

Investigation and Questions for lab. 2

How the Restaurant Agents Work

(csci201 - Wilczynski)

The base class Agent.java

1. What happens when an agent constructor is called?
 - a. The specific agent constructor is called (the base agent has a default constructor; the base agent has a semaphore instance called StateChange, which is initialized with value 1. We'll discuss what StateChange is all about later.)
2. How does the agent thread get created and started? What gets executed?
 - a. The caller of the agent constructor calls startThread() on the agent instance, which in turn constructs the agent thread and makes the java call agentThread.start().
 - b. When a thread's start() method is called, the thread is actually created in the Java Virtual Machine and a program counter is started on its run() method. The run method is analogous to the main() method for the original process' first thread.
3. What does the run() method do?
 - a. Mostly, it is a loop that calls pickAndExecuteAnAction()
4. The professor talks about the agent scheduler. Where is it?
 - a. pickAndExecuteAnAction() is the scheduler, but that is too hard to write.
5. In the base Agent, the scheduler has the following signature:
protected abstract boolean pickAndExecuteAnAction();
What does all that mean?
 - a. protected means only it or a subclass can call it
 - b. abstract means a class that extends Agent must implement it.
 - c. boolean means the scheduler returns a boolean.
6. In the professor's design, his actions did not return anything. What's going on?
 - a. In the prof's design, an agent keeps calling the scheduler even if no rule applied. Nothing terrible happens, but if implemented this way it is a waste of the cpu's precious time.
 - b. So, notice, for example, in the Cook's scheduler, that the rules end with Return F; This means that nothing was found to do, so wouldn't it be a good idea to go to sleep and let some other agent run? Yes, it would. Notice the last effective line says: while (pickAndExecuteAnAction()); This while-clause keeps executing until pickAndExecuteAnAction() returns false.

7. Then what happens?
 - a. That's what the semaphore `StateChange` is all about.
 - b. Notice near the top of the loop in `run()` is the line of code: `stateChange.acquire()`; If the semaphore, `stateChange` is empty (more about that later), then the agent goes to sleep and some other agent will run.
8. How does the semaphore get full?
 - a. When the agent receives a message, the message reception method calls `stateChanged()`. This fills the semaphore and will wake up the agent.
9. The professor didn't call `stateChanged()` in his design. Why not?
 - a. Because all this is implementation details that don't impact the design.
 - b. Notice also that the prof. wrote scheduler rules as follows: `if then action1()`; In the code the rules are written: `if ... then {action1(); return T}` That keeps the while (`pickAndExecuteAnAction()`); running. Only when no action is selected by the scheduler, does the scheduler return `F` and the semaphore `StateChange` comes into play.

The Cook Agent

1. Why in `msgHereIsAnOrder()` does the code put the information into the list `orders` instead of just storing the data into some globals like `currentWaiter`, `currentTable`, `currentChoice`?
 - a. Because the cook gets messages like this from many waiters and storing them into globals will clobber the variables. [Be sure to understand clobbering because it is a very important and common issue.]
2. Suppose I did use globals, will clobbering always make it fail?
 - a. No, under the right timing it could work. Suppose the cook acted on the message before a new message was sent. Then nothing is clobbered. However, this depends on lucky timing, a problem called "race conditions" in concurrent programming.
3. The animation action `DoCooking()` has a call to `timer.schedule()`, which takes two parameters, a call-back object and a time to wait before calling the call-back. How is that first parameter formed?
 - a. In Java, only object instances can be passed as parameters (in some languages you can actually use function pointers as parameters). In Java an object instance must be created from a class definition. Well, here is a case where we only need this one instance once; why create a whole class definition for it? Java gives us a mechanism called Anonymous Inner Class for just this purpose. The method `timer.schedule()` requires an object of type `TimerTask`, which is an interface that must have a `run()` method, which we define right here. How convenient!!
4. Could I have done this more classically? [Please laugh at this pun.]
 - a. Yes. You could defined the following class:

```
private class XYZ implements TimerTask {
    Order o;
    public void run() {order.status = Status.done; stateChanged(); }
}
```

Then in the DoCooking() method:

```
TimerTask tt = new XYZ();
tt.o=order;
timer.schedule (tt, (int)(inventory.get(order.choice).cookTime*1000));
```

Isn't it easier to use the anonymous inner class?

5. Why is the parameter to DoCooking() typed as *final*?
 - a. Final means the method cannot change the parameter. It has to do with anonymous inner classes.

