Université Claude Bernard Lyon 1 4A, 2019–2020
Polytech Lyon - Département Informatique

ISI-3

TD2. Principes avancés de conception orientée objet.(correction)

Laëtitia Matignon

---

1 Inversion des dépendances

La figure 1 représente les relations qui existent entre les classes `Synchronisateur` et `Calculateur`. Un calculateur permet d'effectuer des calculs. Etant donné que n'importe qui peut demander à un calculateur d'effectuer des calculs, la classe `Synchronisateur` a été construite pour réguler les calculs.

Les personnes qui souhaitent demander la réalisation d'un calcul doivent passer par le synchronisateur (*via* l'opération `calculer()`). Celui-ci distribue les calculs aux différents calculateurs avec lesquels il est lié (c'est lui qui appelle l'opération `calculer()` sur les calculateurs). Un calculateur connaît le synchronisateur auquel il est relié grâce à l'attribut `sync` de type `Synchronisateur`. Sa valeur doit être déterminée lors de la création des objets de type `Calculateur`.

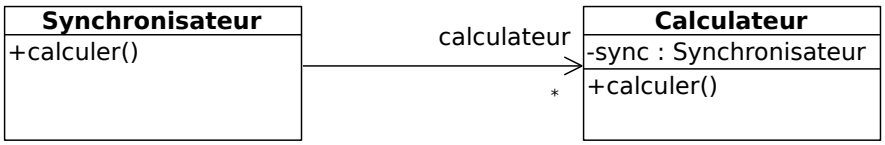


FIGURE 1 –

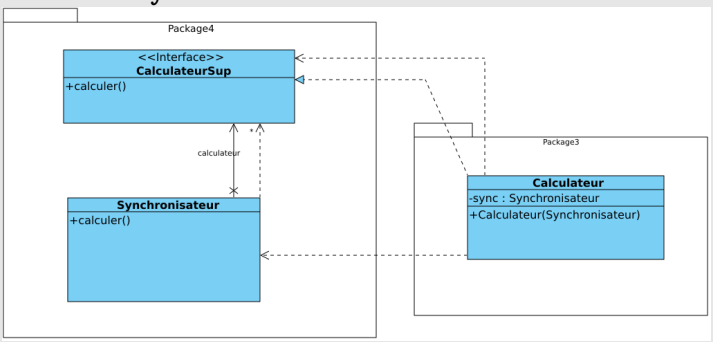
Question 1 *Exprimez en les justifiant les dépendances entre les classes* ***`Synchronisateur`*** *et* ***`Calculateur`****.*

Les dépendances sont rappelées dans le cours 1 slide 43 et 44.
L'association navigable de la classe `Synchronisateur` vers la classe `Calculateur` établit une dépendance de la classe `Synchronisateur` vers la classe `Calculateur`. **Remarque : la notation avec l'association de la classe `Synchronisateur` vers la classe `Calculateur` est équivalente à une notation où la classe `Synchronisateur` aurait un attribut appelé `calculteur` et de type `Calculateur`.**
L'attribut `sync` de type `Synchronisateur` de la classe `Calculateur` établit une dépendance de la classe `Calculateur` vers la classe `Synchronisateur` .
Nous avons donc ici une dépendance cyclique entre les 2 classes que nous pouvons représenter graphiquement de la manière suivante :

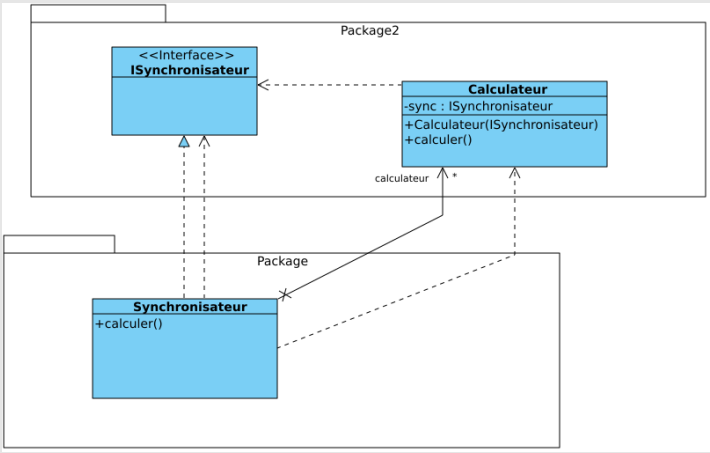
Question 2 *Nous souhaitons que les classes `Synchronisateur` et `Calculateur` soient dans deux packages différents. Proposez une solution respectant le principe des dépendances acycliques entre packages.*

