

Plan du cours

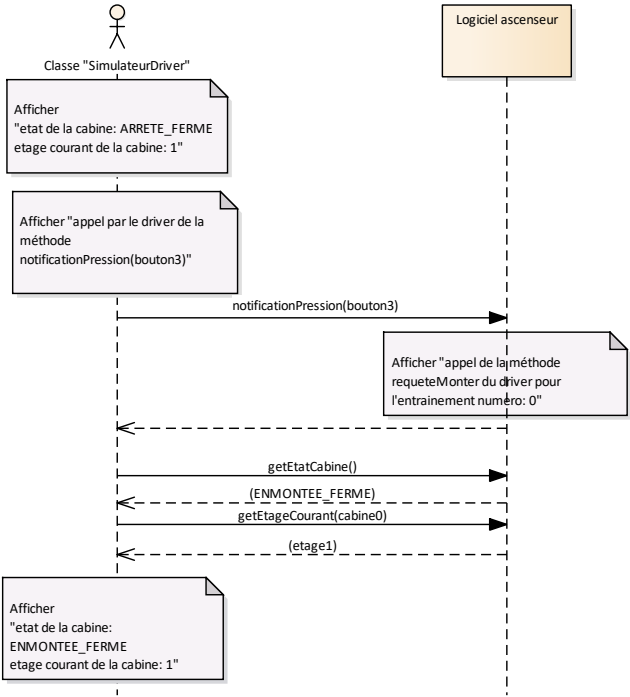
- Objectifs du projet de développement et méthode
 - Les **activités d'ingénierie logicielle** – zoom sur la spécification des **exigences**, la **conception** et **codage**
 - La **gestion de projet itérative** - Mise en place du projet ascenseur
- **Sprint 1**: Réaliser le début du scénario "Appeler l'ascenseur – la **séparation des préoccupations** en unités de codes ("classes")
- **Sprint 2**: Refactoring du code pour améliorer sa maintenabilité et évolutivité – **bonnes pratiques de conception fondamentales - UML**
- **Sprint 3**: Elargir le périmètre fonctionnel au scénario complet - **communication entre classes**
- **Sprint 4**: Refactoring du code – **architecturer en packages**
- **Sprint 5**: Refactoring du code – affiner la communication entre couches - modes **requête/notification**
- **Sprint 6**: Refactoring du code – conception et programmation **OO** 1

§2

Objectifs du sprint 2

- **Objectif:** Refactoring du code pour améliorer sa maintenabilité et évolutivité (qualité interne) =>
 - Sur le même périmètre fonctionnel (i.e. même exigences)
 - Application des bonnes pratiques de conception fondamentales
 - Exprimer clairement la conception du logiciel: préoccupation des classes et dépendances entre les classes(utilisation d'UML)
- **Début-Fin:** 24/11-01/12
- **Revue de sprint:** 02/12
 - Démo – conformité aux exigences
 - Explication claire de la conception du code et recherche de pistes d'amélioration de la maintenabilité et de l'évolutivité₂ du code avec l'architecte logiciel

Sprint 2 – Rappel des exigences/tests de qualification du sprint 1 (1/3)



Sprint 2 – Rappel des exigences/tests de qualification du sprint 1 (2/3)

- Classe simulateurDriver :

```
package test_logiciel_ascenseur;
import logiciel_ascenseur.*;
public class SimulateurDriver {

    public static void main(String[] args) {
        // Initialisation: Ascenseur a 4 etages, dont le RDC
        ????.creerBouton(0,0);
        ...

        //Execution du scenario par simulation des drivers
        System.out.println("...
        ????.notificationPression(3);

        //Affichage état cabine - vérification post-conditions
        System.out.println("etat ...

    }
}
```

Sprint 2 – Rappel des exigences/tests de qualification du sprint 1 (3/3)

- Exécution du scénario de tests - Sortie console:

```
run:
etat de la cabine: ARRETE_FERME
etage courant de la cabine: 1

appel par le driver de la méthode notificationPression(bouton3)
appel de la méthode requeteMonter du driver pour l'entrainement numero: 0

etat de la cabine: ENMONTEE_FERME
etage courant de la cabine: 1
```

Quelles activités doivent être réalisées dans le sprint 2?

Vision utilisateur, externe

Exigences
(et def. tests de qualification)



Tests
de qualification

Vision développeur, interne

Conception
(et def. tests int.)

