Green Elevator

Andreas Gustafsson || andreg@kth.se
Mattias Knutsson || matknu@kth.se

9 mars 2012

Innehåll

1 Assumptions & Simplifications

- The panel button 'Stop' is emergency halt. The elevator is not supposed to restart after a stop event without maintenance. Therefore the controller thread of an elevator that emergency stops will terminate.
- It is assumed that it takes twice the time to open and close the doors than it takes to travel one floor[1].
- We assume the system should run for eternity (the elevators should never be taken offline. Therefore no communication teardown or synchronized thread termination is implemented.
- When a button is pressed (not a panel button), the controllers are told only which direction is desired, not the exact destination.
- The controllers does not know how many floors there is. It is assumed that a floor request from the elevators will always be within the bounds of the building i.e it is not possible to press the button for the seventh floor in a six-story building.
- There is no way to determine how many people are in the elevator at any one time.
  Example:
  Five people embark on an elevator on the second floor, and request traveling to the fourth floor. Whilst they travel from the second to third floor, a lone man presses the up button on the third floor. Assume it takes thirty seconds to traverse one floor, and thirty seconds to open the doors, admit passengers and close them again. If the elevator pause on the third floor to admit the lone man, it wastes two and a half minutes of the passengers time (thirty seconds each). If it instead travel to the fourth floor and let them disembark before going down to the third floor to pick up the lone man it would only waste a minute and a half (the lone man waits thirty seconds for the elevator to travel from the third floor to the fourth floor, thirty seconds while they disembark and thirty seconds for the elevator to return to his floor.
  The controllers will completely disregard this and pick up the lone man (it could be a hundred men waiting, the controllers know only that the button was pressed once).

[1] As observed in the elevators in the E-building, KTH Campus Valhallavägen.

2 Application

The application that has been added to the Elevators project consists of three Java classes; MasterController.java, ElevatorController.java and Message.java.

2.1 MasterController.java

MasterController receives messages as strings over TCP sockets from the elevators, converts them into messages as described in section 2.3 and forward them to the correct ElevatorController, and receives messages, as described in section 2.3 from the ElevatorControllers and forward them to the elevators. The only ”actual” work the MasterController does is to allocate floor button presses to an elevator, for more information see section 3.1.

2.2 ElevatorController.java

This class manages a single elevator and trusts the MasterController makes the best decisions. The class receives tasks from the Master and incorporates them into its current schedule, see section 3.2 for how this is done. The ElevatorController will send four different kinds of messages to its allotted elevator. It will send ”move in X direction”, ”stop moving”, ”open doors” and ”close doors”.

2.3 Message.java

This class represents a multi-purpose message sent between an ElevatorController and the MasterController. When a message is received from the elevators by MasterController, the string is parsed into a Message and will remain a Message until the command is successfully accepted by the Controller. A Message may be created by the ElevatorController and sent to the Master to be converted into a string and sent back to the elevator through the socket.

3 Algorithm

3.1 Elevator selection

This algorithm is used by the master controller when a floor button is pressed and decides which elevator to server the request. The algorithm will first search for an elevator to ”join”. To ”join” an elevator is to enter an elevator that is already heading in your direction, and has an destination past yours. For instance, an elevator is traveling to floor four from floor one, and you press the ”up” button on floor three. The elevator will then pause at floor three to admit you, before proceeding to floor four.
If there is no elevator to join, the algorithm searches for a free elevator to send to your floor. If it finds one it will be assigned to you and others may ”join” your ride.
If there is neither an elevator to join nor and empty, the algorithm will wait until one of the cases is true. If both are true, a ”join” will be prioritized over assigning an empty elevator.

3.2 Floor scheduling

This algorithm is used by each elevator controller, oblivious to what the other elevator controllers are doing, to schedule the order of floors to visit.
When a new task (a task is ”travel to floor X from the current position before open and close the doors”) is added, it goes through a series of checks to determine when it should be performed.

1. If the elevator is not moving, execute the task immediately.

2. If the elevator is moving upwards, and the new target floor is below (if traveling upwards) or above (if traveling downwards) the elevator’s current floor, the button press is discarded. This means that someone has requested to travel one direction, and then pressed a button that would cause the elevator to reverse direction. Douchebaggery is unacceptable.

