Spring Exercises

Exercise 5.1 – Dependency Injection

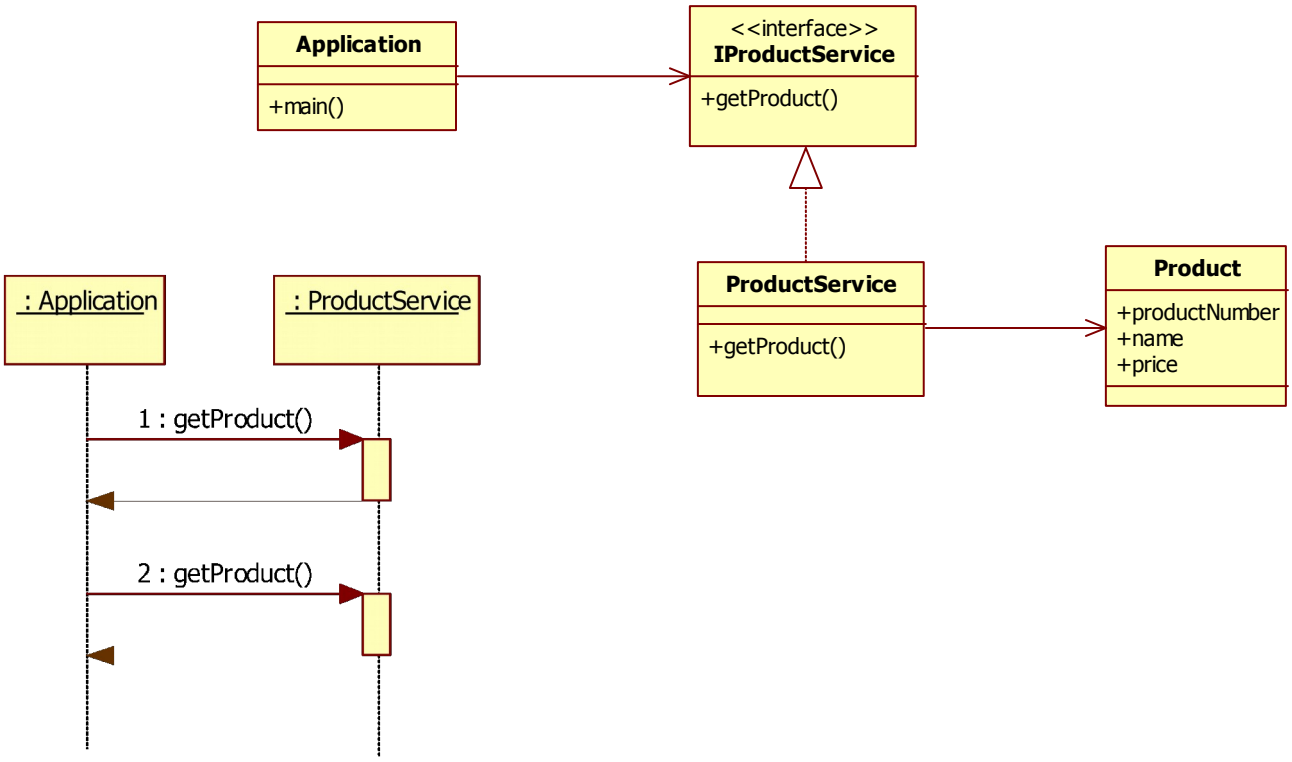
The Setup:

The main objective of this exercise is to work with the springconfig.xml file, configuring Spring beans and use dependency injection.

Start this project by opening **exercise5.1**. and updating its pom.xml file with the spring dependencies.

The Application:

The application provided is a very simple application that has a ProductService class, which implements the IProductService interface and contains a list of Products. Application.java calls the getProduct() method on ProductService.



You can run the file **App.java** by right-clicking on it and selecting **Run File** which should provide the following output:

```
productnumber=423  ,name=Plasma TV ,price=992.55
productnumber=239  ,name=DVD player ,price=315.0.
```

The Exercise:

a) Change the application in such way that **App.java** no longer instantiates **ProductService** but instead retrieves this object from the Spring context. In other words, change the following code:

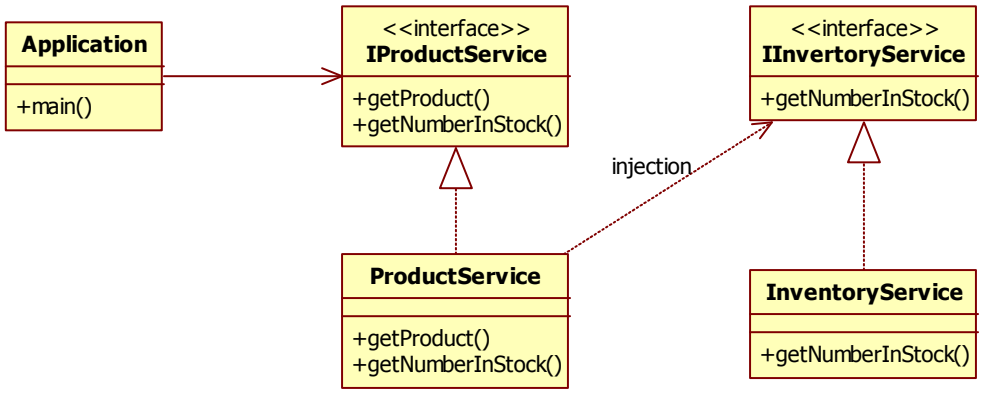
```
IProductService productService = new ProductService();
```

to the following lines of code:

