Advanced Software Development CS525

Lab notes:

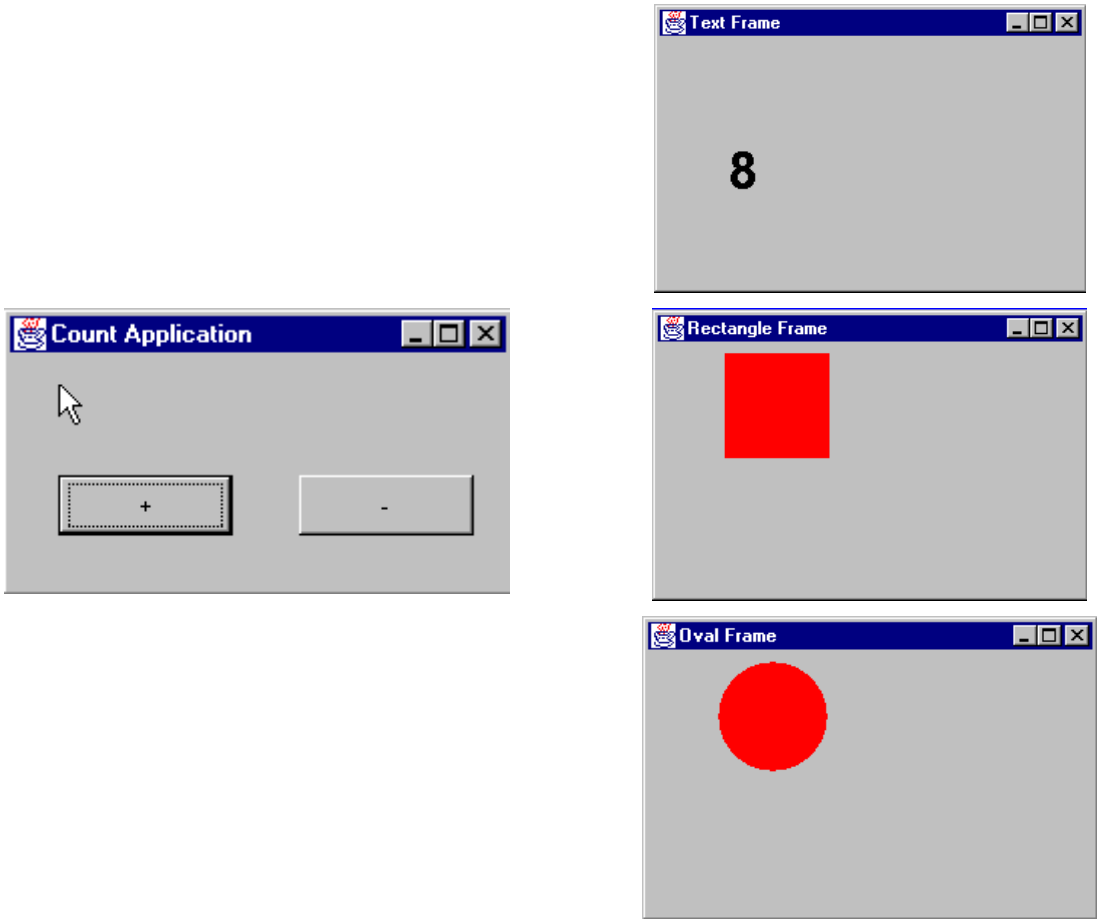
- Observer
- Strategy
- Iterator
- command
- State
- Chain of Responsibility (COR)
- Bridge
- Factory
- Visitor
- Composite

Note: In several places in the notes I pose a question about the lab and its design or implementation, something that you should be able to answer. You should pause there, and think and answer the question yourself, before reading on. Consider this like a miniature *virtual-quiz*! I'll label these places like this:

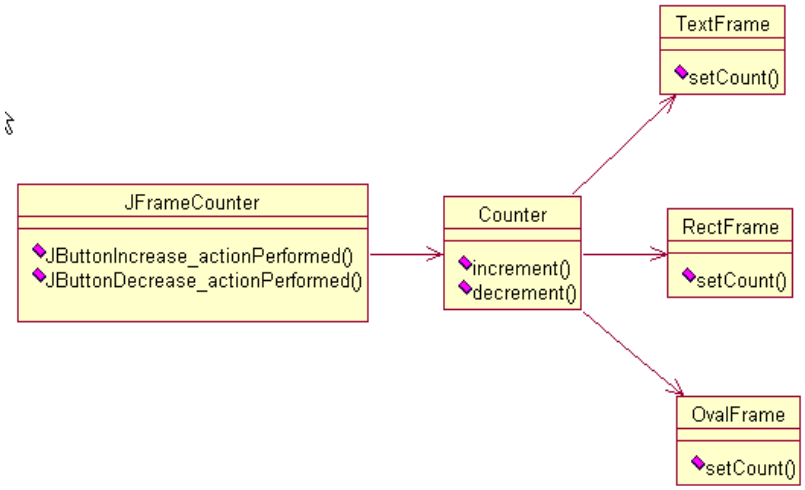
> *VQ* How would you do this ??? in your design…?

Observer Lab:

This simple lab follows directly the GOF reading example of an MVC model, with multiple views of a single model. The basic idea is that changes to the model will automatically cause updates in the views.



Since it is a first lab, we give you the design and UML (in later labs you will do the design!).



The original code is pretty simple, but has some problems:

```
public class Counter
{
...
  public void increment(){
      count++;
      textframe.setCount(count);
      rectframe.setCount(count);
      ovalframe.setCount(count);
   }

   public void decrement(){
      if (count >0){
         count--;
         textframe.setCount(count);
         rectframe.setCount(count);
         ovalframe.setCount(count);
      }
   }
}
```

VQ: What is the problem(s)?

Note that in several places, there is a tight coupling between the counting (model) logic, and the views (GUI display). This is not good, it violates the *Single Responsibility Principle*.

VQ: Describe some specific impacts of this design issue, examples of why it would lead to a poor result.

So you need a better design, that will promote a decoupling and separation (and encapsulation) of each of these two parts of the design. Hint: *Observer Pattern*. ☺

Do two versions; for the first you should implement the observer/observable pattern your self, and for the second use the Java library for these. Note the differences, and any issues in the two approaches.

Phase 2:
For a second part of the lab, have all of the views derive from a common View class. Note what impact this has on also having them be observers.