Tests
d'intégration

Codage
(et TU)

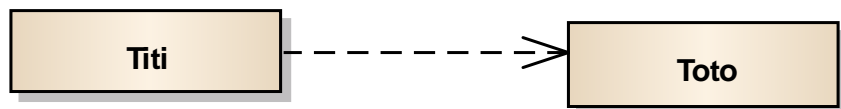
Rappel des bonnes pratiques de conception

- Pratique 1- Séparation des préoccupations en unités de code:
 - Chaque unité de code possède une responsabilité claire, reflétée par son nom
 - La responsabilité d'une unité de code n'est pas partagée avec une autre unité (i.e. pas de duplication de code)
 - Les données et les traitements associés sont définis dans la même unité, qui est responsable de l'intégrité de ses données
 - Structurer par "type de choses à gérer"
 - ...
- Pratique 2- Limitation des dépendances:
 - Eviter la redondance de données
- Pratique 3: Cohérence de l'ensemble: réponse aux exigences

§2

Qu'est-ce qu'une dépendance entre unité de code?

- De façon générale, une unité de code Titi (e.g. classe Titi) dépend d'une unité de code Toto, signifie que:
 - Titi a besoin de Toto pour faire son travail
 - En conséquence, une modification de Toto peut entraîner un besoin de modification de Titi



Conception et gestion des dépendances

- En conception, on va s'attacher à limiter au maximum les dépendances entre unités de code
- De façon générale, la notation UML va nous aider à exprimer clairement la structuration de notre code et donc appliquer les bonnes pratiques:
 - Séparation des préoccupations en unités de code
 - Limitation des dépendances entre unités de code
 - Cohérence de l'ensemble: réponse aux exigences

Programmation structurée et POO

- Les fondamentaux de la programmation structurée dont la POO:
 - unité de code: regroupe des données et traitements
 - les valeurs des données constitue le contexte d'exécution des traitements
- Conception OO:
 - classe ("unité de code")
 - objet et instanciation ("occurrence des données et contexte d'exécution")
 - attribut
 - opération

Unité de code

- Un fichier de code source dans lequel sont définis un ensemble de données et de traitements
- Rappel pratique 1- La séparation des préoccupations en unités de code:
 - Chaque unité de code possède une responsabilité claire, reflétée par son nom
 - La responsabilité d'une unité de code n'est pas partagée avec une autre unité (i.e. pas de duplication de code)
 - Les données et les traitements associés sont définis dans la même unité, qui est responsable de l'intégrité de ses données
 - Structurer par "type de choses à gérer"

En C: Module C = fichier .c

- Fichier GestionCommandes.c:

```
typedef struct Commande {  
    float montantHT;  
    float montantTTC;  
}  
struct Commande lesCommandes [100];  
float tauxTaxes;  
void facturerCdeTTC(int numero){  
    lesCommandes[numero].montantTTC=  
    lesCommandes[numero].montantHT+  
    lesCommandes[numero].montantHT* tauxTaxes;  
}
```

En PHP non objet: fichier .php

- GestionCommandes.php

```
$lesCommandes
$tauxTaxes

function facturerCdeTTC ($numero){
    global $lesCommandes;
    global $tauxTaxes;

    $lesCommandes[$numero]['montantTTC']= $lesCommandes[$numero]['montantHT']+
    $lesCommandes[$numero]['montantHT']* $tauxTaxes;}
}
```

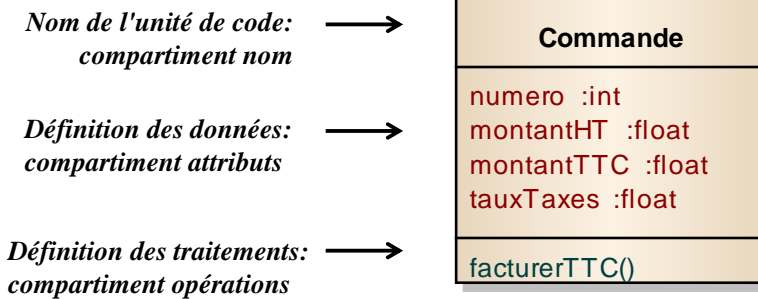
En Java: une classe

- Commande.java

```
class Commande {  
    float montantHT;  
        float montantTTC;  
    static float tauxTaxes;  
  
    void facturerCdeTTC(){  
        this.montantTTC= this.montantHT+this.montantHT*tauxTaxes;  
    }  
}
```

En UML: une classe logicielle

- Une classe logicielle représente une unité de code dans laquelle sont définis un ensemble de données et des traitements sur ces données



Occurrences des données et contexte d'exécution des traitements

Commande
numero :int montantHT :float montantTTC :float tauxTaxes :float
facturerTTC()

```
class Commande {  
    float montantHT;  
        float montantTTC;  
    static float tauxTaxes;  
  
    void facturerCdeTTC(){  
        this.montantTTC= this.montantHT+this.montantHT*tauxTaxes;  
    }  
}
```

- Généralement les données existent en plusieurs occurrences: une occurrence pour chaque commande – i.e. pour chaque **objet** commande
- Généralement les traitements s'exécutent dans le contexte d'un objet particulier

Une problématique classique: comment indiquer le contexte d'exécution d'un traitement – i.e. "l'objet" du traitement?

