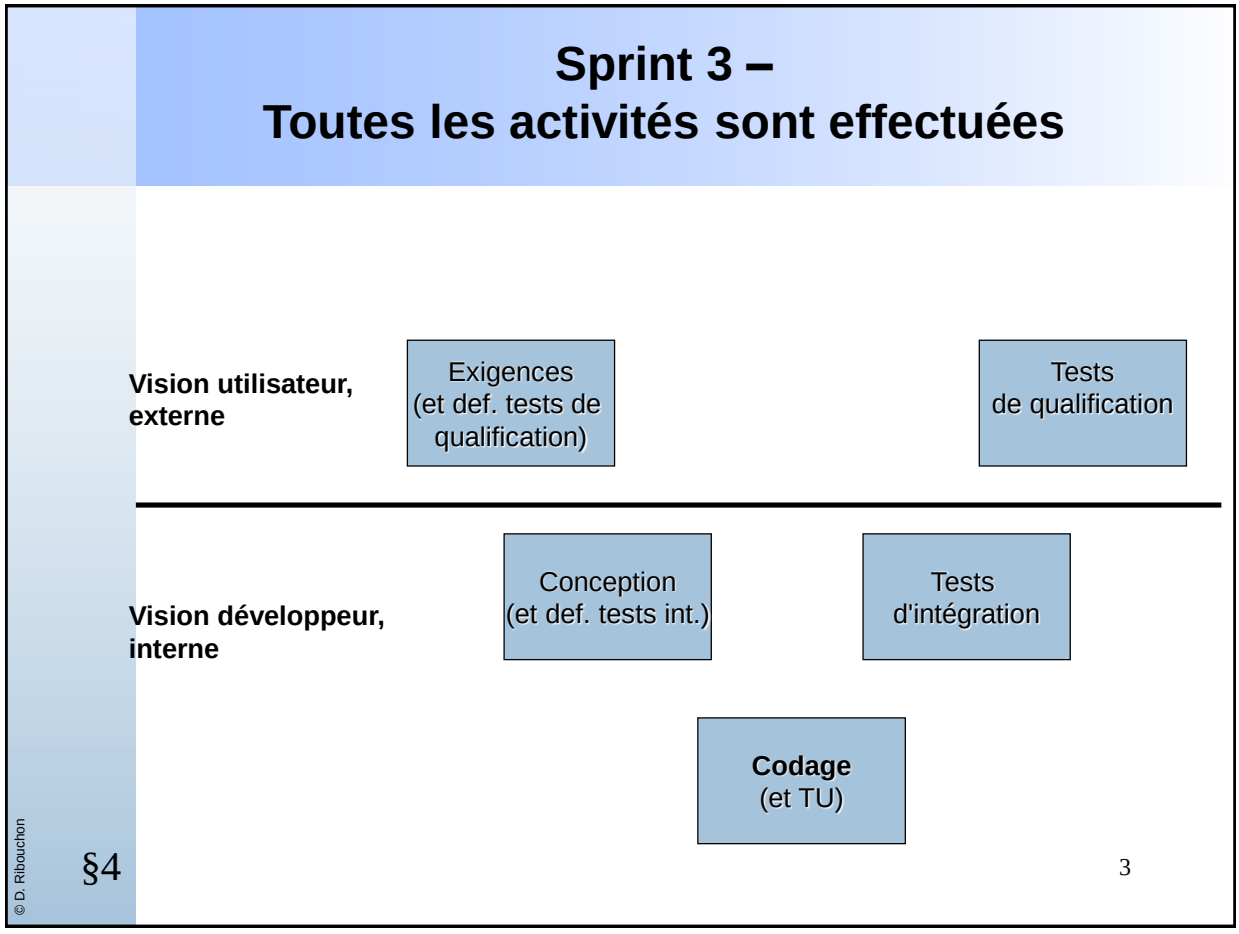
Plan du cours

- 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**

Sprint 3 -
Objectifs et déroulement

- Objectifs:
  - Elargir le périmètre fonctionnel au scénario complet (qualité externe)
  - Réaliser un code maintenable et évolutif - Définir la communication entre les classes
- Déroulement:
  - Exigences/tests de qualification: compléter le scénario
  - Concevoir: concevoir le scénario complet – garantir la cohérence de l'ensemble (en anticipant la communication entre les classes)
  - Coder
  - Tester

Sprint 3 –
Toutes les activités sont effectuées



§4

3

Sprint 3 – Exigences/tests de qualification (1/3)

- Compléter le scénario:
  - Retrouver le scénario d'exigences "à terminaison"
  - Sur la base du scénario du sprint 2, compléter le scénario d'exigences/tests qualification du sprint 3

