4. AMPLIANDO LA PROGRAMACION DEL ROBOT

Este capítulo explica el mecanismo de especificar una nueva clase de robots y adicionar nuevas instrucciones al vocabulario del robot. También, se discutirán algunos métodos de programación para poner en práctica y probar los programas.

4.1. CREANDO UN LENGUAJE DE PROGRAMACIÓN MÁS NATURAL

Se dice que un robot realiza una tarea compleja, también que recibe muchos mensajes para ejecutar tareas.

El lenguaje de programación de robots permite al programador de robots especificar nuevas clases de robots. Esta descripción de clases suministra especificación de nuevas instrucciones de robot. Karel-Werke intentará entonces usar la descripción de clases para crear robots capaces de interpretar los nuevos mensajes.

La disponibilidad para que aprenda un robot es bastante limitada. Karel-Werke construyó cada robot con un *diccionario* de nombres de métodos útiles y sus definiciones, pero cada definición tiene que ser construida con instrucciones más simples que los robots ya entienden.

Para suministrar a los robots con un diccionario de instrucciones que ejecute acciones complejas, se puede construir un vocabulario robot que corresponda a su propietario. Concedido este mecanismo se pueden solucionar nuestros problemas de programación usando instrucciones que sean naturales para nuestra forma de pensar y entonces podemos proveer robot con la definición de estas instrucciones.

Se puede definir una instrucción **moveMile()** con 8 mensajes de **move()**, entonces, cuando un robot es avisado para una instrucción moveMile() en un programa, éste busca la definición asociada con este nombre de mensaje y lo ejecuta. Ahora nuestro inmanejable programa de movimiento de pitos puede ser escrito con una definición **moveMile()**, conteniendo 8 mensajes de **move()**, y otros 15 mensajes de **moveMile().** Este programa contiene estos 23 mensajes, que serán más comprensibles con el programa original, que necesita más de 120 mensajes para culminar la tarea.

En problemas complicados la disposición de ampliar el vocabulario del robot permite la comprensión de los programas

4.2. UN MECANISMO QUE DEFINE NUEVAS CLASES DE ROBOTS

La declaración de la clase primitiva **ur_robot().** Los usuarios del lenguaje robot pueden también declarar nuevas clases de robot y la instalación capaz de enviarlos, exactamente en el estándar de los robots. Para especificar una nueva clase de robots, se incluye una especificación de clase en la sección de declaración al comienzo de cada uno de los programas. Aislada de un programa, la forma general de especificación se muestra a continuación:

class <Nuevo nombre de clase> extends <nombre clase vieja>
{
<lista -de-nuevos-métodos>
}

La clase de especificación usa la palabra reservada **class** and **extends(:)**, y los símbolos especiales entre paréntesis, separan varias partes de la declaración. Esta forma general incluye elementos delimitados por paréntesis angulares, < >, que pueden ser remplazados con una substitución apropiada cuando se incluye una especificación en un programa robot.

Los paréntesis angulares no son parte de los lenguajes de programación de robots, son una estructura de programa donde los elementos del lenguaje pueden aparecer. En este caso, <nueva-clase-nombre> tiene que ser remplazada por un nuevo nombre, todavía no usada en el programa. Este nombre puede ser construido con letras minúsculas y mayúsculas, dígitos y caracteres subrayados pero no enfrentado con cualquier otro nombre en el programa, ni escribir correctamente cualquier palabra reservada.

Los nombres también deben comenzar con una letra. La reposición para r <anterior-clase-nombre> es el nombre de la clase de robot existente, ya sea ur_Robot() o una declarada previamente en el siguiente programa.

Se puede hacer así con la especificación de una nueva clase a continuación:

```
class Caminante_Millas : ur_Robot
{
void moveMile();
};
void Caminante_Millas  :: moveMile()
        {       ...     // instrucciones excluidas por ahora.
        }
}
```

El nombre de la nueva clase de robot es **Caminante_Millas()**. Se indica también, por el nombre de la clase **ur_Robot()** seguida de (**:)** que significa **ampliado** que la clase **Caminante_Millas()** seguido de ampliado (:), ese **Caminante_Millas()** tiene todas las capacidades de los miembros de la clase **ur_Robot()**.

La especificación **ur_Robot()** es la clase padre de Caminante_Millas que **Caminante_Millas()** es una subclase de **ur_Robot()**. También se dice que los robots de la nueva clase heredan todas las habilidades de la clase padre. Por consiguiente, Caminante_Millas sabe como moverse y girar a la izquierda, al

igual que los miembros de la clase **ur_Robot()**. Ellos pueden también recoger y poner pitos y dar vueltas apropiadamente.

Cada método en la lista es escrito con su definición en paréntesis. Esta especificación dice que cuando un robot en la clase es inicialmente cambiada será capaz de ejecutar instrucciones de Caminante_Millas correctamente como todos los métodos heredados de la clase **ur_Robot()**.

La declaración de clase introduce los nombres de nuevos métodos de robots.

4.3. DEFINICIÓN DE NUEVOS MÉTODOS

Tal como se declara una nueva clase de robot, es necesario definir todas las instrucciones introducidas en éste. Estas definiciones son parte de la declaración de clase en la parte de declaración del programa robot. La forma de una definición de instrucción se ilustra a continuación:

```
void <instruccion_nombre> ()
{
        <lista_de_instrucciones>
}
```

Siempre se debe comenzar con la palabra reservada **void**. Se tiene que dar el nombre de la instrucción que se define entre los paréntesis delimitadores, dando una lista de instrucciones similar al bloque de tarea principal que llama a un robot de esta clase, para continuar con la nueva instrucción. Esta lista de instrucciones delimitada por paréntesis recibe el nombre de bloque en el vocabulario de la programación del robot.