- En programmation procédurale:

```
void facturerCdeTTC(int numero){  
    lesCommandes[numero].montantTTC=  
    lesCommandes[numero].montantHT+  
    lesCommandes[numero].montantHT* tauxTaxes;  
}
```

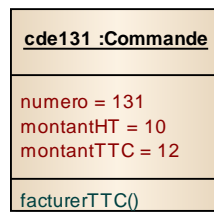
- La notion d'objet est implicite, mais existe: c'est une entrée dans le tableau des commandes.

Quelle donnée est particulière?

- La POO met en avant la notion de contexte d'exécution avec la notion explicite d'objet

Objet – Object Définition UML

- **Objet** = "Entité discrète avec une frontière bien définie et une identité qui encapsule son état et son comportement ; instance d'une classe."



- Définition "plus POO" = quelque chose, à l'intérieur du logiciel à l'exécution, qui a une identité, un état, et un comportement. C'est une instance d'une classe qui aura été codée précédemment.

Objet – Object Illustration Java

- Une zone mémoire contenant la valeur des attributs

<u>cde131 :Commande</u>
numero = 131
montantHT = 10
montantTTC = 12

131
10
12

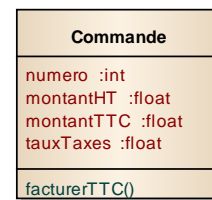
- Quelle est l'identité de l'objet?



Classe – Class Définition UML

- **Classe** = "Descripteur d'un jeu d'objets qui partagent les mêmes caractéristiques (attributs, opérations, relations ...)."

Classe représentées dans un
diagramme de classes – Class Diagram




- **Bonne pratique:** Le nom de la classe reflète la nature des objets, instances de la classe



Classe – Class Illustration Java

- Une **classe** Java:



Commande

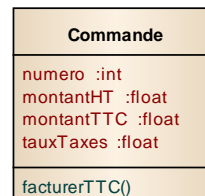
Fichier Commande.java:

```
class Commande {  
    ....  
}
```

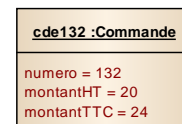
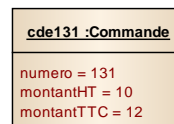
Attribut – Attribute Définition UML

- **Attribut** = "Description d'un élément nommé prenant un type spécifié dans une classe. Chaque objet de la classe détient séparément une valeur du type."

*Attributs définis
dans la classe*



*Valeurs des attributs
dans chaque objet*



- Les valeurs des attributs définissent l'**état** de l'objet

Attribut – Attribute Illustration Java

- Un **attribut** dans la classe Java:

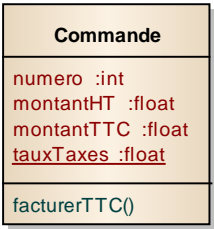
Commande
numero :int montantHT :float montantTTC :float tauxTaxes :float
facturerTTC()

```
class Commande {  
  ...  
  int numero;  
  float montantHT;  
  float montantTTC;  
  ....  
}
```

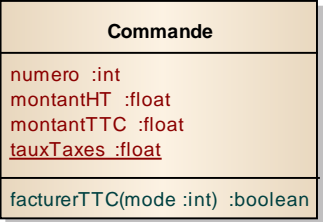
Opération – Operation Définition UML

- **Opération** = "Spécification d'une requête ou d'une transformation de son état, qu'un objet peut être appelé à exécuter."

Opération →



- Le contexte d'exécution d'une opération est donc un objet
- Une opération peut posséder une liste de paramètres, dont des paramètres de retour



Opération – Operation Illustration Java

- Une **méthode** dans la classe Java:

Commande
numero :int montantHT :float montantTTC :float <u>tauxTaxes :float</u>
facturerTTC()

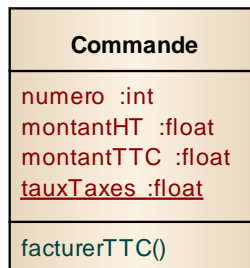
```
class Commande {  
    ...  
    float montantHT;  
    float montantTTC;  
    static float tauxTaxes;  
    ....  
    void facturerTTC() {  
        this.montantTTC = this.montantHT +  
                           this.montantHT * tauxTaxes;  
    }  
    ...  
}
```

Attribut de classe – Class attribute

Définition UML

- **Attribut de classe** = "Attribut dont la valeur est partagée par toutes les instances de la classe. Egalement appelé attribut statique."

