Documentación Trabajo Práctico SI

Domestic-Robot

Carlos Dacunha González
Ruben Fernández Boullon

1. - Agentes

1.1. - DomesticRobot.mas2j

Con esta primera parte del código, presentamos cuales van a ser los agentes involucrados

```
MAS domestic_robot {
        environment: HouseEnv(gui)
        agents:
// Robots móviles
        owner;
         robot;
        repartidor;
        basurero;
// Robots estáticos
        incinerador;
// Supermercados
        mercadona supermarket.asl [beliefs = "money(10)"];
        gadis supermarket.asl [beliefs = "money(10)"];
        proveedor [beliefs = "money(20), precio(beer, bock, 2), precio(beer, 1906, 4),
precio(beer, estrella, 3), precio(tortilla,10), precio(empanada,15), precio(bocata,7)"];
```

Como Agentes móviles estarán:
El Owner: encargado de beber, comer, pedir de beber, tirar las latas por el grid y a algunas veces se levantará y decidirá tirarla por sí mismo y quitarle un poco de trabajo al Basurero.
El Robot: encargado de llevarle de comer y beber al Owner, prepará los pinchos que después servirá, vaciará el lavavajillas y guardará los platos, y será el encargado de controlar y reponer la comida y bebida.
El Repartidor: encargado de llevar los productos de la zona de entrega al frigorífico.
El Basurero: encargado de recoger las latas que algunas veces tirará el owner al suelo.

Como único agente estático está el Incinerador, que cuando la papelera se llena será el encargado de incinerar la basura cambiando su color a un suave rojo durante el proceso.

Y por último están los supermercados, en este caso el Mercadona y el Gadis, que realizarán las mismas acciones. Cuando las despensas del Robot se queden vacías, este contactará con los supermercados para que le suministre más

productos. Una vez que estos se queden sin stock suficiente, contactarán con el Proveedor el cual actuará como un supermercado para ellos.

1.2. - Owner.asl

Inicialmente el owner contará con 6 planes iniciales:

```
!get(beer).
!setMoneyRobot(100).
!setFavBeer(1906).
!aburrimiento.

+!setMoneyRobot(Qtd) <- .send(robot, tell, money(Qtd)).

+!setFavBeer(Marca) <- .send(robot, tell, favBeer(Marca)).
```

Tanto *!setFavBeer* como *!setMoneyBeer*, se encargarán de comunicar al Robot, la cerveza favorita del Owner y de cuanto dinero dispone el Robot para comprar comida y bebida.

Por otra parte, *!aburrimiento* se ejecutará de forma recursiva cada X segundos para preguntarle al Robot la hora como forma de interacción o conversación, con el siguiente plan:

```
+!aburrimiento <-
        X = math.round(math.random(4000));
        .wait(X+4000);
        .send(robot, achieve, tellTime);
        !aburrimiento.
```

Finalmente *!get(beer)*, dará inicio al programa.

```
+!get(beer) : ~couldDrink(beer) <- .println("Owner ha bebido demasiado por hoy.").

+!get(beer) <- .send(robot, achieve, bring(owner,beer)).
```

Mientras que el Owner pueda beber este solicitará al Robot que le traiga cerveza.

```
+has(owner,beer) <- !drink(beer).

-has(owner,beer) : not recogiendoLata <- !get(beer).
```

Antes de poder beber, el Owner debe saber si tiene o no una lata, cuando tenga una la beberá y cuando no, en caso de no estar yendo a tirar la lata, buscará pedirle al Robot que le traiga más cerveza.

```
+!drink(beer) : not has(owner,beer) & not yaElegido <-
        +yaElegido;
```

```
        X = math.round(math.random(1));
        if(X == 0){
                .println("El owner decide lanzar la lata");
                throwBeerCan;
                +recogiendoLata;
                .send(basurero,achieve,recogerLata);
                .send(owner,achieve,get(beer));
  }else{
        .println("El owner decide reciclar la lata");
        +recogiendoLata;
        !recogerLata;
        .send(owner,achieve,get(beer));
  }
  -yaElegido.

+!drink(beer) : not has(owner,beer) & yaElegido <-
        .wait(100);
        !drink(beer).

+!drink(beer) : has(owner,beer) <-
        sip(beer);
        .println("Owner está bebiendo cerveza");
        !drink(beer).
```

Una vez que el Robot haya sido avisado para que le traiga comida y bebida al Owner, este esperará a recibir la cerveza y poder beber. Cuando la premisa de que el Owner tiene cerveza, *has(owner,beer)*, este beberá.

Una vez que se haya terminado la cerveza, elegirá aleatoriamente entre 1 o 0, para el caso de 0, el Owner lanzará la lata vacía al suelo y avisará al Basurero para que vaya a buscarlo. En caso contrario, 1, será él el que se levantará y llevará la lata a la papelera para reciclarla y volverá. En cualquiera de los casos, una vez realizadas estas acciones, el Owner volverá al plan *!get(beer)*, para seguir bebiendo.

```
+!recogerLata <-
        !go_at(bin);
        tirarLata;
        !go_at(ownerchair);
        -recogiendoLata.
```

Para poder moverse hasta la papelera, cuenta con el plan *!recogerLata* y varios planes *!go_at* para conseguir llegar.

1.3. - Robot.asl

Inicialmente el Robot contará con 3 creencias, pero rápidamente obtendrá muchas más de su entorno.

```
available(beer,fridge).
limit(beer,7).
tamPackBeer(3).
```

Estas 3 le indicarán que hay cerveza disponible en el frigorífico, que el límite para el Owner es de 7 cervezas y que el tamaño máximo de compra son 3, respectivamente.

```
too_much(B) :-
        .date(YY,MM,DD) &
        .count(consumed(YY,MM,DD,_,_,_,B),QtdB) &
        limit(B,Limit) &
        QtdB >= Limit.

+!bring(owner,beer) :  too_much(beer) & limit(beer,L) & not healthMsg <-
        .concat("The Department of Health does not allow me to give you more
than ", L," beers a day! I am very sorry about that!",M);
        .send(owner,tell,msg(M));
        +healthMsg;
        !bring(owner,beer).
```

Tras cada cerveza dada al Owner, el Robot llevará un registro de todas ellas, en caso de que supere el límite, dejará de traerle cervezas.

```
+!bring(owner,beer) :  not available(beer,fridge) & available(pincho,fridge) <-
        .println("Busco reponer cervezas, no quedan");
        !checkSupermarkets;
        .wait(500);
        !checkPrice;
        .wait(50);
        !buySupermarketBeer.
```