Sprint 3 – Exigences/tests de qualification (3/3)

- Classe simulateurDriver complétée:

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

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

        //Execution du scenario par simulation des drivers
        ...
        GestionBoutons.notificationPression(3);
        ...
        ???.notificationNouvelEtage(0, 2);
        ...
        ???.notificationNouvelEtage(0, 3);

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

Sprint 3 – Exigences/tests de qualification (2/3)

- Exécution du scénario test:

```
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

appel par le driver de la méthode notificationNouvelEtage(entrainement0,etage2) par le driver

appel par le driver de la méthode notificationNouvelEtage(entrainement0,etage3) par le driver
appel de la méthode requeteArreter du driver pour l'entrainement numero: 0
appel de la méthode requeteOuvrir du driver pour la porte numero: 4
appel de la méthode requeteOuvrir du driver pour la porte numero: 3

etat de la cabine: ARRETE_OUVERT
etage courant de la cabine: 3
```


§4

© D. Ribouchon 6

Sprint 3 – Conception
Rappel des bonnes pratiques

- 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
  - Eviter le plat de spaghetti de dépendances entre les unités de code
- Pratique 3: Cohérence de l'ensemble - réponse aux exigences

Sprint 3 – Conception

- Approche de conception dans le sprint :
  - Compléter la conception du sprint 2 en représentant la communication entre classes – début du scénario (reverse engineering)
  - Élargir la conception au périmètre du sprint 3 – scénario complet, en garantissant la cohérence de l'ensemble

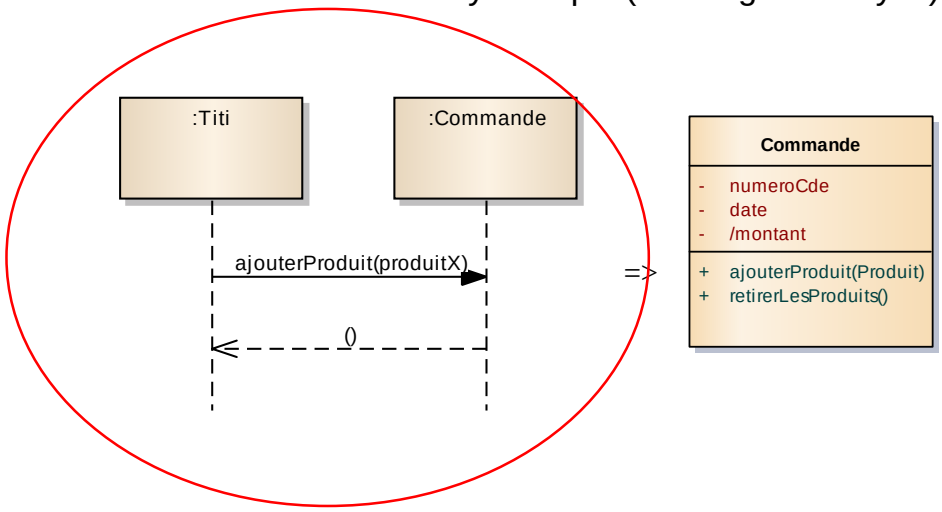
Communication entre classes = appel de méthodes/opérations publiques

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

- Ou plutôt: " Spécification d'une requête qu'un classe peut être appelée à exécuter, et dont le contexte d'exécution est généralement un objet

Démarche de découverte des méthodes publiques

- L'invocation d'une opération correspond à l'envoi d'un message
- Déduites de l'étude de la dynamique (messages envoyés)

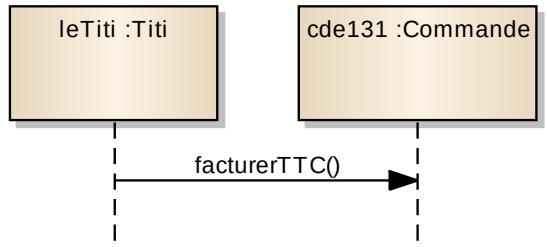


§4

Communication entre objets – Message Définition UML

- **Message entre objets** = "Communication d'un objet à un autre."

Diagramme de séquence – Sequence Diagram →



- Le **contexte de traitement/exécution** du message est l'état de l'objet récepteur
- L'objet émetteur doit connaître **l'identité de l'objet récepteur**

Communication entre objets – Message Illustration Java

- Envoi d'un message = appel d'une méthode

```
class Titi {
  …
  Commande laCommande;
  …
  methodeX(){
    …
    laCommande.facturerTTC();
    …..}
}
```

- Comment indiquerait-on le contexte d'exécution – i.e. "l'objet commande" - dans une approche procédurale classique?

