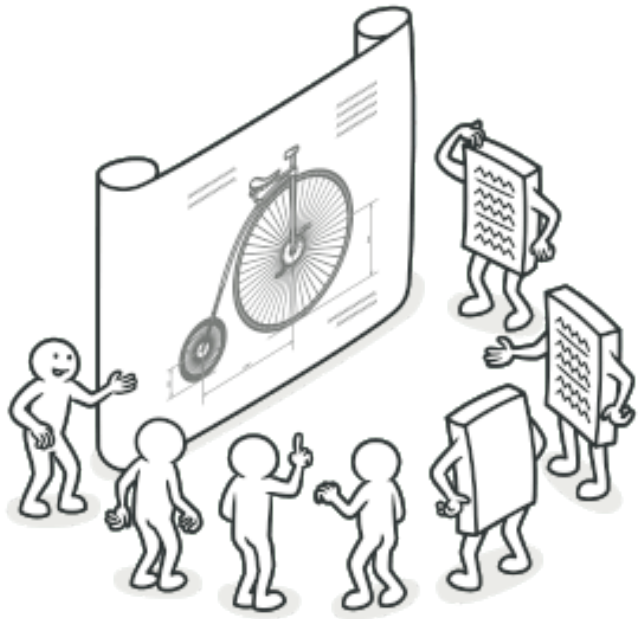
Pattern vs. code

Les design patterns sont indépendants du langage de programmation objet utilisé

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*
- D. Tamzalit 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]

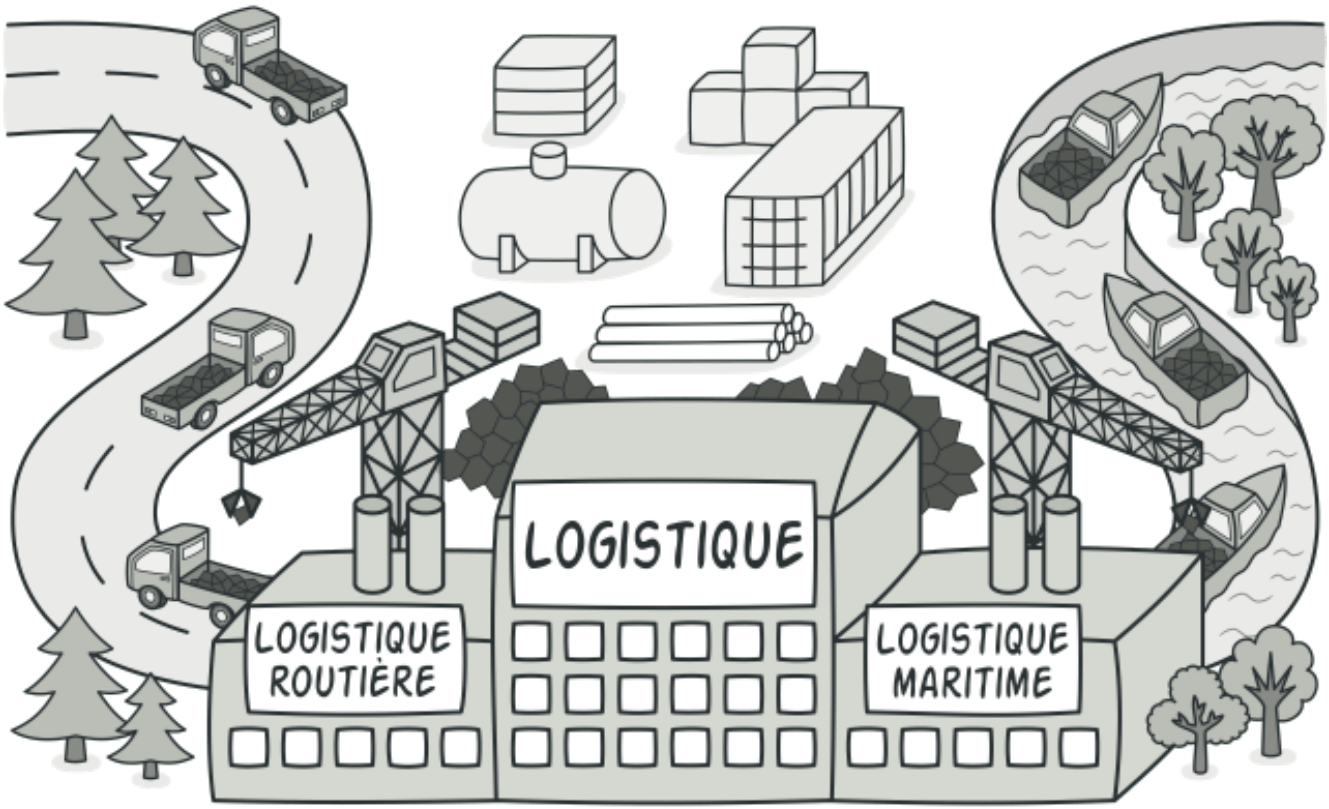


1. Merci également pour les images "humoristiques"

Sommaire

IUT Nantes

Patron Fabrique – Factory pattern



Patron Fabrique – Factory pattern

Problème

Comment construire un objet à partir de paramètres complexes ? Quelle classe instancier parmi une hiérarchie de classes ?

