Assignment #2: Graphics, and GUIs

For this assignment you are to extend your game from Assignment #1 (A1) to incorporate several design patterns, and a Graphical User Interface (GUI) with graphical game output. Most of the code from A1 will be reused, although it will require some modification and reorganization. An important goal for this assignment will be to reorganize your code so that it follows the Model-View-Controller (MVC) architecture. If you followed the structure specified in A1, you should already have a "controller": the `Game` class containing the `play()` method. The `GameWorld` class becomes the data model, containing the collection of game objects and other game state information. You are also required to add two classes acting as views: a score view which will be graphical, and a map view which will also give us our first graphical output of the game. In A3 we will replace this with an animated adaptation and sound.

Single-character commands entered via a text field in A1 will be replaced by GUI components such as side menu items, buttons, and key bindings. Each such component will have an associated *command* object, and the command objects will perform the same operations as previously performed by the single-character commands.

The program must use appropriate interfaces and built-in classes for organizing the required design patterns. The following design patterns are to be implemented in this assignment:

1. Command – to encapsulate the various commands the player can invoke,
2. Singleton – to ensure that only a single instance of player helicopter can exist.
3. Strategy – to control movement for additional (non-player) helicopters.

Model Organization

`GameWorld` is to be reorganized so that it has a `GameObjectCollection`**.** All game objects are to be contained in this single collection. Any routine that needs to process the objects in the collection must access the objects through the collection. The model contains the same game state data as A1 (current clock time and lives remaining). Some of you may have already coded the game objects in this way, hence, you may not need to make many (or any) changes.

Views

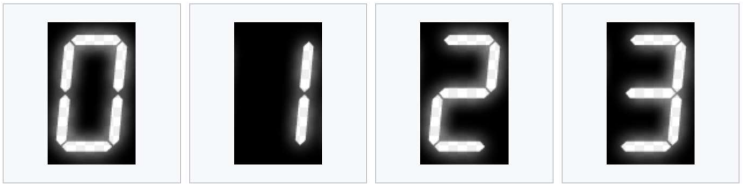
A1 contained two functions to output game information: the 'm' key for outputting a "map" of the game objects in the `GameWorld`, and the 'd' key for outputting current game/player- helicopter state data (i.e., the number of lives left, the current clock value, last skyscraper number the player's helicopter has reached sequentially so far, the player helicopter's current fuel level, and player helicopter's current damage level). Each of these two operations is to be implemented in this assignment as a view of the `GameWorld` model. To do that, you will need to implement two new classes: a `MapView` class containing code to output the map, and a *`GlassCockpit`* class containing code to output the current game/player-helicopter state information. The `GlassCockpit` class represents a view of the of the game state with a vintage *digital dashboard* style.

Copyright 2020,2021, All Rights Reserved, Not for Distribution, For CSUS-CSC133-04/05 Spring 2021 Only

GlassCockpit Details

GlassCockpit, like all views, must extend the CN1 Container class. For each of the non-label elements in the `GlassCockpit`, you must extend the CN1 component class to implement your own components. Each of the numeric displays, e.g., fuel, clock, must employ classic 80s style seven segment displays. To do this you will use an image for each digit and your component will translate input values to display the appropriate digits as images. We will extend this idea in the next project when we implement animation for several of our game objects. You may download a complete set of digits and digits with dots from the Wikimedia commons at the following URL.

https://commons.wikimedia.org/wiki/Seven_segment_display.



These images have transparent digits so that you can set their color by drawing a rectangle behind them. You will build up the `GlassCockpit` using containers and layouts and place your component displays as you would any other CN1 control such as a button or textbox. Your components must maintain the aspect ratio of the digits and cannot stretch then in either direction unless it does so in both directions proportionally. This will require you to override the `CalcPreferredSize` method. Details of this are made clear in the video on implementing a real time clock in CN1.

The image below gives some suggestion on what the GlassCockpit could look like. You are free to change some of the colors and fonts and move things around and add a nice background image. However, you must retain the label above control, you must use components, and you must put these components in an appropriate layout, you must use seven segment displays for the readouts and you must follow the other specific requirements discussed below.

In addition to the above changes note that the game clock displays real time in minutes, seconds, and tenths of seconds. This is required. You are also required to display the tenth second in a slightly darker color so that it is less distracting. It is ok, to display the decimal point in the same color, as is shown above. This is because the above set of images offer versions of each digit that display the dot, consequently, the dot is associated with the unit second's digit.