En aquel caso que no haya cerveza disponible, el Robot contactará con los supermercados para que le digan los precios y calculará en qué supermercado está la cerveza favorita del Owner a menor precio. Y posteriormente, la comprará.

```
+!bring(owner,beer) : available(beer,fridge) & not available(pincho,fridge) & not
yaPedidoPincho <-
        .println("Busco reponer pinchos, no quedan");
        +yaPedidoPincho;
```

```
        siguienteAperitivo;
        !checkSupermarkets;
        .wait(500);
        !checkPrice;
        .wait(50);
        !buySupermarketPincho.
```

En el caso de que sean los pinchos los que faltan el Robot hará las mismas acciones que para comprar más cerveza, con la diferencia que buscará cual supermercado tiene el pincho elegido más barato.

```
+!bring(owner,beer) : not available(beer,fridge) & not available(pincho,fridge) <-
        .println("Busco reponer cervezas y pincho, no quedan");
        siguienteAperitivo;
        !checkSupermarkets;
        .wait(500);
        !checkPrice;
        .wait(50);
        !buySupermarketBeer;
        !buySupermarketPincho.
```

En el caso de que carezca tanto de stock de cervezas como de pinchos, combinará ambos planes para comprarlos todos a la vez.

```
+!bring(owner,beer) : too_much(beer) & healthMsg <-
         .wait(5000);
         !tellTime;
        !bring(owner,beer).
```

En el caso de que el Owner haya bebido demasiado, el Robot esperará pacientemente a que pueda volver a beber.

```
+!bring(owner,beer) :  available(beer,fridge) & not too_much(beer) &
available(pincho,fridge) & not carringPlato & not preparingPincho & not
vaciandoLavavajillas <-
        +carringPlato;
        X = math.round(math.random(200));
        .wait(X);
        !traer(owner,beer).

+!traer(owner,beer) : not working <-
        +working;
        !go_at(fridge);
        open(fridge);
        get(beer);
```

```
        get(pincho);
        close(fridge);
        !go_at(closeownerchair);
        hand_in(beer);
        !go_at(lavavajillas);
        anadirPlatosLavavajillas(1);
        .date(YY,MM,DD); .time(HH,NN,SS);
        +consumed(YY,MM,DD,HH,NN,SS,beer);
        -working;
        -carringPlato.

+!traer(owner,beer) <- true.
```

En este grupo de planes, el Robot se moverá a la nevera, cogerá una cerveza, un pincho, se los llevará al Owner y posteriormente llevará el plato sucio al lavavajillas para limpiarlo cuando esté lleno. El motivo de separarlo en varios planes se debe a la concurrencia en los planes, el Robot puede realizar el plan de *!bring(owner,beer)*, por distintos motivos.
En el caso de que estos planes se ejecuten simultáneamente, es posible que el segundo plan *!bring*, se ejecute antes de que el primero haya podido añadir la creencia *carringPlato*, y por lo tanto, el Robot realizará por duplicado las acciones, e incluso por triplicado.
Para evitar esto, surge la función *math.random*, que forzará a los planes a esperar entre 0 y 200 milisegundos. Así, el más rápido de todos se ejecutará mientras que el resto se mueven al plan secundario *!traer(owner,beer)*, que simplemente los finalizará.

```
+!bring(owner,beer) <-
        X = math.round(math.random(200));
        .wait(X);
        !bring(owner,beer).
```

Aquellos planes *!bring*, que no hayan entrado en los anteriores se quedarán esperando aleatoriamente entre 0 y 200 milisegundos, hasta que cambien las condiciones y puedan entrar en alguno.

```
+lavavajillasLleno <- !vaciar_lavavajillas.

+!vaciar_lavavajillas : not carringPlato & not preparingPincho <-
        +vaciandoLavavajillas;
        !go_at(lavavajillas);
        .wait(500);
        vaciar_lavavajillas;
        !go_at(alacena);
        .wait(500);
```

```
        anadirPlatosAlacena(5);
        -vaciandoLavavajillas;
        -lavavajillasLleno.

+!vaciar_lavavajillas <-
        .wait(100);
        !vaciar_lavavajillas.
```

Una vez que el Lavavajillas esté lleno, 5 platos, y el Robot no esté ni llevándole cerveza al Owner ni preparando los pinchos, este se moverá al lavavajillas donde quitará todos los platos y los guardará en la alacena.

```
+!prepararPincho(Product) : not carringPlato & not vaciandoLavavajillas <-
        +preparingPincho;
        !go_at(alacena);
        quitarPlatosAlacena(5);
        !go_at(fridge);
        .wait(1000);
        prepararPincho(Product);
        -preparingPincho.

+!prepararPincho(Product) <-
        .wait(100);
        !prepararPincho(Product).
```

Cuando el pedido de pinchos llegue a la nevera, el Repartidor avisará al Robot de que puede preparar los pinchos. En el momento, que el Robot no esté llevándole cerveza al Owner ni vaciando el lavavajillas, este se moverá a la alacena, cogerá cinco platos y en la nevera colocará 5 porciones de pinchos. Pudiendo ser entre tortilla, empanada o un trozo de bocata.

```
+reponerCerveza : not enProceso <-
        .println("Busco reponer cervezas");
        +enProceso;
        !checkSupermarkets;
        .wait(500);
        !checkPrice;
        .wait(50);
        !buySupermarketBeer;
        -enProceso;
        -reponerCerveza.

+reponerCerveza <- true.
```

Cuando en la nevera, únicamente haya 1 cerveza, el robot de forma previsora buscará, al igual que en los planes *!bring*, comprar más cerveza.

```
+!buySupermarketBeer : favBeer(Marca) & barataF(Product, Marca, Precio, Stock, Super) & money(DineroInicial) & tamPackBeer(Qtd) <-
   Dinero = DineroInicial;
   if(Precio * Qtd <= Dinero){
        if(Qtd <= Stock){
             .println("Realizo un pedido de ", Qtd, " cervezas ", Marca);
             .send(Super, achieve, order(Product, Marca, Qtd));
             !pagar(Precio, Qtd, Super);
        }else{
             Resto = Qtd - Stock;
             ?barata(Product, MarcaB, PrecioB, StockB, Super);
             if(Stock > 0){
                  .println("Realizo un pedido de ", Stock, " cervezas ", Marca, "y ",Resto," ",MarcaB);
                  .send(Super, achieve, order(Product, Marca, Stock, MarcaB, Resto));
                  !pagar(Precio, Stock, Super);
                  !pagar(PrecioB, Resto, Super);
             }else{
                  ?masbarata(Product, MarcaC, PrecioC, StockC, SuperC);
                  if(StockC >= Qtd){
                       .println("Realizo un pedido de ", Qtd, " cervezas ", MarcaC);
                       .send(SuperC, achieve, order(Product, MarcaC, Qtd));
                       !pagar(PrecioC, Qtd, SuperC);
                  }else{
                       .concat("No he podido comprar más Cervezas, no tienen Stock Suficiente",M);
                       .send(owner, tell, msg(M));
                  }
             }
        }
   }else{
        ?masbarata(Product, MarcaC, PrecioC, StockC, SuperC);
        if(PrecioC * Qtd <= Dinero){
             if(StockC >= Qtd){
                  .println("Realizo un pedido de ", Qtd, " cervezas ", MarcaC);
                  .send(SuperC, achieve, order(Product, MarcaC, Qtd));
                  !pagar(PrecioC, Qtd, SuperC);
             }else{
                  .concat("No he podido comprar más Cervezas, no tienen Stock Suficiente de aquellas que me puedo permitir",M);
```

```
                .send(owner, tell, msg(M));
            }
        }else{
            .concat("No he podido comprar más cerveza, me quedan: ",Dinero,"€
y la más barata cuesta: ",PrecioC*Qtd ,"€",M);
            .send(owner, tell, msg(M));
        }
    }.
```

