Université de Montréal

Travail Pratique #3 (Maintenance)

Par
Charbel Kassis(p0976458)

Baccalauréat en Informatique
Faculté des arts et sciences

Travail présenté à Mohamed Aymen SAIED et Edouard BATOT dans le cadre du cours IFT2255 Génie Logiciel

Décembre 2015

Surligné = du nouveau code
(…)= bloc de code non modifié

Maintenance correctif :

1- Afin d'ajouter une inscription qui indique le démarrage et la fermeture d'un guichet, il faudrait ajouter dans la classe **Journal** deux nouvelles méthodes: `logStartUp() et logShutDown()`. Ces méthodes, comme les autres se trouvant dans la classe **Journal,** vont chercher l'instance de la simulation courante puis impriment un message spécifique dans la console. `LogStartUp()` sera appelée dans la méthode `performStartUp()` de la classe **Guichet,** `logShutDown()` sera appelée dans la méthode `performShutDown()` de la même classe. Dans les deux cas, la date de l'événement sera affichée.

Dans la classe **Journal**

```
public void  logStartUp() {

   String date = new SimpleDateFormat("yyyy/MM/dd HH:mm:ss").format(new Date());
   Simulation.getInstance().printLogLine("Start Up: "+date);
}

public void logShutDown() {

   String date = new SimpleDateFormat("yyyy/MM/dd HH:mm:ss").format(new Date());
   Simulation.getInstance().printLogLine("Shut down: "+date);
}
```

Dans la classe **Guichet**

```
private void performStartup()
{
   log.logStartUp();
   (…)
}

private void performShutdown()
{
   log.logShutDown();
   networkToBank.closeConnection();
}
```

2- Afin d'avertir le client qu'il n'y pas eu d'activités depuis un certain temps ***TEMPS_LIMITE*** (5000 par défaut) en millisecondes, il faudrait émettre un certain nombres ***MAX_NB_BIPS*** de bips sonores (3 par défaut) à un certain intervalle (le temps d'inactivité ***TEMPS_LIMITE*** est utilisé comme intervalle) de temps pour finallement forcer l'éjection de la carte au dernier intervalle de temps. Pour faire ceci, il faudrait modifier la méthode `readInput(int mode, int maxValue)`dans la classe **SimClavier** pour qu'elle initialise un **Timer** du package **javax.swing** qui émettera un bip sonore ou force l'éjection de la carte(méthode `forceCancel() )` selon le nombre de bips qui ont été émis.

Dans la classe **SimClavier**

```
import javax.swing.Timer;
```

```
synchronized String readInput(int mode, int maxValue)
{
```

voir code source, cette méthode a été partiellement réécrite: Ajout d'un BeepListener (classe interne qui implémente ActionListener) et d'un Timer, le timer exécute la méthode actionPerformed du BeepListener à chaque intervalle de TEMPS_LIMITE millisecondes.

```
}
```

Toute méthode qui est affectée par le timer a été mise à jour pour en prendre compte, lorsqu'on entre un digit pour saisir un montant (et le numéro de la facture de la question 5) le timer est arrêté, le nombre de beeps est mis à zéro puis le timer redémarre :`stopAndReset();`

```
public synchronized void forceCancel() {

        timer.stop();
        cancelKeyPressed();
 }

private BeepListener beepListener;

private Timer timer;

private static final int TEMPS_LIMITE = 5000;

public static final int MAX_NB_BIPS = 3;

class BeepListener implements ActionListener { Voir code source }
```

Maintenance adaptative :

3- Afin de permettre à l'utilisateur de reçevoir des billets de 5$, il faut modifier la méthode `getSpecificsFromCustomer()` de la classe **Retrait** de manière à ce que le client ait l'option de retirer 5$, 10$ et 15$, en plus des autres options. Il faudrait aussi modifier la manière d'initialiser l'argent qui sera dans la machine: il faudra demander le nombre 5$ en plus de celui des 20$. De plus, puisqu'il y a deux types d'argents, il faut les compter afin de déterminer s'il reste assez de l'un ou l'autre pour faire un retrait d'argent.