Toda instrucción en la lista debe terminar con un punto y coma. Muchas de las instrucciones en la lista de instrucciones son mensajes, la instrucción **moveMile()** en la clase **CaminadorMile()** puede ser escrita así:

```
class CaminadorMile: ur_Robot
{
 void moveMile();
```

```
};
void CaminadorMile ::moveMile()
        {
                move();
                move();
                move();
                move();
                move();
                move();
                move();
        move();
        }
```

El lenguaje tiene que ser diseñado de tal forma que el robot se refiera a él mismo, como una palabra reservada, en este caso la instrucción move, en el caso citado puede ser remplazada por **this.move()**, pero no se requiere. Esto hace que el lenguaje sea más conciso. "this" significa "this robot."

Si se tiene un CaminadorMile() llamado **Lisa**, puede llegar a caminar una milla así: Lisa.moveMile();
o así:

```
Lisa.move();
Lisa.move();
Lisa.move();
Lisa.move();
Lisa.move();
Lisa.move();
Lisa.move();
Lisa.move();
```

En este caso, **Lisa** se mueve 8 veces recibiendo un mensaje de **moveMile()**. El programa completo del robot para el anterior caso es:

```
class CaminadorMile :ur_Robot
{
  void movemile();
}
void  CaminadorMile::moveMile()
```

```
        {
                move();
                move();
                move();
                move();
                move();
                move();
                move();
                move();
        }

task
{       CaminadorMile Lisa  (3, 2, East, 0); // Declaración de un nuevo CaminadorMile Lisa.

        Lisa.moveMile();
        Lisa.moveMile();
        Lisa.moveMile();
        Lisa.moveMile();
        Lisa.moveMile();
        Lisa.moveMile();
        Lisa.moveMile();
        Lisa.moveMile();
        Lisa.moveMile();
        Lisa.moveMile();
        Lisa.pickBeeper();
        Lisa.turnLeft();
        Lisa.moveMile();
        Lisa.moveMile();
        Lisa.moveMile();
        Lisa.moveMile();
        Lisa.moveMile();
        Lisa.turnOff();
}
```

4.4. EL ENTENDER Y EXACTITUD DEL NUEVO MÉTODO

Un robot es una máquina, un dispositivo completamente desprovisto de inteligencia. El robot hace exactamente lo que le “decimos”, no hay lugar para interpretación. Una declaración de clase de robot es una descripción de la

factoría de robot que dice como construir robot de esta clase. En la fábrica de robot una declaración entera parte de cualquier programa de robot que es leído y examinado para errores. Como parte de la fabricación y proceso de entrega de los robots está dada la definición de cada una de las nuevas instrucciones y sus clases.

Cada robot almacena la definición de las instrucciones en su propio diccionario de instrucciones. Así, cuando llamamos al robot de la clase **CaminadorMile()** para hacer moveMile(), este recibe el mensaje, consulta el diccionario para ver como tiene que responder y entonces ejecuta la requerida acción. El piloto del helicóptero no tiene que leer esta parte del programa cuando asigna el robot para repartos.

Según las reglas de gramática de la programación de robots, esta es una definición perfectamente legal por esto contiene errores lexicográficos o sintácticos. Sin embargo, la instrucción moveMile le dice al robot que ejecute la instrucción moveMile(), equivalente a ejecutar 6 instrucciones de move( );.

Cuando simulamos la ejecución de un robot de una instrucción definida, tenemos que adherirnos a las reglas que el robot usa para ejecutar estas instrucciones. El Robot ejecuta una determinada instrucción definida por desempeño de acciones asociadas con esta definición. El robot no conoce cual instrucción definida denota, el robot solo conoce como está definida. Se puede reconocer el significado de esta distinción y aprender a interpretar el programa robot como literalmente lo hace el robot. El significado de nombres es supuesto para ayudar al lector humano a entender el programa.

Si la instrucción actual define el significado del nombre este desacuerdo con el significado del nombre, es fácil estar desorientado.

4.5. DEFINIENDO NUEVOS MÉTODOS EN UN PROGRAMA

En esta sección se expone un programa completo de robot que usa el método de definición de mecanismo. Primero rastreamos la ejecución del programa.

Se discutirá la forma general del programa que usa el nuevo método de definición de mecanismo. La tarea se ilustra en la figura 58 “Limpiar la escalera”, este recoge cada pito en el micromundo mientras asciende por la escalera.

