Customer Agent

Data

List<AgentEvent> events;
AgentEvent event;
enum State { doingNothing, waitingInRestaurant, beingSeated, seated, eating, calledWatier, leaving, doneEating, ordered, checkRequested, paying, wantChange, payingDeferred, deciding }
enum AgentEvent { none, gotHungry, followWaiter, seated, waiterCame, foodCame, doneEating, doneLeaving, reorderRequested, checkCame, paid, changeCame, deferredPaymentRequested, kickedOut, restaurantFull}
Host host;
Waiter waiter;
Menu menu;
Check check;
Cashier cashier;
double cash;
boolean stayIfFull = true;
boolean isDecent = true;

Messages

```
gotHungry() {
        events.add(gotHungry);
}
followMe(Menu m, Waiter w) {
        events.add(followWaiter);
        menu = m;
        waiter = w;
}
whatWouldYouLike() {
        events.add(waiterCame);
}
hereIsYourFood() {
        events.add(foodCame);
}
pleaseOrderAgain(Menu menu) {
        this.menu = menu;
        events.add(reorderRequested);
        stateChanged();
}
hereIsCheck( Check c ) {
        this.check = c;
        events.add ( checkCame );
}
hereIsChange( double change ) {
        this.cash += change;//for now, not supported
        events.add(paid);
}
payNextTime() {
        events.add(paid);
}
pleasePayDeferredPayment(Check check) {
        this.check = check;
        events.add(deferredPaymentRequested);
}
getOut() {
        events.add(kickedOut);
}
restaurantIsFull() {
```

```
        events.add(restaurantFull);
}
```

Scheduler

```
if events.isEmpty, then
        return false;
event = events.first();

if state == doingNothing and event == gotHungry, then
        goToRestaurant();
        state = waitingInRestaurant;
if sate == waitingInRestaurant and event == deferredPaymentRequested, then
        state = payingDeferred;
        payDeferred();
if state == payingDeferred and event == kickedOut, then
        state = leaving;
        leaveRestaurant();
if state == waitingInRestaurant and event == restaurantFull, then
        state = deciding;
        stayOrLeave();
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 == reorderRequested, then
        state = calledWaiter;
        callWaiter(); // brings the customer back to ordering
if state == ordered and event == foodCame, then
        state = eating;
        EatFood();
if state == eating and event == doneEating, then
        state = checkRequested;
        requestCheck();
if state == checkRequested and event == checkCame, then
        state = paying;
        pay();
if state == paying and event == paid, then
```

```
        state = leaving;
        leaveTable();
if state == leaving and event == doneLeaving, then
        state = doingNothing
```

Action

```
goToRestaurant() {
	host.IWantFood(this);
}
SitDown() {
	DoGoSeat(); // animation
}
callWaiter() {
	//timer.schedule(waiter.readyToOrder(), 1000); // 10s to decide menu
	waiter.readyToOrder();
}
orderFood() {
	String choice = menu.getRandom();
	if (isDecent) { // is not decent will order even if he cannot afford
		choice = menu.getRandomAffordable(cash);
		if (choice == null) {
			leaveTable();
			state = leaving;
			return;
		}
	}
	waiter.hereIsMyChoice(choice);
}
EatFood() {
	timer.schedule(new Task() {
		public void run() {
			events.add(doneEating);
			stateChanged();
		}
	}, timeOfEating);
}
leaveTable() {
	waiter.leaving();
	DoExitRestaurant(); // animation
}
requestCheck() {
	waiter.doneEating(this);
}
pay() {
```

```
        DoGoToCashier(); // animation
        if ( check.getTotal() <= cash )
                cash = cash - check.getTotal();
                cashier.payment(check, check.getTotal(), this);
        else
                cashier.cannotPayBill(check, this);
}
payDeferred() {
        DoGoToCashier(); // animation
        if ( check.getTotal() <= cash )
                cash = cash - check.getTotal();
                cashier.paymentForDeferredPayment(check.getTotal(), this);
                DoGoBackToLine();//Animation
                state = waitingInRestaurant;
        else
                cashier.cannotPayBill(check, this);
}
leaveRestaurant() {
        DoExitRestaurant(); // animation
}
stayOrLeave() {
        host.iAm(stayIfFull, this); // stayIfFull is determined by hack
        if (stayIfFull) {
                state = waitingInRestaurant;
        }else {
                state = leaving;
                leaveRestaurant();
        }
}
```