Attribut de classe



Attribut de classe – Class attribute

Illustration PHP

- Propriété statique:

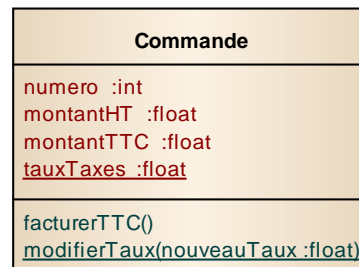
Commande
numero :int montantHT :float montantTTC :float <u>tauxTaxes :float</u>
facturerTTC()

```
class Commande {  
    ...  
    var $numero;  
    var $montantHT;  
    var $montantTTC;  
    static $tauxTaxes;  
    ....  
}
```

Opération de classe – Class operation Définition UML

- **Opération de classe** = "Opération dont l'accès est lié à une classe et non à une instance de la classe."

Opération de classe →



Opération de classe – Class operation Illustration PHP

- Définition d'une **méthode statique** dans la classe:

```
class Commande {  
    var $numero;  
    var $montantHT;  
    var $montantTTC;  
    static $tauxTaxes;  
  
    static function modifierTaux($nouveauTaux){  
        $tauxTaxes = $nouveauTaux;  
    }  
}
```

- Appel de la méthode:

```
class Titi {  
    ...  
    Commande::modifier_taux(21);  
    .....  
}
```

Le principe d'encapsulation objet

- la classe n'expose pas les données dont elle est responsable: ses **attributs sont privés**
- Elle communique avec l'extérieur (les objets des autres classes) via des messages qu'elle reçoit: invocation **d'opérations publiques**

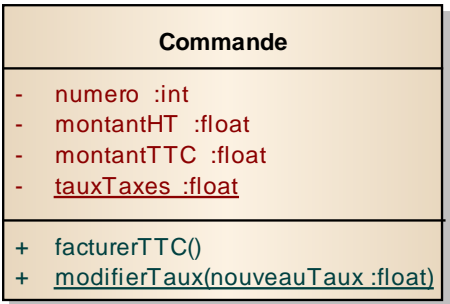
Visibilité – Visibility

Définition UML

- **Visibilité** = "La visibilité d'un élément indique si on peut voir l'élément en dehors de son espace de noms enveloppant."
- Deux visibilités de base pour les attributs et les opérations: **privé**-private et **public**-public

Visibilité privé →

Visibilité publique →



Visibilité – Visibility

Illustration Java

Commande
- numero :int - montantHT :float - montantTTC :float - <u>tauxTaxes :float</u>
+ facturerTTC() + <u>modifierTaux(nouveauTaux :float)</u>

```
class Commande {  
    ...  
    private int numero;  
    private float montantHT;  
    ....  
    public void facturerTTC() {  
        ...  
    }  
    ...  
}
```


Encapsulation – Accesseurs

- Des méthodes d'accès aux informations:

Commande
- numero :int - montantHT :float - <u>tauxTaxes</u> :float
+ getNumero() :int + getMontantHT() :float + <u>getTauxTaxes()</u> :float + setNumero(numero :int) + setMontantHT(montant :float) + <u>setTauxtaxe(taux :float)</u>

- Note: les accesseurs ne sont pas toujours modélisés

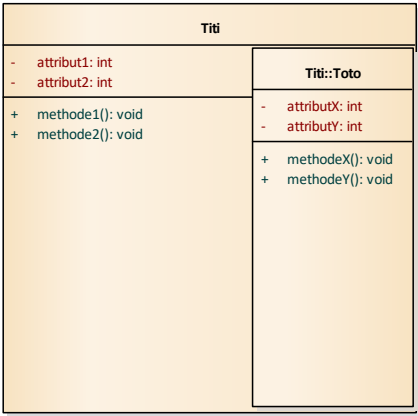
Le principe d'encapsulation est un principe fondamental de conception

- Le principe d'encapsulation aide à mettre en œuvre nos 2 bonnes pratiques de conception:
 - Pratique 1- Séparation des préoccupations en unités de code:
 - ...
 - Les données et les traitements associés sont définis dans la même unité, qui est responsable de l'intégrité de ses données: **les données sont privées**
 - Pratique 2- Limitation des dépendances:
 - **Pas de dépendance directe sur les données**
- Comment traduire le principe d'encapsulation en programmation procédurale classique?