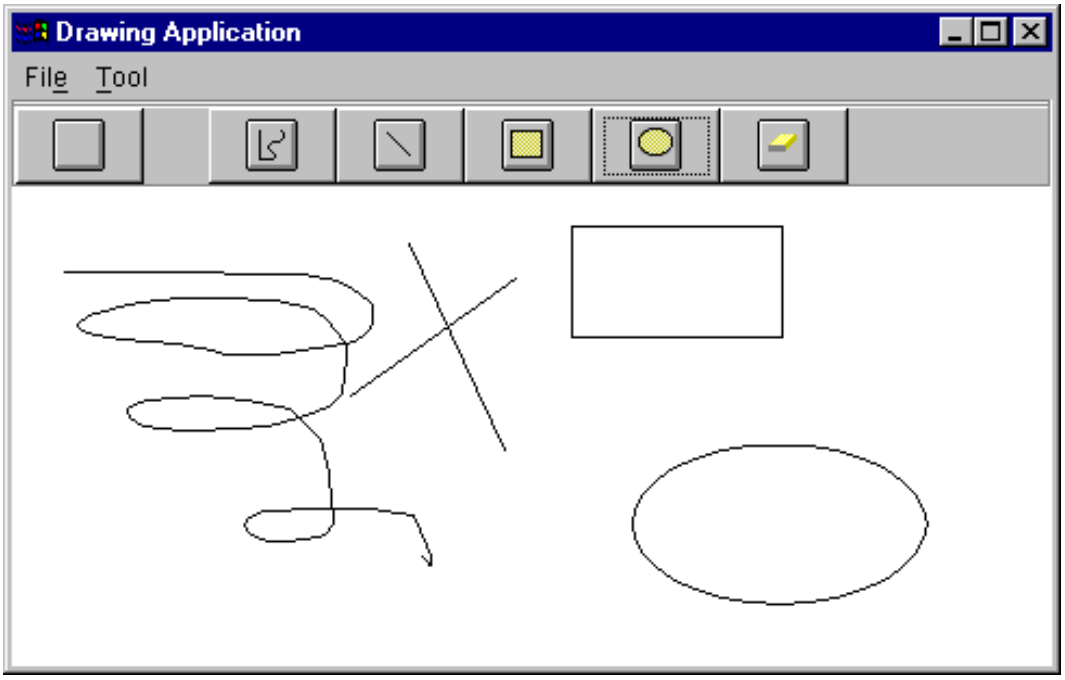
Similarly now change the counter to have a two level design; a simple counter class, and then an *observable* counter (using Java library).

> VQ: What issues arise from the new derived *ObservableCounter* which ***IsA*** Counter, and also ***IsA*** Observable?

Strategy Lab:

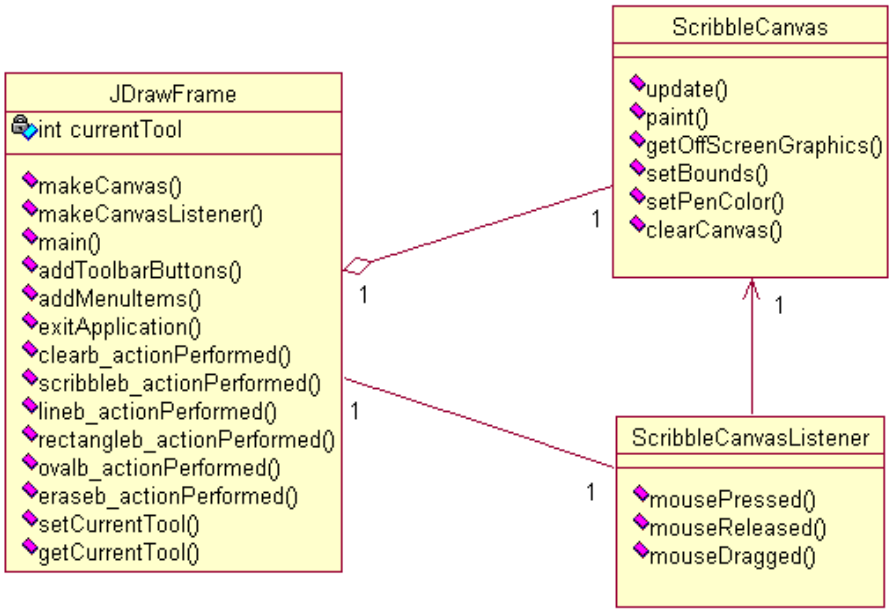
This is an interesting lab. It is purely a *re-factoring*; converting an existing working application into a revised implementation with a better design, with better Software Engineering properties.

The application is a drawing program.



The program has several modes, {rectangle, ellipse, scribble, line, and erase}. Buttons on the GUI are used to change modes. The basic idea is that each drawing mode uses a different algorithm for drawing. To change modes, we then have to be able to dynamically change algorithms; voila, the *Strategy pattern*!

The original program has the code for each drawing mode located in the *ActionListeners* (*ActionPerformed()* methods). Since each drawing mode must have an action for several different user inputs (mouse up, mouse down, mouse moved), there are parts of the algorithm in each of these AL's.



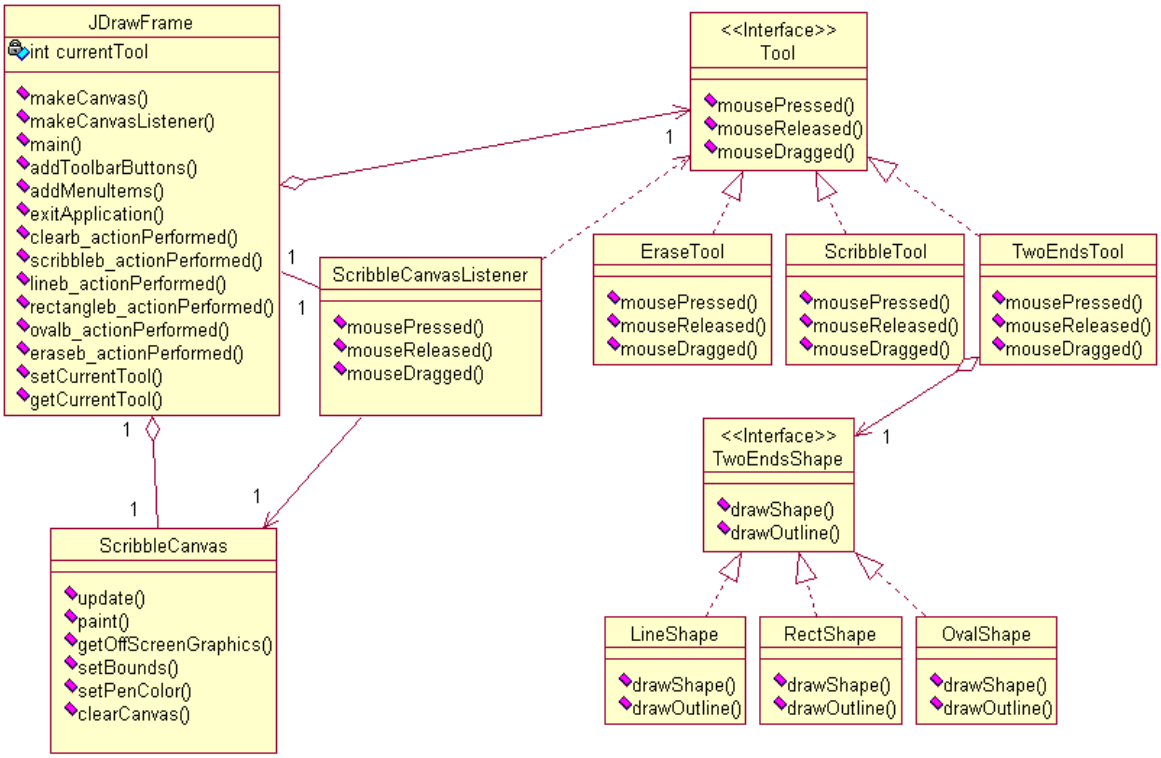
There are several problems with this. One is that the actual definition of the algorithm for any mode (e.g. draw rectangle) is not collected in any one place, nor encapsulated. It is mingled in the GUI code of the ALs, in fact in several ALs.

```
public class ScribbleCanvasListener implements MouseListener, MouseMotionListener {
 public void mousePressed(MouseEvent e) {
   switch (drawframe.getCurrentTool()){
      case 0: // handle mouse pressed for scribble tool
      case 1:  // handle mouse pressed for line tool
      case 2:// handle mouse pressed for rectangle tool
      case 3:// handle mouse pressed for oval tool
      case 4:// handle mouse pressed for eraser tool
   }  }
  public void mouseReleased(MouseEvent e) {
    switch (drawframe.getCurrentTool()){
      case 0: // handle mouse released for scribble tool
      case 1:// handle mouse released for line tool
      case 2:// handle mouse released for rectangle tool
      case 3:// handle mouse released for oval tool
      case 4:// handle mouse released for eraser tool
   }  }
  public void mouseDragged(MouseEvent e) {
   switch (drawframe.getCurrentTool()){
      case 0: // handle mouse dragged for scribble tool
      case 1:// handle mouse dragged for line tool
     case 2:// handle mouse dragged for rectangle tool
     case 3:// handle mouse dragged for oval tool
     case 4:// handle mouse dragged for eraser tool
   }  }
}
```