Il faut d'abord demander à l'utilisateur de saisir un nombre de 20$ et un nombre de 5$ à mettre dans la machine, pour faire ceci il faut créer un deuxième **PanneauBillet** dans la classe **GUI.** Pour qu'il affiche le bon message, le constructeur de **PanneauBillet** prendra en paramètre la valeur du billet. Dans la méthode `getInitialCash()` de la classe **GUI**, on affiche le premier panneau; on saisit la valeur et on affiche le second paneau; on saisit la valeur. Les valeurs saisies seront stockées dans des nouvelles propriétées `private int nbOfBills` de la classe **PanneauBillet** puis `private int nbOf20` et `private int nbOf5` de la classe **Simulation.**

Ces valeurs seront utilisées par la méthode `checkCashOnHand(Argent amount)` de la classe

DistribiteurArgent afin de calculer la possibilité de retirer de l'argent sachant qu'il ne reste qu'un certain nombre de 5$ et de 20$. Pour faire ceci, un algorithme vorace sera utilisé dans la méthode pour déterminer quel type d'argent sera utilisé ainsi que la quantité: Le programme essaiera de n'utiliser que des 20$, mais si ce n'est pas possible il utilisera des 5$. Le nombre de billets dans la machine sera mis à jour dans la méthode `dispenseCash(Argent amount)` la classe **DistribiteurArgent** en utilisant les propriétée `private int nbOf20Used` et `nbOf5Used`.

Dans la classe **Retrait**

```
protected Message getSpecificsFromCustomer() throws ConsoleClient.Cancelled
{
    (…)
     String [] amountOptions = { "$5", "$10", "$15", "$20", "$40", "$60", "$100",
                                 "$200" };
    Argent [] amountValues = {
                                   new Argent(5), new Argent(10), new Argent(15), new
                                 Argent(20), new Argent(40), new Argent(60),
                                   new Argent(100), new Argent(200)
                                 };
  (…)
 }
```

Dans la classe **PanneauGuichet**

```
public static final int DISPLAYABLE_LINES =     12;
```

Dans la classe **PanneauBillet**

```
PanneauBillets(int amount)
{
  (…)
    add(new Label("de billets de $"+amount+" dans le distributeur d'argent.",
                  Label.CENTER));
  (…)
    add(new Label("Veuillez entrer le nombre de billets de $"+amount+".",
                  Label.CENTER));
   (…)
}

synchronized int readBills()
{
   (…)
    this.nbOfBills = billsNumber;
    return billsNumber;
}
```

l'ajout de `nbOfBills` ainsi que son «get»

Dans la classe **GUI**

```
GUI(SimPanneauOperation operatorPanel,(…))

{
    (…)
    billsPanel = new PanneauBillets(20);
    add(billsPanel, "BILLETS20");

    billsPanel2  = new PanneauBillets(5);
    add(billsPanel2, "BILLET5");
}
public Argent getInitialCash()
{
    mainLayout.show(this, "BILLETS20");
    int numberOfBills20 = billsPanel.readBills();
    mainLayout.show(this, "BILLET5");
    int numberOfBills5 = billsPanel2.readBills();
    mainLayout.show(this, "GUICHET");
    Simulation.getInstance().setNbOf20(billsPanel.getNbOfBills());
    Simulation.getInstance().setNbOf5(billsPanel2.getNbOfBills());
    return new Argent(20 * numberOfBills20 + 5* numberOfBills5);
}
```

l'ajout de `billsPanel2` ainsi que son «get»

Dans la classe **Simulation**

l'ajout de `nbOf20` et `nbOf5`, ainsi que leurs «get» et «set» respectives

Dans la classe **DistributerArgent**

```
public boolean checkCashOnHand(Argent amount)
{
   (Voir le code source, cette méthode a été complètement réécrite : si il n'y a pas
   assez d'argent dans la machine, retourner false sinon exécuter un algorithme vorace
   pour déterminer le nombre de billets à envoyer au client, retourne false s'il n'y
   pas assez de 20$ et de 5$ pour former le montant nécessaire, retourne true sinon.
   Dans les deux cas, nbOf20Used et nbOf5Used est mis à jour pour potentiellement
   être utilisées dans la méthode dispenseCash() )
}

public void dispenseCash(Argent amount)
{
   Simulation.getInstance().setNbOf20(Simulation.getInstance().getNbOf20()-nbOf20Used);
   Simulation.getInstance().setNbOf5(Simulation.getInstance().getNbOf5()-nbOf5Used);
   (…)
}
```

Maintenance perfective :

4- Pour mentioner le fait qu'une carte a était retenue par la machine, il faudrait modifier la méthode `retainCard()` de la classe **LecteurCarte** pour recevoir en paramètre la carte à retenir. Cette information sera utilisée dans la méthode `logCardRetained(compte.Carte carte)` de la classe

Journal pour afficher le numéro de la carte retenue par la machine, ainsi que la date à laquelle cette action a été effectuée. Il faudrait aussi mettre à jour l'appel de la méthode `retainCard()` dans la classe **Transaction.**

