De la conception UML vers son implémentation JAVA

Sujet

Au niveau du BurgerResto

Nous souhaitons mettre en place deux fonctionnalités :

- Chaque fois que le client fait une nouvelle commande elle s'affiche sur l'ensemble des écrans des cuisiniers.
- A minuit toutes les commandes qui n'ont pas été récupérées sont retirées.

Au niveau des objectifs Java

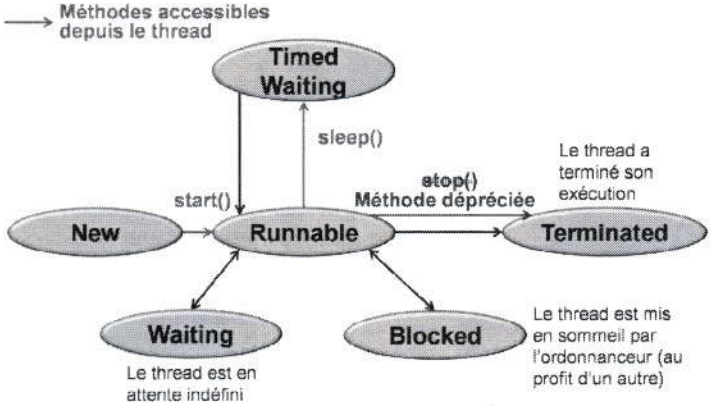
Durant cette séance de TD nous travaillerons sur :

- Le pattern Observer,
- Les threads Java.

Rappel de cours

Les threads

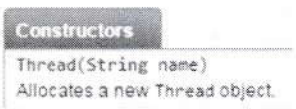
Commençons par visualiser les différents états d'un thread



Puis nous allons travailler à partir d'un exemple sur les différents états d'un thread.

Les états : NEW / RUNNABLE

Nous choisissons d'utiliser le constructeur permettant de nommer le thread.



A sa création le thread est dans l'état « NEW ».

```
public static void main(String[] args) {
    Thread myThread = new Thread("MyThread");
    System.out.println(myThread.getName() + " : " + myThread.getState());
}
```

Console

```
MyThread : NEW
```

A partir de là on peut invoquer la méthode start() pour qu'il soit prêt à travailler, le thread est alors dans l'état « RUNNABLE »

```
public static void main(String[] args) {
    Thread myThread = new Thread("MyThread");
    myThread.start();
    System.out.println(myThread.getName() + " : " + myThread.getState());
}
```

Console

```
MyThread : RUNNABLE
```

L'état : TERMINATED

On attend que le thread se termine en affichant ces états successifs.

```
public static void main(String[] args) {
    Thread myThread = new Thread("MyThread");
    myThread.start();
    for (int i = 0; i < 2; i++) {
        System.out.println(myThread.getName() + " : " + myThread.getState());
    }
}
```

Console

```
MyThread RUNNABLE
MyThread TERMINATED
```

L'écriture d'un thread

Pour écrire un thread il faut qu'il hérite de la classe « Thread ».
Dans cet exemple on choisit le constructeur permettant de le nommer.

```
public class MyThread extends Thread{

    public MyThread(String newName) {
        super(newName);
    }

    public void run() {
    }
}
```

Constructors

`Thread(String name)`
Allocates a new Thread object.

L'état : TIMED_WAITING

Nous allons endormir le thread. Donc dans le run du thread

```
public class MyThread extends Thread {

    public MyThread(String newName) {
        super(newName);
    }

    public void run() {
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

La méthode *sleep* doit être appelée en static (à partir de la classe Thread et non de l'objet this). Cette méthode peut générer une exception d'où sa place à l'intérieur d'un block try / catch.

Dans la classe Test :

```
public class Test {

    public static void main(String[] args) {
        MyThread myThread = new MyThread("MyThread");
        myThread.start();
        System.out.println(myThread.getName() + " : " + myThread.getState());
    }
}
```

Console

```
MyThread : TIMED_WAITING
```

Le thread principal : le main

A savoir que lorsque vous lancez un programme avec votre méthode main vous l'exécutez sur un thread : le thread principal.

