Rules

1. This exam is closed book, closed code, and closed Internet. You may not use the Internet, documents on your computer, or other source code on your computer.

2. You may use the Java API document if its installed on your own computer.

3. You may use an IDE, Java compiler, Java runtime, and an editor.

Instructions

1. Download the zip file for this exam and unpack it.

2. Create an IDE project using this source code. You may have to fix the build path in your IDE. The project uses Log4J (in the **`lib`** directory) and JUnit4.

3. Modify the project as per instructions.

What to Submit

Commit your solution to git on se.cpe.ku.ac.th. The URL of your repository is:

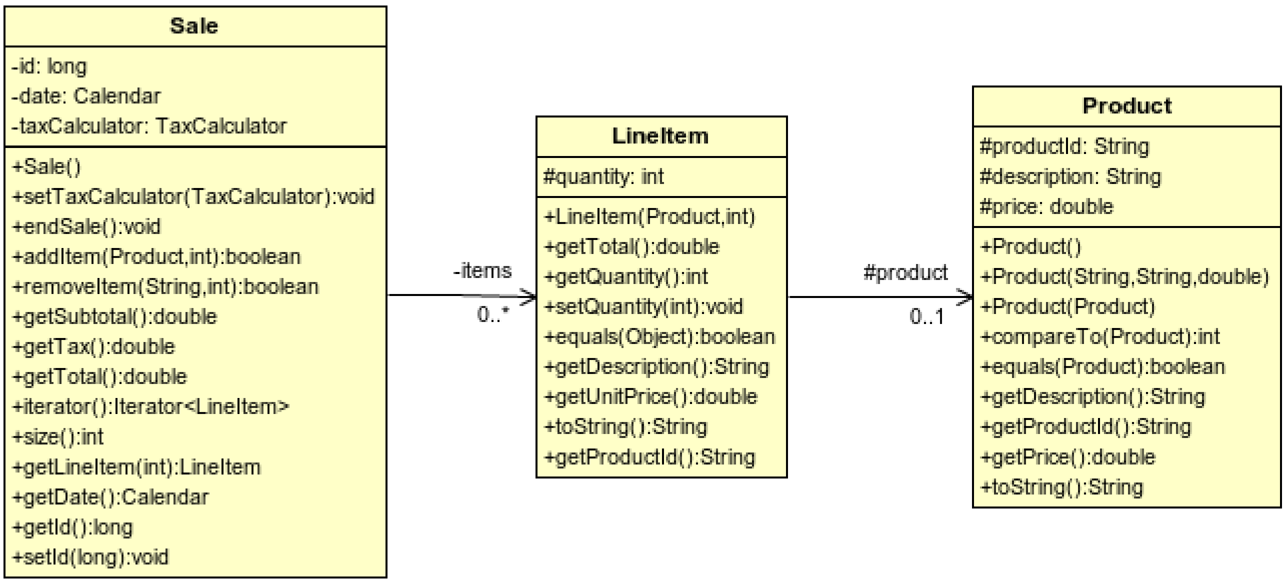
```
b5xxxxxxxxx@se.cpe.ku.ac.th:/home/git/b5xxxxxxxxx/labexam
```

See the last page for instructions how to create a Git project it and commit to remote repository.

Description: POS Application

The FirstGen POS code is a simple POS based on the model in Larman's textbook. For this exam, you will modify some classes in the packages firstgen.domain and firstget.store. Unlike the student ExceedVote projects, the FirstGen code contains extensive Javadoc comments to help you understand the code.

The classes in the package you need to work with are:



Problem 1. Singleton Store

1.1 Modify the **Store** class in package **firstgen.store** to make it a Singleton.

1.2 It should be impossible for other classes to create a store by using "new Store()".

1.3 Use *lazy instantiation*, meaning that you don't create the singleton instance until it is needed.

1.4 Update the **firstpos.Main** class to use the Singleton, instead of "new Store()".

Problem 2. "Buy N Get 1 Free" LineItem as Decorator

The **LineItem** class in package **firstgen.domain** represents buying a quantity of a Product, such as 5 bottles of Green Tea. A Sale consists of LineItems.