Waiter

Data

```
class MyCustomer{
        customer c,
        int table,
        CustomerState s;
        String choice;
        Check check;
}
List<MyCustomer> customers;
enum CustomerState = { waiting, seated, askedToOrder, asked, ordered, waitingForFood,
eating, orderOut, doneEating, leaving, checkBeingIssued, checkIssued, checkDelivered }
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;
Cashier cashier;
```

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));
}
doneEating(Customer c) {
        MyCustomer mc = customers.find(c);
        mc.s = doneEating;
}
leaving(Customer c) {
        MyCustomer mc = customers.find(c);
        mc.s = leaving;
}
outOf(String choice, int table) {
        MyCustomer mc = customers.find(c)
                ∋ (mc.choice == choice && mc.table == table)
        mc.s = orderOut;
}
hereIsCheck(Check check, Customer c) {
        MyCustomer mc = customers.find(c);
        mc.check = check;
        mc.s = checkIssued;
}
```

Scheduler

```
if ∃c in customers ϶ c.s = orderOut, then
        requestReorder(c);
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 = doneEating, then
        requestCheck(c);
if ∃c in customers ϶ c.s = checkIssued, then
        bringCheckToCustomer(c);
if ∃c in customers ϶ c.s =leaving, 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

```
notifyHost() { host.readyToWork(this); }
requestBreak() { host.wantToGoOnBreak(this); }
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(); // animation
        cook.hereIsAnOrder(this, c.choice, c.table);
        c.s.waitingForFood;
}
serveFood(MyFood f) {
        DoGoToCook(); // animation
        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);
}
requestReorder(MyCustomer c) {
        DoGotoTable(c.table); // animation
        menu.removeItemFromMenu(c.choice);
        c.c.pleaseOrderAgain(menu);
        c.s= seated;
}
```

```
requestCheck(MyCustomer c) {
        cashier.produceCheck( c.c, c.choice, this );
        c.s = checkBeingIssued;
}
bringCheckToCustomer(MyCustomer c) {
        c.c.hereIsCheck(c.check);
        c.s = checkDelivered;
}
```

Host

Data

```
List<MyCustomer> customers;
class MyCusomter {
        Customer c;
        CustomerState s;
}
CustomerState = { wantFood, checking, checked, waiting, informed, kickOut }
Collection<Table> tables;
class Table {
        Customer occupiedBy;
        int tableNumber;
}
List<MyWaiter> waiters;
class MyWaiter {
        Waiter w;
        WaiterState s;
}
WaiterState = { available, breakRequested, onBreak };
Cashier cashier;
boolean isRestaurantOpen = false;
```

Message

```
IWantFood(Customer c) {
        customers.add(new MyCustomer(c, wantFood));
}
tableIsFree(int table) {
        ∀t in tables ϶ t.tableNumber == table
                t.occupiedBy == null;
}
readyToWork(Waiter w) {
        if ∃mw ϶ mw.w = w, then
                mw.s = available;
                waiters.remove(mw);
                waiters.add(0, mw);
        else
                waiters.add(0, new MyWaiter(w, available));
        /* 0 is mechanism to select new waiter rather than waiters who were working */
}
wantToGoOnBreak(Watier w) {
        MyWaiter mw = watiers.find(w);
        mw.s = breakRequested;
}
customerClear(Customer c, boolean clear) {
        MyCustomer mc = customers.find(c);
        if (clear) { mc.s = checked; }
        else {  mc.s = kickOut; }
}
iAm(boolean staying, Customer c) {
        if(staying) {
                state = waiting;
        }else {
                customers.remove(c);
        }
}
takeCustomers() {
        isRestaurantOpen = true;
}
```

Scheduler

```
if ! isRestaurantOpen, then
        return false;
if ∃w in waiters ϶ w.s == breakRequested, then
        acceptOrDenyBreak(w);
if ∃c in customers ϶ c.s == checked, then
        informAvailabiltiy(c);
if ∃t in tables ϶ t.occupiedBy == null and ∃c in customers ϶ c.s = waiting and
                        ∃w in waiters ϶ w.s == available, then
        takeCustomerToTable(c, table);
if ∃c in customers ϶ c.s = wantFood, then
        requestHistoryCheck(c);
if ∃c in customers ϶ c.s = kickOut, then
        kickOutCustomer(c);
```