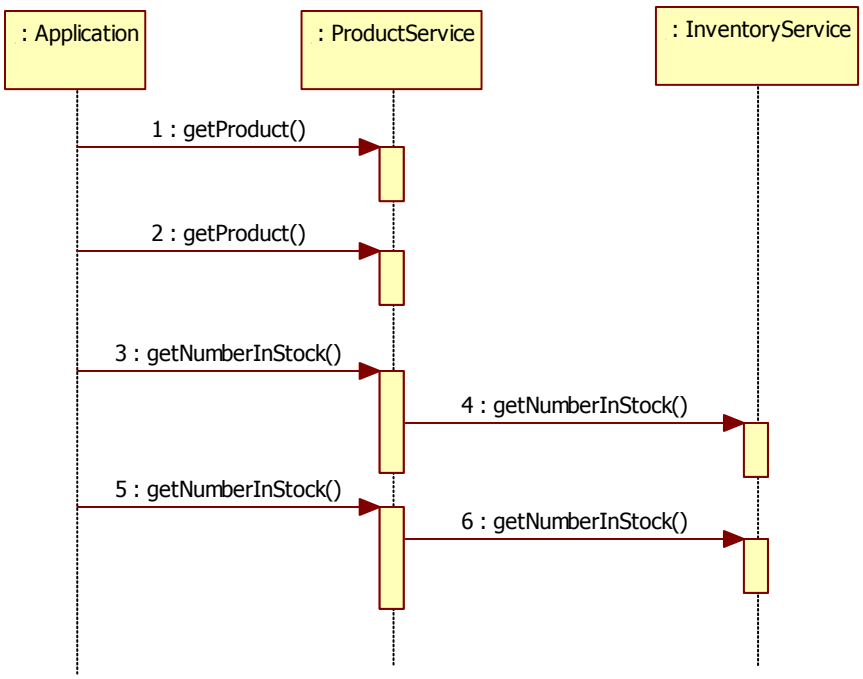
```
ApplicationContext context = new
   ClassPathXmlApplicationContext("springconfig.xml");
IProductService productService =
   context.getBean("productService", IProductService.class);
```

Of course, this will only work if we configure a bean with id=productService in **springconfig.xml**. Run the application and check that everything works correctly.

b) Now we want to add an **InventoryService** object, and inject this InventoryService into the ProductService using Spring dependency injection.



We will also want to add a **getNumberInStock()** method to the ProductService that calls the getNumberInStock() method of InventoryService.



First, create a new interface called IInventoryService and then a class called InventoryService using the code below:

```
public interface IInventoryService {
    public int getNumberInStock(int productNumber);
}
```

```
public class InventoryService implements IInventoryService{

    public int getNumberInStock(int productNumber) {
            return productNumber-200;
    }

}
```

Then, update IproductService and ProductService as shown in the code below.

```
public interface IProductService {
   public Product getProduct(int productNumber);
   public int getNumberInStock(int productNumber);
}
```

```
public class ProductService implements IProductService{
 private IInventoryService inventoryService;
    private Collection productList= new ArrayList();

    public ProductService(){
            productList.add(new Product(234,"LCD TV", 895.50));
            productList.add(new Product(239,"DVD player", 315.00));
            productList.add(new Product(423,"Plasma TV", 992.55));
    }
    public Product getProduct(int productNumber) {
            for (Product product : productList) {
                    if (product.getProductNumber() == productNumber)
                            return product;
            }
            return null;
    }
    public int getNumberInStock(int productNumber) {
            return inventoryService.getNumberInStock(productNumber);
    }
    public void setInventoryService(IInventoryService inventoryService) {
            this.inventoryService = inventoryService;
    }

}
```

Make sure that the inventoryService object gets properly injected into productService by adding the needed XML configuration to **springconfig.xml**.

