CPOA - Support TD 2

Dut/Info-S3/M3105 - (Semaine 46)

Version corrigée 

Cette version comporte des indications pour les réponses aux exercices.

PreReq	1. Je sais programmer en Java. 2. J'ai conscience qu'il faut réfléchir avant de se lancer dans le codage. 3. Je maîtrise les concepts objet de base (héritage, polymorphisme, …). 4. J'ai compris ce qu'est un patron et j'ai grand soif d'en apprendre d'autres que *Strategy*
ObjTD	Aborder le patron **singleton**
Durée	**1** TD et **2** TP de 1,5h (sur 2 semaines).

1. La fabrique de chocolat

Vous participez au développement d'un simulateur de fabriques de chocolat modernes dont des bouilleurs sont assistés par ordinateur.

La tâche du bouilleur consiste à contenir un mélange de chocolat et de lait, à le porter à ébullition puis à le transmettre à la phase suivante où il est transformé en plaquettes de chocolat.

1.1. Problème initial

Voici la classe contrôleur du bouilleur industriel de *Bonchoco, SA*.

Contrôleur du bouilleur en Java

```java
/**
 * @author bruel (taken from Design Pattern - Head First, O'Reilly, 09/2004)
 *
 */
public class BouilleurChocolat {
    private boolean vide;
    private boolean bouilli;

    public BouilleurChocolat() {
        vide = true;
        bouilli = false;
    }

    public void remplir() {
        if (estVide()) {
            vide = false;
            bouilli = false;
            // remplir le bouilleur du mélange lait/chocolat
        }
    }

    public void vider() {
        if (!estVide() && estBouilli()) {
            // vider le mélange
            vide = true;
        }
    }

    public void bouillir() {
        if (!estVide() && !estBouilli()) {
            // porter le contenu à ébullition
            bouilli = true;
        }
    }

    public boolean estVide() { return vide;}

    public boolean estBouilli() { return bouilli;}
}
```

QUESTION

1. À quoi servent les attributs `vide` et `bouilli`?

Solution 

Si vous étudiez le code, vous constatez qu'ils ont essayé très soigneusement d'éviter les catastrophes, par exemple de vider deux mille litres de mélange qui n'a pas bouilli, de remplir un bouilleur déjà plein ou de faire bouillir un bouilleur vide !

Vous faîtes un cauchemar horrible (quoi que) où vous vous noyez dans du chocolat. Vous vous réveillez en sursaut avec une crainte terrible.

QUESTION

1. Que pourrait-il se passer avec plusieurs instances de contrôleurs (pour un seul et même bouilleur)?
2. De quoi faudrait-il s'assurer pour éviter ce problème?
3. Trouvez des exemples de situations où il est important de n'avoir qu'une seule instance d'une classe donnée.

Solution 

1. Que l'un remplisse alors que l'autre n'a pas vidé par exemple.
2. S'assurer de n'avoir qu'une seule instance de ce contrôleur.
3. Quelques exemples :
    - accès unique à une base de données (on vient de le voir)
    - objet "parent" d'une interface
    - ...

1.2. Amélioration 1

Vous vous souvenez des premiers exercices Java sur les variables de classe et vous proposez d'utiliser un compteur d'instance pour solutionner le problème.

QUESTION

Vous essayez de modifier le constructeur pour qu'il ne fonctionne que si le compteur d'instance est à 0. Qu'est-ce qui ne va pas dans le code suivant :

```
public class BouilleurCptChocolat {
    private boolean vide;
    private boolean bouilli;
    private static int nbInstance = 0;

    public BouilleurCptChocolat() {
        vide = true;
        bouilli = false;
        if (nbInstance == 0) {
            nbInstance = 1;
            return this;
        }
        else {
            return null;
        }
```

Pas de return dans un constructeur.