3. If the new task is acceptable, it is placed into a queue of tasks. This queue is a priority queue that is ordered by the target floors as integers descending if the elevator is currently moving downwards and ascending if it is moving upwards.

4 Implementation

4.1 MasterController

This class forwards messages from the elevators to the proper ElevatorController and assigns a proper elevator and controller to a floor request (when someone on floor X request an elevator that is moving in direction Y). MasterController extends Thread, and is started from within the initialization of Elevators. See appendix B on page page 13 for code and fig. 1 on page 9 for a flow chart.

4.1.1 run()

Overrides the Thread.run() method. What it does is to connect to the elevators through TCP socket, opens the I/O streams and invoke the controlElevators function (see ??).

4.1.2 controlElevators()

Main workhorse of the MasterController. It starts a thread that will repeatedly poll each controller for a message they wish to be sent to the elevators, and forward it. This thread is anonymously created by wrapping it around a runnable and started inline. Proceed into an infinite loop that reads messages from the socket, and if it is an ”p” or ”f” message, it is forwarded to the proper ElevatorController. If it is a ”b” message, an Assigner (see ??) is inline instantiated and trusted to manage the message.

4.2 Assigner

This transforms a ”b” message to a ”p” message and forwards it to the proper ElevatorController, see section 3.1 for more info about the algorithm.
Assigner extends Thread. See appendix B on page 13 for code and fig. 2 on page 10 for a flow chart.

4.2.1 run()

The assigner knows which floor is requested, and which direction is requested.
Its task is to determine which controller gets the requested task. There are three different outcomes. First is to ”join” a ride (see section 3.1 for clarification of ”join”), the second is to assign an empty elevator to the task. The third solution, if both previous are false, then simply have the Thread yield and repeat the first two steps during the next context switch. Eventually one of the two will be true and the assigner posts the task to the proper thread and terminates.

4.3 ElevatorController

This controls a single elevator. Will receive messages from the MasterController and handle the tasks contained within those messages. ElevatorController is both a sort of monitor and extends Thread (it interacts with the MasterController through synchronized method calls). See appendix C on page 18 for code and fig. 3 on page 11 for a flow chart.

4.3.1 run()

This is the main workhorse of ElevatorController. While it has nothing to do, it yields. It detects if it has something to do by first checking if its inbox (a queue where MasterController puts messages destined to this controller) is empty, and that its taskQueue is empty.
If either of these two conditions fail, it will look at the first element of its task queue (without removing it). If it is not null, and its current floor is not within a ±0.05 interval of the target floor[2], it means it should move towards that floor. It will decide which type of movement is required with decideMove() (see **??**).
It will the proceed to check its inbox, to see if any messages have arrived from the elevators (via MasterController). If the message is not null, there is something. If the message is a ”p” message with floor value of 32000, it is an emergency stop and a stop message is immediately sent to the elevator.
If it is a ”p” message and not an emergency stop, it means that it is a new task, so the message is added to the taskQueue (see section 4.3.2).
If it is a ”f” message, it means that the elevator has moved and the current floor field must be updated. Now, if the current floor is within the ±0.05 interval of the target floor, a ”stop moving” message is sent to the elevator and the doorAction() function is invoked (see section 4.3.4).

4.3.2 addQueue()

This function will add a task to the task queue, and it will do it in one of two ways. If the elevator is intended to move downwards, it will add the task such that the queue is ordered in a descending manner with the highest floor at the head of the queue.
If the elevator is intended to move upwards, it will att the task such that the queue is ordered in an ascending manner with the lowest floor at the head of the queue.
If the new task would cause the elevator to change directions without the task queue being empty at least once before the direction change, the task is ignored.

4.3.3 decideMove()

This function decides if the elevator should move, and in what direction.
If the elevator is already moving, the function call does nothing.
If the current floor is lower than target floor minus 0.05 (see section 4.3.1 for explanation), the direction is upwards and a ”move up” message will be sent to the elevators.
If the current floor is greater than the target floor plus 0.05 a ”move down” message will be sent to the elevators.
If the current floor is within the interval and the elevator is not moving, this function call does nothing.

4.3.4 doorAction()

This function opens and closes the doors of the elevator. It will send an ”open door” message, sleep for a second, send a ”close doors” message and sleep a second before returning.

[2] The elevators will send floor update messages with 0.04 intervals.

