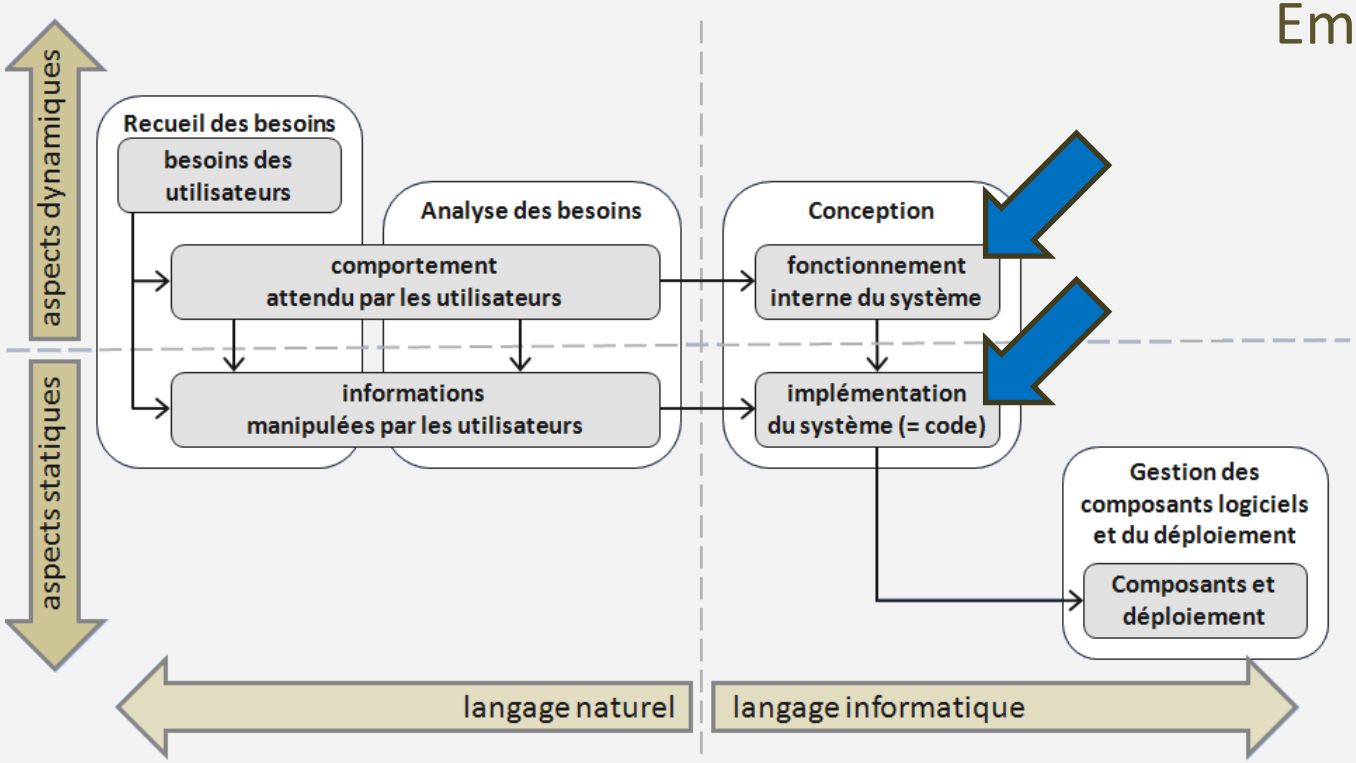
UML
Conception
Encapsulation et interfaces

Emmanuel Pichon
2013



Principe d'encapsulation

- **LE** principe « objet » à connaitre
  - Un objet est une boîte noire qui offre des services
    - Distinction entre comportement et implémentation d'un objet
    - Un objet est responsable du son bon fonctionnement

- Objectif
  - Utiliser les services d'un objet sans connaître son implémentation
  - Pouvoir changer son implémentation sans impacter les services offerts aux autres objets

- Principe applicable à tous niveaux : système, sous-système, classe, …

Principe d’encapsulation en conception

- Objectifs
  - Garantir l’intégrité des données de chaque objet
  - Réduire les adhérences entre les objets (couplage faible)
  - Garantir l’agilité du logiciel face aux évolutions
  - Minimiser les impacts d’une modification
  - Faciliter la correction des bugs
  - Faciliter la réutilisabilité du code logiciel

  = concevoir « objet »

