UNIVERSITY OF COPENHAGEN

XMP: Exam
- Programming

Jens Fredskov (chw752)

January 24, 2014

Implementation

The implementation consists of several different processes, in addition to a few helper functions, types and data structures. The processes largely communicate as outlined in Figure 1.

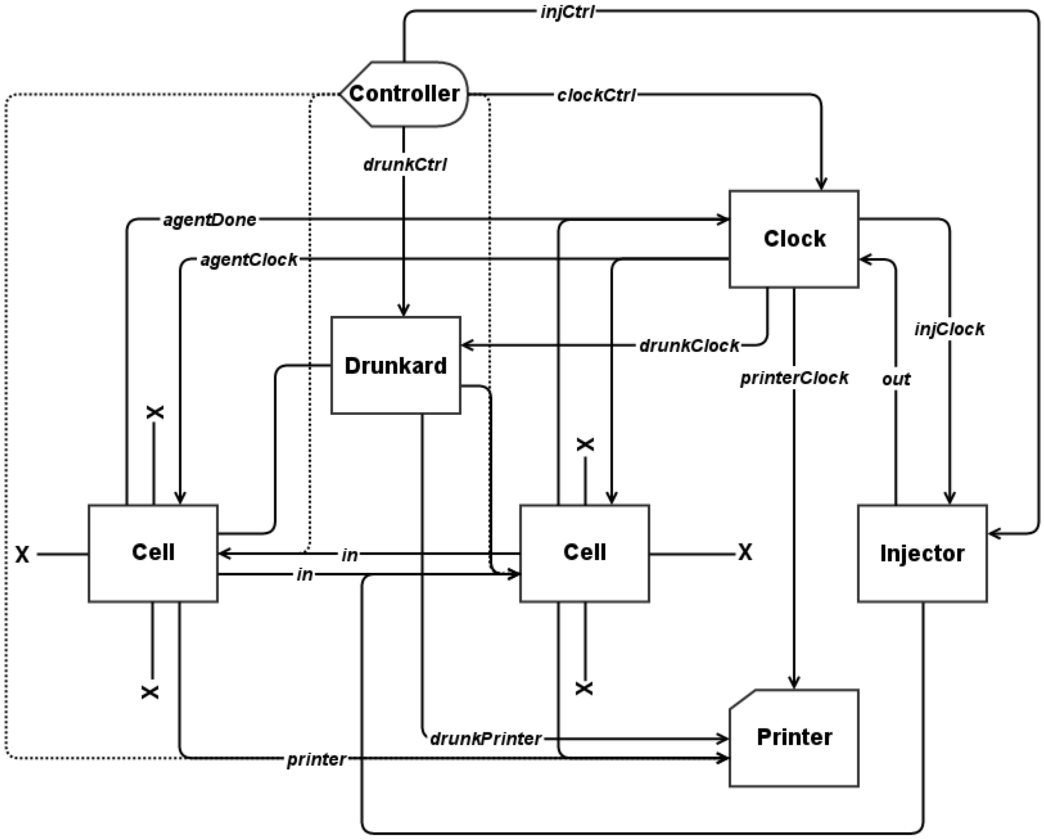

Figure 1: Communication between the processes of a pedestrian simulation. Here only two cells are shown, to keep the figure clearer. The dotted lines indicate that the controller only holds these channels for the purpose of closing them in the exit phase.

As can be seen from the figure the pedestrians themselves are not a process. These are instead data passed between the other processes on their respective channels. In general the communication of the implementation is as follows.

Cell A cell starts by waiting for messages from other cells, injectors and drunkards. When it receive such a message it will respond whether it has accepted the pedestrian, or drunk, in the message or not. It it only declines if it already has a pedestrian or a drunk. When a cell holds a pedestrian it will also wait for the clock to send a tick, after which it will try to send the pedestrian to on of its neighbors. If the cell has no neighbor in a direction it tries to send, it will instead let the pedestrian walk off the board and inform the clock that one less pedestrian exists. The cell also informs the printer each turn when it has a pedestrian. If the cell receives a drunk, it will not try to move it, but instead only decline on its input channel, until the drunkard process informs the cell that the drunk has moved, after which the cell is back to its original state.