§4

Communication entre objets – Message Illustration C++

- Envoi d'un message = appel d'un membre fonction

```
class Titi {
  …
  Commande *laCommande;
  …
  methodeX(){
    …
    laCommande->facturerTTC();
    …..}
}
```

- Comment indiquerait-on le contexte d'exécution – i.e. "l'objet commande" - dans une approche procédurale classique?

§4

Communication entre objets – Message Illustration PHP

- Envoi d'un message = appel d'une méthode

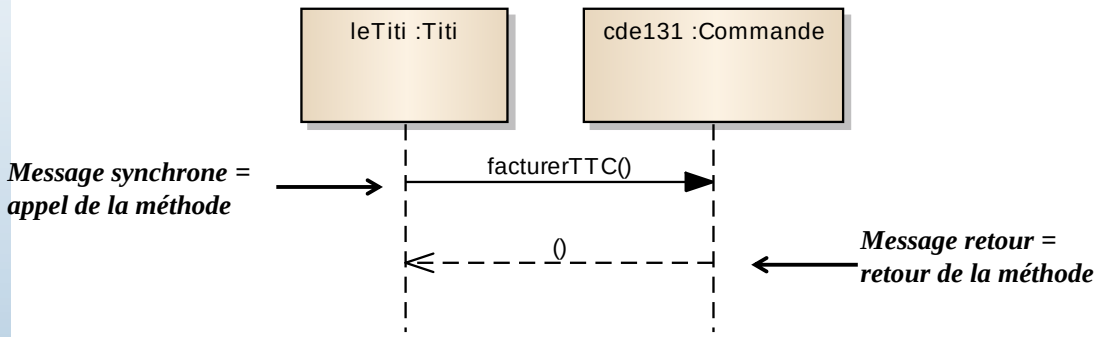
```
class Titi {
   …
   var $laCommande;
   …
   function methodeX(){
      …
      $laCommande->facturer();
      …..}
}
```

- Comment indiquerait-on le contexte d'exécution – i.e. "l'objet commande" - dans une approche procédurale classique?

§4

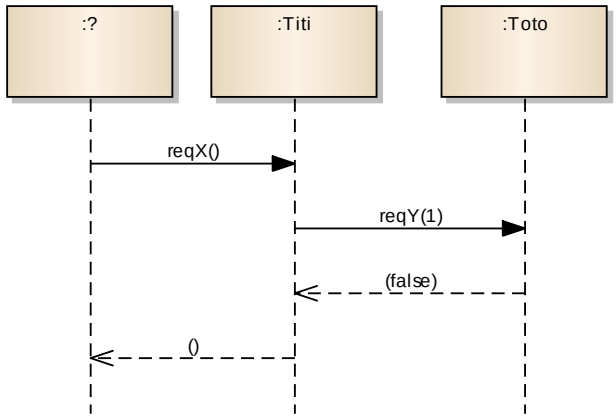
Méthode = message synchrone

- L'appel d'une méthode/opération correspond à l'envoi d'un **message synchrone**



§4

Exercice

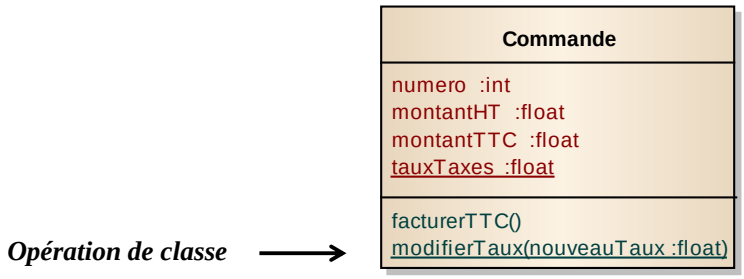


- A partir du diagramme de séquence, compléter la conception des classes Titi et Toto dans un diagramme de classes
- Coder en java ce qui peut être déduit de la conception (**uniquement**):

§4

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 – Class operation
Illustration Java

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

```
class Commande {
   private string numero;
   private float montantHT;
   private float montantTTC;
   private static float tauxTaxes;

   public static void modifierTaux(nouveauTaux){
      tauxTaxes = nouveauTaux;
}
}
```

- Appel de la méthode:

```
class Titi {
   …
   Commande.modifierTaux(valeur);
   …..
}
```


§4

Exercice

- Refaire le même exercice avec des appels de méthodes statiques

Sprint 3 – Conception (1/2)