To test if this works, add the following two lines to **Application.java**:

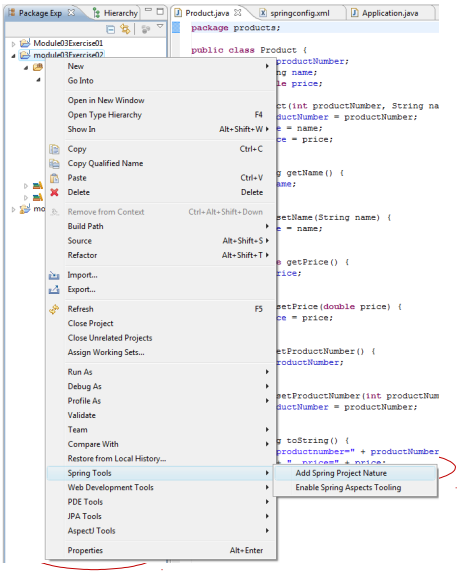
```
System.out.println("we have " + productService.getNumberInStock(423)
            + " product(s) with productNumber 423 in stock");
System.out.println("we have " + productService.getNumberInStock(239)
            + " product(s) with productNumber 239 in stock");
```

Then, run **Application.java** to make sure it gives the following output:

```
productnumber=423  ,name=Plasma TV ,price=992.55
productnumber=239  ,name=DVD player ,price=315.0
we have 223 product(s) with productNumber 423 in stock
we have 39 product(s) with productNumber 239 in stock
```

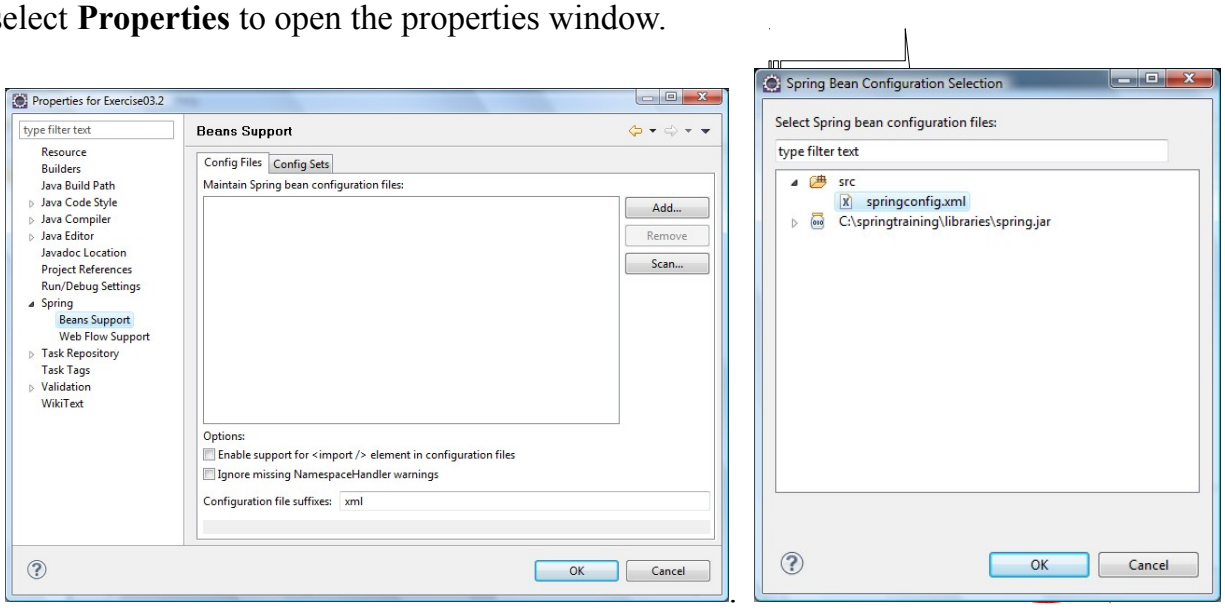
Spring Tool Suite (STS) Optional Exercise:

If you open your maven project in STS you can use it to show a graphical view of the configured Spring beans. Start by enabling and configuring the **Spring Project Nature** on your project:



Right-click the project **Exercise5.1** in the package explorer and select **Spring Tools / Add Spring Project Nature**.

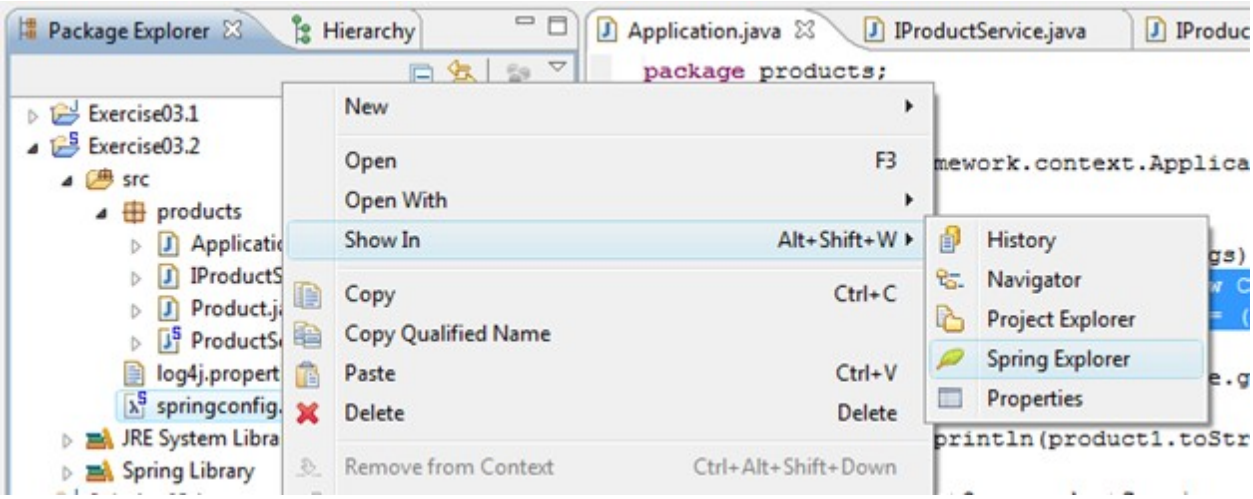
Once you have added the Spring Project Nature, right-click on the project again, and select **Properties** to open the properties window.



