**BUTTON LIGHT**

/\*\*

\* Controls a light behind any button type in an elevator system.

\* @author MattCasiro

\* Created: 23 May 2016

\*/

**public class ButtonLight**

{

// true = button light is on

// false = button light is off

**private boolean isLit;**

/\*\*

\* Instantiate a button light in the OFF state.<br>

\*

\* PRE: A button exists for the light to operate with<br>

\* POST: The light for the button was created in the OFF state<br>

\* Cleanup: N/A<br>

\*/

**public ButtonLight()**

{

isLit = false;

}

/\*\*

\* Change the button light state to ON.<br>

\*

\* PRE: The button is in the OFF state<br>

\* POST: The button was set to ON<br>

\* Cleanup: N/A

\*

\*/

**public void turnOn()**

{

isLit = true;

}

/\*\*

\* Change the button light state to OFF.<br>

\*

\* PRE: The button is in the ON state<br>

\* POST: The button was set to OFF<br>

\* Cleanup: N/A<br>

\*/

**public void turnOff()**

{

isLit = false;

}

/\*\*

\* Get the state of the button light.<br>

\*

\* PRE: N/A<br>

\* POST: N/A<br>

\* Cleanup: N/A<br>

\*

\* @return true if the button light is ON, false if the button light is OFF<br>

\*/

**public boolean getState()**

{

return isLit;

}

}

**CALL BUTTON**

/\*\*

\* A call button located on a floor, requesting service upwards or downwards.

\* @author MattCasiro

\* Created: 23 May 2016

\*/

**public class CallButton {**

private static final String UP = "UP";

private static final String DOWN = "DOWN";

private boolean isActive;

private int floorID;

private String floorName, direction;

private ButtonLight btnLight;

/\*\*

\* Create a button that corresponds to a specific floor and contains a light.<br>

\*

\* PRE: An elevator system exists with a floor for the call button to belong to<br>

\* POST: A call button was created in the inactive state.<br>

\* Cleanup: N/A<br>

\*

\* @param floorID is the floor the button belongs to<br>

\* @param floorName is the label for the floor<br>

\* @param direction is whether the caller wants to go up or down<br>

\*/

**public CallButton(int floorID, String** **floorName, String direction) throws IllegalArgumentException**

{

if (direction.equals(UP) || direction.equals(DOWN))

{

this.direction = direction;

} else {

throw new IllegalArgumentException();

}

isActive = false;

this.floorID = floorID;

this.floorName = floorName;

btnLight = new ButtonLight();

}

/\*\*

\* Get the ID of the floor this button belongs to.

\*

\* PRE: N/A<br>

\* POST: N/A<br>

\* Cleanup: N/A<br>

\*

\* @return the floor ID value as an integer

\*/

**public int getFloorID()**

{

return floorID;

}

/\*\*

\* Get the name label of the floor this button represents.

\*

\* PRE: N/A<br>

\* POST: N/A<br>

\* Cleanup: N/A<br>

\*

\* @return the floor name as a String

\*/

**public String getFloorName()**

{

return floorName;

}

/\*\*

\* Get the direction that this button requests an elevator for.

\*

\* PRE: N/A<br>

\* POST: N/A<br>

\* Cleanup: N/A<br>

\*

\* @return the floor's direction as a String

\*/

**public String getDirection()**

{

return direction;

}

/\*\*

\* Get the current state of the button.

\* @return true if the button is active, false if the button is inactive

\*/

**public boolean getButtonState()**

{

return isActive;

}

/\*\*

\* Get the current state of the light for the button.

\* @return true if the light is on, false if the light is off

\*/

**public boolean getLightState()**

{

return btnLight.getState();

}

/\*\*

\* Activate the floor button.

\* PRE: The button is not active

\* POST: The button was activated and the light turned on

\*/

**public void activate()**

{

isActive = true;

btnLight.turnOn();

}

/\*\*

\* Deactivate the floor button.

\* PRE: The button is activated

\* POST: The button was deactivated and the light was turned off

\*/

**public void deactivate()**

{

isActive = false;

btnLight.turnOff();

}

}

**CALL REQUEST LIST**

import java.util.LinkedList;

import java.util.ListIterator;

/\*\*

\*

\* @author MattCasiro

\* Created: 23 May 2016

\*/

**public class CallRequestList**

{

private static final String UP = "UP";

private static final String DOWN = "DOWN";

private LinkedList<CallButton> destinationList;

private String direction;

/\*\*

\* Instantiate a Call Request List to hold a list of waiting elevator calls

\* that want to move in a specified direction.