[Note: because the program skeleton uses some visual layout generated code, the ALs are not individual separate objects like we have done in our simple Swing examples. Rather there is one main AL, which then demultiplexes the events by looking at the source, and then calls an appropriate AP like method. It is not the same style we would write, but is equivalent.]

Revised Design

So, one goal is to reorganize the code for each drawing algorithm into one place, one object. These objects will be the concrete strategies. We will call them *Tools*. (For the early labs we give you the design and UML, for later labs you will create this.)



We could really probably best call this a *multi-strategy*, since each concrete strategy object will actually have multiple algorithms in it. Interestingly enough some people would label this as a *State Pattern*, because we are changing how we do several things. This is a subtle distinction between the intents of the two patterns, and we will review this more when we study that pattern.

So your refactoring should take all of the parts of an algorithm for one drawing mode and collect them into the tool for that mode. Then do the same for all other modes. You can start by doing just one mode, perhaps the scribble tool.

Note that you don't really have to know anything about how Swing graphics work to do this, it is just refactoring the existing code into a different (better) structure.

Implementation

Now that drawing modes are encapsulated into Tool's, we need a way to switch tools. The Tools are really the *concreteStrategy*'s, so we need to see how we will manage strategies. In the old program strategies were represented by a single integer flag (probably would be better to use an Enum).

In the new version, the current strategy will be [*pause here and answer it yourself*!] VQ ..

an object reference, to a current *Tool*.

Note well the impact of this. All of the long if/case statements sprinkled in the old code, which are a maintenance problem, are gone. They all get replaced by [VQ]…

polymorphism on the *currentTool* object reference.

VQ: If this is the Strategy pattern, where is the *context object*?

Result
Now notice that the definition of any drawing mode is fully localized and encapsulated in one place, that concrete strategy class. Further, there is no more tangle between GUI and drawing algorithm (application logic) code. And, now one can easily add new drawing modes in a purely extensible manner, completely realizing the open/closed principle.

Sub-Strategies
Notice the interesting structure used for the *TwoEndsTool* class. It is based on another refactoring, that all shapes (modes) based on a shape definition by two points have a common structure, and so that commonality should be refactored into a common class, the *TwoEndsShape*.

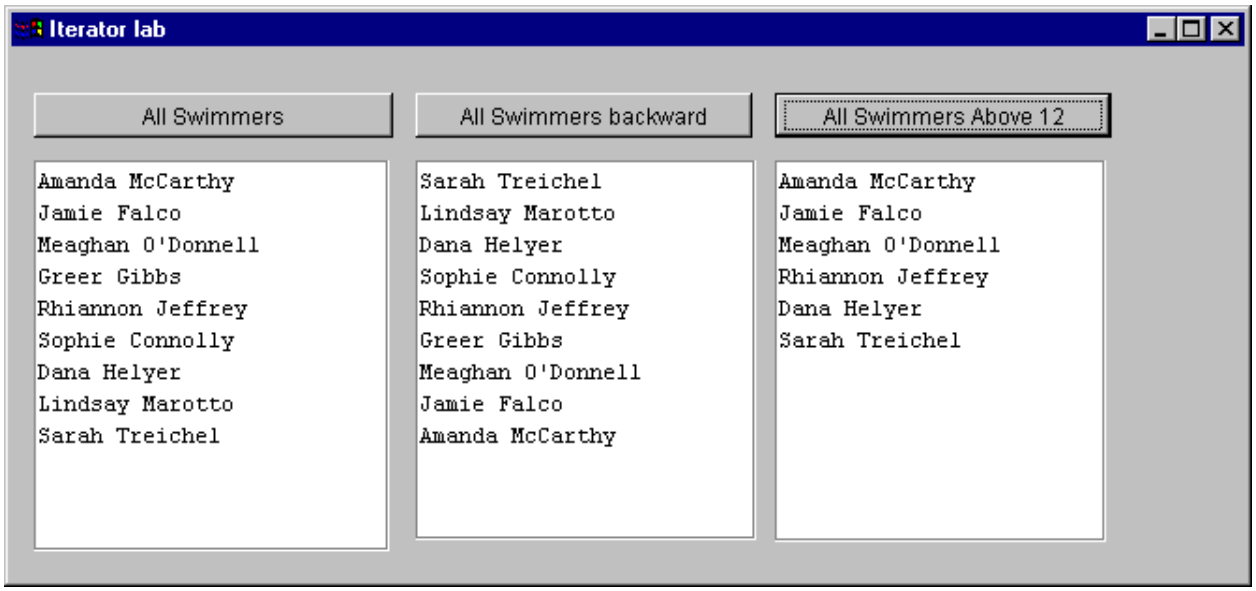
Notice that this class is not derived, but composed from the *TwoEndsTool* class. [Why??]

This makes it basically a strategy which is used by the TET class; a sub-strategy!