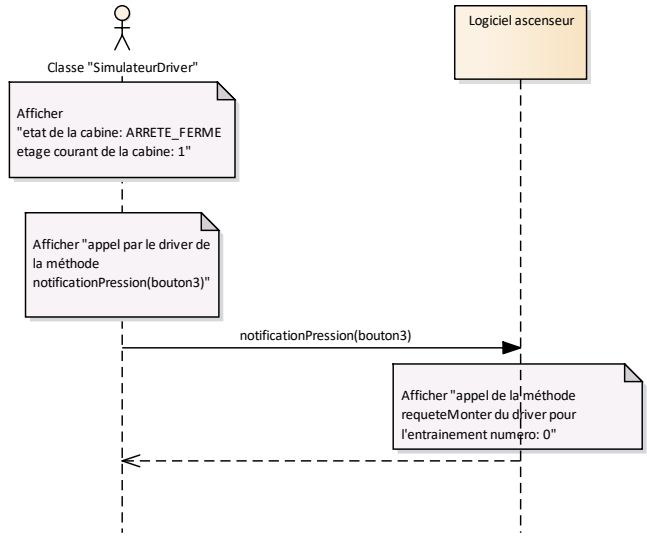
- Compléter la conception du sprint 2 en représentant la communication entre classes – début du scénario (reverse engineering)

Sprint 3 – Conception (2/2)

- Élargir la conception au périmètre du sprint 3:
  - Scénario complet
  - Garantir la cohérence de l'ensemble – réponse aux exigences
- Démarche:
  - A partir du scénario d'exigences/tests de qualification (vison externe)
  - Décrire le même scénario au niveau conception (vision interne)

Sprint 3 – Conception (2/2)

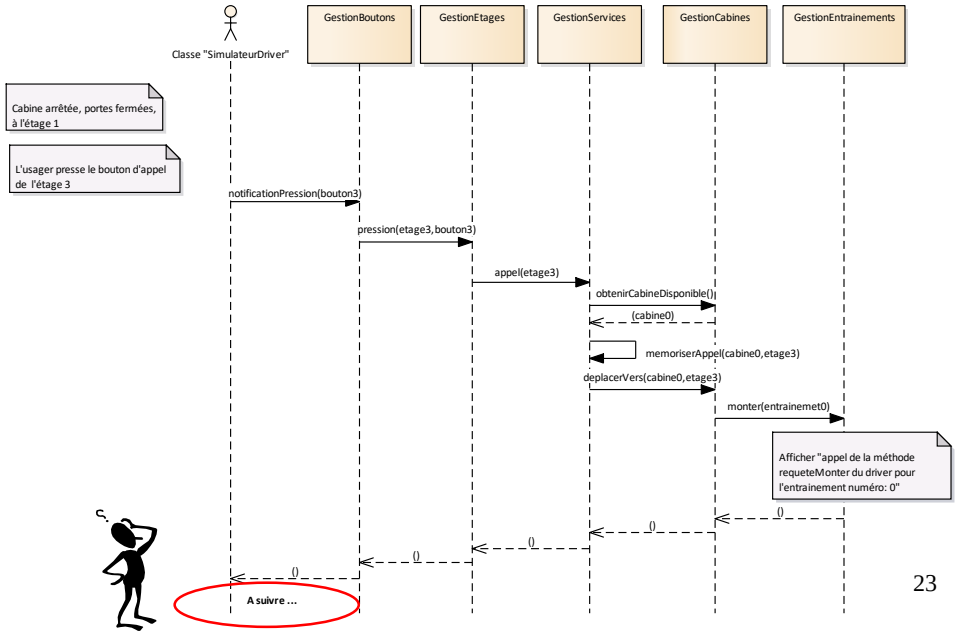
- A partir du scénario de niveau exigences – vision externe :



§4

Sprint 3 – Conception (2/2)

- Décrire le même scénario au niveau conception –vision interne:



Sprint 3 –
Finir le sprint

- Coder
- Exécuter les tests de qualification
- Préparer la revue de sprint démontrant:
  - La réponse aux exigences - qualité externe – la cohérence de l'ensemble: par une démo
  - La maintenabilité/évolutivité – qualité interne: par une présentation de la conception du logiciel et de l'application des bonnes pratiques de conception

§4

Bonnes pratiques de conception – Synthèse

- 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
  - Eviter le plat de spaghetti de dépendances entre les unités de code
- Pratique 3: Cohérence de l'ensemble - réponse aux exigences:
  - **Définir la communication entre classes à l'aide des scénarios de cas d'utilisation (niveau exigences – niveau conception)**

§4