Handed out: Thursday, March 6, 2008
Due: Tuesday, March 25, 2008, by 10PM using Web Turnin

For this project you will be developing a Graphical User Interface (GUI) for an ICritters interaction sandbox called the "ICritter Training Facility". You will be using Java's Swing API for the GUI, and you will also be implementing your system using both the Model-View-Controller (MVC) and Observer design patterns.

Your application will allow a user to load up a set of ICritters, and focusing on one or two of them, edit their data and watch what happens when they interact or participate in a jamboree.

You will be responsible for implementing both the GUI (which acts as both a view and controller) and a model (containing the relevant application data).

You will be given an Eclipse project zip file with all of the necessary icritters.base classes (in a Jar file) along with an icritters.gui Interface class (src/icritters/gui/ModelInterface.java), a set of simple icons (in src/icritters/gui/images), and the default ICritters text file (default.icr). Details on the files are near the end of this document.

This is another pair programming project, and it is intended to continue preparing you for your final project. The Model-View-Controller and Observer (AKA Publish-Subscribe) design patterns contained herein are very important and will assuredly come in handy in the (possibly very near) future.

Diagrams:

Two diagram pages are included with the text of this document. One shows the two top-most levels of layout in the application, and the other names the pieces of the GUI for cross-reference with the text document. These are intended to help you understand the specification and give you an idea on what your application could look like.

GUI conformity:

The GUI must have the functionality listed in this specification, but if you want to use different layout managers (we only describe the top two levels), or even different images, that's ok.

Model-View-Controller pattern:

In the MVC pattern, the model contains the data of your application. A view (there can be several) presents some (possibly all) of the model's data to the user. The controller handles user input and may invoke changes upon the model.

For this assignment, we have specified a ModelInterface which your model must implement.

The view (again, the presentation of the model's data to the user) and controller (which provides the user a way to modify the Model's data) will be implemented via your GUI code.

Observer pattern:

Java supplies a useful mechanism with which to implement the Observer (or Publish-Subscribe) design pattern by providing the Observer interface and the Observable Class.

Briefly, Observer objects "observe" Observable objects and are notified when changes occur to the Observable object(s).

One analogy would be that a group of hungry college students are waiting for a pizza to be delivered. They become tired of waiting and wander off, leaving one person to wait for the pizza. It is understood that this person is to notify the rest of them (via their ubiquitous cell-phones) when the pizza arrives. In this example, the person waiting is the Observable object, who manages the important data (pizza arrival), and notifies the Observers (the horde of hungry wandering students) if there is a change in state.

Your model will be the Observable object and one or more of your GUI objects will be the Observers.

Details on implementing the Observer pattern in Java:

An object which will be observed by other objects must extend Observable. (Your Model class must extend Observable.)

An object which wants to observe (and be notified of changes upon) an Observable object should implement Observer. (Some class(es) in your GUI must implement Observer). When implementing Observer, a class must define the following method:

```
public void update( Observable observable, Object o )
```

This method will be called when the notifyObservers method is called on the Observable object if and only if the Observable object had been changed (by the Observable calling the setChanged method, described below).

The mysterious second argument passed to the update method (Object o), is typically null. The Observable object has the option of passing something to the Observers (when calling the notifyObservers method). This could be used to indicate what changed. You are not required to use this argument.

To be notified of updates to a particular Observable object, an Observer object must be added to the Observable’s list of Observers by calling the addObserver method.

```
For example:  myObservable.addObserver( myObserver )
```

When an Observable object (i.e. your Model) changes internal data that an Observer is interested in (like the outcome text), then the Observable object must call the setChanged method to indicate that it has changed. This does not notify the Observers, it merely indicates that the Observable object has changed. To notify the Observers (i.e. get their update methods called), the notifyObservers method must also be called.