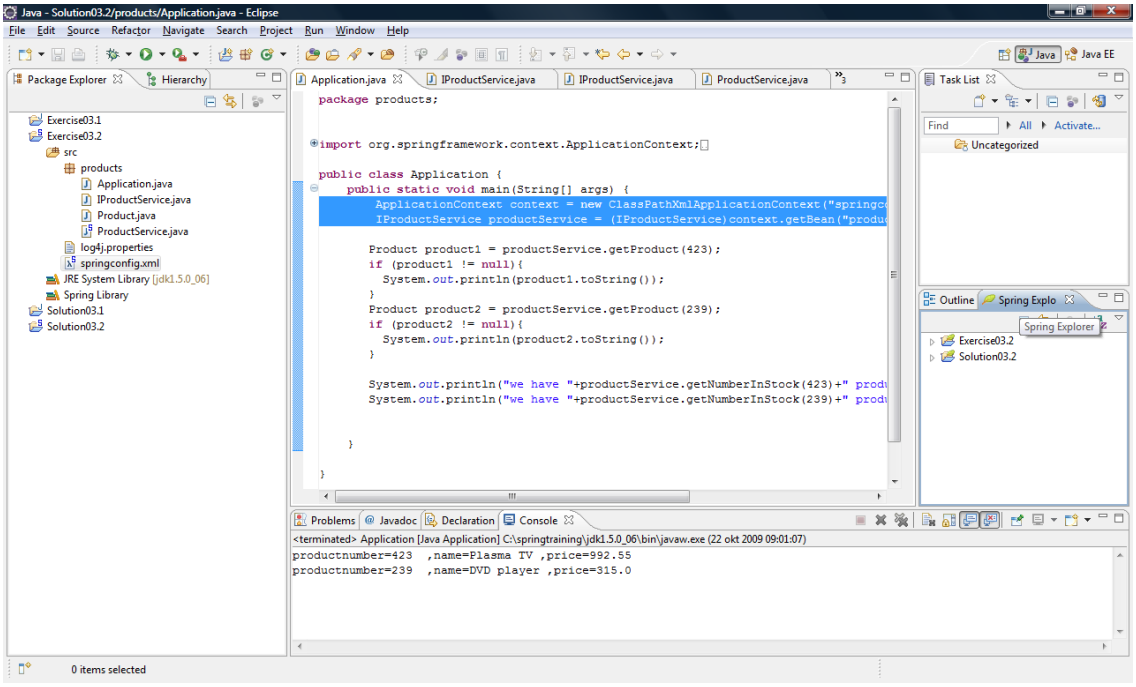
Select the **Spring / Beans Support** section on the left hand side to open the Beans Support page and click on the **Add** to open the Spring Bean Configuration Selection window.

Select your **springconfig.xml** file on the **Spring Bean Configuration Selection** window, (usually found under the **src** folder) and select **OK**. Once your springconfig.xml has been added, you can close the Properties window by clicking **OK** there as well.

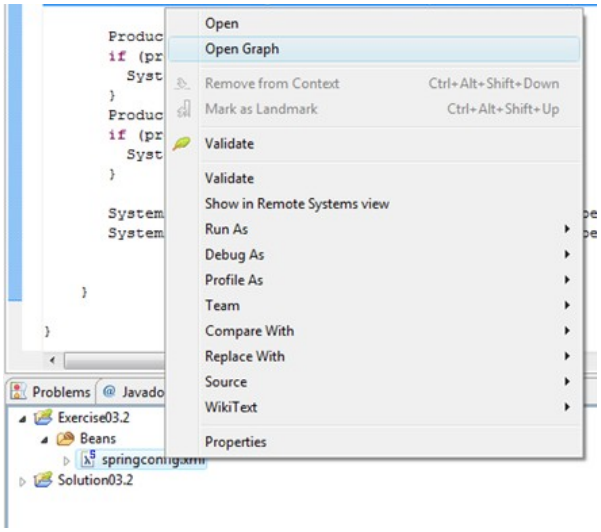
Eclipse can now generate a visual graph of your mapped beans.