Para el proceso de comprar cerveza, el Robot se fijará en múltiples datos para saber cual comprar. Inicialmente, si tiene suficiente dinero y hay el suficiente stock de la cerveza favorita del Owner, la comprará. En el caso de que no haya suficiente stock, el Robot se enfrentará a dos opciones, que haya o no cervezas de la marca favorita.
En el caso de que todavía queden 1 o 2, el Robot intentará rellenar el hueco con la cerveza más barata del mismo supermercado.
En el caso de que no haya suficiente, el Robot ni se molestará y comprará la cerveza más barata de todos los supermercados.
Y en el caso de que no tenga dinero suficiente para comprar las cervezas que le gustan al Owner, comprará las más baratas de todos los supermercados.

```
+!buySupermarketPincho : siguienteAperitivo(Pincho) & barato(Pincho, Precio,
Stock, Super) & money(DineroInicial) <-
    Dinero = DineroInicial;
    if(Dinero >= Precio){
        if(Stock >= 1){
            .println("Realizo un pedido de 1 pincho de ", Pincho);
            .send(Super, achieve, order(Pincho, 1));
            !pagar(Precio, 1, Super);
        }
    }else{
        ?masbarato(PinchoB, PrecioB, StockB, SuperB);
        if(Dinero >= PrecioB){
            if(StockB >= 1){
                .println("Realizo un pedido de 1 pincho de ", PinchoB);
                .send(SuperB, achieve, order(PinchoB, 1));
                !pagar(PrecioB, 1, SuperB);
            }else{
                .concat("No he podido comprar más pinchos, no tengo dinero
suficiente para aquellos que estan disponibles",M);
                .send(owner, tell, msg(M));
            }
        }else{
            .concat("No he podido comprar más aperitios, me quedan:
",Dinero,"€ y el aperitivo más barato cuesta: ",PrecioB,"€",M);
```

```
                .send(owner, tell, msg(M));
        }
    }.
```

En el caso de que el robot vaya a comprar pinchos, inicialmente comprará el siguiente pincho elegido, si no tiene dinero suficiente o no hay stock, comprará el más barato.

```
+!pagar(Precio, Qtd, Super) : money(DineroInicial) <-
        Pago = Qtd * Precio;
        Dinero = DineroInicial - Pago;
        -money(_);
        +money(Dinero);
        .send(Super, achieve, pago(Pago)).
```

Siempre que el Robot realice una compra, le enviará al supermercado un pago con la cantidad correspondiente, descontándola de su monedero.

```
+delivered(beer,Marca,Qtd,OrderId) : available(beer,fridge) <-
        -delivered(beer,Marca,Qtd,OrderId).

+delivered(beer,Marca,Qtd,OrderId) : not carringPlato <-
        +available(beer,fridge);
        !bring(owner,beer);
        -delivered(beer,Marca,Qtd,OrderId).

+delivered(beer,Marca,Qtd,OrderId) <-
        +available(beer,fridge);
        -delivered(beer,Marca,Qtd,OrderId).

+delivered(Aperitivo,Qtd,OrderId) <-
        !prepararPincho(Aperitivo);
        -yaPedidoPincho;
        +available(pincho,fridge);
        !bring(owner,beer);
        -delivered(Aperitivo,Qtd,OrderId).
```

Cada vez que el Repartidor entregue un pedido, este avisará al Robot para que pueda mantener actualizadas sus creencias. En el caso de que el robot esté llevando comida al Owner, para evitar múltiples planes *!bring* se define un plan alternativo. El cual carece de la llamada al plan *!bring*, el motivo por el que es necesario para las cervezas y no para los pinchos se debe a que para las cervezas, el Robot es previsor y realiza un pedido cuando todavía queda una. Pudiendo dar la situación de entrega de producto mientras el Robot sirve al Owner.

```
+stock(beer,N) <-
	if(N = 0){
		-available(beer,fridge);
	}else{
		+available(beer,fridge);
	}
	-stock(beer,N).

+stock(pincho,N) <-
	if(N = 0){
		-available(pincho,fridge);
	}else{
		+available(pincho,fridge);
	}
	-stock(pincho,N).
```

Cada vez que se actualiza su percepción del entorno, el Robot comprueba la cantidad de cervezas y aperitivos para saber si hace falta comprar más.

```
+!checkSupermarkets <-
	.abolish(priceBeer(_,_,_,_));
	.abolish(barataF(_,_,_,_,_));
	.abolish(barata(_,_,_,_,_));
	.abolish(masbarata(_,_,_,_,_));
	.abolish(pricePincho(_,_,_));
	.abolish(barato(_,_,_,_));
	.send(mercadona, tell, tellPrice);
	.send(gadis, tell, tellPrice).
```

