Michael Bock

CSCI 201

Professor Wilczynski

February 3, 2013

Restaurant v4.1 Design Document

Note: Interaction Diagrams Follow Below Agent Designs and include:

- Seating Customer Normative
- Customer will not wait
- Customer has less money than bill
- Ordering from Market
- Ordering Normative
- Cook is out of order choice
- Waiter break normative
- Waiter must wait for break

Cook Agent

Data

```
List<Order> orders;
Map<string, Food> inventory;
List<Market> markets; //list of markets to order from, in order of
most to least favorite
class Market {
      MarketAgent market;
      //maps food type to market's availability
      Map<Food, MarketStatus> status
}
enum MarketStatus {unknown, stocked, out}
enum OrderStatus {pending, cooking, cooked}
class Order {
      Waiter waiter;
      int tableNum;
      String choice;
      OrderStatus status;
}
class Food {
      String type;
      double cookTime;
      int amount;
}
```

Messages

```
msgHereIsAnOrder(Waiter w, string choice, int table){//sent by a
waiter
      orders.add(new Order(w,choice,table, pending));
      stateChanged();
}
msgFoodDone(Order o) { //sent by timer
      o.status = cooked;
      stateChanged();
}
msgFoodDelivery(Market m, Food f) {
      //if order is empty, change market status
      if f.amount==0 {
            Market m.status for Food f = out;
      }
      else {
            for food.type==f.type {
                  food.amount += f.amount;
            }
      }
      stateChanged();
}
```

Scheduler

```
if (there exists a food in inventory such that food.amount is low)
      {orderMore(food.type, amount) ; return true;}
if (there exists an order in orders such that order.status=pending)
      {cookOrder(order) ; return true;}
if (there exists an order in orders such that order.status=cooked)
      {plateOrder(order) ; return true;}
return false;
```

Actions

```
cookOrder(Order o) {
      Food f = inventory.get(o.choice);
      if (f.amount==0) {
            order.waiter.msgOutOfChoice(o.choice);
            return;
      }
      o.status = cooking;
      DoCooking (o); //animation
      startTimer {
            f.cookingTime, //how long before executing the run method
            run { msgFoodDone(o);}
      }
}
plateOrder(Order o) {
      DoPlatingFood(o.choice); //animation
      orders.remove(o);
      o.waiter.orderIsReady(o.choice, o.table);
}
orderMore(String type, int amount) {
```

```
      for market such that status=unknown or stocked for that food type
      {
            market.market.orderFood(type, amount);
            break;
      }
      if no stocked or unkown markets {
            //order from first market again to try
            Markets[0].market.orderFood(type, amount);
      }
}
```

Waiter Agent

Data

```
class MyCustomer {
      CustomerAgent cmr;
      int table;
      CustomerState state;
      String choice;
      Food food;
      boolean choiceIsOut; //false default
}
enum CustomerState = {NEED_SEATED, READY_TO_ORDER, ORDER_PENDING,
ORDER_READY, IS_DONE, NO_ACTION}
List<MyCustomer> customers;
class Menu {
      Map<Food, double> choices; //map food to prices
}
Menu menu;
enum BreakState = {none, wantBreak, mustWaitForBreak, canTake
onBreak};
BreakState breakState; //starts none
HostAgent host;
CookAgent cook;
String name; //waiter's name
```

Messages

```
msgSitCustomerAtTable(CustomerAgent customer, int tableNum) {
      //create new MyCustomer with state NEED_SEATED, tableNum
      stateChanged();
}
msgImReadyToOrder(CustomerAgent customer) {
      for MyCustomer c such that c.cmr==customer {
            c.state = READY_TO_ORDER;
            stateChanged();
      }
}
msgHereIsMyChoice(CustomerAgent customer, String choice) {
      for MyCustomer c such that c.cmr==customer {
            c.choice = choice;
```

```
            c.state = ORDER_PENDING;
            stateChanged();
      }
}
msgOrderIsReady(int tableNum, Food f) {
      for MyCustomer c such that c.cmr==customer {
            c.state = ORDER_READY;
            c.food = f;
            stateChanged();
      }
}
msgDoneEatingAndLeaving(CustomerAgent customer) {
      for MyCustomer c such that c.cmr==customer {
            c.state = IS_DONE;
            stateChanged();
      }
}
msgNotYet() { //from Host, cant take break yet
      breakState = mustWaitForBreak;
      stateChanged();
}
msgTakeBreak() {
      breakState = canTake;
      stateChanged();
}
msgOutOfChoice(String choice) {
      Order o of choice {
            o.choiceIsOut = true;
            stateChanged();
      }
}
```