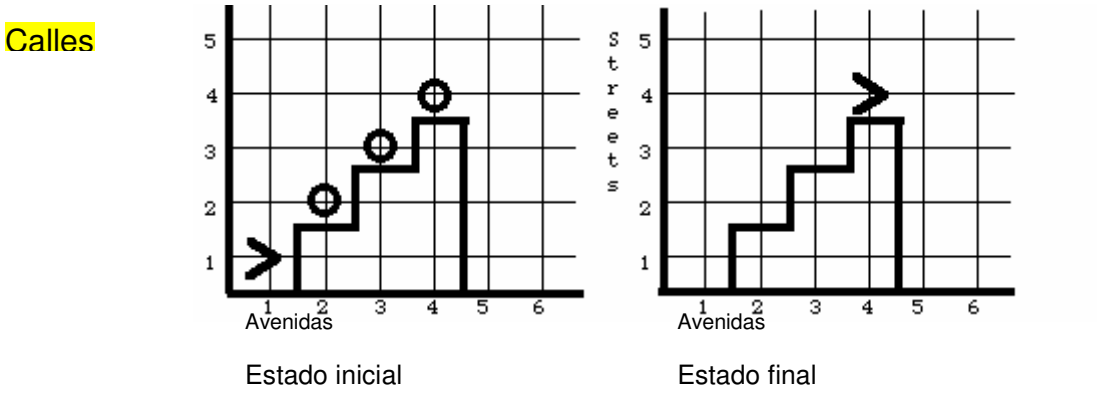


Figura. 102 Programa “Limpiar la escalera”

```
class Limpia escalera : ur_Robot
{
void turnRight();
void climbstair();
};
        void Limpiaescalera ::turnRight()
        {
                turnLeft();
                turnLeft();
                turnLeft();
        }
                void Limpiaescalera ::climbstair()
        {
                turnLeft();
                move();
                turnRight();
                move();
        }

task
{       Limpiaescalera  Alex(1, 1, East, 0);
```

```
        Alex.climbstair();
        Alex.pickBeeper();
        Alex.climbStair();
        Alex.pickBeeper();
        Alex.climbStair();
        Alex.pickBeeper();
        Alex.turnOff();
}
```

A continuación, se suministra una versión comentada del mismo programa con números en cada instrucción y mensajes en el orden en que se están ejecutando, comenzando con la especificación de la instrucción de reparto así #n.

```
class Limpia escalera : ur_Robot
{
}
        void  Limpiaescalera::turnRight()
        {// to here from      #4,        #13 or     #22
                turnLeft();  #5 or      #14 or     #23
                turnLeft();  #6 or      #15 or     #24
                turnLeft();  #7 or      #16 or     #25
        }//   return to      #4         #13        #22

        void  Limpiaescalera::climbStair()
        {// to here from      #1,        #10, or #19
                turnLeft();  #2         #11        #20
                move();                 #3         #12        #21
                turnRight();            #4         #13        #22
                move();                 #8         #17        #26
        }//    return to     #1         #10        #19

task
{        Limpiaescalera  Alex(1, 1, East, 0);          #0

        Alex.climbStair();                                          #1
        Alex.pickBeeper();                                          #9
        Alex.climbStair();                                          #10
```

```
        Alex.pickBeeper();                                    #18
        Alex.climbStair();                                    #19
        Alex.pickBeeper();                                    #27
        Alex. turnOff();                                      #28
}
```

Cuando un programa se está ejecutando, llamamos a la instrucción actual el "foco de la ejecución". Cuando el piloto del helicóptero comienza a ejecutar un programa, el foco está inicialmente en la primera instrucción dentro del bloque de la tarea principal. En este programa de muestra el foco inicial es el mensaje del climbstair, que se anota como # 1. El mensaje del climbStair() se envía a través del satélite al robot Alex. Cuando Alex recibe este mensaje, el foco de la ejecución pasa del piloto a Alex. El piloto, que debe esperar hasta que se haya terminado la instrucción, tiene que ser muy cuidadoso de recordar en que parte del programa estaba cuando recibió el mensaje del climbStair().

En el programa de muestra el nuevo punto de la ejecución se anota como "de aquí # 1". Alex se centra en la lista de las instrucciones que definen la nueva instrucción, climbStair(), y encuentra un mensaje del turnleft (marcado como # 2). Alex entrena al foco de la ejecución en el mensaje del turnLeft() lo ejecuta, y después se centra en # 3, movimiento. Alex ejecuta este movimiento y focos en # 4, turnRight(). Puesto que esto no es una instrucción primitiva, Alex debe recuperar su definición de su diccionario. Después se centra en la lista de la instrucción de la definición del turnRight() y ejecuta las tres instrucciones del turnLeft(), # 5, # 6, y # 7. Terminar la ejecución de la instrucción del turnRight(), Alex ahora vuelve su foco al lugar en el cual el mensaje del turnright ocurrió dentro de climbStair(). Alex cambia de puesto el foco a #8 y ejecuta el movimiento.

Después de que Alex realice este movimiento, se acaba de ejecutar la instrucción climbStair() como las ejecuciones de las producciones de nuevo al piloto puesto que ha realizado totalmente la tarea requerida por el mensaje climbstair. El foco de la ejecución vuelve al lugar en el programa marcado # 1, y el piloto envía Alex el mensaje del pickBeeper() que está marcado # 9.

Con este mensaje, el foco de la ejecución se pasa otra vez de piloto al robot. Alex también interpreta y realiza esta instrucción de pickBeeper() y rinde el foco de nuevo al piloto en # 10. El piloto entonces envía a Alex otro mensaje climbStair().

Alex repite esta misma secuencia de pasos una segunda vez para subir-escalera marcado instrucción #10 y el pickBeeper() que sigue y otra vez para los terceros mensajes de subir-escalera y del pickBeeper(). Alex finalmente ejecuta la instrucción de salida, entonces la ejecución de programa es completa.