Antes de solicitar los precios y stock a los supermercados, el robot elimina las creencias anteriores para evitar fallos al elegir los supermercados más baratos.

```
+!checkPrice <-
	!checkPrice(beer);
	!checkPricePincho.

+!checkPrice(Product) : favBeer(Marca) <-
	.findall(Precio, priceBeer(Product, Marca, Precio, Stock), ListMarcaFav);
	.min(ListMarcaFav, MenorPrecio);
	?priceBeer(Product, Marca, MenorPrecio, Stock)[source(Super)];
	+barataF(Product, Marca, MenorPrecio, Stock, Super);

	.findall(PrecioB, priceBeer(Product, MarcaB, PrecioB, StockB), ListBarata);
	.min(ListBarata, PrecioBarata);
	?priceBeer(Product, MarcaB, PrecioB, StockB)[source(SuperB)];
```

```
        +masbarata(Product, MarcaB, PrecioB, StockB, SuperB);

        .findall(PrecioM, priceBeer(Product, MarcaM, PrecioM,
StockM)[source(mercadona)], ListBarataMercadona);
        .min(ListBarataMercadona, PrecioBarataM);
        ?priceBeer(Product, MarcaM, PrecioM, StockM)[source(mercadona)];
        +barata(Product, MarcaM, PrecioM, StockM, mercadona);

        .findall(PrecioG, priceBeer(Product, MarcaG, PrecioG,
StockG)[source(gadis)], ListBarataGadis);
        .min(ListBarataGadis, PrecioBarataG);
        ?priceBeer(Product, MarcaG, PrecioG, StockG)[source(gadis)];
        +barata(Product, MarcaG, PrecioG, StockG, gadis).

+!checkPricePincho <-
        .findall(PrecioT, pricePincho(tortilla, PrecioT, StockT), ListTortilla);
        .min(ListTortilla, PrecioTortilla);
        ?pricePincho(tortilla, PrecioTortilla, StockT)[source(SuperT)];
        +barato(tortilla, PrecioTortilla, StockT, SuperT);

        .findall(PrecioE, pricePincho(empanada, PrecioE, StockE), ListEmpanada);
        .min(ListEmpanada, PrecioEmpanada);
        ?pricePincho(empanada, PrecioEmpanada, StockE)[source(SuperE)];
        +barato(empanada, PrecioEmpanada, StockE, SuperE);

        .findall(PrecioJ, pricePincho(bocata, PrecioJ, StockJ), ListBocata);
        .min(ListBocata, PrecioBocata);
        ?pricePincho(Bocata, PrecioBocata, StockJ)[source(SuperJ)];
        +barato(bocata, PrecioBocata, StockJ, SuperJ);

        .findall(Precio, pricePincho(Product, Precio, Stock), List);
        .min(List, PrecioBarato);
        ?pricePincho(Product, PrecioBarato, Stock)[source(Super)];
        +masbarato(Product, PrecioBarato, Stock, Super).
```

Cuando el Robot vaya a comprar productos, antes comprobará los precios y seleccionará aquellos más ventajosos. Por orden, buscará en que supermercado está más barata la cerveza favorita del Owner, buscará la cerveza más barata de todos los supermercados y la cerveza más barata de cada supermercado.
En el caso de los pinchos, buscará el supermercado donde está más barato de cada uno de ellos y el más barato de todos los pinchos en todos los supermercados.

Además de estas, el Robot también cuenta con un plan para darle la hora al Owner y varios planes *!go_at* para poder desplazarse por el escenario.

1.4. - Repartidor.asl

El repartidor cuenta con únicamente 2 planes importantes, uno para la entrega de cervezas y otro para la entrega de pinchos.

```
+!delivered(beer, Marca, Qtd, OrderId) : not trabajando <-
        +trabajando;
        repartidorLlega;
        !go_at(delivery);
        .wait(200);
        !go_at(fridge);
        .send(robot, tell, delivered(beer, Marca, Qtd, OrderId));
        deliverBeer(beer, Marca, Qtd);
        .wait(200);
        !go_at(repartidorBase);
        repartidorSeVa;
        -trabajando.

+!delivered(beer, Marca, Qtd, OrderId) <-
        .wait(200);
        !delivered(beer, Marca, Qtd, OrderId).

+!delivered(Aperitivo, Qtd, OrderId) : not trabajando  <-
        +trabajando;
        repartidorLlega;
        !go_at(delivery);
        .wait(200);
        !go_at(fridge);
        .send(robot, tell, delivered(Aperitivo, Qtd, OrderId));
        .wait(200);
        !go_at(repartidorBase);
        repartidorSeVa;
        -trabajando.

+!delivered(Aperitivo, Qtd, OrderId) <-
        .wait(200);
        !delivered(Aperitivo, Qtd, OrderId).
```

En ambos el funcionamiento es el mismo. Si el repartidor no está trabajando, aparecerá en el escenario, se moverá a la zona de delivery, luego a la nevera donde dejará los productos y volverá a su base para salir del escenario. Una vez dejados, avisará al Robot para que pueda ir a preparar los pinchos o para que pueda llevarle más cervezas al owner.

El funcionamiento de aparecer y desaparecer se basa en la utilización de funciones en java que cambian el valor de un booleano entre false y true, que se utiliza a la hora de dibujar para ejecutar o no la parte del código correspondiente al Repartidor.
Obviamente, también cuentan con varios planes para el movimiento por el escenario.

1.5. - Basurero.asl

El Basurero es un agente muy simple, se encarga de recoger las latas que tira el Owner. Para ello cuenta con un plan, que lo lleva hasta la lata, luego a la papelera para reciclar y por último, de nuevo a su base. No sin antes avisar al Owner de que ya no está trabajando y ya puede volver a tirar más basura al suelo.

```
+!recogerLata <-
        !go_at(can);
        !go_at(bin);
        tirarLata;
        .send(owner, untell, recogiendoLata);
        !go_at(basureroBase).
```

Al igual que los anteriores agentes móviles, cuenta con varios planes *!go_at* para moverse por el escenario.

1.6. - Incinerador.asl

El Incinerador es un agente estático, que se queda siempre al lado de la papelera y cuando esta se llena, se encarga de quemar la basura.

```
+papeleraLlena <-
        vaciar_papelera;
        -papeleraLlena.
```

A diferencia del resto de agentes, este no necesita moverse y por lo tanto no cuenta con los planes *!go_at*.

1.7. - Supermercados.asl

Los supermercados son los encargados de suministrar al Robot, los productos necesarios para poder dar de comer al Owner por dinero.

```
+!establecerPrecios :  precioBase(tortilla, PrecioTortilla) & precioBase(empanada, PrecioEmpanada) & precioBase(bocata, PrecioBocata) & precioBase(beer, 1906, Precio1906) & precioBase(beer, bock, PrecioBock) & precioBase(beer, estrella, PrecioEstrella) <-
	.random(T);
	+stock(tortilla, T*3 + PrecioTortilla, 3);
	.random(E);
	+stock(empanada, E*3 + PrecioEmpanada, 3);
	.random(J);
	+stock(bocata, J*3 + PrecioBocata, 3);
	.random(B1906);
	+stock(beer, 1906, B1906*3 + Precio1906, 8);
	.random(BE);
	+stock(beer, estrella, BE*3 + PrecioEstrella, 8);
	.random(BB);
	+stock(beer, bock, BB*3 + PrecioBock, 8).
```