Action

```
takeCustomerToTable(MyCustomer c, Table t, MyWaiter w) {
        t.occupiedBy = c.c;
        w.sitAtTable(c.c, t);
        customers.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
}
acceptOrDenyBreak(MyWaiter w) {
        count = 0;
        ∀mw in waiters ∋ mw.s = available
                count ++;
        if count > 0,
                w.s = onBreak;       //accepted
        else                         // notice I do not notify decision to waiter
                w.s = available;     //denied
requestHistoryCheck(MyCustomer c) {
        cashier.historyCheck(c.c);
        c.s = checking;
}
kickOutCustomer(MyCustomer c) {
        c.c.getOut();
        customers.remove(c);
}
informAvailability(MyCustomer c) {
        if ∃t in tables ∍ t.occupiedBy == null, then
                c.s = waiting; // if available, he's on line
        else
                c.s = informed;
                c.c.restaurantIsFull();
}
```

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,
        int amount,
        int low,
        int restockAmount,
        boolean isOrdered;
        int incomingStock;
}
List<MyMarket> markets;
class MyMarket {
        Market m;
        List<String> availableList; // initialized with all foods
        List<MarketOrder> orders;
}
class MarketOrder { // incoming order that market approved
        String choice;
        int quantity;
        DeliveryState s;
}
enum DeliveryState = { onDelivery, confirmed, delivered }
enum AgentState = { sleeping, atWork, openingRestaurant, initStocked, opened }
AgentState state = sleeping;
Host host;
```

Message

```
hereIsOrder(Waiter w, String choice, int table) {
        orders.add(w, choice, table, pending);
}
foodDone(Order o) {
        o.s = done;
}
weAreOutOf(String choice, Market m) {
        if ∃myM in markets ϶ myM.m == m, then
                myM.availableList.remove( choice );
}
deliveryScheduled(String choice, int quantity, Market m) {
        if ∃myM in markets ϶ myM.m == m, then
                myM.orders.add ( new MarketOrder ( choice, quantity, onDelivery ));
}
deliveryFor(String choice, Market m) {
        if ∃myM in markets ϶ myM.m == m, then
                myM.orders.get(choice).s = delivered;
}
openRestaurant(){
        state = atWork;
}
```

Scheduler

```
if state == atWork, then
        openRestaurant();
if state == initStocked, then
        tellHostToTakeCustomers();
if ∃o in orders ϶ o.s = done, then
        plateIt(o);
if ∃o in orders ϶ o.s = pending, then
        cookIt(o);
if ∃m in markets ϶ (! m.orders.isEmpty() && ∃o in m.orders ϶ o.s = onDelivery), then
        confirmOrder(m);
if ∃m in markets ϶ (! m.orders.isEmpty() && ∃o in m.orders ϶ o.s = delivered), then
        restock(m);
```

Action

```
cookIt(Order o) {
        Food f = foods.get(o.choice);
        if ( f.amount <= f.low && !f.isOrdered) {   //send out order to every market
                f.incomingOrder = 0;
                f.isOrdered = true;
                ∀m in markets ∋ m.availableOrder has f.choice
                        orderFor(f.choice, f.restockAmount);
        }else if ( f.amount == 0 ) {
                o.w.outOf(o.choice, o.table);
                orders.remove(o);
                return;
        }
        f.amount--;

        DoCooking(o); //animation
        o.s = cooking;
        timer.schedule( run(foodDone(o)), f.cookingTime);
}
plateIt(Order o) {
        DoPlating(o); //animation
        o.w.orderIsReady(o.choice, o.table);
        orders.remove(o);
}
confirmOrder(MyMarket m) {
        ∀o in m.orders ∋ o.s == onDelivery, then
                Food f = foods.get(o.choice);
                if (o.quantity <= f.restockAmount - f.incomingOrder)
                        f.incomingOrder += o.quantity;
                        o.s = confirmed;
                        m.m.confirmation(true, o.choice);
                else
                        m.orders.remove(o);
                        m.m.confirmation(false, o.choice);
}
restock(MyMarket m) {
        ∀o in m.orders ∋ o.s = delivered
                Food f = foods.get(o.choice);
                f.amount = f.amount + o.quantity;
```

```
                f.isOrdered = false;
                m.orders.remove(o);
        if state == openingRestaurant, then
                state = initStocked;
}
openRestaurant() {
        state = openingRestaurant;
        boolean nothingToRestock = true;
        ∀t in Menu.Type
                Food f = foods.get(t.toString());
                if ( f.amount <= f.low && !f.isOrdered) {   //send out order to every market
                        nothingToRestock = false;
                        f.incomingOrder = 0;
                        f.isOrdered = true;
                        ∀m in markets ∋ m.availableOrder has f.choice
                                orderFor(f.choice, f.restockAmount);
                }
        if (nothingToRestock) {
                tellHostToTakeCustomers();
        }
}
tellHostToTakeCustomers() {
        state = opened;
        host.takeCustomers();
}
```

Market

Data