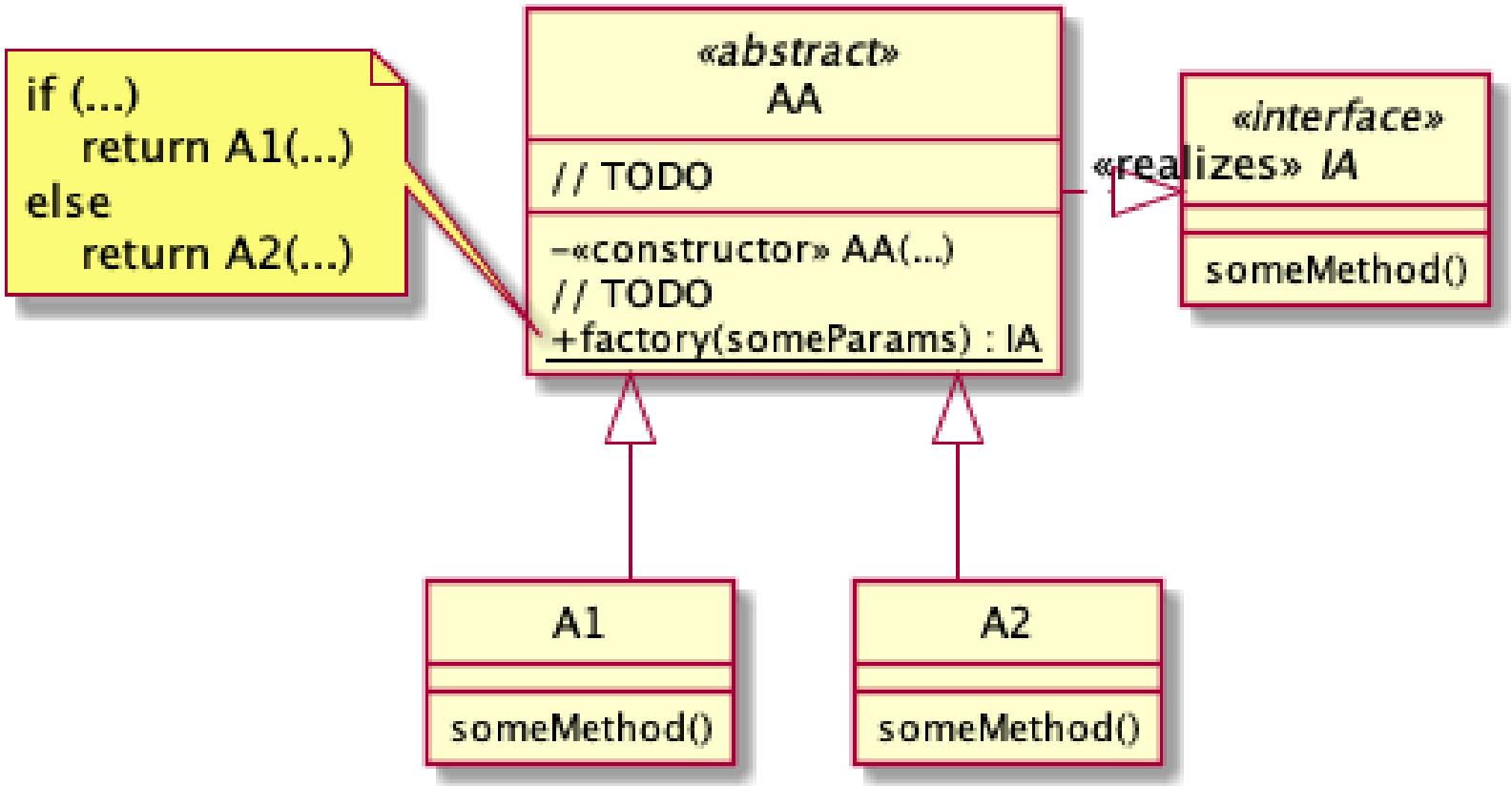
Solution

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

- Il est souhaitable de rendre `private` les constructeurs des classes considérées
- La fabrique permet de séparer la création des objets de leur utilisation.
- La fabrique fourni des noms plus lisibles que les constructeurs.
- Plusieurs fabriques peuvent être regroupées en une fabrique abstraite.

IUT Nantes

Patron Fabrique : schéma UML



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

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)
```

- La méthode `fabrique()` est ici statique.

IUT Nantes

Exemple de fabrique (2)

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

companion object {
    fun fromCartesian(real : Double, imag : Double) =
        Complex(real, imag)

    fun fromPolar(rho : Double, theta : Double) =
        Complex(rho * Math.cos(theta), rho * Math.sin(theta))
}
```

- Ici 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 = r

companion object {
    fun fromCartesian(real : Double, imag : Double) =
        Complex(real, imag)

    fun fromPolar(rho : Double, theta : Double) =
        Complex(rho * Math.cos(theta), rho * Math.sin(theta))
}
```

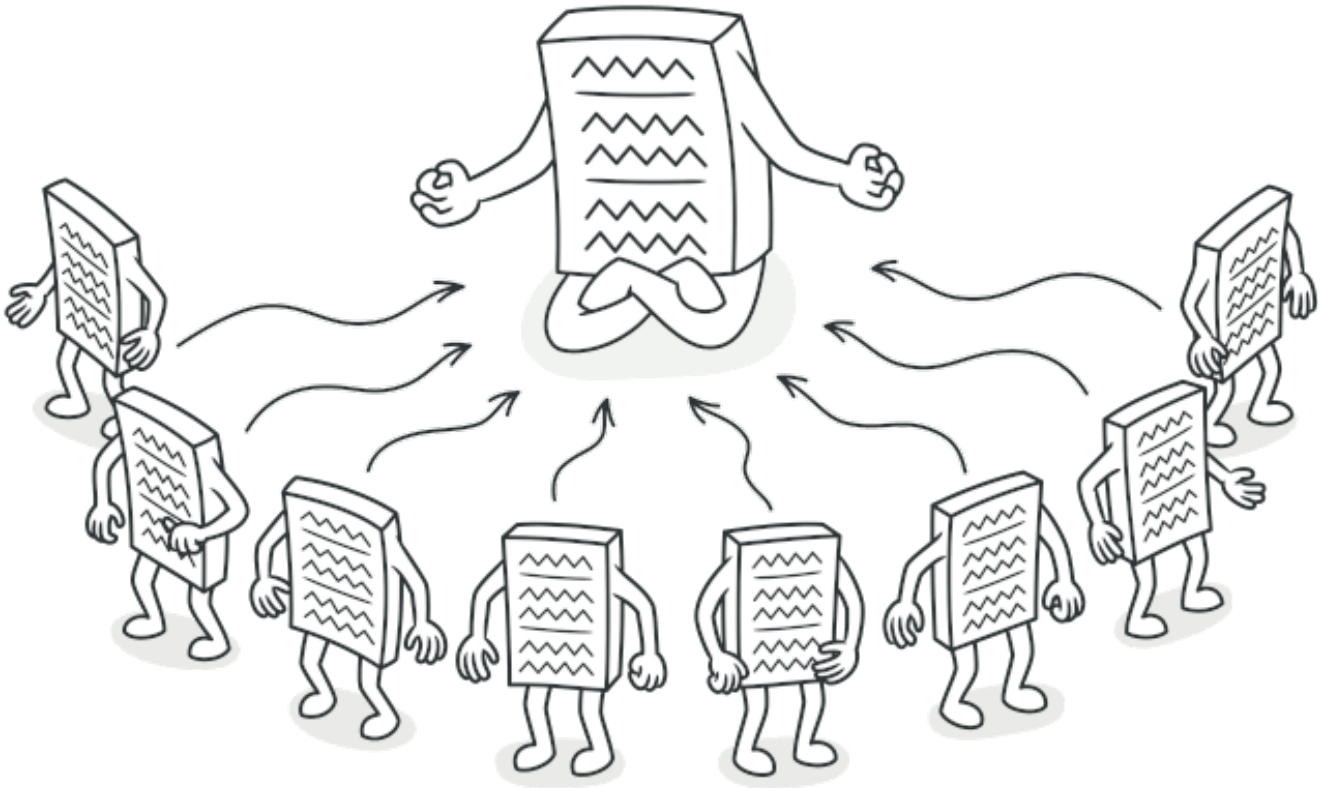
- Ici le constructeur est `private`, ce qui oblige l'utilisation des fabriques

Kotlin

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

IUT Nantes

Patron Singleton – Singleton pattern



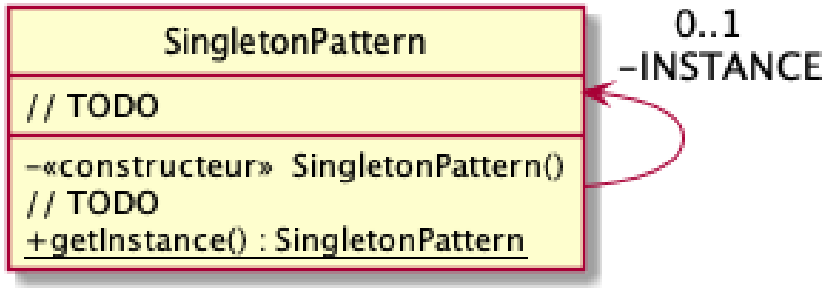
Patron Singleton – Singleton pattern

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() {

  // TODO

  companion object {

    private val INSTANCE
        = SingletonPattern()

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

object

Kotlin propose une construction spécifique de Singleton : `object`