Inicialmente cuentan con un plan dedicado a aleatorizar los precios y hacer que en cada una de las iteraciones tengan precios distintos, siempre por encima del precio base establecido por el proveedor.

```
+!order(Product, Marca, Qtd) : stock(Product, Marca, Price, Stock) & Stock >= Qtd <-
	?last_order_id(N);
	OrderId = N + 1;
	-+last_order_id(OrderId);
	-stock(Product, Marca, Price, Stock);
	+stock(Product, Marca, Price, Stock-Qtd);
	.send(repartidor, achieve, delivered(Product, Marca, Qtd, OrderId));
	if(Stock < 2){
		!reponerStock(Product, Marca, 3);
	}.

+!order(Product, Marca, Qtd, MarcaB, QtdB) : stock(Product, Marca, Price, Stock) & Stock >= Qtd & stock(Product, MarcaB, PriceB, StockB) & StockB >= QtdB <-
	?last_order_id(N);
	OrderId = N + 1;
	-+last_order_id(OrderId);
	-stock(Product, Marca, Price, Stock);
	+stock(Product, Marca, Price, Stock-Qtd);
	-stock(Product, MarcaB, PriceB, StockB);
```

```
        +stock(Product, MarcaB, PriceB, StockB-QtdB);
        .send(repartidor, achieve, delivered(Product, Marca, Qtd, OrderId));
        if(Stock < 2){
                !reponerStock(Product, Marca, 3);
        }
        if(StockB < 2){
                !reponerStock(Product, MarcaB, 3);
        }.

+!order(Product,Qtd) : stock(Product, Price, Stock) & Stock >= Qtd <-
        ?last_order_id(N);
        OrderId = N + 1;
        -+last_order_id(OrderId);
        -stock(Product, Price, Stock);
        +stock(Product, Price, Stock-Qtd);
        .send(repartidor, achieve, delivered(Product, Qtd, OrderId));
        if(Stock < 2){
                !reponerStock(Product, 3);
        }.
```

Cada uno de los supermercados cuenta con un plan para poder hacer frente a los pedidos del Robot. Dos de ellos se encargan de entregar cerveza mientras que el tercero entregará los pinchos.
El motivo de la existencia del segundo plan para entregar cervezas es simplificar el trabajo del repartidor. Si no existiera este tendría que entregar primero el pedido de cerveza preferida y luego el pedido de la cerveza más barata.
El funcionamiento es igual en los tres, actualizarán el número de pedido, el stock del producto comprado y avisará al Repartidor de que entregue el producto pedido.
Finalmente, antes de terminar harán una comprobación de su nivel de stock actual, y en el caso de que sea 1 o 0 realizarán el plan de *!reponerStock* para pedirle más suministros al Proveedor.

```
+!reponerStock(Product, Qtd) : stock(Product, _ , Stock) & precioBase(Product,
Precio) & money(Dinero) <-
   if(Qtd > 0){
        DineroActual = Dinero;
        if(DineroActual >= Precio * Qtd){
                -+money(Dinero - Precio * Qtd);
                .send(proveedor, achieve, pago(Precio * Qtd));
                -stock(Product, _, Stock);
                +stock(Product, _, Stock + Qtd);
        }else{
                !reponerStock(Product,Qtd-1);
        }
```

```
  }.

+!reponerStock(Product, Marca, Qtd) : stock(Product, Marca , _ , Stock) &
precioBase(Product, Marca, Precio) & money(Dinero) <-
  if(Qtd > 0){
        DineroActual = Dinero;
        if(DineroActual >= Precio * Qtd){
              -+money(Dinero - Precio * Qtd);
              .send(proveedor, achieve, pago(Precio * Qtd));
              -stock(Product, Marca, _, Stock);
              +stock(Product, Marca, _, Stock + Qtd);
        }else{
              !reponerStock(Product, Marca, Qtd-1);
        }
  }.
```

Una vez que los supermercados se estén quedando sin stock, realizarán un pedido al proveedor de un máximo de 3 productos y un mínimo de 0, variando dependiendo de la cantidad de efectivo del supermercado. Si el supermercado tiene dinero suficiente, comprará 3 productos, en caso contrario de forma recursiva, lo seguirá intentando con menos cantidad hasta poder comprar o llegar a 0.
En el caso de que puedan reponer su stock, le harán el pedido y el pago al proveedor.

```
+!pago(Qtd) : money(DineroIncial) <-
        Dinero = DineroIncial + Qtd;
        -money(_);
        +money(Dinero).
```

También cuentan con un plan para ingresar el dinero del robot, que les ha pagado cuando ha realizado el pedido.

```
+!tellPrice(X) <-
        .findall(priceBeer(Product, Marca, Precio, Qtd),stock(Product, Marca, Precio,
Qtd),ListaBeer);
        .send(X, tell, ListaBeer);
        .findall(pricePincho(Product, Precio, Qtd),stock(Product, Precio, Qtd),
ListaPinchos);
        .send(X, tell, ListaPinchos).

+tellPrice[source(X)] <-
        !tellPrice(X);
        -tellPrice[source(X)].
```

Y por último, cuenta con el plan *!tellPrice*, cuando el robot quiere actualizar su información con respecto a los precios, envía un tell al supermercado y este le responde.

1.8. - Proveedor.asl

El proveedor es muy similar a los supermercados, con la diferencia de que siempre tiene stock y el mismo precio para sus productos en todas las iteraciones.

```
proveedor [beliefs = "money(20),
                      precio(beer, bock, 2),
                      precio(beer, 1906, 4),
                      precio(beer, estrella, 3),
                      precio(tortilla,10),
                      precio(empanada,15),
                      precio(bocata,7)"];
```

Todos los precios de sus productos se encuentran almacenados en el archivo *DomesticRobot.mas2j*, en forma de creencias.

```
!tellPrice.

+!tellPrice <-
        .findall(precioBase(Beer, MarcaB, PrecioB),precio(Beer, MarcaB,
PrecioB),ListaBeer);
        .send(mercadona, tell, ListaBeer);
        .send(gadis, tell, ListaBeer);

        .findall(precioBase(Pincho, PrecioP),precio(Pincho, PrecioP), ListaPinchos);
        .send(mercadona, tell, ListaPinchos);
        .send(gadis, tell, ListaPinchos).
```

Al inicio de la ejecución, el proveedor le enviará a los supermercados todos los productos que tiene disponible, para que ellos puedan ponerle su propio precio y vendérselo al Robot.

```
+!pago(Qtd) : money(Dinero) <-
        -money(Dinero);
        +money(Dinero+Qtd).
```

Además cuenta con un plan *!pago*, para actualizar el dinero de los pagos que realizan los supermercados cuando reponen su stock.

