Design Patterns - TP4

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

TP4 initial code

This is a template for the students' assignments.

Course material: http://bit.ly/jmb-cpoa

Assignment info

LAST NAME

BRUEL

First Name

Jean-Michel

Group #

[X] Teachers

- ☐ 1
- ☐ 2
- ☐ 3
- ☐ 4
- ☐ Innopolis

Requirements

You'll need:

- ☑ A GitHub account
- ☐ A Git Bash terminal (if you use Window$)

Try the following command in your terminal to check your `git` environment:

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

Initial tasks

- ☑ Click on the Github Classroom link provided by your teacher (in fact, this should be done if you

read this).

- ☐ Clone on your machine the Github project generated by Github Classroom.
- ☐ Modify the README file to add your last name, first name and group number.
- ☐ Commit and push using the following message:

commit/push

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

In the following, every time you'll see à `fix #⋯` text, make sure all your files are committed, and then push your modifications in the distant repo, making sure you used the corresponding message (`fix #⋯`) in one of the `commit` messages.

- If you want to check that you're really ready for `fix #0`, you can run the command in your shell: `make check`.
- If you want to list the ToDos of the day, run `make todos`.

This TD exercise is inspired from the excellent book: "Head First: Design Pattern. Bert Bates, Eric Freeman, Elisabeth Freeman, Kathy Sierra. Editions O'Reilly. 2005."

The goal: "Object-ize" your code

The problem

Import the following java source Banking project BankApp.zip. This application has 3 main classes:

- `Account`: the bank accounts
- `BankAgency`: class, using Account to deal with Bank Agencies
- `BankAgencyApp`: the actual app with the (badly implemented) menus.

This last class (`BankAgencyApp`) has been written in a procedural approach, far from benefiting from

OO principles.

The goal

The purpose of this lab is to move from a functional / procedural approach to an object approach. You will rewrite and refactor the application code in several steps to obtain a code:

- easier to maintain,
- easier to extend,
- more robust in case of evolution.

Improvements and Modification

Without (for now) refactoring the code:

1. Study (and may be correct if some errors remain) the code of BankAgencyApp.
2. Understand the structures of the methods (menus especially)
3. Add the missing methods and sub-menus to obtain the following behavior:

```
--
  Tinkoff Bank (Kazan)
  General Menu
--
  Choose:
    1 - List of the Agency accounts
    2 - See an account (by its number)
    3 - Operation on an account
    4 - Accounts management

    0 - To quit this menu
Choice ?
3

--
  Tinkoff Bank (Kazan)
  Menu Operation on an account
--
  Choose:
    1 - Deposit money on an account
    2 - Withdraw money from an account

    0 - To quit this menu
Choice ?
0
End of Menu Operation on an account

--
```

```
  Tinkoff Bank (Kazan)
  General Menu
--
  Choose:
    1 - List of the Agency accounts
    2 - See an account (by its number)
    3 - Operation on an account
    4 - Accounts management

    0 - To quit this menu
Choice ?
4
--
  Tinkoff Bank (Kazan)
  Menu Accounts management
--
  Choose:
    1 - Add an account
    2 - Delete an account

    0 - To quit this menu
Choice ?
0

--
  Tinkoff Bank (Kazan)
  General Menu
--
  Choose:
    1 - List of the Agency accounts
    2 - See an account (by its number)
    3 - Operation on an account
    4 - Accounts management

    0 - To quit this menu
Choice ?
```

1. Do you have difficulties to add those sub-menus in what becomes now a "spaghetti" code ?
2. How would you describe the evolutivity of such a code ?
3. Didn’t you have the feeling of repeating yourself ?

"Object-ize" the functions

Principle

Think (even just 5 minutes) before jumping into the code: what could become objects in this program and what classes could be added ? (it is smarter to add a new class than to modify an existing class)

We will modify the code into several classes observing that:

1. Each user function could be programmed separately in the form of an object that we will call `Action` (menu option) having:
   a. the message displayed on the screen to "display" the action in a menu,
   b. a method to execute this menu option.
2. A menu could be programmed separately in the form of an object that we will call `ActionList` (list of menu actions) having:
   a. the message displayed on the screen to "display" the menu as a menu sub-menu,
   b. methods to add / remove menu options in this menu,
   c. a method to execute this menu (display and triggering of actions).

