INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES RENNES

Le 18 Décembre 2014

Benali Ibrahim
Kabir Othmane
Leparquier Mathilde
Rasata Liantsoa
Romdhane Nour

# SYNCJ : Mécanismes de synchronisation avancés pour Java

**Encadrant : Jean-Louis Pazat**

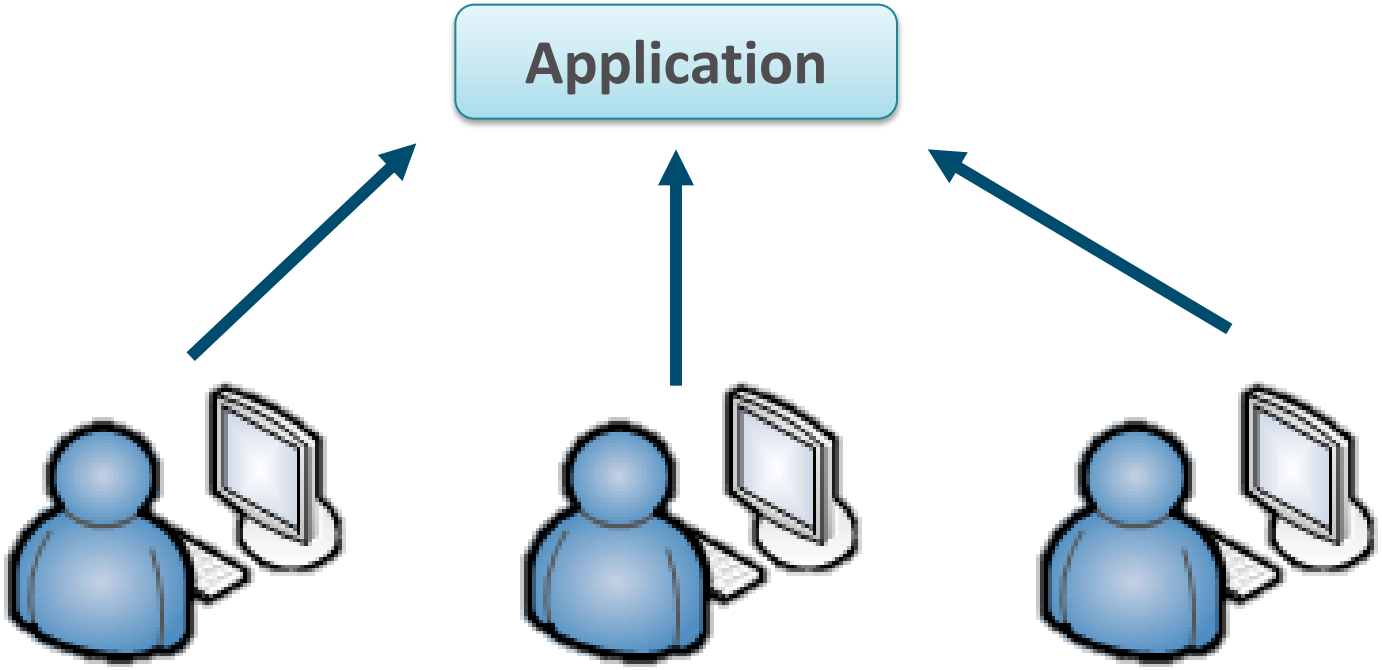
UEB UNIVERSITÉ EUROPÉENNE DE BRETAGNE

# Plan

1. Cadre du projet et Objectif
2. Spécifications générales
3. Spécifications fonctionnelles
4. Eléments de planification initiale
5. Conclusion

