Cashier

Data

```
List<Bill> bills;
class Bill {
    Waiter w;
    Customer c;
    String choice;
    double price, cash, change;
enum BillState {
 none, notComputed, ReturnedFromCustomer
}
    BillState state;
    IMenu menu;
```

Message

```
msg ComputeBill (Waiter w, Customer c, String choice)
    bills.add(new Bill(w, c, choice));

msgHereIsThePayment(Customer c, double check,
double cash) {
    if ∃ bill in bills -> bill.c == c
        bill.cash = cash;
        bill.price = check;
        bill.state = ReturnedFromCustomer;
```

Scheduler

```
if ∃ bill in bills -> bill.state = notComputed
    computeBill(bill);
if ∃ bill in bills -> bill.state = ReturnedFromCustomer
    makeChange(bill);
    bills.remove(bill);
```

Action

```
computeBill(Bill bill)
    bill.state = none;
    bill.w.msgHereIsTheCheck(
    bill.price, bill.customer);

makeChange(Bill bill)
    if (bill.cash - bill.price < 0)
        bill.c.msgYouDoNotHaveEnough
        Money(bill.price - bill.cash);
        return;
    bill.c.msgHereIsTheChange(bill.
    cash - bill.price);
```

Cook

Data: Map<String, Food> inventory;
bool lowInFood = false;
List<Order> orders;

class Order {
    Waiter w,
    String choice,
    int table,
    OrderState state;
enum OrderState:
    none, pending, cooking, cooked }
Timer timer;
List<Market> markets;

class Food {
    String choice.
    int amount, threshold, capacity;
    boolean ordered = false;
}

Scheduler

if ∃ order in orders -> order.state = Cooked
    returnOrder(order);
if ∃ order in orders -> order.state = pending
    cookOrder(order);

if (lowInFood)
    askforSupply();

Message

msgHavelsTheOrder(Waiter w, String choice, int table) {
    orders.add(new Order(w, choice, table, OrderState.pending));
}

msgDone(Order o){
    o.state = cooked; }

msgOrderFulfillment(Map<String, int> reply){
    for each order in inventory
        order.amount += reply.get(order).amount;
    if (order.amount < order.threshold)
        lowInFood = true;

Actions:

giveOrder(Order o) {
    o.w.msgOrderIsReady(o);
    orders.remove(o);
}

cookOrder(Order o) {
    Food f = inventory.get(o.choice)
    if (f.amount < f.threshold)
        lowInFood = true;
    if(f.amount = 0)
        order.w.msgFoodIsRunningOut(o);

```
return;
order.state= cooking;
f.count--;
this.schedule{msgDone(0)};
```

msg HereIsTheChange (double change){
event = changeArrived;
this.money = change;
msgYouDoNotHaveEnoughMoney(double debt){
this.money = 0;
this.debt += debt;
event = badLuck;

Customer.

Data: double money, check, debt;
String name;
Host h;
Waiter w;
Menu m;
State state;
Event event;
enum State {
DoingNothing, Waiting, Seated, deciding,
ReadyToOrder, GivenOrder, Eating, waitingToPay,
DoneEating, Leaving, waitingForChange
enum Event {
none, gotHungry, followHost, doneThinking,
orderFood, gotFood, doneEating, doneLeaving
noSeat, noFood, changeArrived, billArrived, badLuck};
Cashier c;

Scheduler
if (state = DoingNothing && event = gotHungry)
state = WaitingInRestaurant;
goToRestaurant();
if (state = WaitingInRestaurant && event = followHost)
state = Seated
thinkAboutMenu();
if (state = seated && event = doneThinking)
state = ReadyToOrder
AskWaiterToPickUpOrder();

Message.
msgNoSeat()
event = noSeat;
msgNoFood(Menu m)
event = noFood;
this.m = m; double
msgHereIsTheCheck(money, Cashier c)
event = billArrived;
this.c = c; this.check = money;
msgGotHungry(){
event = gotHungry;
}
msgFollowMe(Waiter w, Menu m){
this.w = w;
this.m = m;
event = followHost;
}
msgWhatWouldYouLike(){
event = orderFood;
}
msgHereIsYourFood(String choice){
event = gotFood;
}

