Inventory Tracker Phase 1

Core Object Model & Simple Persistence

Objective

In this phase you will design and implement a core object model for Inventory Tracker that uses Java serialization for persistence.

Design

Your design should address the following areas:

1. Core Object Model
2. Simple Persistence

Core Object Model

Design a core object model for Inventory Tracker. The core object model should include classes that model the application's core concepts, including data structures and algorithms. It should include classes for representing Storage Units, Products, Items, and all other relevant domain concepts and relationships.

Classes in the core object model should be inherent to the application domain. Inherent domain classes are independent of the particular user interface employed. That is, if Inventory Tracker's user interface were to be substantially redesigned, the classes in the object model should be reusable in implementing the new interface. The classes in the object model will be accessed by virtually all parts of the program, because the information they represent is central to everything the program does.

For this initial phase your design need not support the following features, which will be added in future 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.):

- Barcode label printing
- Reports
- Undo/Redo
- Product Auto-Identification

- For this area you should create:
  1) UML class diagrams showing the classes in your core object model and the relationships between them
  2) Javadoc documentation for the classes in your core object model

The following sections describe specific things you should pay attention to while designing your core object model.

Abstraction

Create high-quality classes that effectively model all domain and implementation concepts. Ensure that your model classes support all of the necessary operations required by the program. To ensure this, thoroughly analyze the Functional Specification and Inventory Tracker GUI, and make a list of the data and operations needed to support the program's functionality.

For example, the user can print a Product Statistics report. This means that your object model should keep track of all data needed to generate these statistics. In addition to simple operations such as getters and setters, your core model should also provide the more complicated operations needed by the application, such as:

1. Adding Items
2. Moving and Transferring Items
3. Removing Items
4. Putting Products in a Product Container
5. Deleting Products

Design Principles

When designing your object model, apply the following important design principles:

- Good naming
- Cohesive classes and methods
- Abstract all the way (i.e., avoid primitive obsession)
- Decompose large or complicated classes and methods
- Good data structure and algorithm design
- Information hiding (i.e., hide internal implementation details)
- Avoid code duplication

Design Patterns

Find useful ways to apply the Singleton and Composite design patterns.

Effective Data Encapsulation

Encapsulate your data structures within classes that hide their internal implementation details, and provide convenient method interfaces for performing all necessary operations on the data. Typically, operations are needed for inserting, deleting, searching, iterating, and performing other domain-specific algorithms on the encapsulated data. Rather than giving clients direct, unfettered access to the underlying data structure, instead provide operations that directly support what clients need to do. This will result in better abstraction and fewer dependencies.

Data Integrity Enforcement

Your core model classes should prevent clients from modifying data in illegal ways. In other words, your classes must maintain the integrity of the data by ensuring that it is always correct. Specifically, your core model should enforce the following integrity constraints:

1. All integrity constraints listed in the Data Dictionary section of the Functional Specification
2. All additional integrity constraints defined in the Functional Specification. For your convenience, here is a list of them, including the name of the section in the Functional Specification where they appear:
    a. Adding Items
        i. Section: Data Dictionary

            When a new Item is added to the system, it is placed in a particular Storage Unit (called the "target Storage Unit"). The new Item is added to the same Product Container that contains the Item's Product within the target Storage Unit. If the Item's Product is not already in a Product Container within the target Storage Unit, the Product is placed in the Storage Unit at the root level.
        ii. Section: Adding Items

            New Items are added to the Product Container within the target Storage Unit that contains the Item's Product. If the Item's Product is not already in the Storage Unit, it is automatically added to the Storage Unit at the top level before the Items are added.
    b. Moving / Transferring Items
        i. Section: Data Dictionary

            An Item is contained in exactly one Product Container at a time (until it is removed, at which point it belongs to no Product Container at all).
        ii. Section: Moving Items to a Product Container

            When an Item is dragged into a Product Container, the logic is as follows:

            Target Product Container = the Product Container the user dropped the Item on

            Target Storage Unit = the Storage Unit containing the Target Product Container

            If the Item's Product is already in a Product Container in the Target Storage Unit

                Move the Product and all associated Items from their old Product Container to the Target Product Container

            Else

                Add the Product to the Target Product Container

            Move the selected Item from its old Product Container to the Target Product Container
        iii. Section: Transferring Items to a Storage Unit

When an Item is transferred into a Storage Unit, it is added to the Product Container within the target Storage Unit that contains the Item's Product. If the Item's Product is not already in the Storage Unit, it is automatically added to the Storage Unit at the top level before the Item is transferred.

c. Removing Items
    i. Section: Removing Items

When an Item is removed,
1. The Item is removed from its containing Storage Unit.
2. The Exit Time is stored in the Item.
3. The Item is retained for historical purposes (i.e., for calculating statistics and reporting).

d. Putting Products in a Product Container
    i. Section: Data Dictionary

A Product may be in any number of Storage Units. However, a Product may not be in multiple different Product Containers within the same Storage Unit at the same time. That is, a Product may appear at most once in a particular Storage Unit.

    ii. Section: Putting Products in a Product Container

When a Product is dragged into a Product Container, the logic is as follows:

Target Product Container = the Product Container the user dropped the Product on

Target Storage Unit = the Storage Unit containing the Target Product Container

If the Product is already contained in a Product Container in the Target Storage Unit
    Move the Product and all associated Items from their old Product Container to the Target Product Container
Else
    Add the Product to the Target Product Container

e. Deleting Products
    i. Section: Deleting Products

A Product may be deleted from a Product Container only if there are no Items of the Product remaining in the Product Container.

