Design Patterns - TP4

https://github.com/IUT-Blagnac/cpoa-tp4-template

Code initial pour le TP4

Rappel du cours : http://bit.ly/jmb-cpoa

Informations générales

NOM

BRUEL

Prénom

Jean-Michel

Groupe #

[X] Enseignants

- [ ] 1
- [ ] 2
- [ ] 3
- [ ] 4
- [ ] Innopolis

Pré-requis

Il vous faut :

- [x] Un compte GitHub
- [ ] Un terminal de type Git Bash (si vous utilisez Window$)

Essayez la commande suivante dans votre terminal pour vérifier votre environnement `git` :

```
git config --global -l
```

Tâche initiale

- [x] Clickez sur le lien Github Classroom fourni par votre enseignant (en fait c'est déjà fait si vous lisez ces lignes).

- ☐ Clonez sur votre machine le projet Github généré pour vous par Github Classroom.
- ☐ Modifez le `README` pour modifier Nom, Prénom et Groupe.
- ☐ Commit & push:

commit/push

```
fix #0 Initial task done
```

Dans la suite de ce document, à chaque fois que vous trouverez un énoncé commençant par `fix #⋯` vous devez vérifier que vos scripts/fichiers modifiés sont bien dans votre dépôt local en vue de committer et de pusher les modifications sur votre dépôt distant en utilisant comme message de commit cet énoncé.

- Si vous voulez vérifier que vous êtes prêt pour le `fix #0`, utilisez la commande : `make check`.
- Si vous voulez avoir la liste des ToDos de ce TP/TP, exécutez `make todos`.

Les exercices de ce TD sont tirés de l'excellent livre "Tête la première : Design Pattern". Bert Bates, Eric Freeman, Elisabeth Freeman, Kathy Sierra. Editions O'Reilly. 2005.

Le but : "Objet-ivez" votre code

Le problème

Reprendre l'application initiale de gestion de comptes bancaires (fichier TPMenus_applictionBanque.zip sur votre repo). Cette application utilise 3 classes principales :

- la classe `Compte` : les comptes bancaires proprement dits
- la classe `AgenceBancaire` : classe composée de `Compte` permettant de gérer une agence bancaire
- la classe `ApplicationAgenceBancaire` définissant le code fonctionnel de l'application : un jeu de

menu permettant à un opérateur de banque de réaliser des opérations courantes.

Cette dernière classe (`ApplicationAgenceBancaire`) a été écrite dans une approche purement procédurale (ensemble de méthodes statiques).

Le but

Le but de ce TP est de passer d'une approche fonctionnelle/procédurale du code applicatif à une approche objet. Nous allons réécrire et refactorer le code de l'application en plusieurs étapes pour obtenir un code :

- plus facile à maintenir car plus clair,
- plus facile à étendre,
- donc plus robuste aux changements.

Etude et reprise de l'existant

1. Etudier le code proposé pour `ApplicationAgenceBancaire`.
2. Comprendre la structure des méthodes (menus, méthodes réalisant les opérations)
3. Ajouter les fonctions et sous-menu correspondant à l'exécution suivante :

```
--
  Agence CAISSE ECUREUIL de PIBRAC
  Menu Général
--
  Choisir :
    1 - Liste des comptes de l'agence
    2 - Voir un compte (par son numéro)
    3 - Menu opérations sur comptes
    4 - Menu gestion des comptes

    0 - Pour quitter ce menu
Votre choix ?
3

--
  Agence CAISSE ECUREUIL de PIBRAC
  Menu opérations sur comptes
--
  Choisir :
    1 - Déposer de l'argent sur un compte
    2 - Retirer de l'argent sur un compte

    0 - Pour quitter ce menu
Votre choix ?
0
Fin de Menu opérations sur comptes
```

```
--
  Agence CAISSE ECUREUIL de PIBRAC
  Menu Général
--
  Choisir :
    1 - Liste des comptes de l'agence
    2 - Voir un compte (par son numéro)
    3 - Menu opérations sur comptes
    4 - Menu gestion des comptes

    0 - Pour quitter ce menu
Votre choix ?
4
--
  Agence CAISSE ECUREUIL de PIBRAC
  Menu gestion des comptes
--
  Choisir :
    1 - Ajouter un compte
    2 - Supprimer un compte

    0 - Pour quitter ce menu
Votre choix ?
0

--
  Agence CAISSE ECUREUIL de PIBRAC
  Menu Général
--
  Choisir :
    1 - Liste des comptes de l'agence
    2 - Voir un compte (par son numéro)
    3 - Menu opérations sur comptes
    4 - Menu gestion des comptes

    0 - Pour quitter ce menu
Votre choix ?
```