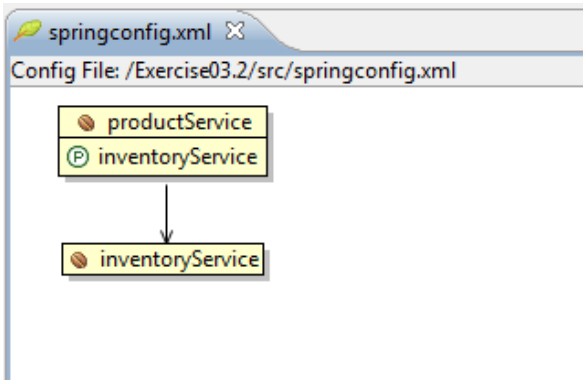


Right-click on your **springconfig.xml** file and select **Show In Spring Explorer**.



The Spring Explorer tab sometimes opens on the right-hand side near the outline section. Feel free to drag the tab to another location, for example next to the console tab.





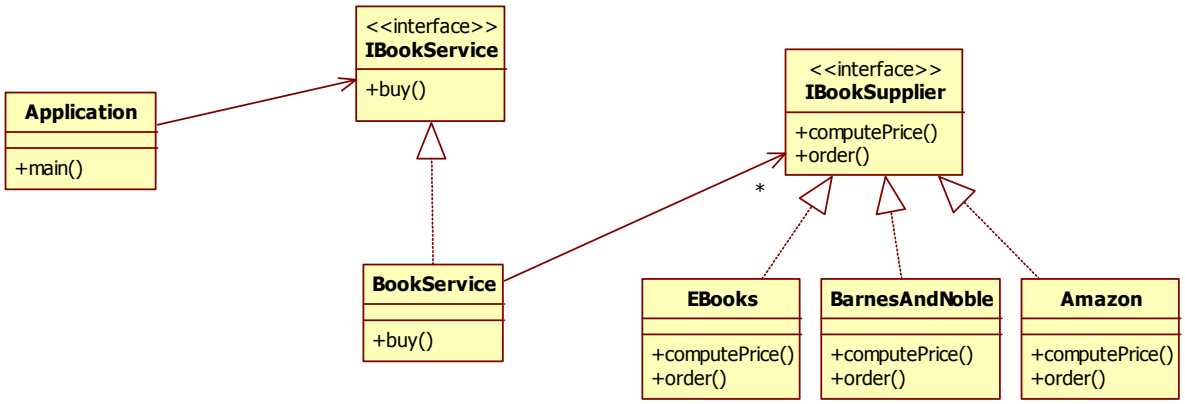
Then, in the Spring Explorer right-click on your springconfig.xml file and select **Open Graph**. The graph shows that the inventoryService bean is injected into the productService bean.

Exercise 5.2 – Dependency Injection using Lists

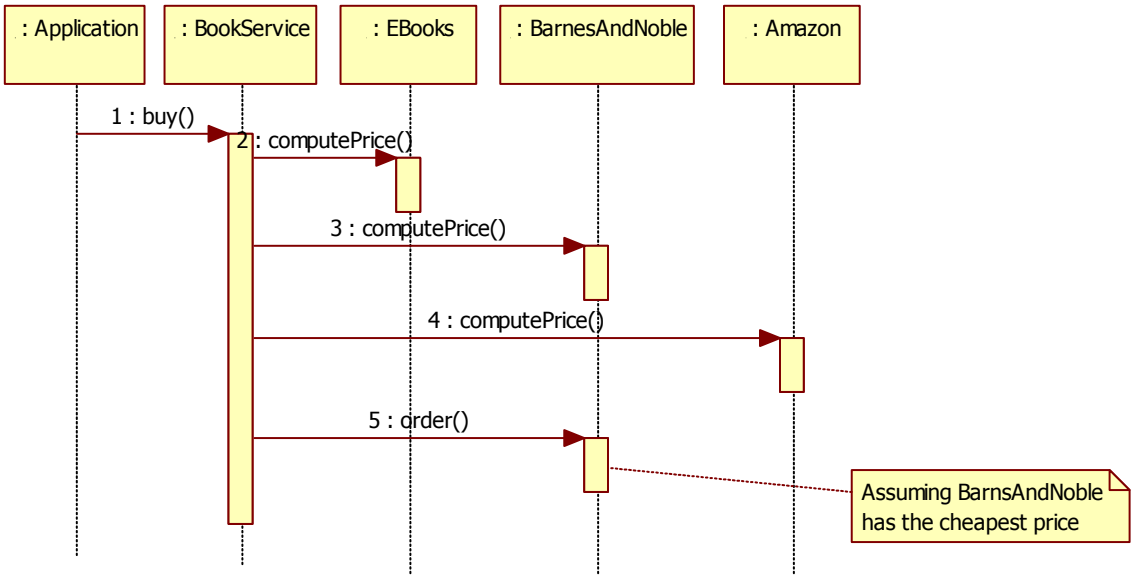
The Setup:

The purpose of this exercise is for you to use the more advanced list configuration feature of dependency injection. Start by opening the project at **exercise5_2** and add the spring dependencies to the pom.xml file.