[Comment on if/why this is a strategy, or not.]

Iterator Lab

This lab uses several different iteration methods to display three different views of a data set (model).

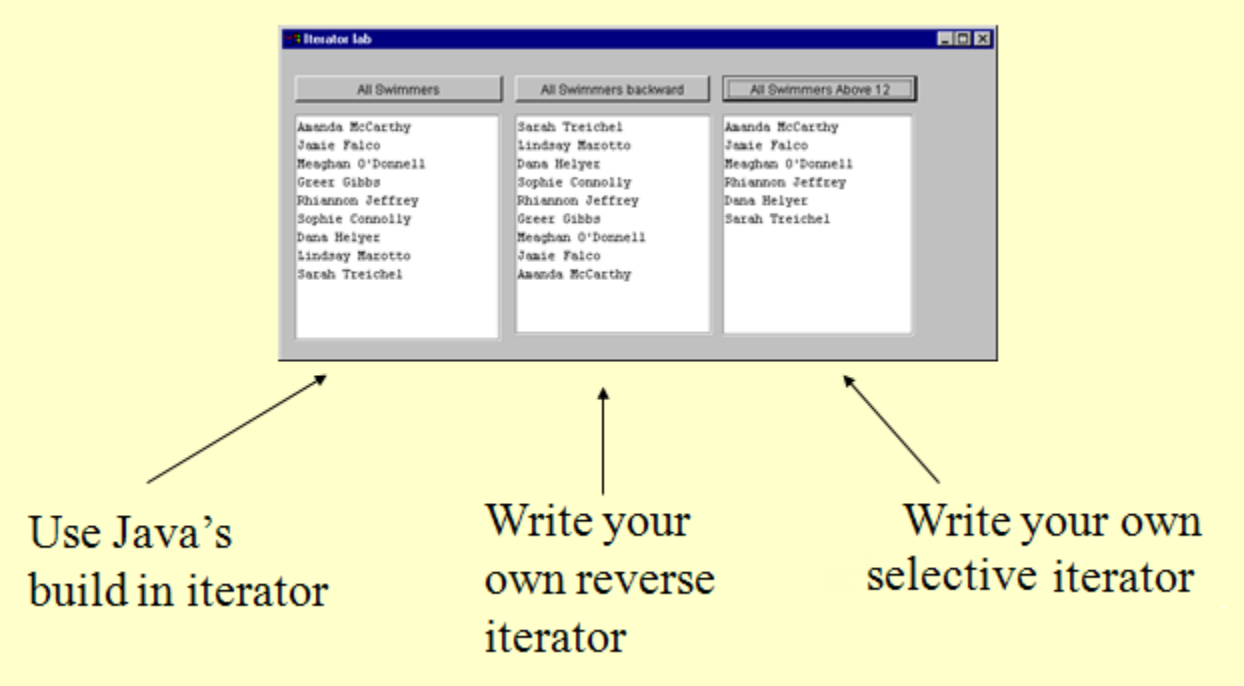


The given code uses some simple loops in the *ActionListeners* to produce the three differing displays.

```
void JButtonAllSwimmers_actionPerformed(java.awt.event.ActionEvent event)
        {       Vector vectorlist = slist.getVector();
                for (int x=0; x<vectorlist.size(); x++){
                   Swimmer swimmer= (Swimmer)vectorlist.elementAt(x);
                   JTextArea1.append(swimmer.getFname()+" "+swimmer.getLname()+ "\n");
                }
        }
        void JButtonAllBackward_actionPerformed(java.awt.event.ActionEvent event)
        {       Vector vectorlist = slist.getVector();
                for (int x=vectorlist.size()-1; x>-1; x--){
                   Swimmer swimmer= (Swimmer)vectorlist.elementAt(x);
                   JTextArea2.append(swimmer.getFname()+" "+swimmer.getLname()+ "\n");
                }
        }
        void JButtonAllAbove12_actionPerformed(java.awt.event.ActionEvent event)
        {       Vector vectorlist = slist.getVector();
                for (int x=0; x<vectorlist.size(); x++){
                   Swimmer swimmer= (Swimmer)vectorlist.elementAt(x);
                   if (swimmer.getAge()>=12)
                     JTextArea3.append(swimmer.getFname()+" "+swimmer.getLname()+ "\n");
                }
        }
```

The problem with this code design is that it mixes the GUI code with the business logic of the details of data and now we want to analyze (present) it. A better design would separate the two, and also encapsulate the details of these special iterations into *iterators*.

Thus, you are to *refactor* this, to use the encapsulation of *Iterators* to do the same result.



1) The lab only requires three simple examples of iterators, not the full generalities we have seen in class.
2) The **first panel** is just a simple iterator, and can use the standard Java *iterator()* method.
3) The **second panel** needs a special iterator, with different (internal) iteration logic. If you were the creator of the collection class, you could directly access it and add some second iterator capability like this directly.

   However, since we are using a standard Java system class for the collection, we cannot do that. Instead we have to create a new Iterator, layered on the old (standard) one that the List<T> interface provides us.
4) This leaves the question of the design for some method to provide us this new iterator.

   We can do this by one of two standard ways; inheritance or composition.

   a) Inheritance would mean that we create some new collection class, perhaps *MyList<T>*, which adds a new method to the standard *List* interface. This new method can be named anything we want, perhaps: *reverseIterator()*.

      VQ: What is the signature of this new method?

      Iterator<T> reverseIterator ();

b) Composition would mean that we create a new class, which provides a wrapper (interface) to the original standard List class object(s). We can choose the same names and interfaces as above, but the implementation is different.

5) So what does this new method do? It has to return an *Iterator*, and it can then use it to access the actual data, and then perform any desired changes before presenting it to the user, through its own Iterator interface.

   For a reverse iterator, it will be necessary to have some underlying iterator that allows one to do absolute indexing into the actual data.

6) For the **third panel**, one needs to take the selection and processing code now found in the AL of the GUI, and encapsulate that into something else.

   We could just make it as a fixed piece of logic in a new *iterator*, a *selectiveIterator*, and that is an acceptable approach for this (simple) lab.

   But a better and more general solution is to put it into a *functor (predicate)*. This is then passed to some method to create a new (*external*) *selective iterator*, which will then return only the elements as desired. Probably the method should follow the standard naming convention, and be: *Iterator<T> iterator (Predicate<T>).*

7) An even better solution might be to create an *internal iterator*, along with a *functor* describing what to do to each element, and it would then apply it to all elements of the collection.

   The *functor* constructor should take some specification of the selection to be used. Then one needs some new method on the collection which will take a functor, and apply it to the collection.

   This new method is also not a part of the standard *List* interface, and thus it would be added in the same manner as above for the reverse iterator.

