Customer Agent

**Data**

List<AgentEvent> events;

AgentEvent event;

enum AgentEvent { none, gotHungry, followWaiter, seated, waiterCame, foodCame, doneEating, doneLeaving }

enum State { doingNothing, waitingInRestaurant, beingSeated, calledWatier, leaving, seated, eating, doneEating, ordered }

Host host;

Waiter waiter;

Menu menu;

**Messages**

gotHungry() {

events.add(gotHungry);

}

followMeToTable(m, w) {

events.add(followWaiter);

menu = m;

waiter = w;

}

whatWouldYouLike() {

events.add(waiterCame);

}

hereIsYourFood() {

events.add(foodCame);

}

**Scheduler**

if events.isEmpty, then

return false;

event = events.first();

if state == doingNothing and event == gotHungry, then

goToRestaurant();

state = waitingInRestaurant;

if state == waitingInRestaurant and event == followWaiter, then

state = beingSeated;

SitDown();

if state == beingSeated and event == seated, then

state = calledWaiter;

callWaiter();

if state == calledWaiter and event == waiterCame, then

state = ordered;

orderFood();

if state == ordered and event == foodCame, then

state = eating;

EatFood();

if state == eating and event == doneEating, then

state = leaving;

leaveTable();

**Action**

goToRestaurant() {

host.IWantFood(this);

}

SitDown() {

DoGoSeat(); // animation

}  
callWaiter() {

timer.schedule(waiter.readyToOrder(), 1000); // 10s to decide menu

}

orderFood() {

String choice = menu.getRandom();

waiter.hereIsMyChoice(choice);

}

EatFood() {

timer.schedule(new Task() {

public void run() {

events.add(doneEating);

stateChanged();

}

}, timeOfEating);

}

leaveTable() {

waiter.doneAndLeaving();

DoExitRestaurant(); // animation

}

Waiter

**Data**

class MyCustomer{

customer c,

int table,

CustomerState s;

String choice;

}

List<MyCustomer> customers;

enum CustomerState = { waiting, seated, askedToOrder, asked, ordered, waitingForFood, eating, doneAndLeaving }

Cook cook; // only one cook assumed

class MyFood{

String choice;

foodState s;

int table;

} // this might be developed later for the use of pay, but redundant for now

List<MyFood> foods;

enum FoodState = { toBeServed } // it is redundant for now

Host host;

**Message**

sitAtTable(Customer c, int table) {

customers.add( new MyCustomer(c, table, waiting) );

}

readyToOrder(Customer c) {

MyCustomer mc = customers.find(c);

mc.s = askedToOrder

}

hereIsMyChoice(Customer c,String choice) {

MyCustomer mc = customers.find(c);

mc.s = ordered;

mc.choice = choice;

}

orderIsReady(String choice, int table) {

foods.add(new MyFood(choice, table, toBeServed));

}

doneEatingAndLeaving(Customer c) {

MyCustomer mc = customers.find(c);

mc.s = doneAndLeaving();

}

**Scheduler**

if ∃f in foods ϶ f.s = toBeServed, then

serveFood(f);

if ∃c in customers ϶ c.s = ordered, then

placeOrder(c);

if ∃c in customers ϶ c.s = doneAndLeaving, then

cleanTable(c.table);

if ∃c in customers ϶ c.s = waiting, then

seatCustomer(c);

if ∃c in customers ϶ c.s = askedToOrder, then

takeOrder(c);

**Action**

seatCustomer(MyCustomer c) {

goBackToCounter(); // animation

c.c.followMe(this, new Menu()); //and tableNumber?

DoSeatCustomer(c); // animation

c.s = seated;

}

takeOrder(MyCustomer c) {

DoGoToTable(c, table); // animation

c.c.WhatWouldYouLike();

c.s = asked;

}

placeOrder(MyCustomer c) {

goBackToCounter(); // going off-screen animation

cook.hereIsAnOrder(this, c.choice, c.table);

c.s.waitingForFood;

}

serveFood(MyFood f) {

BringFoodToTable(f.table); // animation

∀c in customers ϶ c.table == f.table && c.choice == f.choice, then

c.c.hereIsYourFood();

c.s = eating;

foods.remove(f) // maybe changed to f.s = served; in later version

}

cleanTable(int table) {

CleanTable(table); // animation

∀c in customers ϶ c.table == table, then

customers.remove(c);

host.tableIsFree(table);

}

notifyHost() {

host.readyToWork(this);

}

Host

**Data**

List<Customer> waitingCustomers;

Collection<Table> tables;

class Table {

Customer occupiedBy;

int tableNumber;

}

List<MyWaiter> waiters;

class MyWaiter {

Waiter w;

WaiterState s;

}

WaiterState = { available };

**Message**

IWantFood(Customer c) {

waitingCustomers.add(c);

}

tableIsFree(int table) {

∀t in tables ϶ t.tableNumber == table

t.occupiedBy == null;

}

readyToWork(Waiter w) {

waiters.add(0, new MyWaiter(w, available));

/\* 0 is mechanism to select new waiter rather than waiters who were working \*/

}

**Scheduler**

if ∃t in tables ϶ t.occupiedBy == null and ∃c in waitingCustomers and

∃w in waiters ϶ w.s == available, then

takeCustomerToTable(c, table);

**Action**

takeCustomerToTable(Customer c, Table t, MyWaiter w) {

t.occupiedBy = c;  
 w.sitAtTable(c, t);

waitingCustomer.remove(c);

waiters.remove(w); // this makes sure the same waiter does not get

waiters.add(w); // overloaded by work when other waiters are free

}

Cook

**Data**

List<Order> orders;

class Order = {

Watier w,

String choice,

int table,

OrderState s;

}

enum OrderState = { pending, cooking, done }

Timer timer;

Map<String, Food> foods;

class Food {

String type,

int cookingTime,

}

**Message**

hereIsOrder(Waiter w, String choice, int table) {

orders.add(w, choice, table, pending);

}

foodDone(Order o) {

o.s = done;

}

**Scheduler**

if ∃o in orders ϶ o.s = done, then

plateIt(o);

if ∃o in orders ϶ o.s = pending, then

cookIt(o);

**Action**

cookIt(Order o) {

DoCooking(o); //animation

o.s = cooking;

timer.start( run(foodDone(o)), foods.get(o.choice).cookingTime);

}

plateIt(Order o) {

DoPlating(o); //animation

o.w.orderIsReady(o.choice, o.table);

orders.remove(o);

}

