Memoria 2

Advertencias del código

Para que no haya problema con el entendimiento del código dejamos este apartado para indicar como tenemos algunas cosas creadas o con que nombre lo hemos dejado:

```
abastecedor.asl -> proveedor.asl
rmayordomo -> myRobot (cambiado en el view)
rlimpiador -> Cleaner (cambiado en el view)
rpedidos -> Deliver (cambiado en el view)
rbasurero -> Burner (cambiado en el view)
```

Supermercado

Número variable de supermercados

Para comenzar, indicar que en DomesticRobot.mas2j tenemos definidos dos supermercados de la siguiente manera:

```
supermarket1 supermarket.asl;
supermarket2 supermarket.asl;
```

Al tenerlos así definidos, tanto supermarket1 como supermarket2 comparten el mismo código, el que se encuentra en supermarket.asl.

Gestión de múltiples Productos y marcas

1. Múltiples marcas de cervezas
   Una vez recibidos los precios y el stock de las distintas cervezas del abastecedor, el supermercado tendrá los siguientes beliefs:

   ```
   price(beer, B, aguila).
   price(beer, C, estrella).
   price(beer, D, volldamm).
   price(beer, A, heineken).
   price(beer, E, redvintage).
   ```

   Los supermercados tendrán estos beliefs desde el primer momento y serán los encargados de gestionarlos, tanto a la hora de darle información al mayordomo (myRobot) como a la hora de que alguien le haga algún pedido. Además, ‘price’ se compone de tres parámetros: price(cerveza, precio, marca). Los precios, como veremos más adelante, serán calculados por el abastacedor, de ahí que estén definidos mediante letras, cada una posee un valor.

   En cuanto al stock:

   ```
   stock(beer, 50, estrella).
   stock(beer, 50, aguila).
   stock(beer, 50, volldamm).
   stock(beer, 50, redvintage).
   stock(beer, 50, heineken).
   stock(pinchito, 20, tortilla).
   stock(pinchito, 20, empanada).
   ```

   Como en el caso de los beliefs de ‘price’ los de stock son iguales pero el parámetro del medio, en lugar de ser el precio, es la cantidad.

2. Múltiples tipos de productos

   Los productos, que también se los envía el abastecedor al supermercado, se definen igual que los precios de las cervezas, es decir, tendrán los beliefs tal que así:

```
price(pinchito, F, tortilla));
price(pinchito, G, empanada));
```

En lugar de pasarle como primer parámetro una cerveza, le pasamos un pinchito, con un precio establecido por el abastecedor, y el tipo de pinchito que tenemos, en este caso, o empanada, o tortilla.

Gestión del dinero y actualización de stocks

Antes de hablar de las ventas y compras, primero es importante comentar que al principio del código, lo que hacen los supermercados es lanzar un objetivo que a parte de mandarle un mensaje al abastecedor, calcula y estable su dinero base:

```
/*Initial goal*/
!recibir_stockyprecio.

/* Plan */
+!recibir_stockyprecio : true
  <- .random(X);
     F = X*1000000+1;
     +moneySuper(F);
     .send(abastecedor, tell, dar_precioystock);
     .wait(100).
```

1. Ventas (al robot/owner)

   Para crear un sistema de ventas, hemos realizado una cadena entre owner, mayordomo y los supermercados, de tal forma que primero, el owner le pide las cervezas y los pinchitos disponibles al mayordomo, de esta forma:

   ```
   +!pide_lista_productos
     <-
     .send(rmayordomo, achieve, lista_productos(beer));
     .send(rmayordomo, achieve, lista_productos(pinchito));
     !escoge_cerveza;
     !escoge_pinchito.
   ```

   De esta forma, le preguntamos al mayordomo por la lista de productos que dispone (se llama a este plan al principio del código ‘!pide_lista_productos’ para que sea lo primero que haga. A su vez, el mayordomo también le pregunta a los supermercados la lista de productos disponibles para poder enviársela al owner:

   ```
   +!pide_lista_productos_super
     <- .send(supermarket1, achieve, lista_productos(beer));
        .send(supermarket2, achieve, lista_productos(beer));
        .send(supermarket1, achieve, lista_productos(pinchito));
        .send(supermarket2, achieve, lista_productos(pinchito)).
   ```

   Al igual que pide_lista_productos de owner, pide_lista_productos_super del mayordomo se llama al principio del código ‘!pide_lista_productos_super’.

   Una vez el mayordomo le envía un mensaje a los supermercados, estos le responden enviándole una lista con los precios disponibles:

   ```
   +!lista_productos(beer)[source(rmayordomo)]: price(beer, _, _) <-
     .println("Envío de precio al robot mayordomo");
     .findall(q(M, C), price(beer, C, M), L);
     .print("Cervezas disponibles: ", L);
     .send(rmayordomo, tell, seleccion_productos(beer, L)).

   +!lista_productos(pinchito)[source(rmayordomo)]: price(pinchito, _, _) <-
     .println("Envío de precio al robot mayordomo");
     .findall(q(M, C), price(pinchito, C, M), L);
     .print("Pinchito disponibles: ", L);
     .send(rmayordomo, tell, seleccion_productos(pinchito, L)).
   ```

   Lo que hace el plan lista_productos es, a través del producto que se le envía, por ejemplo, cerveza, y teniendo el belief actual de price(beer, _, _) (’_’ quiere decir que no nos interesa el valor, lo descarta, porque solo queremos mirar si tiene un belief con un price de tipo beer) buscamos todos esos beliefs con .findall, cogemos solamente la marca de la cerveza y el

precio ‘q(M,C)’ y la guardamos en la lista ‘L’. Por último le enviamos esa lista al mayordomo mediante seleccion_productos(beer, L). Para los pinchitos es igual

Una vez que le llego al mayordomo seleccion_productos(beer, L), pasamos a enviarle esa lista de ambos supermercados al owner, para ello una vez llegado la selección de productos de ambos, mayordomo los junta y se los envía al owner:

```
+!lista_productos(beer): seleccion_productos(beer, L1)[source(supermarket1)] & seleccion_productos(beer, L2)[source(supermarket2)]
  <-.concat(L1,L2,L3);
    .send(owner, tell, seleccionProductos(beer,L3)).

+!lista_productos(beer): true
  <- .print("Aun no llegaron los productos");
    .wait(100);
    !lista_productos(beer).

+!lista_productos(pinchito): seleccion_productos(pinchito, L1)[source(supermarket1)] & seleccion_productos(pinchito, L2)[source(su
  <-.concat(L1,L2,L3);
    .send(owner, tell, seleccionProductos(pinchito,L3)).

+!lista_productos(pinchito): true
  <- .print("Aun no llegaron los productos");
    .wait(100);
    !lista_productos(pinchito).
```

Nota: si el plan lista_productos todavía no tiene la seleccion_productos, pasaría a entrar por el siguiente plan de lista_productos, se indica que todavía no han llegado y se vuelve a llamar a lista_productos de forma recursiva.

Una vez que se le envía la lista al owner y este elije cerveza, lo que tiene que hacer el mayordomo es evitar que la nevera se quede vacía, para ello disponemos de un método comprar para que se pueda pedir al supermercado más productos y además elegir entre ambos el que esté al mejor precio (este plan esta explicado en la sección Ecosistema de Agentes, MYROBOT, gestión de las compras de cervezas preferida en un supermercado.). Lo que hace es enviarle un mensaje a los supermercados llamado order(owner,T, NbeerPinch, M) que activara el plan definido en supermarket llamado order:

```
+!order(Owner, T, C, M)[source(Ag)] : stock(T, P, M2) & P >= C & M = M2 // comprueba la cantidad de stock
  <-
     !actualiza_order_id(OrderId); //1: se actualiza el order_id
     !decrementa_stock(T, C, M ,P);//3: actualiza stock//2: actualiza stock
     ?price(T, Precio, M);
     G = Precio * C;
     +pending_order(OrderId, G, T, M, C);
     .send(Ag, tell, order_aceptado(Owner, OrderId, G)).//5: Actualiza a rmayordomo y al Agt que lo invocó

+!order(_, T, C, M)[source(Ag)] : stock(T, P, M) & P < C
  <- .print("Out of stock: ", T);
     .send(abastecedor, tell, dar_nuevo_stock(T, P, M));
     .wait(100);
     !order(Owner, T, C, M).
```

- comprobamos los parámetros del order con los de stock para comprobar que la marca sea la misma (M = M2) y haya una cantidad mayor en el stock que la cantidad que estemos solicitando comprar (P >= C) , en caso de que P<C, entonces entrará por el segundo plan de order, indicaremos que estamos fuera de stock con el .print, llamamos al abastecedor para que nos reponga el stock y volvemos a llamar de nuevo al plan order.
- Actualizamos el order id:

  ```
  +!actualiza_order_id(OrderId)
    <- ?last_order_id(A);
      OrderId = A + 1;
      -+last_order_id(OrderId).
  ```

  Preguntamos cuál es el ultimo order id y le sumamos 1, acto seguido, refrescamos el belief que preguntamos al principio de este plan.

- Decrementamos el stock:

  ```
  +!decrementa_stock(T, C, M, P)
    <- -stock(T, P, M);
  ```

```
    +stock(T, P-C, M).
```

Primero quitamos el belief de stock actual y luego lo volvemos a actualizar restando al segundo parámetro la cantidad que mayordomo quiere comprar.