1. Rencontrez-vous des difficultés pour ajouter des fonctions dans ce code qui devient "spaghetti" ?
2. Que pensez-vous de la maintenance de ce code dans 4 ans par un autre programmeur et qui devra ajouter des cascades de menus et de fonctions ?
3. N'avez-vous pas programmé plusieurs fois la même chose pour faire les menus à l'écran ?

"Objet-iver" les fonctions

Principe

Vous pouvez réfléchir 5 minutes avant de commencer cette partie : qu'est ce qui pourrait devenir objet et quelles classes seront à créer ?

Nous allons modifier le code en plusieurs classes en observant que :

1. Chaque fonction utilisateur pourrait être programmée séparément sous forme d'un objet que nous appellerons `Action` (option de menu) possédant :
   a. le message affiché à l'écran pour "afficher" l'action dans un menu,
   b. une méthode pour exécuter cette option de menu.
2. Un menu pourrait être programmé séparément sous forme d'un objet que nous appellerons `ActionList` (liste d'actions de menus) possédant :
   a. le message affiché à l'écran pour "afficher" le menu comme un sous-menu de menu,
   b. des méthodes pour ajouter/retirer des options de menus dans ce menu,
   c. une méthode pour exécuter cette ce menu (affichage et déclenchement des actions).

Les fonctions utilisateurs comme des objets

1. Faire une copie du code source précédent sous le nom `applicationBanqueAction`.
2. Créer les packages suivants :

```
application
application.action
application.actionlist
```

3. Etudier la définition d'interface suivante et insérer sa définition dans votre projet dans le package `action` :

```java
package action;
import banque.AgenceBancaire;
/**
 * An Action is an object that implements some action of a user's menu.<BR>
 * It is defined by a message, an optional code, an execute method to "do" the
action.
 */
public interface Action  {
    /**
     * Message of the action (to show on screen).
     *
     * @return the message of the action
     */
    public String actionMessage ();

    /**
     * Code of the action (may be used to identify the action among other ones).
     *
     * @return the code of the action
     */
    public String actionCode ();

    /**
     * The method to call in order to "execute" the action on <code>ag</code>.
     *
     * @param ag the AgenceBancaire on which the action may act on
     * @throws Exception when an uncaught exception occurs during execution
    */
    public void execute(AgenceBancaire ag) throws Exception;
}
```

4. Déclarer une classe par action (option de menu) à utiliser. Commencer par "Liste des comptes de l'agence" :
    a. Créer une classe (le nom `ActionListeDesComptes` paraît adapté) dans le package `application.action`,
    b. qui implémente `Action`,
    c. dotée de deux attributs (message, code)
    d. écrire le code dont un constructeur correctement paramétré,
    e. la méthode `execute(AgenceBancaire)` réalisera l'affichage écran de la liste des comptes de l'agence bancaire en paramètre.
5. De la même manière, déclarer une classe pour l'action "Voir un compte (par son numéro)" (classe `ActionVoirCompteNumero`) dans le package `application.action`.