\* @param direction is "UP" if the list services up call buttons, otherwise "DOWN"

\*/

**public CallRequestList(String direction) throws IllegalArgumentException**

{

if (direction.equals(DOWN) || direction.equals(UP))

{

this.direction = direction;

} else {

throw new IllegalArgumentException();

}

destinationList = new LinkedList<>();

}

/\*\*

\* Assign valid destinations to a given elevator.<br>

\*

\* PRE: The elevator is moving in the same direction as the Call Request List

\* is serving, or the elevator is stationary<br>

\* POST: The elevator is assigned all valid destinations given its current state<br>

\* Cleanup: N/A<br>

\*

\* @param e is the elevator being assigned new destinations

\*/

**public void getDestinations(Elevator e)**

{

// Iterate over the destination list and add call buttons to the elevator

// if the elevator is in a position to service them

ListIterator i = destinationList.listIterator();

while (i.hasNext()) {

CallButton callBtn;

callBtn = (CallButton) i.next();

if (isValidFloor(callBtn, e))

{

// Add the floor to the elevator and remove from the request list

// if possible, otherwise catch the error to prevent a crash

try {

e.addServiceRequest(callBtn);

i.remove();

} catch (IllegalArgumentException ex) {

// crash prevention - i.remove() fails if exception is thrown

// above so list state is maintained.

}

}

}

}

/\*\*

\* Check if floorID should be added to an elevator's destination list.<br>

\*

\* PRE: N/A<br>

\* POST: N/A<br>

\* Cleanup: N/A<br>

\*

\* @param floorID is the integer representing the floor being checked

\* @param currentFloor is the elevator's current location

\* @return true if the elevator is going up and the floor is above it, or

\* if the elevator is going down and the floor is below it

\*/

**private boolean isValidFloor(CallButton callBtn, Elevator e)**

{

// Compare call button direction to elevator direction and add if matching

if (direction.equals(UP) && e.getDirection() == 1)

{

return callBtn.getFloorID() > e.getCurrentFloor();

} else if (direction.equals(DOWN) && e.getDirection() == -1) {

return callBtn.getFloorID() < e.getCurrentFloor();

} else if (e.getDirection() == 0) {

return true;

}

return false;

}

/\*\*

\* Add a floor to the destination list.<br>

\*

\* PRE: The floorID does not already have an active call button for this direction<br>

\* POST: The floorID is added to the list of floors requesting service<br>

\* Cleanup: N/A<br>

\*

\* @param callBtn is the call button requesting service

\*/

**public void addDestination(CallButton callBtn)**

{

destinationList.add(callBtn);

callBtn.activate();

}

}

**DESTINATION LIST**

import java.util.Collections;

import java.util.PriorityQueue;

/\*\*

\*

\* @author MattCasiro

\* Created: 23 May 2016

\*/

**public class DestinationList**

{

private static final String UP = "UP";

private static final String DOWN = "DOWN";

/\*\*

\* A destination object contains a floor id and the requested direction from

\* a call button OR elevator button service request, direction is the string null

\* for elevator buttons.

\*/

**protected class Destination {**

int floorID;

String direction;

/\*\*

\* Create a destination object to represent a floor requested and direction to travel in.<br>

\*

\* PRE: N/A<br>

\* POST: A destination object is created<br>

\* Cleanup: N/A<br>

\*

\* @param floorID is the ID of the floor to service

\* @param direction is "UP" or "DOWN" for a call button and "null" for an elevator button

\* @throws IllegalArgumentException if direction is invalid or floor ID is out of range for the system

\*/

**private Destination(int floorID, String direction) throws IllegalArgumentException**

{

if (direction.equals(UP) || direction.equals(DOWN) || direction.equals("null"))

{

this.direction = direction;

} else {

throw new IllegalArgumentException();

}

this.floorID = floorID;

}

}

private PriorityQueue<Destination> upwardDestinations;

private PriorityQueue<Destination> downwardDestinations;

/\*\*

\* Instantiate a destination list to track floors assigned to an elevator for servicing.<br>

\*

\* PRE: N/A<br>

\* POST: A destination list was created<br>

\* Cleanup: N/A<br>

\*

\* @param numFloors is the number of floors the elevator has access to

\*/

**public DestinationList(int numFloors)**

{

// Set the heap for upward progressing floors to a MIN heap

upwardDestinations = new PriorityQueue<>(numFloors);

// Set the heap for downward progressing floors to a MAX heap

downwardDestinations = new PriorityQueue<>(numFloors, Collections.reverseOrder());

}

/\*\*

\* Add a destination to the destination list.<br>

\*

\* PRE: The elevator is not already at the destination floor ID<br>

\* POST: A destination was added to the list<br>

\* Cleanup: N/A<br>

\*

\* @param floorID is the destination ID that requires servicing

\* @param location is the current location of the elevator

\* @param direction is the direction the destination wants to go, or null for an

\* internal elevator button

\* @throws IllegalArgumentException if floorID = location

\*/

**public void addDestination(int floorID, int location, String direction) throws IllegalArgumentException**

{

if (floorID > location)

{

upwardDestinations.add(new Destination(floorID, direction));

} else if (floorID < location) {

downwardDestinations.add(new Destination(floorID, direction));

} else {

throw new IllegalArgumentException();

}

}

/\*\*

\* Get the next destination the elevator should service.<br>

\*

\* PRE: N/A<br>

\* POST: A destination was removed from the destination list<br>

\* Cleanup: N/A<br>

\*

\* @param motion is the direction the elevator is moving (1 for up, -1 for down)

\* @return the next destination the elevator should service

\* @throws IllegalArgumentException if motion is not 1 or -1

\*/

**public Destination getNextDestination(int motion) throws IllegalArgumentException**

{

if (Math.abs(motion) != 1) throw new IllegalArgumentException();

if (motion == 1)