- Consultamos el belief price(T, Precio, M) actual del producto y multiplicamos el precio por la cantidad que quiere el mayordomo.
- Añadimos un belief pending_order donde le pasamos el OrderId, el Precio Total, el tipo de producto, la marca del producto y la cantidad.
- Le enviamos al mayordomo un order_aceptado y le pasamos el agente owner, el orderId y el precio total.

Una vez que le llega al mayordomo el order_aceptado, le envía el precio al owner para que realice el pago que, a su vez le envía el dinero al mayordomo a través de pago_cervezaypinchito(Pago, OrderId, Supermarket). El mayordomo lo gestionará haciéndole un pago_order(OrderId, C) al supermercado escogido, como veremos ahora:

```
+pago_cervezaypincho(C, OrderId, Supermarket)[source(Agt)]
  <-  ?money(M);
    L = M + C;
    -money(M);
    +money(L);
    .send(Supermarket, tell, pago_order(OrderId, C)).
```

Como podemos ver, pregunta por su dinero, realiza la suma del dinero que tiene y el que le envió el owner, lo guarda en 'L', quita el belief money(M) y mete el nuevo money(L) con el dinero actualizado. Por último, se envía al supermercado el pago_order con el Id del pedido y el precio que ha pagado el owner.

Cuando le llega el pago_order al supermercado, este lo gestiona de la siguiente manera:

```
+pago_order(OrderId, P)[source(Agt)]: pending_order(OrderId, P, T, M, C)
  <- !actualizar_moneySuper(P);
    deliver(T, P);
    .send(Agt, tell, delivered(T,P,OrderId, M, C));
    -pending_order(OrderId, P, T, M, C).

+pago_order(OrderId, P)[source(Agt)]
  <-
    ?pending_order(Order, P, T, M, C);
    .print(OrderId," ", Order," ", P," ", Pp).
```

- Primero comprueba que tiene un pedido pendiente (pending_order)
- Luego actualiza el dinero del super, llamando al plan actualizar_moneySuper(P):

```
+!actualizar_moneySuper(P):  moneySuper(A)
  <-  -moneySuper(A);
      +moneySuper(A + P).
+!actualizar_moneySuper(P)
  <-  .print("No se puede actualizar dinero").
```

Aquí podemos ver como se quita el belief actual con el dinero del super (-moneySuper(A)) para posteriormente actualizarlo con el nuevo dinero (moneySuper(A + P)).

- A continuación se llama a la función deliver(T, P) definida en java.
- Entregamos al mayordomo el producto solicitado a través del mensaje .send y enviándole un delivered.
- Eliminamos el belief pending_order ya que finalizó la ejecución de ese pedido y ya no está pendiente.

2. Compras (proveedor)

   La compra se realiza al abastecedor una vez que los supermercados le piden un nuevo stock y un precio a los productos, en el plan dar_precioystock de abastecedor:

El plan es el mismo que actuaizar_moneySuper (explicada anteriormente), pero en vez de actualizar sumando, actualiza restando como resultado de las compras que ha realizado. Destacar además que el order (el segundo plan ya explicado anteriormente), también se realiza una compra cuando el supermercado se queda sin stock.

```
+!actualizar_moneySuperNeg(P):  moneySuper(A)
  <-  -moneySuper(A);
    +moneySuper(A - P).
+!actualizar_moneySuperNeg(P)
  <-  .print("No se puede actualizar dinero").
```

Proveedor

Como da el proveedor los precios y stock al supermercado

Para el proveedor ocurre lo siguiente. Primero el supermercado le pide el stock posible que puede hacer el supermercado con un precio random de cada producto, siendo de la siguiente manera:

```
!recibir_stockyprecio. // Initial goal del supermercado
+!recibir_stockyprecio : true
  <-
    .random(X);
    F = X*1000000+1;
    +moneySuper(F);
    .send(abastecedor, tell, dar_precioystock);
    .wait(100).
```

Nota: en esta parte también se genera el dinero random.

El resultado dado del tell al abastecedor se divide la obtención según el supermarket que le dio la orden del tell dar_precioystock

```
+dar_precioystock[source(supermarket1)] : true
  <-  .send(supermarket1, tell, stock(beer, 50, estrella));
    .send(supermarket1, tell, stock(beer, 50, aguila));
    .send(supermarket1, tell, stock(beer, 50, volldamm));
    .send(supermarket1, tell, stock(beer, 50, redvintage));
    .send(supermarket1, tell, stock(beer, 50, heineken));
    .send(supermarket1, tell, stock(pinchito, 20, tortilla));
    .send(supermarket1, tell, stock(pinchito, 20, empanada));
    .random(X);
    A = X*10+1;
    .random(Y);
    B = Y*10+1;
    .random(W);
    C = W*10+1;
    .random(Z);
    D = Z*10+1;
    .random(N);
    E = N*10+1;
    .random(H);
    F=H*10+1;
    .random(I);
    G=I*10+1;
    .send(supermarket1, tell, price(beer, B, aguila));
    .send(supermarket1, tell, price(beer, C, estrella));
    .send(supermarket1, tell, price(beer, D, volldamm));
    .send(supermarket1, tell, price(beer, A, heineken));
    .send(supermarket1, tell, price(beer, E, redvintage));
    .send(supermarket1, tell, price(pinchito, F, tortilla));
    .send(supermarket1, tell, price(pinchito, G, empanada));
    .send(supermarket1, achieve, actualizar_moneySuperNeg(50*A+50*B+50*C+50*D+50*E+20*F+20*G)).
+dar_precioystock[source(supermarket2)] : true
  <-  .send(supermarket2, tell, stock(beer, 50, estrella));
    .send(supermarket2, tell, stock(beer, 50, aguila));
    .send(supermarket2, tell, stock(beer, 50, volldamm));
    .send(supermarket2, tell, stock(beer, 50, redvintage));
    .send(supermarket2, tell, stock(beer, 50, heineken));
    .send(supermarket2, tell, stock(pinchito, 20, tortilla));
    .send(supermarket2, tell, stock(pinchito, 20, empanada));
    .random(X);
    A = X*10+1;
    .random(Y);
    B = Y*10+1;
```

```
    .random(W);
    C = W*10+1;
    .random(Z);
    D = Z*10+1;
    .random(N);
    E = N*10+1;
    .random(H);
    F=H*10+1;
    .random(I);
    G=I*10+1;
    .send(supermarket2, tell, price(beer, B, aguila));
    .send(supermarket2, tell, price(beer, C, estrella));
    .send(supermarket2, tell, price(beer, D, volldamm));
    .send(supermarket2, tell, price(beer, A, heineken));
    .send(supermarket2, tell, price(beer, E, redvintage));
    .send(supermarket2, tell, price(pinchito, F, tortilla));
    .send(supermarket2, tell, price(pinchito, G, empanada));
    .send(supermarket2, achieve, actualizar_moneySuperNeg(50*A+50*B+50*C+50*D+50*E+20*F+20*G)).
```

El código lo podríamos dividir en tres partes: primero tenemos los send de stock los cuales dan el stock de cada supermercado, en este caso, de cada producto tanto beer y su tipo como pinchito y el tipo de este, lo segundo son el precio de cada producto y, por ultimo, la aleatoriedad de los precios.

```
.send(supermarket1, tell, stock(beer, 50, volldamm));
.send(supermarket1, tell, stock(beer, 50, redvintage));
.send(supermarket1, tell, price(beer, E, redvintage));
.send(supermarket1, tell, price(pinchito, F, tortilla));
```

También, para recalcar habría que comentar que los precios son aleatorios esto es tan fácil como usar el .random() con una variable de por medio y luego como la variable es un valor entre 0 y 1 hay que multiplicarla por 10 y sumarle 1 por si es de un 0 o más.

```
.random(X);
A = X*10+1;
```

Nota: una vez el stock y precios es recibido se le resta en conjunto todo esto a los supermarkets con este código:

```
.send(supermarket2, achieve, actualizar_moneySuperNeg(50*A+50*B+50*C+50*D+50*E+20*F+20*G)).
+!actualizar_moneySuperNeg(P):  moneySuper(A)
  <-  -moneySuper(A);
      +moneySuper(A - P).
+!actualizar_moneySuperNeg(P)
  <-  .print("No se puede actualizar dinero").
```

Actualización de productos cuando el supermercado los solicite (stock es menor al necesitado)

Para esto ocurre una condición la cual es que el stock necesario para un order es menor dándonos este código:

```
+!order(_, T, C, M)[source(Ag)] : stock(T, P, M) & P < C
  <- .print("Out of stock: ", T);
     .send(abastecedor, tell, dar_nuevo_stock(T, P, M));
     .wait(100);
     !order(Owner, T, C, M).
```

Cuando ocurre esta condición se le manda un tell al abastecedor que nos de ese producto de nuevo con un stock y un precio random obviamente calculado de la forma explicada anteriormente.

```
+dar_nuevo_stock(T, P, M) [source(Agt)] <- .send(Agt, tell, stock(T, 50, M)); .random(Y); A=Y*10+1; .send(Agt, tell, price(T,A,M)).
```

Ecosistemas de Agentes

Owner

Colaboración/cooperación solicitud de cervezas

Como ya se ha descrito en Supermercado, podemos ver que el owner colabora y coopera con el mayordomo y el propio super tanto a la hora de la elección de los productos como en el pago de los mismos.