```
public class Test {

    public static void main(String[] args) {
        System.out.println(Thread.currentThread().getName() + " : "
                + Thread.currentThread().getState());
    }
}
```

Console

```
main : RUNNABLE
```

Remarque : La méthode `static currentThread` de la classe « Thread » permet de connaitre le thread qui est en train de s'exécuter.

Thread en concurrence

Si on demande aux deux threads d'écrire leur nom dans la console, ils le feront en concurrence.

```
public class Test {

    public static void main(String[] args) {
        MyThread myThread = new MyThread("MyThread");
        myThread.start();
        for (int i = 0; i < 3; i++) {
            System.out.println(Thread.currentThread().getName());
        }
    }
}
public class MyThread extends Thread {

    public MyThread(String newName) {
        super(newName);
    }

    public void run() {
        for (int i = 0; i < 3; i++) {
            System.out.println(this.getName());
        }
    }
}
```

Différentes sorties possibles :

```
main        main      main
MyThread    MyThread  main
main        MyThread  main
MyThread    MyThread  MyThread
main        main      MyThread
MyThread    main      MyThread
```

Faire boucler un thread

Un thread peut boucler jusqu'à ce qu'une condition soit vrai. Par exemple un thread peut boucler jusqu'à ce que l'utilisateur ferme l'application ( ). Un évènement est alors émis par la fenêtre qui peut être captée par le contrôleur qui peut arrêter l'ensemble des threads.

Exemple simple : le thread principal démarre « myThread » dort durant 1000 ms puis l'arrête.

```
public class Test {

    public static void main(String[] args) {
        MyThread myThread = new MyThread("MyThread");
        myThread.start();

        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        myThread.arret();
    }
}
```

Faisons maintenant boucler le thread :

```
public class MyThread extends Thread {

    boolean condition = true;