Injector The injector waits on input from either the clock or the controller. If it receives a tick from the clock it will check whether it should inject a pedestrian, and if so try to do this. If the cell it tries to inject at declines, i.e. it is occupied, the injector simply skips the injections this turn. The injector then informs the clock whether or not an extra pedestrian now exists on the board. If the injector receives a input from the controller it will change its injection rate to the one it has received.

Clock The clock is in charge of keeping, pedestrians (i.e. cell containing a pedestrian), injectors, the drunk and the printer in lock-step with each other. This is achieved by the clock sending a tick on its pedestrian clock for each pedestrian that exists, a tick to the drunkard, a tick to each injector and tick to the printer. After a tick, the clock expects a response back when the processes are done with their turn. When initialized the clock knows of no pedestrians and thus no ticks are being sent to cells. When the injectors receive a tick, they respond back informing the clock whether they successfully have injected a pedestrian. If so the pedestrian count is increased for the next turn. When cells holding a pedestrian receives a tick, they after a ended turn respond back whether the pedestrian still exists. A player ceases to exists if it walks of the board. After each turn the clock check whether the controller is ready to send a command. If it is the command is executed, else the clock moves on to executing the next turn. The clock is artificially delayed using a sleep-command, to allow humans to actually perceive every step. The default is 500*ms* between steps, but this can be changed from the controller to anything between 0 and the maximum size of an integer.

Drunkard The drunkard controls the drunk on the board. The drunkard accepts messages from the controller, the clock and the cells. When the drunkard receives a message from the controller it tries to place the drunk at the specified coordinates, by sending to the correct cell. If successful a drunk will now be active. When a drunk is placed on the board the drunkard tries to move it in a random direction every time it receives a message from the clock. should the drunk walk off the board, it will be gone and a new one can be placed from the controller if so desired. When the drunkard receives a message from a cell it will respond, on the piggybacked response channel from the message, whether or not a drunk is within 3 cells of the cell in question. The drunkard uses the euclidean distance formula in two dimensions to calculate this.

Printer The printer is in charge of printing out the state of the board/sidewalk every time-step. Is is done, by the printer receiving messages from the clock, the cells and the drunkard. Every time a clock tick is received the state is printed to the console and the printer resets itself. When the printer receives messages from either the drunkard or the cells its stores these positions in its state so that it can print them on the next clock tick. Because the clock synchronizes with the other processes the printer is certain that when is receives the tick, it must have received all state updates from the drunkard or the cells. When printing to the screen the printer utilizes ANSI escape characters to overwrite the old state and thus keep a uncluttered, consistent console interface.

Controller The controller is the process which handles all input from the user. The controller can communicate with the clock, the injectors and the drunkard, to let them know when a user decision has been made. The controller as with the printer utilizes ANSI escape characters, and prints a simple console interface underneath the state of the board. When a user requests to exit the program the controller first informs the clock, effectively pausing the simulation. This ensures us that all processes are waiting on specific channels, which we then close from the controller. When the processes see that their channel is closed they return, thus closing themselves. The trickiest of the processes is the cell which can be waiting in many different places, however as we pause the simulation we now can be sure that the cell will either wait at its initial state, after it has gotten a drunk, or when it has a pedestrian, but has not yet received a clock tick. Finally the controller closes the channel to the clock and then returns, effectively exiting the complete simulation cleanly.

Sidewalk The sidewalk process simply binds all the other process together in a $X \times Y$ sized sidewalk with injectors at the requested locations. This process is thus used to instantiate a simulation with the requested amount of steps.