Nous avons vu précédemment qu'il y a un cycle de dépendances entre les 2 classes. Cela signifie que les 2 classes dépendent mutuellement l'une de l'autre : tout changement apporté à la partie publique de l'une peut aussi impacter la partie publique de l'autre. Une dépendance cyclique entre des classes n'est pas une faute de conception (cf. slide 98 cours 1).
Cependant, si on souhaite que 2 classes dépendant mutuellement l'une de l'autre soient dans des packages différents (c'est le cas ici pour cet exercice), on aura alors une **dépendance cyclique entre les 2 packages**, ce qui **ne respecte pas le principe des dépendances acycliques entre packages** (cf. slide 101 cours 1) , selon lequel les dépendances entre paquetages doivent former un graphe acyclique. !
Afin de respecter ce principe tout en mettant les 2 classes dans 2 packages différents, on va appliquer la solution consistant à casser l'une des dépendances (méthode présentée dans le slide 102 du cours 1), en déportant la cause de la dépendance à casser hors de sa classe d'origine.
Deux solutions sont alors possibles, selon la dépendance que l'on casse :

— Si on choisit de **déporter la dépendance qui va de la classe `Synchronisateur` vers `Calculateur`**, celle-ci est due à la fonction `Calculateur:calculer()` et à l'attribut `calculateur` (représenté par l'association entre `Synchronisateur` et `Calculateur`). En effet, un `Synchronisateur` doit pouvoir identifier les calculateurs qui dépendent de lui pour pouvoir appeler leur opération `calculer()`.
Pour déporter cette cause de dépendance, on va créer une nouvelle classe `CalculateurSup` dans laquelle on déporte la méthode `calculer()`. L'association de `Synchronisateur` vers `Calculateur` est aussi déportée de `Synchronisateur` vers `CalculateurSup` ; et `Calculateur` hérite de `CalculateurSup` en gardant son attribut `sync`. Ainsi, on peut mettre dans le même package `Synchronisateur` et `CalculateurSup` sans avoir de dépendance cyclique entre les packages contenant `Synchronisateur` et `Calculateur`.



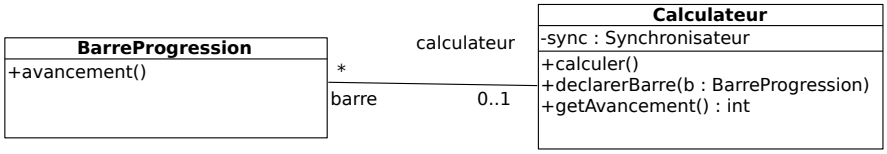
— Si on choisit de déporter la dépendance qui va de la classe `Calculateur` vers `Synchronisateur`, celle-ci est due à l'attribut `sync` qui est un paramètre d'entrée du constructeur de `Calculateur`.
Pour déporter cette cause de dépendance, on va créer une interface `ISynchronisateur` et l'attribut dans `Calculateur` devient de type `ISynchronisateur`. La classe `Synchronisateur` hérite alors de `ISynchronisateur`.