```
List<Order> orders;
class Order = {
        String choice;
        int quantity;
        OrderState s;
}
OrderState = { orderReceived, preparing, toBeDelivered }
Map<String, Item> inventory;
class Item= {
        String type;
        int stockAmount;
        int deliveryTime;
}
Timer timer;
CookAgent cook;
```

Messages

```
orderFor (String choice, int quantity) {
        orders.add( new Order( choice, quantity, orderReceived ) );
}
confirmation (boolean approval, String choice) {
        if (approval) {
                if ∃o in orders ϶ o.s = preparing && o.choice = choice, then
                        o.s = toBeDelivered;
        }else {
                if ∃o in orders ϶ o.s = preparing && o.choice = choice, then
                        orders.remove(o);
        }
}
```

Scheduler

if ∃o in orders ϶ o.s = orderReceived, then
    processOrder(o);
if ∃o in orders ϶ o.s = toBeDelivered, then
    deliver(o);

Action

```
processOrder( Order o ) {
        Item i = inventory.get(o.choice);
        if ( i.stockAmount <= 0) {
                cook.weAreOutOf(o.choice);
                orders.remove(o);
        }else if ( i.stockAmount > 0) {
                o.s = preparing;
                if ( i.stockAmount < o.quantity )
                        cook.deliveryScheduled(o.choice, i.stockAmount);
                        i.stockAmount = i.stockAmount - i.stockAmount;
                else
                        cook.deliveryScheduled(o.choice, o.quantity);
                        i.stockAmount = i.stockAmount - o.quantity;
        }
}
deliver( Order o ) {
        orders.remove(o);
        timer.schedule (
                cook.deliveryFor(o.choice, this), Inventory.get(o.choice).deliveryTime );
}
```

Cashier

Data