```
object SingletonPattern2 {

  // TODO

}
```

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() {

  // TODO

  companion object {

    private val INSTANCE
        = SingletonPattern()

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

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

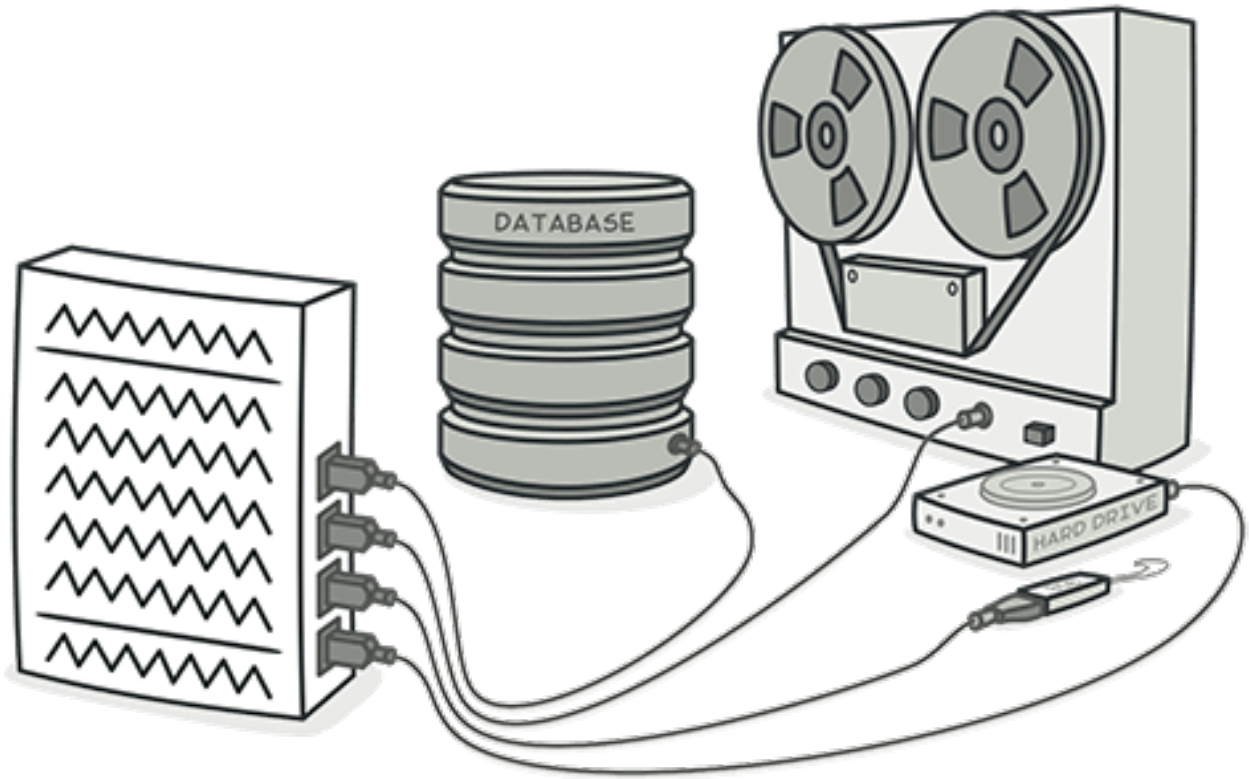
object

Kotlin propose une construction spécifique de Singleton : `object`