Scheduler

```
if (breakState = onBreak) {
      return false;
}
if (there exists a MyCustomer c in customers such that
c.state=ORDER_READY) {
      giveFoodToCustomer(c); return true;
}
if (there exists a MyCustomer c in customers such that
c.state=IS_DONE) {
      clearTable(c); return true;
}
if (there exists a MyCustomer c in customers such that
c.state=NEED_SEATED) {
      seatCustomer(c); return true;
}
if (there exists a MyCustomer c in customers such that
c.state=ORDER_PENDING) {
      if (c.order.choiceIsOut==true) {
```

```
            c.order.choiceIsOut=false;
            takeNewOrder(menu-c.order.choice);
            return true;
      }
      giveOrderToCook(c);
      return true;
}
if (there exists a MyCustomer c in customers such that
c.state=READY_TO_ORDER) {
      takeOrder(c); return true;
}
if (breakState = wantBreak) {
      askForBreak();
      return true;
}
if (breakState = mustWaitForBreak) {
      return false;
}
if (breakState = canTake) [
      takeBreak();
      return true;
}
return false;
```

Actions

```
seatCustomer(MyCustomer c) {
      c.state = NO_ACTION;
      c.cmr.msgFollowMeToTable(this, new Menu());
      stateChanged();
}
takeOrder(MyCustomer c) {
      c.state = NO_ACTION;
      c.cmr.msgWhatWouldYouLike();
      stateChanged();
}
takeNewOrder(Menu newMenu, MyCustomer c) {
      c.state = READY_TO_ORDER;
      c.cmr.msgWhatWouldYouLikeNow(newMenu);
      stateChanged();
}
giveOrderToCook(MyCustomer c) {
      c.state = NO_ACTION;
      cook.msgHereIsAnOrder(this, c.tableNum, c.choice);
      stateChanged();
}
giveFoodToCustomer(MyCustomer c) {
      c.state = c.NO_ACTION;
      c.cmr.msgHereISYourFood(c.choice);
      stateChanged();
}
askForBreak() {
```

```
      host.msgCanITakeBreak(this);
      stateChanged();
}
takeBreak() {
      host.msgGoingOnBreak(this);
      breakState = onBreak;
      timer.schedule(new TimerTask(){public void run(){goOffBreak()}},
10000);
      stateChanged();
}
goOffBreak() {
      host.msgGettingOffBreak(this);
      breakState = none;
      stateChanged();
}
clearTable(MyCustomer c) {
      c.state = NO_ACTION;
      stateChanged();
}
```

Host Agent

Data

```
class Table {
      int tableNum;
      boolean occupied;
}
class MyWaiter {
      WaiterAgent wtr;
      boolean working; //starts true
      boolean wantsBreak; //starts false
}
Map<CustomerAgent,boolean> waitlist; //Collections.synchronizedList
//boolean in Map above signals if customer is waiting
List<MyWaiter> waiters; //Collections.synchronizedList
int nextWaiter; //starts at 0
int nTables;
Tables tables[];
String name; //host's name
```