User function as objects

1. Make a copy of your current project and call it `BankAgencyApplication`.
2. Create the following packages:

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

3. Study the following code and use it accordingly in your `action` package:

```java
package action;
import bank.BankAgency;
/**
 * 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 BankAgency on which the action may act on
     * @throws Exception when an uncaught exception occurs during execution
    */
    public void execute(BankAgency ag) throws Exception;
}
```

4. Declare a class by action (menu option) to use. Start with "List of agency accounts":
    a. Create a class (the name `ActionAccountsLists` seems suitable) in the package `application.action`,
    b. which implements `Action`,
    c. with two attributes `(message, code)`
    d. write the code including a correctly configured constructor,
    e. the `execute(BankAgency)` method will display the screen of the list of bank agency accounts as a parameter.
5. In the same way, declare a class for the action "View an account (by its number)" (class `ActionSeeAccountNumber`) in the package `application.action`.

User menus as objects

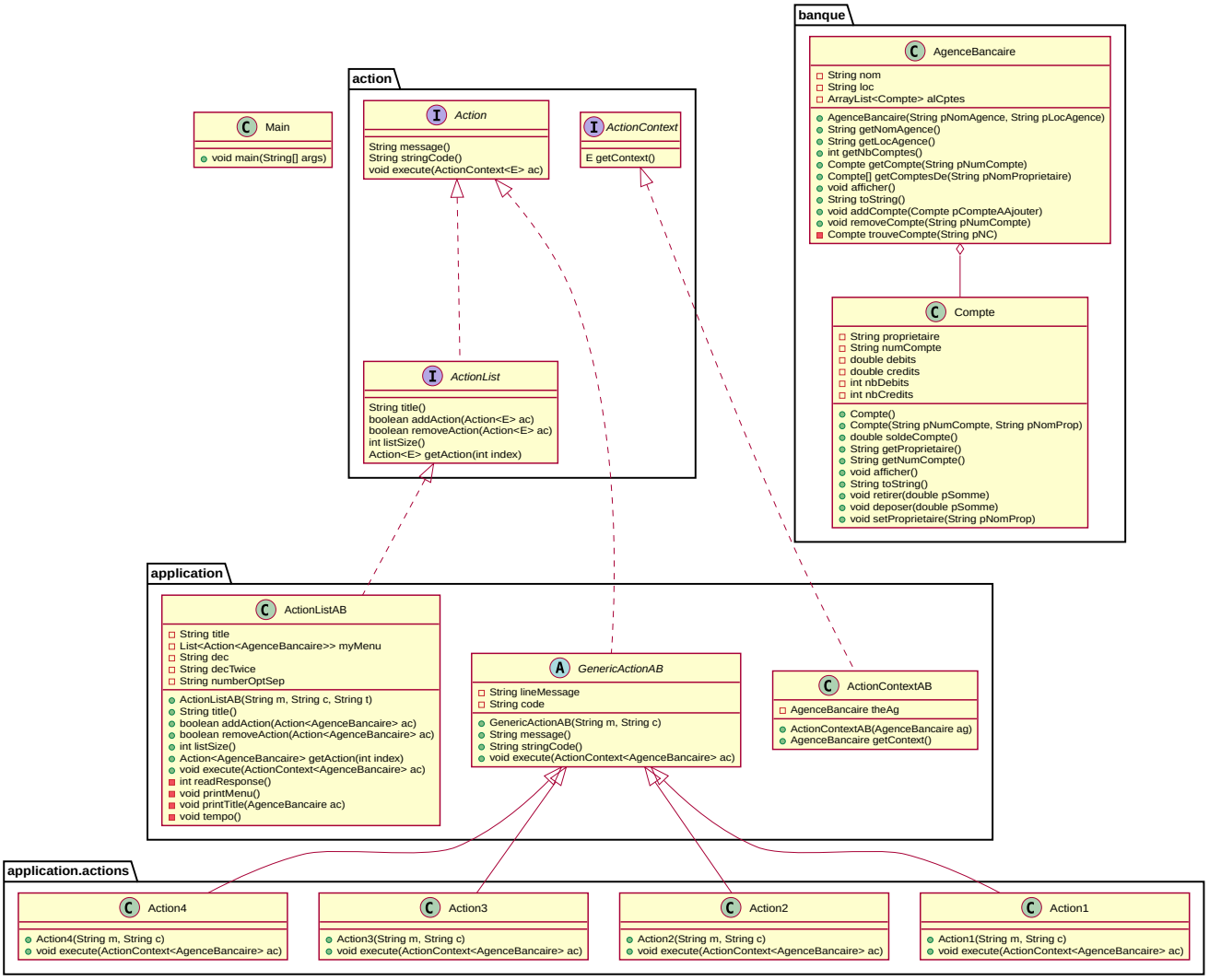
1. Study the following code and use it accordingly in your `action` package:

```java
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. Declare a class `GenericActionList` in the `application.actionlist` package,
    a. which implements `ActionList`,
    b. with 4 attributes `(message, code, title, actionlist)`. The action list being the different options that the action list (menu) will display.
    c. write the code including a correctly configured constructor,
    d. the `execute(BankAgency)` method will realize what'sin the documentation (comments). The menu will be the same as before. Each option will be numbered by `execute()` from 1 à n (nb of actions) + 0 to quit the menu.

You must obtain an architecture like this (be careful, slightly different):

Structure of the application (sorry for the French)



And now: go ! Let's check how easy the code is now to evolve

1. Create a class containing a main and allowing:
   a. to create an instance of each class `Action` created,
   b. to create an instance of `GenericActionList`,
   c. launch `execute()` on the instance of `GenericActionList`.

   Is it working ?

2. You can create the other actions and sub-menus.
3. Why `ActionList` inherits from `Action` ?

   We could have used another pattern called Composite ... later maybe

Abstraction of the problem

A new application ... too bad ...

Suppose we have to develop an application for managing a list of students (Student and StudentsGroup classes). It is based on a menu allowing to:

- See the list of students.
- Display a student by name.
- Edit a student's grades.
- Add a student to the group.
- Remove a student from the group.
- ...

Does that remind you if something ?

Questions:

1. Considering the new classes `Student` and `StudentsGroup`, can we reuse the `Action` and `ActionList` interfaces in the new application as is (without modifying them)?
2. If yes, why ?
3. If not why ?

Let us abstract a little bit the problem

Given the observations in the previous section, we would need classes `Action` and `ActionList` whose `execute()` would take any object as a parameter. Use Object? No, genericity is there to help us ...

1. Make a copy of the previous source code under the name `genericBankApp`.
2. Modify the declarations of the `Action` and `ActionList` interfaces as follows (be careful, all the code will become "wrong"):