The Application:



Looking through the code, you will see that the application buys 3 books through the IBookService implemented by BookService. In the buy method, the BookService checks each of its IBookSuppliers, finding the cheapest one and ordering the book from there.



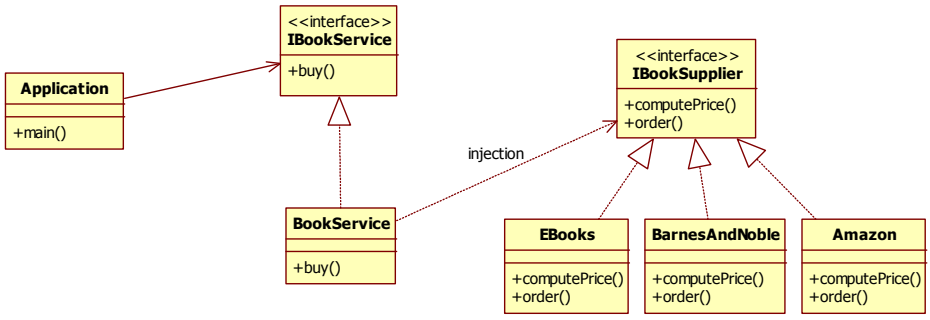
Running Application.java should produce the following (randomized) output:

```
Amazon charges $30.693970696674995 for book with isbn 123433267
Barnes&Noble charges $6.554986457356226 for book with isbn 123433267
EBooks charges $41.282876907999295 for book with isbn 123433267
Book with isbn = 123433267 is ordered from Barnes&Noble
Amazon charges $42.430314310592614 for book with isbn 888832678
Barnes&Noble charges $21.83991052230513 for book with isbn 888832678
EBooks charges $15.977769931887757 for book with isbn 888832678
Book with isbn = 888832678 is ordered from EBooks
Amazon charges $35.20968199800302 for book with isbn 999923156
Barnes&Noble charges $0.5630258200828281 for book with isbn 999923156
EBooks charges $10.151810464638245 for book with isbn 999923156
Book with isbn = 999923156 is ordered from Barnes&Noble
```

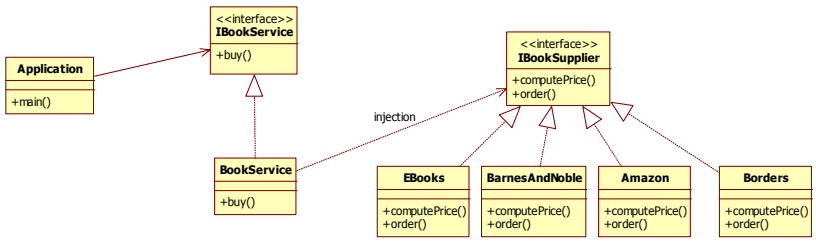
The downside of this application is that the BookService is hardcoded with Amazon, EBooks and Barnes & Noble as IBookSuppliers. If we should want to add another book supplier, we would have to go in and change the code.

The Exercise:

a) Change the application in such a way that the BookSuppliers are injected into the BookService using dependency injection, rather than being instantiated with ***new*** as they are now, and have the application retrieve the BookService from the Spring context.



b) Then add the **Borders** BookSupplier to the list without changing any of the existing code.