Messages

```
msgIWantToEat(CustomerAgent c) {
      waitlist.add(c);
      stateChanged();
}
msgTableIsFree(int tableNum) {
      tables[tableNum].occupied = false;
      stateChanged();
}
```

```
msgCanITakeBreak(WaiterAgent w) {
      waiters.get(w).wantsBreak = true;
      stateChanged();
}
msgGoingOnBreak(WaiterAgent w) {
      waiters.get(w).working = false;
      stateChanged();
}
msgGoingOffBreak(WaiterAgent w) {
      waiters.get(w).working = true;
      waiters.get(w).wantsBreak = false;
      stateChanged();
}
//from customer who does not want to wait
msgThatIsTooLong(CustomerAgent c) {
      //do not sit customer, take off waitlist
      waitList.remove(find(c));
      stateChanged();
}
msgIWillWait(CustomerAgent c) {
      waitList.find(c).boolean = true; //set as waiting
      stateChaged();
}
```

Scheduler

```
if (!waitList.isEmpty() and !waiters.isEmpty()) {
      if (tables are all occupied) {
            tellCustomerThereIsWait(waitList.get(0));
            return true;
      }
      //find next working waiter: nextWaiter;
      for first un-occupied table {
            tellWaiterToSitCustomerAtTable(waiters.get(nextWaiter),
waitList.get(0), tableNum);
      return true;
}
if (there exists Waiter w in waiters such that w.wantsBreak=true) {
      //decide if waiter can take break
      if (can take break) {
            sendWaiterOnBreak(w);
            return true;
      }
      else {
            tellWaiterToWait(w);
            return true;
      }
}
return false;
```

Actions

```
tellWaiterToSitCustomerAtTable(MyWaiter w, CustomerAgent c, int
tableNum) {
      w.wtr.msgSitCustomerAtTable(c, tableNum);
      tables[tableNum].occupied = true;
      waitList.remove(c);
      increment nextWaiter;
      stateChanged();
}
sendWaiterOnBreak(WaiterAgent w) {
      w.msgTakeBreak();
      stateChanged();
}
tellWaiterToWait(WaiterAgent w) {
      w.msgNotYet();
      stateChanged();
}
tellCustomerThereIsWait(CustomerAgent c) {
      c.msgLongWait();
      stateChanged();
}
```

Customer Agent

Data

```
String name; //name of customer
int hungerLevel; //starts at 5
HostAgent host;
WaiterAgent waiter;
Restaurant restaurant; //gui?
Menu menu;
Timer timer;
boolean isHungry; //starts false
enum AgentState { DoingNothing, WaitingInRestaurant, SeatedWithMenu,
WaiterCalled, WaitingForFood, Eating, Paying, Working};
AgentState state; //starts DoingNothing
enum AgentEvent { gotHungry, beingSeated, decidedChoice,
waiterToTakeOrder, foodDelivered, doneEating, receivedBill,
recieveReceipt, mustGoWash, doneWorking};
List<AgentEvent> events = new ArrayList<AgentEvent>();
double cash;
Bill bill;
double hoursToWork; //if necessary
CashierAgent cashier;
```

Messages

```
msgFollowMeToTable(WaiterAgent w, Menu m) {
      this.menu = m;
      this.waiter = w;
```

```
      events.add(AgentEvent.beingSeated);
      stateChanged();
}
// Waiter sends this message to take the customer's order
msgDecided() {
      events.add(AgentEvent.decidedChoice);
      stateChanged();
}
msgWhatWouldYouLike() {
      events.add(AgentEvent.waiterToTakeOrder);
      stateChanged();
}
// Waiter sends this when the food is ready
msgHereIsYourFood(String choice) {
      events.add(AgentEvent.foodDelivered);
      stateChanged();
}
//Timer sends this message when finished eating
msgDoneEating() {
      events.add(AgentEvent.doneEating);
      stateChanged();
}
msgHereIsBill(Bill b) {
      this.bill = b;
      events.add(AgentEvent.receivedBill);
      stateChanged();
}
msgThanks(double change, Receipt receipt) {
      events.add(AgentEvent.receiveReceipt);
      stateChanged();
}
msgNotEnoughMoneyGetToWork(Receipt receipt, double hoursToWash) {
      events.add(AgentEvent.mustGoWash);
      stateChanged();
}
//from host
msgLongWait() {
      events.add(AgentEvent.thereIsWait);
      stateChanged();
}
```