Principe d'encapsulation en conception

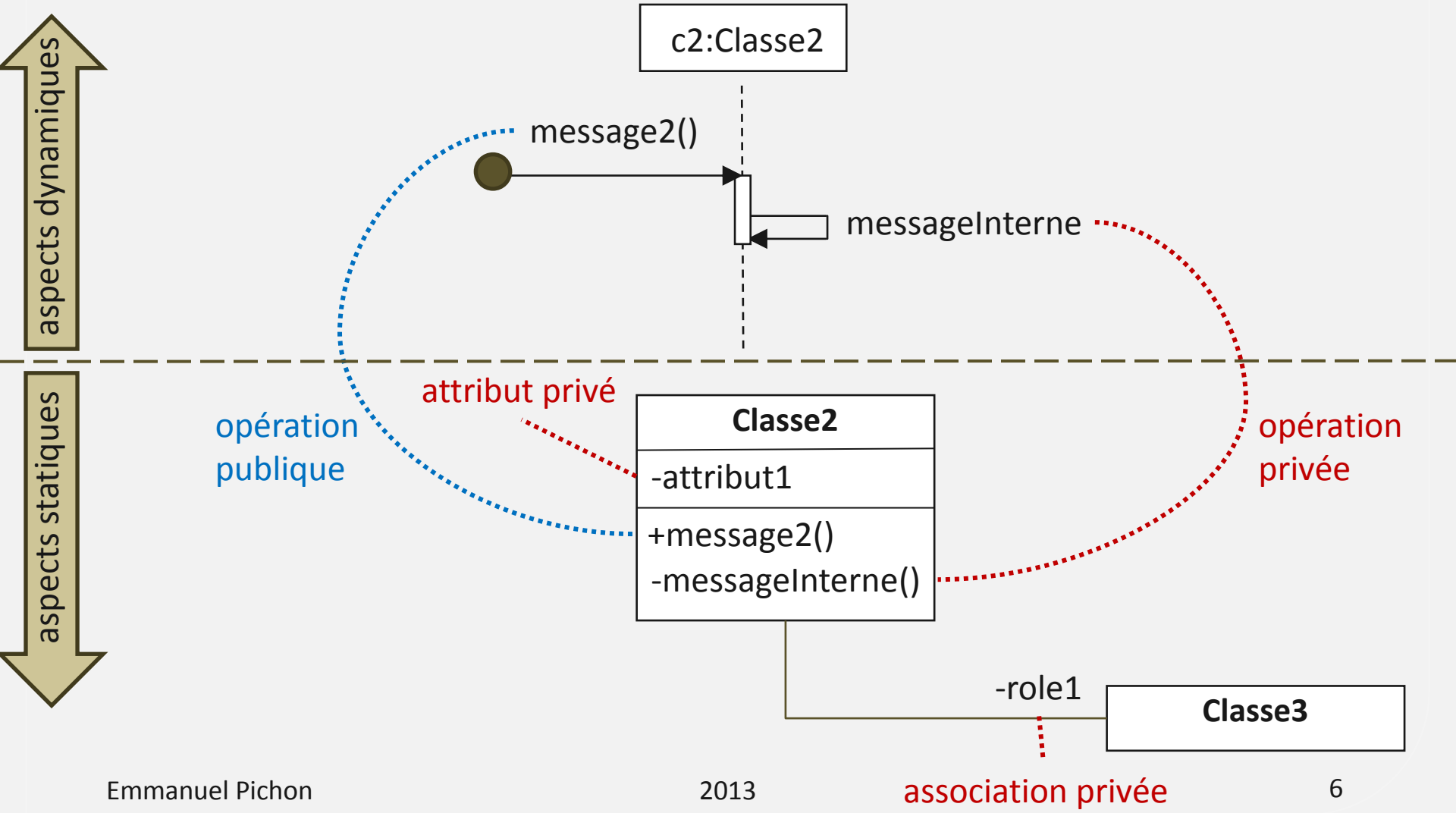
- En conception objet, on distingue
  - L'interface d'un objet
    - Attributs, opérations et associations visibles depuis l'extérieur de l'objet
    - Visibilité publique indiquée par un plus (+) en UML
    - Visibilité package indiquée par un tilde (~) en UML
  - L'implémentation interne d'un objet
    - Attributs, opérations et associations invisibles depuis l'extérieur de l'objet
    - Visibilité privée indiquée par un moins (-) en UML

Application du principe d'encapsulation

- Objectif
  - Limiter les accès depuis l'extérieur d'un objet
  - = limiter les dépendances entre les objets

- Interface minimale d'un objet par rapport aux autres objets
  - Les opérations visibles de l'extérieur correspondent aux messages définis dans les diagrammes d'interaction
  - Les autres opérations sont privées
  - Les attributs et les rôles sur les associations sont privés
    - = pas d'accès aux données depuis l'extérieur d'un objet