```
object SingletonPattern2 {

  // TODO

}
```

IUT Nantes

Patron Objet d'accès aux données – DAO pattern



IUT Nantes

Patron Objet d'accès aux données – DAO pattern

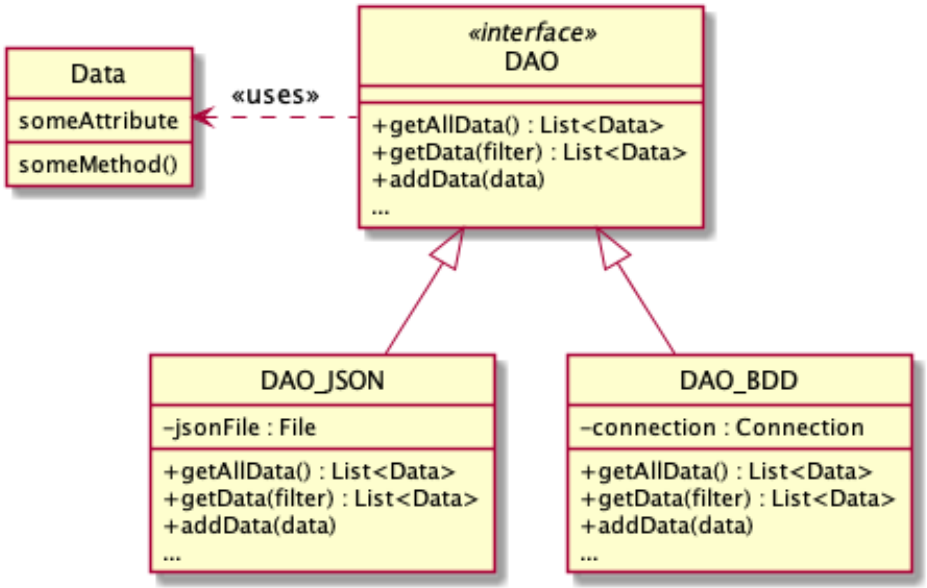
Problème

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 l'accès aux données sans impacter les classes métiers



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

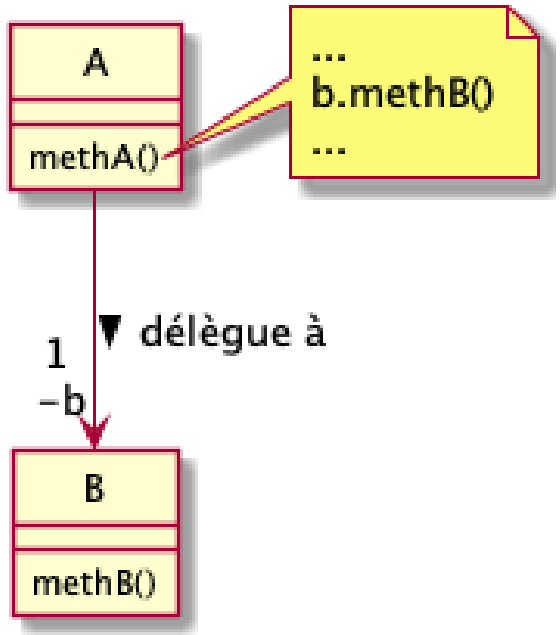
Patron Délégation – Delegate pattern

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`



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

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

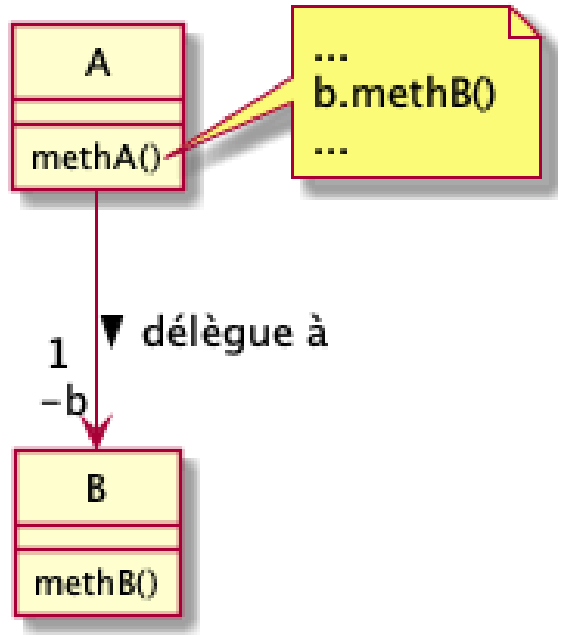
Patron Délégation – Delegate pattern

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

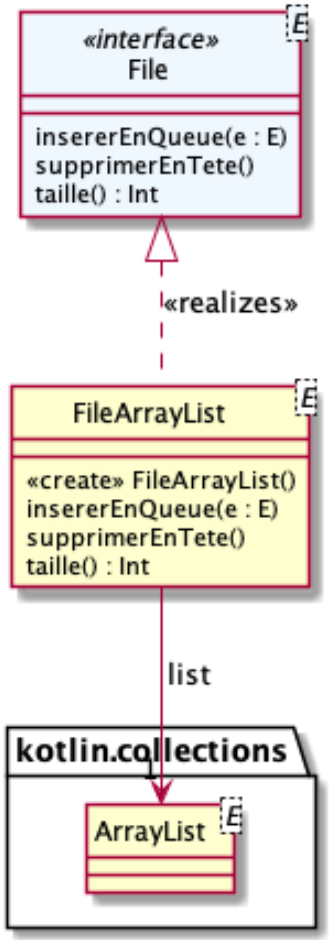
Déléguer à la classe `B` les traitements à réaliser, en ajoutant à `A` un attribut de type B et en appelant les méthodes de `B` dans `A`



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

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

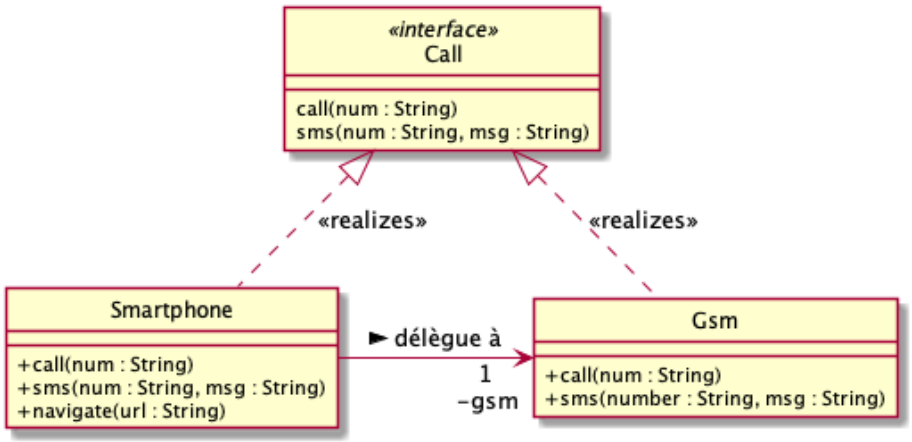
Exemple de délégation