    public MyThread(String newName) {
        super(newName);
    }

    public void arret() {
        this.condition = false;
    }

    public void run() {
        int i = 0;
        do {
            System.out.println(this.getName() + " : Dans la boucle "+i);
            i++;
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        } while (condition);
        System.out.println(this.getName() + " : Sortie de la boucle");
    }
}
```

Tant que le thread n'est pas arrêté, il affiche le nombre de boucle qu'il a fait, puis dort 500 ms. Une fois arrêté il affiche qu'il est sortie de la boucle puis passe dans l'état « TERMINATED »

Proprement on devrait utiliser des wait dans les threads et les réveiller par des notify ...

Vu l'ensemble des concepts à absorber et le peu de temps restons simple ...

Donc vu la méthode que l'on utilise prendre en compte l'avertissement suivant :
ATTENTION Faire dormir les threads régulièrement sinon votre temps CPU va exploser !

Le pattern Observer

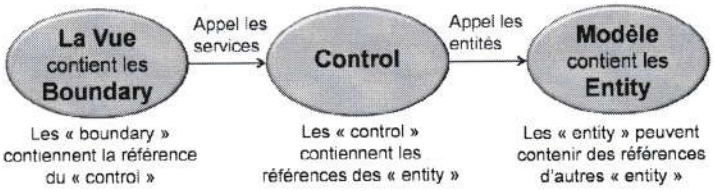
Le principe

Le pattern Observateur (en anglais Observer) définit une relation entre objets de type un-à-plusieurs, de façon que, si un objet change d'état, tous ceux qui en dépendent en soient informés et mis à jour automatiquement.

Ce pattern est souvent utilisé dans un contexte MVC. En effet dans ce contexte :

- la vue propose des actions à effectuer aux utilisateurs. La vue connaissant le contrôleur, utilise les services proposés par celui-ci pour répondre aux actions utilisateurs.
- Le contrôleur, connaissant les entités du modèle, interagit avec elles pour répondre aux services qu'il déclare.

Donc dans le contexte MVC, plus particulièrement dans une implémentation UML des stéréotypes Boundary, Control, Entity on obtient le schéma ci-dessous.



Les entités n'ont pas la référence du control et le control n'a pas la référence de la vue.
C'est pourquoi si dans la conception le modèle doit mettre à jour la vue il doit passer par le pattern Observer.

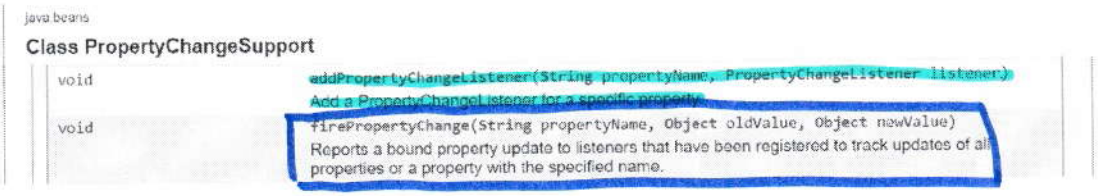
Mise en place du pattern Observer en JAVA.

Pour faire le lien entre la vue est le modèle jusqu'à Java 9 on utilisait l'interface *Observer* et la classe « Observable », or depuis Java 9 l'interface *Observer* et la classe « Observable » ont été dépréciées.

Une solution possible est d'utiliser l'interface *PropertyChangeListener*, et les classes « PropertyChangeEvent » et « PropertyChangeSupport » du package java.beans

Un objet de la classe « PropertyChangeSupport » permet de simplifier la gestion d'une liste d'écouteurs (listener) en les informant des changements de valeur d'une propriété. Cette classe définit les méthodes :

- `addPropertyChangeListener()` pour enregistrer un composant désirant être informé du changement de la valeur de la propriété,
- `removePropertyChangeListener()` pour supprimer un composant de la liste.
- `firePropertyChange()` pour informer tous les composants enregistrés du changement de la valeur de la propriété.



- L'entité observée possède :
  - un attribut (par exemple support) de type « PropertyChangeSupport »,
  - une méthode permettant d'ajouter un écouteur (de type « PropertyChangeListener ») à la liste d'écouteur géré par l'attribut support.