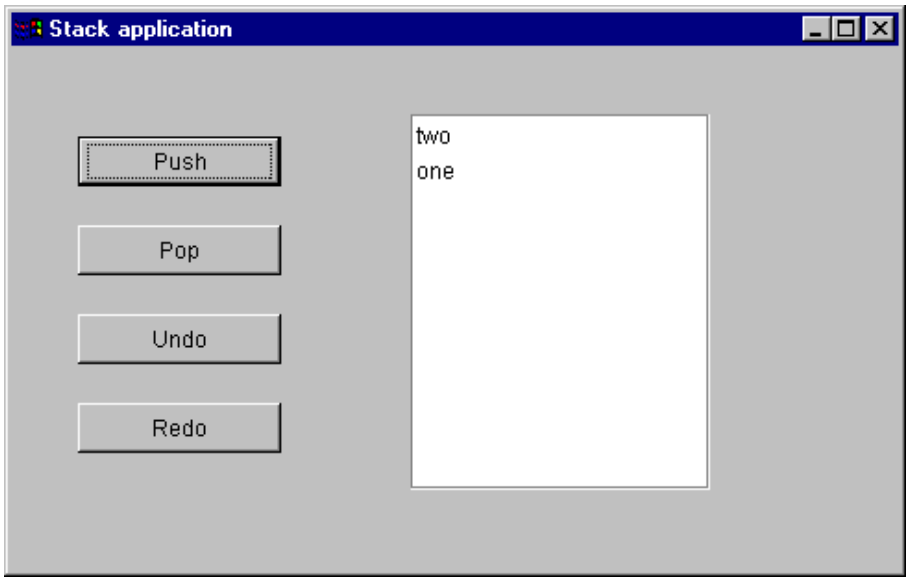
   It is also possible to have this *doAll* method be a member of some other "*third-party*" class, which gets knowledge of the collection and the functor, and then does the application.

8) For the internal iteration version, you need to have the functor do the actual GUI output, so it needs to have access to the output destination. This can be done in the constructor of the functor.

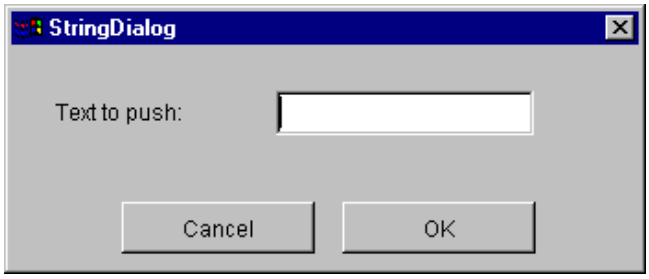
Command Lab:

In this lab you are to create a simple application where you can enter values and then push them onto a stack, or pop values from the stack, and the stack contents are always displayed.

The user interface and GUI code are given;

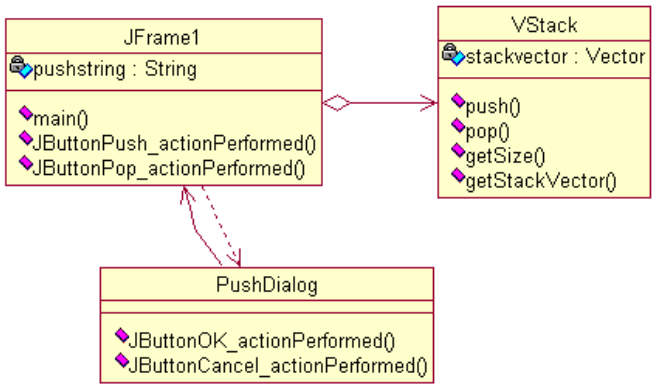


The values to push are given through a simple input dialog box (code also given):



(For simplicity in testing, I just replaced this with some serialized input strings to simplify things.)

The design prior to application of the Command Pattern is very simple;



But the problem with this approach is that every component is coupled with the others, with the standard SE problems. SO our solution is to refactor into a command pattern.

Do the lab in several simple phases;

1) Convert the lab into several components, using the MVC pattern
2) add commands for the operations on the model (stack)
3) move domain logic (stack usage) code from the GUI AL/AP code into commands.
4) Add undo/redo capability.

   This means that the invoker gets more complex, and maintains a command history, and can use it to access and re-use previous commands.

Discussion
The main goal of the command pattern is to separate the command creator, the command target (called receiver in GOF), and the command invoker. This decoupling has all of the standard SE benefits; modularity, encapsulation, re-use, etc.

One criterion for a correct design is that there is no reference to any of these components for the other, except through the *Command* interface. Note that when doing the design for any application, you need to map the general GOF terms used in the pattern description onto meaningful application domain terms. Thus if the word push or pop (as an action on the target stack) appears in the *CommandManager*, something is wrong.

VQ : What are the UML class descriptions of *Command Pattern* as applied to this lab?

Command Design
The client program creates commands, and sends to the invoker. How is the relationship between the target (receiver, model) and the commands established? It could be done on every command as a parameter, or when the command(s) are created, or … [other options?].

Command Manager
Another important design aspect is the undo/redo capability. Note that these are not commands, but are *meta-commands*. That is, they do not operate on the target (the model; a VStack), but rather operate on the invoker (Command Manager). Each *ConcreteCommand* class should contain (encapsulate) the full knowledge of that action, including both its operatoin (execute()), and its reversal (undo()). It would be a bad design to have some external agent (e.g. the command manager) try to know what the proper undo was for any specific command.

Command state
Another main design issue for all commands is that they may (often) require some state. Some will require it for execution, others for the ability to perform an *undo()*. In any case, make sure that all commands are self-contained, with both an *execute()* and *undo()* method.

You will have to consider if the push command has any initial state, specifically:

VQ what is the relationship of the get-data dialog and the push command?

One good way to test this decoupling is by utilizing its Open/Closed property. Consider adding a new command; clear, and make sure that you would not have to change any other existing command, nor the command manager.

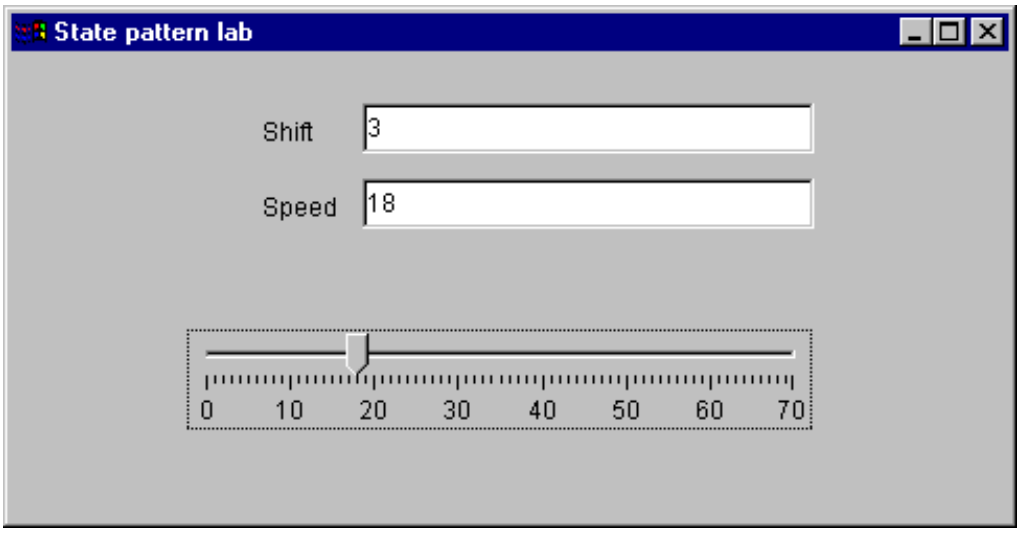
Additional Steps:
- Add a new capability (and GUI button) to clear the stack

- [Harder]
  add a second stack, and a button to switch stacks. This means deciding how to associate the (current) target with commands.