{

if (upwardDestinations.peek() != null)

{

return upwardDestinations.poll();

} else if (downwardDestinations.peek() != null) {

return downwardDestinations.poll();

}

} else if (motion == -1) {

if (downwardDestinations.peek() != null)

{

return downwardDestinations.poll();

} else if (upwardDestinations.peek() != null) {

return upwardDestinations.poll();

}

}

return new Destination(0, "null");

}

}**DOOR**

/\*\*

\*

\* @author MattCasiro

\* Created: 23 May 2016

\*/

**public class Door {**

private final String[] DOOR\_TYPES = {"OUTER", "INNER"};

private boolean isOpen;

private int floorID, shaftNumber;

private String doorType, floorName;

/\*\*

\* Instantiate an INNER door belonging to an elevator.

\*

\* PRE: An elevator exists for the door to belong to<br>

\* POST: An elevator door was created for an elevator<br>

\* Cleanup: N/A<br>

\*/

**public Door()**

{

this.floorID = 0;

this.floorName = "N/A";

this.shaftNumber = 0;

isOpen = false;

doorType = DOOR\_TYPES[1];

}

/\*\*

\* Instantiate an OUTER door belonging to an elevator.<br>

\*

\* PRE: A floor exists for the elevator to below to<br>

\* POST: A door was created for a floor<br>

\* Cleanup: N/A<br>

\*

\* @param floorID is the id number for the floor the door is on

\* @param floorName is the label for the floor the door is on

\* @param shaftNumber is the shaft number that the door services

\*/

**public Door(int floorID, String floorName, int shaftNumber)**

{

this.floorID = floorID;

this.floorName = floorName;

this.shaftNumber = shaftNumber;

isOpen = false;

doorType = DOOR\_TYPES[0];

}

/\*\*

\* Get the state of the door.<br>

\*

\* PRE: N/A<br>

\* POST: N/A<br>

\* Cleanup: N/A<br>

\*

\* @return true if the door is open, otherwise false

\*/

**public boolean isOpen()**

{

return isOpen;

}

/\*\*

\* Get the type of the door object.<br>

\*

\* PRE: N/A<br>

\* POST: N/A<br>

\* Cleanup: N/A<br>

\*

\* @return "OUTER" for an outer door or "INNER" for an inner door

\*/

**public String getDoorType()**

{

return doorType;

}

/\*\*

\* Open the door.<br>

\*

\* PRE: The door is closed<br>

\* POST: The door was opened<br>

\* Cleanup: N/A<br>

\*/

**public void open()**

{

isOpen = true;

}

/\*\*

\* Close the door.<br>

\*

\* PRE: The door is open<br>

\* POST: The door was closed<br>

\* Cleanup: N/A<br>

\*/

**public void close()**

{

isOpen = false;

}

}**ELEVATOR**

/\*\*

\* An elevator object that moves between floors and maintains a destination list

\* of what floors it needs to service.

\* @author MattCasiro

\* Created: 23 May 2016

\*/

**public class Elevator**

{

private final String UP = "UP";

private final String DOWN = "DOWN";

private static final int MAX\_FLRS = 999;

private static final int MIN\_FLRS = 2;

private static final int MIN\_SUB\_FLRS = 0;

private boolean hasThirteen, hasArrived;

private int elevatorID, shaftID, currentFloor, destination, topFloor, bottomFloor;

private int offset; // Offset is used to account for sub-floors when returning values

private static int count = 0; // ID's to track multiple elevators in larger system

private Door innerDoor;

private FloorButton[] floorButtons;

private DestinationList destinationList;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Track motion state: \*

\* -1 : moving down \*

\* 0 : stationary \*

\* 1 : moving up \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

private int motion;

/\*\*

\* Instantiate an elevator object with only above-ground floors.<br>

\*

\* Pre: A system exists to control the elevator<br>

\* Post: An elevator object was created and set to be on the first floor<br>

\* Cleanup: N/A<br>

\*

\* @param numFloors is a positive number greater than one that is the number of above-ground floors the elevator can access

\* @param shaftID is the shaft in which the elevator operates, or 0 in a single elevator system

\* @param hasThirteen is True if the system has a 13th floor, otherwise False

\*/

**public Elevator(int numFloors, int shaftID, boolean hasThirteen) throws IllegalArgumentException**

{

if (MIN\_FLRS > numFloors || numFloors > MAX\_FLRS)

{

throw new IllegalArgumentException();

}

motion = 0;

this.hasThirteen = hasThirteen;

elevatorID = getNextID();

currentFloor = 1 + offset;

this.shaftID = shaftID;

this.topFloor = hasThirteen ? numFloors : (numFloors + 1);

this.offset = 0;

bottomFloor = 1;

innerDoor = new Door();

destinationList = new DestinationList(numFloors);

try

{

setFloorButtons();

} catch (IllegalStateException ex) {

System.out.println("Illegal state reached when created elevator! Floors not instantiated.");

System.out.println(ex);

}

}

/\*\*

\* Instantiate an elevator object with above and below ground floors.<br>

\*

\* Pre: A system exists to control the elevator<br>

\* Post: An elevator object was created and set to be on the first floor<br>

\* Cleanup: N/A<br>

\*

\* @param numFloors is a positive number greater than one that is the number of above-ground floors the elevator can access