- La vue :
  - implémente l'interface *PropertyChangeListener*,
  - donne sa référence au contrôleur afin qu'il puisse la transmettre à l'entité observé : `addPropertyChangeListener`
  - écoute l'environnement pour capter un évènement concernant une entité qu'il observe : `propertyChange`

Pour que l'entité observée agisse sur la vue :

- L'entité envoie les modifications avec la méthode ***firePropertyChange(String propertyName, PropertyChangeListener listener)***,
- La vue reçoit les modifications ***propertyChange(PropertyChangeEvent evt)*** et met à jour son interface utilisateur en conséquence.

Package java.beans

Class PropertyChangeEvent

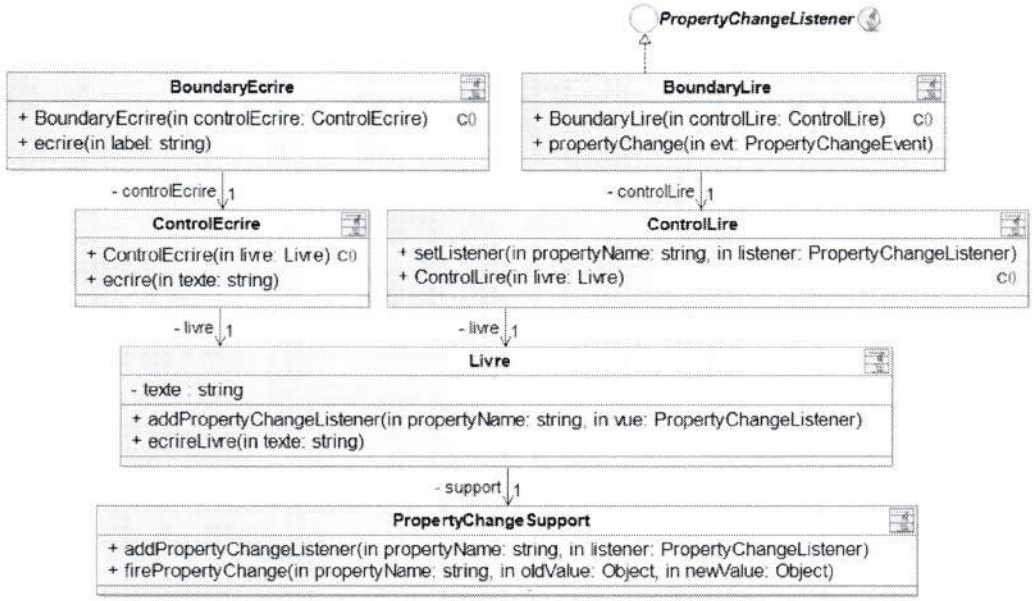
Method Summary

All Methods	Instance Methods	Concrete Methods

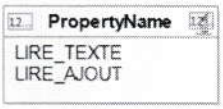
Modifier and Type	Method	Description
Object	getNewValue()	Gets the new value for the property, expressed as an Object.
Object	getOldValue()	Gets the old value for the property, expressed as an Object.
String	getPropertyName()	Gets the programmatic name of the property that was changed.

Exemple d'implémentation

Un écrivain écrit dans un livre. A chaque fois qu'il écrit, les lecteurs peuvent lire le texte sur leur écran. Nous sommes dans un contexte MVC. L'objet livre sera l'entité observée et le ou les objets de la classe « BoundaryLecteur » les observateurs.

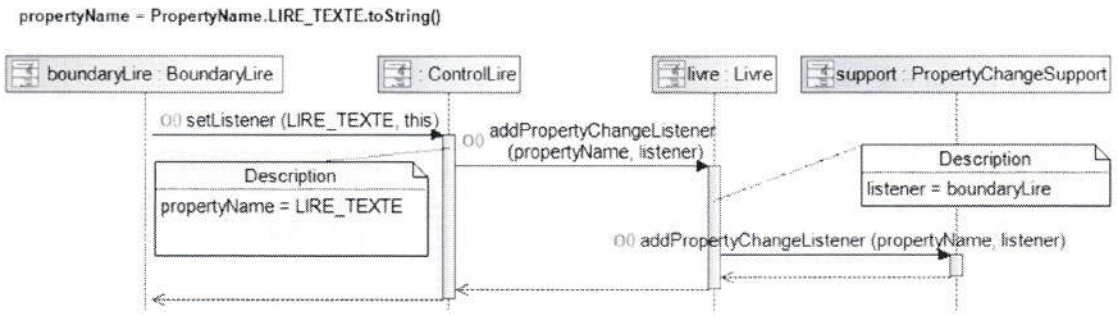


Pour plus de sureté nous utiliserons un énuméré au lieu d'une chaine de caractères.



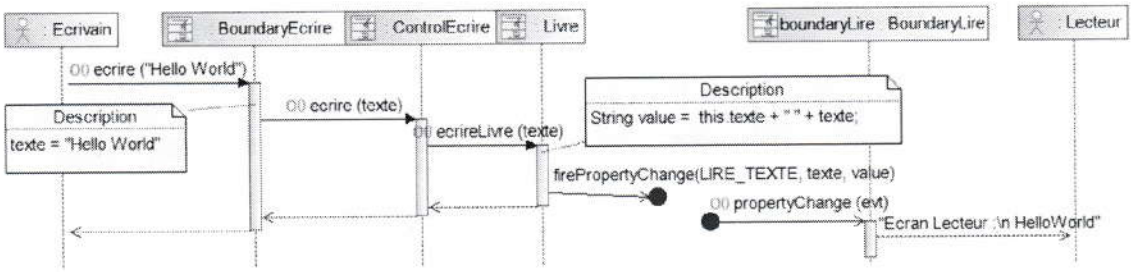
Mise en place du lien entre le modèle et la vue.

Tout d'abord les observateurs doivent s'enregistrer à l'objet observé pour une propriété donnée : à sa création, l'objet boundaryLire, demande au contrôleur (qui connait toutes les entités du cas) de l'ajouter comme observateur du livre. Cet ajout se fait dans l'objet observé (ici le livre) grâce à la méthode ***addPropertyChangeListener*** hérité de la classe « PropertyChangeSupport ».



Envoie et réception d'évènement du modèle vers la vue.

Une fois que tout est en place nous pouvons nous intéresser au cas « écrire ». Par l'intermédiaire de son boundary, l'écrivain écrit « Hello Word ». Le contrôleur fait passer le message au livre en appelant sa méthode ***ecrireLivre***. Cette méthode concatène le texte au texte existant. En ce qui concerne l'écrivain le cas est fini.



Sauf qu'en ce qui concerne les lecteurs, il faut leur afficher le nouveau texte. Donc la méthode ***ecrireLivre*** concatène le texte au texte existant, mais prévient aussi les observateurs que la classe a été modifiée ! Pour cela il utilise la méthode ***firePropertyChange*** de la classe « PropertyChangeSupport » prenant en paramètre d'entrée le nom de la propriété qui a été modifiée, sa valeur précédente et sa nouvelle valeur. Avec ces informations, la méthode crée un évènement qui est transmis à tous les observateurs enregistrés pour connaitre les changements de cette propriété.

L'information parvient aux observateurs par la méthode ***propertyChange*** avec pour argument l'évènement construit par l'objet observé. L'objet boundaryLire peut donc afficher le texte à l'écran du lecteur.

Code Java de l'exemple.