1.9. - Sistema de Movimiento

Pese a que el sistema de movimiento está presente en todos los agentes móviles, ocupa tanto espacio que merece un punto aparte.
La lógica de los planes *!go_at*, es muy simple, si enfrente no hay un obstáculo, avanza, en caso contrario, esquiva y avanza, hasta llegar a su destino.
Antes de nada hace falta aclarar cómo funcionan y qué son los obstáculos, Cada uno de los agentes móviles tiene una diferente percepción del entorno, por ejemplo, la silla del Owner es un obstáculo para todos y buscarán evitarlo, excepto para el Owner.

```
// Posiciones que los agentes pueden atravesar
        public static final Literal posCloseFridge =
Literal.parseLiteral("position(fridge,0,1)");
        public static final Literal posCloseOwnerChair =
Literal.parseLiteral("position(closeownerchair,10,9)");
        public static final Literal posCloseBin =
Literal.parseLiteral("position(bin,8,1)");
        public static final Literal posCloseLavavajillas =
Literal.parseLiteral("position(lavavajillas,2,1)");
        public static final Literal posCloseAlacena =
Literal.parseLiteral("position(alacena,1,1)");

    //Solo el owner puede atravesar esta posicion
        public static final Literal posOwnerChair =
Literal.parseLiteral("position(ownerchair,10,10)");
        public static final Literal posOwnerChairObs =
Literal.parseLiteral("position(obstaculo,10,10)");

    //Solo el repartidor puede atravesar esta posicion
        public static final Literal posDelivery =
Literal.parseLiteral("position(delivery,0,10)");
        public static final Literal posDeliveryObs =
Literal.parseLiteral("position(obstaculo,0,10)");
        public static final Literal posRepartidorBase =
Literal.parseLiteral("position(repartidorBase,1,10)");
        public static final Literal posRepartidorBaseObs =
Literal.parseLiteral("position(obstaculo,1,10)");

    //Solo el basurero puede atravesar esta posicion
        public static final Literal posBasureroBase =
Literal.parseLiteral("position(basureroBase,10,0)");
        public static final Literal posBasureroBaseObs =
Literal.parseLiteral("position(obstaculo,10,0)");
```

```
   // Posiciones no atravesables
        public static final Literal posFridge =
Literal.parseLiteral("position(obstaculo,0,0)");
        public static final Literal posIncinerador =
Literal.parseLiteral("position(obstaculo,7,0)");
        public static final Literal posAlacena =
Literal.parseLiteral("position(obstaculo,1,0)");
        public static final Literal posLavavajillas =
Literal.parseLiteral("position(obstaculo,2,0)");
        public static final Literal posBin =
Literal.parseLiteral("position(obstaculo,8,0)");
```

En esta parte, se definen todos los obstáculos o puntos de interés estáticos, que no cambian en cada iteración. Aquellos que siguen la estructura position(obstáculos,_,_) serán evitados mientras que los que tengan un nombre distinto, serán posibles destinos.

```
String[] agentList = {"robot", "owner", "basurero", "repartidor"};

for(String agent : agentList){

        clearPercepts(agent);

  // Pos atravesables
        addPercept(agent, posCloseFridge);
        addPercept(agent, posCloseOwnerChair);
        addPercept(agent, posCloseBin);
        addPercept(agent, posCloseLavavajillas);
        addPercept(agent, posCloseAlacena);

  //Pos no atravesables
        addPercept(agent, posIncinerador);
        addPercept(agent, posFridge);
        addPercept(agent, posAlacena);
        addPercept(agent, posLavavajillas);
        addPercept(agent, posBin);

        addPercept(agent, Literal.parseLiteral("position(robot," +
model.getAgPos(0).x + "," + model.getAgPos(0).y + ")"));

        if(model.beerCanShow){
                addPercept(agent, Literal.parseLiteral("position(can," +
model.getAgPos(1).x + "," + model.getAgPos(1).y + ")"));
        }
```

```
        addPercept(agent, Literal.parseLiteral("position(basurero," +
model.getAgPos(2).x + "," + model.getAgPos(2).y + ")"));
        addPercept(agent, Literal.parseLiteral("position(owner," +
model.getAgPos(3).x + "," + model.getAgPos(3).y + ")"));
        addPercept(agent, Literal.parseLiteral("position(repartidor," +
model.getAgPos(4).x + "," + model.getAgPos(4).y + ")"));

        for(int i=0; i < 7; i++){
  addPercept(agent, Literal.parseLiteral("position(obstaculo,"+
model.getObstaculoPosX(i) +","+model.getObstaculoPosY(i)+")"));
        }
}

  addPercept("robot", posOwnerChairObs);
  addPercept("basurero", posOwnerChairObs);
  addPercept("repartidor", posOwnerChairObs);
  addPercept("owner", posOwnerChair);

  addPercept("robot", posDeliveryObs);
  addPercept("owner", posDeliveryObs);
  addPercept("basurero", posDeliveryObs);
  addPercept("repartidor", posDelivery);

  addPercept("robot", posRepartidorBaseObs);
  addPercept("owner", posRepartidorBaseObs);
  addPercept("basurero", posRepartidorBaseObs);
  addPercept("repartidor", posRepartidorBase);

  addPercept("robot", posBasureroBaseObs);
  addPercept("owner", posBasureroBaseObs);
  addPercept("basurero", posBasureroBase);
  addPercept("repartidor", posBasureroBaseObs);
```

En este trozo de código, podemos ver 4 partes diferenciadas, una primera dentro del bucle *for*, donde se añadirán a cada uno de los agentes móviles la mayoría de los puntos de interés definidos anteriormente.
Una segunda parte donde se añaden las posiciones de los agentes móviles y la lata, las cuales cambian todo el rato.
Una tercera donde de forma recursiva se añaden las posiciones de los 7 obstáculos que se desperdigaran por el grid de forma aleatoria en cada iteración.
Y una cuarta donde de forma personalizada se añadirán ciertas localizaciones que solo algunos agentes pueden entrar, como son la silla del Owner para el Owner, la zona de Delivery para el Repartidor o la base del Basurero para el propio Basurero.

```
boolean moveTowards(String mov, int agent) {

        Location r1 = getAgPos(agent);
        if(mov.equals("up")){
                if(r1.y == 0){
                        return false;
                }else{
                        r1.y--;
                }
        }else if(mov.equals("down")){
                if(r1.y == 10){
                        return false;
                }else{
                        r1.y++;
                }
        }else if(mov.equals("left")){
                if(r1.x == 0){
                        return false;
                }else{
                        r1.x--;
                }
        }else if(mov.equals("right")){
                if(r1.x == 10){
                        return false;
                }else{
                        r1.x++;
                }
        }

        setAgPos(agent, r1);

        if (view != null) {
       view.update(lFridge.x,lFridge.y);
                view.update(lOwner.x,lOwner.y);
               view.update(lDelivery.x,lDelivery.y);
                view.update(lBin.x,lBin.y);
                view.update(lLavavajillas.x,lLavavajillas.y);
                view.update(lRepartidor.x, lRepartidor.y);
                view.update();
       }
       return true;
  }
```