Put some effort into thinking about how you are going to abstract and share code for the various displays. There's enough boiler-plate code that some kind of class hierarchy makes sense, but you don't want to go crazy overgeneralizing if you don't need it for this project.

Copyright 2020,2021, All Rights Reserved, Not for Distribution, For CSUS-CSC133-04/05 Spring 2021 Only

Display Requirements

The real time game clock is not a merely a view on game data, rather, it is a self-animating component and provides elapsed *real* time information to the game world. Yes, this breaks our pure MVC pattern. In any case, `GameClockComponent` must therefore provide methods for `resetResetElapsedTime`(), `startElapsedTime`(), `stopElapsedTime`, and `getElapsedTime`(). The GameClockComponent will run at the full framerate of CN1 and is not dependent on the `UITimer` that is responsible for updating the rest of the game. These methods are exposed to GameWorld via an appropriate interface defined in GlassCockpit. In other words, if we change the interface to the component, we should not have to change our game world. You do not necessarily need to expose the full interface of `GameClockComponent` to `GameWorld`. `GlassCockpit` should expose only the functions that `GameWorld` needs and in the language of the domain of `GameWorld`. For example, if the game never needs the clock to restart from any time other than time zero, then there is no need to separate reset and start as exposed to `GameWorld`. You may still use game time to determine win or loss, as discussed below, but your game state must also report real time.

The `GameClockComponent` must change the background color to red when it reaches ten minutes of playing time. For the sake of playability, you are free to make this a changeable parameter so that it can be easily adjusted for a different time in the future. You must still maintain a darker color for the tenths digit. The fuel indicator must be four digits and can be whatever color you choose. The damage must be two digits and must represent percent of total damage. You do not have to represent damage internally as a two-digit integer, you may use whatever is convenient, but the display must indicate a two-digit percentage. When damage is below 50% you may display it in whatever color you choose, the display turns yellow when damage is above 50% and it switches to red when damage is above 85%. Heading is whatever color you choose but it must be three digits and display the *compass* heading in degrees. Lives and last skyscraper reached must be single digit displays but may be any color you choose. Note that this enforces constraints on the game with respect to the possible number of both parameters.

While you must implement the timer as a self-animated component, the other components in `GlassCockpit` may also be self-animated controls if you wish. For these you do not have to update them at full rate but can choose to return false in the `animate()` method on some iterations to reduce repaint rate. Having the displays update at a slower or faster rate can add to the playability and fun of a game. Real vintage controls may not have updated all at the same rate.

If you choose to implement the remainder of the controls as self-animated components then you have to make sure that your properly register each control within `GlassCockpit` and you have to recognize that the updates are not necessarily synced with changes in the game state. If you don’t choose to do this (they still must extend component) then you have to manage the repainting of your components within `GlassCockpit` whenever you notify it that the game state has changed.

`GlassCockpit` manages all of the components in its view including their construction and initialization. Thus, if we were to change our minds completely about how we display the cockpit information, the rest of the game should not have to be modified.

Copyright 2020,2021, All Rights Reserved, Not for Distribution, For CSUS-CSC133-04/05 Spring 2021 Only

MapView

For this assignment, `MapView` will also display the contents of the game graphically in the container in the middle of the game screen (in addition to displaying it in the text form on the console). When the `MapView` `update()` is invoked, it should call `repaint()` on itself. As described in the lecture, one way to implement this is to have `MapView` override `paint()`, which will be invoked as a result of calling `repaint()`. It is then the duty of `paint()` to iterate through the game objects invoking `draw()` in each object – thus redrawing all the objects in the world in the container. Note that `paint()` must have access to the `GameWorld`. That means that the reference to the `GameWorld` must be saved when `MapView` is constructed, or alternatively the `update()` method must save it prior to calling `repaint()`. Note that the modified `MapView` class communicates with the rest of the program exactly as it did previously (e.g., it is an observer of `GameWorld`).

As indicated in A#1, each type of game object has a different shape which can be bounded by a square. The size attribute provides the length of this bounding square. The different graphical representation of each game object is as follows:

For this project you will create your own graphics assets as images or composite images. You may use any software you like to create assets and you may not use assets from the web. Please do not complain to me that you are not an artist. Neither am I and I have managed to use very amateur programs, including PowerPoint, to create such elements. You may use any style you like, including "1st grade hand drawn" if that suits your style.