4.3.5 Remaining methods in ElevatorController

They are merely synchronized getters/setters adders/pollers.

4.4 Message

Message is a multi-purposed message that is sent between the MasterController and ElevatorControllers. See appendix D on page 25 for code.
A message contains the following fields:

- type: 'p', 'f', 'm', 'd'.
- elevator: Which elevator/controller to the message should go to.
- targetFloor: if it is a 'p' message, it will be the floor the elevator should go to. If it is a 'd' message, it will be 1 for open doors and -1 for close doors. If it is a 'm' message it will be 1 for move up and -1 for move down. If it is an 'f' message, targetFloor is unused and may be whatever.
- curPos: If it is an 'f' message, it will be the current floor. Otherwise it is unused and it will be whatever.

4.4.1 Methods of Message

They are simply getters/setters and overrides of equals and toString. Just standard stuff.

5 Environments

5.1 OS & Java

The program has been developed on, and tested on, a computer running Genuine Windows 7 and Java (TM) 6 Update 22 (64-bit).

5.2 System Specifications

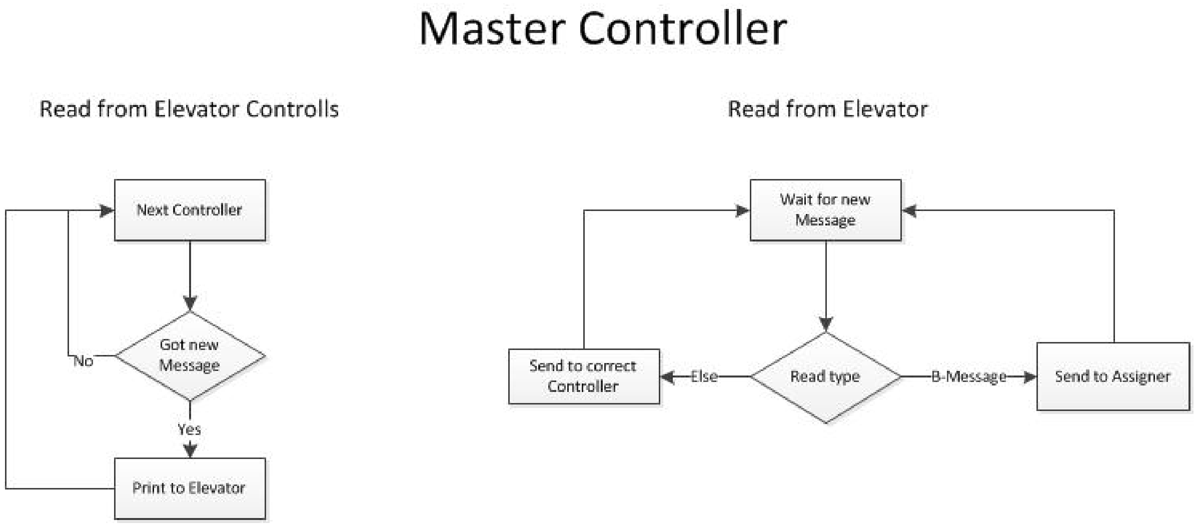
Computer model: Asus UL30VT
OS: Windows 7 Professional 64-bit (6.1, Build 7601)
System Manufacturer: ASUSTeK ComputerINC
Processor: Genuine Intel(R) CPUÛ7300 @ 1.30GHz (2 CPUs), 1̃.3GHz[3]
Memory: 4096MB RAM

[3] Clocked to 80% efficiency for power conserving purposes.

