Do problem A OR problem B as individually assigned. You can refer to the file NavSystem.java for the stubbed-in utility classes referred to below. You must include a README explaining the challenges you found and any design choices you had to make.

**Problem A:** Write a Java implementation of the four components described below. Assume the given library classes and the UIPanel are already done (e.g. use the stubs in the given file NavSystem.java). Your code will not necessarily do anything (unless you fill out some of the stubs) but it should at least compile. (You can make the UIPanel an inner class of the UI, where you can define the otherwise undefined reference routeCalculator shown in the button callback.) Pay careful attention to threading, race conditions, and responsiveness.

**Problem B.** Write a Panini implementation of the four components described. Assume the given library classes and the UIPanel are already done (e.g. use the stubs in the given file NavSystem.java). Remember that Java classes can be imported into Panini capsules and, where relevant, can be defined within Panini capsules. Your code will not necessarily do anything (unless you fill out some of the stubs) but it should at least compile. (Note: in order to compile, you will need to comment out the undefined reference routeCalculator shown in the button callback. The UIPanel class can be defined within a UI capsule. Ideally, it could then refer to a capsule called routeCalculator that is part of the UI capsule's state, but the Panini compiler currently does not allow this.)

**Design problem**

*Based on a true story.*

Consider the following scenario for a simplified navigation system. A sketch of the code can be found on the following pages. The system consists of four components: a route calculator, a maneuver generator, an interface to a GPS unit, and a UI.

In response to user input, the UI requests a new route by invoking a calculateRoute operation on the route calculator, an operation that is assumed to be computationally intensive. When finished, the route is passed to the maneuver generator via a method newRoute. The maneuver generator in turn calls updateRoute in the UI so it can draw the route on the map. The GPS component continually parses the data stream from hardware and periodically updates the maneuver generator with the current position via setNewPosition. The maneuver generator in turn calls updatePosition in the UI so it can update the display; then it checks the position against the current route and generates a new turn instruction for the UI if needed (an operation which we assume is not computationally intensive), calling announceNextTurn whenever there is a new instruction.
We assume that there are three simple classes \textit{Route}, \textit{Position}, and \textit{Instruction}, all of which are immutable.

More precisely we can describe the components as follows:

\textbf{ManeuverGenerator component}

Has a reference to the UI. Invokes the UI's \texttt{updatePosition} operation whenever a new position is received from the GPS and invokes its \texttt{updateRoute} operation whenever a new route is received from the RouteCalculator. Invokes \texttt{announceNextTurn} on the UI whenever a new instruction is obtained from the internal ManeuverUtil class (see below).

There are three public operations.

\begin{verbatim}
void setNewRoute(Route r)
void setNewPosition(Position p)
Position getCurrentPosition()
\end{verbatim}

The logic of \texttt{setNewPosition} is as follows:

\begin{verbatim}
update the UI with the new position
get an instruction from ManeuverUtil
if the instruction has changed from the previous instruction
   update the UI with the new instruction
\end{verbatim}

Internally, the ManeuverGenerator uses the library class ManeuverUtil to produce an Instruction from the current route and current position:

\begin{verbatim}
class ManeuverUtil
{
   // assumed not to be computationally intensive
   public static Instruction createNextInstruction(Route currentRoute,
                                                  Position currentPosition)
   {
      /* details not shown */
   }
}
\end{verbatim}

\textbf{RouteCalculator component}

Has a reference to the ManeuverGenerator; invokes its \texttt{getCurrentPosition} operation during calculation of a route and invokes its \texttt{setNewRoute} operation on completion of a route calculation. Has one public operation:

\begin{verbatim}
void calculateRoute(Position destination)
\end{verbatim}

(the only argument is the destination because it gets the current position from the ManeuverGenerator's \texttt{getCurrentPosition} method). The method returns immediately; on completion of the actual calculation, the maneuver generator is notified.

Internally, this component uses the library class RouteUtil to actually find the route:
class RouteUtil
{
    // This operation may take a long time (many seconds or even minutes)
    public static Route doCalculation(Position src, Position dst)
    {
        /* details not shown */
    }
}

GPS component

Has no public operations. Has a reference to ManeuverGenerator component and periodically invokes its setNewPosition operation. Internally uses the class GPSUtil to get actual data:

class GPSUtil
{
    // returns a new Position from the GPS hardware,
    // blocking if no data available (normally
    // every 3 to 5 seconds)
    public static native Position readData();
}

The logic of the GPU component is pretty simple:

```
while (true)
    get a position from readData()
    update the ManeuverGenerator with the new position
```

The UI component

The UI is a typical Swing-based UI with a display including a map with the current route and current position highlighted. It has a reference to the RouteCalculator and may invoke its calculateRoute method with a desired destination based on user input.

The UI has three public operations:

```java
void updatePosition(Position p);
void updateRoute(Route r);
void announceNextTurn(Instruction inst);
```

Internally, the UI uses an instance of the UIPanel component, a sketch of which is shown below.

class UIPanel extends JPanel {

    // only relevant fragments are shown...

    private Position currentPosition;
    private Position currentDestination;
    private Route currentRoute;

    private JPanel mapDisplay;
    private JButton confirmDestinationButton;
    private JTextArea nextTurn;

    void updatePositionInternal(Position p) {
        currentPosition = p;
    }
}
void updateRouteInternal(Route r) {
    currentRoute = r;
    repaint();
}

void announceNextTurnInternal(Instruction inst) {
    nextTurn.setText(inst.toString());
    repaint();
}

static void createAndShow()
{
    // create window and start UI machinery
}

// listener is attached to confirmDestinationButton...
private class DestinationButtonListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        routeCalculator.calculateRoute(currentDestination);
    }
}

Note

You may notice a potential problem: if the route calculation takes a long time, and the vehicle is moving, maybe it is already off the route by the time the route is calculated. We will ignore this problem.