Les menus utilisateurs comme des objets

1. Etudier la définition d'interface suivante et insérer sa définition dans votre projet dans le

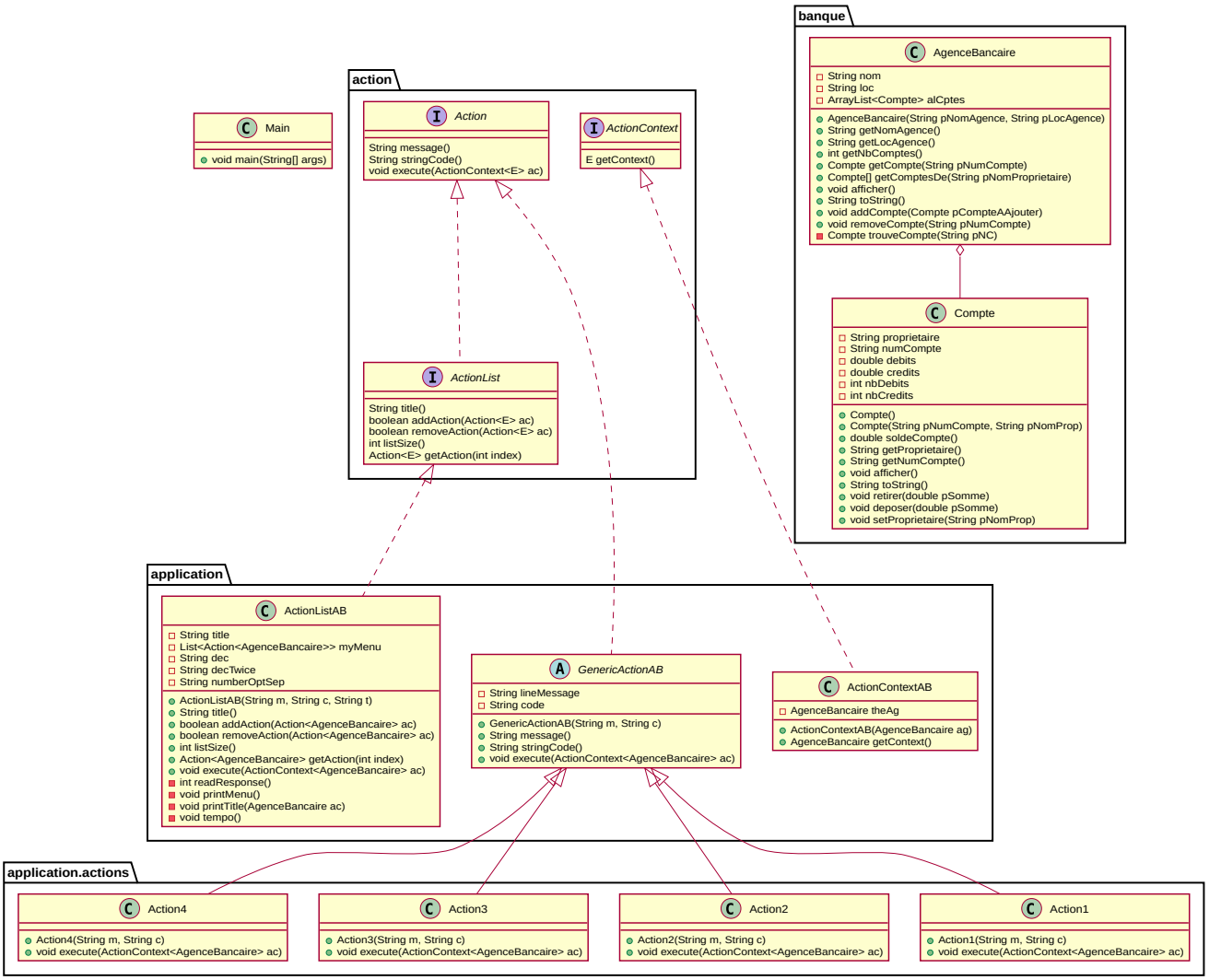
package `action` :

```
package action;

/**
 * An ActionList is an object that implements a end-user menu.<BR>
 * It is defined by a title (printed on top of the menu).<BR>
 * It is also defined by a list of different action objects that the menu
manages.<BR>
 * It is attended to :<BR>
 * - display the end-user menu numbered from 1 (list of messages of actions).<BR>
 * - display a quit option (0).<BR>
 * - wait for some user response.<BR>
 * - launch the requested action.<BR>
 */
public interface ActionList extends Action {
    /**
     * Title of the list of actions (menu).
     *
     * @return the title of the action list
     */
    public String listTitle();

    /**
     * The number of actions in the action list.
     *
     * @return number of actions in the action list.
     */
    public int size();

    /**
     * Add an action at the end of the list action if it does not yet exists.
     *
     * @param ac the action to add
     * @return true if action is added, else false
     */
    public boolean addAction(Action ac);
}
```

2. Déclarer une classe `ActionListAgenceBancaire` dans le package `application.actionlist`,
    a. qui implémente `ActionList`,
    b. dotée de quatre attributs (message, code, title, liste des actions). La liste des actions sera les différentes options que l'action list (menu) devra afficher.
    c. écrire le code dont un constructeur correctement paramétré,
    d. la méthode `execute(AgenceBancaire)` réalisera ce qui est défini dans la documentation. Le menu affiché sera identique à celui de la version antérieure de l'application. Chaque option de menu sera numérotée par `execute()` de 1 à n (nombre d'action) + 0 pour sortir du menu.