Para poder moverse por el entorno, los agentes móviles decidirán moverse en 4 direcciones: right, left, up y down. En las cuales se controlará que el agente no se salga de los límites.

```
+!go_at(Destino) : .my_name(MyName) & position(MyName,MX, MY) &
position(Destino, DX, DY) & MX == DX & MY == DY <-
        .println("HE LLEGADO A MI DESTINO ", Destino).

+!go_at(Destino) : .my_name(MyName) & position(MyName,MX, MY) &
position(Destino, DX, DY) & MX < DX <-
   !go_right;
        !go_at(Destino).

+!go_at(Destino) : .my_name(MyName) & position(MyName,MX, MY) &
position(Destino, DX, DY) & MX > DX <-
        !go_left;
        !go_at(Destino).

+!go_at(Destino) : .my_name(MyName) & position(MyName,MX, MY) &
position(Destino, DX, DY) & MY < DY <-
        !go_down;
        !go_at(Destino).

+!go_at(Destino) : .my_name(MyName) & position(MyName,MX, MY) &
position(Destino, DX, DY) & MY > DY <-
        !go_up;
        !go_at(Destino).

+!go_at(Destino) : not position(Destino,_,_) <-
        .println("HA SUCEDIDO UN ERROR, NO PUEDO LLEGAR A MI DESTINO ",
Destino).
```

La primera parte del algoritmo, cuenta con 6 planes, el primero y el último determinan los estados de éxito y fracaso y los 4 restantes, las 4 posibles direcciones a elegir.

```
+!go_right : .my_name(MyName) & position(MyName,MX, MY) &
position(obstaculo, MX+1, MY) <-
   MY2 = MY;
   if(MY2 == 0){
        !go_down;
        !go_right2;
   }else{
        if(MY2 == 10){
                !go_up;
```

```
                !go_right2;
        }else{
                X = math.round(math.random(1));
                if(X == 0){
                        !go_up;
                        !go_right2;
                }else{
                        !go_down;
                        !go_right2;
                }
        }
  }.

+!go_right : .my_name(MyName) & position(MyName,MX, MY) & MX < 10 <-
        move_towards(right).

+!go_right <- true.

+!go_left : .my_name(MyName) & position(MyName,MX, MY) & position(obstaculo,
MX-1, MY) <-
   MY2 = MY;
   if(MY2 == 0){
        !go_down;
        !go_left2;
   }else{
        if(MY2 == 10){
                !go_up;
                !go_left2;
        }else{
                X = math.round(math.random(1));
                if(X == 0){
                        !go_up;
                        !go_left2;
                }else{
                        !go_down;
                        !go_left2;
                }
        }
   }.

+!go_left : .my_name(MyName) & position(MyName,MX, MY) & MX > 0 <-
        move_towards(left).

+!go_left <- true.
```

```
+!go_up : .my_name(MyName) & position(MyName,MX, MY) & position(obstaculo,
MX, MY-1) <-
   MX2 = MX;
        if(MX2 == 10){
        !go_left;
        !go_up2;
   }else{
        if(MX2 == 0){
                !go_right;
                !go_up2;
        }else{
                X = math.round(math.random(1));
                if(X == 0){
                        !go_left;
                        !go_up2;
                }else{
                        !go_right;
                        !go_up2;
                }
        }
   }.

+!go_up : .my_name(MyName) & position(MyName,MX, MY) & MY > 0 <-
        move_towards(up).

+!go_up <- true.

+!go_down : .my_name(MyName) & position(MyName,MX, MY) &
position(obstaculo, MX, MY+1) <-
   MX2 = MX;
   if(MX2 == 10){
        !go_left;
        !go_down2;
   }else{
        if(MX2 == 0){
                !go_right;
                !go_down2;
        }else{
                X = math.round(math.random(1));
                if(X == 0){
                        !go_left;
                        !go_down2;
                }else{
                        !go_right;
                        !go_down2;
```

```
                }
        }
  }.

+!go_down : .my_name(MyName) & position(MyName,MX, MY) & MY < 10 <-
        move_towards(down).

+!go_down <- true.
```

Una vez definida la dirección a la que queremos ir se ejecutará uno de los 4 planes, *!go_right*, *!go_left*, *!go_up* y *!go_down*. Todos ellos cuentan con 3 opciones, una donde hay un obstáculo delante, el cual intentarán esquivar moviéndose lateralmente de forma aleatoria, a menos que el agente se encuentre en el borde del grid, en ese caso se moverá para el lado válido y luego llamará a la segunda parte de los planes, *!go_right2*, *!go_left2*, *!go_up2* y *!go_down2*.
Una segunda opción donde se movería, en el caso de que no haya un obstáculo delante ni el movimiento te llevaría a salirte del Grid.
Y en el caso de que no hay un movimiento posible, simplemente retorna un true, para intentar volver a ejecutar el plan *!go_at* y gracias a la aleatoriedad, esquivar el obstáculo a base de intentos.

```
+!go_right2 : .my_name(MyName) & position(MyName,MX, MY) & not
position(obstaculo, MX+1, MY) <-
        move_towards(right).

+!go_right2 : .my_name(MyName) & position(MyName,MX, MY) & not
position(obstaculo, MX+1, MY) & MX < 10 <-
        move_towards(right).

+!go_right2 <- true.

+!go_left2 : .my_name(MyName) & position(MyName,MX, MY) & not
position(obstaculo, MX-1, MY) <-
        move_towards(left).

+!go_left2 : .my_name(MyName) & position(MyName,MX, MY) & not
position(obstaculo, MX-1, MY) & MX > 0 <-
        move_towards(left).

+!go_left2 <- true.

+!go_up2 : .my_name(MyName) & position(MyName,MX, MY) & not
position(obstaculo, MX, MY-1) <-
        move_towards(up).
```

```
+!go_up2 : .my_name(MyName) & position(MyName,MX, MY) & not
position(obstaculo, MX, MY-1) & MY > 0 <-
        move_towards(down).

+!go_up2 <- true.

+!go_down2 : .my_name(MyName) & position(MyName,MX, MY) & not
position(obstaculo, MX, MY+1) <-
        move_towards(down).

+!go_down2 : .my_name(MyName) & position(MyName,MX, MY) & not
position(obstaculo, MX, MY+1) & MY < 10 <-
        move_towards(down).

+!go_down2 <- true.
```

Y por último los planes *!go_right2*, *!go_left2*, *!go_up2* y *!go_down2*, estos intentan evadir los obstáculos, avanzando hacia el destino después de la maniobra evasiva de los anteriores planes.

Visto en conjunto los planes provocan el avance del agente hasta que encuentre un obstáculo. Si lo encuentra intentará evitarlo rodeándolo haciendo una L, si no funciona volverá a intentarlo. Dado su componente aleatorio, lo intentará una y otra vez hasta que lo consiga.