1.3. Amélioration 2

Vous changez de stratégie car vous vous souvenez avoir déjà vu ce type de code :

Idée!

```
public class MaClasse {
    private MaClasse() {...}
}
```

QUESTION

1. Est-ce autorisé de rendre privé le constructeur?
2. Comment créer une instance dans ces conditions? N'a-t'on pas tout simplement une classe inutilisable?
3. Complétez le code suivant de façon à résoudre le problème :

```
public class BouilleurChocolat {
    private boolean vide;
    private boolean bouilli;
    ...
    ...

            BouilleurChocolat() {
        ...
        ...
        }

    ...
    ...
    ...
    ...

    public void remplir() {
        if (estVide()) {
            vide = false;
            bouilli = false;
            // remplir le bouilleur du me  lange lait/chocolat }
        }
        // reste du code de BouilleurChocolat...
}
```

4. Donnez un exemple d'utilisation de cette classe.

1. Oui!
2. En implémentant une fonction qui s'en charge.

Extrait de la solution

```
public class BouilleurSafeChocolat {
    private boolean vide;
    private boolean bouilli;
    private static BouilleurSafeChocolat uniqueInstance;

    private BouilleurSafeChocolat() {
        vide = true;
        bouilli = false;
        }

    public static final BouilleurSafeChocolat getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new BouilleurSafeChocolat();
        }
        return uniqueInstance;
    }
```

1.4. C'est pas fini!

Vos cauchemars continuent! Mais cette fois ils sont en anglais! Vous voyez un grand gaillard irlandais vous menacer (en fait vous confondez *threat* et *thread*...).

QUESTION

1. En quoi les *threads* peuvent-ils poser des problèmes dans votre solution?
2. Recopiez sur des bouts de feuilles les fragments de code ci-dessous en les plaçant dans les colonnes du tableau suivant pour mettre en évidence le problème en reconstituant un enchaînement erroné possible avec deux threads. :