Vous devez obtenir une architecture comme celle-ci (attention, légèrement différente, avec un attribut code par exemple) :

Structure de l'application



Et maintenant : go ! Maintenance et extension facilitées

1. Créer une classe contenant un main et permettant :
   a. de créer une instance de chaque classe `Action` créée,
   b. de créer une instance de `ActionListAgenceBancaire`,
   c. lancer `execute()` sur l'instance de `ActionListAgenceBancaire`.

Ca marche ?

2. Vous pouvez créer les autres actions et sous-menus.
3. Pourquoi ActionList hérite de Action à votre avis ?

On aurait pu utiliser un autre patron appelé Composite … plus tard peut être

Abstraire le problème

Une nouvelle application ... et mince ...

Supposons que nous devions développer une application de gestion d'une liste d'élèves (classes Eleve et GroupeEleve). Elle est basée sur un menu permettant de :

- Voir la liste des élèves.
- Afficher un élève à partir du nom.
- Modifier les notes d'une élève.
- Ajouter un élève dans le groupe.
- Retirer un élève du groupe.
- ...

Ca vous rappelle des choses ?

Questions :

1. Considérant les nouvelles classes `Eleve` et `GroupeEleve`, peut-on réutiliser telles quelles (sans les modifier) les interfaces `Action` et `ActionList` dans la nouvelle application ?
2. Si oui : pourquoi ?
3. Si non : pourquoi ?

Abstrayons un peu le problème

Compte tenu des observations de la section précédente, il faudrait des classes `Action` et `ActionList` dont `execute()` prendrait en paramètre n'importe quel objet. Utiliser la classe Object ? Non, la généricité est là pour nous aider ...

1. Faire une copie du code source précédent sous le nom `applicationBanqueActionGenerique`.
2. Modifier les déclarations des interfaces `Action` et `ActionList` comme suit (attention, tout le code va devenir "erroné") :

```
package action;
/**
 * An Action is an object that implements some action of a user's menu.<BR>
 * It is defined by a message, an optional code, an execute method to "do" the
action.<BR>
 * It is parameterized by the  type of object on which the action may act on
(execute on).
 *
 * @param <E> The type of object on which the action may act on.
 */
public interface Action <E> {
    /**
```

```
     * Message of the action (to show on screen).
     *
     * @return the message of the action
     */
    public String actionMessage ();

    /**
     * Code of the action (may be used to identify the action among an action
list).
     *
     * @return the code of the action
     */
    public String actionCode ();

    /**
     * The method to call in order to "execute" the action on <code>e</code>.
     *
     * @param e the object on which the action may act on
     * @throws Exception when an uncaught exception occurs during execution
     */
    public void execute(E e) throws Exception;
}

package action;
/**
 * An ActionList is an object that implements a end-user menu.<BR>
 * It is defined by a title (printed on top of the menu).<BR>
 * It is also defined by a list of different action objects that the menu
manages.<BR>
 * It is attended to :<BR>
 * - display the end-user menu numbered from 1 (list of messages of actions).<BR>
 * - display a quit option (0).<BR>
 * - wait for some user-response.<BR>
 * - launch the requested action.<BR>
 *
 * It is parameterized by the  type of object on which the actions of the list
action may act on (execute on).<BR>
 *
 * @param <E> The type of object on which the list action may act on.
 */
public interface ActionList<E> extends Action<E>{
    /**
     * Title of the list of actions (menu).
     *
     * @return the title of the action list
     */
    public String listTitle();

    /**
     * The number of actions in the action list.
     *
```

```
     * @return number of actions in the action list.
     */
    public int size();

    /**
     * Add an action at the end of the list action if it does not yet exists.
     *
     * @param ac the action to add
     * @return true if action is added, else false
     */
    public boolean addAction(Action<E> ac);
}
```

3. Modifier chaque classe créée (les `Action` puis `ActionList` puis le `main()`) pour soit implémenter la bonne instanciation des interfaces, soit instancier correctement les objets.
4. Tout doit fonctionner.
5. Il n'y a plus qu'à faire la nouvelle application.