Illustration du principe d'encapsulation

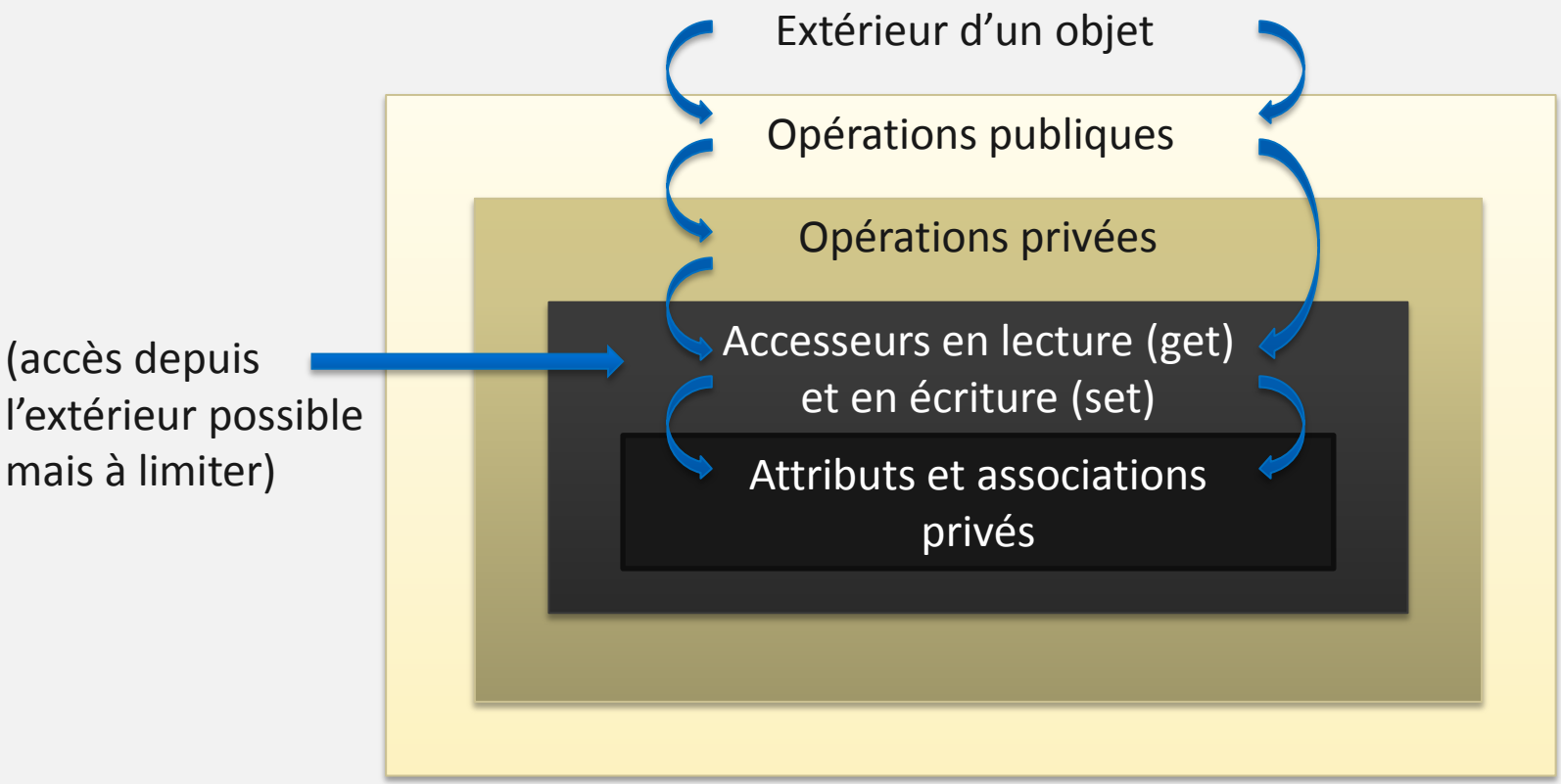


Application du principe d'encapsulation

- Deuxième objectif
  - Limiter les accès aux données depuis l'intérieur d'un objet
  - = garantir l'intégrité des données de chaque objet

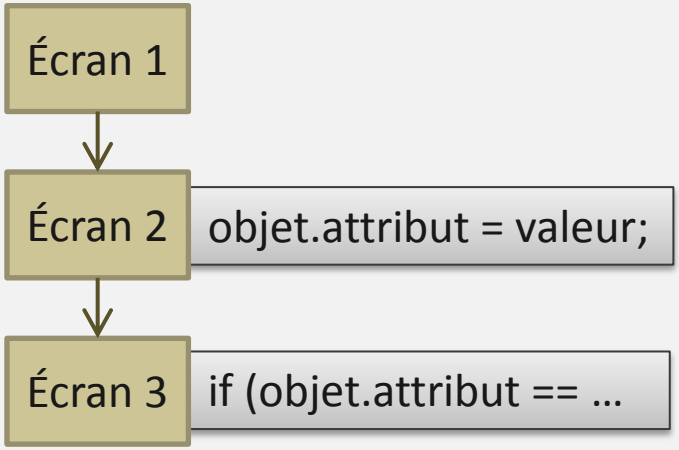
- Encapsulation des données (attributs et associations)
  - Pas d'accès direct aux données même pour les opérations de la classe détenant ces données
  - Utilisation d'un accesseur en lecture par donnée
    - Gestion de l'initialisation de cette donnée
  - Utilisation d'un accesseur en écriture par donnée
    - Gestion de l'intégrité de cette donnée
  - NB : limiter les accesseurs publics (en lecture et en écriture)