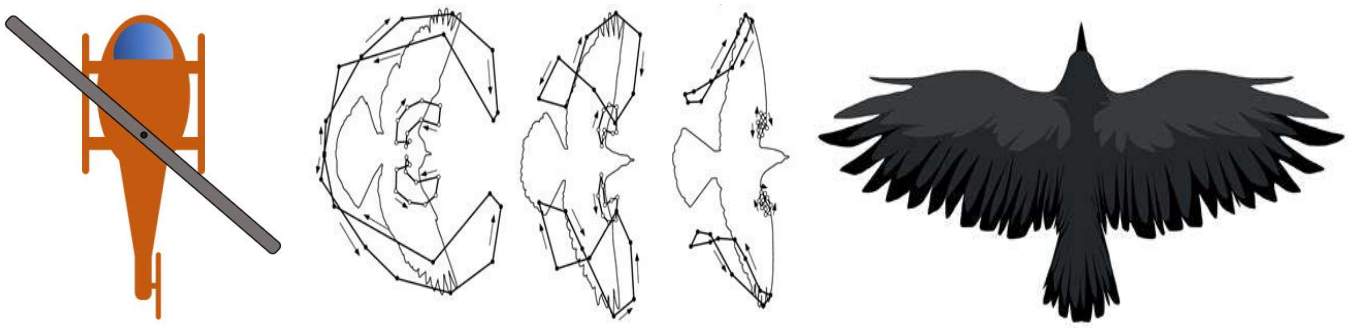
Skyscrapers are filled squares with their number printed in the center. Use your graphics software to create 10 of these with digits 0-9. It doesn't take much to make them look like the top of a skyscraper with a helipad. A gray rectangle with a 3d border, a circle drawn with a thick line, an H. Be creative, make your game your own.

Refueling blimps can be made reasonably realistic by just using concentric ellipses to approximate a blimp's outline and ribs along their length. As long as what you draw looks something like a blimp, you'll be fine. Blimps fit into a rectangle and for the purposes of this project, the rectangle will be the collision region. So, for maximum realism, you want the blimps to occupy much of the rectangle.

The player helicopter occupies a circle. Now, we may put some effort into making this look good, and if you're up to the challenge you can work on that now. However, if that seems daunting at the moment then you can just use a top-down outline of a simple helicopter. Think carefully about how to align the image of the helicopter so that as you change it's heading it changes as expected. I would suggest that you start by just using CN1 to draw a circle and then represent heading with a line of length equal to the radius of the circle drawn from the center in the direction of the helicopter's current heading. Once you have this working, then try replacing the circle with your image that rotates as expected. This can be challenging and one alternative that may work better is to use a film-strip of images such that heading module some value, e.g., 5 degrees, selects one of the images from the filmstrip. Think about what the rotor would look like moving, can you fake that with something simple? Don't worry, for anything that you can't animate this time around, we will revisit that in project 3. If you have static graphics at this point then you will be fine.

Copyright 2020,2021, All Rights Reserved, Not for Distribution, For CSUS-CSC133-04/05 Spring 2021 Only

As with Helicopters, you can begin your birds as unfilled circles with a line drawn as with the helicopter to indicate their heading. However, our goal will be to use a minimum of three bird images in a filmstrip to animate the bird's wing flapping. With birds, heading accuracy isn't as critical so we will just rotate the animated images. See the graphic below of how a bird moves it's wings and then think about how you could just cut up the image (taken from the interwebs) on the right to make an animated bird. The helicopter is simply a composite of basic shapes.



You will want to scale your images by size; hence, the size attribute of a blimp indicates the length of the ellipse along its major axis. The size of a skyscraper or bird indicates the length of the of the rectangle or the diameter of the circle, and size of a helicopter indicates the diameter of the circle.

Skyscrapers should include a text showing their number. When reached they should visually change reflecting that they have been reached. You can either do this programmatically, or, you can create two different versions of each numbered skyscraper asset, reached, and unreached. Refueling blimps should include a text showing their capacity. You can use the `Graphics` method `drawString()` to draw the text on Skyscrapers and refueling blimps.