```
List<CheckOrder> checkOrders;
class CheckOrder {
        Customer c;
        String choice;
        Waiter w;
        CheckOrderState s;
}
CheckOrderState = {
        requested,
}
List<Payment> payments;
class Payment {
        Check check;
        double cash;
        Customer c;
        PaymentState s;
}
PaymentState = {
        pending, paid, unpaidPending, unpaid, unpaidRevisit, unpaidProcessing,
        unpaidPaid, unpaidPendingAgain
}
Host host;
List<Customer> cleanCustomers;
```

Messages

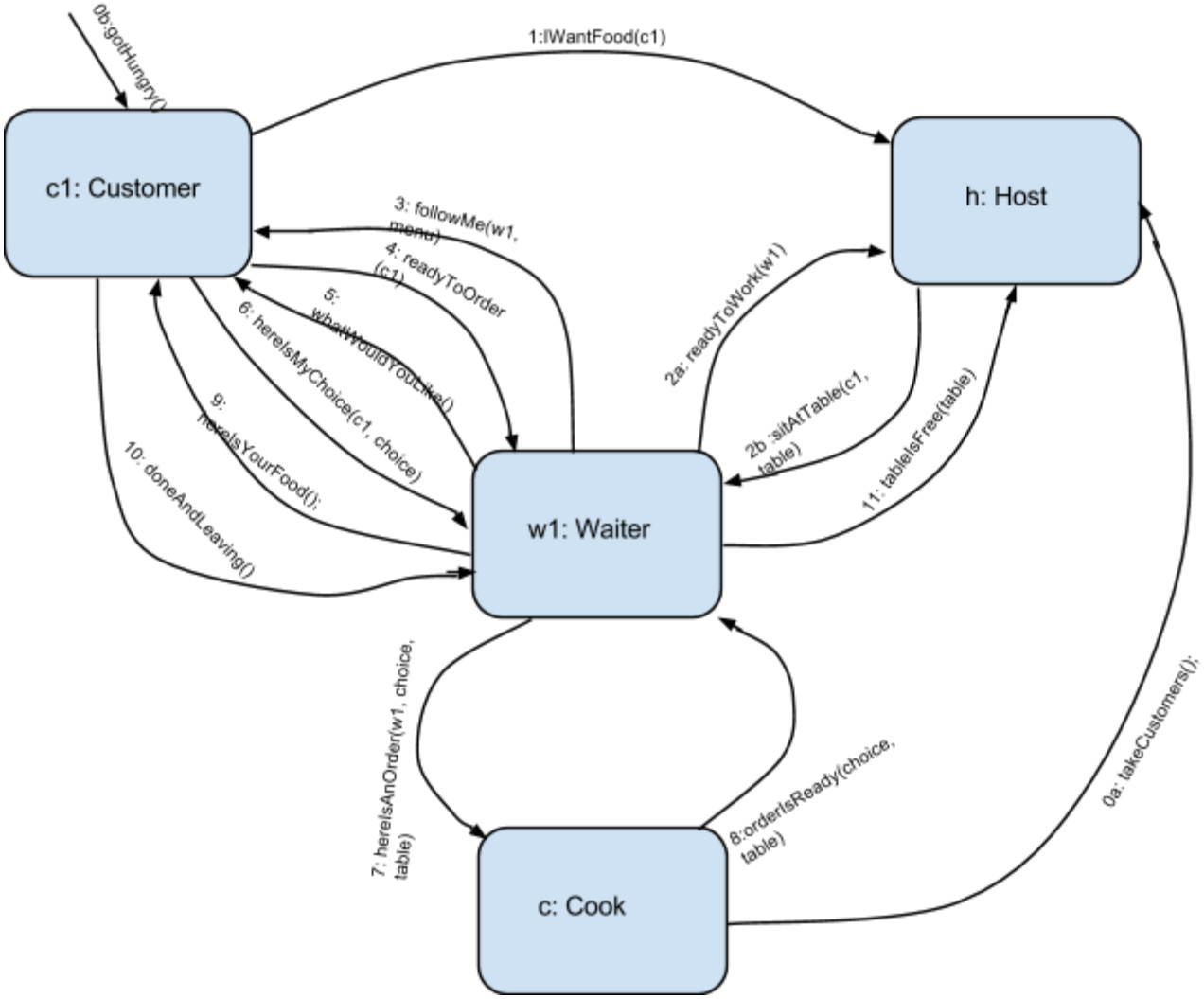
```
produceCheck(Customer c, String choice, Waiter w) {
        checkOrders.add ( new checkOrder(c, choice, w, requested) );
}
payment(Check check, double cash, Customer c) {
        payments.add( new Payment ( check, cash, c, pending ) );
}
cannotPayBill(Check check, Customer c) {
        if ∃p in payments ϶ p.c = c and p.s = unpaidProcessing, then
                p.s = unpaidPendingAgain;
        else