Actions:
goToRestaurant(){
host.msg(WantFood();
}
thinkAboutMenu(){
int choice = random()%2;
if (!hasAffordableFood && choice == 1)
waiter.msgNoMoneyAndLeaving();
event = noMoney;
return;
int food_choice = random()%m.size();
if (!hasAffordableFood)
food_choice++ until price is affordable;
this.choice = food_choice;
}then.schedule(event = doneThinking);
AskWaiterToPickUpOrder(){
w.msgReadyToOrder(); }

```
if (state = ReadyToOrder && event = orderFood)
    state = GivenOrder;
    GiveOrder();
if (state = GivenOrder && event = gotFood)
    state = Eating;
    EatFood();
if (state = Eating && event = doneEating)
    state = waitingForBill;
    askForBill();
if (state = waitingInRestaurant && event = noSeat)
    state = decideToLeave
    ThinkAboutLeaving;
if (state = decideToLeave && event = toStay)
    state = waitingInRestaurant.
if (state = decideToLeave && event = toLeave
    state = Leaving
    leaveTable();
if (state = GivenOrder && event = noFood)
    state = seated.
    ReThinkAboutMenu();
if (state = waitingForBill && event = BillArrived)
    state = waitingForChange
    askForChange();
if (state = waitingForChange && event = changeArrived)
    state = leaving;
    leaveRestaurant();
if (state = waitingForChange && event = badLuck)
    state = leaving;
    leaveRestaurant;
```

```
GiveOrder(){
    w.msgHereIsTheChoice(choice);
}
EatFood(){
    timer(){
        event = doneEating;
    }
}
leaveTable(){
    w.msgDoneEating();
}
ThinkAboutLeaving(){
    if random() = 0
        event = toLeave;
    else
        event = toStay;}
ReThinkAboutMenu(){
    similar to ThinkAboutMenu
    except when !HasAffordableFood
    the customer just leaves
}
askForBill(){
    waiter.msgDoneEating(); }
askForChange(){
    waiter.msgLeavingRestaurant();
}
```

Host

Data:

class MyWaiter {
waiter w;
state(none, askingforBreak);
}

List<MyCustomer> waitingCustomers

List<Table> tables

List<MyWaiter> waiters

class Table {
Customer c;
int tableNumber;}

class MyCustomer {
Customer c;
state (waiting, deciding, staying);}

Scheduler

bool hasEmptyTable=false;
if ∃ table in tables ->
table.isUnOccupied
hasEmptyTable=true;
if ∃ customer in waitingCustomers ->
customer.state=waiting||staying
&& waiters.size!=0
seatCustomer(customer, table);
waitingCustomers.remove(customer);
if ∃ waiter in waiters -> waiter.state=
askforBreak && waiters.size()>1
waiter.w.msgBreakGranted();
waiters.remove(waiter);
if(!hasEmptyTable) && ∃ customer in
waitingCustomers -> customer.state=
waiting
customer.c.msgNoSeat();
customer.state=deciding;

Message:

msgIAmLeaving(Customer cust){
if ∃ customer in waitingCustomer ->
customer.c = cust;
waitingCustomers.remove(customer);
msgIWantToStay(Customer cust){
if ∃ customer in waitingCustomer ->
customer.c = cust;
customer.state=staying;

msgIWantFood(Customer cust){
waitingCustomers.add(cust);
}

msgTableIsFree(Customer cust, int t_num){
if ∃ table in tables ->
table.tableNumber = t_num
table.setUnOccupied();
}

msgWantToBreak(Waiter w){
if ∃ waiter in waiters -> waiter.w
waiter.state=askingforBreak;

msgWantToComeBack(Waiter w){
waiters.add(new MyWaiter(w));
}

Action:

seatCustomer(Customer c, Table t){
waiters.get(random).w.msg
SitAtTable(c, t.tableNumber);
table.setOccupied();

Market

Data

```
Cook c;
Map<String, Int> order;
Map<String, Int> inventory;
bool orderReceived, orderFinished;
   =false       =false
```