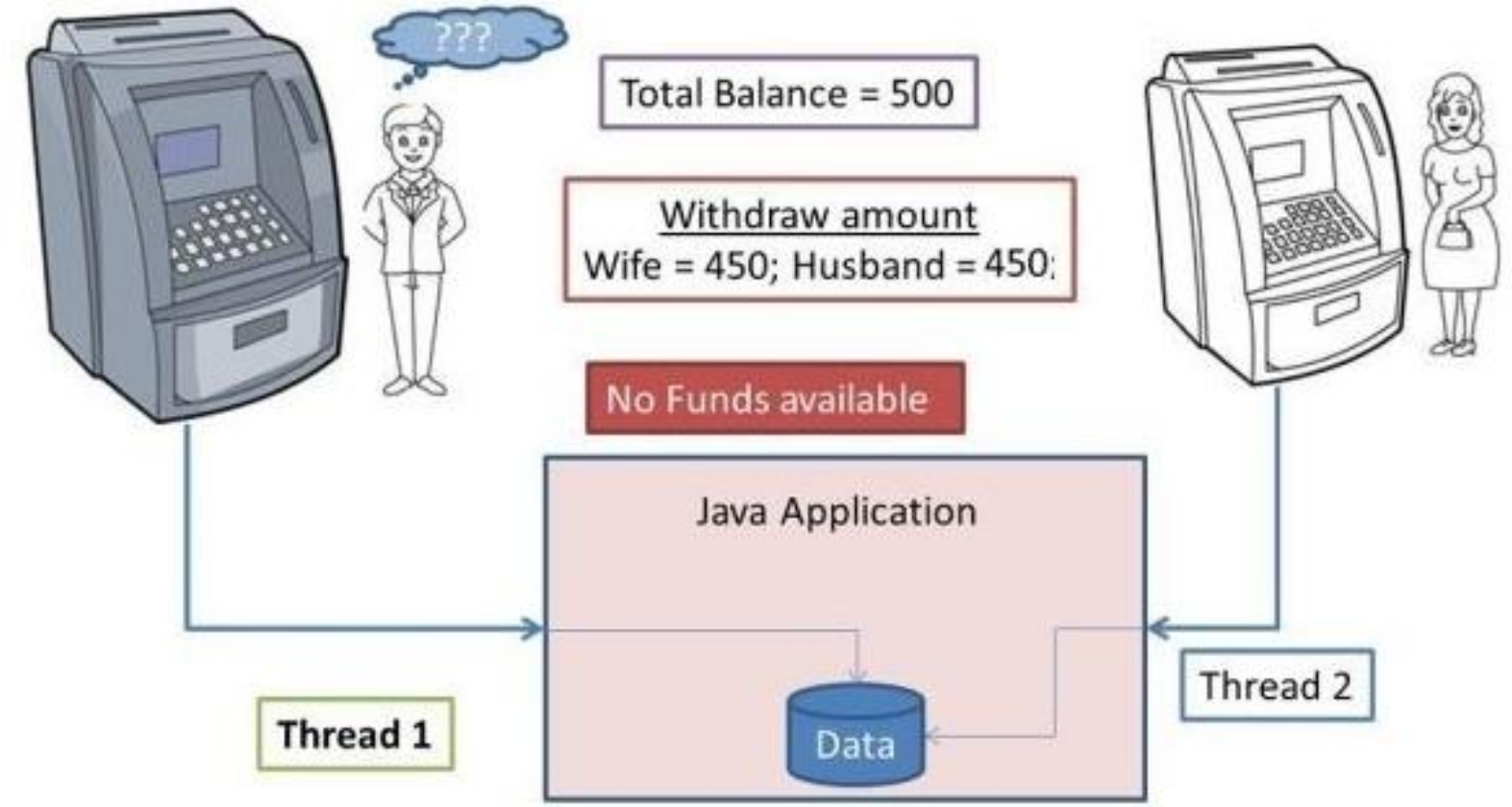
# Cadre du projet

# Parallélisme

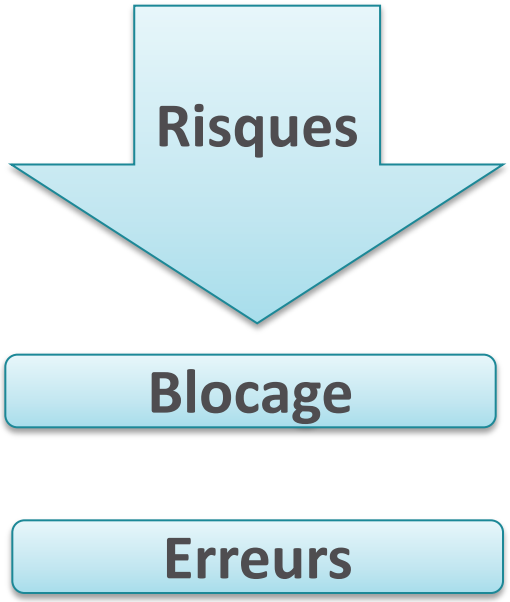



- Cohérence des données
- Communication entre utilisateurs