\* @param shaftID is the shaft in which the elevator operates, or 0 in a single elevator system

\* @param numSubFloors is a number greater than or equal to zero that is the number of below-ground floors the elevator can access

\* @param hasThirteen is True if the system has a 13th floor, otherwise False

\*/

**public Elevator(int numFloors, int shaftID, int numSubFloors, boolean hasThirteen) throws IllegalArgumentException**

{

if (MIN\_FLRS > numFloors || MIN\_SUB\_FLRS > numSubFloors ||

numFloors > MAX\_FLRS || numSubFloors > MAX\_FLRS)

{

throw new IllegalArgumentException();

}

motion = 0;

this.hasThirteen = hasThirteen;

elevatorID = getNextID();

currentFloor = 1 + offset;

this.shaftID = shaftID;

this.topFloor = (!hasThirteen && numFloors > 12) ? (numFloors + 1) : numFloors;

this.offset = numSubFloors;

bottomFloor = -offset;

innerDoor = new Door();

destinationList = new DestinationList(numFloors + numSubFloors);

try

{

setFloorButtons();

} catch (IllegalStateException ex) {

System.out.println("Illegal state reached when created elevator! Floors not instantiated.");

System.out.println(ex);

}

}

/\*\*

\* Create an array of FloorButton objects representing the

\* floors the elevator has access to.

\*/

**private void setFloorButtons()**

{

int tmp = topFloor + offset;

// Add one to account for ignoring floor zero

floorButtons = new FloorButton[tmp + 1];

for (int i = 0, j = offset; i <= tmp; i++)

{

// Skip floor zero (and thirteen if required)

if ((i - offset == 0) || (!hasThirteen && i == 13)) continue;

// Generate labels for each floor for use in GUI applications

String name;

if (j > 0)

{

if (j < 10) name = "B:00";

else if (j < 100) name = "B:0";

else name = "B:";

floorButtons[i] = new FloorButton(-j, name + j);

j--;

} else {

if (i < 10) name = "F:00";

else if (i < 100) name = "F:0";

else name = "F:";

floorButtons[i] = new FloorButton(i-offset, name + (i-offset));

}

}

}

/\*\*

\* Increment elevator count and return an ID for an elevator.

\*/

**private int getNextID()**

{

return (++count) + 100;

}

/\*\*

\* Get the state of the button for a specific floor.<br>

\*

\* PRE: N/A<br>

\* POST: N/A<br>

\* Cleanup: N/A<br>

\*

\* @param floor is the reference for which button to check

\* @return true if the button is active, false if the button is inactive

\*/

**public boolean getButtonState(int floor)**

{

return floorButtons[floor].getButtonState();

}

/\*\*

\* Get the state of the light for a button for a specific floor.<br>

\*

\* PRE: N/A<br>

\* POST: N/A<br>

\* Cleanup: N/A<br>

\*

\* @param floor is the reference for which button to check

\* @return true if the light is active, false if the light is inactive

\*/

**public boolean getButtonLightState(int floor)**

{

return floorButtons[floor].getLightState();

}

/\*\*

\* Get the state of the elevator's door.<br>

\*

\* PRE: N/A<br>

\* POST: N/A<br>

\* Cleanup: N/A<br>

\*

\* @return true if the door is open, false if the door is closed

\*/

**public boolean getDoorState()**

{

return innerDoor.isOpen();

}

/\*\*

\* Get the current number of elevators created.<br>

\*

\* PRE: N/A<br>

\* POST: N/A<br>

\* Cleanup: N/A<br>

\*

\* @return the current number of elevators

\*/

**public int getElevatorCount()**

{

return count;

}

/\*\*

\* Get the ID of the elevator object.<br>

\*

\* PRE: N/A<br>

\* POST: N/A<br>

\* Cleanup: N/A<br>

\*

\* @return the elevator ID as an integer

\*/

**public int getElevatorID()**

{

return elevatorID;

}

/\*\*

\* Get the shaft ID in which the elevator operates.<br>

\*

\* PRE: N/A<br>

\* POST: N/A<br>

\* Cleanup: N/A<br>

\*

\* @return the ID number for the shaft the elevator was assigned to

\*/

**public int getShaftID()**

{

return shaftID;

}

/\*\*

\* Get the current floor the elevator is on.<br>

\*

\* PRE: N/A<br>

\* POST: N/A<br>

\* Cleanup: N/A<br>

\*

\* @return the current floor as an integer

\*/

**public int getCurrentFloor()**

{

return currentFloor;

}

/\*\*

\* Get the top floor the elevator has access to.<br>

\*

\* PRE: N/A<br>

\* POST: N/A<br>

\* Cleanup: N/A<br>

\*

\* @return the top floor as an integer

\*/

**public int getTopFloor()**

{

return topFloor;

}

/\*\*

\* Get the lowest floor the elevator has access to.<br>

\*

\* PRE: N/A<br>

\* POST: N/A<br>

\* Cleanup: N/A<br>

\*

\* @return the lowest floor as an integer (first floor is 1, basements are

\* reverse indexed starting at -1 and descending, floor zero never exists)