Thread 1	**Thread 2**	**Valeur de `uniqueInstance`**

Bloc 1

```
public static BouilleurChocolat getInstance() {
```

Bloc 2

```
if (uniqueInstance == null) {
```

Bloc 3

```
uniqueInstance = new BouilleurSafeChocolat();
```

Bloc 4

```
        }
```

Bloc 5

```
        return uniqueInstance;
```

Bloc 6

```
    }
```

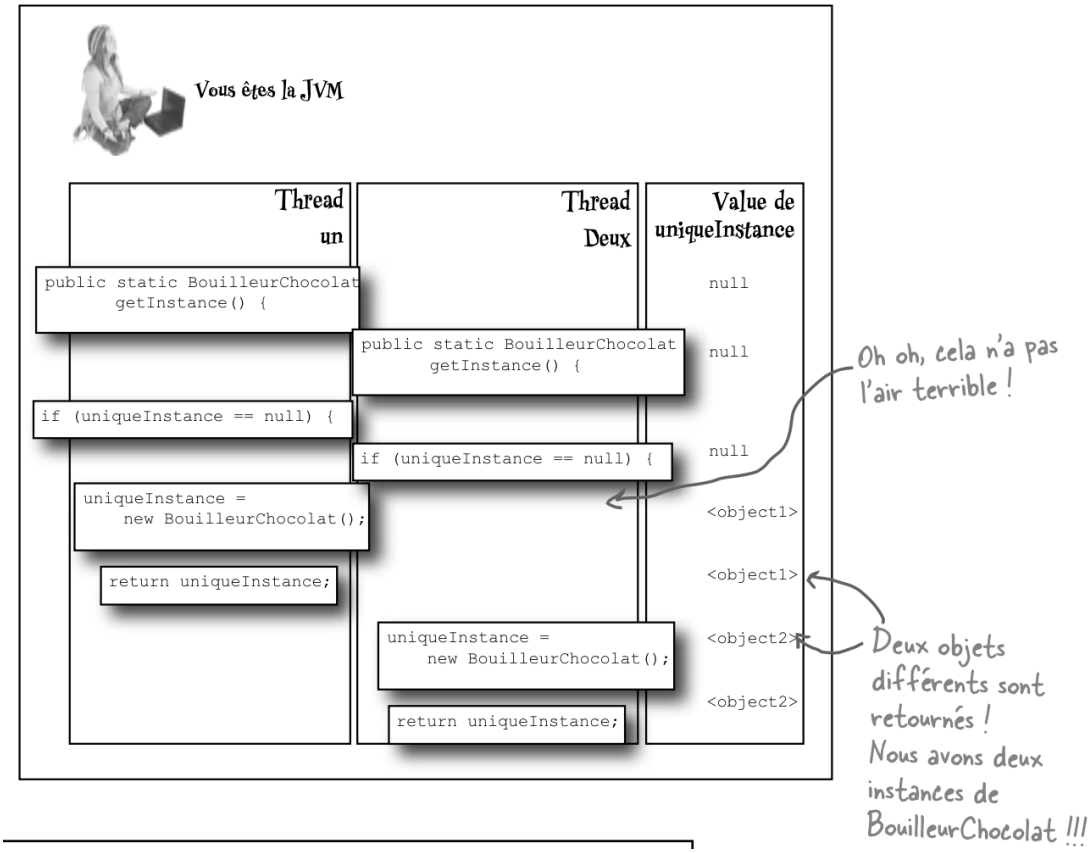


Figure 1. Solution (source [Freeman05])

```
public class BouilleurSafeChocolat {
    private boolean vide;
    private boolean bouilli;
    private static BouilleurSafeChocolat uniqueInstance;

    private BouilleurSafeChocolat() {
      vide = true;
      bouilli = false;
    }

    public static final BouilleurSafeChocolat getInstance() {
      if (uniqueInstance == null) {
        uniqueInstance = new BouilleurSafeChocolat();
      }
      return uniqueInstance;
    }
```

Explications :

1. Thread 1 appelle `getInstance()` et détermine que `uniqueInstance` est `null` en ligne 12
2. Thread 1 entre dans le bloc `if` puis est préempté par le thread 2 avant l'exécution de la ligne 13
3. Thread 2 appelle `getInstance()` et détermine que `uniqueInstance` est `null` en ligne 12
4. Thread 2 entre dans le bloc `if`, crée un nouveau `BouilleurSafeChocolat` et assigne ce nouvel objet à la variable `uniqueInstance` en ligne 13
5. Thread 2 retourne la référence au `BouilleurSafeChocolat` en ligne 15
6. Thread 2 est préempté par le Thread 1
7. Thread 1 reprend où il s'était arrêté et exécute la ligne 13 créant alors une autre instance de `BouilleurSafeChocolat`
8. Thread 1 retourne cette nouvelle instance en ligne 15

1.5. Solution au multithreading

Vous vous souvenez heureusement de vos cours de début d'année sur les *threads* :

QUESTION

1. Proposez une solution simple à ce problème.

Il suffit de faire de `getInstance()` une méthode **synchronisée** :

```
public class BouilleurSafeChocolat {
  private boolean vide;
  private boolean bouilli;
  private static BouilleurSafeChocolat uniqueInstance;

  private BouilleurSafeChocolat() {
    vide = true;
    bouilli = false;
  }

  public static synchronized BouilleurSafeChocolat getInstance() {
    if (uniqueInstance == null) {
      uniqueInstance = new BouilleurSafeChocolat();
    }
    return uniqueInstance;
  }
```

1.6. Problème de la solution!!

QUESTION

1. Combien de fois le mécanisme mis en place va-t'il être utile ?
2. Que pensez-vous alors de cette solution ?
3. Proposez une solution où l'instance est créé au démarrage plutôt qu'à la demande.

1. Une seule fois, lors du 1er passage dans la méthode!!
2. C'est bien trop consomateur en ressource! En pratique, il y a des copies de blocs de mémoire, ce qui prend du temps.
3. Voici un exemple :

Création de l'instance unique au démarrage

```
public class Singleton {
    private static final Singleton uniqueInstance = new
Singleton();
    private Singleton() {}
    public static Singleton getInstance() { return uniqueInstance;}
}
```

En adoptant cette approche, nous nous reposons sur la JVM pour créer l'unique instance du Singleton quand la classe est chargée. La JVM garantit que l'instance sera créée avant qu'un thread quelconque n'accède à la variable statique `uniqueInstance`.

Il peut y avoir des situations où le coût de la synchronisation est inférieur au coût de créer dès le départ une instance (par exemple gourmande en mémoire).

2. Singleton

Félicitations, vous venez de mettre en oeuvre votre deuxième patron, le **Singleton**.