Illustration de l'encapsulation des données



Exemple classique de non encapsulation

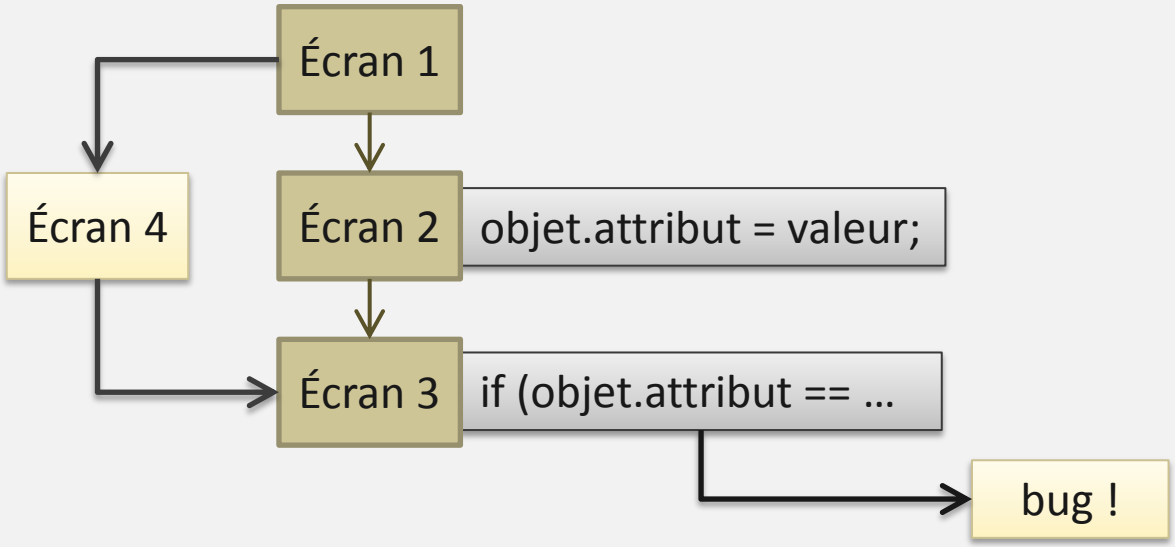
- Initialisation des données dans les écrans
  - Exemple : une donnée est initialisée dans l'écran 2



  - Et elle est utilisée dans l'écran 3

Exemple classique de non encapsulation

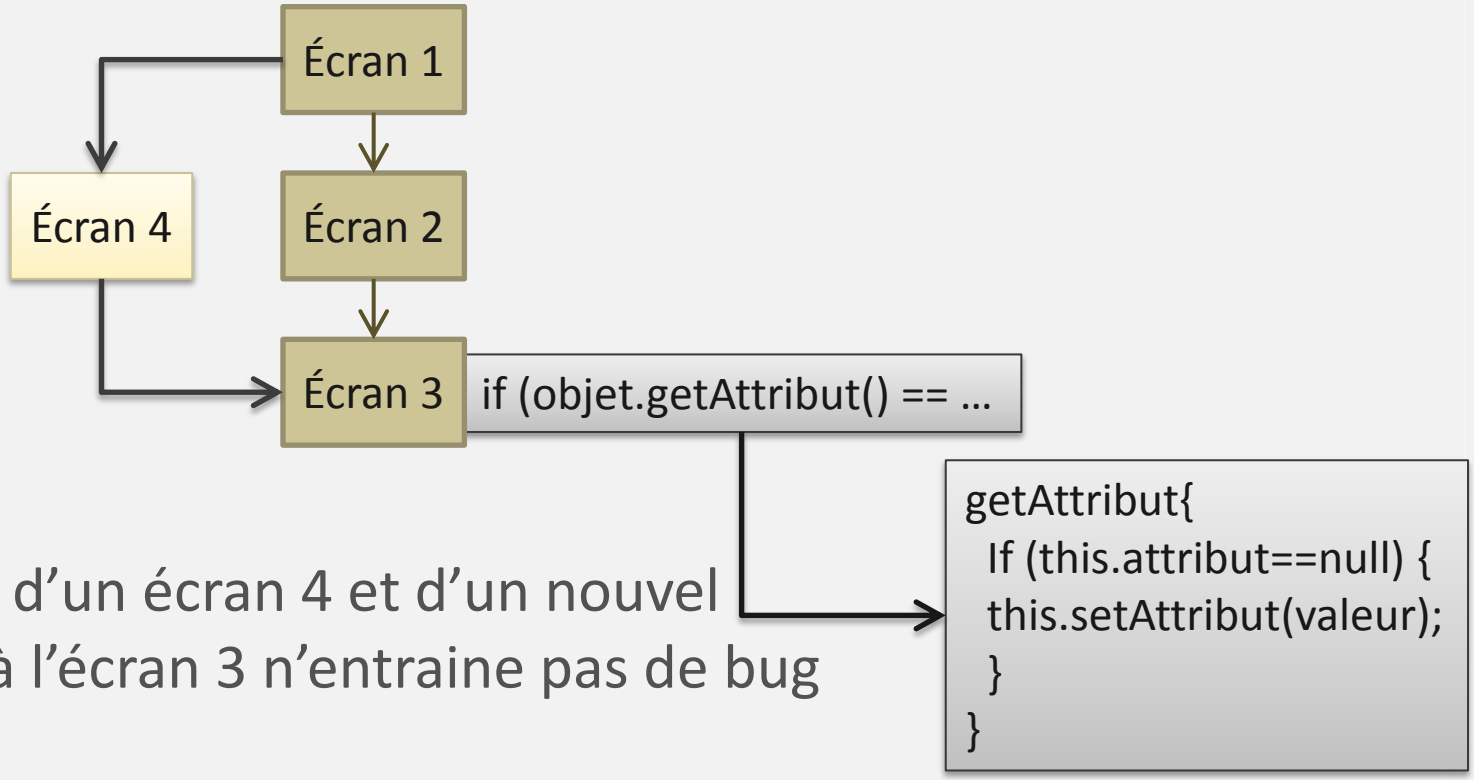
- Initialisation des données dans les écrans
  - Exemple : une donnée est initialisée dans l'écran 2



