Agent Design for SimCity201

David Wilczynski

In SimCity201 we have person agents doing person things and then when it is eating in a restaurant doing customer things. Also a person agent works and has jobs like being a waiter or bank teller. Then there are questions about changing jobs. How should all this be modeled?

My Own Naïve Solution

At first I thought agents would have the following design:

```
class PersonAgent extends Agent implements Person {…}
class WaiterAgent extends PersonAgent implements Waiter{…}
class BankTellerAgent  extends PersonAgent implements BankTeller{…}
```

and so forth. All agents would have PersonAgent messages and behavior and then have the added behavior of the subclass. In this design PersonAgent, of course, has a scheduler:

```
void boolean pickAndExecuteAnAction(){ …}
```

But so would WaiterAgent, whose scheduler would look like:

```
void boolean pickAndExecuteAnAction(){
    boolean super =  pickAndExecuteAnAction();//call the PersonAgent scheduler
   //   what follow would be rules for the waiter agent
   if there exists … such that … (action1(…); return T);
   …
   return super;
} //or something like that
```

Problems with this Naïve Solution

How would someone change jobs? Using the above design, perhaps like this:

```
Agent ag = new WaiterAgent(…);
ag.startThread();
```

Then if ag were to change jobs from a waiter to a bank teller:

```
ag.stopThread();
ag= new BankTellerAgent(… , ag.personData);
ag.startThread();
```

Get it? When we create the new job, we pass in all the person data so that ag has all its context as a person. But this is no good. The ag waiter pointer is held by many agents in SimCity201, ag's friends, contractors, etc. We would have to inform them all of the switch.

That's not the only problem. Most agents in SimCity201 will play at least 3 roles: as a person, holding a job, being a restaurant customer. How would that work? I don't think there is a satisfactory answer to this problem based on the naïve solution.

A Better Solution

The following design is based on the concept of a person having different roles, such as being a waiter, bank teller, customer etc. The architecture structure is as follows:

class WaiterRole extends Role implements Waiter{…} //explained on next page
class BankTellerRole extends Role implements BankTeller{…} //explained on next page

An object that extend the class Role is just an unthreaded agent, i.e., anything that extends Role will have messages, a scheduler, and actions, just like any threaded agent.

The PersonAgent is like our old agents with some additions:

```
class PersonAgent extends Agent implements Person {
 //normal slots like before
   …
//and this new thing
   List<Role>roles; //like WaiterRole, BankTellerRole, etc.
 //and this new method
public void addRole(Role r) {roles.add(r); r.setPerson(this)}
//addRole can be called from a SimCity configuration script, or by the PersonAgent itself if it decides
//to add a role. Of course, the PersonAgent could remove roles using: roles.remove(r);

//the person agent scheduler has more stuff in it
public void pickAndExecuteAnAction(){
//   what follow are the rules for the person
    if there exists … such that … (action1(…); return T);
    if there exists … such that … (action2(…); return T);
    …
//Instead of return false, we now call the scheduler for the roles:
  boolean anytrue = false;
  for (r in roles) such that r.active // active is in each role and tells if the person is playing the role now.
            anytrue = anytrue OR r. pickAndExecuteAnAction();
  return anytrue; //if anytrue is false, it means NO rule fired, so have the person thread block.
}
```

You'd have to think hard about the order of all that code. Putting all person rules in front of the scheduler call to role player might get you fired. Suppose as a person you're doing actions to call friends,

look at your smartphone, etc. You might be ignoring your actions in your role that is your job. So, maybe a person's rules that are emergencies come first, then call the role scheduler, then the rest of the person rules. It's your design.

The class Role will look something like this:

```
class Role {
        PersonAgent myPerson;
        public setPerson(PersonAgent a) {myPerson=a;}
        public PersonAgent getPersonAgent() {
                return myPerson;} //so other agents or role players can send you Person messages.

        private void stateChanged(){ myPerson.stateChanged};
}
```

Now a role player will look like this:

```
class WaiterRole extends Role implements Waiter {
 //The code in here is almost identical to your current WaiterAgent.
 //Any message in here will call stateChanged(), which will be handled by the Role base class
 //and send it on to the PersonAgent running the thread.
}
```

So, here is how it works. Say you have a person who is going to play the role of a waiter and customer. The code to create him would look something like:

```
Person p = new PersonAgent(...);
p.addRole(new CustomerRole(...));
p.addRole(new WaiterRole(...));
```

When p decides to go to a restaurant r, he will write action code that looks something like:

```
GoToRestaurant(Restaurant r) {
  ...
   r.getHost().ImHungry(roles.findCustomerRole());
 ...
}
```

Get it? the PersonAgent sends the same ImHungry() message to the host as before to introduce himself, but sends the CustomerRole pointer. The restaurant role players will deal the CustomerRole object exactly the same as in your v2.2 restaurant.