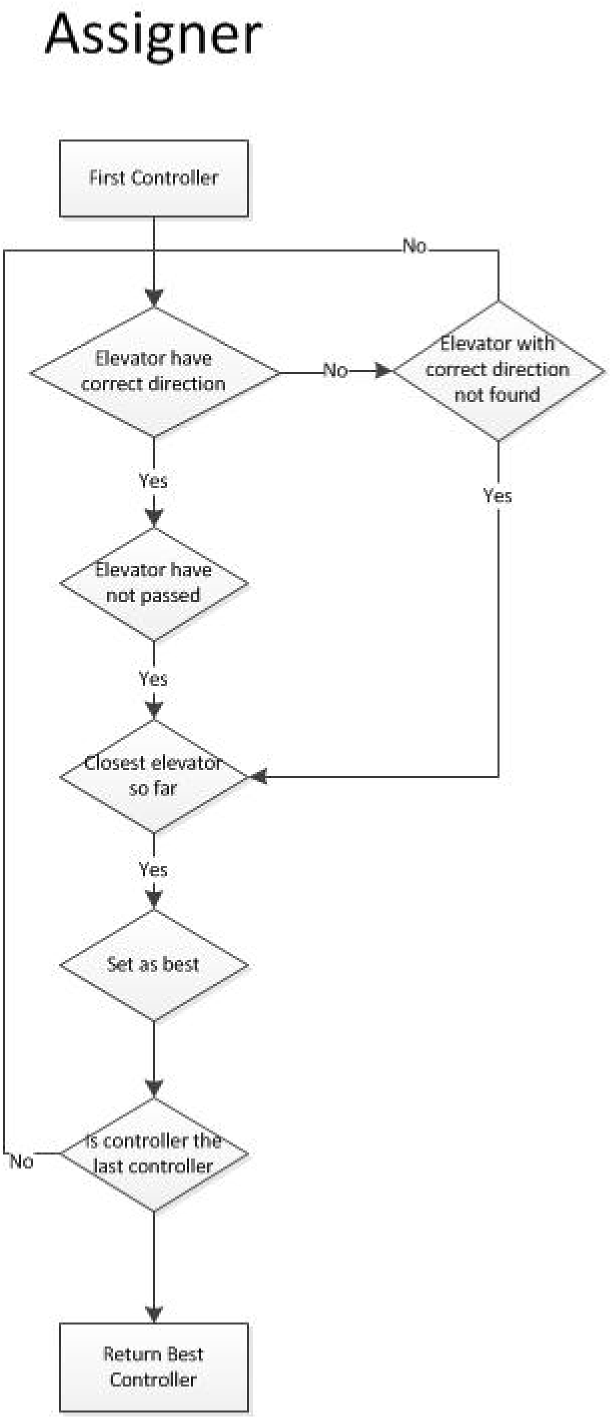
6 Achievements

The task was to minimize service time and elevator movement. It is not possible to do both, one must be sacrificed to benefit the others. The controllers cannot make accurate computations since it may not observe the entire theater. The elevator does not know how many people there are in the elevator, nor how many wish to embark. Nor is it possible for the controller to know how far a person (or group of people) want to travel when they summon they summon the elevator. The controller only knows the desired direction.
Therefore we chose to, if possible, combine trips with the elevator. For instance, one person travels from the second to the fourth floor, and someone summons the elevator to travel upwards from the third floor. Then the elevator already traveling upwards will gather this request and pause at the third floor. This way the goal of minimizing elevator movement (by having ”required moves” overlap, and minimize service time by not locking an additional elevator.

