Object-Oriented Language and Theory

Lecturer: NGUYEN Thi Thu Trang, trangntt@soict.hust.edu.vn
Teaching Assistant: DO Minh Hieu, hieudominh@hotmail.com

Lab 06: Aggregation and Inheritance

* Objectives:

In this lab, you will practice with:

- JAVA Inheritance mechanism
- Use the Collections framework (specifically, **`ArrayList`**)
- Refactoring your JAVA code

In this exercise you extend the AIMS system that you created in the previous exercises to allow the ordering of books. You will create a **`Book`** class which stores the title, category, cost and an **`ArrayList`** of authors. Since the **`Book`** and **`DigitalVideoDisc`** classes share some common fields and methods, the classes are refactored, and a common superclass **`Media`** is created. The **`Order`** class is updated to accept any type of media object (either books or DVDs)

0. Branch your repository (read at Home)

Day after day, your repository becomes more and more sophisticated, which makes your codes harder to manage. Luckily, a Git workflow can help you tackle this. A Git workflow is a **recipe for how to use Git** to control source code in a consistent and productive manner. Release Flow[1] is a lightweight but effective Git workflow that helps teams cooperate with a large size and regardless of technical expertise.

Applying Release Flow is required from this lab forward.

However, we would use a modified version of Release Flow for simplicity.

- We can create as many branches as we need.
- We name branches with meaningful names. See Table 1-Branching policy.
- We had better **keep branches as close to `master` as possible**; otherwise, we could face merge hell.
- Generally, when we merge a branch with its origin, that branch has been history. We usually do not touch it a second time.
- **We must strictly follow the policy for release branch. Others are flexible.**

[1] https://docs.microsoft.com/en-us/azure/devops/repos/git/git-branching-guidance?view=azure-devops

Branch	Naming convention	Origin	Merge to	Purpose
feature *or* **topic**	+ feature/feature-name + feature/feature-area/feature-name + topic/description	master	master	Add **a** new feature or **a** topic
bugfix	bugfix/description	master	master	Fix **a** bug
		feature	feature	
hotfix	hotfix/description	master	master & releases [1]	Fix **a** bug in a **submitted assignment after deadline**
refactor	refactor/description	master	master	Refactor
		feature	feature	
release	release/labXX	master	none	Submit assignment [2]

Table 1-Branching policy

[1] If we want to update your solutions within a week after the deadline, we could make a new hotfix branch (e.g., `hotfix/stop-the-world`). Then we merge the `hotfix` branch with `master` and with `release` branch for last submitted assignment (e.g., `release/lab05`). In case we already create a `release` branch for the current week assignment (e.g., `release/lab06`), we could merge the `hotfix` branch with the current `release` branch **if need be,** or we can delete and then recreate current release branch.

[2] **Latest versions of projects in release branch serve as the submitted assignment.**

Let's use Release Flow as our Git workflow and apply it to refactor our repositories.

Step 1: Create new branch in our local repository. We create a new branch `refactor/apply-release-flow` from our `master` branch.

Step 2: Make our changes, test them, and push them. We move the latest versions of the two projects `AimsProject` and `OtherProjects` such that they are under the master branch directly. Note that the use case diagram of `AimsProject` should be kept inside a directory `design` of this project**. We can remove other directories if we want.**

See https://www.atlassian.com/git/tutorials/undoing-changes to undo changes in case of problems. To improve commit message, see https://thoughtbot.com/blog/5-useful-tips-for-a-better-commit-message.

Step 3: Make a pull request for reviews from our teammates[2]. We **skip** this step since we are **solo** in this repository. We, however, had better never omit this step when we work as a team.

Step 4: Merge branches. Merge the new branch `refactor/apply-release-flow` into `master` branch.

[2] https://www.atlassian.com/git/tutorials/making-a-pull-request

The result is shown in the following figure.

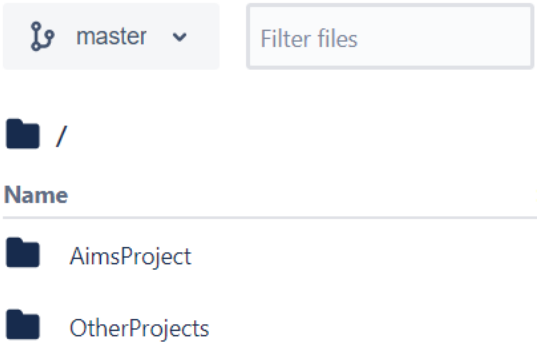


Figure 1-Result of the Demonstration

Hints:

Typical steps for a new branch:

- Create and switch to a new branch (e.g. abc) in the local repo: **git checkout -b abc**
- Make modification in the local repo
- Commit the change in the local repo: **git commit -m "What you had change"**
- Create a new branch (e.g. abc) in the remote repo (bitbucket through GUI)
- Push the local branch to the remote branch: **git push origin abc**
- Merge the remote branch (e.g. abc) to the master branch (bitbucket through GUI)

After completing all the tasks of that week, and merge all branches into master branch, you should create a release/labxx branch from the master in the remote repo (bitbucket).