So, typically a call to the setChanged method is followed immediately by a call to the notifyObservers method in the Observable class' code. This would look like the following snippet of code, which would be in a method in the Observable class.

```
// Modify the thing
someArray[ anIndex ] = newValue;

// Things have changed, set the changed flag
this.setChanged();

// Alert the Observers that something has changed.
this.notifyObservers();
```

MVC meets Observer:

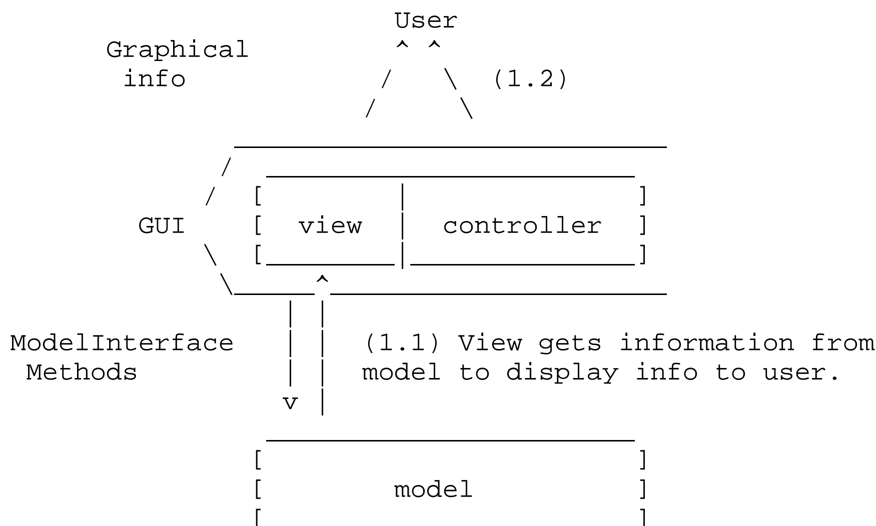
The GUI puts together a "view" using information it retrieves from the model using methods from the ModelInterface. Your GUI will have some "control" elements like buttons as well. When the user clicks on a button, they are interacting with the "controller" (specifically via an ActionListener in the GUI).

The controller code in the GUI can affect the model by using methods from the ModelInterface Class. If the model has been modified, then the model would call the setChanged method, and the notifyObservers method. This would initiate a call to the update method on any Observers of the model.

A walkthrough of these steps follows:

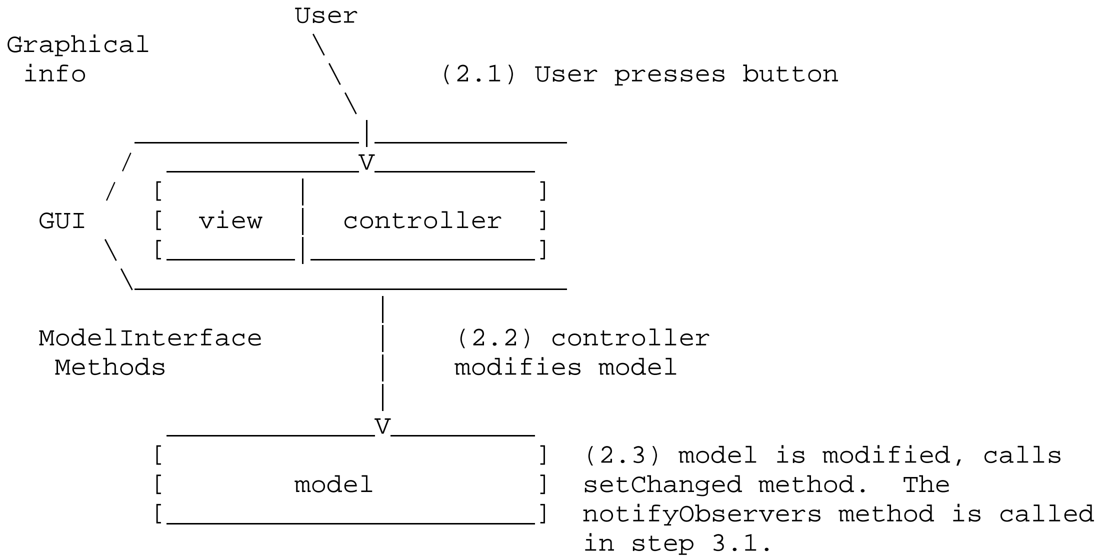
Step 1: Initialization

1.1 Code in GUI gets information needed from model.
1.2 Code in GUI generates graphical info "view" and controller components (buttons, etc.) which is presented to the User.