\*/

**public int getBottomFloor()**

{

return bottomFloor;

}

/\*\*

\* Get the current destination for the elevator.<br>

\*

\* PRE: N/A<br>

\* POST: N/A<br>

\* Cleanup: N/A<br>

\*

\* @return the next location the elevator will stop at given it's current state, as an integer

\*/

**public int getDestination()**

{

return destination;

}

/\*\*

\* Get the current direction the elevator is traveling in.<br>

\*

\* PRE: N/A<br>

\* POST: N/A<br>

\* Cleanup: N/A<br>

\*

\* @return 1 if the elevator is moving up, 0 if it is stationary, and -1 if it is moving down

\*/

**public int getDirection()**

{

return motion;

}

/\*\*

\* Check if the elevator has arrived at the set destination.<br>

\*

\* PRE: N/A<br>

\* POST: N/A<br>

\* Cleanup: N/A<br>

\*

\* @return true if the elevator has just arrived, false if not

\*/

**public boolean isArrived()**

{

return hasArrived;

}

/\*\*

\* Adds a floor number to the destination list for this elevator.<br>

\*

\* PRE: The request was generated by a floor button<br>

\* POST: The floor is added to the elevator's list of destinations, or the

\* elevator signals it's arrival if the floor is the current floor, the

\* floor button is activated and the button lit<br>

\* Cleanup: N/A<br>

\*

\* @param floorID is a floor that is accessible by the elevator

\* @throws IllegalArgumentException if the floorID is invalid

\*/

**public void selectFloor(int floorID) throws IllegalArgumentException**

{

// Close elevator doors once user selects a floor

innerDoor.close();

// DEBUG: System.out.println("Top floor: " + topFloor);

if (floorID > topFloor || floorID < bottomFloor)

{

throw new IllegalArgumentException();

}

floorButtons[floorID].activate();

// Signal arrived if the current floor is selected

if (floorID == currentFloor)

{

arrived();

// If the elevator has no destination, set the floorID as the destination

} else if (destination == 0) {

destination = floorID;

// If the selected floor should be the next destination, put the current

// destination back in the destination list and use the selected floor

// as the new next destination

} else if (motion == 1 && floorID > currentFloor && floorID < destination) {

destinationList.addDestination(destination, currentFloor, null);

destination = floorID;

} else if (motion == -1 && floorID < currentFloor && floorID > destination) {

destinationList.addDestination(destination, currentFloor, null);

destination = floorID;

} else {

destinationList.addDestination(floorID, currentFloor, null);

}

// If the elevator is not moving, start it in the direction of the latest request

if (motion == 0 && destination != 0)

{

motion = (destination > currentFloor ? 1 : -1);

}

}

/\*\*

\* Adds a floor number to the destination list for this elevator.<br>

\*

\* PRE: The request was generated by a call button<br>

\* POST: The floor is added to the elevator's list of destinations, or the

\* elevator signals it's arrival if the floor is the current floor<br

\* Cleanup: N/A<br>

\*

\* @param callBtn is the call button making the elevator service request

\*/

**public void addServiceRequest(CallButton callBtn) throws IllegalArgumentException**

{

int floorID = callBtn.getFloorID();

String direction = callBtn.getDirection();

if (floorID > topFloor || floorID < bottomFloor)

{

throw new IllegalArgumentException();

}

// Check if call should be new destination

// Signal arrived if the current floor is selected

if (floorID == currentFloor)

{

arrived();

// If the elevator has no destination, set the floorID as the destination

} else if (destination == 0) {

destination = floorID;

// If the selected floor should be the next destination, put the current

// destination back in the destination list and use the selected floor

// as the new next destination

} else if (motion == 1 && floorID > currentFloor && floorID < destination && direction.equals(UP)) {

destinationList.addDestination(destination, currentFloor, "null");

destination = floorID;

} else if (motion == -1 && floorID < currentFloor && floorID > destination && direction.equals(DOWN)) {

destinationList.addDestination(destination, currentFloor, "null");

destination = floorID;

} else {

destinationList.addDestination(floorID, currentFloor, direction);

}

}

/\*\*

\* Conduct a single time-step event for an elevator.<br>

\*

\* PRE: N/A<br>

\* POST: The elevator moves as required and the state is updated to represent

\* any required changes<br>

\* Cleanup: N/A<br>

\*/

**public void tick() throws IllegalStateException**

{

hasArrived = false;

// Close door before moving, if open

if (innerDoor.isOpen()) innerDoor.close();

// Move in the current direction for the elevator

switch (motion) {

case 0:

break;

case 1:

incrementFloor();

break;

case -1:

decrementFloor();

break;

default:

throw new IllegalStateException();

}

// Check if elevator is at next destination

if (currentFloor == destination)

{

arrived();

}

// Update motion for next destination

if (destination == 0)

{

motion = 0;

} else if (destination > currentFloor) {

motion = 1;

} else if (destination < currentFloor) {

motion = -1;

} else {

throw new IllegalStateException();

}

}

/\*\*

\* Move the elevator up one floor.