Primero, podemos ver que se llama a !pide_lista_productos, que lo que hace es preguntarle al mayordomo por los productos disponibles. A su vez, mayordomo le pide a lista al supermercado, que este se lo enviará al mayordomo, para que finalmente se la vuelva a enviar al owner:

```
+!pide_lista_productos
  <-
  .send(rmayordomo, achieve, lista_productos(beer));
  .send(rmayordomo, achieve, lista_productos(pinchito));
  !escoge_cerveza;
  !escoge_pinchito.
```

Una vez que el owner pide la lista al mayordomo, este escoge tanto una cerveza como un pinchito de la lista que le envía de vuelta el mayordomo. Vayamos a ver estos planes:

```
+!escoge_cerveza: seleccionProductos(beer, L1)[source(rmayordomo)]
  <- .random(L1, X);
     !despieza(X, M, _); // _ = no me interesa valor
     .print("Cerveza elegida: ", M);
     .send(rmayordomo, tell, cerveza_escogida(M)).

+!escoge_cerveza
  <- .wait(100); //Esperando por la cerveza TODO: intentar solucionar recursividad
     !escoge_cerveza.

+!escoge_pinchito : seleccionProductos(pinchito, L1)[source(rmayordomo)]
  <- .random(L1, X);
    !despieza(X, M,_);
    .print("Pinchito elegido: ", M);
    .send(rmayordomo, tell, pinchito_escogido(M)).

+!escoge_pinchito
  <- .wait(100); //Esperando por la cerveza TODO: intentar solucionar recursividad
     !escoge_pinchito.
/*
```

Como vemos en estos planes, es necesario pasarles la lista que nos da el mayordomo y, a partir de ella, elegimos un producto aleatorio de esa lista, que contiene la marca del producto y el precio. Después de esto, llamamos a despieza(X, M, _) que es un plan que creamos nosotros para que dada una lista, nos separe los elementos que queramos de ella. En este caso solamente nos interesa la marca, que es lo que le vamos a enviar al mayordomo después de llamar a despieza. En caso de que no tenga todavía la lista que la proporciona el mayordomo, espera 100ms y llama recursivamente a este plan, hasta que tenga la lista. A continuación muestro el plan despieza:

```
+!despieza([],[],[]).
+!despieza(q(X,Y),X,Y).
```

El primer despieza simplemente es por si tenemos una lista vacía, no hace nada. El segundo, dados una lista de dos elementos ‘q(X, Y)’ lo que hace es retornar los dos elementos por separado X, Y.

Tanto cerveza_escogida(M) como pinchito_escogido(M), se usará en mayordomo para traerle la cerveza que owner está pidiendo. Por lo que podemos ver como se va pidiendo y pasando la información desde supermercado hasta el owner (incluso desde abastecedor, que es quien le pasa los beliefs al supermercado que luego este le pasará la información al mayordomo y mayordomo al owner)

Por otro lado, también podemos ver esta colaboración/cooperación en ‘+pagar_cervezaypincho’, ya que, nuevamente cooperan entre ellos para que le llegue el dinero al supermercado. Cuando le llega un order_aceptado a mayordomo desde un supermercado, este le envía un pagar _cervezaypincho:

```
+pagar_cervezaypincho(C, OrderId, Supermarket)[source(Agt)]
  <-  !restarDinero(C);//TODO: se asume que restar dinero va a salir bien
    .send(Agt, tell, pago_cervezaypincho(C, OrderId, Supermarket));
    -pagar_cervezapincho(C, OrderId, Supermarket).
```

Este plan llama a restar dinero para descontar el dinero de su pedido de su monedero:

```
+!restarDinero(C): C == 0 <- true.

+!restarDinero(C) : money(M) & M >= C
  <-  .print("Cerveza y pincho pagado.");
    L = M - C;
    -money(M);
    +money(L).

+!restarDinero(C) : money(M) & M < C
  <- .print("Cantidad de dinero insuficiente.");
     false.
```

Lo que hace simplemente es restar el dinero de los productos y actualizar el nuevo belief money(L) con la nueva cantidad. Si la cantidad es menor al precio del producto retorna false y dice que no tiene dinero suficiente.

Después de restar el dinero le envía un pago_cervezaypincho al mayordomo con el dinero de los productos, el orderId para saber que pedido es y el supermercado al que el mayordomo tiene que ingresar el dinero de los productos. Una vez recibido el dinero del owner, el mayordomo se lo envía de nuevo al supermercado escogido. Esto también demuestra la colaboración y cooperación entre estos tres agentes para el pago de los productos, tanto de pinchito como de cervezas.

Colaboración/Cooperación reciclado de latas y owner movible (no fijo)

La colaboración o cooperación del reciclado de latas lo tenemos implementado de tal forma que el mayordomo es el encargado de elegir quien irá a buscar la basura (trash, latas). Esto se realiza en el plan escoge_agente del mayordomo:

```
+!escoge_agente(Agt)
  <- .random([owner, rlimpiador], Agt);
    .print("El agente que va a tirar la basura es: ", Agt).
```

Como podemos ver, elige al owner o a robot limpiador y el elegido será el encargado de ir a buscar la lata. Este plan se llama en el primer método bring, cuando el mayordomo detecta que hay basura en el entorno:

```
+!bring(owner,beer, pinchito)[source(self)]:  trashInEnv(T) & T>0 & not entornoLimpio & cerveza_escogida(M) & pinchito_escogido(P)
   <- .println("El robot mayordomo revisa si hay basura");
      +entornoLimpio;
    !escoge_agente(Agt);
      .send(Agt, tell, hay_basura(Agt,trash));
      !bring(owner, beer, pinchito).
```

Si hay basura en el entorno (trashInEnv(T) & T>0 & not entornoLimpio), entonces escoge a un agente para que vaya a recoger la basura. Al agente escogido, le llegará un mensaje indicándole que hay basura pendiente, en caso del owner será el siguiente (muy similar a los planes de rlimpiador):

```
+hay_basura(owner, trash) [source(rmayordomo)]
   <- !recogerBasura(owner, trash);
      !tirarBasura(owner, bin);
      !go_at(owner, couch);
    !hay_papeleraBin(rbasurero, bin);
      .abolish(hay_basura(owner, trash));
    .send(rmayordomo, achieve, limpiezaTerminada).
```

Cuando le llega este mensaje, llama a recoger basura, que como su nombre indica se encarga de ir a recoger la basura:

```
+!recogerBasura(owner, trash) : trashInEnv(T) & T > 0
   <- !go_at(owner, trash);
      pickTrashO(owner,trash);
      ?trash(can, C);
      -+trash(can, C+1);
      !recogerBasura(owner,trash).
+!recogerBasura(owner, trash).
```

El owner se desplaza desde su ubicación actual hasta la lata (go_at). Después, se encarga de coger esa lata encontrada, pregunta cuantas latas hay (gracias al belief de trash(can, C) que lleva la cuenta de estas) y actualiza el belief en +1. Luego se llama recursivamente para comprobar que no hay más basura, una vez que no hay acaba el plan y se llama al siguiente, que es tirarBasura:

```
+!tirarBasura(owner, bin): trash(can, X) & X>0
   <- !go_at(owner, bin);
      !desechar(owner, trash).
+!tirarBasura(owner, bin).
```

Este plan lo que hace simplemente es ir con la lata hasta la basura y llamar a desechar:

```
+!desechar(owner, trash) : trash(can, X) & X>0
   <- desechar(owner,trash);
      -+trash(can, X-1);
      !desechar(owner, trash).
+!desechar(owner,trash)
   <- .println("Owner ha depositado toda la basura en el cubo").
```

Desechar comprueba el belief actualizado en el plan de recogerBasura, y si es mayor a 0 (si hay), lo que hace es llamar a desechar de java que deja la lata en la basura, actualiza el belief de trash en -1 y se llama recursivamente mientras siga habiendo alguna lata. Cuando no hay ninguna lata, entra por el plan de abajo, indicando que ya se han depositado todas las latas y pasa al siguiente plan, hay_papeleraBin:

```
+!hay_papeleraBin(rbasurero, bin) <-
  .send(rbasurero, tell, papelera_llena(rbasurero, bin));
  .wait(1000);
  .send(rbasurero, untell, papelera_llena(rbasurero, bin)).
```

Simplemente este plan le manda un mensaje al robot basurero diciéndole que hay basura en la papelera para que la elimine definitivamente, espera un segundo, y se lo elimina para que no queden acumulados y evitar que se produzcan errores.

Por último, como se hizo también en papelera_llena del robot basurero, eliminamos hay_basura, para evitar errores y le enviamos un último mensaje al mayordomo indicándole que la limpieza ha sido terminada.

Estos planes añadidos a hay_basura, también los tiene el robot limpiador, por lo que cooperarán entre ellos para que no vaya siempre el mismo a recoger y a tirar la basura.

Gestión del dinero

El dinero del owner se gestiona de la siguiente forma:

- Primero le asignamos un dinero aleatorio a través del plan +!money_aleatorio:

  ```
  +!money_aleatorio : true
    <- .random(X);
      F = X*10000+1;
      +money(F).
  ```

- También dispone de un método de restar dinero para cuando tiene que hacer el pago de algún producto, ya mencionado anteriormente

  ```
  +!restarDinero(C): C == 0 <- true.

  +!restarDinero(C) : money(M) & M >= C
    <-  .print("Cerveza y pincho pagado.");
      L = M - C;
      -money(M);
      +money(L).

  +!restarDinero(C) : money(M) & M < C
  ```

```
   <- .print("Cantidad de dinero insuficiente.");
      false.
```

- Además del también comentado +pagar_cervezaypincho, que cuando mayordomo le indica a owner que pague los productos solicitados, este actualiza su monedero restando el valor de los productos y se lo envía al owner:

```
+pagar_cervezaypincho(C, OrderId, Supermarket)[source(Agt)]
  <-  !restarDinero(C);//TODO: se asume que restar dinero va a salir bien
    .send(Agt, tell, pago_cervezaypincho(C, OrderId, Supermarket));
    -pagar_cervezapincho(C, OrderId, Supermarket).
```

MyRobot

Para el myRobot tenemos lo siguiente, es el encargado de varias cosas a parte de entregar cerveza y pinchos sino que también tiene que llevar los platos sucios al lavavajillas ponerlo y llevar esos platos a la lacena y gestión de la compra de cervezas y comida.