Scheduler

//Agent FSM-based scheduler

```
if (events.isEmpty()) {
      return false;
}
AgentEvent event = events.pop(); //pop first event
//FSM begins here
if (state=DoingNothing) {
      if (event=gotHungry) {
            goingToRestaurant();
```

```
            state = WaitingInRestaurant();
            return true;
      }
}
if (state=WaitingInRestaurant) {
      if (event=thereIsWait) {
            //decide to wait or leave
            if (want to wait) { return false };
            if (want to leave) { leaveRestaurant(); return true; }
      }
      if (event=beingSeated) {
            makeMenuChoice();
            state = seatedWithMenu;
            return true;
      }
}
if (state == AgentState.SeatedWithMenu) {
      if (event == AgentEvent.decidedChoice) {
            callWaiter();
            state = AgentState.WaiterCalled;
            return true;
      }
}
if (state == AgentState.WaiterCalled) {
      if (event == AgentEvent.waiterToTakeOrder) {
            orderFood();
            state = AgentState.WaitingForFood;
            return true;
      }
}
if (state == AgentState.WaitingForFood) {
      if (event == AgentEvent.foodDelivered) {
            eatFood();
            state = AgentState.Eating;
            return true;
      }
}
if (state == AgentState.Eating) {
      if (event == AgentEvent.doneEating) {
            state = AgentState.Paying;
            return true;
      }
}
if (state == Paying) [
      if (event == receivedBill) {
            payBill();
            return true;
      }
      else if (event == receivedReceipt) {
            leaveRestaurant();
            state = AgentState.DoingNothing;
            return true;
```

```
      }
      else if (event == mustGoWash) {
            washDishes();
            state = AgentState.Working;
            return true;
      }
}
if (state=working) {
      if (event == doneWorking) {
            leaveRestauarant();
            state = AgentState.DoingNothing;
            return true;
      }
}
return false;
```

Actions

```
goingToRestaurant() {
      host.msgIWantToEat(this);
      stateChanged();
}
makeMenuChoice() {
      timer.schedule(new TimerTask(){public void run(){msgDecided()}},
3000);
      stateChanged();
}
callWaiter() {
      waiter.msgImReadyToOrder(this);
      stateChanged();
}
orderFood() {
      String choice = menu.choices[(int)(Math.random()*4];
      waiter.msgHereIsMyChoice(this, choice);
      stateChanged();
}
eatFood() {
      timer.schedule(new TimerTask() {public void
run(){msgDoneEating();}}, getHungerLevel*1000);//how long to eat
      stateChanged();
}
payBill() {
      cashier.msgPayment(bill, cash);
      stateChanged();
}
leaveRestaurant() {
      waiter.msgLeaving(this);
      isHungry = false;
      stateChanged();
}
washDishes() {
```

```
      timer.schedule(new TimerTask() {public void run()
{events.add(doneWorking);}}, bill.hoursToWork*1000);
      stateChanged();
}
```

Cashier Agent

Data

```
Menu menu;
class Menu {
      //maps food items to prices
      Map<String, double> items;
}
List<Bill> customerBills;
List<Bill> marketBills;
class Bill {
      double grandTotal;
      Food item;
      CustomerAgent cmr;
      MarketAgent mrkt;
      double amountReceived;
      double change;
      BillState state = unpaid;
      double hoursNeeded = 0; //for dishwashing if necessary
}
enum BillState = {unpaid, paidInFull, underPaid, receiptGiven};
class Receipt {
      //constructor takes in bill to print receipt
}
```

Messages

```
//payment from customer
msgPayment(CustomerAgent c, Bill b, Double cash) {
      for bill in customerBills such that b=bill {
            bill.amountReceived = cash;
            bill.cmr = c;
            if (bill.amountReceived < bill.grandTotal) {
                  bill.state = underPaid;
                  bill.hoursNeeded = calculate hours for dishwashing;
                  stateChanged();
            }
            else {
                  bill.state = paidInFull;
                  bill.change = b.amountReceived - b.grandTotal;
                  stateChanged();
            }
      }
}
//bill from market
msgBill(MarketAgent m, Bill b) {
```

```
      new Bill() with mrkt = m from b;
      marketBills.add(b);
      stateChanged();
}
```