\*

\* PRE: The elevator is not on the top floor

\* POST: The elevator ascends one floor

\*

\* @throws IllegalStateException if the elevator tries to ascend above the highest floor

\*/

**private void incrementFloor() throws IllegalStateException**

{

int tmpFloor = currentFloor;

currentFloor++;

// Move past null floors

if (currentFloor == 0 || (!hasThirteen && currentFloor == 13))

{

currentFloor++;

}

// Prevent crashing in to the top of the shaft

if (currentFloor > topFloor)

{

currentFloor = tmpFloor;

throw new IllegalStateException();

}

}

/\*\*

\* Move the elevator down one floor.

\*

\* PRE: The elevator is not on the bottom floor

\* POST: The elevator descends one floor

\*

\* @throws IllegalStateException if the elevator tries to descend below the lowest floor

\*/

**private void decrementFloor() throws IllegalStateException**

{

int tmpFloor = currentFloor;

currentFloor--;

// Move past null floors

if (currentFloor == 0 || (!hasThirteen && currentFloor == 13))

{

currentFloor--;

}

// Prevent crashing in to the bottom of the shaft

if (currentFloor < -offset)

{

currentFloor = tmpFloor;

throw new IllegalStateException();

}

}

/\*\*

\* Play a chime noise if supported, otherwise output a chime message to screen.

\*/

**private void playChime()**

{

System.out.println("Elevator: " + elevatorID + " played it's chime!");

}

/\*\*

\* Adjust the elevator state to account for arriving at the given destination

\*/

**private void arrived() {**

if(!hasArrived) {

floorButtons[currentFloor].deactivate();

innerDoor.open();

playChime();

hasArrived = true;

destination = destinationList.getNextDestination(motion).floorID;

}

}

}

**ELEVATOR SYSTEM**

import java.util.Stack;

/\*\*

\* An elevator system controller.

\* @author MattCasiro

\* Created: 23 May 2016

\*/

**public class ElevatorSystem {**

public static final int MAX\_FLRS = 999;

public static final int MIN\_FLRS = 2;

public static final int MIN\_SUB\_FLRS = 0;

private static final String UP = "UP";

private static final String DOWN = "DOWN";

private boolean hasThirteen;

private int topFloor, bottomFloor, numShafts;

private int offset; // Offset is used to account for sub-floors when returning values

private CallRequestList upRequestList;

private CallRequestList downRequestList;

private Door[][] outerDoors;

private Stack<Door> openDoors;

private CallButton[] upButtonList;

private CallButton[] downButtonList;

private Elevator[] elevators;

/\*\*

\* Instantiate an elevator system with a given number of floors and elevator shafts.<br>

\*

\* PRE: N/A<br>

\* POST: N/A<br>

\* Cleanup: N/A<br>

\*

\* @param numFloors is the number of above-ground floors the elevator services

\* @param numSubFloors is the number of below-ground floors the elevator services

\* @param hasThirteen indicates whether or not there is a 13th floor

\* @param numShafts is the number of separate elevator shafts in the system

\* @throws IllegalArgumentException if numFloors or numSubFloors is out of the ranges:<br>

\* (MIN\_FLRS <= numFloors <= MAX\_FLRS) and (MIN\_SUB\_FLRS <= numSubFloors <= MAX\_FLOORS)

\*/

**public ElevatorSystem(int numFloors, int numSubFloors, boolean hasThirteen, int numShafts)**

throws IllegalArgumentException

{

if (MIN\_FLRS > numFloors || MIN\_SUB\_FLRS > numSubFloors ||

numFloors > MAX\_FLRS || numSubFloors > MAX\_FLRS)

{

throw new IllegalArgumentException();

}

this.hasThirteen = hasThirteen;

this.numShafts = numShafts;

// Add a floor if skipping thirteen unless floor count is under 13

this.topFloor = (!hasThirteen && numFloors > 12) ? (numFloors + 1) : numFloors;

this.offset = numSubFloors;

this.bottomFloor = (offset != 0) ? -offset : 1;

upRequestList = new CallRequestList(UP);

downRequestList = new CallRequestList(DOWN);

// Add one to account for ignoring floor zero

upButtonList = new CallButton[topFloor + offset + 1];

downButtonList = new CallButton[topFloor + offset + 1];

setCallButtons(upButtonList, UP);

setCallButtons(downButtonList, DOWN);

createDoors();

createElevators(numFloors, numSubFloors);

openDoors = new Stack<>();

}

/\*\*

\* Generate the buttons for each floor as required

\*/

**private void setCallButtons(CallButton[] buttonList, String direction)**

{

int tmp = topFloor + offset;

for (int i = 0, j = offset; i <= tmp; i++)

{

// No up button on highest floor

if (i == tmp && direction.equals(UP)) continue;

// No down button on lowest floor

if (i == 0 && direction.equals(DOWN)) continue;

// Skip floor zero (and thirteen if required)

if ((i - offset == 0) || (!hasThirteen && i == 13)) continue;

// Generate labels for each floor for use in GUI applications

String name;

if (j > 0)

{

if (j < 10) name = "B:00";

else if (j < 100) name = "B:0";

else name = "B:";

buttonList[i] = new CallButton(-j, name + j, direction);

j--;

} else {

if (i < 10) name = "F:00";

else if (i < 100) name = "F:0";

else name = "F:";

buttonList[i] = new CallButton(i-offset, name + (i-offset), direction);