Step 2: User Interaction

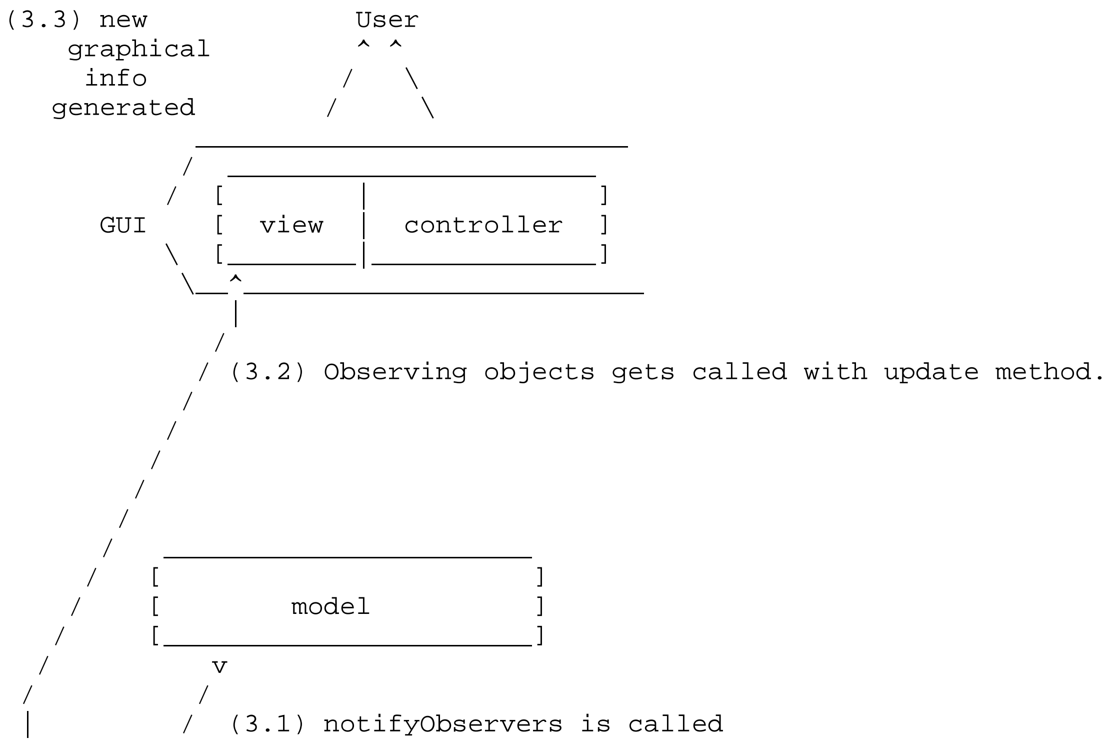
2.1 User interacts with GUI, which generates a call to the controller (an ActionListener).
2.2 controller modifies model using method(s) in ModelInterface.
2.3 model calls setChanged



Step 3: Model is changed, GUI is updated

Note order carefully (order of operations is from bottom-up):

3.1 model calls notifyObservers,
3.2 Observers get update method,
3.3 The graphical info (view) is refreshed (back to step 1.1)



The ModelInterface (in icritters.gui) and your Model class:

You have been given a ModelInterface which you must implement in your Model class (which should go in icritters.gui as well). This interface defines the inputs and outputs of your Model class. Your Model class must contain the data that your application will be acting upon, some of which is described below. By encapsulating the data in one class (your Model class) you can easily control (using the Observer pattern) what happens when data is changed.

In addition to implementing the ModelInterface, your Model class must also extend Observable. By doing this, object(s) in your GUI can add themselves as Observers, be notified when changes occur, and redraw themselves as needed.

Next, we explain some concepts in the ModelInterface. Some of these correspond directly to GUI components (JButtons, JPanels, etc.) in the GUI, so keep the diagrams handy while looking at these terms.

Indices to ICritters:

Your Model needs to have access to an ICritterWorld, whose resident ICritters will be referenced by their place index in the List<ICritter> returned by ICritterWorld's listICritters method. This index is called an ICritter index.

Your Model probably will contain an ICritterWorld:

Your Model is responsible for maintaining an ICritterWorld, and while the interface doesn't force you to have an ICritterWorld in your Model class, it probably makes sense to contain one.