State Lab:

This is a very simple lab, but embodies a good example of the State pattern.

It is to represent a car, which has a set of gears that it uses to drive at various speeds. Each gear can only support a specific speed range, and outside of that a gear change is needed. The control is via a simple GUI;



Changes in the slider create an event which arrives at the ActionListener of the GUI. The actual logic for each state is very simple, basically the series of nested if statements represents the various states. The gear Ranges are:

speed	*Gear*
x=0	park
0<x<5	1
5<x<10	2
10<x<30	3
30<x<55	4
x>55	5

Be sure that you have a model which represents the real world; that means you should have a car, transmission, and a set of gears. The original code that you get has one big if/case logic with all the states in it implicitly;

```
public class Car{
   public int changeSpeed(int speed){
      if (speed == 0) {
            return 0;
         } else {
            if (speed > 0 && speed < 5) {
               return 1;
            } else {
               if (speed > 5 && speed < 10) {
                  return 2;
```

```
                } else {
                    if (speed > 10 && speed < 30) {
                        return 3;
                    } else {
                        if (speed > 30 && speed < 55) {
                            return 4;
                        } else {
                            if (speed > 55 ) {
                                return 5;
                            }
                        }
                    }
                }
            }
        }
        return 0;
    }
}
```

VQ: What's the problem with this?

The basic idea of course is to replace the single complex if statement with polymorphism over the set of concrete states. You also need to consider the context object, which manages the states.

Because the current state is maintained in the context, you need to decide who/how/where the state transitions are determined and managed. We have discussed that in general the state trajectory should be encoded either,

- as a state table (or equivalent) in the context,
- or as a series of state transitions in the actual concrete states.

The second of these is often the most appropriate and simple. In any case it should be the concrete states which determine the transition, either by explicitly returning the new state, or by returning some information to the context by which it will determine the transition.

This implies that there is some link between the context and state. As always, you need to then determine how to best establish /use this relationship. Should the context pass a reference to itself to the state when it invokes the *action* method, by which the state can push the new state back to it? Or should the state actually return a next-state? (There could of course be other methods also.)

State Trajectory
Also note that you must respect the state trajectory, that only certain state transitions are allowed. This means that if you are in perhaps 1st gear, and it would require 5th gear to obtain the required speed, you need to have a series of shifts $1 \rightarrow 2 \rightarrow \ldots \rightarrow 5$, you cannot directly jump from $1 \rightarrow 5$.

Whatever scheme you use for encoding state transitions, they must be in this sequential incremental manner.

GUI Update
Also notice that you need to have some way to update the GUI when a state changes. This is to be able to display the current gear (state). There are several ways to do this, but in any/every case you will have to consider how to have a link between the state(s) and the GUI for the update.

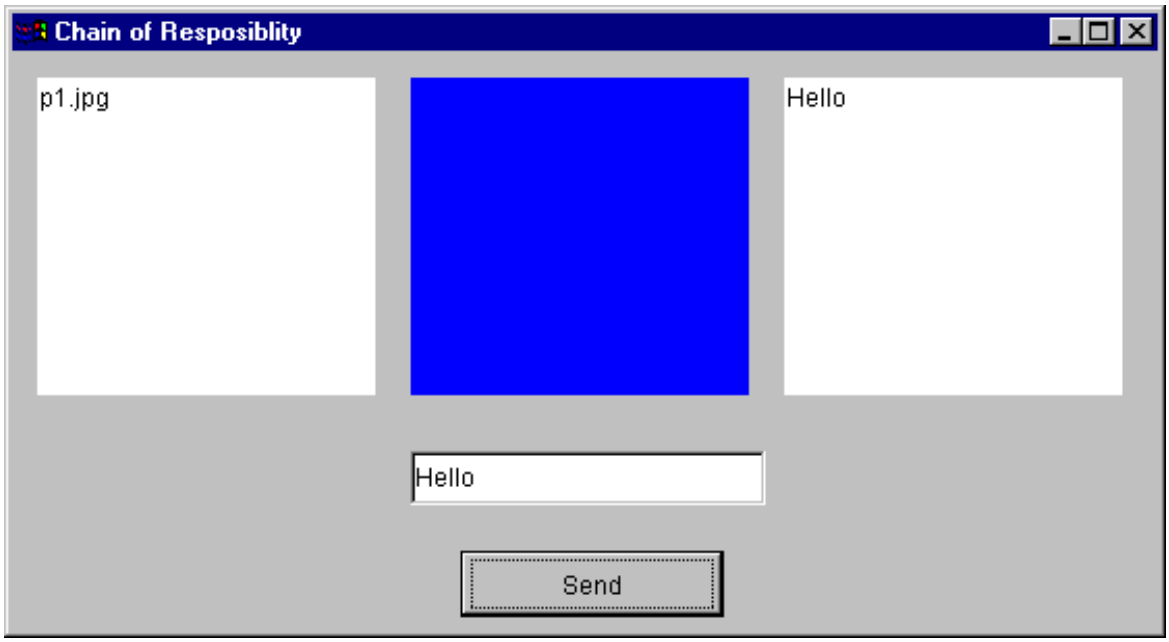
VQ: How could one have an update relationship without an explicit coupling?

State Objects

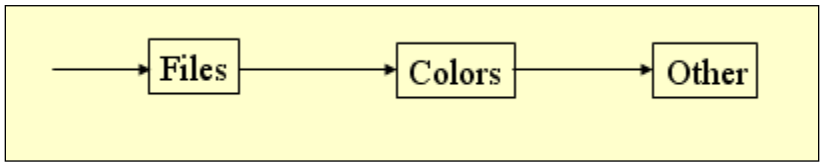
When and where should state objects be created? Should they be re-used? How, and who should do this? [We will later see how the *singleton factory* can solve this nicely.]

COR Lab:

This is another simple lab. Here you have several independent viewing components, and you need to be able to compose them into a single unified viewer, which is extensible, and yet keep each of the components modular. The viewer is then dynamically extensible.