Source code

main.go

```go
package main

import (
	. "./sidewalk"
)

func main() {
	Sidewalk(5, 10, []Coordinate{Coordinate{2, 0}, Coordinate{2, 9}}, []Direction{Up, Down}, 200)
}
```

sidewalk/processes.go

```go
package sidewalk

import (
	"os"
	"strconv"
	"time"
)

/**
 * Used to synchronize all agents and keep them in lock-step with each other,
 * the injectors, the printer and the drunk.
 */
func Clock(in chan Pedestrian, agentClock chan bool, agentDone chan bool,
	injClock [](chan bool), printerClock chan bool, steps int,
	controller chan ClockRequest, drunkClock chan bool) {
	agents := 0
	wait := 500
	<-controller
	for step := 0; step <= steps; step++ {
		select {
		case cmd := <-controller:
```

```
            switch cmd.Cmd {
            case CmdPause:
                paused := true
                for paused {
                    cmd := <-controller
                    switch cmd.Cmd {
                    case CmdResume:
                        paused = false
                    case CmdTime:
                        wait = cmd.Arg[0]
                    default:
                        // Do nothing
                    }
                }
            case CmdTime:
                wait = cmd.Arg[0]
            case CmdExit:
                // Pause the clock until controller is closed after which, close
                ok := true
                for ok {
                    _, ok = <-controller
                }
                return
            default:
                // Do nothing
            }
        default:
            // do nothing
        }
        for i := 0; i < agents; i++ {
            // tick the clock for agents
            agentClock <- true
        }
        agentRemove := 0
        for i := 0; i < agents; i++ {
            // get response from agents
            if !<-agentDone {
                agentRemove++
            }
        }
        agents -= agentRemove
        drunkClock <- true
        for _, inj := range injClock {
            // tick every injector
            inj <- true
        }
        for i := 0; i < len(injClock); i++ {
            // as many times as there are injectors check input
            agent := <-in
            if agent.Ok {
                agents++
            }
        }
        // Now everone has moved and the printer can print
        printerClock <- true
        <-printerClock
        time.Sleep(time.Duration(wait) * time.Millisecond)
    }
}

/**
 * Injects players a specific cell every given step (default = 4).
 */
func Injector(clock chan bool, cell chan Msg, out chan Pedestrian, d Direction,
    ctrl chan int) {
    step := 4
    n := 0
```

```
    rsp := make(chan bool)
    for {
        select {
        case i, ok := <-ctrl:
            if !ok {
                return
            }
            step = i
        case <-clock:
            var agent Pedestrian
            if n%step == 0 { // create a new player
                agent = Pedestrian{d, true}
                cell <- Msg{rsp, agent, Regular}
                if !<-rsp {
                    agent.Ok = false
                }
            }
            out <- agent
            n++
        }
    }
}

/**
 * Prints the sidewalk every timestep. The printer expects that
 * ANSI escape characters work in the terminal, as these are used to
 * overwrite the board every step.
 */
func Printer(in chan Coordinate, drunkPrinter chan Coordinate, clock chan bool,
    x int, y int) {
    coords := makeCoords(x, y)
    drunkCoord := Coordinate{-1, -1}
    step := 0
    for {
        select {
        case c, ok := <-in:
            if !ok {
                return
            }
            coords[c.Y][c.X] = true // An agent is here in this turn
        case c := <-drunkPrinter:
            drunkCoord = c
        case <-clock:
            os.Stdout.WriteString("\033[s\033[0;0H\r\nStep: " +
                strconv.Itoa(step) + "\r\n")
            hline := ""
            for i := 0; i < x+2; i++ {
                hline += "="
            }
            hline += "\r\n"
            os.Stdout.WriteString(hline)
            for i, _ := range coords {
                line := "|"