# Problème de cohérence



# Synchronisation

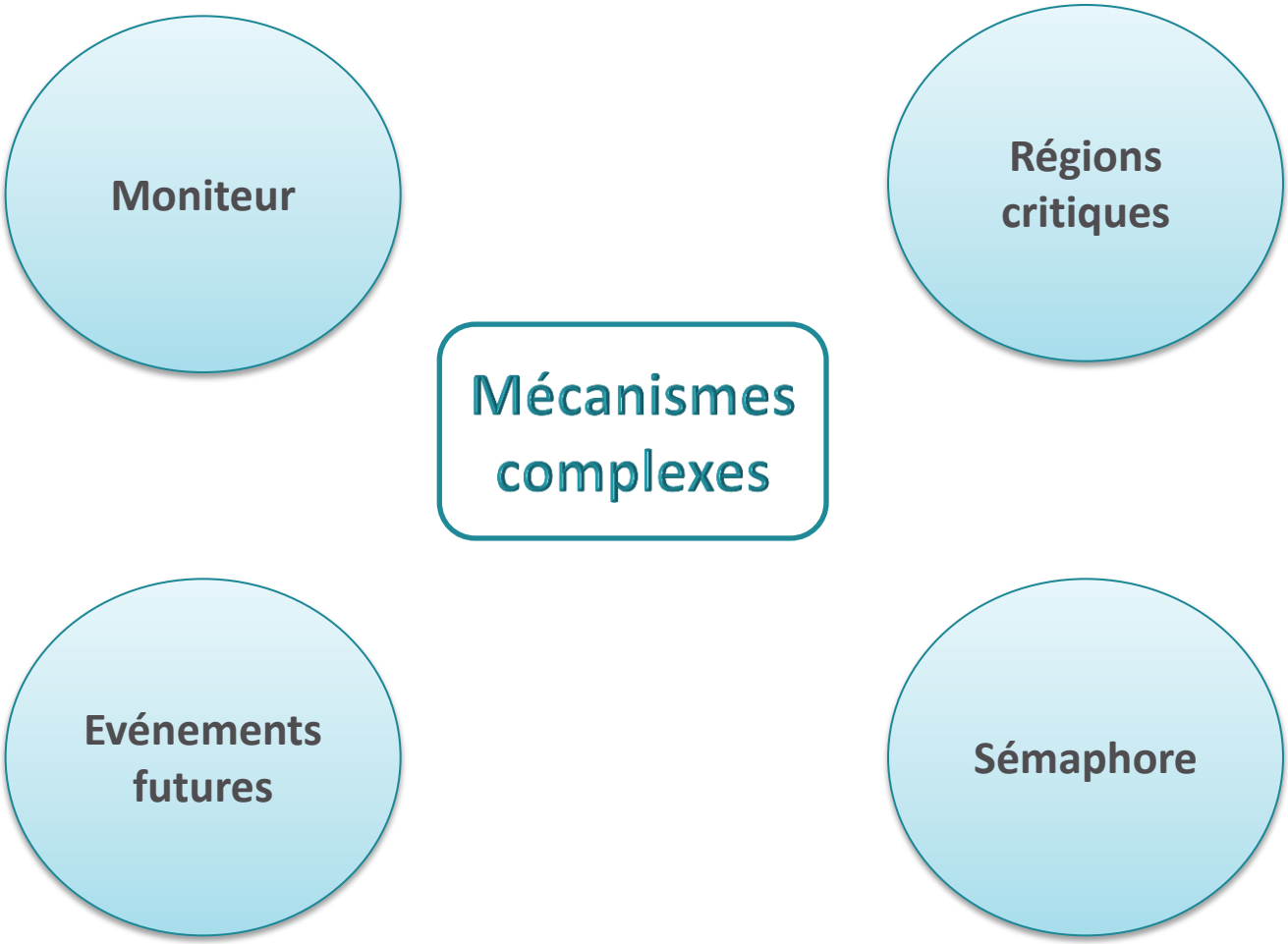



- Complexité des outils de synchronisations
- Maîtrise des outils de synchronisations