Assignment 5-3: Bank Application Dependency Injection

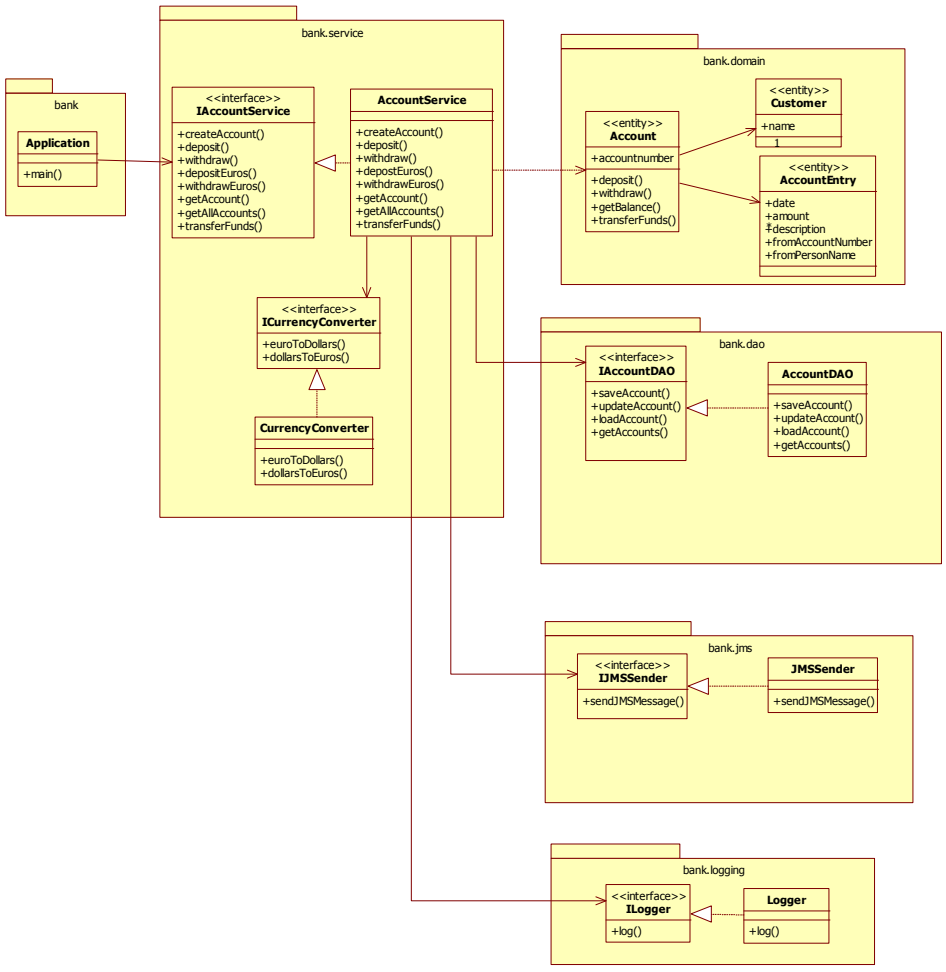
The Setup:

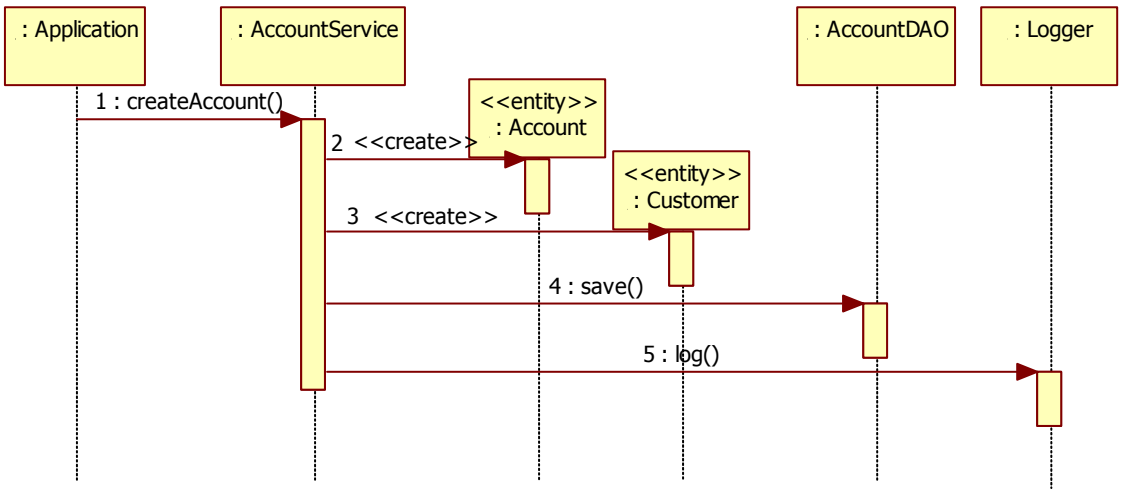
This exercise introduces the bank application. The bank application is a small application that embodies most of the architectural needs of a more real world enterprise application. Although the application that we start with in this exercise does not use any of Spring's features (yet), many areas in this application could benefit from them.

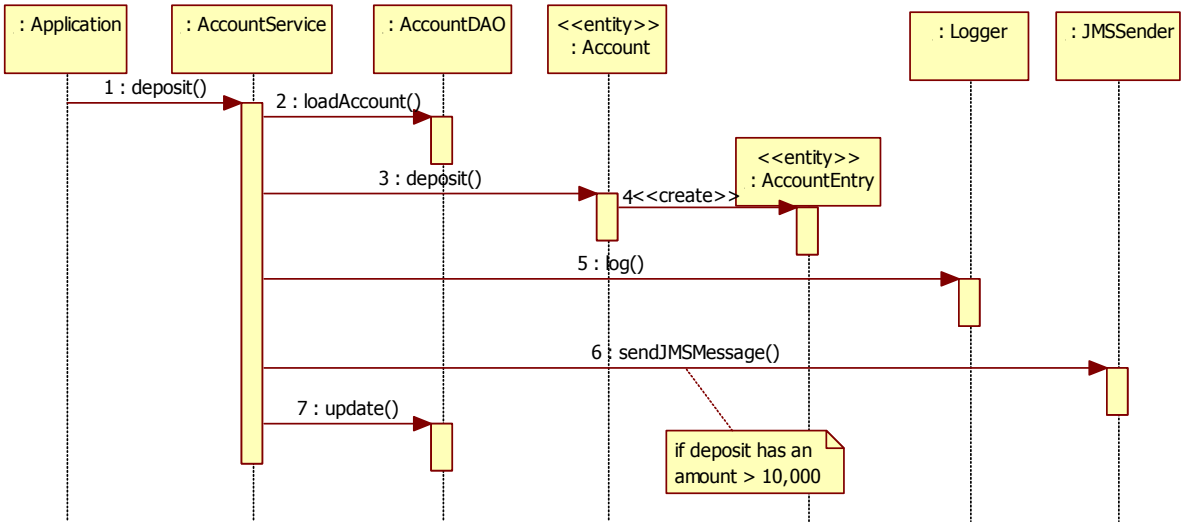
In this exercise, we will start by adding dependency injection to the application. In subsequent exercises we will continue to build on this, adding new features as they are covered.