}

}

}

/\*\*

\* Create outer door objects for each shaft on each floor

\*/

**private void createDoors()**

{

int tmp = topFloor + offset;

// Add one to account for no floor zero or shaft zero

outerDoors = new Door[tmp + 1][numShafts + 1];

// instantiate Doors in nested loops (Floors -> Shafts)

for (int i = 0, j = offset; i <= tmp; i++)

{

// Skip floor zero (and thirteen if required)

if ((i - offset == 0) || (!hasThirteen && i == 13)) continue;

for (int s = 0; s < numShafts; s++)

{

// Generate labels for each door in the system

String name;

if (j > 0)

{

if (j < 10) name = "B:00" + j + "-" + s;

else if (j < 100) name = "B:0" + j + "-" + s;

else name = "B:" + j + "-" + s;

outerDoors[i][s] = new Door(-j, name, s);

j--;

} else {

if (i < 10) name = "F:00" + (i-offset) + "-" + s;

else if (i < 100) name = "F:0" + (i-offset) + "-" + s;

else name = "F:" + (i-offset) + "-" + s;

outerDoors[i][s] = new Door(i-offset, name, s);

}

}

}

}

/\*\*

\* Create elevator objects for each shaft

\*/

**private void createElevators(int numFloors, int numSubFloors)**

{

elevators = new Elevator[numShafts];

for (int s = 0; s < numShafts; s++)

{

elevators[s] = new Elevator(numFloors, numSubFloors, hasThirteen);

}

}

/\*\*

\* Get the state of the door for a given floor -> shaft combination.<br>

\*

\* PRE: N/A<br>

\* POST: N/A<br>

\* Cleanup: N/A<br>

\*

\* @param floor is the floor the door is on

\* @param shaft is the shaft the door belongs to

\* @return true if the door is open, false if the door is closed

\*/

**public boolean isFloorDoorOpen(int floor, int shaft) throws IllegalArgumentException**

{

if (bottomFloor > floor || floor > topFloor ||

shaft < 0 || shaft > elevators.length - 1)

{

throw new IllegalArgumentException();

}

return outerDoors[floor][shaft].isOpen();

}

/\*\*

\* Get the state of a given call button.<br>

\*

\* PRE: N/A<br>

\* POST: N/A<br>

\* Cleanup: N/A<br>

\*

\* @param floor is the floor the button is on

\* @param direction indicates if you are checking the UP button or the DOWN button

\* @return true if the button is active, false if the button is inactive

\*/

**public boolean isCallButtonActive(int floor, String direction)**

{

return (direction.equals(UP) ? upButtonList[floor].getButtonState() : downButtonList[floor].getButtonState());

}

/\*\*

\* Get the state of a given call button's light.<br>

\*

\* PRE: N/A<br>

\* POST: N/A<br>

\* Cleanup: N/A<br>

\*

\* @param floor is the floor the button is on

\* @param direction indicates if you are checking the UP button or the DOWN button

\* @return true if the light is on, false if the light is off

\*/

**public boolean isCallButtonLightLit(int floor, String direction)**

{

return (direction.equals(UP) ? upButtonList[floor].getLightState() : downButtonList[floor].getLightState());

}

/\*\*

\* Get the state of a given elevator's given floor button.<br>

\*

\* PRE: N/A<br>

\* POST: N/A<br>

\* Cleanup: N/A<br>

\*

\* @param floor is the button in the elevator to check

\* @param shaft is the shaft the elevator to check is in

\* @return true if the button is active, false if the button is inactive

\*/

**public boolean isElevatorButtonActive(int floor, int shaft)**

{

return elevators[shaft].getButtonState(floor);

}

/\*\*

\* Get the state of a given elevator's given button's light.<br>

\*

\* PRE: N/A<br>

\* POST: N/A<br>

\* Cleanup: N/A<br>

\*

\* @param floor is the button in the elevator to check

\* @param shaft is the shaft the elevator to check is in

\* @return true if the light is on, false if the light is off

\*/

**public boolean isElevatorButtonLightLit(int floor, int shaft)**

{

System.out.println("Shaft " + shaft);

return elevators[shaft].getButtonLightState(floor);

}

/\*\*

\* Get the state of a given elevator's door.<br>

\*

\* PRE: N/A<br>

\* POST: N/A<br>

\* Cleanup: N/A<br>

\*

\* @param shaft is the shaft the elevator to check is in

\* @return true if the door is open, false otherwise

\*/

**public boolean isElevatorDoorOpen(int shaft)**

{

return elevators[shaft].getDoorState();

}

/\*\*

\* Get the location of a given elevator.<br>

\*

\* PRE: N/A<br>

\* POST: N/A<br>

\* Cleanup: N/A<br>

\*

\* @param shaft is the shaft in which the elevator resides

\* @return the floor the elevator is on as an integer

\*/ **public int getElevatorLocation(int shaft)**

{

return elevators[shaft].getCurrentFloor();