Servir cervezas y pinchos

Primero, tenemos la parte de servir, el owner en un momento dado le da la orden al rmayordomo de ir a la nevera a coger una cerveza y un pincho que seria de la siguiente manera:

```
//Owner
+!bebe(owner, beer) : not has(owner, beer) & not asked(beer)
   <- .println("Owner no tiene cerveza y un pincho.");
      .random(X);
      !get(beer);
      !bebe(owner, beer).
+!get(beer) : not asked(beer)
   <- .send(rmayordomo, tell, asked(beer));//1: Preguntarle a mayordomo por una cerveza
      .println("Owner ha pedido una cerveza al robot mayordomo.");
      +asked(beer); //4: actualizar belief de que ya se ha pedido una cerveza
      .wait(200).
// Codigo referente del rmayordomo para ir a por el pincho y la cerveza
+!bring(owner,beer, pinchito)[source(self)]:  available(beer,fridge) & not too_much(beer) & asked(beer) & cerveza_escogida(M) & pinchi
   <- .println("El robot mayordomo va a buscar una cerveza");
      !go_at(rmayordomo,fridge);
      open(fridge);
      get(beer, pinchito);
      ?cerveza_escogida(M);
      ?pinchito_escogido(P);
      !comprar(supermarket, beer, M);
      .wait(100);
      !comprar(supermarket, pinchito, P);
      close(fridge);
      !go_at(rmayordomo,couch);
      .wait(1000);
      !hasBeer(owner);
      .abolish(asked(beer));
      !bring(owner,beer, pinchito).
//get(beer, pinchito) es el que se usa para recoger la racion de comida con el plato y cerveza
```

Nota: aunque no este pinchito como parámetro la gestión sigue siendo correcta de todo.

El bebe se hace de forma recursiva según unas creencias, en este caso, es cuando no tiene una cerveza en mano y no pregunto por una (asked(beer)) y con el !get(beer) este le manda una creencia al rmayordomo que es asked(beer) y este va hacia la nevera y coge una cerveza y un pinchito y se lo entrega al owner.

Para saber si el owner esta tomando la cerveza y comiendo el pincho se hace a través del plan sip(beer):

```
// Codigo sip(beer) este incluye la utilizacion del pinchito
+!sip(beer): has(owner,beer) & asked(beer)
   <- .println("Owner va a empezar a beber cerveza y comer un pincho.");
      -asked(beer);
      sip(beer);
      !sip(beer).
+!sip(beer): has(owner,beer) & not asked(beer)
   <- sip(beer);
      .println("Owner está bebiendo cerveza y comiendo un pincho.");
      !sip(beer).
```

Nota: el sip(beer) que no es creencia se utiliza en el .java en este caso pasa por el HouseEnv como una acción y luego termina en el HouseModel de la siguiente manera.

Recoger platos sucios y llevarlos al lavavajillas y lacena por MyRobot

Lo primero es que en este caso por como creamos el código los platos se van almacenando en el owner lo cual hace que se cree una pila de ellos que luego se recogerán una vez pase el limite de cervezas (limit(beer, 5)).

```
//codigo de generacion de platos sucios
+!recogerplatosucio(Elem) : platoVa(Elem, D) & D>0
 <-   .println("Owner deja su plato sucio.");
      generatePlato(Elem);
      ?contarPlaLav(plato, C);
        E=C+1;
      -+platoVa(Elem, D-1);
      -+contarPlaLav(plato,E);
       if(E >= 5){
      .send(rmayordomo, tell, platosSucios);
      .send(rmayordomo, achieve, recogerPlatoSucio(rmayordomo,plato));
      }.
```

Nota: el código de generarPlato(Elem) es el siguiente (código del HouseModel):

```
// Crear los platos sucios
    boolean generatePlato() {
    platosSucios++;
    return true;
    }
```

Una vez se alcance la condición los platos los recoge el rmayordomo y los lleva al lavavajillas:

```
//Recoleccion de los platos
+!recogerPlatoSucio(rmayordomo,plato) : platosSucios
   <- !go_at(rmayordomo, couch);
      .println("El robot rmayordomo recoge los platos con tranquilidad");
      pickPlato(rmayordomo, plato);
    -platosSucios;
      !desecharP(rmayordomo, plato).
// Dejarlos en el lavavajillas
+!desecharP(rmayordomo, plato)
   <-  !go_at(rmayordomo,lavavajillas);
       desecharP(rmayordomo, plato);
       !lavavajillas_lleno(rmayordomo, lavavajillas).


//Apartado coger platos limpios y llevarlos a la lacena
+!lavavajillas_lleno(rmayordomo, lavavajillas)
    <-  .println("Poniendo el lavavajillas");
        .wait(6000);
        vaciarLa(lavavajillas);
        !go_at(rmayordomo, lacena);
        ponerPla(rmayordomo, lacena);
        !go_at(rmayordomo, baseRMayordomo).
```

Nota: el wait sirve como si el rmayordomo pusiese un programa de lavado de platos. El código del HouseModel de ponerPla o otros es básico principalmente pasar una variable a otra como si fuese un distinto estado de los platos. Por ejemplo:

```
// Codigo de ponerPla
// Poner platos lacena
  boolean ponerPlatos(){
      platosEnLacena+=platosLimpios;
    platosLimpios = 0;
    if (view != null) view.update(lLacena.x,lLacena.y);
            return true;
  }
```

Obviamente para reflejar estos cambios de estado se hace con la opción de view.update

Gestión de pinchos y compra de productos de comida

Para la gestión de pinchos se haría de la siguiente forma, primero el rmayodomo pide la lista de los productos de comida en este caso, tortilla y empanada, a través del siguiente plan:

```
/*
  pide_lista_productos_super/0
  private
  return: le pide a los supermercados que le envíen una lista de productos
*/
+!pide_lista_productos_super
  <- .send(supermarket1, achieve, lista_productos(beer));
     .send(supermarket2, achieve, lista_productos(beer));
     .send(supermarket1, achieve, lista_productos(pinchito));
     .send(supermarket2, achieve, lista_productos(pinchito)).
```

Es un achieve sencillo que le manda al supermarket y este le devuelve los productos, que le denominamos directamente como pinchito.

Lo siguiente es la respuesta de la lista producto por parte del supermarket tanto 1 como 2 y seria de la siguiente forma:

```
+!lista_productos(pinchito)[source(rmayordomo)]: price(pinchito, _, _) <-
  .println("Envío de precio al robot mayordomo");
  .findall(q(M, C), price(pinchito, C, M), L);
  .print("Pinchito disponibles: ", L);
  .send(rmayordomo, tell, seleccion_productos(pinchito, L)).
```

En este comando podemos observar que la pedida de la lista de productos es de la siguiente forma, el findall como su nombre indica encuentra todos los productos que se pueden comprar, a través, de la salida L y este se lo manda a rmayordomo como una creencia la cual es seleccion_productos(pinchito, L).

Luego, el owner le va a pedir la lista de estos productos de comida para obviamente elegir cual quiere en este caso seria su favorito porque su elección es única.

El código de la pedida de la lista de productos seria la siguiente:

```
// La pedidad de la lista de productos
+!pide_lista_productos
  <-
  .send(rmayordomo, achieve, lista_productos(beer));
  .send(rmayordomo, achieve, lista_productos(pinchito));
  !escoge_cerveza;
  !escoge_pinchito.
//Junta la respuesta rmayordomo para mandarsela al owner
+!lista_productos(pinchito): seleccion_productos(pinchito, L1)[source(supermarket1)] & seleccion_productos(pinchito, L2)[source(superm
  <-.concat(L1,L2,L3);
    .send(owner, tell, seleccionProductos(pinchito,L3)).

+!lista_productos(pinchito): true
  <- .print("Aun no llegaron los productos");
    .wait(100);
    !lista_productos(pinchito).
// En esta parte la escoger se aplica un random haciendo que sea su "favorita" porque en la compra
//sería siempre esa la unica que se compra
+!escoge_pinchito : seleccionProductos(pinchito, L1)[source(rmayordomo)]
  <- .random(L1, X);
    !despieza(X, M,_);
    .print("Pinchito elegido: ", M);
    .send(rmayordomo, tell, pinchito_escogido(M)).

+!escoge_pinchito
  <- .wait(100); //Esperando por la cerveza TODO: intentar solucionar recursividad
     !escoge_pinchito.
```

Según ya tenemos su pincho favorito ya sabemos el producto a comprar.

El rmayordomo tiene conocimiento del tipo de pinchito escogido con la creencia pinchito_escogido(M), por tanto cuando se haga la compra de la comida el rmayordomo le dará el tipo de pinchito que tiene que escoger. Con el código siguiente:

```
//El plan se puede encontrar en el bring del rmayordomo
    !comprar(supermarket, pinchito, P);
// El plan va asi
+!comprar(supermarket, T, M) : not ordered(T)
   <-
    !consulta_precio(supermarket1,T, M, PS1);
    !consulta_precio(supermarket2,T, M, PS2);
    L = [q(PS2, supermarket2), q(PS1, supermarket1)];
    .min(L, X);
    !despieza(X, Precio, Supermarket);
      ?nBeerPinchperTime(NBeerPinch);
```

```
    .print("Comprando ", M, " en ", Supermarket);
    .send(Supermarket, achieve, order(owner, T, NBeerPinch, M));
    +ordered(T).

+!comprar(supermarket, T, M).
```

Se hace una consulta del precio de la comida y con el despieza y el mínimo se obtiene el precio menor a pedir y cual supermercado lo tiene.