Pour aller plus loin : complétons et encore plus d'abstraction

Une interface `ActionList` plus complète

1. Faire une copie du projet précédent
2. Pour de vraies applications, ajouter à l'interface `ActionList` les opérations suivantes :

```
    /**
     * Add an action in the list action at the specified index if it does not yet
exists.
     *
     * @param ac the action to add
     * @param index index to add the action
     * @return true if action is added, else false
     * @throws IndexOutOfBoundsException if (index < 0) || (index > size())
     */
    public boolean addAction(Action<E> ac, int index);

    /**
     * Remove an action from the list action at the specified index.
     *
     * @param index index to remove the action
     * @return true
     * @throws IndexOutOfBoundsException if (index < 0) || (index > size())
     */
    public boolean removeAction(int index);

    /**
     * Remove an action from the list action.
     *
     * @param ac the action to remove
     * @return true if action is removed (found), else false
     */
    public boolean removeAction(Action<E> ac);

    /**
     * List of the messages of actions contained in the action list
     *
     * @return an array of messages of the list action
     */
    public String[] listOfActions() ;
}
```

Quid d'`ActionList` ?

La classe `ActionListAgenceBancaire` qui met en oeuvre un menu (qui implémente `ActionList`) est ici créée spécifiquement pour `AgenceBancaire`. Mais cela est il nécessaire dans chaque application ? (en supposant ne rien afficher de l'`AgenceBancaire`). On devrait pouvoir faire une classe "générique" de gestion de menus composés d'actions et réutilisable dans chaque application.

Alors essayons :

1. Faire une copie du projet de la section précédente
2. Déplacer la classe `ActionListAgenceBancaire` dans le package `action`.
3. Renommer cette classe en un nom contenant "ActionList" et bien choisi. AbstractActionList

serait TRES mal choisi.

4. Pour rendre cette classe générique (et non pas abstraite), modifier l'en-tête en

```
public class GenericActionList<E>
    implements ActionList<E>
```

5. Attention, tout le code va maintenant "klaxonner" en rouge ! normal ...
6. Modifier chaque fois que nécessaire pour utiliser le type générique E
7. Enlever tout accès à la classe `AgenceBancaire` (affichage nom de la banque, ...)
8. Vous devriez arriver au bout ...
9. Enfin dans le main il y aura quelques "klaxons warnings" sur la création d'objets de cette nouvelle classe car il faudra indiquer le type paramètre à la création.

ATTENTION : faire une classe générique n'est pas toujours aussi simple. Ici le cas a été simplifié à l'extrême.

Troisième étape : abstraction encore plus

Le problème :

1. Supposons que l'on veuille utiliser notre application dans une système différent où saisies et affichages ne se font pas sur le terminal d'exécution de l'application ... Les instructions utilisant `new Scanner (System.in)` ou `System.out.println ⋯` deviennent obsolètes.
2. Tout comme l'agence utilisée dans les traitements, ces 2 éléments font maintenant partie du **contexte d'exécution** des actions.
3. D'autres éléments pourraient être utilisés : des transactions en cours (réservations aériennes), des bases de données, des connexions diverses, ...
4. Il faut donc créer un **contexte d'exécution** qui sera en paramètre des `Action` et `ActionList`.

Allons-y !

1. Faire une copie du projet de la section précédente (sans généricité).
2. Dans le package `application`, créer une classe `ApplicationContextAgenceBancaire` implémentant le pattern `Singleton` permettant d'accéder :
   a. A l'agence bancaire "en cours".
   b. Au `Scanner` à utiliser. L'initialiser ici avec un Scanner sur System.in mais autre chose pourrait être utilisé (un fichier, un flux vers un terminal, ...).
   c. A la sortie `PrintStream` à utiliser. Ici ce sera System.out mais autre chose pourrait être utilisé (un fichier, un flux vers un terminal, ...).
3. Refactorer tout le code :
   a. Les classes `Action` et `ActionList` utilisant maintenant le type

`ApplicationContextAgenceBancaire` (à la place de `AgenceBancaire`)

b. Modifier les accès à l'agence bancaire en utilisant `ApplicationContextAgenceBancaire`.

c. Modifier les accès à l'entrée standard en utilisant `ApplicationContextAgenceBancaire`.

d. Modifier les accès à la sortie standard en utilisant `ApplicationContextAgenceBancaire`.

4. Ca marche ??

commit/push

```
fix #All: Completed all duties
```

Contributeurs

- Jean-Michel Bruel

À propos...