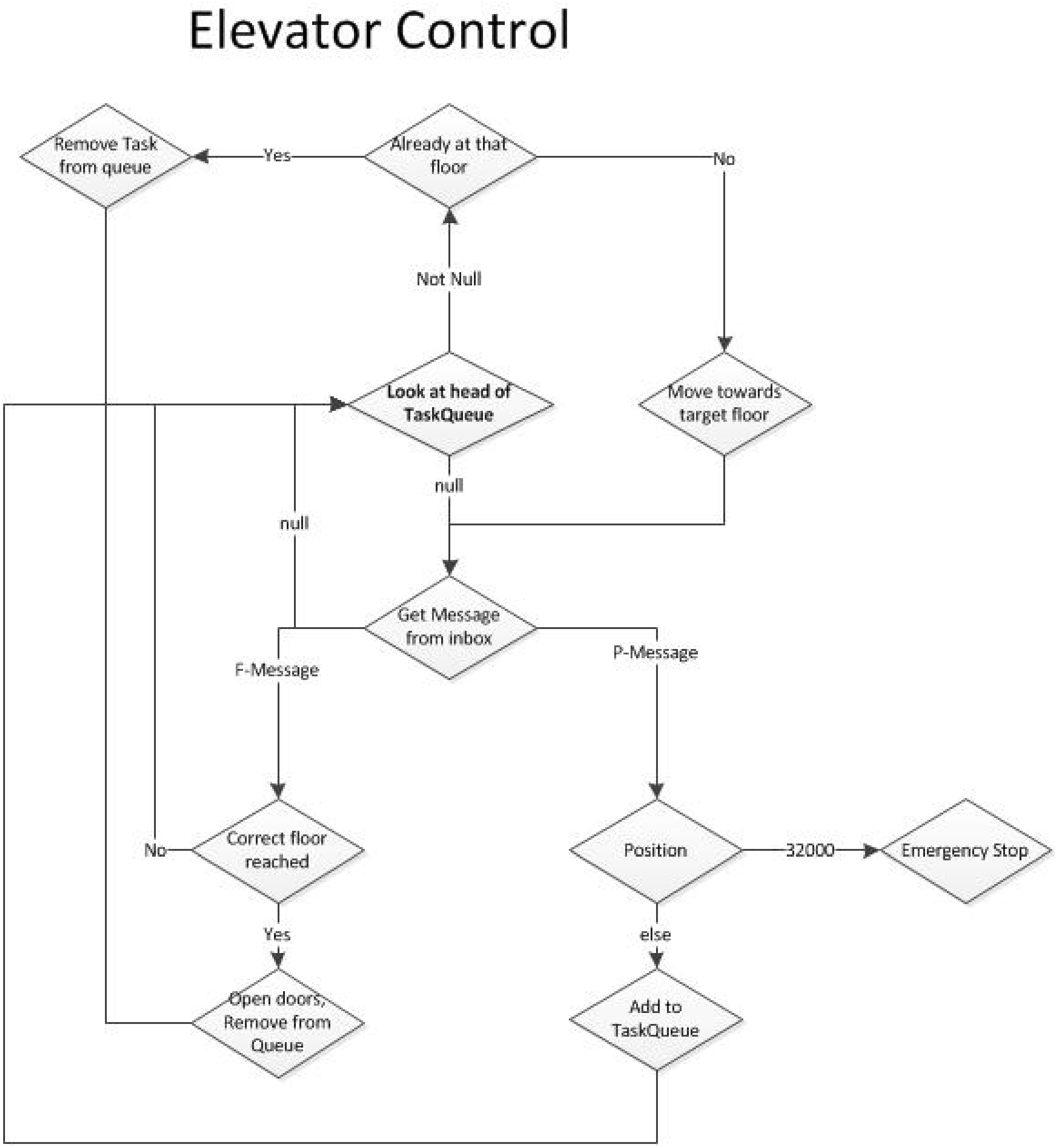
A Flow charts



Figur 1: Flowchart for MasterController.



Figur 2: Flowchart for Assigner.



Figur 3: Flowchart for ElevatorController.