Message

```
msgHereIsTheOrder(Map order)
    this.order = order;
    orderReceived = true;
msgDone
    orderFinished = true;
```

Scheduler

```
if(orderReceived)
    timer(msgDone());
    orderReceived = false;
if (orderFinished)
    returnOrder(order);
    orderFinished = false;
```

Action

```
returnOrder(Map Order)
Map new_order;
for each o in orders.
    int storage = inventory.get(o);
    int requirement = Order.get(o);
    if(storage >= requirement)
        new_order.put(o, requirement)
        inventory.put(o, requirement)
    else
        new_order.put(o, storage)
        inventory.put(o, 0);
```

Waiter.

Data boolean backRequest, breakRequest;

List<MyCustomer> customers;

class MyCustomer {
  Customer c;
  int table;
  String food;
  CustomerState state;
  double check;
}

enum CustomerState
none, waiting, noMoney, readyToOrder,
orderGiven, orderReady, noFood,
finishedEating, checkComputed, leaving

Host h;
Cook c;
Cashier ca;

Scheduler
if ∃ c in customers → c.state = finishedEating
  computeBill(c);
if ∃ c in customers → c.state = waiting
  seatCustomer(c);
if ∃ c in customers → c.state = readyToOrder
  askForChoice(c);
if ∃ c in customers → c.state = orderGiven
  processOrder(c);
if ∃ c in customers → c.state = orderReady
  giveOrderToCustomer(c);

Message;

msgFoodRunsOut(String choice, int table){
  if ∃ customer in customers → customer.
  tableNumber = table
    customer.state = noFood;
msgHereIsTheCheck(double money, Customer c){
  if ∃ customer in customers → customer.c
    c.state = checkComputed;
    c.check = money;

msgSitAtTable((Customer cust, int table){
  customers.add(new MyCustomer(cust,
table, waiting));}
msgLeavingRestaurant(Customer c){
  if ∃ customer in customers → customer.c = c
  customer.state = leaving;
msgReadyToOrder((Customer cust){
  if(∃ customer in customers →
    customer.c = cust
  = customer.state = readyToOrder}

msgHereIsTheChoice((Customer cust, String choice){
  if(∃ customer in customers → customer.c =
  cust
  customer.food = choice;
  customer.state = orderGiven;}
msgOrderIsReady(Order o){
  if(∃ customer in customers → customer.table =
  o.table)
  customer.state = orderReady}
msgDoneEating((Customer cust){
  if(∃ customer in customers → customer.c =
  cust
    customer.state = finishedEating};
msgNoMoneyAndLeaving((Customer cust){
  if ∃ customer in customers → customer.c
  = cust
    customer.state = noMoney;}

```
if ∃c in customers -> c.state = noMoney
    clearCustomer(c);
if ∃c in customers -> c.state = checkComputed
    giveCheck(c);
if ∃ c in customers -> c.state = leaving
    clearCustomer(c);
if ∃ c in customers -> c.state = noFood
    giveNewMenu(c);
```

Action

```
seatCustomer(MyCustomer c) {
    c.state = none;
    c.c.msgFollowMe(menu);
}
askForChoice(MyCustomer c) {
    c.state = none;
    c.c.msgWhatWouldYouLike();
}
processOrder(MyCustomer c) {
    c.state = none;
    cook.msgHereIsTheOrder(c.choice, c.table)
}
clearCustomer(MyCustomer c) {
    host.msgTableIsFree(c.c, c.table);
    customers.remove(c);
}
giveOrderToCustomer(MyCustomer c) {
    c.state = none;
    c.c.msgHereIsYourFood(c.choice);
}
giveNewMenu(MyCustomer c) {
    c.state = none;
    c.c.msgNoFood(new Menu().remove
                  (c.choice)) }
computeBill(MyCustomer c) {
    c.state = none;
    ca.msgComputeBill(c.c, c.choice)}
giveCheck(MyCustomer c) {
    c.c.msgHereIsTheCheck(c.check, ca);
    c.state = none;
}
```