```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 en Kotlin grâce à `by`



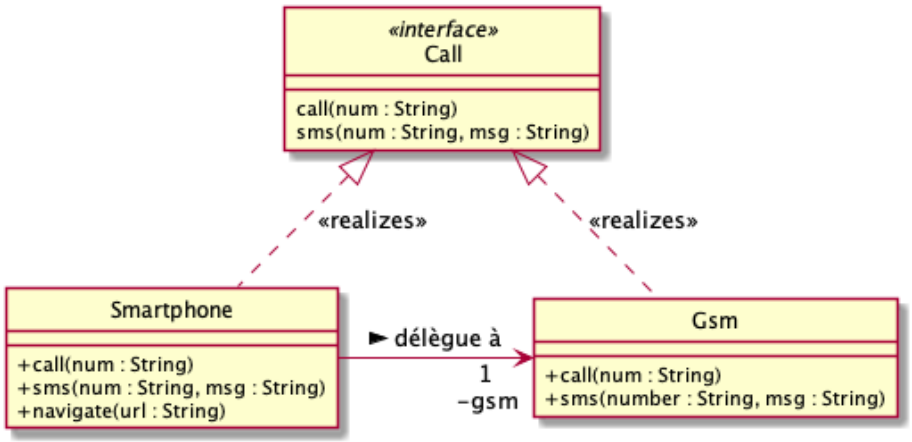

```kotlin
class Gsm : Call {
  override fun call(num: String) {
      println("call $num")
  }

  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`



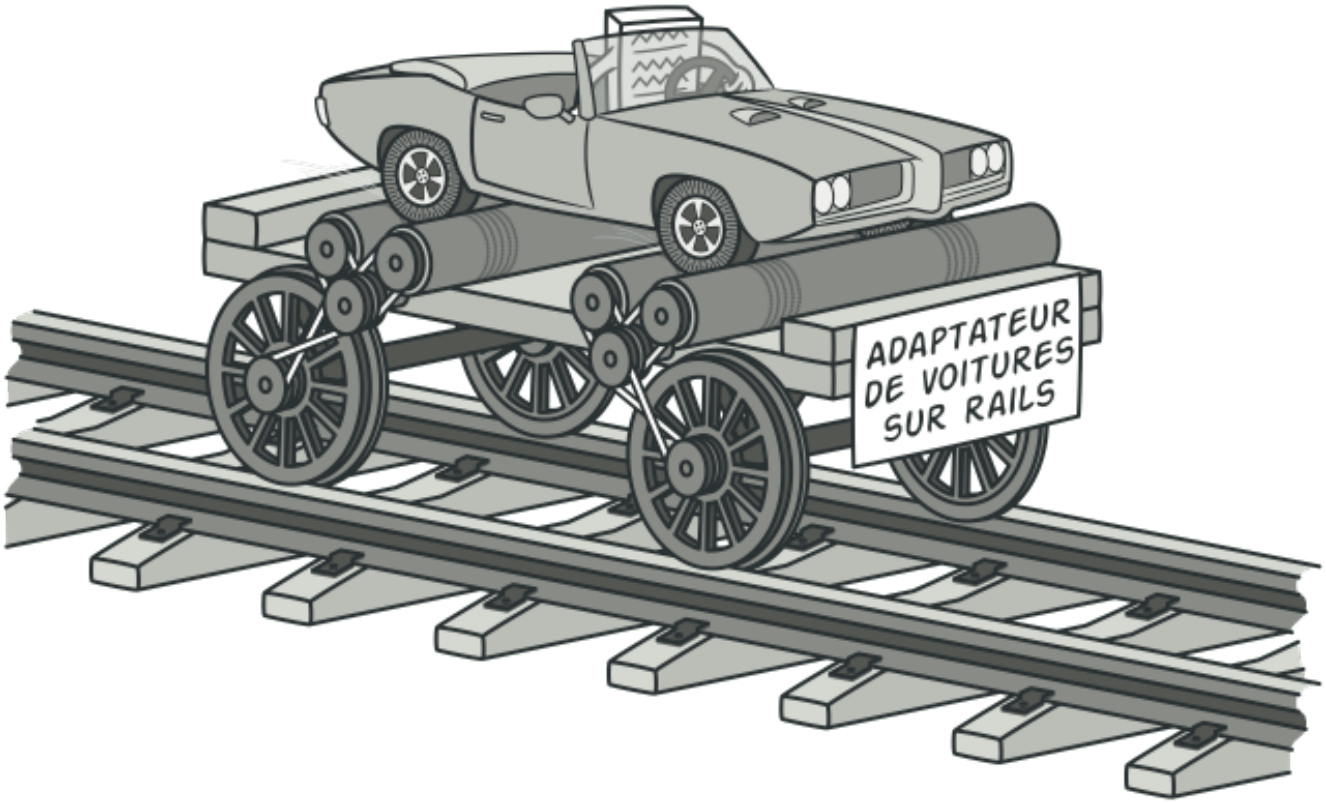

```kotlin
class Gsm : Call {
  override fun call(num: String) {
      println("call $num")
  }

  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



IUT Nantes

Patron Adaptateur – Adapter pattern

Problème

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

Solution

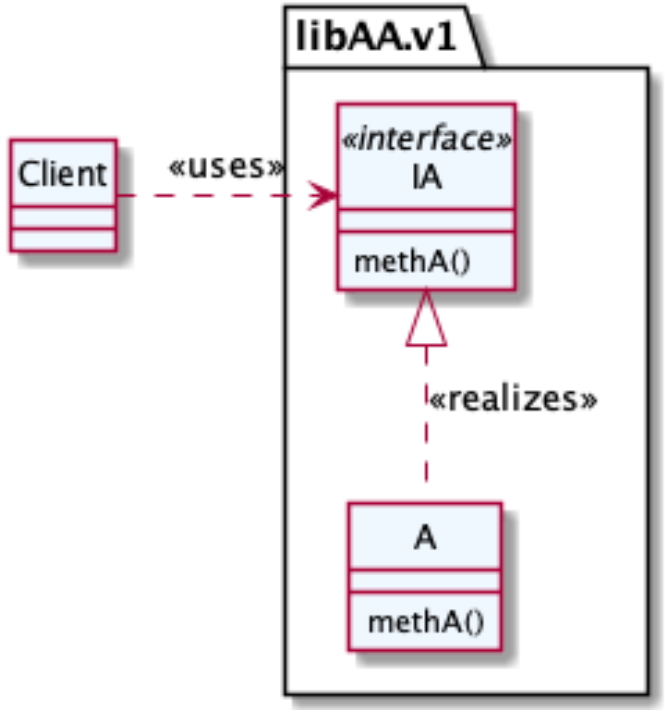
Une nouvelle classe va exposer l'ancienne interface et utiliser les méthodes de la nouvelle pour réaliser l'ancienne interface

- 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.