                for j, _ := range coords[len(coords)-i-1] {
                    switch {
                    case drunkCoord.X == j && drunkCoord.Y == (len(coords)-i-1):
                        line += "@"
                    case coords[len(coords)-i-1][j]:
                        line += "o"
                    default:
                        line += " "
                    }
                }
                line += "|\n"
                os.Stdout.WriteString(line)

            }
```

```go
            os.Stdout.WriteString(hline)
            os.Stdout.WriteString("o = pedestrian, @ = drunk\r\n")
            os.Stdout.WriteString("\033[u")
            setCoords(false, &coords)
            step++
            clock <- true
        }
    }
}

/**
 * Represents a single cell on the sidewalk. The cells handle moving
 * pedestrians around. Whenever a cell holds a pedestrian it will try to move
 * it every timestep, until it suceeds.
 */
func Cell(in chan Msg, out [](chan Msg), printer chan Coordinate, coord Coordinate,
    agentClock chan bool, agentDone chan bool, drunk chan DrunkRequest) {
    var agent Pedestrian
    oldAgent := false
    rsp := make(chan bool)
    isDrunk := false
    for {
        done := false
        if !oldAgent {
            msg, ok := <-in
            if !ok {
                return
            }
            agent = msg.Agent
            msg.Rsp <- true
            printer <- coord
            if msg.Status == GotDrunk {
                // this cell is now permanently occupied by a drunk
                isDrunk = true
                for isDrunk {
                    msg, ok := <-in
                    if !ok {
                        return
                    }
                    msg.Rsp <- false
                    if msg.Status == GotSober {
                        isDrunk = false
                        done = true
                    }
                }
            }
        }
        for !done {
            select {
            case <-agentClock:
                d := directions(agent.Dirc)
                tries := 0
                drunk <- DrunkRequest{coord, d[tries], rsp}
                if <-rsp {
                    // Do not go the original direction if a drunk is nearby
                    // that way. We hate drunkards!
                    tries++
                }

                for tries < 3 && !done {
                    if out[d[tries]] == nil {
                        // Agent walked off the board
                        agentDone <- false
                        oldAgent = false
                        done = true
                    } else {
                        select {
```

```
                            case out[d[tries]] <- Msg{rsp, agent, Regular}:
                                if <-rsp {
                                    oldAgent = false
                                    done = true
                                    agentDone <- true
                                } else {
                                    tries++
                                }
                            case msg := <-in:
                                msg.Rsp <- false
                            }
                        }
                    }
                    if !done {
                        if !oldAgent {
                            oldAgent = true
                        } else {
                            printer <- coord
                        }
                        agentDone <- true
                    }
                case msg, ok := <-in:
                    if !ok {
                        return
                    }
                    msg.Rsp <- false
                }
            }
        }
}

/**
 * Controls the drunk (insertion and movement).
 * Cells query for the position of the drunk.
 * The process is also in charge of moving the drunk around.
 */
func Drunkard(in chan DrunkRequest, cells [][](chan Msg), ctrl chan Coordinate,
    printer chan Coordinate, clock chan bool, x int, y int) {
    rsp := make(chan bool)
    drunk := Coordinate{-1, -1}
    for {
        select {
        case req := <-in:
            req.Rsp <- drunkNearby(drunk, req.ReqCoord, x, y)
        case coord, ok := <-ctrl:
            if !ok {
                return
            }
            if !validCoord(drunk, x, y) {
                done := false
                cellMsg := Msg{rsp, Pedestrian{Up, false}, GotDrunk}
                for !done {
                    select {
                    case cells[coord.Y][coord.X] <- cellMsg:
                        done = true
                    case req := <-in:
                        req.Rsp <- drunkNearby(drunk, req.ReqCoord, x, y)
                    }
                }
                if <-rsp {
                    drunk = coord
                    printer <- coord
                    ctrl <- coord
                } else {
                    // Indicates that something went wrong
                    ctrl <- Coordinate{-1, -1}
                }
```

```
            } else {
                ctrl <- Coordinate{-1, -1}
            }
        case <-clock:
            if validCoord(drunk, x, y) {
                randomCoord := randomDirc(drunk, x, y)
                if validCoord(randomCoord, x, y) {
                    done := false
                    cellMsg := Msg{rsp, Pedestrian{Up, false}, GotDrunk}
                    for !done {
                        select {
                        case cells[randomCoord.Y][randomCoord.X] <- cellMsg:
                            done = true
                        case req := <-in:
                            req.Rsp <- drunkNearby(drunk, req.ReqCoord, x, y)
                        }
                    }
                    if <-rsp {
                        cellMsg := Msg{rsp, Pedestrian{Up, false}, GotSober}
                        done = false
                        for !done {
                            select {
                            case cells[drunk.Y][drunk.X] <- cellMsg:
                                <-rsp
                                done = true
                            case req := <-in:
                                req.Rsp <- drunkNearby(drunk, req.ReqCoord, x, y)
                            }
                        }
                        drunk = randomCoord
                        printer <- randomCoord
                    }
                } else {
                    cellMsg := Msg{rsp, Pedestrian{Up, false}, GotSober}
                    cells[drunk.Y][drunk.X] <- cellMsg
                    <-rsp
                    drunk = Coordinate{-1, -1}
                    printer <- drunk
                }
            }
        }
    }
}

/**
 * Creates a rectangular sidewalk with the given dimensions, the simulation
 * runs the specified number of steps. The injectors are added at the given
 * coordinates, and with the given directions.
 */
func Sidewalk(x int, y int, injs []Coordinate, ds []Direction, steps int) {
    agentClock := make(chan bool)
    agentDone := make(chan bool)
    cells := makeMsgChs(x, y)
    clockCtrl := make(chan ClockRequest)
    clockIn := make(chan Pedestrian)
    drunkClock := make(chan bool)
    drunkCtrl := make(chan Coordinate)
    drunkPrinter := make(chan Coordinate)
    drunkReq := make(chan DrunkRequest)
    injChs := make([](chan bool), len(injs))
    injCtrl := make([](chan int), len(injs))
    printer := make(chan Coordinate)
    printerClock := make(chan bool)
    for i := 0; i < y; i++ {
        for j := 0; j < x; j++ {
            out := getOutChs(j, i, x, y, cells)
            go Cell(cells[i][j], out, printer, Coordinate{j, i}, agentClock,
```

```
                agentDone, drunkReq)
        }
    }
    for i, _ := range injs {
        injChs[i] = make(chan bool)
        injCtrl[i] = make(chan int)
        injx := injs[i].X
        injy := injs[i].Y
        go Injector(injChs[i], cells[injy][injx], clockIn, ds[i], injCtrl[i])
    }
    go Printer(printer, drunkPrinter, printerClock, x, y)
    go Drunkard(drunkReq, cells, drunkCtrl, drunkPrinter, drunkClock, x, y)
    go Clock(clockIn, agentClock, agentDone, injChs, printerClock,
        steps, clockCtrl, drunkClock)
    Controller(clockCtrl, injCtrl, drunkCtrl, printer, cells, x, y)
}
```