The Waiter Agent

1. The waiter agent has a list of customers of type MyCustomer similar to the cook's orders list. In the MyCustomer class is a state variable for keeping track of the customer. One of the states is NO_ACTION. Can you explain what that is about?
 - a. Often you have taken care of the customer and are waiting for the customer or the cook to do something. In that case you make the state NO_ACTION so that no scheduler rules apply.
2. In many of the waiter message reception calls, one of the parameters is the customer agent pointer and in the code there is a search to find the appropriate MyCustomer instance. How will that look in the Waiter design?
 - a. Just like there is no search code in the scheduler, there is no search code in message reception design. The design pseudo-code will look as follows:


```
msg1(Customer cust, ...) {
    if there exists c in customers such that c.cmr=cust then {do stuff to c} }
```
3. What if there is NO customer in the list? What does the "design" do?
 - a. The "design" does nothing. Something went wrong during the implementation execution. That is an implementation issue, NOT a design issue. Remember, we are trying to keep the design easy to understand. That's why we try to keep implementation issues out of the design.

4. There is a lot of code to support how the waiter moves around the animation. All that A* stuff is quite confusing. How come none of it appears in the design?
 - a. It's in the design but encapsulation by DoXXX animation calls. For example: DoGiveFoodToCustomer(customer); is such a call. This call encapsulate all the A* handling so that the "design" is easy to understand.

The Host Agent

1. How does the Host find his waiters? How does a waiter find its host?
 - a. By a hack. Every time a waiter is instantiated, two things happen (1) its constructor identifies the Host agent; and (2) it sends the host a setWaiter() message. No point specifying details like this in the design.
2. How does the host pick a waiter to assign a new customer?
 - a. That's not specified in the design so the student gets to decide. In the code a simple hack: `nextWaiter = (nextWaiter+1)%waiters.size();` "rotates" through the waiter indices.
3. What are Waiter indices?
 - a. The host's *waiters* are typed as List but instantiated as an ArrayList. In an arraylist, an index is used to address the nth waiter as *waiters.get(n)*
4. What is all that *synchronized* stuff in the code?
 - a. We'll discuss that later...

The Customer Agent

1. The customer agent looks different from the other agents. Why?
 - a. Everything it does is in lock-step. First he gets seated, then he orders, then he eats, then he leaves. Nothing interrupts that chain of events. This kind of control structure is easily implement using a finite-state machine.
 - b. Notice that every rule in the scheduler is simply of the form:


```

          if state=S1 then {
            if event=e1 then {state=someOtherState; action1();}
            else if state=S2 {
              if event=e2 then {state=someOtherState; action2();}
            }
            else ...
          
```

 Pretty simple. Many students try to write their agent schedulers like this, and it's almost always wrong. And, in fact, it is wrong here. This is an incorrect implementation of a finite state machine.
2. What is the proper state machine formalism?
 - a. A state machine transition function is a mapping:

$$f: \text{State} \times \text{event} \rightarrow \text{State} \times \text{OutputAction}$$

3. How should a state machine look inside an agent?
 - a. The messages should set events and scheduler should test states AND events. Something like this in message reception:

```
msg1(...) { ...; events.add(event1); statechanged();}
```

and something like this in the scheduler:

```
if state=S1 and event1 then { state=S2; action1(...)};
else if state=S1 and event2 then { state=S3; action2(...)};
else if ...
```

...

4. What might destroy the fsm quality of the customer agent.
 - a. Not really. What if the customer specification changed so that he got messages during the meal, or had to go to the bathroom, or went to the restaurant's bar for a drink during the meal. This simple state-machine work wouldn't work. The agent would have to be redesigned and reimplemented to be more like the Waiter. The state machine formalism is a simple mechanism that is appropriate in limited situations--like some in the factory.

5. Suppose the host asked the customer if he want to wait. Would that still fit the fsm framework?

- a. Yes. The customer is currently is state= WaitingInRestaurant. The doYouWantToWait() message could set a new event; call it wantToWait. Then the scheduler would like like this:

```
if (state == AgentState.WaitingInRestaurant) {
  if (event == AgentEvent.beingSeated)      {
    makeMenuChoice();
    state = AgentState.SeatedWithMenu;
    return true;
  }
  if (event == AgentEvent.wantToWait){
    doIWantToWait(); //decides on leaving or not
    state = ...; //some state which depends on the result
    return true;
  }
}
```