IUT Nantes

Adaptateur : schéma UML

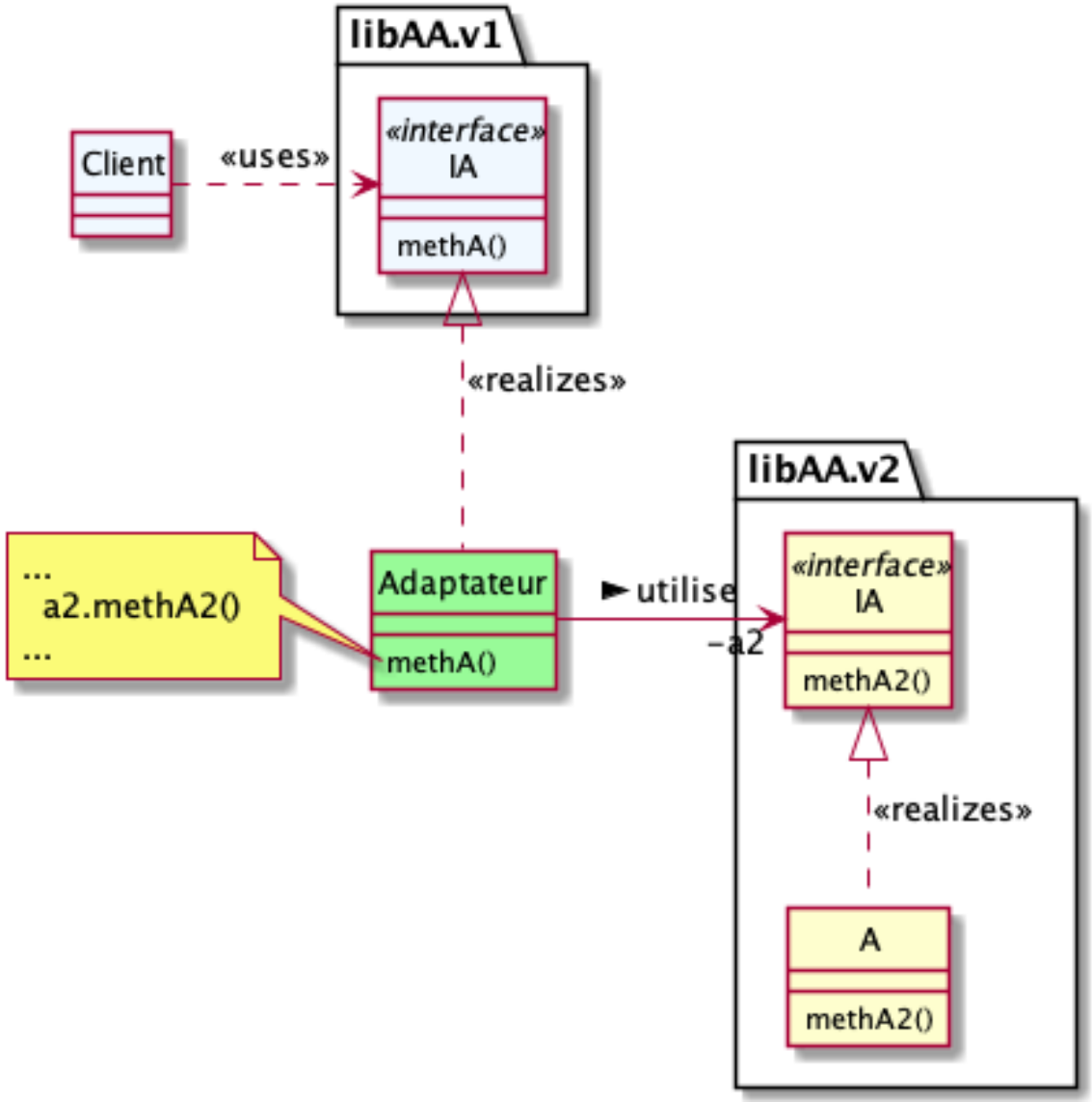
La librairie `libAA` est mise à jour : l'interface d'utilisation a changée