  - L'ajout d'un écran 4 et d'un nouvel accès à l'écran 3 entraine un bug car l'attribut n'est pas initialisé

Exemple classique d'encapsulation

- Initialisation des données dans les accesseurs en lecture (get)
  - Exemple :



  - L'ajout d'un écran 4 et d'un nouvel accès à l'écran 3 n'entraine pas de bug

L’encapsulation s’applique aussi dans les getteurs

- Le getteur prend en charge le contrôle de la valeur de l’attribut
- Si la valeur n’est pas initialisée le getteur appelle le setteur en passant la valeur par défaut de l’attribut (constante ou résultat issu de l’appel d’une opération)
- Le setteur contrôle la valeur passée en paramètre et met à jour l’attribut si la valeur
- Le setteur ne doit pas appeler le getteur (boucle infinie)

Limites d'application du principe d'encapsulation

- L'intérêt réel n'apparait qu'en cas d'évolution ou de réutilisation
  - L'évolutivité justifie à elle seule l'intérêt de ce principe
    - Les évolutions arrivent toujours plus tôt que prévu…
  - La réutilisation nécessite en plus de connaitre ce qui est réutilisable dans les librairies Java et dans le projet
    - Nécessité de communiquer sur les éléments réutilisables

- Les accès aux données hors langage objet ne respectent pas le principe d'encapsulation
  - Par exemple, accès SQL à une base de données
  - Les accesseurs en lecture peuvent encapsuler certains contrôles lors de la lecture des données à partir de la base de données (valeurs incorrectes ou incohérentes)

Le principe d'encapsulation est-il respecté ?

- Exemple d'accesseurs utilisés en cascade pour accéder à un objet puis pour lui demander de faire quelque chose

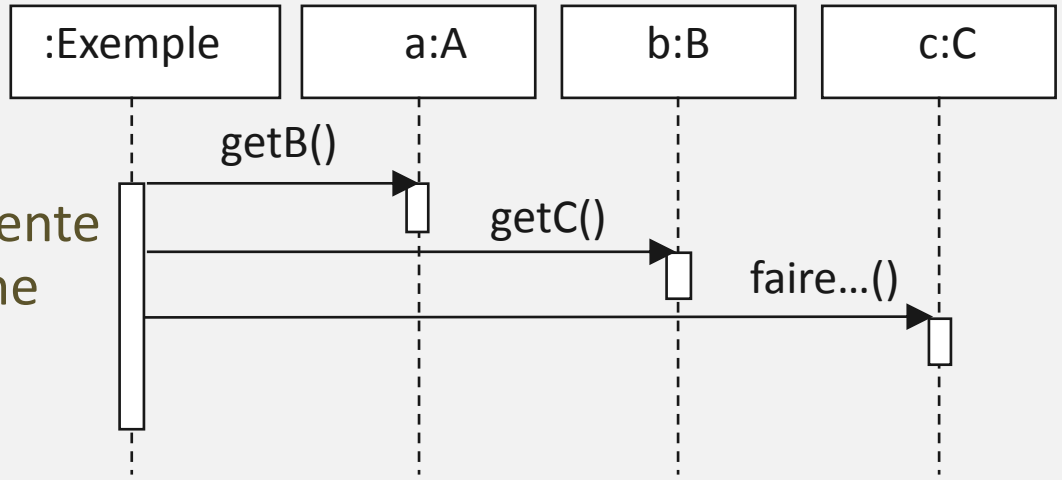
```
class Exemple{
 … { …
  a.getB().getC().faire…();
  …
```

- NB : ce genre de code est souvent copié / collé plusieurs fois (à chaque fois qu'on a besoin d'accéder à l'objet c:C…)