```
public class BoundaryEcrire {
 private ControlEcrire controlEcrire;
  public BoundaryEcrire(ControlEcrire controlEcrire) {
   this.controlEcrire = controlEcrire;
  }
  public void ecrire(String texte) {
   controlEcrire.ecrire(texte);
  }
}

public class ControlEcrire {
  private Livre livre;
  public ControlEcrire(Livre livre) {
   this.livre = livre;
  }
  public void ecrire(String texte) {
   this.livre.ecrireLivre(texte);
  }
}

public class Livre {
 private PropertyChangeSupport support
   = new PropertyChangeSupport(this);
 private String texte = "";

 public void addPropertyChangeListener(String propertyName,
    PropertyChangeListener listener) {
  support.addPropertyChangeListener(propertyName, listener);
 }

 public void ecrireLivre(String texte) {
  String value = this.texte + " " + texte;
  support.firePropertyChange(
     PropertyName.LIRE_TEXTE.toString(),
     this.texte, value);
  this.texte = value;
 }
}
```

```
public class BoundaryLire implements PropertyChangeListener {
 private ControlLire controlLire;
 public BoundaryLire(ControlLire controlLire) {
  this.controlLire = controlLire;
  this.controlLire.setListener(
   PropertyName.LIRE_TEXTE.toString(),
   this);
 }
 @Override
 public void propertyChange(PropertyChangeEvent evt) {
  String propertyName = evt.getPropertyName();
  PropertyName choix = PropertyName.valueOf(propertyName);
  switch (choix) {
   case LIRE_TEXTE:
    String newValue = (String) evt.getNewValue();
    System.out.println("Ecran du lecteur phrase :");
    System.out.println(newValue);
    break;
   default:
    System.out.println("type de message inconnu");
    break;
  }
 }
}

public class ControlLire {
 private Livre livre;

 public ControlLire(Livre livre) {
  this.livre = livre;
 }

 public void setListener(
    String propertyName, PropertyChangeListener listener) {
  livre.addPropertyChangeListener(propertyName, listener);
 }
}

public enum PropertyName {
 LIRE_TEXTE, LIRE_AJOUT;
}
```

Travail à effectuer pour le quatrième TD

Travail sur les threads.

Nous voulons que tous les jours à minuit toutes les commandes qui n'ont pas été récupérées soient retirées.
Compléter la classe « ThreadViderCommande » en page 8 selon les étapes ci-dessous.

- Spécifier la classe « ThreadViderCommande » comme étant un thread qui boucle (cf. partie cours).
- Ensuite créer l'attribut **jourReference** de type entier qui sera initialisé par le constructeur avec le jour d'aujourd'hui :

```
//création de l'instance avec le point temporel courant
Calendar calendar = Calendar.getInstance();
//accès au jour du point temporel calendar
jourReference = calendar.get(Calendar.DAY_OF_MONTH);
```

- Dans la boucle du run, faire dormir le thread 1 minute (60000), puis lire la date d'aujourd'hui et la comparer à la date de référence, si ce n'est pas la même alors modifier la date de référence et afficher à l'écran « vider commandes »,

En séance de TD / TP : pour pouvoir réellement tester notre travail, nous allons faire notre test sur les minutes (et non sur le jour !). Par exemple s'il est 11H52 nous effacerons les commandes à 11 H 54

Javadoc java.util.Calendar

Modifier and Type	Method and Description
`static Calendar`	`getInstance()` Gets a calendar using the default time zone and locale.
`static int`	`DAY_OF_MONTH` Field number for `get` and `set` indicating the day of the month.
`static int`	`MINUTE` Field number for `get` and `set` indicating the minute within the hour.