Luego, se le pasa el order al supermercado de lo que se necesita:

```
/*order/3 -> tipo, cantidad, marca
public: cualquiera(rmayordomo)
return: */
+!order(Owner, T, C, M)[source(Ag)] : stock(T, P, M2) & P >= C & M = M2 // comprueba la cantidad de stock
  <-
     !actualiza_order_id(OrderId); //1: se actualiza el order_id
     !decrementa_stock(T, C, M ,P);//3: actualiza stock//2: actualiza stock
     ?price(T, Precio, M);
     G = Precio * C;
     +pending_order(OrderId, G, T, M, C);
     .send(Ag, tell, order_aceptado(Owner, OrderId, G)).//5: Actualiza a rmayordomo y al Agt que lo invocó
```

Una vez el pago este aceptado, es decir, se le pasó el order a rmayordomo y este envía el dinero a pagar al owner se le manda un pago_order al supermercado el cual es el siguiente:

```
+pago_order(OrderId, P)[source(Agt)]: pending_order(OrderId, P, T, M, C)
  <- !actualizar_moneySuper(P);
    deliver(T, P);
    .send(Agt, tell, delivered(T,P,OrderId, M, C));
    -pending_order(OrderId, P, T, M, C).

+pago_order(OrderId, P)[source(Agt)]
  <-
    ?pending_order(Order, P, T, M, C);
    .print(OrderId," ", Order," ", P," ", Pp).
```

Luego ya va al pedidos que pide al supermercado esa comida y la deja en la nevera.

Luego para la separación del plato a la ración dada esta explicado anteriormente, siendo lo siguiente:

```
// Crear los platos sucios
    boolean generatePlato() {
    platosSucios++;
    return true;
    }
```

Gestión compras cervezas preferidas

Para la gestión de compras de cervezas ocurriría lo mismo que la comida, primero se pide una lista de las cervezas por parte del rmayordomo:

```
/*
pide_lista_productos_super/0
private
return: le pide a los supermercados que le envíen una lista de productos
*/
+!pide_lista_productos_super
<- .send(supermarket1, achieve, lista_productos(beer));
.send(supermarket2, achieve, lista_productos(beer));
.send(supermarket1, achieve, lista_productos(pinchito));
.send(supermarket2, achieve, lista_productos(pinchito)).
```

Luego, el supermarket le devuelve una lista de productos, en este caso, son beer.

Lo siguiente es la respuesta de la lista producto por parte del supermarket tanto 1 como 2 y seria de la siguiente forma:

```
//lista_productos/1 -> tipo
// public: rmayordomo
//return: lista de seleccion_productos(L[price/3])
+!lista_productos(beer)[source(rmayordomo)]: price(beer, _, _) <-
```

```
  .println("Envío de precio al robot mayordomo");
  .findall(q(M, C), price(beer, C, M), L);
  .print("Cervezas disponibles: ", L);
  .send(rmayordomo, tell, seleccion_productos(beer, L)).
```

Cada supermarket le da los productos que tienen al rmayordomo, luego el owner le pide la lista de cervezas:

```
+!pide_lista_productos
  <-
  .send(rmayordomo, achieve, lista_productos(beer));
  .send(rmayordomo, achieve, lista_productos(pinchito));
  !escoge_cerveza;
  !escoge_pinchito.
```

En este caso, nos centramos en cervezas y lo que hace el rmayordomo es enviar la lista de todas las cervezas de cada supermercado con un concat de la siguiente manera:

```
+!lista_productos(beer): seleccion_productos(beer, L1)[source(supermarket1)] & seleccion_productos(beer, L2)[source(supermarket2)]
  <-.concat(L1,L2,L3);
    .send(owner, tell, seleccionProductos(beer,L3)).

+!lista_productos(beer): true
  <- .print("Aun no llegaron los productos");
    .wait(100);
    !lista_productos(beer).
```

Una vez que reciba la lista de productos, el owner escogerá su favorita que será la que se utilizará en todo el proceso de compra y demás.

El código para escoger la cerveza es el siguiente:

```
+!escoge_cerveza: seleccionProductos(beer, L1)[source(rmayordomo)]
  <- .random(L1, X);
     !despieza(X, M, _); // _ = no me interesa valor
     .print("Cerveza elegida: ", M);
     .send(rmayordomo, tell, cerveza_escogida(M)).

+!escoge_cerveza
  <- .wait(100); //Esperando por la cerveza TODO: intentar solucionar recursividad
     !escoge_cerveza.
```

Una vez se tenga la creencia de cual es la cerveza “favorita” en el caso de que se vaya a hacer el plan comprar ya tiene conocimiento de cual es la cerveza que tiene que comprar el rmayordomo y cual tiene que llevar el deliver a la nevera.

El código de la compra y su proceso ya quedo explicado allá arriba, principalmente siempre logra obtener el menor de todos y si no hay stock de este vuelve a pedirlo al proveedor.

Escenario 1: funciona correctamente haciendo que se pida la más barata.

Escenario 2: Compra parcial no exacta, Caso C correcto y para el Caso B vuelve a pedir las cervezas al proveedor.

Escenario 3: No realizado.

Cleaner

En este caso el robot lo hemos denominado como rlimpiador pero, en líneas generales, este robot lo que hace es ir por el grid eliminando las distintas latas y llevarlas a la papelera.

Para que el cleaner este en funcionamiento utilizamos una creencia la cual es TrashInEnv(A), con una cantidad para tener constancia de la cantidad de basura que hay en el grid.

El código para cuando ocurra que se tenga que llamar al cleaner es el siguiente:

```
+!bring(owner,beer, pinchito)[source(self)]:  trashInEnv(T) & T>0 & not entornoLimpio & cerveza_escogida(M) & pinchito_escogido(P)
   <- .println("El robot mayordomo revisa si hay basura");
      +entornoLimpio;
    !escoge_agente(Agt);
      .send(Agt, tell, hay_basura(Agt,trash));
      !bring(owner, beer, pinchito).
```

El código es del rmayordomo y es de su pedida recursiva bring este cuando detecte el trashInEnv se activa para saber este valor primero hay que comentar el código del owner.

El owner una vez tenga la cerveza y acabe la cerveza este tira la basura con el código que es el siguiente:

```
+!bebe(owner, beer) : has(owner, beer)
   <- .println("Owner ya tiene una cerveza y se dispone a beberla.");
      !sip(beer);
      ?trash(can,C);
    ?platoVa(plato, D);
      -+trash(can, C+1);
    -+platoVa(plato, D+1);
      !lanzar(can);
    !recogerplatosucio(plato);
      !bebe(owner, beer).
// El codigo de lanzar es el siguiente
+!lanzar(Elem) : trash(Elem, C) & C>0
   <- .println("Owner va a lanzar ", Elem);
      generateTrash(Elem);
      -+trash(Elem, C-1).
+!lanzar(Elem).
```

En este código se puede observar el generateTrash que va directo al model el cual hace lo siguiente, sabiendo de ante mano que pasa por el env:

```
//HouseEnv codigo
// Acción de tirar una lata en el entorno
  public static final Literal generateTrash = Literal.parseLiteral("generateTrash(can)");
  // Acción de tirar basura en el entorno
    else if(action.equals(generateTrash) && ag.equals("owner")){
            result = model.generateTrash();
    }
//HouseModel codigo
// Desperdigar la basura que tira el owner por el entorno
    boolean generateTrash() {
        Location location;
    int j = ~0;
    int x,y;
        do{
             x = (int) (Math.random()*(GSize-1));
             y = (int) (Math.random()*(GSize-1));
            location = new Location(x, y);
        }while(
            !isFree(j, x ,y)
        );

        add(TRASH, location);
        lTrash.add(location);
        return true;

    }
```

Comentándolo de forma resumida el generateTrash lo que hace es crear la basura en una posición que este libre en este caso el isFree es una operación de Jason, que es la siguiente:

```
/** returns true if the location x,y has not the object obj */
    public boolean isFree(int obj, int x, int y) {
        return inGrid(x, y) && (data[x][y] & obj) == 0;
    }
```

Se tiene constancia de que el trash aumento en el grid con un percept que se apoya del arraylist de trash, que es:

```
// Percepción de basura en el entorno para el agente mayordomo
        if(model.lTrash.size() > 0){
            addPercept("rmayordomo", Literal.parseLiteral("trashInEnv("+model.lTrash.size()+")"));
            addPercept("rlimpiador", Literal.parseLiteral("trashInEnv("+model.lTrash.size()+")"));
            addPercept("owner", Literal.parseLiteral("trashInEnv("+model.lTrash.size()+")"));

    }
```

