Inventory Tracker Phase 2

Model-View-Controller, Barcode Label Printing

Objective

In this phase you will integrate the Inventory Tracker UI with the core object model and persistence subsystems developed in Phase 1. You will also implement barcode label printing. This exercise will give you hands-on experience with the following concepts:

Layering – The UI and core model constitute separate layers in your design. Of course, the UI layer is built on top of the core model layer.

User Interface/Model Separation – A well-designed program maintains a strict separation between the core model and the user interface layers. Such a separation simplifies the design, and makes the core model independent of a particular UI design so that it can be used to implement other UIs or applications in the same domain.

Model-View-Controller – The provided UI code is based on the Model-View-Controller design pattern. All of the View classes have been provided, but you will need to implement all of the Controller classes to integrate the UI with your Model.

The Functional Specification describes the functionality of the Inventory Tracker program. All user-observable functionality must be implemented in this phase, except for the following features, which will be added in later phases. (In fact, I prefer that you not design them now, so you can do a better job of it later when you have learned more design principles and patterns.):

- Reports
- Undo/Redo
- Product Auto-Identification

Design

Much of this phase is an integration exercise, but there is also additional design work to be done. Specifically, your design should cover the following areas:

1. MVC
2. Barcode Label Printing

MVC

The Inventory Tracker UI is based on the Model-View-Controller design pattern. Your core object model will serve as the Model. The provided UI code contains all of the View and Controller classes. The View classes are complete, and you need not change them. The Controller classes, however, are merely stubs, and you will need to implement them. The Controller classes will integrate your Model classes with my View classes.

The state of the UI and the state of the core object model must be kept in synch at all times. For example:

1) The Name of the currently-selected Storage Unit in the Storage Unit / Product Group Tree is displayed in three places at the same time: in the Storage Unit / Product Group Tree, in the Context Panel, and in the Item Table. When the Name of the currently-selected Storage Unit is modified, it must be immediately updated in all places that it is displayed. Similarly, when the Name of the currently-selected Product Group is changed, it must also be immediately updated in all places it appears.

2) When the Name of a Storage Unit or Product Group is modified, the modified node and its sibling nodes in the Storage Unit / Product Group Tree must be re-ordered to keep the tree properly sorted.

3) When a Product's Description is modified, the Product Table must be updated and re-sorted to reflect the change. Similarly, when an Item's Entry Date is modified, the Item Table must be updated and re-sorted to reflect the change.

4) In the Add Item Batch View, as Items are added, the count fields in the Product Table must be immediately updated in both the Add Item Batch View and in the Inventory View. These counts must also be updated when Undo/Redo operations are performed. (The Remove Item Batch View and Transfer Item Batch View behave similarly.)

5) The Item Table in the Add Item Batch View must be updated when Items are added, and when Undo/Redo operations are performed. (The Remove Item Batch View and Transfer Item Batch View behave similarly.)

In these and other cases, any time the state of the core model is changed, all parts of the UI affected by the changes must be immediately updated. For this design area, explain specifically how you will keep the state of the UI in synch with the state of the core model. Explain in detail how your design will handle the scenarios listed above. Provide enough detail to convince me that you have thought this through and know exactly how you will solve it.

- <u>For this area you should provide:</u>
  1) <u>UML class and sequence diagrams that show how you will keep the UI and core model in synch</u>
  2) <u>Updated Javadoc documentation for the classes involved in your MVC design</u>

NOTE: If you implement MVC properly in this phase, implementing Undo/Redo will be much easier in Phase 3.

Barcode Label Printing

Design one or more classes in your core object model that will implement barcode label printing.

- Provide Javadoc documentation for the classes that will implement barcode label printing.

When the user finishes adding a batch of Items, your program should generate and display a PDF file containing the Item barcode labels, which the user can then print. The iText Java library is included in the Inventory Tracker source tree for generating PDF files. Safari Books Online contains a good book that shows how to use iText to create PDF files, and other examples are available on the web. Safari Books Online is available through the HBLL web site at the following URL:

http://proquest.safaribooksonline.com.erl.lib.byu.edu/book/programming/9781935182610

After generating a PDF file, it can be displayed to the user as follows:

```
java.awt.Desktop.getDesktop().open(new File(filename));
```

Design Division of Labor

Everyone should help document the MVC portion of the design. The biggest part will be creating the UML class and sequence diagrams, but the Javadocs will also need to be updated. Someone will also need to document your Barcode Label Printing design, but that part is relatively small.

Implementation

Implement your MVC and Barcode Label Printing designs. This will require you to implement all user-observable Inventory Tracker functionality, except Reports, Undo/Redo, and Product Auto-Identification. This includes making sure that all UI controls are enabled or disabled appropriately at all times according to the behavior described in the Functional Specification.

Make sure that your program compiles and runs successfully when invoked with the `ant run` command (which is how the TA will run your program). Make sure that all of your test cases compile and run successfully when invoked with the `ant test` command (which is how the TA will run your test cases).

Use Subversion to share files with your teammates. Check in your code early and often, but don't break the build. Notify your teammates when you commit new code.

Remove errors reported by Checkstyle. For each phase, you have 5 free Checkstyle errors that you can ignore without penalty. For Phase 2 you can have a total of 10 unresolved errors. This is for cases where you disagree with Checkstyle, and don't want

to follow its advice. Checkstyle will not be graded until after phase pass off. You have until then to remove Checkstyle errors.

Implementation Division of Labor

The implementation work should be divided up as follows:

MVC – 2 or 3 people
Barcode Label Printing – 1 person

Deliverables

- ☐ Group Design
  - UML diagrams (submit PDF file online)
  - Javadocs (submit zip file online)
- ☐ Group Implementation (send your zip file to your testing team and the TA)
  - All Inventory Tracker functionality implemented (except Reports, Undo/Redo, and Product Auto-Identification)
  - All files checked into Subversion
  - Everything should build and run successfully, including your JUnit tests
  - Remove Checkstyle errors
  - Create a zip file containing your `inventory-tracker.jar` file, the `lib/` directory, the `images/` directory, and any other directories or files needed by your program. It should be possible for a classmate or TA to unzip your zip file and run your program with the following command:
    `java –jar inventory-tracker.jar`
    - Use the ant script to generate your `inventory-tracker.jar` file rather than creating it by hand. This will ensure that it is created correctly.
    - Include the `lib/` directory (and everything in it) in your zip file because `inventory-tracker.jar` depends on the libraries in that directory
    - Include the `images/` directory (and everything in it) in your zip file because `inventory-tracker.jar` depends on the image files in that directory
    - Test your zip file on both Windows and Linux to ensure that it will work for others when they try to use it
  - **Send your implementation zip file to your testing team and the TA**
- ☐ Individual Phase Report / Exam (submit PDF file online)