➢ **Faciliter l'apprentissage des outils de synchronisation**

# Spécifications générales

# Outils de synchronisation existants



# Synchronisation Java



**Primitives JAVA (moniteurs):**

**Wait()** : bloque un thread jusqu'à ce qu'un autre thread le débloque.

**Notify()** : debloque un seul thread.

**NotifyAll()** : debloque tous les threads.

# Synchronisation Java



Le mot clé Synchronized permet de créer une section critique :

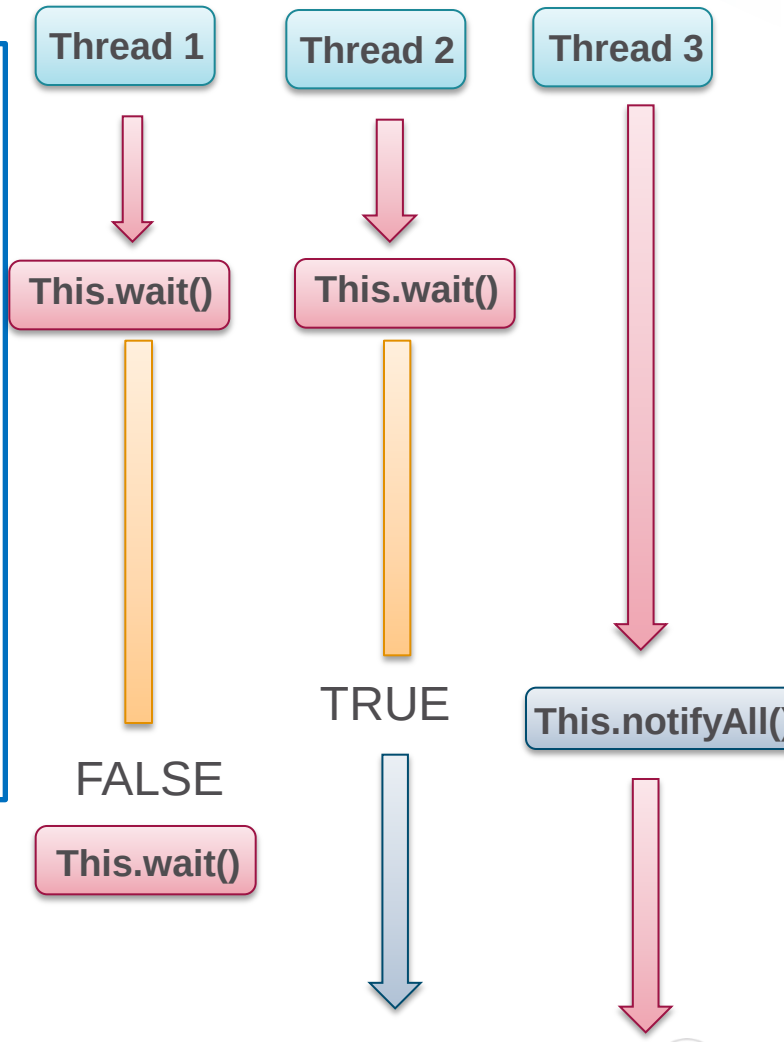
Code exécutable par un seul thread à la fois

```
public synchronized void codeProtege() {
    ... code protégé ...
}
```

```
... code non protégé ...
synchronized(objet) {
    ... code protégé ...
}
... code non protégé ...
```

# Synchronisation Java

```
public void produire(Object o){
        synchronized ( this ) {
                while( ! cond_produire ){
                        this.wait();
                }
        }
        tab[tab_prod]=o;
        tab_prod=(tab_prod+1)%TAILLE;
        synchronized ( this ) {
                notifyAll();
        }
}
```




**Condition pour produire :**
le tableau doit contenir des cases vides

# Compteurs de synchronisation

**Compteurs de synchronisation d'une méthode m :**

**m.req:** Nombre d'**invocations** m
**m.aut :** Nombre d'**autorisations** d'accès à m
**m.term:** Nombre d'**exécution terminées** de m

**m.att = m.aut - m.req:** Nombre de thread **en attente** d'autorisation d'accès à m
**m.act = m.term - m.aut** : Nombre de thread **en cours** d'exécution de m