Design pattern : **Singleton**

Singleton garantit qu'une classe n'a qu'une seule instance et fournit un point d'accès global à cette instance.

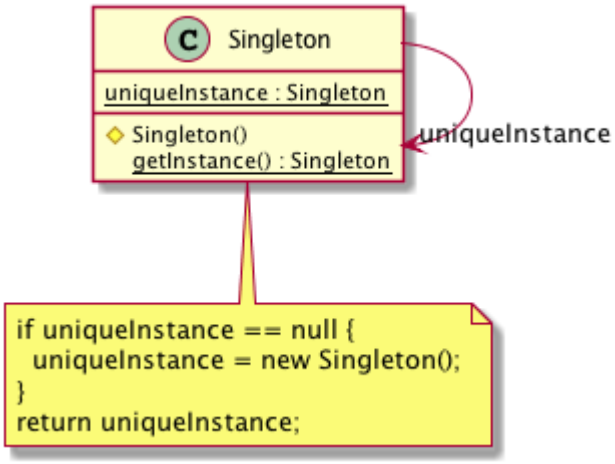


Figure 2. Modèle UML du patron Singleton

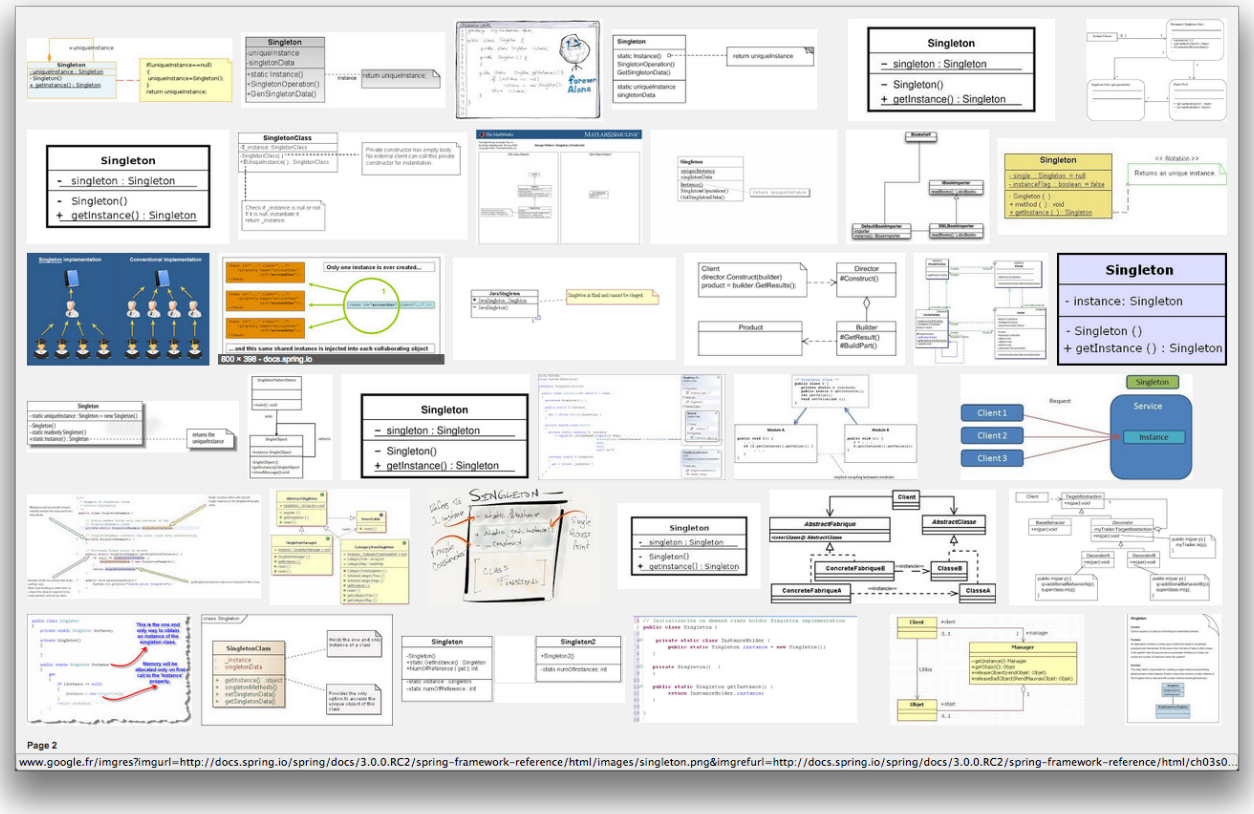


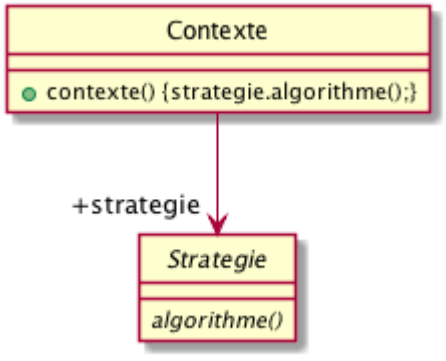
Figure 3. Quelques exemples de description du patron Singleton

3. Le singleton pour le jeu d'aventure

3.1. Combiner plusieurs patrons?

Peut-on combiner les deux derniers patrons vus en TD (*Strategy* et Singleton)? En effet, les comportements sont portés par des objets pour l'aspect algorithme, mais il n'y a pas de raison de ne pas les partager entre tous les objets qui "utilisent" ce comportement?!

Dans la plupart des cas ces deux patrons ne vont **pas du tout ensemble**. Cette stratégie n'est recommandée que dans un cas bien précis d'utilisation de *Strategy* : celui où les comportements sont simples et "statiques" (pas de consommation de ressources par exemple) et où l'on utilise une association :



Avec une implémentation du type :

```
...
vol = new VolerAvecDesAiles();
cri = new Cancan();
c1 = new Colvert(vol,cri);
...
```

3.2. Et si on améliorait le jeu d'aventure avec Singleton?

QUESTION

1. Faites en sorte que les instances d'objet affectées à chaque comportement d'un `Personnage` soient uniques pour chaque comportement distinct.
2. Pourquoi ne devrait-on pas utiliser `getInstance()` dans le cas d'une composition (dans le constructeur du composé) ?

1. Extrait de solution (disponible sur le GitHub pour les profs)

   Extrait de ComportemenEpee.java

   ```
   public class ComportementEpee implements ComportementArme {

       private final static ComportementEpee uniqueInstance = new
   ComportementEpee();

       public final static  ComportementEpee getInstance() {