B MasterController.java

```java
package controller;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;
import java.net.SocketException;
import java.net.UnknownHostException;
import java.util.ArrayList;

import elevator.Elevators;

/**
 *
 * @author Mattias Knutsson and Andreas Gustafsson
 *
 */
public class MasterController extends Thread {
    private Socket s;
    private BufferedReader in; // The reader from the socket
    private PrintWriter out; // The writer to the socket
    private final ArrayList<ElevatorController> controllers; // The list of controllers
    private final int N;

    public MasterController(int N) {
        this.N = N;
        controllers = new ArrayList<ElevatorController>();
    }



    public static void main(String[] args) {
        MasterController mc;
        if(args.length>0)
            mc=new MasterController(Integer.valueOf(args[0]));
        else
            mc=new MasterController(1);
        mc.start();
    }
    /**
```

```java
     * Initialize the socket and read/write, then run controllElevators
     */
    @Override
    public void run() {
        while (true) {
            try {
                s = new Socket("localhost", 4711);
                in = new BufferedReader(new InputStreamReader(
                        s.getInputStream()));
                out = new PrintWriter(s.getOutputStream(), true);
                controllElevators();
            } catch (UnknownHostException e) {
                continue;
            } catch (IOException e) {
                continue;
            }
            break;
        }
    }

    /**
     *
     * @throws SocketException
     */
    private void controllElevators() throws SocketException {
        System.err.println("Controller starts.");
        for (int i = 0; i < N; i++) {          //Start all controllers
            controllers.add(new ElevatorController(i + 1));
            controllers.get(i).start();
        }

        //A thread that read the controllers outbox and send to Elevators
        new Thread(new Runnable() {
            public void run() {
                while (true) {
                    for (ElevatorController c : controllers) {
                        Message msg = null;
                        if ((msg = c.retrieveMessage()) != null) {
                            out.println(msg.toString());
                        }
                    }
                    Thread.yield();
                }
            }
        }).start();


        //Read input from Elevators and convert to a message
        //If the message is a b-message send it to assigner
        //Else send it to the correct controller
        while (true) {
            while (!s.isInputShutdown()) {
                String[] message = null;
                try {
                    message = in.readLine().split(" "); //Parse
                      the message
                    char type = message[0].charAt(0);
                    int elevator = Integer.valueOf(message[1]);
                    double modifier =
                      Double.valueOf(message[2]);
                    if (type == 'b') {
                        new Assigner((int) elevator, (int)
                          modifier).start(); //If b-message
                          send to assigner
                        continue;
                    }

                    //Create and send message
                    Message msg = new Message(type, elevator,
                      (int) modifier,
                            modifier);
                    controllers.get(elevator -
                      1).postMessage(msg);
                } catch (Exception e) {
                }

            }
        }
    }
    /**
     * Assigner is threads trying to assign a task to a elevator
     * @author Mattias Knutsson and Andreas Gustafsson
     */
    private class Assigner extends Thread {

        private final int floor;
        private final int direction;

    /**
     * Constructs a Assigner.
     * @param floor - The floor a person want to leave.
     * @param direction - The direction the person want to
       go.
     */
    public Assigner(int floor, int direction) {
        this.floor = floor;
        this.direction = direction;
    }

    /**
     * Try to assign the task to a elevatorController
     * The assigner prefers to assign the task to a
       elevator passing the floor
     * If that isn't possible take the closest unassigned
       elevator
     * Or what until one of this gets possible
     */
    @Override
    public void run() {
        ElevatorController closestEmpty = null, closestJoin
          = null;
        double costEmpty, costJoin;
        costEmpty = costJoin = Double.POSITIVE_INFINITY;
        for (boolean first = true; closestEmpty == null
                && closestJoin == null; first = false) {
                  //While the task haven't a possible
                  assignable elevatorController
            if (!first)
                Thread.yield();
                  //Yield at the start everytime except
                  the first

            for (ElevatorController c : controllers) {

                // Join to a current tour
                if (c.getIntendedDirection() == direction
                              //If the elevators direction
                  is same as the persons direction and the
                  elevator havn't yet passed the floor
                        && c.getDirection() * c.getFloor()
                          < c
                              .getDirection() * floor) {
```

```
                        double thisCost = Math.abs(floor - c.getFloor()) + 2     //Calculate the cost
                                * c.getTaskQueueSize();
                        if (thisCost < costJoin) {
                            costJoin = thisCost;
                            closestJoin = c;                    //Set the controller as closest if it has the least cost
                        }
                    }else if (c.getIntendedDirection()!=c.getDirection() && c.getIntendedDirection()==direction){ //If the elevator have the same intented direction but not started to move in that direction yet
                        double thisCost = Math.abs(floor - c.getFloor()) + 2
                                * c.getTaskQueueSize();
                        if (thisCost < costJoin) {
                            costJoin = thisCost;
                            closestJoin = c;                    //Set the controller as closest if it has the least cost
                        }
                    }

                    // Assign Empty ELevator
                    if (c.getTaskQueueSize() == 0 && c.getInboxSize() == 0) {    //If the elevator is unassigned
                        double thisCost = Math.abs(floor - c.getFloor()) + 2
                                * c.getTaskQueueSize();
                        if (thisCost < costEmpty) {
                            costEmpty = thisCost;
                            closestEmpty = c;                   //Set the controller as closest if it has the least cost
                        }
                    }
                }
            }
            ElevatorController cf = closestEmpty;
            if (closestJoin != null)                //If the is a possible join
                cf = closestJoin;
            else
                cf.setIntendedDirection(direction);

            Message m = new Message('p', cf.getElevator(), floor, floor);   //Create the message translated to a p-message
            System.out.println("ASSIGNER MESSAGE: " + m);
            cf.postMessage(m);                      //Send the message
        }
    }
}
```