**Conditions d'accès aux méthodes :**

**cond_produire= (produire_aut - consommer_term < N)**

# Compteurs de synchronisation

Code avec les compteurs et les conditions

Code avec les outils Java

# Spécifications fonctionnelles

# Introspection

## Utilisation :

- Informations dynamiques sur les classes
- Création dynamique des instances
- Sur JAVA (API REFLECTION)

## Exemples d'utilisation :

- **`Field[] getDeclaredFields()` :** Renvoie un tableau de tous les attributs définis dans la classe
- **`Class[] getDeclaredClasses()` :** Renvoie un tableau des classes définies comme membres dans la classe
- **`Constructor[] getDeclaredConstructors()` :** Renvoie tous les constructeurs de la classe

# Annotations

Utilisation :

- Ajouter des métadonnées aux classes, méthodes ou attributs.
- Aider le compilateur et le programmeur.

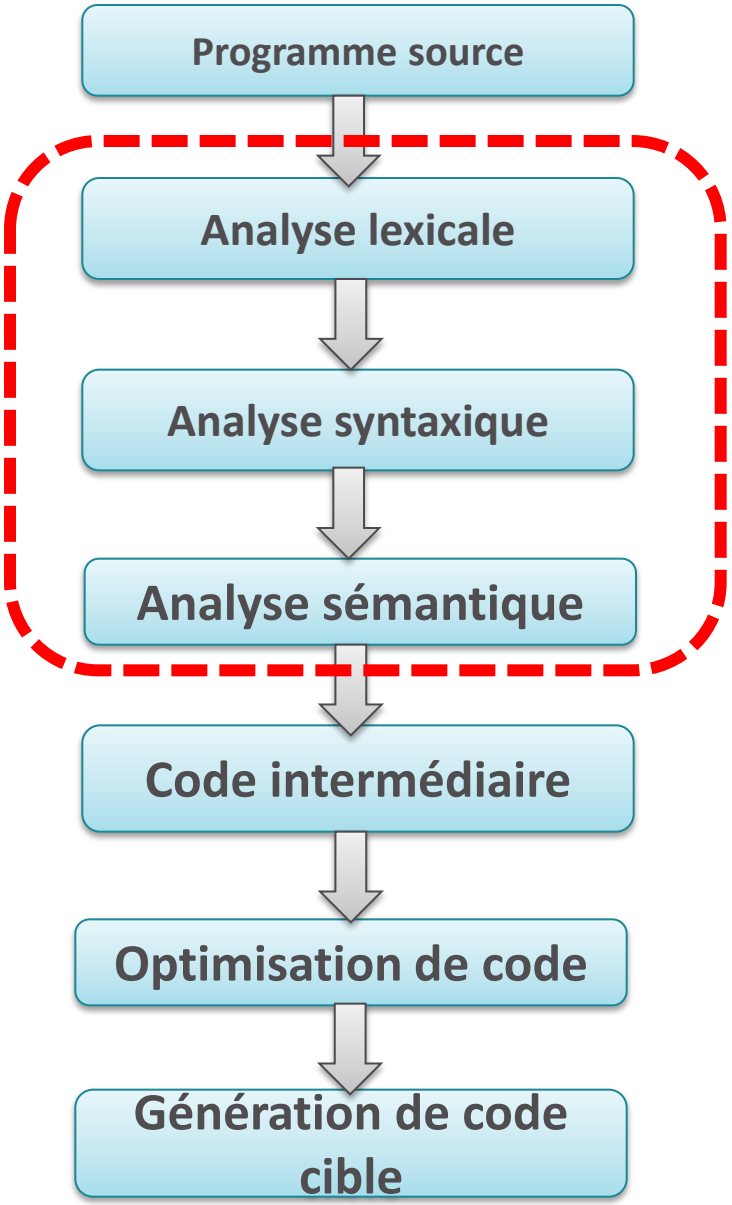
Exemples d'utilisation :

- `@condition produire = (produire_aut - consommer_term < N);`
- `@condition consommer = (produire_term - consommer_aut >0);`

# Compilation



**JavaCC**



# Compilation



Code source : Java avec des **conditions d'accès**

- Utilisant les **compteurs de synchronisation** m.req, m.aut, m.term, m.att et m.act.
- En annotation

COMPILATEUR

Code généré : Java

- Utilisant le mots clés synchronized et les primitives wait(), notify() et notifyAll().