El piloto y el robot deben recordar siempre la posición exacta en el programa donde estaban cuando el foco cambia. Esto permite que la ejecución vuelva a su lugar correcto y continúe ejecutando el programa. Es importante que entendamos que no hay reglas complejas para ejecutar un programa que contiene nuevas instrucciones.

Se debe entender como el piloto del helicóptero y el robot trabajan juntos para ejecutar un programa que incluya el mecanismo de la definición de la instrucción.

Anteriormente, se menciono que las declaraciones están escritas siempre antes del bloque de la tarea principal. En nuestro ejemplo de programación, vimos que la declaración de la nueva clase y la definición de las nuevas instrucciones para la clase están consignadas aquí. Debemos escribir siempre nuestras nuevas definiciones de la instrucción en esta área.

Los nombres definidos dentro de una clase del robot, incluyendo los nombres de la clase del padre y del padre del padre, así sucesivamente (llamados los antepasados), se llama el diccionario de la clase. La declaración del ur_Robot() no necesita ser incluido en los programas del robot, puesto que es un "estándar de la fábrica."

Cualquier otra clase que usted necesite definir se debe escribir totalmente en la pieza de la declaración, junto con las definiciones de todas sus instrucciones. Pronto veremos un método de taquigrafía para hacer la escritura de este diccionario mucho más fácil. Cada instrucción en un bloque es terminada por un punto y coma.

Las palabras de la clase no son permanentes y el mundo no recuerda ninguna definición del programa al programa. Cada vez que escribimos un programa del robot, debemos incluir un sistema completo de todas las palabras requeridas en ese programa.

4.6. LOS MÉTODOS HEREDADOS DE MODIFICACIÓN

La construcción de la clase CaminadorMile() le dio al robot la capacidad de caminar una milla a la vez, se conservó su capacidad de caminar un bloque a la vez. Construir una clase en la cual se redefine una instrucción previamente definida debe tener un nuevo significado. Por ejemplo, suponga que teníamos un problema en el cual un robot necesitó siempre moverse por millas, pero nunca al lado de los bloques.

En este caso sería una ventaja crear una clase con una nueva definición de instrucciones de movimiento, de modo que, cuando un robot de esta clase reciba una orden para moverse, éste se mueva una milla.

```
class Mile_Mover extends ur_Robot
{       void move()
        {       super.move();
                super.move();
                super.move();
                super.move();
                super.move();
                super.move();
                super.move();
                super.move();
        }
```

En la nueva definición de movimiento **move()** esta clase elimina la definición original heredada de la clase ur_Robot(). Ahora tenemos un problema, puesto que para mover una milla, necesitamos poder mover ocho bloques, pero estamos definiendo un movimiento que equivale a una milla. Por lo tanto, no podemos decir movimiento ocho veces, necesitamos indicar que deseemos utilizar la original, o eliminar, la instrucción **move()**, desde la clase ur_Robot().

Podemos hacer esto puesto que ur_Robot() es la clase del padre. Apenas necesitamos introducir el mensaje del movimiento **move()** con la palabra clave estupenda y un período. Nos da una manera de especificar un método particular de la clase del padre. Ahora terminamos el programa anterior con:

```
task
{       Mile_Mover Karel(5, 2, North, 0);
        Karel.move();
        Karel.pickBeeper();
        Karel.move();
        Karel.putBeeper();
        Karel.turnOff();
}
```

Karel encontrará el pito en (13, 2) y lo dejarán en (21, 2).

Aviso ahora que si teníamos robots en el mismo programa y enviar a cada uno de ellos los mismos mensajes, puede suceder que cada uno de ellos responda en forma diferente a estos mensajes.

En detalle, un CaminadorMile() mueve solamente un bloque cuando está **move()**, mientras que un Mover_Mile(), mueve una milla.

4.7. UN PROGRAMA GRAMATICALMENTE ERRÓNEO

En el ejemplo que se ilustra a continuación se observa un error de programación común, omitir apoyos necesarios alrededor de un bloque. La definición del longMove() define la instrucción correctamente, pero se ha

omitido el apoyo de la abertura del par que debe incluir las tres instrucciones del movimiento. ¿Usted debe encontrar el error? ¿Está usted confundido por el otro error en este ejemplo?

Encontrar el error de la sintaxis no es fácil, porque las identaciones ayudan a que sea correcto para nosotros.

```
class Big_Stepper : ur_Robot
{
void longMove();
};
        void Big_Stepper:: longMove()
                move();
                move();
                move();
        }

task
{       Big_Stepper Tony(5, 2, North, 0);
        Tony.longMove();
        Tony.turnLeft();
        Tony.turnOff();
}
```

La instalación lee la parte de la declaración de un programa y del bloque de la tarea principal del programa para comprobar si hay errores léxicos y de la sintaxis.

Un lector descubre errores de sintaxis comprobando los componentes "significativos" del programa y comprobando si la gramática y la puntuación son apropiadas. Los ejemplos de componentes significativos son declaraciones de la clase, definiciones del método, y el bloque de la tarea principal. En efecto verificamos los componentes significativos por separado.

Ilustremos cómo la instalación encuentra el error en el programa haciendo uso de esta técnica. La instalación lee sólo las palabras del programa y no es

influenciada por nuestra identación. La fábrica examina la nueva declaración de la clase del robot. Tiene un nombre, una clase del padre, y una lista correcta de características.