                payments.add( new Payment ( check, 0, c, unpaidPending) );
}
historyCheck(Customer c) {
        if ∃p in payments ϶ p.c = c and p.s = unpaid, then
                p.s = unpaidRevisit;
        else
                cleanCustomers.add(c);
}
paymentForDeferredPayment(Customer c, double cash) {
        if ∃p in payments ϶ p.c = c and p.s = unpaidProcessing, then
                p.cash = cash;
                p.s = unpaidPaid;
}
```

Scheduler

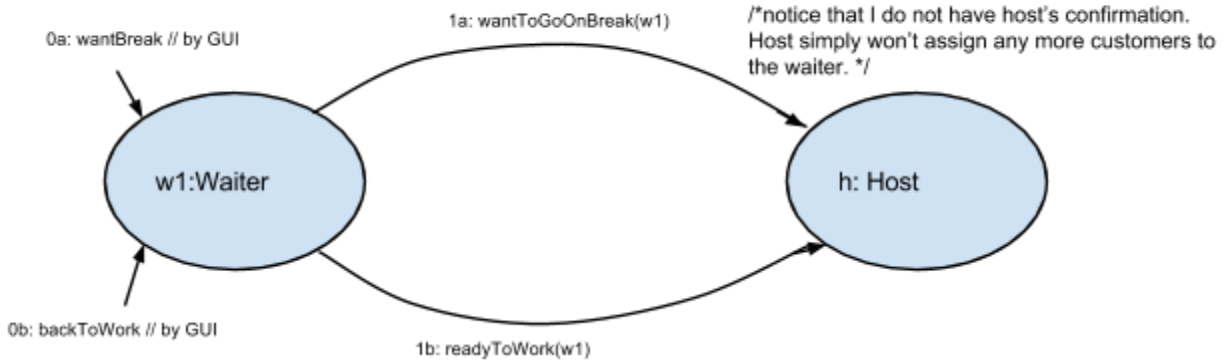
```
if !cleanCustomers.isEmpty, then
        clearCustomer(cleanCustomers.remove(0));
if ∃o in checkOrders ϶ o.s = requested, then
        deliverCheck(o);
if ∃p in payments ϶ p.s = pending, then
        processPayment(p);
if ∃p in payments ϶ p.s =unpaidPending || p.s = unpaidPendingAgain, then
        payNextVisit(p);
if ∃p in payments ϶ p.s = unpaidRevisit, then
        requestDeferredPayment(p);
if ∃p in payments ϶ p.c = c and p.s = unpaidPaid, then
        processDeferredPayment(p);