A Product may be deleted from the system only if there are no Items of the Product remaining in the system.

Error Handling

Part of class design is deciding how errors will be reported to clients. When an operation fails, the caller must be notified of the error and given information about the nature of the error. The two primary design choices for doing this are Result Objects and Exceptions. You should pick one of these approaches, and apply it consistently throughout your design. Identify all operations that could fail, and have them return Result Objects or throw Exceptions (whichever you decide). It is unacceptable for an operation to silently fail without notifying the caller of the error. For example, if the caller tries to create or modify data in an illegal way, the operation should be rejected, and an error returned to the caller.

Enable/Disable Support

The primary client for the core object model will be the user interface (UI) layer. The UI will often need to ask the model if data entered by the user is valid or if a particular operation is allowed so that it can enable or disable each UI control appropriately. For example, the OK button in a dialog box should be disabled unless the data entered by the user is valid, in which case it should be enabled. How does the UI determine if the data entered by the user is valid? It asks the model. Therefore, your core model must provide operations that allow the UI to ask if user input is valid, if particular operations are allowed, etc.

Simple Persistence

Inventory Tracker stores data persistently on disk so that users can exit the application without losing information. In Phase 1 you will implement data persistence using Java Serialization. This will allow you to save your core object model to/from disk with relatively little effort. In Phase 4 you will replace this with a more realistic persistence implementation that uses a Relational Database instead of Java Serialization.

The Serialization implementation (i.e., Phases 1, 2, & 3) will use the following approach:

1) At startup the entire object model will be loaded from disk into memory
2) All runtime operations will be performed on the in-memory object model
3) When the user exits the program, the in-memory object model will be saved to disk

The Relational Database implementation (i.e., Phase 4) will use a slightly different approach:

1) At startup the entire object model will be loaded from the database into memory
2) All runtime operations will be performed on the in-memory object model
3) All changes to the in-memory object model will be immediately saved to the database. (This way there will be no need to save the in-memory model to the database when the user exits, because all changes will have already been saved.)

For Phase 1 all you need to do is design a class (or two) that is responsible for saving and loading your core object model using Java Serialization. Java Serialization makes it very simple to save and load an in-memory object graph to/from disk. Using the `ObjectOutputStream` class, you can save an entire graph of objects to a file with

one method call. Similarly, using the `ObjectInputStream` class, an object graph can be read back into memory with a single method call. Java does all the work required to follow links between objects and save them to disk. You just need to implement the `Serializable` interface on all of your model classes that need to be persisted.

NOTE: Java Serialization does not save the values of static variables. Static variables lose their values when the object model is saved and then re-loaded from disk. Keep this in mind.

- Provide Javadoc documentation for your class (or classes) that is responsible for saving/loading your object model to/from disk.

Design Division of Labor

All team members should help create the class diagrams and Javadoc documentation for the Core Object Model and Persistence areas.

Do not underestimate the amount of work required to create the Javadocs for this phase. The Javadocs are an important part of this (and every) phase. Since many new classes will be created in this phase, substantial work is required to create the Javadocs. Therefore, all team members should help. Doing a cursory, incomplete, or inconsistent job on the Javadocs will significantly harm your grade, so please take the Javadocs seriously, and make sure they are consistent across all team members.

Implementation

Implement your Core Object Model and Simple Persistence designs. For each public method in your code, do the following:

1. At the top of the method, use Java assert statements to define and check all of the method's pre-conditions.
2. In the `test-src` directory of your project, implement JUnit tests that define and validate all of the method's post-conditions.

This applies both to simple methods such as getters and setters, and more complex operations such as those listed previously in the Data Integrity Enforcement section.

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. This is for cases where you disagree with

Checkstyle, and don't want to follow its advice. Your test cases do not need to pass Checkstyle.

To pass off your Phase 1 implementation, the TA will review your code to ensure that all functionality has been implemented, and that all public methods define and check their pre-conditions with assert statements.

Additionally, the TA will review your JUnit test code to ensure that it is well done. To assist the TA in this effort, you should submit a group report (PDF file) on the course web site that explains exactly where each of the data integrity constraints listed in the Data Integrity Enforcement section are 1) implemented, and 2) tested in your JUnit code.

Implementation Division of Labor

Everyone should help implement the core object model and write the test cases for the classes they write.

Deliverables

- ☐ Group Design
  - UML diagrams (submit PDF file online)
  - Javadocs (submit a zip file online)
- ☐ Group Implementation
  - All core object model and simple persistence functionality implemented
  - All public methods define and check their pre-conditions with assert statements
  - All public methods have JUnit test cases that define and validate their post-conditions
  - All files checked into Subversion
  - Everything should build and run successfully
  - Remove Checkstyle errors
- ☐ Group Data Integrity Constraint Report (submit PDF file online)
- ☐ Individual Phase Report / Exam (submit PDF file online)