Compruebe la puntuación, verifique el nombre de la clase y el nombre del método en la definición de la instrucción. Determine un bloque para incluir la definición. No encuentra el apoyo de la abertura, en su lugar encuentra un nombre. La instalación nos dice que existe un error de sintaxis.

En resumen, el olvidarse de utilizar apoyos necesarios alrededor de un bloque puede conducir a errores de sintaxis. Debemos ser expertos que rápidamente nos convierta en analista de programas. Dado un programa de robot se debe verificar la gramática y errores de puntuación.

4.8. HERRAMIENTAS PARA DISEÑAR Y ESCRIBIR LOS PROGRAMAS DE KAREL

Diseñar las soluciones para los problemas y escribir programas del robot implica como solucionar un problema.

Un modelo describe como solucionar un problema. Un proceso consta de cuatro actividades:

Definición del problema
Planeación del problema
Solución del problema
Ejecución y análisis de la solución del problema

La definición inicial del problema se presenta por medio de figuras proporcionadas de las situaciones iniciales y finales. Una vez que examinemos estas situaciones y entendamos qué tarea debe realizar un robot, comenzamos a planear, a poner, y a analizar una solución en ejecución.

Esta sección examina las herramientas para diseñar y escribir programas poniendo y analizando programas del robot ejecución. Combinando estas

técnicas con el nuevo mecanismo de la clase y de la instrucción, podemos desarrollar las soluciones que son fáciles de leer y de entender.

Para desarrollar y escribir los programas que solucionan los problemas del robot, debe seguirse estas tres pautas:

- Los programas deben ser fáciles de leer y de entender,
- Los programas deben ser fáciles de eliminar errores.
- Los programas deben ser fáciles de modificarse para tomar variadas formas de solución respecto de la tarea original.

4.9. TÉCNICA DE REFINAMIENTO PASO A PASO PARA EL USO DE LAS HERRAMIENTAS DE DISEÑO Y ESCRITURA DE PROGRAMAS DE KAREL

En esta sección, se describe el refinamiento paso a paso de un método que podemos utilizar para diseñar y escribir los programas del robot. Este método permite resolver el problema de una forma más simple que sean fáciles de leer y de entender, para llegar a una solución más entendible y correcta.

Es natural definir todas las nuevas clases y métodos que necesitamos para una tarea y después escribir el programa usando las instrucciones correctas.

Lo importante es definir que robot se va a utilizar, igualmente que instrucciones nuevas se necesitan para escribir el programa solución.

La técnica de refinamiento paso a paso determina escribir primero el programa usando un robot cualquiera y nombre de las instrucciones que se necesita, seguidamente definir el nuevo robot y sus instrucciones. Es decir, escribir la secuencia de mensajes en el bloque de la tarea principal para escribir las definiciones y los nuevos nombres de las instrucciones que se usan dentro de este bloque. Finalmente, se agrupan estos segmentos del programa en un programa completo.

En la figura 59 “Tarea del cosechador”, presenta una tarea de cosecha que requiere un robot para recoger un campo rectangular de pitos. El cuadro ilustra

la tarea de la cosecha, nuestro primer paso es desarrollar un plan total para dirigirnos a escribir un programa del robot que permite que Karel realice la tarea. El planeamiento es probablemente el mejor hecho como actividad del grupo.

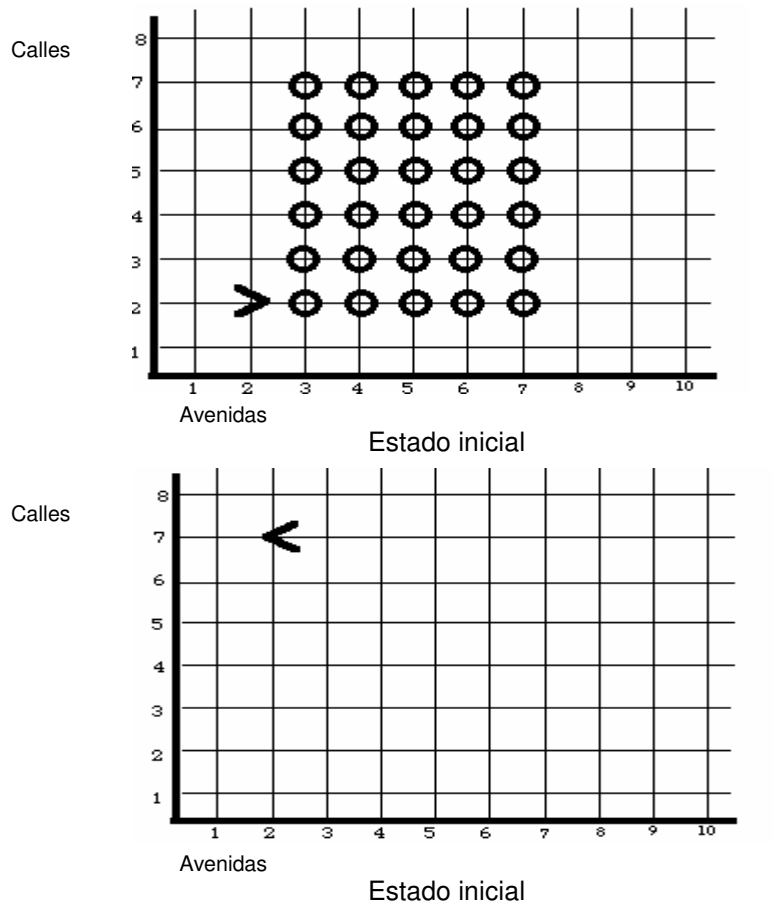


Figura 103 Estados inicial y final "Tarea del cosechador"

Compartir las ideas de solución de los problemas en grupo permite que los miembros presenten diversas formas de solución que se puedan discutir cuidadosamente para establecer verdadera solución al problema. Se pueden usar patrones de preguntas y de respuestas, tal como se relacionan a continuación:

PREGUNTA: ¿Cuántos robots se necesitan para el desarrollo de esta tarea?