The following methods relate to the ICritterWorld in your Model and accessing individual ICritters. Note that all accessing of ICritters is done via their index (see Indices to ICritters above).

public ICritter getWorldICritter(int iCritterIndex)
Return the ICritter instance at specified iCritterIndex location in the list of ICritters contained in your Model's ICritterWorld object.

public List<ICritter> getWorldICritters()
Return a list of the ICritters in the ICritterWorld stored by the model.

public boolean loadICritterFile(String filePath)
Clear the current ICritterWorld, reset the workingSlots to ModelInterface.EMPTY, and load ICritters in the text file (which is in the ICritter File Format specified below) into the ICritterWorld. Returns true if load was successful, false otherwise.

What ICritters are being "worked-on" in your application.

Your Model is responsible for keeping track of what ICritters are currently under scrutiny in the two "working" ICritter slots (top-left and top-center of the GUI). These will be called "working slots". Working slot 0 is the left slot, working slot 1 is the right slot. Your Model must provide access to the contents of these slots via the {get/set}WorkingSlot methods. Working slots should be ModelInterface.EMPTY when they are empty. Note that these slots should contain ints which are indices to ICritters in the model's world (see Indices to ICritters above). Methods relating to working slots follow.

public void setWorkingSlot(int workingSlot, int iCritterIndex)
Set the workingSlot indicated to the iCritterIndex specified.

public int getWorkingSlot(int workingSlot)
Return the index of the ICritter in the workingSlot specified.
Returns ModelInterface.EMPTY if the slot is empty.

WorkingSlot and ICritterIndex example:

Suppose your ICritterWorld contains ICritters: "Socks", "Spot", and "Felix", and the GUI currently had "Felix" in the first workingSlot (workingSlot 0), then getWorkingSlot(0) would return 2 (being the index of "Felix" in the list initially described above).

Selected WorkingSlot:

Remember that this application maintains two "working slots" for ICritters. These are represented by the top-left, and top-center panels called workingPanels[0,1] (which contain workingButtons[0,1]).

Your Model must keep track of which working slot is currently selected. This working slot is "filled" with whatever ICritter is selected from the selector panel.

For example, continuing with the world given in the WorkingSlot and ICritterIndex example: if "Felix" (with ICritterIndex of 2) is selected from the selector panel while the selected working slot is 0, then 2 (the iCritter index of "Felix") should be stored in the Model's working slot 0. This would be done by your GUI code and would look like:

```
setWorkingSlot( getSelectedWorkingSlot(), 2 )
```

The getters and setters of the selected working slots are:

public int getSelectedWorkingSlot()
Return the workingSlot index of the selected working slot. 0 for left, 1 for right.

public void setSelectedWorkingSlot(int workingSlotIndex)
Set the selected working slot index to the working slot index specified, where workingSlotIndex will be 0 for left, 1 for right.

Outcome text:

The interact and runJamboree methods from ICritter have been modified to return Strings representing the outcome of their respective operations.

Your model should keep track of the output that is generated by the interact and runJamboree methods. The getOutcomeText method lets your GUI get that text. Both doRunJamboree and doInteract should append to this text. New output must be appended to the end of the outcome String.

public String getOutcomeText()
Return the complete outcome text accumulated by the application.

The remaining methods of ModelInterface are:

public void doInteract(int iCritterIndexA, int iCritterIndexB)
Generates a call to interact with ICritter at iCritterIndexA being the interactor, and ICritter at iCritterIndexB being the interactee. Note that all output from this call must be appended to the outcome text.

public void doRunJamboree()
Conduct a jamboree on the ICritterWorld in the model. Note that all output from this call must be appended to the outcome text.

NOTE: that it will be up to you to figure out when to call setChanged and notifyObservers in your Model Class, though some hints are in the spec.

The GUI:

Here is a walkthrough of what your application is expected to do:

- When the application is started, the default ICritters (default.icr) file is loaded. The ICritters loaded will be displayed in the selectorPanel on the right of the GUI as small icons.

- The application provides two "workingSlots" (top-left and top-center) in which to view and/or edit one of the ICritters from the selectorPanel. The workingButtons inside the workingPanels each have an empty icon image and display "empty" as lablels.

- The starting selectedWorkingSlot (0 for left, 1 for right) defaults to 0, left. The User can switch the selectedWorkingSlot by clicking a corresponding workingButton or by selecting the correspinding workingRadioButton.