Classe imbriquée/Nested class

- A nested class is a member of its enclosing class



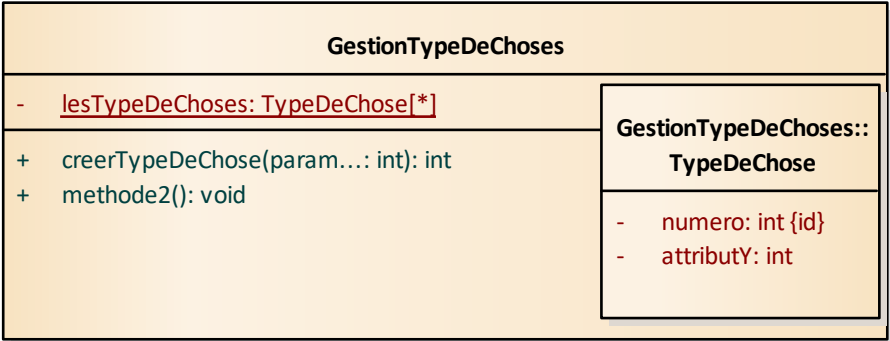
Intérêt des classes imbriquées

- Why Use Nested Classes?
 - It is a way of logically grouping classes that are only used in one place: If a class is useful to only one other class, then it is logical to embed it in that class and keep the two together. Nesting such "helper classes" makes their package more streamlined.
 - It increases encapsulation: Consider two top-level classes, A and B, where B needs access to members of A that would otherwise be declared private. By hiding class B within class A, A's members can be declared private and B can access them. In addition, B itself can be hidden from the outside world.
 - It can lead to more readable and maintainable code: Nesting small classes within top-level classes places the code closer to where it is used.

Notre pattern "Gestionnaire d'objets statique" (1/2)

```
public class GestionRectangles {  
    static class Rectangle {  
        private int numero;  
        private int largeur;  
        private int longueur;  
    }  
  
    private final static int NB_MAX_RECTANGLES = 2;  
    private static Rectangle[] lesRectangles = new Rectangle[NB_MAX_RECTANGLES];  
    private static int nbRectangles = 0;  
  
    public static int creer (int largeur, int longueur){  
        lesRectangles[nbRectangles] = new Rectangle();  
        lesRectangles[nbRectangles].largeur = largeur;  
        lesRectangles[nbRectangles].longueur = longueur;  
        nbRectangles++;  
        return nbRectangles - 1;  
    }  
  
    public static int getLargeur(int numRectangle){  
        return lesRectangles[numRectangle].largeur;  
    }  
  
    public static int getLongueur(int numRectangle){  
        return lesRectangles[numRectangle].longueur;  
    }  
  
    public static int getSurface(int numRectangle){  
        return lesRectangles[numRectangle].largeur * lesRectangles[numRectangle].longueur;  
    }  
  
    public static void setLongueur(int numRectangle, int longueur){  
        lesRectangles[numRectangle].longueur = longueur;  
    }  
}
```

Notre pattern "Gestionnaire d'objets statique" (2/2)



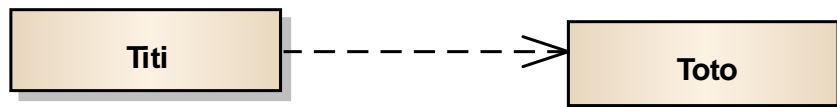
Dans quelle mesure ce pattern est-il en accord avec les bonnes pratiques de conception?



- Pratique 1- Séparation des préoccupations en unités de code:
 - Chaque unité de code possède une responsabilité claire, reflétée par son nom
 - La responsabilité d'une unité de code n'est pas partagée avec une autre unité (i.e. pas de duplication de code)
 - Les données et les traitements associés sont définis dans la même unité, qui est responsable de l'intégrité de ses données: les données sont privées
 - Structurer par "type de choses à gérer"
 - ...
- Pratique 2- Limitation des dépendances:
 - Eviter la redondance de données
 - Pas de dépendance directe sur les données
 - ..
- Pratique 3: Cohérence de l'ensemble: réponse aux exigences

Relations entre classes – Importance de la relation de dépendance

- La notation UML permet de représenter différentes relations entre classes: dépendance, association, généralisation, imbrication
- La relation la plus importante pour la qualité interne de notre code est la relation de dépendance



- Nous voulons **éviter le plat de spaghetti de dépendances entre les unités de code**