```
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. Modify each class created (the `Action` then `ActionList` then the `main ()`) to either implement the correct instantiation of the interfaces, or correctly instantiate the objects.
4. Everything must work.
5. All you have to do is make the new application.

Let's go a little bit further: even more abstraction

A more complete `ActionList` interface

1. Make a copy of the previous project
2. For real applications, add the following operations to the `ActionList` interface:

```
    /**
     * 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() ;
}
```

And what about `ActionList` ?

`GenericActionList` which implements a menu (which implements `ActionList`) is here created specifically for BankAgency. But is this necessary in each application? (assuming nothing is displayed from the BankAgency). We should be able to make a "generic" class for managing menus composed of actions and reusable in each application.

So let's try:

1. Make a copy of the project from the previous section.
2. Move `GenericActionList` to the `action` package.
3. Rename this class to a name containing "ActionList" and well chosen. `AbstractActionList` would

be VERY poorly chosen.

4. To make this class generic (and not abstract), change its header.

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

5. Attention, all the code will now "warn" in red! normal ...
6. Modify whenever necessary to use generic type E
7. Remove all access to BankAgency (display name of bank, ...)
8. You should get to the end ...
9. Finally in the main there will be some "horns warnings" on the creation of objects of this new class because it will be necessary to indicate the type parameter at creation.

ATTENTION: making a generic class is not always that simple. Here the case has been simplified to the extreme.

3rd step: abstracting even more!

The problem :

1. Suppose we want to use our application in a different system where entries and displays are not done on the application's execution terminal ... The instructions using `new Scanner(System.in)` or `System.out.println ⋯` become obsolete.
2. Like the agency used in processing, these 2 elements are now part of the **execution context** of the actions.
3. Other elements could be used: transactions in progress (air reservations), databases, various connections, ...
4. It is therefore necessary to create an **execution context** which will be in parameter of the `Action` and `ActionList`.

Let's go!

1. Make a copy of the project from the previous section (without genericity).
2. In the `application` package, create a `ApplicationContextBankAgency` class implementing the `Singleton` pattern allowing access:
   a. At the "in progress" bank branch.
   b. To the `Scanner` to use. Initialize it here with a Scanner on System.in but something else could be used (a file, a stream to a terminal, ...).
   c. At the `PrintStream` output to use. Here it will be System.out but something else could be used (a file, a stream to a terminal, ...).
3. Refactor all the code:

a. The `Action` and `ActionList` classes now using the `ApplicationContextBankAgency` type (instead of` BankAgency`)

b. Modify access to the bank branch using `ApplicationContextBankAgency`.

c. Modify access to standard input using `ApplicationContextBankAgency`.

d. Modify access to standard output using `ApplicationContextBankAgency`.

4. It works ??

commit/push

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

Contributors

- Jean-Michel Bruel

About...