# Code source



```
@condition produire = (produire_aut - consommer_term < N);

public void produire(object o){
    tab[tab_prod]=o;
    tab_prod=(tab_prod+1)%TAILLE;
}

@condition consommer = (produire_term - consommer_aut >0);

public object consommer (){
    return tab[tab_cons];
    tab_cons=(tab_cons+1)%TAILLE;
}
```

# Code généré



```
public void produire ( Object o ){
  synchronized ( this ) {
    produire_req++;
    this.notifyAll();
    produire_att++;
    this.notifyAll();

    while ( ! cond_produire() ){
         this.wait();
    }
    produire_aut++;
    this.notifyAll();
    produire_att--;
    this.notifyAll();
    produire_act++;
    this.notifyAll();
  }
    tab[tab_red]=s;
    tab_red=(tab_red+1)%TAILLE;
    synchronized(this){
         produire_term++;
         this.notifyAll();
         produire_act--;
         this.notifyAll();
    }
}
```

**Méthode pour évaluer la condition**

**Le thread est en attente tant que celle-ci n'est pas vérifiée.**

**Dans les blocs Synchronized :**

- **Mise à jour de tous les compteurs**
- **On libère tous les autres threads après chaque mise à jour de compteur**

# Optimisation



```
public void produire ( Object o ){
     synchronized ( this ) {
               while ( ! cond_produire() ){
                         this.wait();
               }
               produire_aut++;
               this.notifyAll();
     }
     tab[tab_red]=s;
     tab_red=(tab_red+1)%TAILLE;
     synchronized(this){
               produire_term++;
               this.notifyAll();
     }
}
```

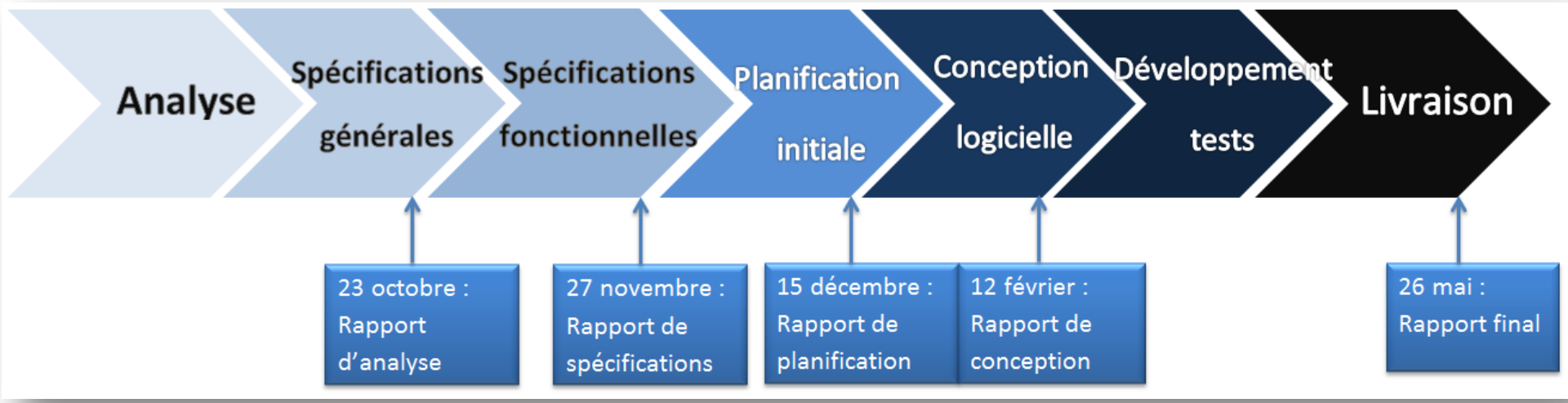
**Mise à jour des compteurs uniquement s'ils sont utilisés dans les conditions**