For now, the most appropriate place to put the responsibility for drawing each shape is within each type of game object (that is, to use a polymorphic drawing capability). In future versions we may treat this differently. The program should define a new interface named `IDrawable` specifying a method `draw(Graphics g, Point containerOrigin)`. `GameObject` class should implement this interface and each concrete game object class should then provide code for drawing that particular object using the received Graphics object `g` (which belong to `MapView`) and Point object `containerOrigin`, which is the component location (`MapView's` origin location which is located at its the upper left corner) relative to its parent container's origin (parent of `MapView` is the content pane of the Game form and origin of the parent is also located at its upper left corner). Remember that calling `getX()` and `getY()` methods on `MapView` would return the `MapView` component's location relative to its parent container's origin.

Each object's `draw()` method draws the object in its current color and size, at its current location. Recall that current location of the object is defined relative to the origin of the game world (which corresponds to the origin of the `MapView` in A#2). Hence, do not forget to add `MapView's` origin location to the current location while drawing your game objects.

Also, recall that the location of each object is the position of the center of that object. Each `draw()` method must take this definition into account when drawing an object. Remember that the `draw…()` method of the Graphics class expects to be given the X,Y coordinates of the upper

Copyright 2020,2021, All Rights Reserved, Not for Distribution, For CSUS-CSC133-04/05 Spring 2021 Only

left corner of the shape to be drawn. Thus, a `draw()` method would need to use the location and size attributes of the object to determine where to draw the object so its center coincides with its location (i.e., the X,Y coordinate of the upper left corner of a game object would be at center_location.x – size/2, center_location.y – size/2 relative to the origin of the MapView**,** which is the origin of the game world).

We will not directly implement the Observer pattern as we have only two, non-dynamic, observers of our gameworld. As is commonly done, we will hardcode our simple observer pattern to retain flexibility. To implement this, `GameWorld` should be thought of as observable, with two observers– `MapView` and `GlassCockpit`**.** Rather than go to the trouble of holding a collection of two observers via registration, we will follow the pattern of implementing update in each of these views and require `GameWorld` to call these update methods when appropriate.

When the controller invokes a method in `GameWorld` that causes a change in the world (such as a game object moving, or a new refueling blimp being added to the world) the `GameWorld` notifies its observers that the world has changed. Each observer then automatically produces a new output view of the data it is observing – the game world objects in the case of `MapView`, and a description of the game/player-helicopter state values in the case of `GlassCockpit`. The `MapView` output for this assignment displays both text output on the console showing all game objects which exist in the world as well as their graphical representation on the map. `GlassCockpit`, as described above, also presents a graphical display of the game/player-helicopter state values.

Recall that there are multiple approaches which can be used to implement the Observer pattern. You **may not** use `the java.util Oberservable` class. Since, we are only implementing the pattern in a lightweight sense here since we don't need a registration system. You must follow this guidance and implement only the necessary aspects of the observer pattern manually.

The essence of this requirement is that the game world must update its views when the game state changes. It is up to the views to manage what state that they need to update themselves.

GUI Operations

`Game` class extends `Form` (as in A1) representing the top-level container of the GUI. The form should be divided into three areas: one for status information at the top of the display, one for commands at the bottom of the display, and one for the map.

Note that since user input is via GUI components, flow of control will now be event-driven and there is no longer any need to invoke the `play()` method – once the Game is constructed it simply waits for user-generated input events. Note however that it is still a requirement to have a `AppMain` class as described in A1. Hence, in this assignment, the play() method used in A1 which prompts for single- character commands and reads them from a text field on the form is to be discarded. In its place, commands will be input through menu items, on-screen buttons and key bindings, You should add a title bar with the name of the game to your form and we will use a simple menu for some actions. However, note that games often take over the entire screen, we have to strike a balance between writing a game and completing an exercise.