RESPUESTA: Podríamos hacerla con un robot que camina hacia adelante y hacia atrás sobre todas las filas que se cosecharán, o podríamos hacerla con un equipo de robots.

PREGUNTA: ¿Cuántos robots se van a utilizar?

RESPUESTA: Vamos a intentarlo con uno y el nombre del robot es Marcos.

PREGUNTA: ¿Cómo puede Marcos recoger una fila?

RESPUESTA: Marcos podría moverse del oeste al este a través de la fila de pitos, recogiendo un pito cada vez que se mueva.

PREGUNTA: ¿Cómo puede Marcos recoger el campo entero?

RESPUESTA: Marcos podía dar vuelta alrededor y moverse de nuevo al lado occidental del campo, moviendo al norte un bloque, cara al este, y repitiendo las ordenes de la instrucción. Marcos podía hacer esto para cada fila de pitos en el campo.
Marcos no está parado en un pito, lo movemos al primer pito antes de comenzar a cosechar la primera fila. El siguiente paso es escribir el bloque de la instrucción de la tarea principal del programa usando en inglés las primitivas de los nuevos nombres del mensaje.

```
task
{       Cosechador  Marco(2, 2, East, 0);
        Marco.move();
        Marco.harvestOneRow();
        Marco.returnToStart();
        Marco.moveNorthOneBlock();
        Marco.harvestOneRow();
        Marco.returnToStart();
        Marco.moveNorthOneBlock();
        Marco.harvestOneRow();
        Marco.returnToStart;
        Marco.moveNorthOneBlock();
        Marco.harvestOneRow();
        Marco.returnToStart();
        Marco.moveNorthOneBlock();
```

```
        Marco.harvestOneRow();
        Marco.returnToStart();
        Marco.moveNorthOneBlock();
        Marco.harvestOneRow();
        Marco.returnToStart();
        Marco.turnOff();
}
```

Obsérvese como se ha diseñado una nueva clase del robot que pueda realizar tres nuevos mensajes. Se puede pensar en una clase de robot como mecanismo para crear abastecedores de servicio.
Estos robots son un ejemplo de objetos en la programación orientada a objetos, proporcionan servicios específicos cuando los mensajes enviados solicitan su servicio más allá de los servicios básicos que todos los ur_Robots() pueden proporcionar, como son servicios de: harvestOneRow(), returnToStart(), y moveNorthOneBlock().

PREGUNTA: ¿Cuáles son las fortalezas de este plan?

RESPUESTA: El plan se aprovecha del nuevo mecanismo de la instrucción y permite que Marcos coseche los pitos.

PREGUNTA: ¿Cuáles son las debilidades del plan?

RESPUESTA: Marcos hace algunos viajes "para vaciar".

PREGUNTA: ¿Cuáles son estos viajes vacíos?

RESPUESTA: Marcos vuelve al punto de partida en la fila que acaba de ser cosechada.

PREGUNTA: ¿Por qué esto es malo?

RESPUESTA: Porque el robot es un recurso valioso que se debe utilizar eficientemente.

PREGUNTA: ¿Puede Marcos recoger más pitos?

RESPUESTA: En vez de cosechar solamente una fila y entonces dar vuelta alrededor y de volver al comienzo, Marcos puede cosechar una fila, mover al norte una calle y volverse al oeste que coseche una segunda fila. Marcos entonces mueve una calle al norte para comenzar el proceso entero encima para las dos filas siguientes.

PREGUNTA: ¿Qué ventaja ofrece el primer plan?

RESPUESTA: Marcos hace solamente seis viajes a través del campo en vez de doce. No hay viajes vacíos.

PREGUNTA: ¿Cuáles son las debilidades de este nuevo plan?

RESPUESTA: Ninguna que podemos ver, ya que existe un número par de filas. Cuando planteamos soluciones del planeamiento, debemos ser muy críticos y no apenas aceptar el primer plan como el mejor. Ahora tenemos dos diversos planes y usted puede pensar probablemente en otras soluciones. Lo importante es evitar los viajes vacíos y poner el segundo plan en ejecución.

```
task
{       cosechador Marco(2, 2, East, 0);
        Marco.move();
        Marco.harvestTwoRows();
        Marco.positionForNextHarvest();
        Marco.harvestTwoRows();
        Marco.positionForNextHarvest();
        Marco.harvestTwoRows();
        Marco.move();
        Marco.turnOff();
}
```

4.10. PLANEAR CON HARVESTTWOROWS Y LA POSICIÓN DE FORNEXTHARVEST

La segunda etapa del plan contiene dos subtareas: Una cosechar dos filas y la otra posicionar a Marcos para cosechar dos filas más. El planteamiento de estas dos subtareas debe ser tan cuidadoso y exacto como debe ser para la tarea total.

4.10.1..Planear con la instrucción HarvestTwoRows

PREGUNTA: ¿Qué se hace en HarvestTwoRows?