Start by opening the project at **Exercise5.3** and add the **Spring dependencies** to it.

The Application:







The bank application uses an AccountService object to perform the various bank-related services such as creating accounts, depositing money (Euros or Dollars), withdrawing money (Euros or Dollars), and transferring funds between accounts.

The AccountService object manipulates the domain objects Account, Customer and AccountEntry through the methods mentioned above. An Account object and a Customer object are created when CreateAccount() is called, and an AccountEntry is created for every deposit or withdrawal. All changes are then saved to the database through the AccountDAO.

Since the Accounts only work internally with dollar amounts, all euro deposits and withdrawals are first converted to dollars using a CurrencyConverter object.

All AccountService methods are logged through the Logger object, and whenever an amount greater than 10,000 dollars is deposited or transferred into an account, an additional JMS message is sent to the taxation services department.

Running **App.java** in the **cs544.exercise5_3.bank** package should output:

```
Jun 12, 2009 11:45:24 PM bank.logging.Logger log
INFO: createAccount with parameters accountNumber= 1263862 , customerName= Frank Brown
Jun 12, 2009 11:45:24 PM bank.logging.Logger log
INFO: createAccount with parameters accountNumber= 4253892 , customerName= John Doe
Jun 12, 2009 11:45:24 PM bank.logging.Logger log
INFO: deposit with parameters accountNumber= 1263862 , amount= 240.0
Jun 12, 2009 11:45:24 PM bank.logging.Logger log
INFO: deposit with parameters accountNumber= 1263862 , amount= 529.0
CurrencyConverter: converting 230.0 dollars to euros
Jun 12, 2009 11:45:24 PM bank.logging.Logger log
INFO: withdrawEuros with parameters accountNumber= 1263862 , amount= 230.0
Jun 12, 2009 11:45:24 PM bank.logging.Logger log
INFO: deposit with parameters accountNumber= 4253892 , amount= 12450.0
JMSSender: sending JMS message =Deposit of $ 12450.0 to account with accountNumber=
4253892
CurrencyConverter: converting 200.0 dollars to euros
Jun 12, 2009 11:45:24 PM bank.logging.Logger log
INFO: depositEuros with parameters accountNumber= 4253892 , amount= 200.0
Jun 12, 2009 11:45:24 PM bank.logging.Logger log
INFO: transferFunds with parameters fromAccountNumber= 4253892 , toAccountNumber=
1263862 , amount= 100.0 , description= payment of invoice 10232
Statement For Account: 4253892
Account Holder: John Doe
-Date--------------------------Description-------------------Amount-------------
  Fri Jun 12 23:45:24 GMT 2009                       deposit            12450.00
  Fri Jun 12 23:45:24 GMT 2009                       deposit              314.00
  Fri Jun 12 23:45:24 GMT 2009     payment of invoice 10232              -100.00
--------------------------------------------------------------------------------
                                       Current Balance:            12664.00

Statement For Account: 1263862
Account Holder: Frank Brown
-Date--------------------------Description-------------------Amount-------------
  Fri Jun 12 23:45:24 GMT 2009                       deposit              240.00
  Fri Jun 12 23:45:24 GMT 2009                       deposit              529.00
  Fri Jun 12 23:45:24 GMT 2009                      withdraw             -361.10
  Fri Jun 12 23:45:24 GMT 2009     payment of invoice 10232              100.00
--------------------------------------------------------------------------------
                                       Current Balance:              507.90
```

The Exercise:

Change the bank application in such a way that the Logger, CurrencyConverter, AccountDAO and JMSSender are injected into the AccountService, rather than being instantiated with ***new***. In other word, AccountService should no longer contain these lines:

```
accountDAO = new AccountDAO();
currencyConverter = new CurrencyConverter();
jmsSender =  new JMSSender();
logger = new Logger();
```

Also update **App.java** so that it retrieves the AccountService from the Spring context.