Y luego el cleaner se aproxima a la posición del trash con ayuda del move_towards y lo recoge, llevándolo posteriormente al bin, el código es el siguiente:

```
// Esto se activa por que se lo manda el rmayordomo
+hay_basura(rlimpiador, trash) [source(rmayordomo)]
   <-

      !recogerBasura(rlimpiador, trash);
      !tirarBasura(rlimpiador, bin);
      .println("El robot limpiador vuelve a su posición");
      !go_at(rlimpiador, baseRLimpiador);
    !hay_papeleraBin(rbasurero, bin);
      .abolish(hay_basura(rlimpiador, trash));
    .send(rmayordomo, achieve, limpiezaTerminada).
// Este plan hace que la basura se recoga
+!recogerBasura(rlimpiador, trash) : trashInEnv(T) & T > 0
   <- .println("El robot limpiador va a buscar basura");
      !go_at(rlimpiador, trash);
      .println("El robot limpiador recoge basura");
      pickTrash(rlimpiador, trash);
      ?trash(can, C);
      -+trash(can, C+1);
      !recogerBasura(rlimpiador,trash).
+!recogerBasura(rlimpiador, trash).

//Con estos doss ultimos planes se hace que se tire la basura
//a la papelera
+!tirarBasura(rlimpiador, bin): trash(can, X) & X>0
   <- !go_at(rlimpiador, bin);
      !desechar(rlimpiador, trash).
+!tirarBasura(rlimpiador, bin).

+!desechar(rlimpiador, trash) : trash(can, X) & X>0
   <- desechar(rlimpiador, trash);
      -+trash(can, X-1);
      !desechar(rlimpiador, trash).
+!desechar(rlimpiador,trash)
   <- .println("El robot limpiador ha depositado toda la basura en el cubo").
```

También para la recogida hay que recalcar que el rlimpiador o cleaner llega a una posición adyacente a este, para agregar el pickTrash que hace la recogida de basura tiene el siguiente código:

```
// Recoger la basura que se encuentra en el entorno
    boolean pickTrash() {
        Location r1 = lTrash.get(0);

        if (hasObject(TRASH, r1)) {
            if(lTrash.contains(r1)){
                lTrash.remove(lTrash.indexOf(r1));

            }
            remove(TRASH, r1);
      return true;
        }
    return false;
    }
```

Para que el cleaner tenga una base como las posiciones se configuraron para ser adyacentes por el tema del move_towards en este caso se hace otra rutina dentro de este para este robot y el owner.

Siendo el código de esta rutina el siguiente:

```
if((ag.equals("rlimpiador") && dest.equals(lRLimpiador)) || (ag.equals("owner") && dest.equals(lCouch))){

      move_to(nAg, lAgent, dest);
    }


  boolean move_to(int nAg, Location ag, Location dest){
    if(ag.isNeigbour(dest)){
      setAgPos(nAg, dest);
      return true;
    }
    return false;
  }
```

Se puede observar mejor probando el código en JASON.

Deliver

Como se ha comentado anteriormente en la pedida de comida aquí, una vez se haga el pago_order, el delivered se manda al mayordomo para que este calcule el dinero y luego le manda la operación a rpedidos o delivery.

El código es el siguiente primero a rmayordomo:

```
// when the supermarket makes a delivery, try the 'has' goal again
+delivered(T,Precio,OrderId,Marca, Cantidad)[source(S)]
  <-  ?money(Money)[source(self)];
      -+money(Money-Precio);
      .send(rpedidos, tell, money(Precio));
      .send(rpedidos, tell, delivered(T, Cantidad, OrderId, S, Marca)).
```

El código de rpedidos es el siguiente:

```
// Este codigo lo que hace es la pedida de la cantidad que se necesita de comida o cerveza
//luego va al fridge y repone la cantidad
+delivered(T, Qtd, OrderId, S, M)[source(rmayordomo)]
   <- .println("El robot de pedidos se dirige a la zona de entrega");
      !go_at(rpedidos, delivery);
      .concat("La orden es de: ", Qtd, " de ", T, Ms);
     .send(S, tell, msg(Ms));
      .abolish(money(_));
      getDelivery(T, Qtd, M);
      .println("El robot de pedidos se dirige al frigorifico");
      !go_at(rpedidos, fridge);
      reponer(T, 3);
      .send(rmayordomo, tell, available(beer, fridge));
      !go_at(rpedidos, baseRPedidos).
```

El código de getDelivery coge en el env dos valores que es lo que se necesita, es decir, cerveza o pinchito y luego lo agrega a una variable. El código es el siguiente:

```
//Codigo del HouseEnv
// Acción de coger cerveza del punto de recogida
        else if(action.getFunctor().equals("getDelivery") && ag.equals("rpedidos")){
            try {
                result = model.getDelivery(action.getTerm(0).toString(),(int)((NumberTerm)action.getTerm(1)).solve());
            } catch (NoValueException e) {
                logger.info("Failed to execute action deliver!"+e);
            }
        }
//Codigo del HouseModel
// Recoger las cervezas del punto de recogida
    public boolean getDelivery(String m, int n) {
    if(m.equals("beer")){
      if (deliveryBeers > 0) {
            deliveryBeers-=n;
            rpedidosBeers+=n;
      if(view != null){
                view.update(lDelivery.x,lDelivery.y);
            }
       return true;
      }
    } else if(m.equals("pinchito")){
      if(deliveryPinch > 0){
      deliveryPinch-=n;
      rpedidosPinch+=n;
      if(view != null){
                view.update(lDelivery.x,lDelivery.y);
            }
      }

            return true;
        } else {
            return false;
        }
    return true;
    }
```

Principalmente, se usa el string para dividir a quien va ese pedido porque aunque el owner pide las dos cosas a la vez para comprar y conseguir los objetos en el deliver se pide por separado.

Luego, una vez los tenga este va al fridge y los agrega con la operacion reponer. El código de reponer es el siguiente:

```
//Codigo de HouseEnv
// Acción de reponer las cervezas de la nevera
        else if(action.getFunctor().equals("reponer") && ag.equals("rpedidos")){
            try {
                result = model.addBeerPinchFridge(action.getTerm(0).toString(),(int)((NumberTerm)action.getTerm(1)).solve());
            } catch (NoValueException e) {
                logger.info("Failed to execute action deliver!"+e);
            }
        }
//Codigo HouseModel
boolean addBeerPinchFridge(String m,int n) {
    if(m.equals("beer")){
      if(rpedidosBeers > 0 ){
        rpedidosBeers-=n;
        availableBeers+=n;
        if (view != null) view.update(lFridge.x,lFridge.y);
      }
      return true;
    } else if(m.equals("pinchito")){
      if(rpedidosPinch > 0){
        rpedidosPinch-=n;
        availablePinch+=n;
            if (view != null) view.update(lFridge.x,lFridge.y);
            return true;
      }

    } else{
            return false;
        }
    return true;
    }
```

Burner

Este robot principalmente tiene un solo uso llegar al bin y en el momento que se elimine la basura se implementa una animación en el cual el bin se pone verde.

Esta operación del burner de eliminar la basura del bin se la da el rlimpiador o el owner una vez pongan basura en esta. El código tanto de un owner como de rlimpiador son muy parecidos, por lo tanto pondre el del rlimpiador:

```
La llamada de este plan se hace en el +hay_basura del rlimpiador
//Este es el plan del rlimpiador que le indica al burner que hay basura
+!hay_papeleraBin(rbasurero, bin) <-
  .send(rbasurero, tell, papelera_llena(rbasurero, bin));
  .wait(100);
  .send(rbasurero, untell, papelera_llena(rbasurero, bin)).
```

El código referente al burner es este:

```
+papelera_llena(rbasurero, bin) [source(Agt)]
    <-  !go_at(rbasurero, bin);
        vaciar(bin);
    burnerOn;
        !go_at(rbasurero, baseRBasurero);
    burnerOff.
```

Se dirige al bin y elimina la basura las operaciones de burnerOn y burnerOff es la para activar la animación. Estos cuando se activan a través del env, en el modelo hay un boolean que cambia a true o false y con el view.update hace que cambie el color del bin. El código seria este:

```
//Codigo del HouseModel
boolean cambiarCol_Off(){
    burnerOn = false;
    view.update(lBin.x, lBin.y);
    return true;
  }
//Codigo HouseView
case HouseModel.BIN:
                String b = "Bin";
                if (hmodel.cansInTrash > 0) {
                    b +=  " ("+hmodel.cansInTrash+"/10)";
                }
```

```
        if(hmodel.burnerOn == false){
                super.drawAgent(g, x, y, Color.RED, -1);
        } else if(hmodel.burnerOn == true) {
        super.drawAgent(g,x,y,Color.GREEN, -1);
        }
                g.setColor(Color.black);
                drawString(g, x, y, defaultFont, b);
                break;
```

Obstáculos

Para esta parte, primero tendríamos que inicializar con la creación en el view del case del obstáculo con su color, dejándolo de esta forma:

```
// Dibujado de los posibles obstaculos del entorno
            case HouseModel.OBSTACULE:
                super.drawAgent(g, x, y, Color.darkGray, -1);
                g.setColor(Color.black);
                drawString(g, x, y, defaultFont, "Obstacule");
                break;
```

Luego, se crearía un arraylist vacío de estos obstáculos y en el constructor del HouseModel se crean y almacenan estos obstáculos. El código es:

```
public HouseModel() {
        // Creación del grid con el tamaño definido en GSize
        // Número de agentes móviles: 5
        super(GSize, GSize, 5);

        // Añadido de posiciones iniciales para los agentes (móviles)
        setAgPos(agents.get("rmayordomo"), lRMayordomo);
        setAgPos(agents.get("rlimpiador"), lRLimpiador);
        setAgPos(agents.get("rbasurero"), lRBasurero);
        setAgPos(agents.get("rpedidos"), lRPedidos);
        setAgPos(agents.get("owner"), lOwner);

        // Añadido de posiciones para los elementos del entorno (no móviles)
        add(FRIDGE, lFridge);
        add(DELIVERY, lDelivery);
        add(BIN, lBin);
        add(COUCH, lCouch);
    add(LAVAVAJILLAS, lLavavajillas);
    add(LACENA, lLacena);
    int j = ~0;
    int i = 0;


    while(i < 8){
      int x = (int) (Math.random()*(GSize-1));
      int y = (int) (Math.random()*(GSize-1));
      if(isFree(j, x, y )){
        lObstacules.add( new Location(x, y));
        add(OBSTACULE, lObstacules.get(i));
        i++;
      }

    }

    }
```

Como hemos explicado antes en esta operación es usa el isFree en este caso ponemos una mascara de 0 porque no es necesario ningún requisito específico como que esté por encima de algo.