2. - Entorno y Funciones

En el programa hay una gran variedad de funciones utilizadas para distintas acciones, pero todos actúan de la misma forma. Cuando un agente quiere realizar un acción llama por ella, en el caso de que esta se encuentre almacenada en el entorno y tenga un función asignada, se ejecutará

```
public static final Literal of = Literal.parseLiteral("open(fridge)");
public static final Literal cf = Literal.parseLiteral("close(fridge)");
public static final Literal gb = Literal.parseLiteral("get(beer)");
public static final Literal gp = Literal.parseLiteral("get(pincho)");

} else if (action.getFunctor().equals("anadirPlatosAlacena") & ag.equals("robot")) {
      int platos = Integer.parseInt(action.getTerm(0).toString());
      result = model.anadirPlatosAlacena(platos);

} else if (action.getFunctor().equals("quitarPlatosAlacena") & ag.equals("robot")) {
      int platos = Integer.parseInt(action.getTerm(0).toString());
      result = model.quitarPlatosAlacena(platos);

} else if (action.getFunctor().equals("anadirPlatosLavavajillas") & ag.equals("robot"))
{
      int platos = Integer.parseInt(action.getTerm(0).toString());
      result = model.anadirPlatosLavavajillas(platos);
```

Por ejemplo, en la situación que el Robot intente realizar cualquiera de estas 3 acciones, *anadirPlatosAlacena*, *quitarPlatosAlacena*, *anadirPlatosLavavajillas*, se llamará a las funciones correspondientes.

Las funciones utilizadas más importantes son:

```
boolean siguienteAperitivo() {
        double random = Math.random() * 2;
        siguienteAperitivo = (int) random;
        return true;
}
```

Esta función calcula un número entero entre 0 y 2, y acompañado con la siguiente, permite elegir un pincho aleatorio antes de cada compra.

```
if (model.siguienteAperitivo == 0){
        addPercept("robot", Literal.parseLiteral("siguienteAperitivo(tortilla)"));
        } else if (model.siguienteAperitivo == 1){
        addPercept("robot", Literal.parseLiteral("siguienteAperitivo(empanada)"));
        } else if (model.siguienteAperitivo == 2){
```

```
        addPercept("robot", Literal.parseLiteral("siguienteAperitivo(bocata)"));
        }

boolean quitarPlatosAlacena(int platos) {
   platosEnAlacena -= platos;
   return true;
}

boolean anadirPlatosAlacena(int platos) {
   platosEnAlacena += platos;
   return true;
}

boolean anadirPlatosLavavajillas(int platos) {
   platosEnLavavajillas += platos;
   return true;
}

boolean vaciar_lavavajillas() {
   anadirPlatosLavavajillas(platosEnLavavajillas);
   platosEnLavavajillas = 0;
   return true;
}
```

Las 4 funciones tienen un comportamiento similar, permiten añadir o quitar un cantidad X de platos del lavavajillas o la alacena.

```
boolean prepararPincho(String pincode) {
        if(pincode.equals("tortilla")) {
            pinchosTortilla += 5;
        } else if (pincode.equals("empanada")) {
            pinchosEmpanada += 5;
        } else if (pincode.equals("bocata")) {
            pinchosBocata += 5;
        }
        return true;
}
```

En esta función ayuda al robot a hacer las raciones de los pinchos, dependiendo del pincode añadirá 5 raciones de un tipo u otro, el pincode es indicado por el robot.

```
boolean getPincho() {
        if(pinchosTortilla > 0){
            pinchosTortilla --;
```

```
        } else if (pinchosEmpanada > 0) {
                pinchosEmpanada --;
        } else if (pinchosBocata > 0) {
                pinchosBocata --;
        }
        return true;
}
```

Esta función se ejecuta cuando el robot llega a la nevera a por pinchos y cerveza. Dependiendo de la cantidad de pinchos que haya, cogerá de un tipo u otro.

```
} else if (action.getFunctor().equals("vaciar_papelera") & ag.equals("incinerador")) {
        try {
                model.quemandoBasura = true;
                result = model.vaciar_papelera();
        } catch (Exception e) {
                logger.info("Failed to execute action tirarLata! " + e);
 }

 boolean vaciar_papelera() {
         if (view != null){
                 view.update(lIncinerador.x, lIncinerador.y);
         }
         try{
                 Thread.sleep(2000);
         }catch (Exception e) {
                 logger.info("Failed to execute action tirarLata! " + e);
         }
        cansInBin = 0;
        quemandoBasura = false;
        if (view != null)
                 view.update(lIncinerador.x, lIncinerador.y);
                 view.update(lBin.x, lBin.y);
        return true;
        }

 case 5:
        Location lIncinerador = hmodel.getAgPos(5);
        if(hmodel.quemandoBasura){
                c = new Color(0xE46545);
        }else{
                c = new Color(0x6EAF79);
        }
        super.drawAgent(g, x, y, c, -1);
        g.setColor(Color.black);
```

```
		o = "Incinerador";
		super.drawString(g, x, y, defaultFont, o);
		break;
```

Estas funciones son las encargadas de cambiar el color del incinerador cuando está quemando la basura. De esta manera, que cuando llama a la función *vaciar_papelera*, el valor del booleano cambia de false a true, cambiando en el HouseView, de verde a rojo. Una vez que "termina" de quemar la basura, se vuelve al verde.

```
boolean throwBeerCan() {
		lCan = getAgPos(1);
		boolean notValid = true;
		do {
			notValid = false;
			lCan = new Location((int) (Math.random() * 4 + 1), (int) (Math.random()
* 4 + 1));
			for (Location lObs : lObstaculos) {
				if (sameLocation(lObs, lCan)) {
					notValid = true;
				}
			}
		} while (notValid);

		setAgPos(1, lCan);
		beerCanShow = true;
		if (view != null) view.update(lCan.x, lCan.y);
		return true;
}
```

En esta función se decide donde va a caer la lata, evitando que caiga en los bordes y en la misma posición que los obstáculos.

```
ArrayList<Location> lObstaculos = new ArrayList<Location>();
Location lObstaculo1 = newObstacle();
Location lObstaculo2 = newObstacle();
Location lObstaculo3 = newObstacle();
Location lObstaculo4 = newObstacle();
Location lObstaculo5 = newObstacle();
Location lObstaculo6 = newObstacle();
Location lObstaculo7 = newObstacle();

Location newObstacle() {
		Location lnewObs;
		boolean repetir;
```

```
	do {
		repetir = false;
		lnewObs = new Location((int) (Math.random() * 6 + 2), (int)
(Math.random() * 6 + 2));
		for (Location lObs : lObstaculos) {
			if (sameLocation(lnewObs, lObs)) {
				repetir = true;
			}
		}
	} while (repetir);
	lObstaculos.add(lnewObs);
	return lnewObs;
}
```

Y por último, la colocación de obstáculo, está a igual que la de latas, evita que se coloque en los bordes y unos encima de otros.