RESPUESTA: El plan HarvestTwoRows consiste en cosechar dos filas de pitos. Una será cosechada como recorrido de Marcos al este y la segunda será cosechada como vuelta de Marco al oeste.

PREGUNTA: ¿Qué tiene que hacer Marcos?

RESPUESTA: Marcos debe recoger los pitos y moverse mientras que viaja al este. En el final de la fila de pitos, Marcos debe mover al norte un bloque, hacer frente al oeste, y volver al borde occidental de los pitos de la cosecha del campo mientras que viaja al oeste.

Modelo para un plan general con la instrucción HarvestTwoRows().

```
class Harvester extends ur_Robot
{
        void  harvestTwoRows()
        {
                harvestOneRowMovingEast();
                goNorthToNextRow();
                harvestOneRowMovingWest();
        }
        ...
}
```

Analicemos el plan HarvestTwoRows() buscando sus fortalezas y sus debilidades.

PREGUNTA: ¿Cuáles son las fortalezas del plan HarvestTwoRows()?

RESPUESTA: Solución de problemas.

PREGUNTA: ¿Cuáles son las debilidades de este plan?

RESPUESTA: Una, tener dos posibles instrucciones para cosechar una sola fila de pitos.

PREGUNTA: ¿Realmente se necesitan dos posibles instrucciones para cosechar una sola fila?

RESPUESTA: Necesitamos una instrucción para ir al este y otra para regresar al oeste.

PREGUNTA: ¿Realmente necesitamos una instrucción separada para cada dirección?

RESPUESTA: En la dirección que se mueva Marcos no importa. Si planeamos el goToNextRow cuidadosamente, podemos utilizar una instrucción de cosechar una fila de pitos cuando va Marcos al este y la misma instrucción para que vaya al oeste. Nuestro análisis nos demuestra que podemos reutilizar una sola palabra (harvestOneRow) en vez de definir dos instrucciones similares, haciendo nuestro programa más pequeño.

A continuación se muestra un modelo para el uso de la instrucción harvestTwoRows()

```
void harvestTwoRows()
{
```

```
                    // Antes de realizar la instrucción el robot debe estar orientado al este
                    // Sobre el primer pito que se encuentra en la fila
                     harvestOneRow();
                     goToNextRow();
                     harvestOneRow();
          }
```

4.10.2 Planear con la instrucción Fornextharvest()

PREGUNTA: ¿Qué hace la instrucción ForNextHarvest()?

RESPUESTA: Se utiliza esta instrucción cuando Marcos está en el lado occidental del campo del pito. Se mueve el robot al norte un bloque y hace giro al este en la posición para cosechar dos más filas de pitos.

PREGUNTA: ¿Qué tiene que hacer Marcos?

RESPUESTA: Marcos debe dar vuelta a la derecha girando al norte, se mueve un bloque y da vuelta a la derecha girando a este. Ponemos esta instrucción en ejecución como sigue.

```
class Cosechador  : ur_Robot
  {
      void positionForNextHarvest();
      void  turnRight();
  }
          void Cosechador ::positionForNextHarvest()
          {
               // Antes de realizar la instrucción el robot debe estar orientado al oeste
              //  Sobre la primera esquina de la fila
                    turnRight();
                    move();
                    turnRight();
          }

          void Cosechador :: turnRight()
          {
```

```
        turnLeft();
        turnLeft();
        turnLeft();
    }
```

4.10.3 Planear con Harvestonerow() y Gotonextrow()

Ahora centramos nuestros esfuerzos en harvestOneRow() y finalmente goToNextRow().

PREGUNTA: ¿Qué hace harvestOneRow()?

RESPUESTA: Comenzando en el primer pito y haciendo frente a la dirección correcta, Marcos debe cosechar cada uno de las esquinas que encuentra, parando en la localización del último pito en la fila.

PREGUNTA: ¿Qué tiene que hacer Marcos?

RESPUESTA: Marcos debe ejecutar una secuencia de harvestCorner() y mover instrucciones de recoger los cinco pitos en la fila.

PREGUNTA: ¿Cómo Marcos cosecha una sola esquina?

RESPUESTA: Marcos debe ejecutar una instrucción del recogerpitos. Podemos poner el harvestOneRow() y el harvestCorner() en ejecución como sigue.

```
class Cosechador : ur_Robot
 {
   void harvestOneRow();
   void harvestCorner();
 };
        ...
        void Cosechador:: harvestOneRow()
        {
```

```
        harvestCorner();
        move();
        harvestCorner();
        move();
        harvestCorner();
        move();
        harvestCorner();
        move();
        harvestCorner();
}

void Cosechador:: harvestCorner()
{
        pickBeeper();
}
```

4.10.4 Planear con la instrucción goToNextRow

PREGUNTA: ¿Qué hace goToNextRow()?

RESPUESTA: Esta instrucción mueve a Marcos hacia el norte un bloque, a la fila siguiente.

PREGUNTA: Por que no se puede utilizar la posición ForNextHarvest()?

RESPUESTA: Porque no trabajará correctamente. Cuando utilizamos la posición ForNextHarvest(), Marcos debe estar mirando al oeste. Marcos ahora está mirando al este, en está posición la instrucción ForNextHarvest() no trabaja.

PREGUNTA: ¿Qué tiene que hacer Marcos?