The **getTotal()** method of LineItem computes the total price of the line item using the simple formula:

```
 quantity * product.getPrice()
```

Sometimes the store has "Buy 2 Get 1 Free" or "Buy 1 Get 1 Free" sales. We don't want to add this logic to the LineItem class. So, we'll use the *Decorater Pattern*.

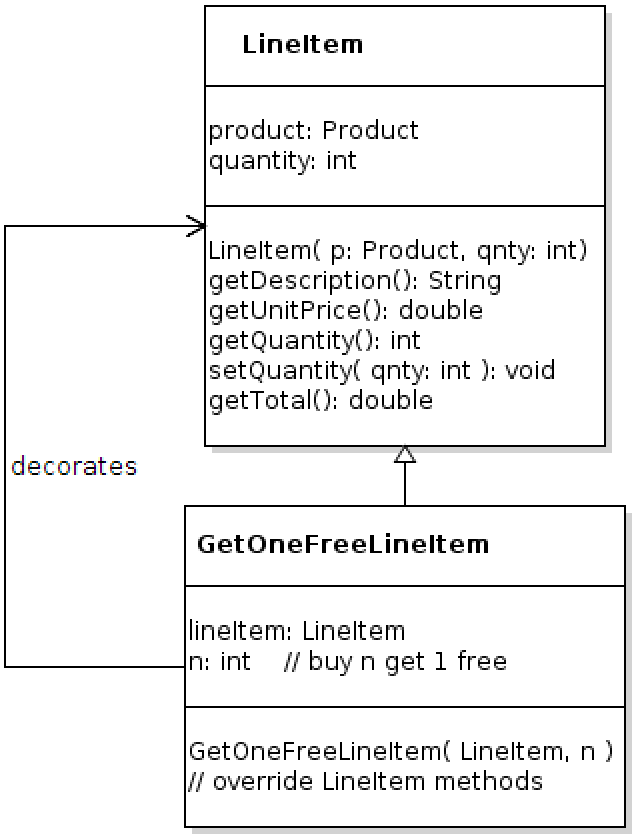
2.1 Write a **GetOneFreeLineItem** class that uses the *Decorator Pattern* to apply "Buy N Get 1 Free" pricing.

In this case, GetOneFreeLineItem overrides the getTotal method to return a special price: get 1 free for each n units purchased.

2.2 It also "decorates" the **getDescription()** method to append "Buy N Get 1 Free", for example:

Green Tea Buy 2 Get 1 Free

For other methods it just *delegates* them to the LineItem object it decorates..



2.3 The constructor for the decorator is:

GetOneFreeLineItem(LineItem item, int n)

item is the LineItem to apply special pricing to (object being decorated)

n is how many units customer must buy to get 1 free, e.g. n=2 for Buy **2** Get 1 Free

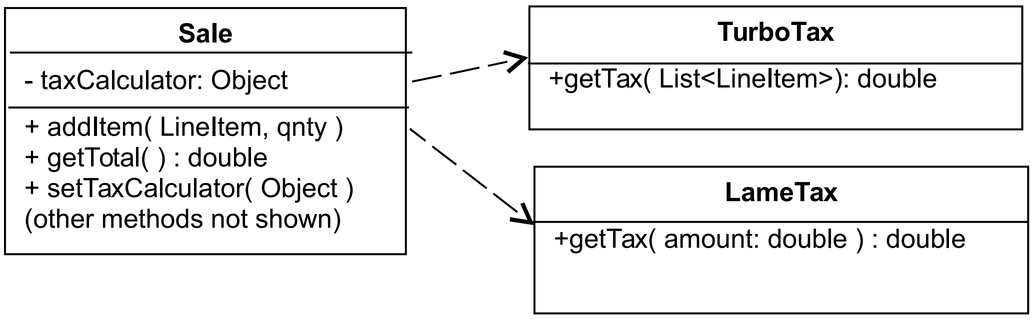
Example: Apply a "buy 2 get 1 free" discount

```
// product #10 is Green Tea (real green tea, not Oishi fake green tea)
Product tea = Store.getInstance().getProductCatelog().findById( 10 );
tea.getPrice();                            // returns 60.0
LineItem item = new LineItem( tea, 4 );    // buy 4 units of green tea
item.getDescription();                     // returns "Green Tea"
item.getTotal( );                          // returns 240.0 = 4 * 60.0
LineItem promo = GetOneFreeLineItem( item, 2 ); // buy 2 get 1 free!
promo.getDescription();                 // "Green Tea Buy 2 Get 1 Free"
promo.getQuantity();                       // returns 4
promo.getTotal( );                         // returns 180.0
```