sidewalk/controller.go

```
package sidewalk

import (
    "bufio"
    "os"
    "strconv"
    "strings"
)

/**
 * The shell used for interaction with the player. The shell expects ANSI
 * escape characters to work in the terminal, as these are used to overwrite
 * old output.
 */
func Controller(clockCtrl chan ClockRequest, injCtrl [](chan int),
    drunkCtrl chan Coordinate, printer chan Coordinate, cells [][](chan Msg),
    x int, y int) {
    os.Stdout.WriteString("\033[0;0H\033[J\033[" + strconv.Itoa(y+6) + ";0H")
    clockCtrl <- ClockRequest{CmdResume, nil}
    play := true
    exit := false
    os.Stdout.WriteString("\r\n" + HelpText + "\033[F")
    for !exit {
        os.Stdout.WriteString("\033[Kcmd> ")
        stdin := bufio.NewReader(os.Stdin)
        input, _ := stdin.ReadString('\n')
        cmd := strings.Split(strings.ToLower(strings.TrimSpace(input)), " ")
        os.Stdout.WriteString("\033[J")
        switch cmd[0] {
        case "pause":
            if play {
                clockCtrl <- ClockRequest{CmdPause, nil}
                os.Stdout.WriteString("Simulation paused")
                play = !play
            } else {
                os.Stdout.WriteString("Simulation already paused")
            }
        case "resume":
            if !play {
                clockCtrl <- ClockRequest{CmdResume, nil}
                os.Stdout.WriteString("Simulation resumed")
                play = !play
            } else {
                os.Stdout.WriteString("Simulation already running")
            }
        case "rate":
            i, _ := strconv.Atoi(cmd[1])
            if i > 0 && i < 11 {
                for _, inj := range injCtrl {
```

```go
                    inj <- i
                }
                os.Stdout.WriteString("Rate changed to " + cmd[1])
            } else {
                os.Stdout.WriteString("Rate but be between 1 and 10")
            }
        case "drunk":
            drunkX, _ := strconv.Atoi(cmd[1])
            drunkY, _ := strconv.Atoi(cmd[2])
            if drunkX >= 0 && drunkX < x && drunkY >= 0 && drunkY < y {
                drunkCtrl <- Coordinate{drunkX, drunkY}
                msg := <-drunkCtrl
                if msg.X == drunkX && msg.Y == drunkY {
                    os.Stdout.WriteString("Drunk placed at (" + cmd[1] + "," +
                        cmd[2] + ")")
                } else {
                    os.Stdout.WriteString("Drunk not placed, coordinates are" +
                        " occupied or drunk already exists on board")
                }
            } else {
                os.Stdout.WriteString("Drunk not placed, coordinates are invalid")
            }
        case "timestep":
            wait, _ := strconv.Atoi(cmd[1])
            if wait >= 0 {
                clockCtrl <- ClockRequest{CmdTime, []int{wait}}
                os.Stdout.WriteString("Time between steps changed")
            } else {
                os.Stdout.WriteString("Time between steps cannot be negative")
            }
        case "help":
            os.Stdout.WriteString(HelpText)
        case "exit":
            clockCtrl <- ClockRequest{CmdExit, nil}
            for _, i := range cells {
                for _, j := range i {
                    close(j)
                }
            }
            for _, i := range injCtrl {
                close(i)
            }
            close(drunkCtrl)
            close(printer)
            close(clockCtrl)
            return
        default:
            os.Stdout.WriteString("Unknown command")
        }
        os.Stdout.WriteString("\033[F")
    }
}

const HelpText = "The following commands can be called\r\n" +
    "pause\033[25Gpauses the simulation after the current step has executed\r\n" +
    "resume\033[25Gresumes the simulation if it has been paused\r\n" +
    "rate [ARG]\033[25Gsets the number of steps between injections of pedestrians\r\n" +
    "drunk [ARG_X] [ARG_Y]\033[25Ginsert a drunk at the given position\r\n" +
    "timestep [ARG]\033[25Gchanges the time (in milliseconds) " +
    "simulation waits between step\r\n" +
    "help\033[25Gshows this text\r\n" +
    "exit\033[25Gexits the simulation\033[7F"
```