```
public class ThreadViderCommandeJour extends Thread {
    private boolean condition = true;
    private int jourReference;
    public ThreadViderCommandeJour() {
        Calendar calendar = Calendar.getInstance();
        jourReference = calendar.get(Calendar.DAY_OF_MONTH);
    }
    public void arret() {
        condition = false;
    }
    public void run() {
        do {
            try {
                Thread.sleep(60000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            // En reel
            Calendar calendar = Calendar.getInstance();
            int jour = calendar.get(Calendar.DAY_OF_MONTH);
            if(jour != jourReference){
                jourReference = jour;
                System.out.println("Vider Commande");
            }
            // Pour le test
            Calendar calendar = Calendar.getInstance();
            int minute = calendar.get(Calendar.MINUTE);
            if(minute == 54) {
                System.out.println("Vider Commande");
            }

        } while (condition);
    }
}
```

Travail sur le pattern Observer.

Nous voulons qu'à chaque fois que le client effectue une nouvelle commande, elle s'affiche sur l'ensemble des écrans des cuisiniers.

Diagramme de cas

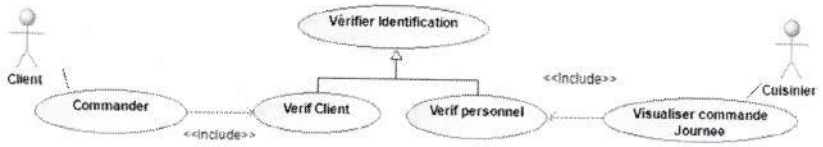


Diagramme de séquence détaillé

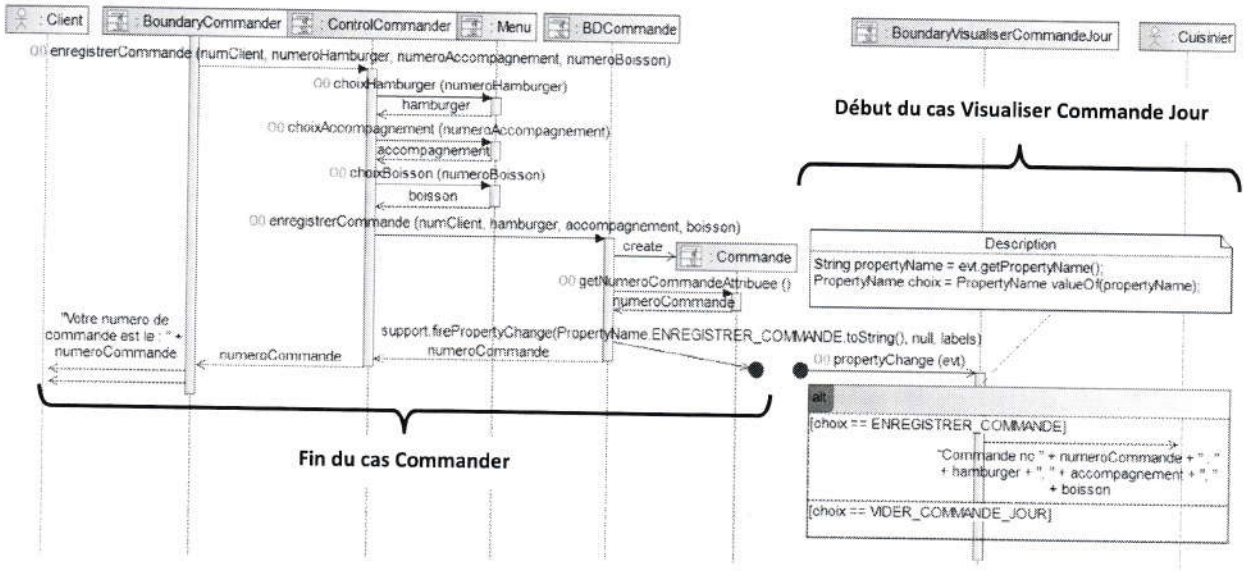
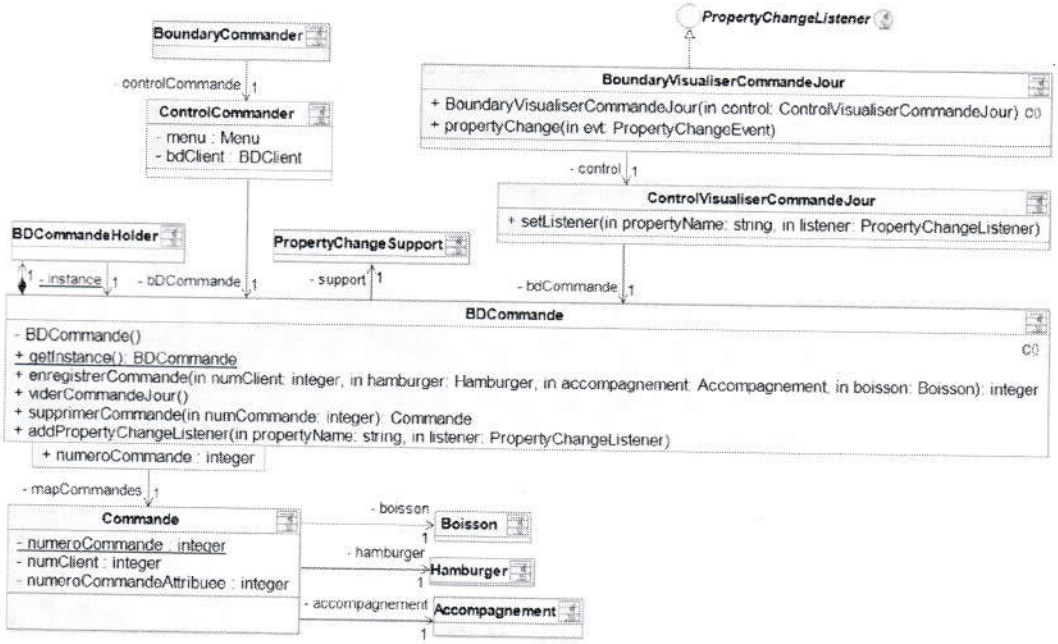


Diagramme de classes participantes simplifié



Questions

Compléter les classes données en pages 11 et 12 selon les étapes décrites ci-dessous.

Q1- Identification des rôles.

Nous avons identifié sur les diagrammes UML le boundary « BoundaryVisualiserCommandeJour » comme étant l'observateur et l'entité « BDCommande » comme l'observé. Compléter la signature du boundary « BoundaryVisualiserCommandeJour », et les attributs de la classe « BDCommande ».

Q2- Initialisation : lien entre l'observateur et l'observé.