Problem 3. Tax Calculator

The Sale class requires a tax calculator, which is set (injected) by the Register. Tax rules may vary, so we need to be able to change the kind of tax calculator. Right now the design is terrible. The Sale knows about 2 tax calculators:

- *TurboTax* has a getTax method that requires a list of LineItem as parameter. It examines each LineItem and returns the total tax for the list.
- *LameTax* requires only the total sale price and returns the tax. This works for simple VAT taxes.


Sale
- taxCalculator: Object
+ addItem(LineItem, qnty)
+ getTotal( ) : double
+ setTaxCalculator(Object)
(other methods not shown)
TurboTax
+getTax(List<LineItem>): double
LameTax
+getTax(amount: double) : double

Here is how the Sale computes tax. It tests the type of calculator object and then *casts* it, since each calculator has a different API.

```
public class Sale {                    // partial code for the Sale class
   private List<LineItem> items;
   private Object taxCalculator;

   /** set the tax calculator */
   public void setTaxCalculator( Object calculator ){
       this.taxCalculator = calculator;
   }
   /** compute tax on sale. */
   private double getTax( ) {
       double tax = 0;
       if ( taxCalculator instanceof TurboTax ) {
            TurboTax calc = (TurboTax) taxCalculator;
            tax = calc.getTax( items );
       }
       else if ( taxCalculator instanceof LameTax ) {
           LameTax calc = (LameTax) taxCalculator;
           // subtotal does not include tax
           tax = calc.getTax( this.getSubtotal() );
       }
       else logger.error("Unknown Tax Calculator type " + taxCalculator);
       return tax;
   }
```

3.1 Redesign this code: (a) eliminate coupling between Sale and the 2 tax calculator services, and (b) enable Sale to use to a different tax calculator service without modifying the Sale class.
Hint: Define an interface for all Tax Calculators. Use the package **`firstgen.tax`**.

You **can not modify the tax calculator classes in `extern.tax` -- the code is fixed by the company**!

3.2 Write one or more **adapters** for actual tax calculators so they implement your interface.
3.3 The Store knows its location, so the Store should create the tax calculator (*Information Expert*). The Register should ask the Store instead of trying to create its own tax calculator. Modify the code for Store and Register.

Initialize a Git Project for your lab exam

1. In your project directory create a new Git repository:

```
> cd yourworkspace/labexam
> git init
```

2. Create a a .gitignore file to ignore unwanted files. To create this file in Eclipse use: New → File.

```
> edit .gitignore
# add these lines to .gitignore
bin
*.class
.settings
.project
.classpath
```

3. Commit your source code to your local git repository.

```
> git add src .gitignore
> git commit -m "initial checkin"
```

4. Add a remote origin to your repository (use your KU login name). Type the whole command on one line.

```
> git remote add origin
              b5xxxxxxxx@se.cpe.ku.ac.th:/home/git/b5xxxxxxxx/labexam
> git push --set-upstream origin master
```

Password: *Your password is your KU Gmail login name in lowercase, such as* `fatalai.j`

Commit and Push Your Work (each time you do work)

Each time you modify or add files, you need to "commit" and "push" your changes.

`> git status`	check for new and modified files
`> git add src/file1 src/file2`	add new and modified files to "commit"
`> git commit -a -m "your message"`	commit to local repository. "-a" means "all" (optional).
`> git push`	push local repository to the remote repository.

You don't need to enter "--set-upstream origin master" each time.

Verify That You Correctly Submitted <u>All</u> Your Work

Verify you have submitted all work. Change to a totally different directory and try to "clone" your remote repository.

`> mkdir C:\temp`	a temporary directory for clone (any location you like)
`> cd C:\temp`	on Linux use "`cd /tmp`"

```
> git clone b5xxxxxxxx@se.cpe.ku.ac.th:/home/git/bxxxxxxxx/labexam
```

Now verify that the project you just cloned contains all the files you want to submit, and work is up to date.