Question 3 *Nous souhaitons ajouter à la classe* ***`Synchronisateur`*** *une opération* ***`ajouterCalculateur()`*** *qui permette d'assigner un calculateur à un synchronisateur, l'identité du calculateur étant un paramètre d'entrée de l'opération. Compléter le diagramme UML obtenu à la question précédente en y ajoutant cette opération correctement définie.*

La seule difficulté est de faire attention à ne pas re-créer un cycle de dépendances en ajoutant cette opération. Les dépendances nouvelles créées lors de l'ajout d'une opération proviennent des paramètres de l'opération. Il ne faut donc pas qu'une dépendance soit créée de la classe `Synchronisateur` vers la classe `Calculateur`. Le type du paramètre de l'opération `ajouterCalculateur()` doit donc être la classe `CalculateurSup` dans le cas de la première solution.

Nous souhaitons maintenant définir une classe représentant une barre de progression. Cette barre affiche l'état d'avancement du calcul (en pourcentage) d'un calculateur. Une barre de progression reçoit des messages d'un calculateur qui l'informe que l'état d'avancement du calcul a changé. Une barre se déclare auprès d'un `Calculateur` et est associée à au plus un `Calculateur`. Le `Calculateur` offre une opération permettant de connaître le pourcentage d'avancement du calcul.

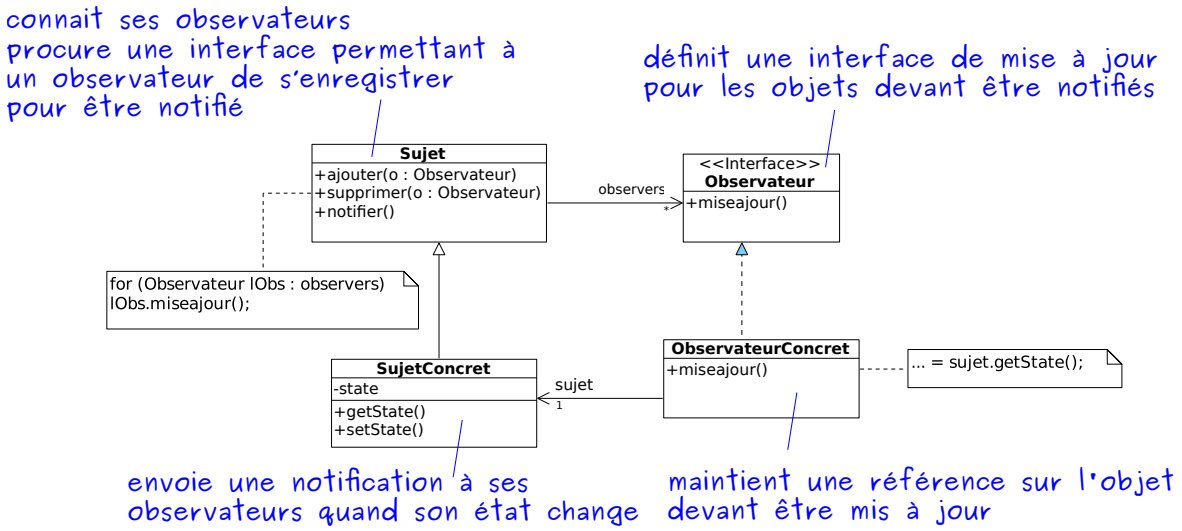


Question 4 *Nous souhaitons que ces deux classes soient dans deux packages différents. Proposez une solution pour respecter le principe des dépendances acycliques en appliquant le patron de conception* ***`Observer`*** *(cf. cours2 sur les design patterns).*

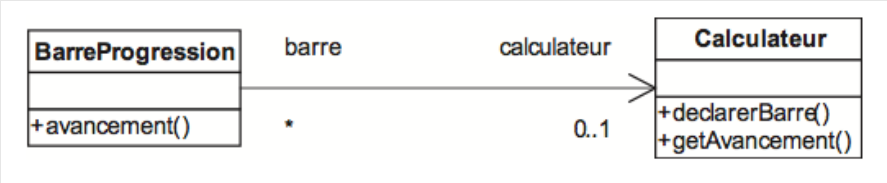
Rappels : Pattern Observateur/Observé (*Observer*)

Les objets observés (`Sujet`) enregistrent dynamiquement des observateurs (`Observateur`) et les notifient des changements lorsque cela est utile (`notifier()`).