Scheduler

```
if (there exists bill b in customerBills such that b.state=paidInFull)
{
      sendReceipt(b);
      return true;
}
if (there exists bill b in customerBills such that b.state=underPaid){
      assignCustomerToWork(b);
      return true;
}
if (there exists bill b in marketBills such that b.state=unpaid) {
      payMarketBill(b);
      return true;
}
```

Actions

```
sendReceipt(Bill b) {
      receipt = new Receipt(b);
      b.cmr.msgThanks(b.change, receipt);
      b.state = receiptGivent;
}
assignCustomerToWork(Bill b) {
      receipt = new Receipt(b);
      b.cmr.msgNotEnoughMoneyGetToWork(receipt,b.hoursNeeded);
      b.state = receiptGiven;
}
payMarketBill(Bill b) {
      b.mrkt.msgPay(b, b.grandTotal);
      b.state = paidInFull;
}
```

Market Agent

Data

```
List<Order> orders;
class Bill {
      double grandTotal;
      Food item;
      double amountReceived; //default=upaid
}
class Order {
      int amount;
      Food type;
```

```
      OrderState state; //default=received
      Bill bill;
      CashierAgent csr;
      CookAgent cook;
}
enum OrderStae = {received, billed, completed, unfulfilled};
Map<Food,int> inventory; //map food to amount in stock
```

Messages

```
msgPay(Bill b, double cash) {
      for bill in bills such that b==bill {
            b.amountReceived = cash;
            b.state = paid;
            stateChanged();
      }
}
msgOrderFood(Food type, int amount) {
      orders.add(type, amount);
      stateChanged();
}
```

Scheduler

```
if (there exists o in orders such that o.state=received and inventory
has inventory.get(o.type)>=o.amount) {
      sendBill(o);
      return true;
}
else if (there exists o in orders such that o.state=received and
inventory.get(o.type)<o.amount) {
      declineOrder(o);
      return true;
}
if (there exists o in order such that o.state=billed) {
      sendOrder(o);
      return true;
}
return false;
```

Actions

```
sendBill(Order o) {
      //calculate bill for order o
      o.csr.msgBill(o.bill);
      o.state = billed;
}
declineOrder(Order o) {
      o.cook.msgFoodDelivery(o.type, 0); //food delivery with 0 amount
      o.state = unfulfilled;
}
sendOrder(Order o) {
      o.cook.msgFoodDelivery(o.type, o.amount);
      o.state = completed;
```

```
}
```

1 getHungry()

c:Customer

2 I'mHungry()

Host

3 Seat Customer (c, table)

Follow Me (menu)