sidewalk/utils.go

```go
package sidewalk

import (
```

```go
    "math"
    "math/rand"
)

/**
 * Creates a list of the given direction and its two orthogonal directions
 * the orthogonals order are randomized.
 */
func directions(d Direction) (l []Direction) {
    r := rand.Intn(2)
    switch d {
    case Up:
        if r > 0 {
            l = []Direction{Up, Left, Right}
        } else {
            l = []Direction{Up, Right, Left}
        }
    case Down:
        if r > 0 {
            l = []Direction{Down, Right, Left}
        } else {
            l = []Direction{Down, Right, Left}
        }
    case Left:
        if r > 0 {
            l = []Direction{Left, Down, Up}
        } else {
            l = []Direction{Left, Up, Down}
        }
    case Right:
        if r > 0 {
            l = []Direction{Right, Down, Up}
        } else {
            l = []Direction{Right, Up, Down}
        }
    }
    return
}

/**
 * Creates a slice of slices containing bools.
 */
func makeCoords(x int, y int) [][]bool {
    coords := make([][]bool, y)
    for i, _ := range coords {
        coords[i] = make([]bool, x)
        for j, _ := range coords[i] {
            coords[i][j] = false
        }
    }
    return coords
}

/**
 * Sets all values in the slice of slices to the specified value.
 */
func setCoords(val bool, coords *[][]bool) {
    for i, _ := range *coords {
        for j, _ := range (*coords)[i] {
            (*coords)[i][j] = val
        }
    }
}

/**
 * Creates a slice of slices containing message channels used by cells.
 */
```

```go
func makeMsgChs(x int, y int) [][](chan Msg) {
    chs := make([][](chan Msg), y)
    for i, _ := range chs {
        chs[i] = make([](chan Msg), x)
        for j, _ := range chs[i] {
            chs[i][j] = make(chan Msg)
        }
    }
    return chs
}

/**
 * Returns a list of the four channels corresponding to cells adjacent to
 * the cell at the position (j,i). If no cell is adjacent in a direction the
 * channel is nil.
 */
func getOutChs(j int, i int, x int, y int, chs [][](chan Msg)) [](chan Msg) {
    var left, right, up, down chan Msg = nil, nil, nil, nil
    if j == 0 {
        right = chs[i][j+1]
    } else if j == x-1 {
        left = chs[i][j-1]
    } else {
        left = chs[i][j-1]
        right = chs[i][j+1]
    }
    if i == 0 {
        up = chs[i+1][j]
    } else if i == y-1 {
        down = chs[i-1][j]
    } else {
        down = chs[i-1][j]
        up = chs[i+1][j]
    }
    return [](chan Msg){up, right, down, left}
}

/**
 * Checks whether a coordinate is valid (non-negative) and
 * within the specified bounds.
 */
func validCoord(coord Coordinate, x int, y int) bool {
    return coord.X >= 0 && coord.Y >= 0 && coord.X < x && coord.Y < y
}

func randomDirc(coord Coordinate, x int, y int) Coordinate {
    randomCoord := Coordinate{-1, -1}
    switch Direction(rand.Intn(4)) {
    case Up:
        if coord.Y < y-1 {
            randomCoord = Coordinate{coord.X, coord.Y + 1}
        }
    case Down:
        if coord.Y > 0 {
            randomCoord = Coordinate{coord.X, coord.Y - 1}
        }
    case Left:
        if coord.X > 0 {
            randomCoord = Coordinate{coord.X - 1, coord.Y}
        }
    case Right:
        if coord.X < x-1 {
            randomCoord = Coordinate{coord.X + 1, coord.Y}
        }
    }
    return randomCoord
}
```

```go

/**
 * Calculates whether a given drunk coordinate is within 3 cells.
 * The calculation is the euclidian distance formula in two dimensions.
 */
func drunkNearby(drunk Coordinate, reqCoord Coordinate, x int, y int) bool {
    if validCoord(drunk, x, y) {
        x := float64(drunk.X - reqCoord.X)
        y := float64(drunk.Y - reqCoord.Y)
        dist := math.Sqrt(math.Pow(x, 2) + math.Pow(y, 2))
        return dist <= 3
    } else {
        return false
    }
}
```

sidewalk/types.go

```go
package sidewalk

type Direction int
type State int
type CmdWord int

const (
    Up Direction = iota
    Right
    Down
    Left
)

const (
    GotDrunk State = iota
    GotSober
    Regular
)

const (
    CmdResume CmdWord = iota
    CmdPause
    CmdTime
    CmdExit
)

type Msg struct {
    Rsp    chan bool
    Agent  Pedestrian
    Status State
}

type Pedestrian struct {
    Dirc Direction
    Ok   bool
}

type Coordinate struct {
    X int
    Y int
}

type DrunkRequest struct {
    ReqCoord  Coordinate
    DrunkDirc Direction
    Rsp       chan bool
}

type ClockRequest struct {
    Cmd CmdWord
```

```
    Arg []int
}
```