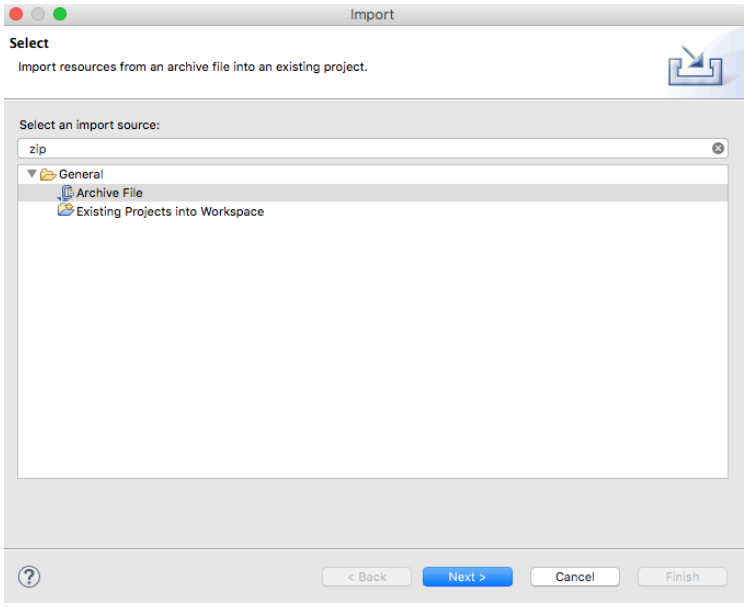
For example, in the lab06, from my view, there may be 2 main tasks => Create 2 branches:

- Create a branch **refactor/branch-organization** for refactoring the repository following the Release Flow
- Create a branch **topic/aggregation-inheritance** for the topic of this lab: Create Book, Media; modify DVD, Order and Aim classes:

1. Import the existing project into the workspace of Eclipse

- Open Eclipse

- Open File -> Import. Type zip to find Archive File if you have exported as a zip file before. You may choose Existing Projects into Workspace if you want to open an existing project in your computer. Ignore this step if the AimsProject is already opened in the workspace.



- Click Next and Browse to a zip file or a project to open

Once the project is imported, you should see the classes you created in the previous lab, namely, **`Aims`**, **`Order`**, **`DigitalVideoDisc`**.

- We can apply Release Flow here by creating a branch, e.g., **`topic/aims-project/add-media-class`**, testing our codes, pushing them, and then merging it with master.

2. Creating the `Book` class

- In the Package Explorer view, right-click the project and select New -> Class. Adhere to the following specifications:

- Package: **`hust.soict.globalict.aims.media`**
- Name: **`Book`**
- Access modifier: **`public`**
- Superclass: **`java.lang.Object`**
- **`public static void main(String[] args)`**: **do not check**
- Constructors from Superclass: **Check**
- All other boxes: **Do not check**

Add fields to the Book class

- To store the information about a **`Book`**, the class requires four fields: **`String`** fields **`title`** and **`category`**, a **`float`** field **`cost`** and an **`ArrayList`** of **`authors`**. You will want to make these fields private, with public accessor methods for all but the authors field

```
public class Book {

    private String title;
    private String category;
    private float cost;
    private List<String> authors = new ArrayList<String>();

    public Book() {
        // TODO Auto-generated constructor stub
    }
```

- Instead of typing the accessor methods for these fields, you may use the **Generate Getter and Setter** option in the **Outline** view pop-up menu (i.e., Right Click -> Source -> Generate Getters and Setters...)
- Next, create **`addAuthor`**(**`String authorName`**) and **`removeAuthor`**(**`String authorName`**) for the **`Book`** class
    - The **`addAuthor`**(...) method should ensure that the author is not already in the **`ArrayList`** before adding
    - The **`removeAuthor`**(...) method should ensure that the author is present in the **`ArrayList`** before removing
    - Reference to some useful methods of the **`ArrayList`** class

3. Creating the Media class

At this point, the **`DigitalVideoDisc`** and the **`Book`** classes have some fields in common namely title, category and cost. Here is a good opportunity to create a common superclass between the two, to eliminate the duplication of code. This process is known as refactoring. You will create a class called **`Media`** which contains these three fields and their associated get and set methods.