}

/\*\*

\* Call an elevator to service the specified floor.<br>

\*

\* PRE: N/A<br>

\* POST: The service request is logged for assignment by the system<br>

\* Cleanup: N/A<br>

\*

\* @param floor is the floor requesting the service

\* @param direction is the intended direction of travel for the service request

\*/

**public void callElevator(int floor, String direction) throws IllegalArgumentException**

{

if (bottomFloor > floor || floor > topFloor)

{

throw new IllegalArgumentException();

}

switch (direction) {

case UP:

upRequestList.addDestination(upButtonList[floor]);

break;

case DOWN:

downRequestList.addDestination(downButtonList[floor]);

break;

default:

throw new IllegalArgumentException();

}

}

/\*\*

\* Record the selection of a destination from inside an elevator.<br>

\*

\* PRE: N/A<br>

\* POST: The system updates the elevator with the given destination<br>

\* Cleanup: N/A<br>

\*

\* @param floor is the floor the user wishes to stop at

\* @param shaft is the shaft in which the elevator resides

\* @throws IllegalArgumentException if floor or shaft are invalid

\*/

**public void selectFloor(int floor, int shaft) throws IllegalArgumentException**

{

// Is this a valid way to check parameters for validity?

try {

outerDoors[elevators[shaft].getCurrentFloor()][shaft].close();

elevators[shaft].selectFloor(floor);

} catch (ArrayIndexOutOfBoundsException ex) {

throw new IllegalArgumentException();

}

}

/\*\*

\* Conduct a single time-event for the elevator system.<br>

\*

\* PRE: N/A<br>

\* POST: All open floor doors are closed, all elevators are assigned any

\* possible destinations, all elevator time events are called, and arrival

\* states checked<br>

\* Cleanup: N/A<br>

\*

\*/

**public void tick()**

{

// Close any open doors

while (!openDoors.empty())

{

openDoors.pop().close();

}

// Add destinations to elevators as possible

for (Elevator e : elevators)

{

if (e.getDirection() >= 0)

{

upRequestList.getDestinations(e);

}

if (e.getDirection() <= 0)

{

downRequestList.getDestinations(e);

}

}

// Tick all elevators and check if they have arrived at a destination

for (Elevator e : elevators)

{

String direction;

e.tick();

if (e.isArrived())

{

if (e.getDirection() == 1)

{

direction = UP;

} else {

direction = DOWN;

}

elevatorArrived(e.getCurrentFloor(), e.getShaftID(), direction);

}

}

}

/\*\*

\* Adjust door and button states when an elevator arrives.

\*/

**private void elevatorArrived(int floorID, int shaftID, String direction) {**

outerDoors[floorID][shaftID].open();

if (direction.equals(UP)) {

upButtonList[floorID].deactivate();

} else {

downButtonList[floorID].deactivate();

}

}

}

**FLOOR BUTTON**

/\*\*

\* A Floor Button object that is identified for a specific floor and has the ability to

\* be lit or un-lit by a light.

\* @author MattCasiro

\* Created: 23 May 2016

\*/

**public class FloorButton**

{

private boolean isActive;

private int floorID;

private String floorName;

private ButtonLight btnLight;

/\*\*

\* Create a button that corresponds to a specific floor and contains a light.<br>

\*

\* PRE: An elevator system exists to contain the floor button<br>

\* POST: A floor button was created with the given floor ID and name<br>

\* Cleanup: N/A<br>

\*

\* @param floorID is the floor the button belongs to

\* @param floorName is the label for the floor

\*/

**public FloorButton(int floorID, String floorName)**

{

isActive = false;

this.floorID = floorID;

this.floorName = floorName;

btnLight = new ButtonLight();

}

/\*\*

\* Get the ID of the floor this button represents.<br>

\*

\* PRE: N/A<br>

\* POST: N/A<br>

\* Cleanup: N/A<br>

\*

\* @return the floor ID value as an integer

\*/

**public int getFloorID()**

{

return floorID;

}

/\*\*

\* Get the name label of the floor this button represents.<br>

\*

\* PRE: N/A<br>

\* POST: N/A<br>

\* Cleanup: N/A<br>

\*

\* @return the floor name as a String

\*/

**public String getFloorName()**

{

return floorName;

}

/\*\*

\* Get the current state of the button.<br>

\*

\* PRE: N/A<br>

\* POST: N/A<br>

\* Cleanup: N/A<br>

\*

\* @return true if the button is active, false if the button is inactive

\*/

**public boolean getButtonState()**

{

return isActive;

}

/\*\*

\* Get the current state of the light for the button.<br>

\*

\* PRE: N/A<br>

\* POST: N/A<br>

\* Cleanup: N/A<br>

\*

\* @return true if the light is on, false if the light is off

\*/

**public boolean getLightState()**

{

return btnLight.getState();

}

/\*\*

\* Activate the floor button.<br>

\*

\* PRE: The button is not active<br>

\* POST: The button was activated and the light turned on<br>

\* Cleanup: N/A<br>

\*/

**public void activate()**

{

isActive = true;

btnLight.turnOn();

}

/\*\*

\* Deactivate the floor button.<br>

\*

\* PRE: The button is activated<br>

\* POST: The button was deactivated and the light was turned off<br>

\* Cleanup: N/A<br>

\*/

**public void deactivate()**

{

isActive = false;

btnLight.turnOff();

}

}