Réponse : localement oui mais globalement non
Le code est lié à la structure des classe

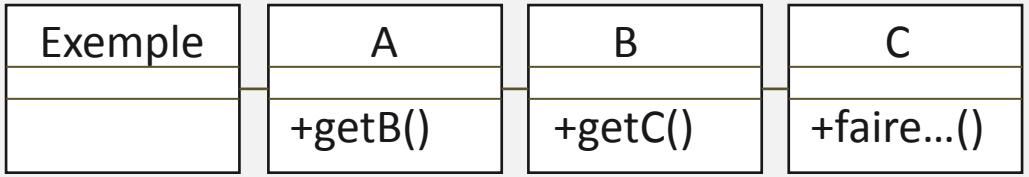
aspects dynamiques

- Le code de la page précédente correspond à ce diagramme



aspects statiques

- Exemple de structure correspondante
  - getB et getC doivent être publics dans ce cas

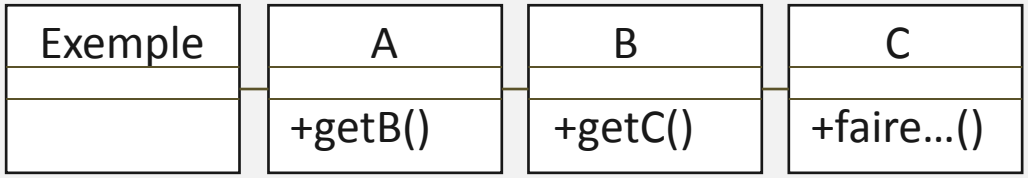


- La classe Exemple est dépendante des classes A, B et C

NB : on doit connaitre la structure des classes pour savoir coder
`a.getB().getC().faire…();`

Comment appliquer le principe d'encapsulation dans ce cas ?

- Hypothèse de travail = on conserve
  - Les mêmes classes
  - Les mêmes associations

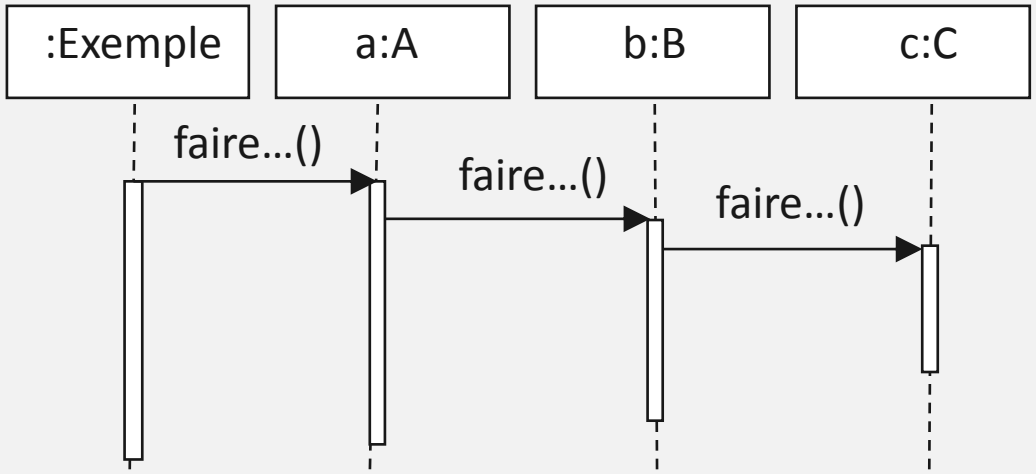


- On peut
  - Ajouter des opérations dans les classes
  - Modifier la visibilité des opérations
  - Modifier l'enchainement des envois de message

Application du principe d'encapsulation

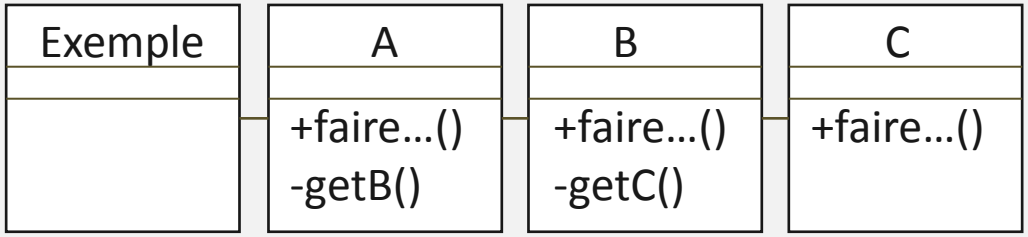
aspects dynamiques

- Demander de faire l'opération souhaitée à l'objet auquel on peut accéder
- Il fera de même…



aspects statiques

- Exemple de structure correspondante
  - Les getB et getC sont encapsulés (privés)
- La classe Exemple n'est plus dépendante que de la classe A



Application du principe d'encapsulation

- Sans encapsulation

```
class Exemple{
 … { …
  a.getB().getC().faire…();
  …
```