```

Actions

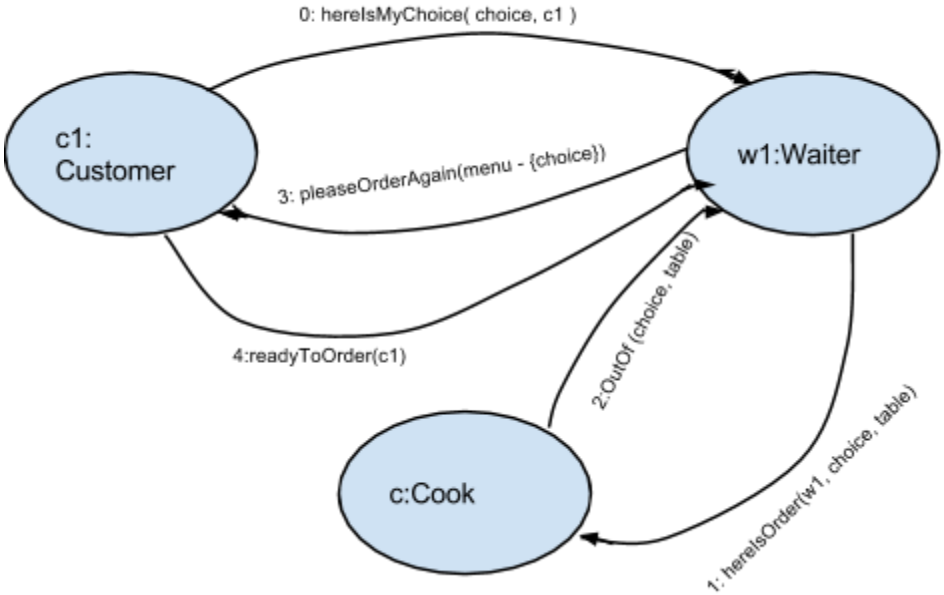
```
deliverCheck(checkOrder o) {
        check = new Check();
        check.addItem( o.choice );
        o.w.hereIsCheck( check, o.c );
        checkOrders.remove( o );
}
processPayment(Payment p) {
        double change = p.check.getTotal() - p.cash;
        p.c.hereIsChange(change);
        p.s = paid;
}
payNextVisit(Payment p) {
        if (p.s = unpaidPendingAgain)
                host.customerClear(p.c, false);
        p.s = unpaid;
        p.c.payNextTime();
}
requestDeferredPayment(Payment p) {
        p.s = unpaidProcessing;
        p.c.pleasePayDeferredPayment(p.check);
}
processDeferredPayment(Payment p) {
        double change = p.check.getTotal() - p.cash;
        p.c.hereIsChange(change);
        p.s = paid;
        host.customerClear(p.c, true);
}
clearCustomer(Customer c) {
        host.customerClear(c, true);
}
```


0b:gotHungry()
1:IWantFood(c1)
c1: Customer
h: Host