The user enters a color, and sends it to the viewer. The viewer then has a series of view components each of which knows (only) how to display a specific type of entry. The current viewers are:



The file viewer will display the name of a file if the request matches that template, the colors displays a color if the request is recognized as that, else the final viewer just displays the request as text.

Note that the components all answer a common protocol, and that each provides semantically similar but disjoint service. That means that they all do the same thing, just differently, and that only one will actually perform the requested task.

Also note that there is an implicit ordering in the chain elements;

VQ: What is it, and why?

This is because the candidate processors are given a chance to do the work in a sequential order, and thus the ordering on the chain must be from most specific to most general.

You should first give the UML of your design, and then implement it.

Issues:

- Who does the chain management?
- What happens if no processor in the chain can process the request?

Visitor Lab:

This is another simple lab. Here we have a very simple collection that has a variety of data stored in it. It is a very old-style approach, using Object as the generic (void) type to allow this.

The goal is then to collect the data from each node (item) in the list, and ass the numbers to find the total.

The sample program does this by doing a type-test on each item, and then converting it to an addable number. You can consider this code to do the conversion(s) the adaptors for each element type.

There is an alternate version included as well in the lab description, where each disparate data type is is wrapped in a class, but the general result is the same.

You are to simplify this, by having a visitor which can contain all the required conversion codes for each type, and then can visit each node in the data structure (list), and collect the numbers into a sum.

Then evaluate the value of this refactoring; why /how is the visitor version better than the old version(s).

Issues:

- Iteration of the visitor?

Bridge Lab:

This lab is basically a version of the example given in GOF for the bridge pattern. .

Let's say I have been given the task of writing a program that will draw rectangles with either of two drawing programs. I've been told that when I instantiate a rectangle, I'll know whether I should use drawing program 1 (**DP1**) or drawing program 2 (**DP2**).

My rectangles are defined as 2 pairs of points (the bottom left and upper right). This is represented in Figure 9-1 and the differences between the drawing programs are summarized in Table 9-1.

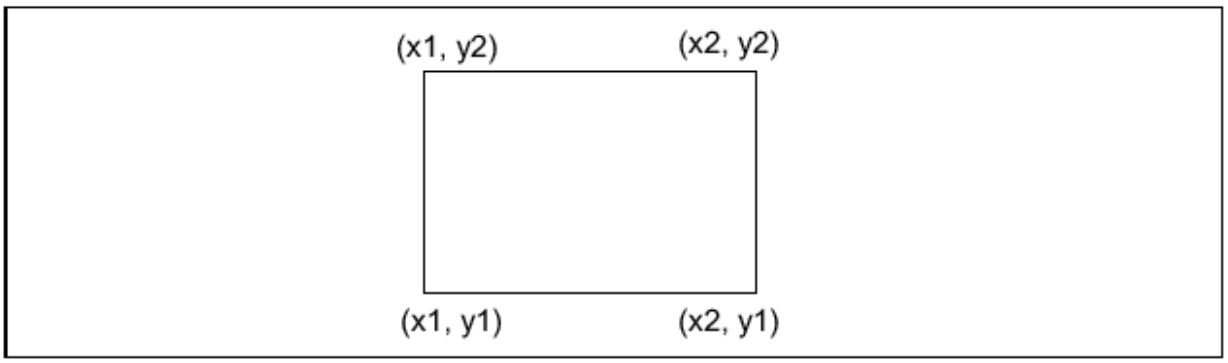


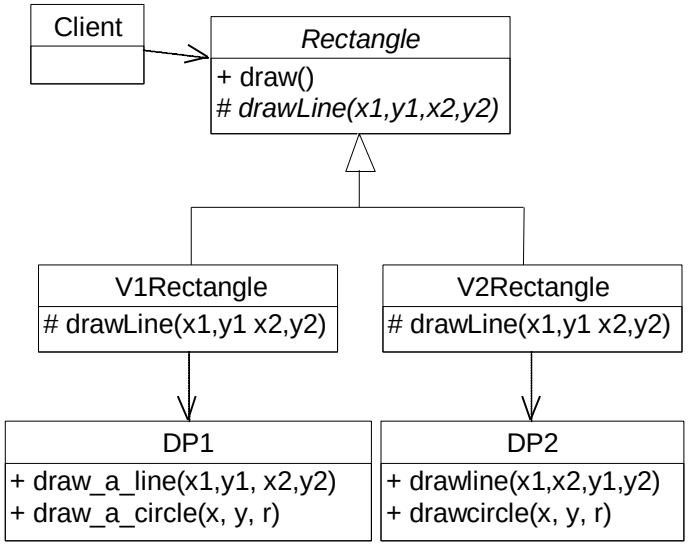
Figure 9-1. Positioning the rectangle

Each library has a different API (interface), although they each provide similar and adequate functinality.

	DP1	DP2
used to draw a line	draw_a_line(x1, y1, x2, y2)	drawline(x1, x2, y1, y2)
used to draw a circle	draw_a_circle(x, y, r)	drawcircle(x, y, r)

Table 9-1. Different drawing programs

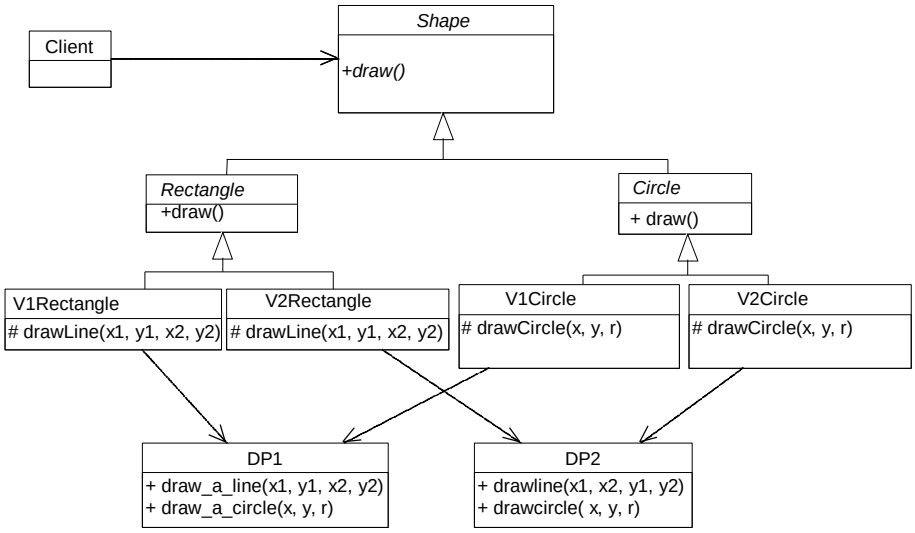
So, we can make two drawing classes,



Then, our usage code might look like this:

```
class Client {
     public static void main (String argv[]) {
          Rectangle s1, s2;
          s1= new V1Rectangle( (double)1,(double)1,(double)2,(double)2);
          s2= new V2Rectangle( (double)1,(double)1,(double)3,(double)3);
          …
          s1.draw();
          s2.draw();
            }
}
```