Copyright 2020,2021, All Rights Reserved, Not for Distribution, For CSUS-CSC133-04/05 Spring 2021 Only

Each command will be invokable via a combination of these mechanisms. You are to create command objects (see below) for each of the commands from A1 (except “d” and “m” which are implemented as views in A2, and “y” and “n” which are replaced by an exit dialog box – see below) and for new commands introduced in A2 (i.e., change strategies, sound, about, and help). You are to attach the objects as commands to various combinations of invokers as shown in the following table (a ‘+’ in a column indicates that the command in that row is to be able to be invoked by the mechanism in the column header):

Command	Key Binding	Menu	Button
accelerate	+		+
brake	+		+
left turn	+		+
right turn	+		+
collide with **n**PH	+		
collide with **s**kyscraper	+		
collide with **e**-blimp	+		
collide with **g**-oony bird	+		
e**x**it	+	+	
change strategies		+	
give about information		+	
give help information		+	

For north and center regions, you should have a GlassCockpit and a MapView object created, respectively. GlassCockpit and MapView classes should extend from Container. MapView will display the running game objects whereas you should add components to the GlassCockpit container to display the game state information as described above. For the south region you will add the play control buttons. They should have images that have arrows point left and right for changing heading and up and down for accelerating and braking respectively. You will place these buttons in a container with appropriate layout in the south region of the game.

Each button is to have an appropriate command object attached to it, so that when a button gets pushed it invokes the `actionPerformed()` method in the corresponding command object. This is the command pattern’s `execute()` method that contains the code to perform the command. Most commands execute exactly the same code as was implemented in A1. One exception is the “collide with skyscraper” command. Previously this command consisted of a number (between 1-9) entered from the text field located on the form which we eliminate in A2. Now, this command is to use the static method `Dialog.show()` to display a dialog box that allows the user to enter the number on a text field located on the dialog box Note that if the user inputs an invalid value; your program should handle this gracefully.

The other command which must operate slightly different in this assignment is the **c** command that indicates that the player’s helicopter collided with NPH. In A1, this command just increased the damage level of the player’s helicopter. Now, the command is to also add damage to one of the NPH: the **c** command code should choose one NPH and increase that NPH’s damage as well. To increase game fairness, you must alternate between NPHs when this command is entered, but it is acceptable to use any algorithm that approximates this over time, for example, choosing an NPH randomly.

Copyright 2020,2021, All Rights Reserved, Not for Distribution, For CSUS-CSC133-04/05 Spring 2021 Only

The key binding input mechanism will use the CN1 key binding concept so that the command keys invoke command objects corresponding to the code previously executed when the single-character commands were entered. Note that using key bindings means that whenever a key is pressed, the program will immediately invoke the corresponding action (without the user pressing the Enter key). If you want, you may also use key bindings to map any of the other command keys from A1, but only the ones listed here are required. The side menu will use a CN1 defined side menu; see the CN1 videos for details on this using the CN1 Toolbar class.

Your GUI will contain a side menu that contains entries for About, Help, and Exit. The About menu item is to display a dialog box using CN1'sbuilt-in Dialog class that gives your name, the course name, and a version number. **You will lose points if your instructor's name appears anywhere in your code or your running project!**

The Exit menu item should invoke the **x** command. Note that in A2 the **x** command should prompt graphically for confirmation and then exit the program using a dialog box. Invoking Help displays a dialog box listing the key commands for the game. Selecting a side menu item that performs a game command (e.g., "Accelerate" menu item) should invoke the same code, when the button of the same name had been pushed (e.g., "Accelerate" button) and/or its related key has been hit (e.g., "a" key). Recall that there is a requirement that commands be implemented using the Command design pattern. This means that there must be *only one* of each type of command object, which in turn means that the menu items, key bindings, and buttons must share their command objects. (We could enforce the rule using the Singleton design pattern, but that is not a requirement in A2; just don't create more than one of any given type of command object).

Although we cover most of the GUI building details in the lectures, there will most likely be some details that you will need to look up using the CN1 documentation (i.e., CN1 JavaDocs and developer guide). It will become increasingly important that you learn to use these resources.

Non-Player Helicopters (NPHs)

The game object hierarchy will be the same as in A1 except that you will add a new kind of movable object called `NonPlayerHelicopter`, which extends `Helicopter`. The game initialization code should create and add to the game world a minimum of three instances of `NonPlayerHelicopter`. Each NPH is to have an initial location near the first skyscraper, but not exactly at the first skyscraper; instead, each NPH should be at least several helicopter lengths away from the first skyscraper. You should adjust this distance, the number of NPHs, NPH speed and damage, described below, to facilitate game play

When an NPH collides with the player's helicopter, the player's helicopter sustains damage just as described in A1 – including that if the helicopter sustains so much damage that it can no longer move the player loses a life, and the world is reinitialized. NPHs also sustain damage when they collide. Consider how much damage they can sustain to facilitate game play.

