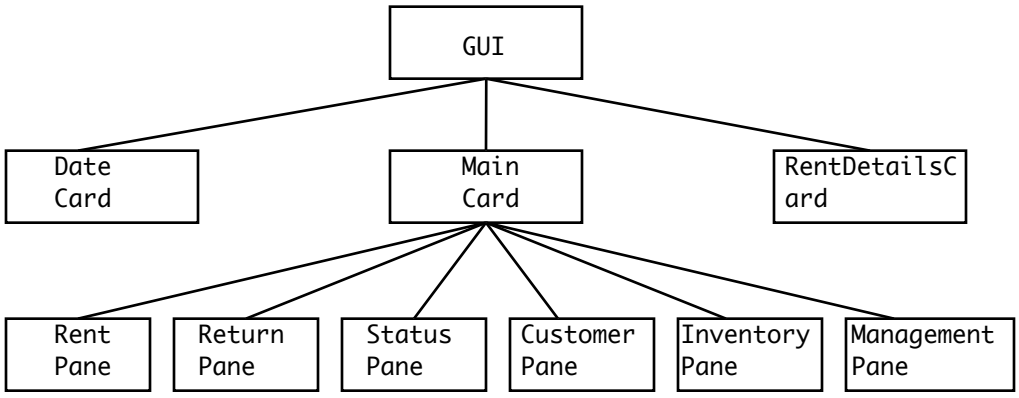
Project Structure

The common volume contains a "starter" NetBeans project including a partial GUI for the course project. It consists of three packages.

- `videostore.model`: Classes in this package represent the entities the system manipulates - i.e. those appearing in your Class Diagram. Though they will need to depend on each other, they should not depend on classes in the other packages - e.g. a statement of the form `import videostore.controller...` or `import videostore.gui...` should not appear in any class in this package.
- `videostore.controller`: Classes in this package implement the various use cases of the system. In the case of complex use cases, it is appropriate to create a separate class for it (e.g.`RentUseCase` in the code you have been given). Responsibility for several simpler (but related) use cases 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 - e.g. a statement of the form `import videostore.gui...` should usually not appear in any class in this package. (The `main()` routine in class `videostore.controller.Main` is an exception, since it must create and show the GUI at program startup.)
- `videostore.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 `videostore.gui` package and specific, related classes in the `videostore.model` package - e.g. `videostore.gui.DateCard` depends on `videostore.model.SimpleDate`, the class (es) that provide(s) a graphical interface for adding/modifying customers will depend on the representation for a customer in the model, etc.

Structure of the GUI

The graphical user interface consists of a set of cards, only one of which is showing at a time. One of these (the "main" card) consists, in turn, of a tabbed pane with a set of individual panes, only one of which is showing when the main card is showing. The initial class structure of the GUI is as follows, but you are certainly free to add additional classes to it.



An additional card can be added by creating a new class as a `JPanel` form, and then creating an instance of it in the constructor of class `videostore.gui.GUI` as done with the other cards. An additional pane can be added to the main card by creating a new class as a `JPanel` form, and then creating an instance of it in the constructor of class `videostore.gui.MainCard` as done with the other panes.

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 only one instance of the class can ever exist, and other classes need to share access to this one object. This is true of `videostore.model.StoreDatabase` as well as the classes in the `videostore.controller` and some of the classes in the `videostore.gui` packages. 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.

Saving the Database

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 class `videostore.model.StoreDatabase` has a method called `save()`, which is called when the user chooses the `Save` option in the `File` menu, or when the user quits the program. Code in this method writes the entire database to a file called `video.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 dialog and the comment markers around the code that actually saves the database. Do not leave the popup dialog in the code you turn in!

- When the singleton accessor for the `videostore.model.StoreDatabase` class needs to create the singleton object, it first attempts to read it from a file in the project called `video.database`. If this file does not exist, it calls the constructor to create a new one.

- In order for all this to work properly, you need to do two things in each of the classes in the `model` package only. (Do not do this in the other two packages):

  - Each class must include `implements java.io.Serializable.` in its class declaration.
  - Each class must include the following symbolic constant definition:
    `static final long serialVersionUID = 1;`

    (Both of these are similar to what has been done for you in the classes in this package that have already been written for you.)

- 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 `video.database` file. Then your program should load correctly - though, of course, any information you had saved will have been lost.

Creating Test Data for Iteration 1

In order to be able to test iteration 1, it will, of course, be necessary to have some customers, titles, and copies in the database. However, the use cases for creating these are part of iteration 2, So how can you test iteration 1?

As originally written, the `videostore.model.StoreDatabase` constructor contains a call to a method called `createDummyData()`. This method should create some "dummy" objects that you will use for initial testing for iteration 1. Of course, when you get to iteration 2 and are using "real" data, you should delete `createDummyData()` and the call to it in the constructor!

Formatting Dollar Amounts

The project represents dollar amounts as doubles - e.g. $2.50 is represented by the number 2.5. The `java.text.NumberFormat` class provides a mechanism for converting a real number into a character string formatting using standard formatting for dollar amounts (e.g. $, two decimal places, and commas separating thousands). You should use this whenever you need to convert a dollar amount to a string - see `updateAmountToCollect()` in `videostore.gui.RentDetailsCard` for an example of using this.

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 `videostore.gui.GUI`

This class contains methods `showMessage()`, `askQuestion()`, and `askYesNoQuestion()` that can be used for simple dialogs with the user via a popup dialog without creating a separate card or pane. There are several examples of using the first of these in other classes written for you.

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.

Displayable Collections

The starter project also contains as a library the `displayablecollections` package. Documentation for this package is on the course web site under "Project-related materials". Because the library is installed into the project, you can use any of the classes in the package by importing the same way you would import something from the standard Java library - e.g something like.

```
import displayablecollections.DisplayableMap;
...
DisplayableMap customers;
```

We will discuss this further during our class sessions on Design Patterns. Until then, you can simply ignore this facility. In particular, you will not need it for Iteration 1, but you may find it useful for Iteration 2 - though you are not required to use it!