3: followMe(w1, menu)
4: readyToOrder (c1)
5: whatWouldYouLike()
6: hereIsMyChoice(c1, choice)
9: hereIsYourFood();
10: doneAndLeaving()
2a: readyToWork(w1)
2b :sitAtTable(c1, table)
11: tableIsFree(table)
w1: Waiter
7: hereIsAnOrder(w1, choice, table)
8:orderIsReady(choice, table)
c: Cook
0a: takeCustomers();

Waiter Going On Break and Coming Back To Work From Break

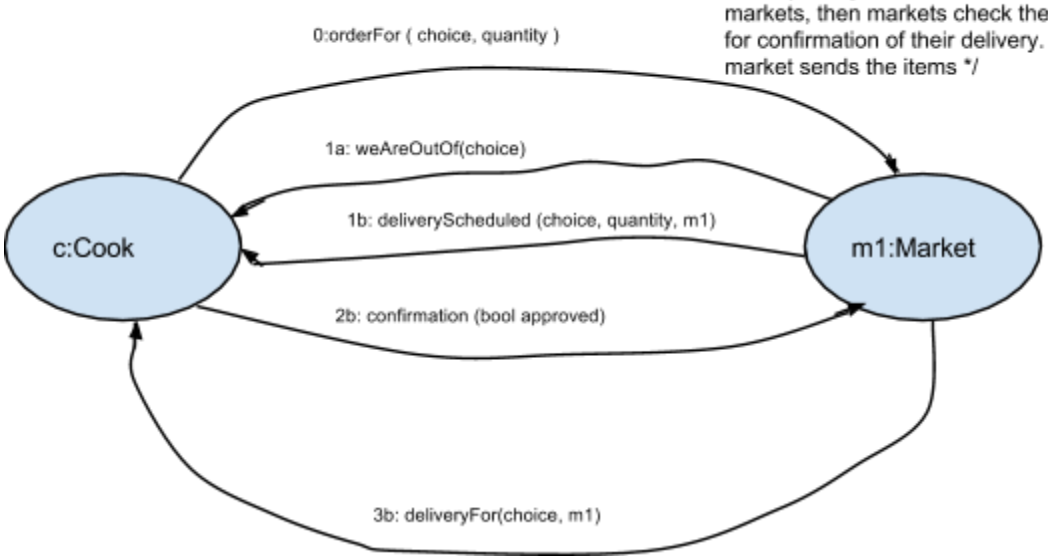


Out of Food Scenario

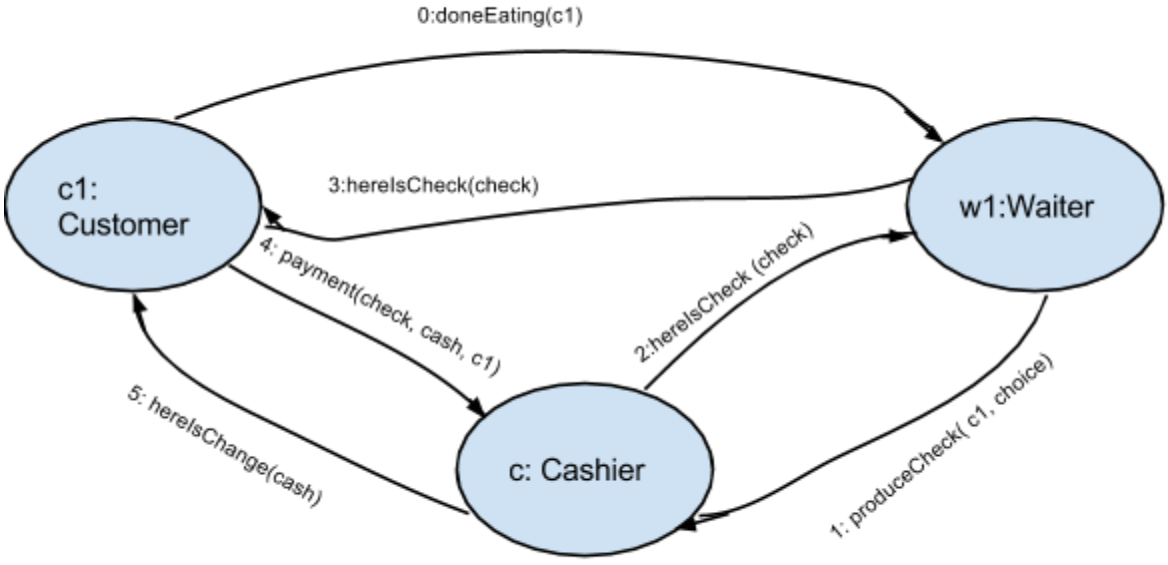


Market - Cook

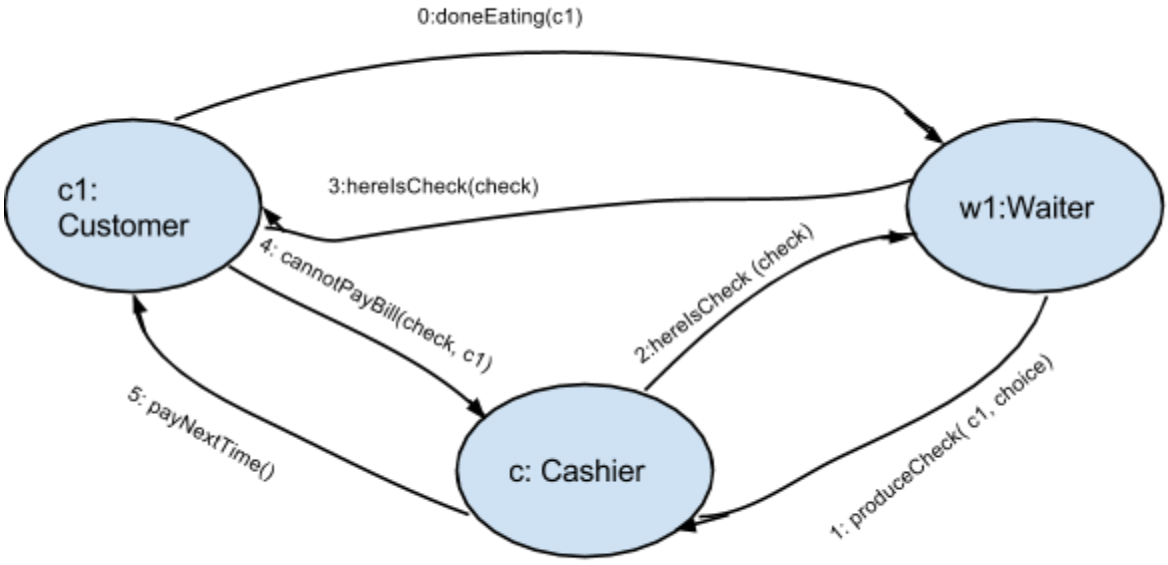
/* in my design, the cook sends out orders to all markets, then markets check their inventory, ask for confirmation of their delivery. If confirmed, then market sends the items */



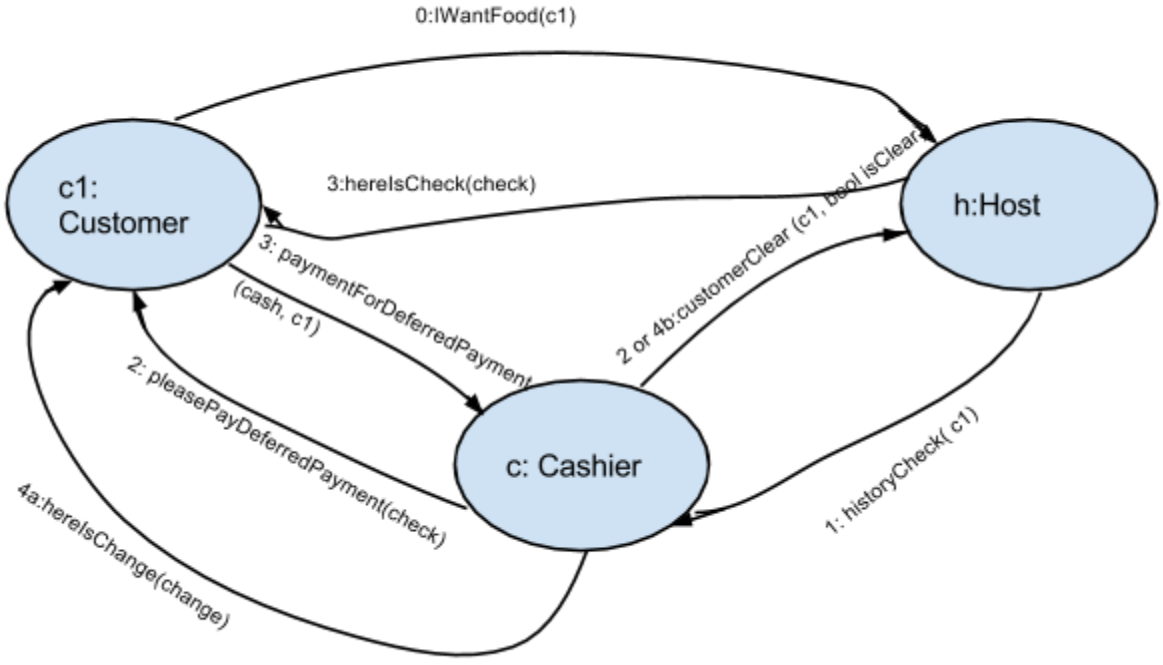
Cashier - Waiter - Customer : normative paying scenario



Customer does not have enough money to pay for his meal



Customer who hasn't paid bill is back to restaurant



restaurant is full, customer leaves or stays