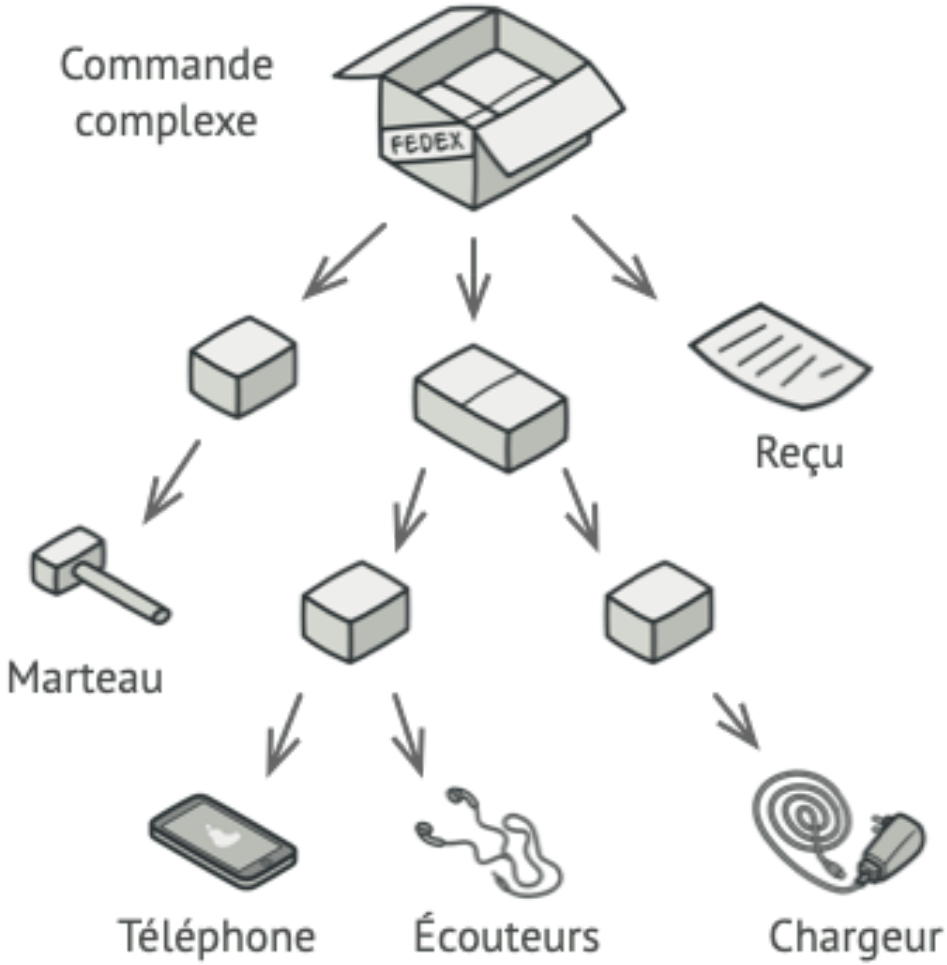
IUT Nantes

Adaptateur : schéma UML

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



Patron Composite – Composite pattern



IUT Nantes

Patron Composite – Composite pattern

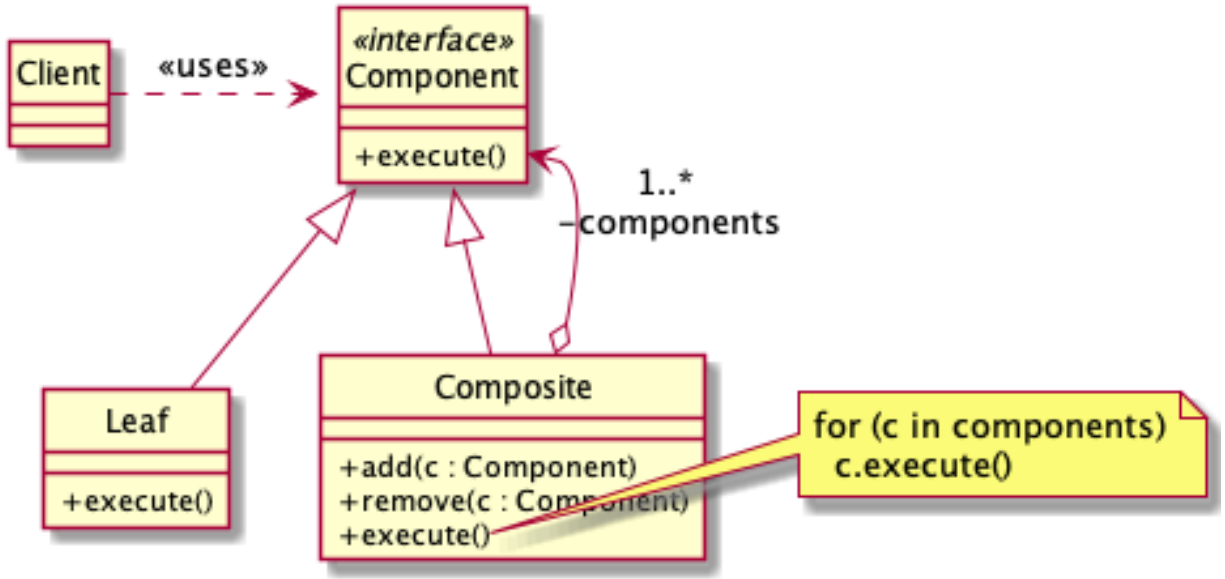
Problème

Comment représenter une structure arborescente ?

Solution

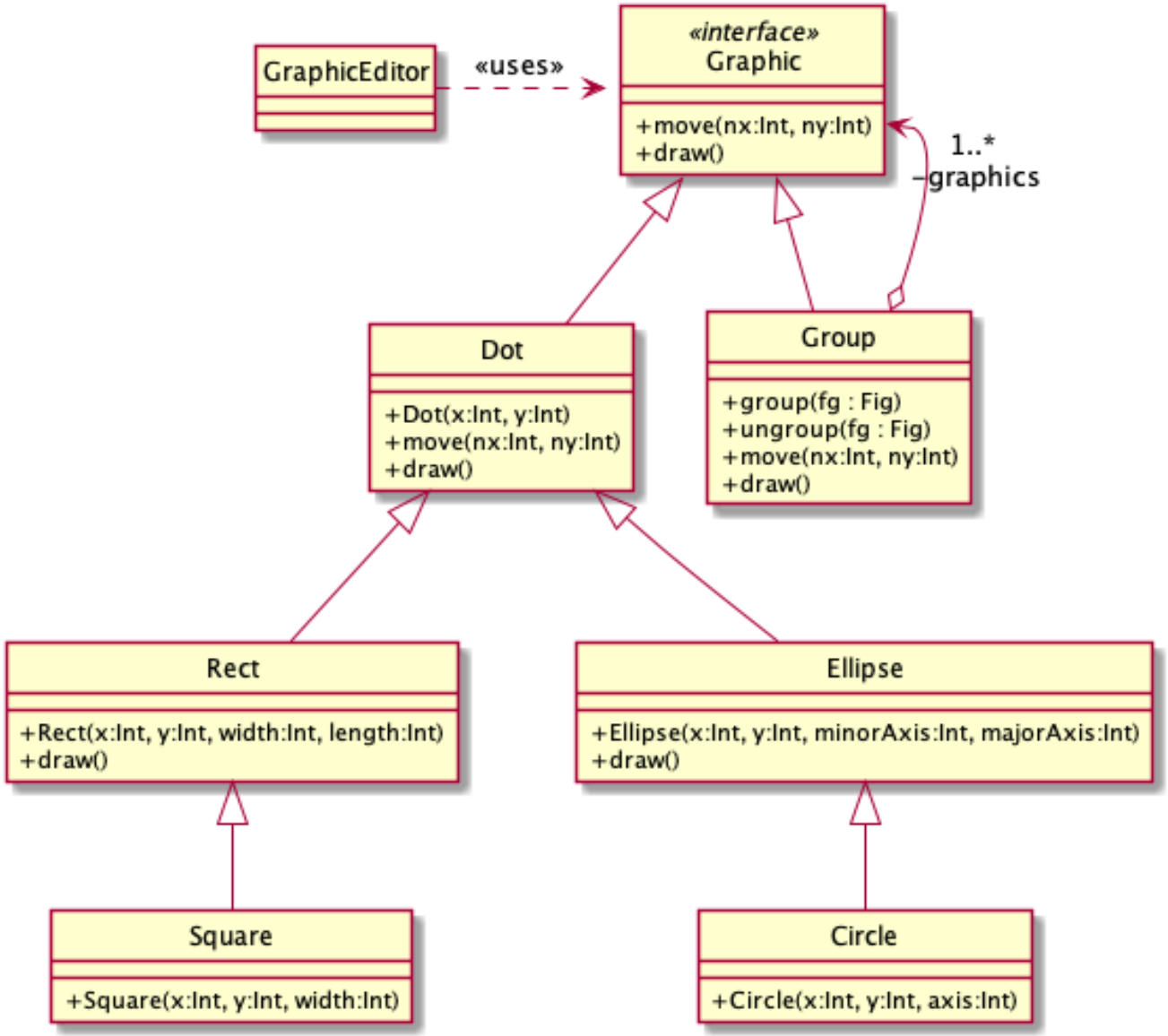
Les composants-feuilles et les composite-conteneurs implémentent une même interface

