CPS122 - OBJECT-ORIENTED SOFTWARE DEVELOPMENT

Notes on the "Starter Project" for the Team Project

Project Structure

The "starter" NetBeans project includes a partial GUI for the course project plus several other classes (all partially or fully implemented.) It consists of three packages that use a layered structure with some slight exceptions.

- `library.model`: Classes in this package represent the entities the system manipulates - i.e. those appearing in your Class Diagram. Though they may need to depend on each other, they should not depend on classes in the other packages - e.g. a statement of the form `import library.controller...` or `import library.gui...` should not appear in any class in this package.
- `library.controller`: Classes in this package implement the various use cases of the system. In the case of the use cases for iteration 1, it is appropriate to create a separate class for it (e.g. `CheckoutUseCase` in the code you have been given). Responsibility for several simpler (but related) use cases for iteration 2 may be assigned to a single class, though you can create a separate class for each if you wish. Classes in this package may depend on each other and will definitely need to depend on classes in the `model` package, but they should generally not depend on classes in the `gui` package except for the utility methods in class GUI - e.g. while `import library.gui.GUI` may be needed, a statement of the form `import library.gui...` for any other class should not appear in any class in this package other than for this purpose. (The main class `library.controller.Main` is an exception to the layered structure, since it must create and show the GUI at program startup.)
- `library.gui`: Classes in this package constitute the graphical user interface - e.g. they display information to the user and accept gestures from the user. Classes in this package may depend on each other and will definitely need to depend on classes in the `controller` package, but dependencies on classes in the `model` package should be restricted to dependencies between classes in the `library.gui` package and specific, related classes in the `library.model` package - e.g. `library.gui.DateCard` depends on `library.model.SimpleDate`, the class(es) that provide(s) a graphical interface for adding/modifying patrons will depend on the representation for a patron in the model, etc.

Use of the Singleton Pattern

Several classes in the project make use of the singleton pattern, which is a design pattern that is appropriate when the the logic of the system is such that exactly one instance of the class must exist, and several other classes need to be able to access this one object. This is true of `library.controller.Main`, `library.controller.LoginOutUseCase`, `library.gui.GUI`, and `library.model.LibraryDatabase`, Such a class is implemented as follows:

- The class has a `private static` variable (called something like `theWhatever`) which holds a reference to the one and only object of this class.
- The class has a method called `getInstance()` which returns a reference to the one and only object of this class (i.e. the value of the variable `theWhatever`). If this variable is `null` (which it will be when the program starts), this method first creates the object. Thus, the first call to this method actually creates the object; all subsequent calls return a reference to it.
- The class's constructor is `private`, which ensures that only `getInstance()` will create an object of this class.

Note that the singleton pattern is used for the LoginLogout use cases, because state (whether or not a manager is logged in) needs to be preserved across multiple uses. It is **not** appropriate for checking books out, because while state needs to be preserved across the different actor actions during a single checkout, each checkout involves different state (and hence a new object is needed in each case). Likewise, is **not** needed for use cases like Return, Renew, and Status because there is no state that needs to be preserved.

Two `gui` classes use a variant of this pattern which allows for explicit construction by class `GUI` while still supporting access by other classes using `getInstance()`.

Saving the Database to a File

Database saving/ loading will be done using a Java mechanism called serialization. We will not be able to cover this in class until we get to Input-Output later in the course. For now, you will need to use some "cookbook" code and code written for you in the starter project. The approach described here will use Java serialization to provide persistence for data that needs to be saved between program runs. There is no need to provide for multiple data files - therefore, the name `library.database` has been "hardwired" into the program. (Thus, you will need the traditional File Save and Quit options, but not New, Open, or Save As.)

This is how this should work:

a. When the program starts up, it will look for a file with this name and read it in to initialize the stored data. If no file with the specified name exists, it will create a new, empty internal database.

b. When a user quits the program, it will first write its internal database to the a file with the specified name.

c. The `File` menu contains a `Save` option to allow users to save the state of the internal database to the file at any time - it would not do to risk losing an entire day's activities if there is a system crash a few minutes before closing!