- Avec encapsulation

```
class Exemple {
 … { …
  a.faire();
…
```

```
class A {
 faire(){…
  this.getB().faire();
…
```

```
class B {
 faire(){…
  this.getC().faire();
…
```

Exemple comparatif
Suppression de la classe B

- Sans encapsulation

```
class Exemple{
 … { …
  a.getC().faire…();
  …
```

- Avec encapsulation

```
class Exemple {
 … { …
  a.faire();
…
```

```
class A {
 faire(){…
  this.getC().faire();
…
```

- Autant d'impacts que d'utilisation de la classe B

  = Les classes Exemple, …

- Autant d'impacts que d'utilisation de la classe B

  = Seulement la classe A

Impact du principe d'encapsulation sur le code du logiciel

- Inconvénients (pour ceux qui ne sont pas habitués…)
  - Le code est plus morcelé
    - On ne voit pas le code d'un traitement d'un seul coup d'œil
  - Plus de code dans un premier temps

- Avantages
  - Moins de code à chaque réutilisation
  - Facilite le développement à plusieurs personnes et la réutilisation
    - Un concepteur peut utiliser un objet qu'un autre concepteur a conçu sans connaître l'implémentation interne de cet objet
  - Facilite la maintenance, l'évolutivité et le *refactoring*
    - Une modification locale du besoin entraîne une modification locale de la conception et du code
    - Une modification de la structure du logiciel n'aura que des impacts limités

Encapsulation au niveau d'un package

- Les éléments contenus dans un package peuvent avoir une visibilité
  - Publique
  - Privée
  - (les autres visibilités ne sont pas autorisées à ce niveau)

- Seuls les éléments publics sont accessibles depuis l'extérieur du package
  - L'accès nécessite l'usage d'un nom qualifié *(qualified name)*
    - Séparateur « :: » en UML, par exemple : java::awt::color
    - Séparateur « . » en Java, par exemple : java.awt.color

Interface

- Permet de définir une encapsulation indépendamment de
  - L'implémentation
  - L'utilisation

- Pourquoi définir une interface ?
  - Pour permettre le développement en parallèle (multi-équipes)
  - Pour faciliter la réutilisation d'une brique logicielle

- La définition d'une interface implique deux nouveaux rôles
  - Classe cliente = classe utilisatrice de l'interface
  - Classe d'implémentation = classe réalisant le contrat défini par l'interface

Caractéristiques d’une interface

- En UML, c’est une classe stéréotypée «interface»
  - Ne contenant que des opérations publiques
  - (Peut contenir des associations et des attributs publics)
  - Définition d’un « contrat » = une responsabilité
  - Cette classe n’est pas utilisée pour instancier des objets

- Une interface doit définir un service cohérent et limité

Réalisation d'une interface

- La classe réalisant (= implémentant) l'interface doit
  - Contenir les opérations définies dans l'interface
  - (Contenir les attributs définis dans l'interface)
  - Respecter le « contrat » défini dans l'interface

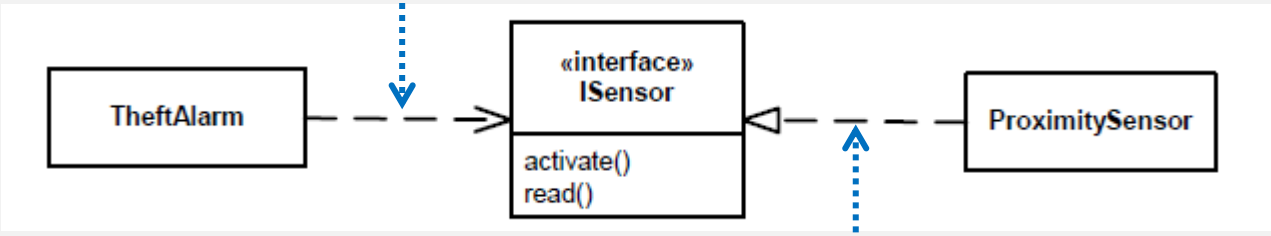
- La classe réalisant (= implémentant) l'interface peut
  - Déléguer une partie de sa responsabilité sur d'autres classes dont elle devient la façade

- NB : une classe peut réaliser plusieurs interfaces

Notation UML

- Notation UML d'origine
  (appelée aujourd'hui notation alternative)
  - Relation de dépendance de la classe cliente vers l'interface (*dependency*, cf. cours sur les éléments transversaux)

aspects statiques

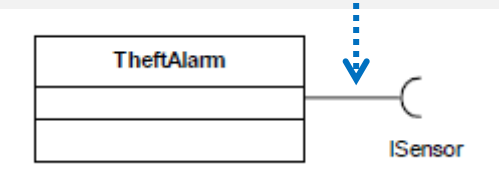


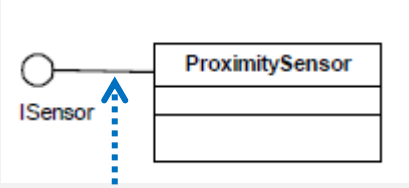
  - Relation de réalisation de la classe d'implémentation vers l'interface *(interfaceRealization)*