a) Ecrire dans la classe observée « BDCommande » la méthode *addPropertyChangeListener* qui prend une chaine (propertyName) et un objet de type « PropertyChangeListener » (listener) en paramètre d'entrée et l'ajoute en tant qu'observateur à la classe observée (par l'intermédiaire de la méthode *addPropertyChangeListener* de l'attribut support de type « PropertyChangeSupport »).
b) Ecrire dans le contrôleur du cas « Visualisation de la commande du jour » la méthode *setListener* qui prend une chaine (propertyName) et un objet de type « PropertyChangeListener » (listener) en paramètre d'entrée et l'ajoute en tant qu'observateur à la classe Observée par l'intermédiaire de la méthode que l'on vient d'écrire (*addPropertyChangeListener*).
c) Dans le constructeur de l'observateur « BoundaryVisualiserCommandeJour », appeler la méthode que l'on vient d'écrire (*setListener*) pour l'enregistrer auprès de l'observée « BDCommande » en tant qu'observateur pour la propriété PropertyName.***ENREGISTRER_COMMANDE***.

Q3- Changement dans la classe observée.

Lorsqu'on crée une nouvelle commande, la classe Observée prévient les observateurs qui l'observent des changements. Compléter la méthode *enregistrerCommande* suivant les étapes ci-dessous.

a) Créer un tableau de chaine labels contenant 4 chaînes :
    1. La case 0 contiendra le numéro de la commande,
    2. La case 1 contiendra le nom de l'hamburger,
    3. La case 2 contiendra le nom de l'accompagnement,
    4. La case 3 contiendra le nom de la boisson.
b) Notifier aux observateurs que la classe a changé pour la propriété PropertyName.***ENREGISTRER_COMMANDE*** et lui envoyer null en ancienne valeur et le tableau en nouvelle valeur.

Q4- Traitement des changements dans l'observateur

Compléter la méthode *propertyChange* de l'observateur.

a) Récupérer le nom de la propriété qui a été modifiée, et le caster en PropertyName.
b) Selon le nom de la propriété, si ce dernier est ***ENREGISTRER_COMMANDE***, alors récupérer dans les cases allant de 0 à 3, respectivement, le numéro de la commande, le nom de l'hamburger, le nom de l'accompagnement et le nom de la boisson. Les afficher dans la console.
c) Par défaut écrire dans la console : « Affichage non reconnu ».

Diagramme de classes

Extrait du cas « commander »