También indicar que su posición es aleatoria. También para comprobar que los agentes lo esquiven tanto esto como otros objetos por ejemplo la basura utilizamos el move_towards de tal forma que no pase por encima de los objetos utilizando una mascara de 0 y un isFree.

El move_towards seria el siguiente:

```
// Movimiento de los agentes por el entorno
    boolean moveTowards(String ag, Location dest) {

        int nAg = this.agents.get(ag);
        Location lAgent = getAgPos(nAg);
```

```
        String move = getNextMove(dest, lAgent);


    if(!move.isEmpty() && move.charAt(0) == 'u') {
      lAgent.y--;
      move = move.substring(1);
    } else if(  !move.isEmpty() && move.charAt(0) == 'l') {
      lAgent.x--;
      move = move.substring(1);
    } else if( !move.isEmpty()  && move.charAt(0) == 'b' ){
      lAgent.y++;
      move = move.substring(1);
    } else if ( !move.isEmpty() && move.charAt(0) == 'r' ) {
      lAgent.x++;
      move = move.substring(1);
    } else if(!move.isEmpty() && move.charAt(0) == 'n' ){
        move = getNextMove(dest, lAgent);
      move = move.substring(1);
    }

        setAgPos(nAg, lAgent);

    if((ag.equals("rlimpiador") && dest.equals(lRLimpiador)) || (ag.equals("owner") && dest.equals(lCouch))){

      move_to(nAg, lAgent, dest);
    }

        return true;
    }
```

Para hallar el movimiento y que de tal forma tampoco pase por encima de obstáculos u otros objetos usamos la operacion getNextMove() la cual calcula los movimientos posibles a hacer por parte del agente, almacenándolos en movimientos explorados y por explorar o Unchecked, este le pasa un string del movimiento y por el cual no pueda chocar con nada.

```
// Clase par para almacenar posiciones para la funcion del move_towards
  public class Pair<L,R> {
  private L l;
  private R r;
  public Pair(L l, R r){
    this.l = l;
    this.r = r;
  }

  public L getL() { return l; }
  public R getR() { return r; }
  public void setL(L l) { this.l = l; }
  public void setR(R r) { this.r = r; }
  }

  // Siguiente movimiento para el agente
  private String getNextMove(Location dest, Location or){
    ArrayList<Pair<Location, String>> uncheckedMoves = new ArrayList<Pair<Location, String>>();

    ArrayList<Integer> explored = new ArrayList<Integer>();

    int i = ~0;

    uncheckedMoves.add(new Pair<Location, String>(or, ""));
    explored.add(or.x + or.y * GSize);

    do{
      Location l = uncheckedMoves.get(0).getL();
      String moves = uncheckedMoves.get(0).getR();
      uncheckedMoves.remove(0);


      if(l.isNeigbour(dest)) return moves;

      // TOP
      if   (isFree(i, l.x, l.y - 1) && !explored.contains(l.x + (l.y - 1) * GSize)) {
        uncheckedMoves.add(new Pair<Location, String>(new Location(l.x, l.y - 1), moves + 'u'));
        explored.add(l.x + (l.y - 1) * GSize);

      }

      // BOTTOM
      if(isFree(i, l.x, l.y + 1) && !explored.contains(l.x + (l.y + 1) * GSize)) {
        uncheckedMoves.add(new Pair<Location, String>(new Location(l.x, l.y + 1), moves + 'b'));
        explored.add(l.x + (l.y + 1) * GSize);

      }
```

```
      // LEFT
      if   (isFree(i, l.x - 1, l.y) && !explored.contains(l.x - 1 + l.y * GSize)) {
        uncheckedMoves.add(new Pair<Location, String>(new Location(l.x - 1, l.y), moves + 'l'));
        explored.add(l.x - 1 + l.y * GSize);

      }

      // RIGHT
      if(isFree(i, l.x + 1, l.y) && !explored.contains(l.x + 1 + l.y * GSize)) {
        uncheckedMoves.add(new Pair<Location, String>(new Location(l.x + 1, l.y), moves + 'r'));
        explored.add(l.x + 1 + l.y * GSize);
      }


    } while(!uncheckedMoves.isEmpty());

    return "n";

  }
```

Nevera

Gestión de diferentes tipos de pinchos

La gestión y comprobación de los pinchos en la nevera la hacemos mediante el método get(beer, pinchito) y comprobando el belief de evailable(beer, fridge) ya que al gestionar pedidos y cervezas a la vez, si hay pinchitos también hay cerveza:

```
boolean getBeerAndPinchito(String ag) {
        if (availableBeers > 0) {
            if(fridgeOpenMayordomo && ag.equals("rmayordomo") && !carryingBeerMayordomo){
                carryingBeerMayordomo = true;
            } else if(fridgeOpenOwner && ag.equals("owner") && !carryingBeerOwner){
                carryingBeerOwner = true;
            }

      availablePinch--;
            availableBeers--;

            if (view != null) view.update(lFridge.x,lFridge.y);
            return true;
        } else {
            return false;
        }
    }
```

Lo que hace es comprobar si hay cervezas disponibles y, si la hay, el mayordomo le lleva al owner una cerveza ‘carryingBeerMayordomo = true’, se disminuye a la vez los pinchos y las cervezas disponibles y actualizamos las vistas.

Evitar que la nevera se quede vacía

Evitamos que la nevera se quede vacía gracias al belief available(beer, fridge) que comprueba que haya pinchos o cerveza siempre de la nevera antes de llevarle los productos al owner y si no hay disponibles, realizamos la compra a la vez tanto de pinchitos como de cervezas. Esto lo realizamos en el siguiente plan de bring:

```
+!bring(owner,beer, pinchito) [source(self)]:  not available(beer,fridge) & not ordered(beer) & cerverza_escogida(M) & pinchito_escogi
   <- .println("El robot mayordomo realiza un pedido de ", M);
      ?cerveza_escogida(M);
    ?pinchito_escogido(P);
      !comprar(supermarket, beer, M);
    .wait(100);
    println("El robot mayordomo realiza un pedido de ", P);
    !comprar(supermarket, pinchito, P);
      !bring(owner,beer, pinchito).
```

Además, antes de que esto suceda, y quedarnos sin cervezas o pinchos cuando el owner nos pida, realizamos la compra a medida que le vamos dando productos al owner. Esto lo podemos ver en el siguiente plan de bring del mayordomo:

```
+!bring(owner,beer, pinchito)[source(self)]:  available(beer,fridge) & not too_much(beer) & asked(beer) & cerveza_escogida(M) & pinchi
   <- .println("El robot mayordomo va a buscar una cerveza");
      !go_at(rmayordomo,fridge);
```

```
      open(fridge);
      get(beer, pinchito);
    ?cerveza_escogida(M);
    ?pinchito_escogido(P);
    !comprar(supermarket, beer, M);
    .wait(100);
    !comprar(supermarket, pinchito, P);
      close(fridge);
      !go_at(rmayordomo,couch);
    .wait(1000);
      !hasBeer(owner);
      .abolish(asked(beer));
      !bring(owner,beer, pinchito).
```

Como podemos ver, a parte de comprobar que hay cervezas disponibles, realizamos la compra tanto de pinchitos como de cerveza para evitar que para el siguiente producto que nos pida el owner nos quedemos sin el.

Lógica de movimiento

Para la lógica del movimiento al no poder aplicar por JASON, la aplicamos en Java de la siguiente forma. Primero, se formula el move_towads en un agente. Luego, pasa por el env de la siguiente forma:

```
//Codigo en Env
// Acciones de movimiento
        if (action.getFunctor().equals("move_towards")){
            String l = action.getTerm(0).toString();
            Location dest = null;

            // Acciones de movimiento para el robot mayordomo
            if(ag.equals("rmayordomo")){
                if (l.equals("fridge")) {
                    dest = model.lFridge;
                } else if (l.equals("couch")) {
                    dest = model.lCouch;
                } else if(l.equals("baseRMayordomo")){
                    dest = model.lRMayordomo;
                } else if (l.equals("lavavajillas")) {
                    dest = model.lLavavajillas;
                } else if(l.equals("lacena")){
                    dest = model.lLacena;
                } else if(l.equals("owner")){
          dest = model.lOwner;
        }
            }

       // Acciones de movimiento para el robot limpiador
            else if(ag.equals("rlimpiador")){
                if(l.equals("bin")){
                    dest = model.lBin;
                } else if(l.equals("trash")){
                    dest = model.lTrash.get(0);
                } else if(l.equals("baseRLimpiador")){
                    dest = model.lRLimpiador;
                }else if (l.equals("couch")) {
                    dest = model.lCouch;
                }else if (l.equals("lavavajillas")) {
                    dest = model.lLavavajillas;
                }
            }


            // Acciones de movimiento para el robot basurero
            else if(ag.equals("rbasurero")){
                if(l.equals("bin")){
                    dest = model.lBin;
                } else if(l.equals("baseRBasurero")){
                    dest = model.lRBasurero;
                }
            }



            // Acciones de movimiento para el robot de pedidos
            else if(ag.equals("rpedidos")){
                if (l.equals("delivery")){
                    dest = model.lDelivery;
                } else if (l.equals("fridge")) {
                    dest = model.lFridge;
                } else if(l.equals("baseRPedidos")){
                    dest = model.lRPedidos;
                }
```

```
            }

            // Acciones de movimiento para el owner
            else if(ag.equals("owner")){
                if (l.equals("fridge")) {
                    dest = model.lFridge;
                } else if (l.equals("couch")) {
                    dest = model.lCouch;
                } else if(l.equals("bin")){
                    dest = model.lBin;
                }else if(l.equals("trash")){
                    dest = model.lTrash.get(0);
                }
            }

            try {
                result = model.moveTowards(ag, dest);
            } catch (Exception e) {
                e.printStackTrace();
            }

        }
```

Según el agente se le da como destino los distintos que puede tener tiendo los casos especiales del trash en el cual se pasa el primero de su arraylist.