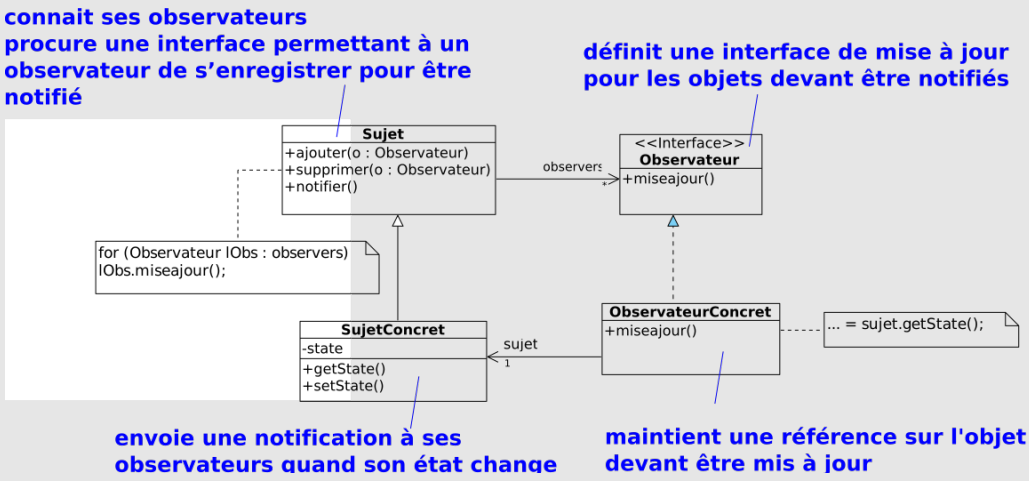
— Faible couplage entre `ObservateurConcret` et `SujetConcret`
— Les données de `Sujet` peuvent être “poussées” (dans *notifier*) ou “tirées” (avec des getters)
— Se retrouve dans de nombreuses API Java



Une première solution possible serait la suivante. Il faut modifier la classe `Calculateur`. La barre de progression devant se déclarer aupreè du calculateur, c'est ce dernier qui contient l'opération `declarerBarre()`. Pour que le calculateur puisse identifier la barre qui se déclare, il faut que son identité soit passée en paramètre de l'opération.
Pour qu'une barre de progression puisse avoir accès à l'état d'avancement du calcul, il faut créer une association allant de la classe `BarreProgression` vers la classe `Calculateur` et ajouter l'opération `getAvancement():integer` dans la classe `Calculateur`. La multiplicité de l'association est 0..1 à l'extrémité du calculateur et 0..∗ à l'extrémité de la barre de progression. Une barre de progression est associée à au plus un calculateur, et nous faisons l'hypothèse qu'un calculateur est associé à plusieurs barres de progression. La figure ci-dessous représente cette solution.



Cependant, on constate que dans cette solution, il y a un cycle de dépendances entre `BarreProgression` et `Calculateur`. Nous devons donc casser ce cycle car l'objectif est ici de pouvoir mettre les 2 classes dans des packages différents. Pour cela, nous allons appliquer le patron de conception Observer (cf. cours 2) qui permet de respecter le principe des dépendances acycliques. En effet, le problème résolu par ce patron de conception est : « Créer un lien entre un objet source et plusieurs objets cibles permettant de notifier les objets cibles lorsque l'état de l'objet source change. De plus, il faut pouvoir dynamiquement lier à (ou délier de) l'objet source autant d'objets cibles que nous le voulons. » La figure ci-dessous présente le patron `Observer`, détaillé dans le Cours 2.



Dans notre cas, l'objet `Observer` (Observateur) est la barre de progression et l'objet `Subject` (Observé) le calculateur, puisque notre problème est d'informer la barre de progression des avancements du calculateur. La figure ci-dessous représente l'application du patron `Observer`.