Create the Media class in the project

- In the **Package Explorer** view, right click to the project and select New -> Class. Adhere to the following specifications for the new class:

- Package: **`hust.soict.globalict.aims.media`**
- Name: **`Media`**
- Access Modifier: **`public`**
- Superclass: **`java.lang.Object`**
- Constructors from Superclass: Check
- **`public static void main (String[] args)`**: do not check
- All other boxes: Do not check

- Add fields to the **`Media`** class

- To store the information common to the **DigitalVideoDisc, CompactDisc** and the **Book** classes, the **Media** class requires three private fields: **String title**, **String category** and **float cost**
- You will want to make public accessor methods for these fields (by using **Generate Getter and Setter** option in the **Outline** view pop-up menu)

- Remove fields and methods which appear in Media class from **Book** and **DigitalVideoDisc, CompactDisc** classes

- Open the Book.java in the editor
- Locate the Outline view on the right-hand side
- Select the fields title, category, cost and their accessors & mutators (if exist)
- Right click the selection and select Delete from the pop-up menu
- Save your changes

- Do the same for the **DigitalVideoDisc, CompactDisc** classes by moving it to the package **hust.soict.globalict.aims.media**. Remove the package **hust.soict.globalict.aims.disc.**

- After doing that you will see a lot of errors because of the missing fields
- Extend the **Media** class for both **Book** and **DigitalVideoDisc**
    - **public class Book extends Media**
    - **public class DigitalVideoDisc extends Media**
- Save your changes.

For Hedspi, the package is **hust.soict.hedspi.aims.media**

4. Update the Order class to work with Media

You must now update the **Order** class to accept both **DigitalVideoDisc** and **Book**. Currently, the **Order** class has methods:

- **addDigitalVideoDisc**()
- **removeDigitalVideoDisc**().

You could add two more methods to add and remove **Book**, but since **DigitalVideoDisc** and **Book** are both subclasses of type **Media**, you can simply change **Order** to maintain a collection of **Media** objects. Thus, you can add either a **DigitalVideoDisc** or a **Book** using the same methods.

- Remove the **itemsOrdered** array, as well as its add and remove methods.
    a. From the **Package Explorer** view, expand the project
    b. Double-click **Order.java** to open it in the editor
    c. In the **Outline** view, select the **itemsOrdered** array and the methods **addDigitalVideoDisc**() and **removeDigitalVideoDisc**() and hit the **Delete** key
    d. Click **Yes** when prompted to confirm the deletion

- The **qtyOrdered** field is no longer needed since it was used to track the number of **DigitalVideoDiscs** in the **itemsOrdered** array, so remove it and its accessor and mutator (if exist).
- Add the **itemsOrdered** to the **Order** class
    e. Recreate the **itemsOrdered** field, this time as an object **ArrayList** instead of an array.
    f. To create this field, type the following code in the **Order** class, in place of the **itemsOrdered** array declaration that you deleted:
    ```
    private ArrayList<Media> itemsOrdered = new ArrayList<Media>();
    ```
- Note that you should import the **java.util.ArrayList** in the **Order** class
    g. A quicker way to achieve the same affect is to use the Organize Imports feature within Eclipse
    h. Right-click anywhere in the editor for the Order class and select Source -> Organize Imports (Or Ctrl+Shift+O). This will insert the appropriate import statements in your code.
    i. Save your class
- Create **addMedia**() and **removeMedia**() to replace **addDigitalVideoDisc**() and **removeDigitalVideoDisc**()
- Update the **totalCost**() method

5. Constructors of whole classes and parent classes

- Draw a UML class diagram[3] for the **AimsProject**. Attach the design image in the **design** directory. We can apply Release Flow here by creating a branch, e.g., **topic/class-diagram/aims-project/lab06,** push the diagram and its image, and then merge with master.
- Which classes are aggregates of other classes? Checking all constructors of whole classes if they initialize for their parts?
- Write constructors for parent and child classes. Remove redundant setter methods if any.

In Media class (superclass)

```
Media(String title){
      this.title = title;
}
Media(String title,
      String category){
      this(title);
      this.category = category;
}
```

[3] See Astah UML for class diagram at https://astah.net/support/astah-pro/user-guide/class-diagrams/

In Book class:

```
Book(String title){
	super(title);
}

Book(String title,
	String category){
	super(title, category);
}

Book(String title,
	String category,
	List<String> authors){
	super(title, category);
	this.authors = authors;
//TODO: check author condition
}
```

6. Create a complete console application in the `Aims` class

Open the Aims class. You will create a prompted menu as following:

```
public static void showMenu() {
		System.out.println("Order Management Application: ");
		System.out.println("--------------------------------");
		System.out.println("1. Create new order");
		System.out.println("2. Add item to the order");
		System.out.println("3. Delete item by id");
		System.out.println("4. Display the items list of order");
		System.out.println("0. Exit");
		System.out.println("--------------------------------");
		System.out.println("Please choose a number: 0-1-2-3-4");
}
```

You will modify classes **`Media`**, **`Order`** by adding more field **`id`**. Then create a complete application that allows to user to interact with through the above menu. Please use corresponding constructors to create objects.

- When adding an item in the order, the user can also choose between Book-CompactDisc-DigitalVideoDisc
- You must also think about methods to create each kind of Media
- You must think about methods to show the information of each kind of Media, for example: overriding the method toString() in each subcalss of Media to show its information

You can apply Release Flow by creating a feature branch for each option developed.