**Absence du notifyAll() si le compteur n'a pas évolué « dans le bon sens »**

# Mode de pilotage



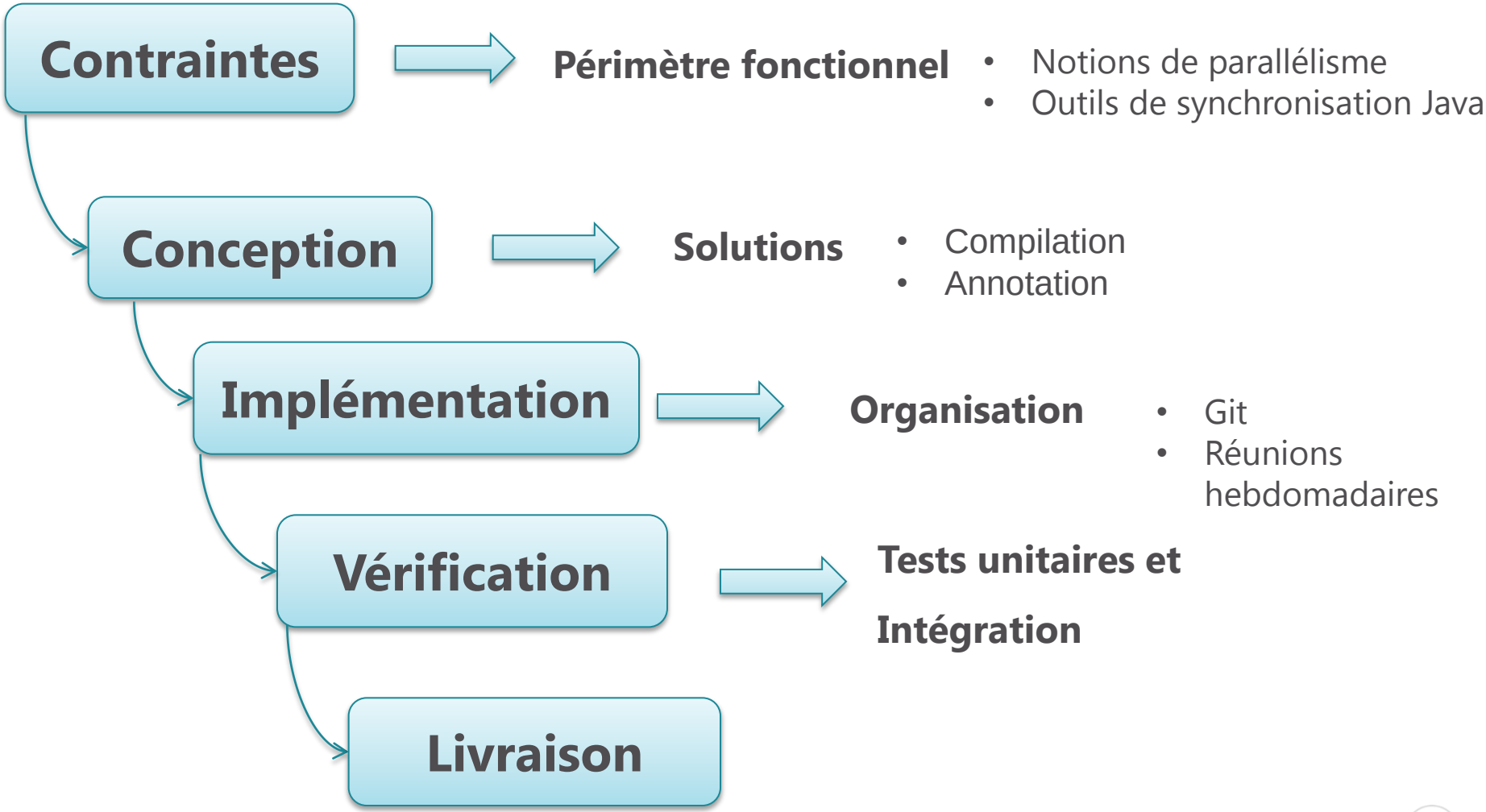
## Timeboxing



# Cycle de développement



## Modèle en cascade



# Conclusion



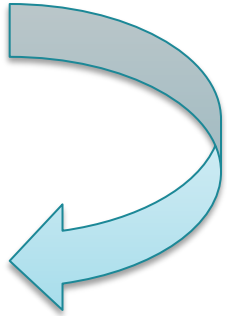
**Complexité**

Parallélisme

**Solution technique**

Compilation

**Planification initiale**



**Implémentation**

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES RENNES

# Questions

UNIVERSITÉ EUROPÉENNE DE BRETAGNE