Dans la classe **LecteurCarte**

```
public void retainCard(Carte carte)
{
atm.getLog().logCardRetained(carte);
  Simulation.getInstance().retainCard();
}
```

Dans la classe **Journal**

```
public void logCardRetained(compte.Carte carte) {

String date = new SimpleDateFormat("yyyy/MM/dd HH:mm:ss").format(new Date());
Simulation.getInstance().printLogLine("Retained: card "+ carte.getNumber() + " @
"+date);
}
```

Dans la classe **Transaction**

```
public Etat performInvalidPINExtension() throws ConsoleClient.Cancelled,CardRetained
{
(…)
  atm.getCardReader().retainCard(card);\
(…)
}
```

5- Pour ajouter l'option de payer une facture, il faudrait s'assurer de trois éléments :
(1) L'utilisateur doit entrer un numéro de facture.
(2) Un message avec le numéro de la facture doit être envoyé à la banque.
(3) La banque doit pouvoir traiter les demandes de payement de facture.

Il faudrait d'abord ajouter un nouveau type de menu dans ***TRANSACTION_TYPES_MENU*** et un nouveau type de Transaction dans la méthode `makeTransaction()` de la classe **Transaction** pour donner la possibilité de payer la facture au client.

Dans la nouvelle classe **PayementFacture** qui hérite de **Transaction,** la méthode `getSpecificsFromCustomer()` demandera le compte sur lequel l'utilisateur veut payer sa facture. Ensuite, pour satisfaire (1), on demande à l'utilisateur d'entrer le numéro de la facture. Il faudrait donc créer un mode de lecture du numéro de la facture en créant un nouveau mode dans la classe **SimClavier.**

Pour satisfaire (2), il faudrait crée un message à partir de la méthode `getSpecificsFromCustomer()` de la classe **PayementFacture** qui prend en compte le numéro de la facture, donc le message sera de type **MessageFacture** qui héritera de **Message.**

Pour satisfaire (3), la banque qui reçoit le message détectera que c'est un payement de facture,

puis appelera la méthode `billPayement()` de la classe **BanqueSimulee** pour déterminer si oui ou non le payement sera accepté.

Dans la classe **Transaction**

```
public static Transaction makeTransaction(...) throws ConsoleClient.Cancelled
{
          (…)
         case 4:

          return new PayementFacture(atm, session, card, pin);
          (…)
}

 private static final String [] TRANSACTION_TYPES_MENU =
     { "Retrait", "Depot", "Transfert", "Demande solde", "Payement de facture"};
```

Dans la classe **PayementFacture**

Voir la nouvelle classe dans le code source

Dans la classe **Message**

```
 public String toString()
 {
 (…)
         case BILL_PAYEMENT:

          result += "FACTURE ";
          break;
 (…)
 }

 public static final int BILL_PAYEMENT = 5;
```

Dans la classe **MessageFacture**

voir la nouvelle classe dans le code source

Dans la classe **ConsoleClient**

```
 public int readBillNumber(String prompt) throws Cancelled {

    Simulation.getInstance().clearDisplay();
      Simulation.getInstance().display(prompt);
      Simulation.getInstance().display("");

      String input = Simulation.getInstance().readInput(Simulation.BILL_MODE, 0);

      Simulation.getInstance().clearDisplay();

      if (input == null)
          throw new Cancelled();
```

```
        else
            return Integer.parseInt(input);
    }
```

Dans la classe **Simulation**

```
public static final int BILL_MODE = 4;
```

Dans la classe **SimClavier**

Les méthodes digitKeyPressed(**int** digit), enterKeyPressed(), clearKeyPressed() et cancelKeyPressed() ont été modifiés pour prendre en compte le nouveau mode ***BILL_MODE.***

```
private static final int BILL_MODE = Simulation.BILL_MODE;
```

Dans la classe **BanqueSimulee**

Ajout d'un constructeur qui initialize un HashMap dont la clé est le numéro de compte et la valeur est un HashMap dont la clé est le numéro de la facture et la valeur est le montant de la facture.

```
public Etat handleMessage(Message message, Balances balances)
{
   (…)

         case Message.BILL_PAYEMENT:

          return billPayement(message,balance);


   (…)
}

 private Etat billPayement(Message message) {
```

Voir code source, cette méthode retourne un new Failure() si le numéro de compte est invalide ou que le numéro de la facture n'existe pas pour l'utilisateur en question ou qu'il n'y a pas assez d'argents sur le compte ou que le montant entré dépasse le montant de la facture. Elle met à jour les balances et le montant restant à payer pour la facture et retourne new Success() sinon.

}