Support for this has been built into the code given to you, as follows:

- The class `library.model.LibraryDatabase` has a method called `save()`, which should be called when the user chooses the `Save` option in the `File` menu, or when the user quits the program. (A call to this method is already present in the action listeners for the Save and Quit menu items.) Code in this method writes the entire database to a file called `library.database`; however, as the project is distributed this feature is disabled (and a popup dialog appears instead.) When you are ready to start having the database saved, you will need to remove the line that produces the pop-up dialog and the comment markers around the code that actually saves the database. **<u>Do not leave the popup dialog in the code you turn in!</u>**
- When the singleton accessor for the `library.model.LibraryDatabase` class needs to create the singleton object, it first attempts to read it from a file in the project called `library.database`. If this file does not exist, it calls the constructor to create a new one.
- When a class that is stored in the database is changed in certain ways, it may be that objects saved by a program using the "old" version of the class cannot be read by a program using the "new" version of the class. This will result in a crash when starting the program, typically with an `InvalidClassException`. Should this happen, you should delete the existing `library.database` file, which was based on code that is now obselete. Then your program should load correctly - though, of course, any information you had saved will have been lost.

The class `SimpleDate`

The `model` package contains a class named `SimpleDate` which you can use for representing things like due dates. It has useful methods for things like comparing dates. Carefully familiarizing yourself with its Javadoc can save you hours of grief figuring out how to do things!

Useful Methods in `library.gui.GUI`

This class contains methods `showMessage()`, `askQuestion()`, `askYesNoQuestion()`, `requestPassword()` and `notImplemented()` that can be used for simple dialogs with the user via a popup dialog without creating a separate card or pane. There are two examples of using the first of these in `library.controller.StatusUseCase`. Since these are instance methods of the `GUI`, they must always be used with the singleton instance obtained via `GUI.getInstance()`.

It also contains the methods `gotoCard()` and `goBack()` that can be used to go back and forth between GUI cards. Again, there are examples of using these in `library.gui.CheckoutPane` and `library.gui.CheckoutDetailsCard`.

Javadoc

You can view the javadoc for the project by opening `dist/javadoc/index.html` in the project in a web browser. Initially, this will be the javadoc for the project as it is given to you. If you rebuild the javadoc while working on the project, this will always give you the most recent version.

Accessing the Project from two or more Computers

Your team will need to share a single copy of the project for the team by using something like OneDrive or Google Drive or DropBox. **Do not** open a copy of the team project on two different computers at the same time. This can result in loss of work and strange problems in some cases.