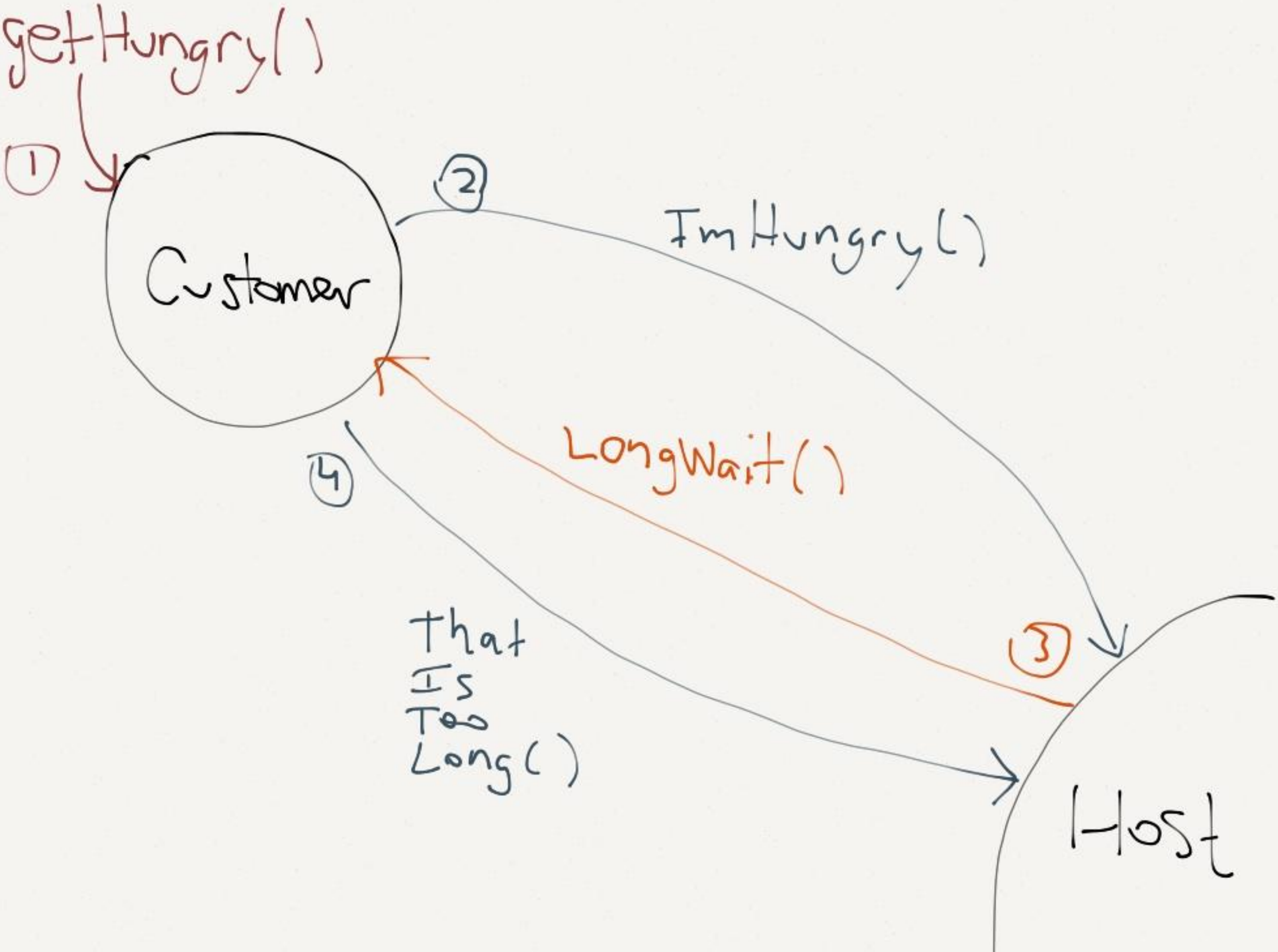
5a Ready to Order()

What Would You Like()

4

5b

Waiter


getHungry()
1
Customer
2
ImHungry()
LongWait()
3
4
That
Is
Too
Long()
Host

Here Is Bill (bill) (1)

Customer

Waiter

Payment
(bill, cash)

Not
Enough
(receipt,
hours To Wash)

Will
Work
For (hours)

Cashier

Cook

Market

① orderFood(type, amount)

② food Delivery(self, food)

Bill(bill)

Pay()

③

Cashier

Customer

Waiter

Cashier

Cook

6 HereIsMyChoice(choice)

9 HereIsFood(choice)

10 DoneEating()

11

12 HereIsBill(bill)

Thanks (change, receipt)

Payment(bill, cash)

HereIsBill(bill)

9b

9a NeedBill(customer, choice)

7 HereIsOrder(choice)

DoneEating(choice)

8

Customer

1

Here Is Choice (choice)

Waiter

What Would You Like Instead (next choice)

4

2

Here Is Order (order)

Out of Choice (order)

3

Cook

Waiter

1

Can I Take Break? (this)

3

Take Break()

Going On. Break()

2

Host

Waiter

1

Can I Take Break? (this)

4

Not Yet()

Take Break()

Going On Break()

2

3

Host