```
public enum PropertyName {
        ENREGISTRER_COMMANDE, SUPPRIMER_COMMANDE, VIDER_COMMANDE_JOUR;
}


public class ControlCommander {
    private Menu menu = Menu.getInstance();
    private BDCommande bdCommande = BDCommande.getInstance();

    public int enregistrerCommande(int numClient, int numeroHamburger, int
                numeroAccompagnement, int numeroBoisson) {
        Hamburger hamburger = menu.choixHamburger(numeroHamburger);
        Accompagnement accompagnement = menu
                .choixAccompagnement(numeroAccompagnement);
        Boisson boisson = menu.choixBoisson(numeroBoisson);

        return bdCommande.enregistrerCommande(numClient, hamburger,
                accompagnement, boisson);
    }
}


public class BDCommande {

    private Map<Integer, Commande> mapCommandes = new HashMap<Integer, Commande>();
```

Q1 → private PropertyChangeSupport support = new PropertyChangeSupport(this)

```
    public void addPropertyChangeListener(String propertyName,
        PropertyChangeListener listener) {
```

Q2 support.addPropertyChangeListener(propertyName, listener);

```
}

    public int enregistrerCommande(int numClient, Hamburger hamburger,
            Accompagnement accompagnement, Boisson boisson) {
        Commande commande = new Commande(numClient, hamburger, boisson, accompagnement);
        int numCommande = commande.getNumeroCommandeAttribuee();
        mapCommandes.put(numCommande, commande);
```

Q3

```
String[] labels = new String[4];
labels[0] = String.valueOf(numCommande);
labels[1] = hamburger.getNom();
labels[2] = accompagnement.getNom();
labels[3] = boisson.getNom();
support.firePropertyChange(PropertyName.ENREGISTRER_COMMANDE.
    toString(), null, labels);
```

```
        return numCommande;
    }
}
```

Extrait du cas « visualiser commande du jour »

Q1
```
public class BoundaryVisualiserCommandeJour implements PropertyChangeListener
{
    ControlVisualiserCommandeJour control;

Q2  public BoundaryVisualiserCommandeJour(ControlVisualiserCommandeJour control) {
        this.control = control;
        this.control.setListener(PropertyName.ENREGISTRER_COMMANDE.
        toString(), this);
    }

Q4  public void propertyChange(PropertyChangeEvent evt) {
        String propertyName = evt.PropertyName();
        PropertyName choix = PropertyName.valueOf(propertyName);
        switch(choix){
            case ENREGISTRER_COMMANDE:
                Object objet = evt.getNewValue;
                String[] labels = (String[]) objet;
                String numeroCommande = labels[0];
                String hamburger = labels[1];
                String accompagnement = labels[2];
                String boisson = labels[3];
                System.out.println("Commande n°" + numeroCommande
                    + ";
                default : System.out.println("numero d'affichage
                    non reconnu");
                    break;
            }
        }
}

public class ControlVisualiserCommandeJour {
    BDCommande bdCommande = BDCommande.getInstance();

Q2  public void setListener(String propertyName, PropertyChangeListener listener) {
        bdCommande.addPropertyChangeListener(propertyName, listener);
    }
}
```