C ElevatorController.java

```
package controller;

import java.util.ArrayDeque;
import java.util.ArrayList;
import java.util.Queue;

/**
 * Controll
 * @author Mattias Knutsson and Andreas Gustafsson
 *
 */
public class ElevatorController extends Thread {
    private final Queue<Message> inbox;         //The inbox with messages from MasterController
    private final Queue<Message> outbox;        //The outbox with messages to MasterController
    private final ArrayList<Message> taskQueue; //The queue with tasks the controller will execute
    private double floor;                       //The current position
    private double targetFloor;                 //The destination floor
    private final int elevator;                 //The ID
    private int direction;                      //The direction (1 = UP, -1 = DOWN)
    private int intendedDirection;              //The intended direction (the direction after first pickup)
```

```java

    /**
     *  Constructs a new elevator with standard settings
     * @param elevator - The number ID of the elevator
     */
    public ElevatorController(int elevator) {
        setDirection(0);
        setIntendedDirection(0);
        floor = setTargetFloor(0);
        this.elevator = elevator;
        taskQueue = new ArrayList<Message>();
        outbox = new ArrayDeque<Message>();
        inbox = new ArrayDeque<Message>();
    }

    /**
     * The running class in a elevator controller.
     * Parsed message from MasterController and run tasks from a queue
     */
    @Override
    public void run() {
        try {
            while (true) {
                while (inbox.isEmpty() && taskQueue.isEmpty()){        //While elevator can stay idle
                    Thread.yield();
                }
                Message m = peekQueue();        //Looks at a the top task in the elevators quere
                if (m != null) {        //If was a task in queue
                    setTargetFloor(m.getTargetFloor());        //Set target to top task floor
                    if (getFloor() >= getTargetFloor() - 0.05 && getFloor() <= getTargetFloor() + 0.05) { //If correct floor is reached
                        doorAction();        //Open the doors
                        pollQueue();        //Remove the task from the queue
                        setDirection(0);        //Set the elevator to not move
                        continue;
                    }
                    decideMove();
                          //Get the elevator moving in the
                      correct direction
                }
                Message msg = pollMessage();
                      //Get message from mastercontroller
                if (msg != null) {
                    switch (msg.getType()) {
                          //Get the type of the message
                    case 'p':
                          //P-Message is a message with a move
                      to this floor order
                        if (msg.getTargetFloor() == 32000) {
                              //Emergency stop
                            addMessage(new Message('m',
                              getElevator(), 0, 0)); //Stop
                              elevator
                            synchronized (this) {
                                inbox.clear();
                                      //Clear both queues
                                taskQueue.clear();
                            }
                            setDirection(0);
                                  //Set direction to not move
                            return;
                        }
                        addQueue(msg);
                              //Add the message to the queue
                        break;
                    case 'f':
                          //F-Message information to the
                      controller about the elevators current
                      position
                        if (getDirection() == 0)
                              //If the elevator have stopped
                          moving
                            continue;
                        setFloor(msg.getcurPos());
                              //Set the elevators position
                        if (Math.abs(getFloor() -
                          getTargetFloor()) < 0.05) {    //If
                          the correct floor is reached
                            addMessage(new Message('m',
                              getElevator(), 0, 0));  //Stop
                              the elevator
                            doorAction();                           //Open and Close the door
                            pollQueue();                            //Remove the task from the taskqueue
                            setDirection(0);                        //Set the direction to not moving
                        }
                        break;
                    default:
                        System.err.println("Unhandled message.");
                }
            }
            decideMove();               //Get the elevator moving in the correct direction
            Thread.yield();             //Yield the remaining timeslice
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

/**
 * Adding a task to taskqueue sorted by prior order (earlies approach first)
 * @param msg
 */
public synchronized void addQueue(Message msg) {
    System.out.println("Adding task: " + msg);

    if (!(getDirection() * msg.getcurPos() < getDirection()         //Assume that the task is in the elevators direction
            * getFloor())) {
        for (int i = 0; i < taskQueue.size(); i++) {                //Sort in the task in correct position