           return uniqueInstance;
       }
   ```

   Qu'on pourra éventuellement enrichir pour être complet avec :

   ```
   private ComportementEpee(){

       }
   ```

   Extrait de Chevalier.java

   ```
   public class Chevalier extends Personnage {

       protected Chevalier(String nom) {
           this(nom, ComportementEpee.getInstance(),
   ComportementACheval.getInstance());
       }
   ```

   Il faudra alors changer les appels comme `compAdequat = new ComportementEpee();` en `compAdequat = ComportementEpee.getInstance();`

2. Car dans une composition les objets possèdent les instances de leur comportement. Elles sont donc uniques et leurs instances ne doivent pas être transmises, pour être sûr que la destruction du composite détruit les composés.

On voit que ce n'est pas toujours évident de combiner les patrons entre eux.

Pour aller plus loin

QUESTION

Quelle est la différence entre un singleton et une variable globale?

Quelques éléments de solution :

- En Java les variables globales sont des références statiques à des objets.
- Problème déjà vu de l'instanciation à la demande vs. au démarrage.

QUESTION

Comment testeriez-vous la mise en oeuvre du patron Singleton?

Exemples de test :

- Tentative d'instanciation depuis l'extérieur de la classe
- Tentative de construction de deux objets de type Singleton

QUESTION

Il existe une autre façon de gérer le problème du multithreading. Cherchez sur Internet les articles sur le "verrouillage à double vérification" (qui ne fonctionne que depuis Java `1.5`).

N'hésitez pas à consulter les liens suivants :

- http://thecodersbreakfast.net/index.php?post/2008/02/25/26-de-la-bonne-implementation-du-singleton-en-java
- http://christophej.developpez.com/tutoriel/java/singleton/multithread/