NPHs will be controlled by separate pieces of code called strategies which can be altered using a new Change Strategies command; this is described under Strategy Pattern, below. Having NPHs in this version of the game introduces an additional case for ending the game such that if a NPH reaches the last skyscraper before the player does, the game will end with the following message displayed on the console: "Game over, a non-player helicopter wins!".

Copyright 2020,2021, All Rights Reserved, Not for Distribution, For CSUS-CSC133-04/05 Spring 2021 Only

Animation Control

The `Game` class will include a timer in this assignment. You should use the `UITimer`, a built-in CN1 class, to drive the animation. In this context, animation means the movement of movable objects, in the next assignment we will animate the birds and helicopters independently of object movement. `Game` should also implement `Runnable` which is a standard Java interface. Each tick generated by the timer invokes `run()` method which in turn invokes `GameWorld's tick()` method from the previous assignment. This *animates* all moveable objects.

There are some changes in the way the Tick method works for this assignment. In order for the animation to look smooth, the timer itself will have to tick at a fairly fast rate (about every 20 msec or so). In order for each movable object to know how far it should move, each timer tick should pass an "elapsed time" value to the `move()` method. The `move()` method should use this elapsed time value when it computes a new location. For simplicity, you can simply pass the value of the timer event rate (e.g., 20 msec), rather than computing a true elapsed time. However, it is a requirement that each `move()` computes movement based on the value of the elapsed time parameter passed in, not by assuming a hard-coded time value within the `move()` method itself. You should experiment to determine appropriate movement values (e.g., in A#1, we have specified the initial speed of the bird to be a random value between 5 and 10, you may need to adjust this range to make your birds have reasonable speed which is not to too fast or too slow). In addition, be aware that methods of the built-in CN1 `Math` class that you will use in `move()` method (e.g., `Math.cos(), Math.sin()`) expects the angles to be provided in radians not degrees. You can use `Math.toRadians()`to convert degrees to radians. Likewise, the built-in `MathUtil.atan()` method that you might have used in the strategy classes also produces an angle in `radians`. You can use `Math.toDegrees()`to convert degrees to radians.

Remember that a `UITimer` starts as soon as its `schedule()` method is called. To stop a `UITimer`call its `cancel()`method. To re-start it call the `schedule()`method again.

Command Design Pattern

The approach you must use for implementing command classes is to have each command extend the CN1 built-in `Command` class (which implements the `ActionListener` interface). Code to perform the command operation then goes in the command's `actionPerformed()` method. Hence, `actionPerformed()` method of each command class that performs an operation invoked by a single-character command in A1, should call the appropriate method in the `GameWorld` that you have implemented in A1 when related single-character command is entered from the text field (e.g., accelerate command's `actionPerformed()` would call `accelerate()` method in `GameWorld`). Hence, most command objects need to have the `GameWorld` as its target since they need to access the methods defined in this class. You could implement the specification of a command's target either by passing the target (reference to `GameWorld`) to the command constructor, or by including a "`setTarget()`" method in the command.

The `Button` class can contain command objects; `Button` has a `setCommand()` method which injects a command object into the button. `Command` automatically a listener when added to a `Button` via `setCommand()` and it is not necessary to also call `addActionListener()`. The specified `Command` is automatically invoked when the button is pressed, so if you use the CN1 facilities correctly then this particular observer/observable relationship is taken care of automatically.

Copyright 2020,2021, All Rights Reserved, Not for Distribution, For CSUS-CSC133-04/05 Spring 2021 Only

The `Game` constructor should create a single instance of each command object, e.g., Brake, Accelerate, and insert the command objects into the command holders such as buttons, side menu items, title bar area items. Use the appropriate method for each object: `setCommand()` for buttons, `addCommandToSideMenu()` for side menu items, and `addCommandToRightBar()` for title bar area items. You should also bind these command objects to the keys using the `addKeyListener()` method of `Form`. You must call `super("command_name")` in the constructors of command objects by providing appropriate command names. These command names will be used to override the labels of buttons and/or to provide the labels of side menu items or title bar area item(s).

Note that commands can be invoked using multiple mechanisms, e.g., from a keystroke and from a button; it is a requirement that only one command object be implemented for each command and that the same command object be invoked from each different command holder. As a result, it is important to make sure that nothing in any command object contains knowledge of how, e.g., through which mechanism, the command was invoked.