This seems OK, but then suppose that we now get new requirements, to also be able to add another shape, a Circle. If we try to handle it in the same design approach, things start to get messy;



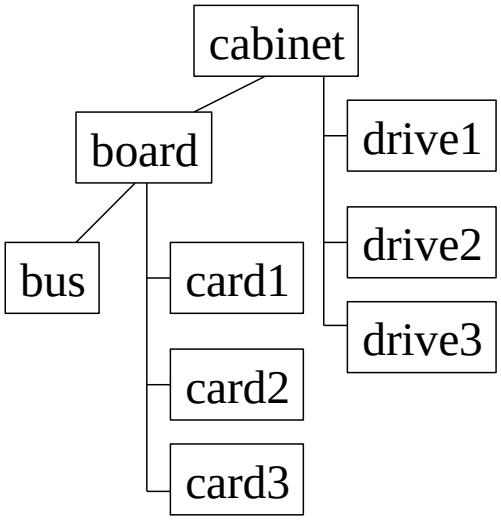
We clearly have the N*M class proliferation described in class. The solution is (of course) the bridge pattern.

Design Issues:

Composite Lab:

Apply the Composite Pattern to the source code from the Composite Lab. First draw the complete UML class diagram. The given source code creates the following tree structure

Consider a simple hierarchical system assembly;



Use the *composite pattern* to compute the total price of this computer system.(or any computer system)

Your program should use the given classes like cabinet, drive, card, board, etc.

Design Issues:

- Should composite nodes themselves have properties (e.g. price)?
- The details of how you build a computer system are not so important; the main thing is the pattern of the overall structure.
- One might want to have some constraints on the dynamic structure built, e.g. only allow cards in board, only drives in cabinet, only 4 cards per board, etc.

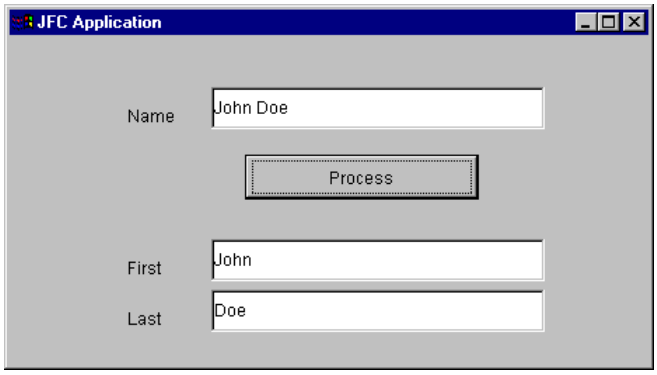
  These types of constraints are not easily handled in a standard composite pattern. Some alternative proposals to accommodate this include a layered composite, where each level would include the appropriate type constraints for membership. Dynamic properties like member count and mix must be handled … dynamically (of course!).

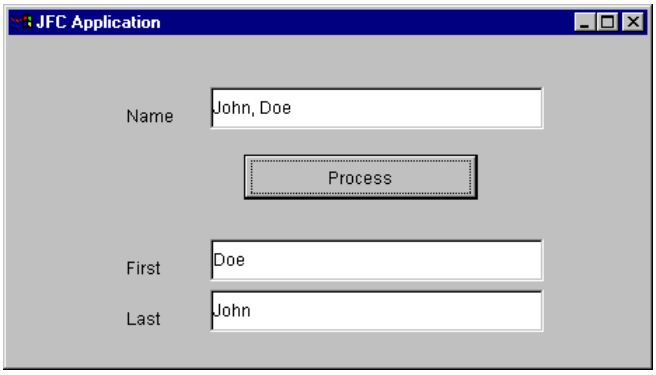
Factory Lab:

The basic idea is that we want to instantiate a concrete tool to parse input names, which may be in two different formats, and thus we need to choose which of two parsers to instantiate to do the parsing.

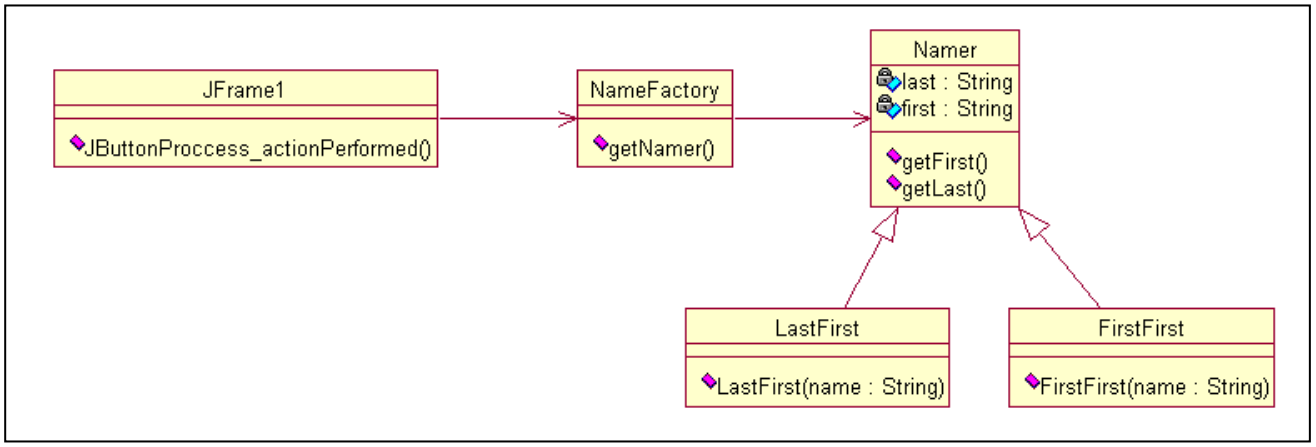
To do this we would look at the first name input, and make a decision based on an analysis of its format, and then instantiate a concrete parser to use, and just use this for all other input data.

The logic and constructors for creating this parser will be our *NameFactory*.





If you type in the name without a comma, then the first word is the first name, and the 2nd word is the last name. If you type in the name with a comma, then the first word is the last name, and the 2nd word is the first name.
Implement this application according the following class diagram:



The implementation of the process button action is as follows:

```
void JButtonProccess_actionPerformed(java.awt.event.ActionEvent event)
{
        String name=JTextFieldName.getText();
        Namer namer=nfactory.getNamer(name);
        JTextFieldFirst.setText(namer.getFirst());
        JTextFieldLast.setText(namer.getLast());
}
```

Depending on the name, the *NameFactory* decides if it creates a *FirstFirst* object or a *LastFirst* object. The methods *getFirst* and *getLast* will then give the correct first and last name.