            if (getDirection() * msg.getcurPos() < getDirection()
                    * taskQueue.get(i).getcurPos()) {
                taskQueue.add(i, msg);                              //Add task to position i in the
```

```
                        queue
                    return;
                }
            }
            taskQueue.add(msg);
                        //Add last in queue
        }else{
            System.err.println("The Elevator is not heading
              that way, douche.");    //If a button push is in
              the wrong direction
        }
    }

    /**
     * Let the controller check the top prior task
     * @return The Message in top of the taskqueue
     */
    public synchronized Message peekQueue() {
        return taskQueue.isEmpty() ? null : taskQueue.get(0);
    }

    /**
     * Remove and returns the first element in the taskqueue
     * @return the message
     */
    public synchronized Message pollQueue() {
        if(taskQueue.size() == 1) setIntendedDirection(0);
        return taskQueue.isEmpty() ? null : taskQueue.remove(0);
    }

    /**
     * Sets direction and sends a message to start move the
        elevator in the correct direction
     */
    private void decideMove() {
        if (getDirection() == 0
                && Math.abs(getFloor() - getTargetFloor()) >
                   0.05) {
            System.err.println("DECIDE MOVE");
            int modifier;
            if (getFloor() < getTargetFloor() - 0.05) {
              //If the elevator should move up
                System.err.println("Going up.");
                setDirection(1);
                modifier = 1;
            } else if (getFloor() > getTargetFloor() + 0.05) { //If the elevator should move down
                System.err.println("Going down.");
                setDirection(-1);
                modifier = -1;
            } else { //If the elevator is in the correct floor
                return;
            }
            Message msg1 = new Message('m', getElevator(), modifier, 0);
            addMessage(msg1); //Add the message to outbox
        }
    }

    /**
     * Send message to open the door, wait a sec, then close the doors again
     * @throws InterruptedException If the sleep interrupts
     */
    private void doorAction() throws InterruptedException {
        addMessage(new Message('d', getElevator(), 1, 0));
        Thread.sleep(1000);
        addMessage(new Message('d', getElevator(), -1, 0));
        Thread.sleep(1000);
    }



    /**
     * Below here is only getters and setters.
     */

    public synchronized Message retrieveMessage() {
        return outbox.poll();
    }

    public synchronized void addMessage(Message msg) {
        System.err.println("Adding message: " + msg);
        outbox.add(msg);
    }

    public synchronized int getInboxSize(){
        return inbox.size();
    }

    public synchronized int getTaskQueueSize(){
        return taskQueue.size();
    }

    public synchronized void postMessage(Message msg) {
        inbox.add(msg);
    }

    public synchronized Message pollMessage() {
        return inbox.poll();
    }

    public synchronized double getFloor() {
        return floor;
    }

    public synchronized void setFloor(double floor) {
        this.floor = floor;
    }

    public synchronized int getDirection() {
        return direction;
    }

    public synchronized void setDirection(int direction) {
        // System.err.println("Setting direction to " + direction);
        this.direction = direction;
    }

    public synchronized double getTargetFloor() {
        return targetFloor;
    }

    public synchronized double setTargetFloor(double targetFloor) {
        this.targetFloor = targetFloor;
        return targetFloor;
    }

    public int getElevator() {
        return elevator;
    }

    public synchronized int getIntendedDirection() {
```

```java
        return intendedDirection;
    }

    public synchronized void setIntendedDirection(int intendedDirection) {
        this.intendedDirection = intendedDirection;
    }

}
```

D Message.java

```java
package controller;

/**
 * Message used to communicate between threads.
 * The class consists of getters and setters and overrides toString and equals for our convenience.
 * @author Mattias Knutsson and Andreas Gustafsson
 */
public class Message {

    private final char type;                //The type of the message (F,P,M etc.)
    private final int elevator;             //The ID of the elevator
    private final int targetFloor;          //The target floor
    private final double curPos;            //The current position

    public Message(char type, int elevator, int targetFloor, double curPos) {
        this.type = type;
        this.elevator = elevator;
        this.targetFloor = targetFloor;
        this.curPos = curPos;
    }

    public char getType() {
        return type;
    }

    public int getElevator() {
        return elevator;
    }

    public double getTargetFloor() {
        return targetFloor;
    }

    @Override
    public boolean equals(Object obj) {
        if (obj instanceof Message) {
            Message m = (Message) obj;
            return type == m.type && elevator == m.elevator
                    && targetFloor == m.targetFloor;
        }
        return false;
    }

    @Override
    public String toString() {
        if(type == 'f')
            return type + " " + elevator + " " + curPos;
        else
            return type + " " + elevator + " " + targetFloor;
    }

    public double getcurPos() {
        return curPos;
    }
}
```