Strategy Design Pattern

NPHs move around the world according to their current strategy; this strategy is used to determine the stick-angle and speed and it can be changed on the fly, i.e., while the program is running. You must implement at least three different NPH movement strategies: the first strategy causes the NPH to move directly toward the next skyscraper and the second strategy causes the NPH to update its heading every time it is told to move so that the heading points toward the location of the player's helicopter; in other words, its strategy is to play "Attack" by trying to collide into the player's helicopter. The third strategy must be one of your choosing and this is a good opportunity to demonstrate creativity and ingenuity for more points. For example, you might have a strategy where an NPH simply moves in circles, or a strategy where it moves back-and-forth between two skyscrapers. These are simple examples, however, and you should improve on these if your goal is to demonstrate creativity and ingenuity. Think about how delivery helicopters might actually operate, what would be their goal? What would make for interesting game play? You may also implement additional strategies if you like, although this is not required.

The program must use the Strategy design pattern to define the strategy for each NPH. You may use an `IStrategy` interface rather than an abstract strategy super class if you think that it's more appropriate. Think about the choices in this tradeoff and make the right choice for your game. Defend your choice in your writeup. You should use two methods called `setStrategy()` and `invokeStrategy()` and a field for saving the current strategy to the NPH class. When an NPH is created it should be assigned a strategy chosen from the available strategies.It is a requirement that NPHs may not all have the same initial strategy; other than that you may assign initial strategies however you choose. Like all game objects, NPHs should include a `toString()` method; the NPH `toString()`should return a string which includes the NPH's information provided for a player helicopter plus an identification of that NPH's current strategy.

When the *switch strategy* command is invoked, the game should cause all NPHs to switch to a different movement strategy. Switching strategies is to be done by invoking the NPH's `setStrategy()` method. You may choose the algorithm that determines the new NPH strategy,

Copyright 2020,2021, All Rights Reserved, Not for Distribution, For CSUS-CSC133-04/05 Spring 2021 Only

as long as every NPH acquires a new strategy each time the “switch strategy” command is invoked. Additionally, you should arrange that, as a side effect of switching strategies, each NPH gets the next “last skyscraper reached” value (otherwise, NPHs will never move beyond Skyscraper #2 since we have no command analogous to “collide with skyscraper” which applies to NPHs). Also, you should arrange so that NPHs never run out of fuel (e.g., NPHs do not need to collide with fuel, for now, if the NPH’s fuel level is getting low, you should set it to a reasonable value).

Game Modes

In order for us to support the flow of control changes with dialog boxes, we are going to add an additional capability to the game. The game is to have two modes: “*play*” and “*pause*”. The normal game play with animation as implemented above is “play” mode. In “pause” mode, animation stops – the game objects don’t move and the clock stops. It is not necessary to add a pause button or key command for this lab. You only need pause the game when a dialog box would otherwise prevent normal gameplay. However, you may find this helpful for testing.

Commands should be enabled only when their functionality is appropriate. For example, commands that involve playing the game should be disabled while in pause mode. Note that disabling a command should disable it for all invokers (buttons, keys, *and* menu items). Note also that a disabled button or menu item should still be *visible*; it just cannot be enabled. This is indicated by changing the appearance of the button or menu item. To disable a button or a menu item added by `addComponentToSideMenu()`use the `setEnabled()` method of the button. To disable a menu item added by `addCommandToSideMenu()` use the `setEnabled()` method of `Command`. To disable a key, use `removeKeyListener()` method of `Form`; remember to re-add the key listener when the command is enabled. You can set disabled style of a button using `getDisabledStyle().set...()` methods on the `Button`.

Copyright 2020,2021, All Rights Reserved, Not for Distribution, For CSUS-CSC133-04/05 Spring 2021 Only

Additional Notes

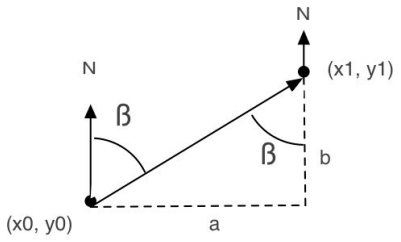
1. It is a requirement that the program contain only a single instance of player helicopter. This requirement must be enforced via the Singleton design pattern. You must `BorderLayout` as the layout manager of your form and use appropriate layouts to meet the requirements elsewhere.