Structure of the GUI

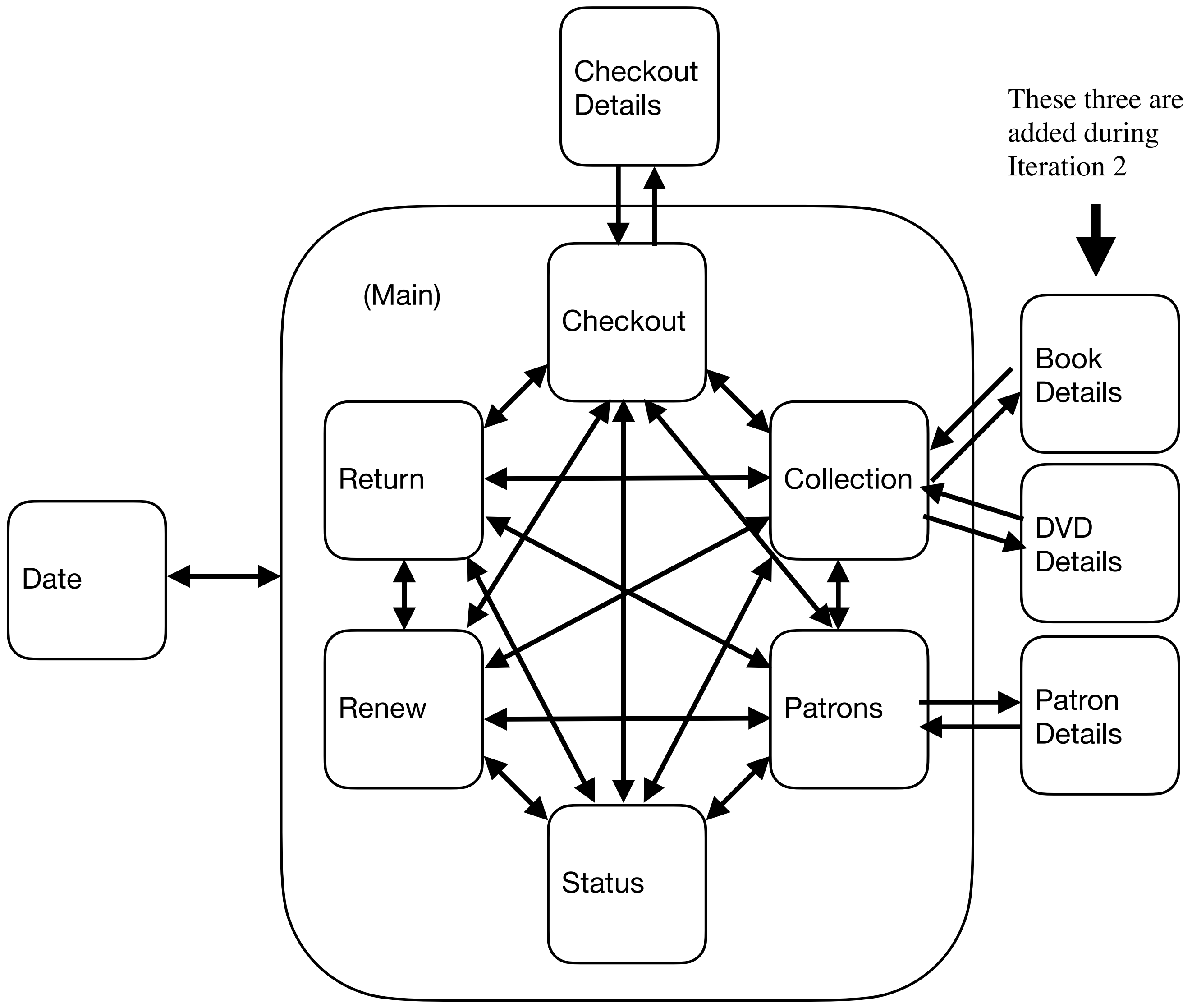
The diagrams below show two things;

1. The overall structure of the GUI after Iteration 2 has been completed. (Book, DVD, and Patron Details are added during Iteration 2; everything else is part of the GUI inclujded in the starter project.)

2. The structure of the GUI for entering Checkout Details. The GUI itself is included in the starter code, but in Iteration 1 you will need to implement the code needed to actually carry out the operations requested by the GUI.