Notation UML

- Notation préconisée par UML (même signification)
  - Relation de dépendance de la classe cliente vers l'interface (*dependency*, cf. cours sur les éléments transversaux)

aspects statiques



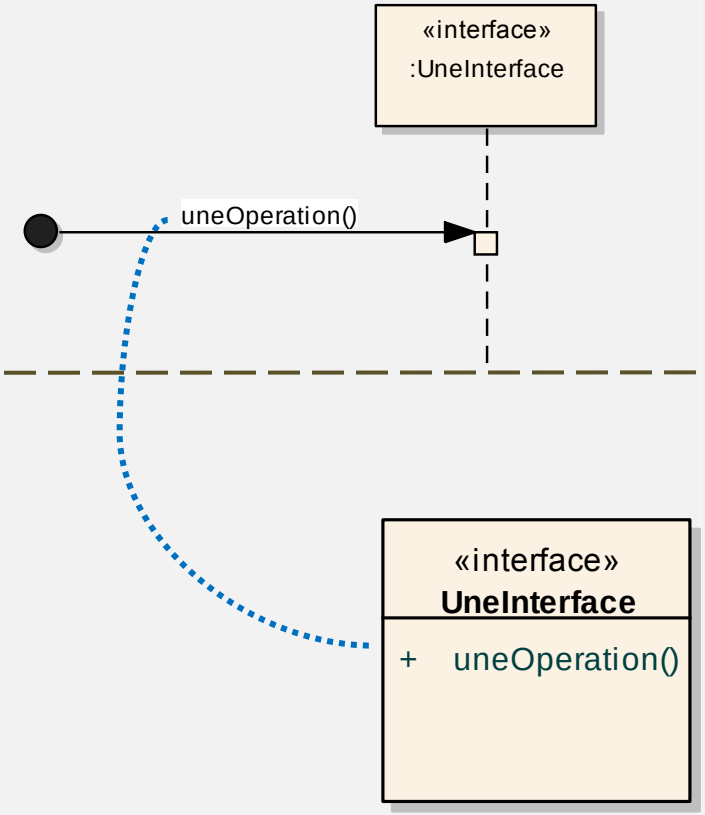


  - Relation de réalisation de la classe d'implémentation vers l'interface *(interfaceRealization)*

Notation UML
Une interface s'utilise comme une classe

aspects dynamiques

- Diagramme de séquence
  - Ne pas décrire le fonctionnement interne ici (celui-ci peut être décrit dans un autre diagramme avec une classe d'implémentation)



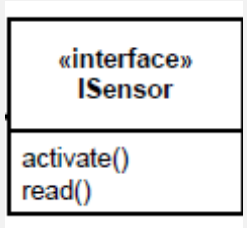
aspects statiques

- Diagramme de classes correspondant
- Code Java correspondant

```
interface UneInterface {
   public … uneOperation();
}
```

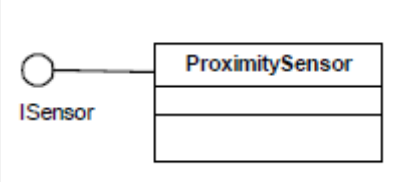
Correspondances en Java

- Définition d'une interface



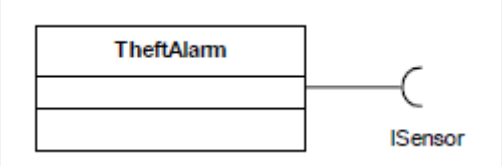

```
interface ISensor {
   public … activate();
   public … read();…
}
```

- Implémentation d'une interface




```
class ProximitySensor implements Isensor {
   public … activate() {…}
   public … read() {…} …
   }… }
```

- Utilisation d'une interface




```
class TheftAlarm {…
   Isensor mySensor = new ProximitySensor;
   mySensor.activate();
…}
```

Exemple d'interface prédéfinie dans Java

- Définition de l'interface Comparable (compare deux objets)

```
interface Comparable {
   int compareTo(o) // o = objet à comparer
}
```

- Implémentation de l'interface Comparable (choix de comparer un numéro d'ordre)

```
class Compte implements Comparable {
   int compareTo(o){ // o = objet à comparer
   return this.ordre - o.ordre}…
}
```

- Utilisation de l'interface Comparable (dans l'opération « sort » de la classe Collections)

```
class ListeFormulaires{…
   ArrayList liste = new ArrayList;
   …
   Collections.sort(liste);
…}
```

Synthèse concernant les interfaces

- Application du principe d'encapsulation pour un ensemble d'objets
- Définition de contrats entre équipes ou entre développeurs permettant de développer des parties de logiciel en parallèle
- Favorise la réutilisation et le travail en équipes multiples
- Limite les impacts en cas de modification