- The user has the option of selecting (from the selectorPanel) any one of the ICritters displayed there. If an ICritter is selected, its larger image appears in the workingButton indicated by the selectedWorkingSlot. For example, since the default starting selectedWorkingSlot is 0, if a user clicked on an ICritter in the selectorPanel without changing the selectedWorkingSlot, the ICritter would "appear" on the left side.

- The user can change the selected working slot by either clicking on the appropriate workingButton or by clicking on the appropriate workingRadioButton.

- Since there are no ICritters in the workingSlots at the start of the application, both Edit and Interact buttons should disabled.

- An Edit button is enabled when its corresponding workingSlot is "filled" with an ICritter. If an Edit button is selected, then an Edit Frame pops up for the ICritter in the corresponding workingSlot.

- The Interact buttons are enabled only when both workingSlots are "filled". The left-most Interact button (if clicked when enabled) would cause the left-most (workingSlot 0) ICritter to interact with the right-most (workingSlot 1) ICritter, and vice-versa. Output from the interaction, along with all previous output, will appear in the outcomeTextArea.

- The user can have two ICritters interact (if both working slots are filled) by selecting one of the Interact buttons.

- The user can edit an ICritter in a working slot by pressing its corresponding Edit button. This pops up an Edit Frame.

- The user will see all cumulative output in the outcomeTextArea (bottom-left of the GUI) from all of the interactions and/or jamborees.

- Both the selectorPanel and the outcomeTextArea are scrollable.

- The user can load another set of ICritters (replacing the current set) by using the "Action" menu to select "Load".

- The user can run a jamboree by using the "Action" menu to select "Run Jamboree".

- The user can quit the application "Action" menu to select "Quit". This closes the application window(s) and terminates the application. Clicking the Close ('X') box in the upper-right of the window has the same effect.

- An Edit Frame is a pop up which presents editable information to the user. The title of the Edit Frame should be the type and name of the ICritter being edited.

  All of the possible interests (the union of all the ICritters interests) should be listed in a scrollable JList. The interests of the ICritter being edited will be pre-selected.

  The Edit Frame also displays the "Correlation Cutoff" if the ICritter is of type MarineICritter. It will not display the Correlation Cutoff label and editable Correlation Cutoff field if the ICritter is not of type MarineICritter.

  The Edit Frame has Close and Cancel buttons. Both bring up a confirmation pop up, which, if answered in the negative, simply goes back to the Edit Frame. The confirmation pop up can be done with the JOptionPane.showConfirmdialog method that Swing provides.

  If the Close operation is confirmed, then the changes made in the Edit Frame are applied to the selected ICritter, and the Edit Frame closes. If the correlation cutoff value is not valid (see Testing below) then a JOptionPane.showMessageDialog should appear letting the user know that the value was invalid, no changes will be applied, and the Edit Frame is not closed.

  If the Cancel operation is confirmed, then the changes made in the Edit Frame are discarded, the ICritter is not modified, and the Edit Frame closes.

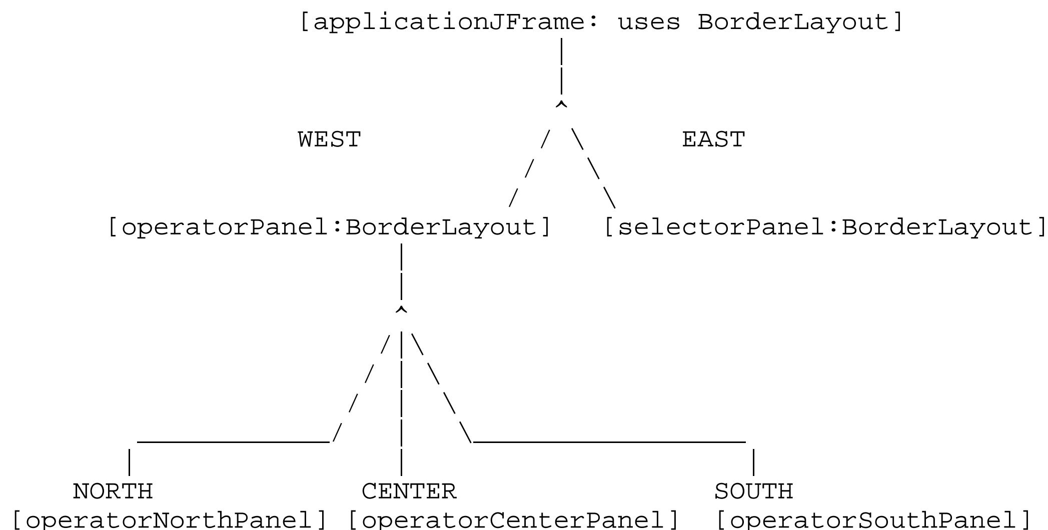
Main Jframe details:

The outermost part of the GUI is a JFrame. You should set the title of your JFrame to be "ICritter Training Facility". It should be set to be 900 pixels wide by 700 pixels high, and should not be resizable.