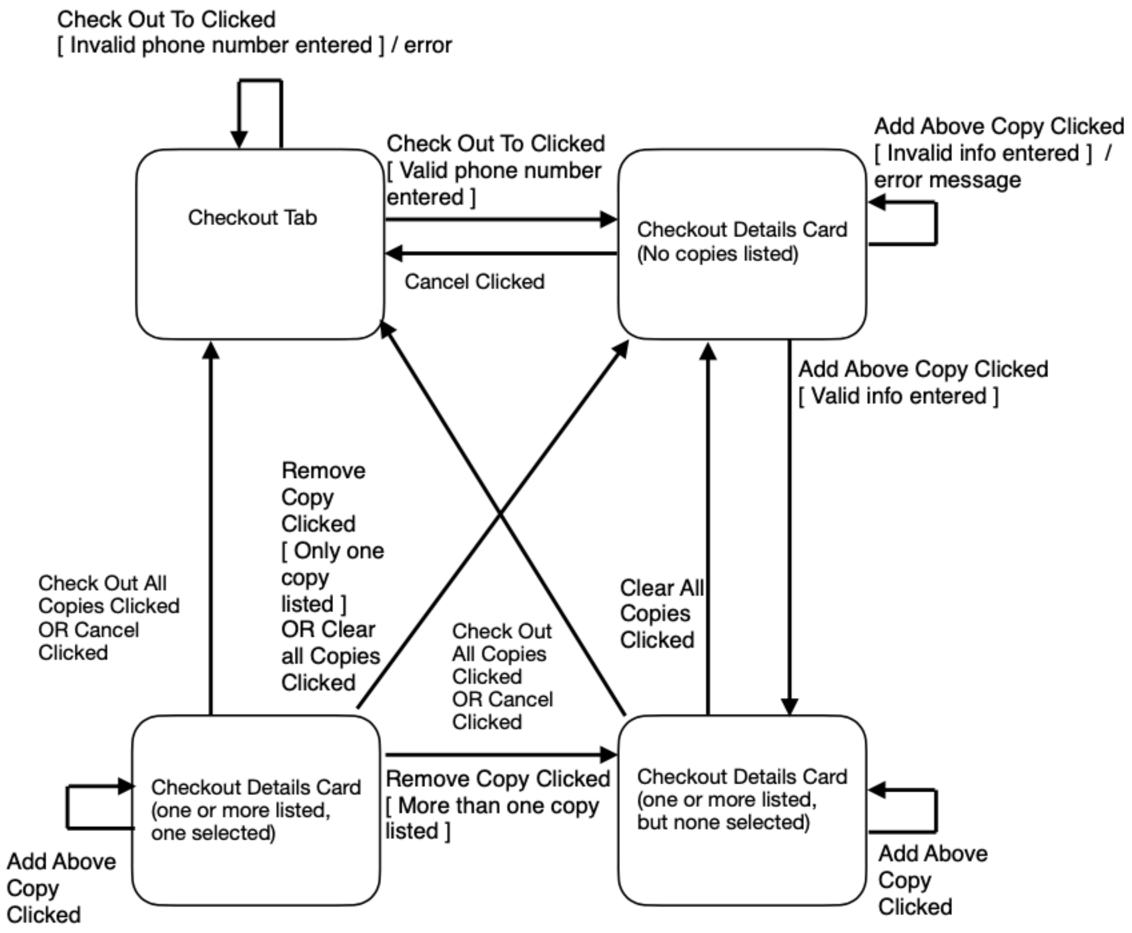
As initially distributed, the GUI consists of three cards (Main, Date, and Checkout Details), of which only one is visible at a time. The Main card, in turn is a tabbed pane that shows just one of six tabs at any time. The tabs in the main card correspond to the program functions (individual or a group of related functions). The Date card is shown when the "Diddle" option is chosen in the Date menu. The Checkout Details card is used during the Checkout function, and the Book, DVD, and Patron Details cards are used during the Add functions in Iteration 2 and for the Update and Delete functions in later iterations.

Overall GUI



Checkout Details
These three are added during Iteration 2
(Main)
Checkout
Return
Collection
Book Details
DVD Details
Date
Renew
Patrons
Patron Details
Status

Checkout Details GUI


Check Out To Clicked
[Invalid phone number entered] / error
Checkout Tab
Check Out To Clicked
[Valid phone number entered]
Cancel Clicked
Checkout Details Card (No copies listed)
Add Above Copy Clicked
[Invalid info entered] / error message
Add Above Copy Clicked
[Valid info entered]
Check Out All Copies Clicked OR Cancel Clicked
Remove Copy Clicked
[Only one copy listed]
OR Clear all Copies Clicked
Check Out All Copies Clicked OR Cancel Clicked
Clear All Copies Clicked
Checkout Details Card (one or more listed, one selected)
Remove Copy Clicked
[More than one copy listed]
Checkout Details Card (one or more listed, but none selected)
Add Above Copy Clicked
Add Above Copy Clicked