Luego, con el conocimiento del agente que es un string se comprueba cual es su valor en el map se observa su localización. Sabiendo la localización del agente a mover y su destino se le pasa a la operación getNextMove(). Código del move_towards:

```
// Movimiento de los agentes por el entorno
    boolean moveTowards(String ag, Location dest) {

        int nAg = this.agents.get(ag);
        Location lAgent = getAgPos(nAg);

        String move = getNextMove(dest, lAgent);


    if(!move.isEmpty() && move.charAt(0) == 'u') {
      lAgent.y--;
      move = move.substring(1);
    } else if(  !move.isEmpty() && move.charAt(0) == 'l') {
      lAgent.x--;
      move = move.substring(1);
    } else if( !move.isEmpty()  && move.charAt(0) == 'b' ){
      lAgent.y++;
      move = move.substring(1);
    } else if ( !move.isEmpty() && move.charAt(0) == 'r' ) {
      lAgent.x++;
      move = move.substring(1);
    } else if(!move.isEmpty() && move.charAt(0) == 'n' ){
       move = getNextMove(dest, lAgent);
      move = move.substring(1);
    }

        setAgPos(nAg, lAgent);

    if((ag.equals("rlimpiador") && dest.equals(lRLimpiador)) || (ag.equals("owner") && dest.equals(lCouch))){

      move_to(nAg, lAgent, dest);
    }

        return true;
    }
```

En el getNextMove(), se tiene dos arraylist la cual, una de ellas tiene un pair que almacena los valores de localización y un string que son los movimientos totales que haría ese agente hacia su destino.

El código del getNextMove() es el siguiente:

```
// Clase par para almacenar posiciones para la funcion del move_towards
  public class Pair<L,R> {
  private L l;
  private R r;
  public Pair(L l, R r){
    this.l = l;
    this.r = r;
  }

  public L getL() { return l; }
```

```
  public R getR() { return r; }
  public void setL(L l) { this.l = l; }
  public void setR(R r) { this.r = r; }
  }

  // Siguiente movimiento para el agente
  private String getNextMove(Location dest, Location or){
    ArrayList<Pair<Location, String>> uncheckedMoves = new ArrayList<Pair<Location, String>>();

    ArrayList<Integer> explored = new ArrayList<Integer>();

    int i = ~0;

    uncheckedMoves.add(new Pair<Location, String>(or, ""));
    explored.add(or.x + or.y * GSize);

    do{
      Location l = uncheckedMoves.get(0).getL();
      String moves = uncheckedMoves.get(0).getR();
      uncheckedMoves.remove(0);


      if(l.isNeigbour(dest)) return moves;

      // TOP
      if   (isFree(i, l.x, l.y - 1) && !explored.contains(l.x + (l.y - 1) * GSize)) {
        uncheckedMoves.add(new Pair<Location, String>(new Location(l.x, l.y - 1), moves + 'u'));
        explored.add(l.x + (l.y - 1) * GSize);

      }

      // BOTTOM
      if(isFree(i, l.x, l.y + 1) && !explored.contains(l.x + (l.y + 1) * GSize)) {
        uncheckedMoves.add(new Pair<Location, String>(new Location(l.x, l.y + 1), moves + 'b'));
        explored.add(l.x + (l.y + 1) * GSize);

      }


      // LEFT
      if   (isFree(i, l.x - 1, l.y) && !explored.contains(l.x - 1 + l.y * GSize)) {
        uncheckedMoves.add(new Pair<Location, String>(new Location(l.x - 1, l.y), moves + 'l'));
        explored.add(l.x - 1 + l.y * GSize);

      }

      // RIGHT
      if(isFree(i, l.x + 1, l.y) && !explored.contains(l.x + 1 + l.y * GSize)) {
        uncheckedMoves.add(new Pair<Location, String>(new Location(l.x + 1, l.y), moves + 'r'));
        explored.add(l.x + 1 + l.y * GSize);
      }


    } while(!uncheckedMoves.isEmpty());

    return "n";

  }
```

Como se puede observar el código que nos va dando es una string la cual consta de movimientos no diagonales y no solo eso si no que su movimiento es up/down/left/right según es necesario para llegar a su destino. Para añadir más información tanto en el Env para añadir percepts se utilizo el isNeigbour para tener los agentes en posiciones adyacentes como se pedía en la memoria aunque hay dos casos especiales que se puede observar en la rutina move_to del move_towards.

Interfaz

Por ultimo, tenemos la explicación del HouseView en este caso empezamos hablando del grid que en este caso es 11x11 para agregar este conocimiento y sea correcto se indica aquí:

```
// Tamaño del grid
    public static final int GSize = 11;
```

Para el formato de letra y su tamaño se hace propiamente en el view justamente así:

```
HouseModel hmodel;
```

```
    public HouseView(HouseModel model) {
        super(model, "Domestic Robot", 700);
        hmodel = model;
        defaultFont = new Font("Arial", Font.BOLD, 11); // change default font
        setVisible(true);
        repaint();
    }
```

Por temas de asincronismo con los agentes y el move_towards tuvimos que agregar un repaint temporal, al final del draw que se ejecute cada 50 veces que se dibuja el view al ser este con un Override.

```
//Al final del draw
if(cont > 50){
        repaint();
    cont = 0;
    }
```

Para finalizar, tenemos la parte de como se muestra las cantidades en el frigorífico, lavavajillas, alacena y basura. Como el máximo nunca se ha pedido en ninguna entrega lo que hicimos es agregarlo en el propio string del nombre de ese objeto. El código del view para todo esto sería el siguiente manera:

```
// Dibujado de los elementos que no son agentes
        switch (object) {

            // Dibujado del frigorifico
            case HouseModel.FRIDGE:
                super.drawAgent(g, x, y, Color.WHITE, -1);
                g.setColor(Color.black);
                drawString(g, x, y, defaultFont, "F=(B:"+ hmodel.availableBeers + " ,P:" + hmodel.availablePinch +")");
                break;

            // Dibujado del sillón del owner
            case HouseModel.COUCH:
                super.drawAgent(g, x, y, Color.PINK, -1);
                g.setColor(Color.black);
                drawString(g, x, y, defaultFont, "Couch");
                break;

            // Dibujado de la zona de entrega
            case HouseModel.DELIVERY:
                super.drawAgent(g, x, y, Color.GRAY, -1);
                g.setColor(Color.black);
                drawString(g, x, y, defaultFont, "Delivery");
                break;

            // Dibujado del cubo de basura
            case HouseModel.BIN:
                String b = "Bin";
                if (hmodel.cansInTrash > 0) {
                    b +=  " ("+hmodel.cansInTrash+"/10)";
                }
        if(hmodel.burnerOn == false){
                super.drawAgent(g, x, y, Color.RED, -1);
        } else if(hmodel.burnerOn == true) {
        super.drawAgent(g,x,y,Color.GREEN, -1);
        }
                g.setColor(Color.black);
                drawString(g, x, y, defaultFont, b);
                break;

            // Dibujado de la basura repartida por el entorno
            case HouseModel.TRASH:
                super.drawAgent(g, x, y, Color.LIGHT_GRAY, -1);
                g.setColor(Color.black);
                drawString(g, x, y, defaultFont, "Trash");
                break;
      // Dibujo para el lavavajillas
      case HouseModel.LAVAVAJILLAS:
        String a = "Lav";
        Color c = Color.yellow;
                if (hmodel.platosLimpiar > 0) {
                    a +=  " ("+hmodel.platosLimpiar+"/10)";
                }
                super.drawAgent(g, x, y, c, -1);
                g.setColor(Color.black);
                drawString(g, x, y, defaultFont, a);
                break;

      // Dibujo para la lacena
            case HouseModel.LACENA:
```

```
        String d = "Lac";
        Color r = Color.orange;
                if (hmodel.platosLimpios == 0) {
                    d +=  " ("+hmodel.platosEnLacena+"/10)";
                }
                super.drawAgent(g, x, y, r, -1);
                g.setColor(Color.black);
                drawString(g, x, y, defaultFont, d);
                break;

            // Dibujado de los posibles obstaculos del entorno
            case HouseModel.OBSTACULE:
                super.drawAgent(g, x, y, Color.darkGray, -1);
                g.setColor(Color.black);
                drawString(g, x, y, defaultFont, "Obstacule");
                break;

        }
```