Edit Frame details:

We recommend you create a separate JFrame to handle this "pop up". It can be closed by calling the dispose method.

Here is a tree diagram of the first few levels of the the layout (see Layout diagram for corresponding pieces in the GUI):



ICritter File Format:

```
ICRITTER <type> <name>
INTERESTS <interest1> <interest2> ... <interestN>
FRIENDS <friendName1> <friendName2> ... <friendNameN>
[optional THRESHOLD <threshold>]
ENDICRITTER <name>
```

<type> would be ICritterDog, ICritterCat, or ICritterPenguin

Spaces are not permitted in interests, or in names.

See src/icritters/default.icr for example file.

Changes to the icritters.base package and the Jar file:

The relevant changes to the icritters.base package are the following two methods, which now both return Strings which represent the results of the interaction(s) brought about by the call:

```
class ICritter:
    public String interact( ICritter other )

class ICritterWorld:
    public String runJamboree()
```

Note that we have not given you any source code for the icritters.base package. The src/icrittersbase.jar file contains only byte-code. You can however click on the icrittersbase.jar file in the Eclipse Package Explorer and see the methods and signatures. The javadoc for a subset of the icritters.base package is also accessible through the assignments page.

Images Files:

There are 3 png image files per type of ICritter (Cat, Dog, Penguin) in src/icritters/gui/images. The numbered files (and Empty.png) are the large (250x250) workingButton images. The files called "Small" (like DogSmall.png) are the selectorPanel images. They are all usable in Swing.

Testing:

Testing your model throroughly is very important. Please turn in your test classes as well. Your application should be robust and not be broken by bad user input. To implement the correlation cutoff parser, you can use Double’s static parseDouble method and catch the NumberFormatException.

Documentation:

You will be required to fully comment using javadoc comments. You should have an @Author tag and description at the start of every class, and you should also use, at the beginning of every method, the @param, @return, and @throws tags as appropriate. (i.e. If your method doesn’t throw an exception, you don’t need an @throws tag.)

Extra Credit:

You can earn up to a total of 10% of extra credit on this assignment.

Save File (+2% Extra Credit):
Add a "Save" option to the menu bar. It should prompt for a file location to save to using JFileChooser, and popup a warning message if the file was not saved successfully.

View Friends (+4% Extra Credit):
Add a "View Friends" option to the menu bar. This will pop up a Frame which graphically display which ICritters are currently friends with whom. This would be a popup window like the edit popup.

Animate Working Slot Buttons (+2% Extra Credit):
We have provided two large ICritter image files per type of ICritter. You can animate this button, going between the two images for extra credit.

Technical Notes:

Model-View-Controller on wikipedia:
http://en.wikipedia.org/wiki/Model-view-controller

Administrative Notes:

Turnin your entire project excepting class files.

Assuming that the two members of the example pair are named John Doe and Heironymous Bosch, they decide (as you will have to in your pair) that Heironymous will be responsible for turning in the project. Since Heironymous’ section leader is Mr. Jerabek, their Eclipse project and turnin folder would be called:

```
Jerabek_Bosch_Heironymous_Doe_John_hw7
   |       \          \   \      \________________________
   |        \          \   \______________                \
   |_       _\ ______   \____________     _\__________     _\__________
  /  \     /         \   /            \  /            \   /            \
<SLlast>_<ST1lastname>_<ST1firstname>_<ST2lastname>_<ST2firstname>_hw7
```

Where SL means Section Leader and ST1 means Student 1 (in charge of turnin), and ST2 means Student 2.

You will be graded together. If you run into problems with your partner, talk to your SL immediately.

$Id: ICrittersAssignment7.txt,v 1.9 2008/03/06 07:14:24 ivan Exp $