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



IUT Nantes

Exemple de structure Composite

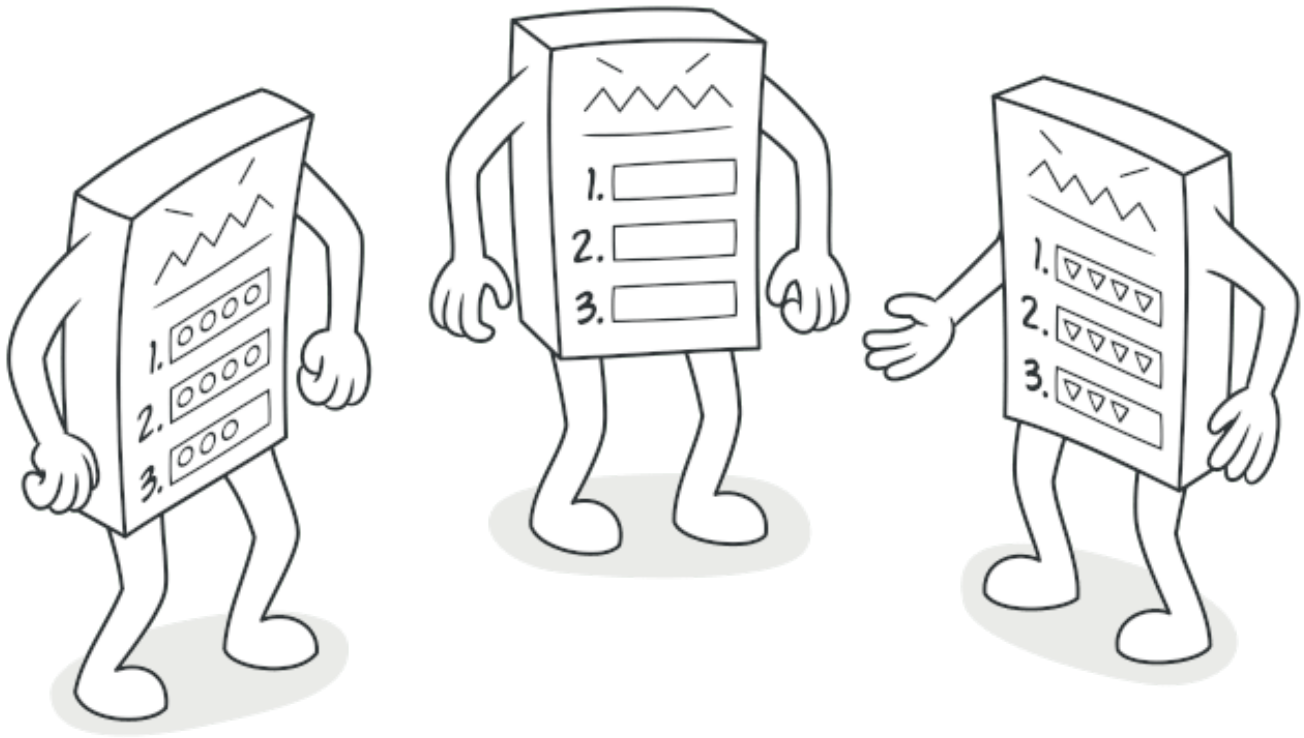


IUT Nantes

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

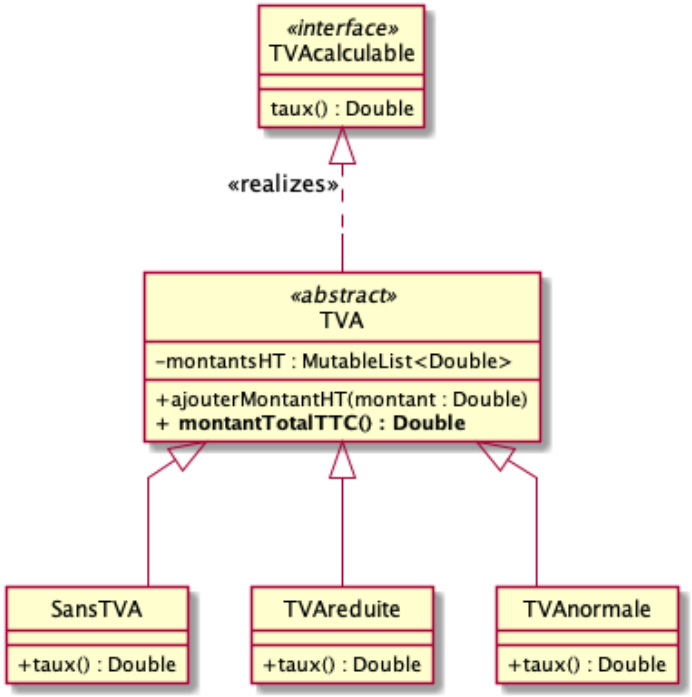
Le patron de méthode propose de

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




```
abstract class TVA : TVAcalculable {
  private val montantsHT
          = mutableListOf<Double>()

  fun ajouterMontantHT(montant : Double) {
    montantsHT+= montant
  }

  fun montantTotalTTC() : Double {
    var total = 0.0
    for (montant in montantsHT) {
        total += montant
    }
    return total * taux()
  }
}
```

```
interface TVAcalculable {
  fun taux() : Double
}
```

```
class SansTVA : TVA() {
  override fun taux() = 0.0
}
```

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

```
class TVAreduite : TVA() {
  override fun taux() = 5.5
}
```

IUT Nantes

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