2. You can change the size of your buttons/labels using `setPadding()` method of `Style` class. Each You can also use `setPadding()` on left and right control containers to start adding buttons at positions which are certain pixels below their upper borders. In A2, you must assign the size of your game world by querying the size of your `MapView` container, instead of assigning your width and height to 1024x768 as in A1, assign them to width and height of `MapView`) using `getWidth()` and `getHeight()` method of `Component` as discussed in the video lectures.

3. Note that all game data manipulation by a command is accomplished by the command object invoking appropriate methods in the `GameWorld` (the model). Note also that every change to the `GameWorld` will invoke both the `MapView` and `GlassCockpit` observers – and hence generate the output formerly associated with the "m" and "d" commands. This means you do not need the "d" or "m" commands; the output formerly produced by those commands is now generated by the observer/observable process.

4. Since almost all commands change the `GameWorld` (i.e., all command but exit, about, and help), they produce updated views as side effects. For instance, since the 'tick' command causes opponents to move, every 'tick' will result in a new map view being output (because each tick changes the model). Note however that it is not the responsibility of the 'tick' command code to produce this output; it is a side effect of the observable/observer pattern. You should verify that your program correctly produces updated views automatically whenever it should.

5. You may find the following formula useful while implementing the NPH strategies:



ß = arc tan (a , b)
(image from a tutorial at http://www.invasivecode.com/)

Here, (x0,y0) is the current location of the NPH, (x1,y1) is where it wants to go (target), then B is the ideal compass angle (90 – ideal_heading) for the NPH so it can go towards its target. Based on this information NPH should change its stick angle accordingly in order to approach this ideal heading. To calculate arc tan you can use `atan()` method of CN1 `MathUtil` class which generates angle in "radians". If you need, you can use `MathUtil.toDegrees()` to convert radians to degrees.

Copyright 2020,2021, All Rights Reserved, Not for Distribution, For CSUS-CSC133-04/05 Spring 2021 Only

6. Please make sure that you add your `MapView` object directly to the center of the form. Adding an additional `Container` object to the center and then adding a `MapView` object onto it would cause incorrect results when a default layout is used for this center container (e.g., you cannot see any of the game objects being drawn in the center of the form).

7. To draw a filled circle with radius `r` at `(x,y)`use `fillArc(x, y, 2*r, 2*r, 0,360).`

8. The origin of the game world (which corresponds to the origin of the `MapView` for now) is considered to be in its *lower left* corner. Hence, the Y coordinate of the game world grows *upward* (Y values increase *upward*). However, origin of the `MapView` container is at its upper left corner and Y coordinate of the `MapView` grows *downward*. So when a game object moves north (e.g., it's heading is 0 and hence, its Y values is increasing) in the game world, they would move up in the game world. However, due to the coordinate systems mismatch mentioned above, heading up in the game world will result in moving down on the `MapView` (screen). Hence your game will be displayed as upside down on the screen. This also means that when helicopters turn left they will go west in game world, but they will go east on the `MapView` (and turning right means going west on the `MapView`). In addition, triangles (e.g. skyscrapers and birds) will be drawn upside down in `MapView`. Leave it like this we will fix it in the next assignment.

9. The simple shape representations for game objects will produce some slightly weird visual effects in `MapView`. For example, squares or triangles will always be aligned with the X/Y axes even if the object being represented is moving at an angle relative to the axes. This is acceptable for now; we will see how to fix it in the next assignment.

10. You may adjust the size of game objects for playability if you wish; just don't make them so large (or small) that the game becomes unwieldy.

11. Boundaries of the game world are equal to dimensions of `MapView`. Hence, birds should consider the width and the height of `MapView` when they move, not to be out of boundaries.

12. You should tweak the parameters of your program for playability after you get the animation working. Things that impact playability include object sizes and speeds, etc. Your game is expected to operate in a reasonably-playable manner.

13. As before, you may not use a "GUI Builder" for this assignment.

14. All functionality from the previous assignment must be retained unless it is explicitly changed or deleted by the requirements for this assignment.

Deliverables

See the Canvas assignment for submission details. ***All submitted work must be strictly your own!***

Copyright 2020,2021, All Rights Reserved, Not for Distribution, For CSUS-CSC133-04/05 Spring 2021 Only