RESPUESTA: Marcos debe girar a la izquierda girando al norte, mover un bloque, y dar vuelta a la izquierda girando al oeste. Finalmente pone en práctica la nueva instrucción.

Se debe simular la instrucción posición ForNextHarvest() en el papel. Comience ubicando a Marcos al oeste y observe donde está el robot cuando usted acabe de simular la instrucción.

Uso de la instrucción goToNextRow()

```
class Cosechador: ur_Robot
{
void goToNextRow();
};          ...
            void Cosechador goToNextRow()
            {
                        // Antes de realizar la instrucción el robot debe estar orientado al este
                        //   Sobre la ultima esquina de la fila
                        turnLeft();
                        move();
                        turnLeft();
            }
```

Se puede utilizar la simulación para analizar esta instrucción e igualmente demostrar que es correcta y el programa está realizado.

Paso final: Verifica que el programa completo es correcto

La verificación del programa nos permite comprobar una vez más que la solución del problema es correcta.

```
class Cosechador : ur_Robot
 {
   void harvestTwoRows();
   void positionForNextHarvest();
   void turnRight();
   void harvestOneRow();
   void harvestCorner();
   void goToNextRow();
 };
            void  Cosechador::harvestTwoRows()
            {
                        // Antes de realizar la instrucción el robot debe estar orientado al este
```

```
            //  Sobre el primer pito encontrado en la fila
            harvestOneRow();
            goToNextRow();
            harvestOneRow();
}
void  Cosechador::positionForNextHarvest()
{
            // Antes de realizar la instrucción el robot debe estar orientado al oeste
            //   Sobre la ultima esquina de la fila
            turnRight();
            move();
            turnRight();
}

void Cosechador:: turnRight()
{           turnLeft();
            turnLeft();
            turnLeft();
}

void   Cosechador::harvestOneRow()
{           harvestCorner();
            move();
            harvestCorner();
            move();
            harvestCorner();
            move();
            harvestCorner();
            move();
            harvestCorner();
}

void   Cosechador::harvestCorner()
{           pickBeeper();
}
void   Cosechador::goToNextRow()
{
            // Antes de realizar la instrucción el robot debe estar orientado al este
            // Sobre la ultima esquina de la fila
            turnLeft();
```

```
            move();
            turnLeft();
        }

task
{       Cosechador Marco(2, 2, East, 0);
        Marco.move();
        Marco.harvestTwoRows();
        Marco.positionForNextHarvest();
        Marco.harvestTwoRows();
        Marco.positionForNextHarvest();
        Marco.harvestTwoRows();
        Marco.move();
        Marco.turnOff();
}
```

Se debe simular la ejecución del programa de Marcos completamente para demostrar que todos los subprogramas trabajan correctamente y para estar seguros que el programa es correcto.

4.11 LAS VENTAJAS DE USAR NUEVAS INSTRUCCIONES

Una ventaja de dividir un programa en subprogramas utilizando nuevas instrucciones, así estas instrucciones se ejecuten solamente una vez. Las nuevas instrucciones estructuran programas, y las palabras y las frases inglesas hacen programas más comprensibles.

Los programas apenas se han leído y escrito y vemos con el fin de encontrar instrucciones que estén confusas o difíciles de entender.

En el ejemplo de la cosechadora es típico para utilizar la instrucción harvestField() como servicio. La utilización de este servicio nos permite cosechar un campo entero. Podemos agregar fácilmente esta característica a la clase Cosechadora. Su uso puede esstar en todas las declaraciones del bloque de la tarea principal, exceptuando la primera y la ultima.

```
class Field_Harvester : Cosechador
{
        void harvestField()
        {       move();
                harvestTwoRows();
                positionForNextHarvest();
                harvestTwoRows();
                positionForNextHarvest();
                harvestTwoRows();
                move();
        }
}
```

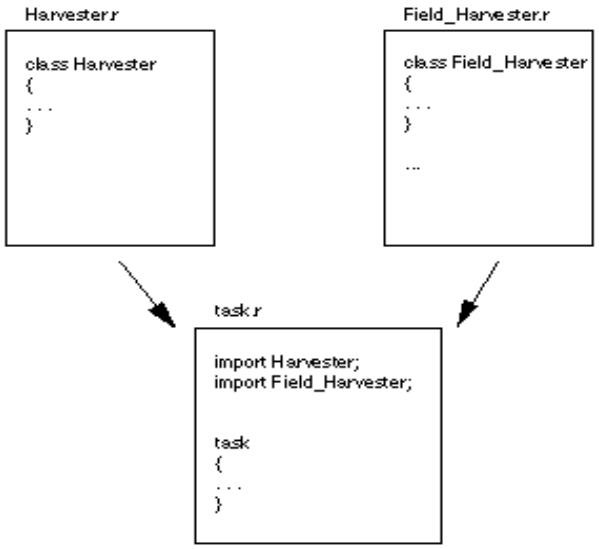


Figura. 104 Definición de la clase cosechador

Para utilizar esta nueva clase se debe incluir su definición, igualmente la definición de la clase Cosechador y el bloque de la tarea principal en un sólo archivo. La forma más eficiente para hacer esto es aprovechar la característica de la importación de la lengua del robot .

Si con antelación se conoce la definición del archivo nombrado "Field_Harvester.r" y pusimos las definiciones de la clase Cosechador() en otro diverso archivo "Harvester.r." Ninguno de los dos archivos contiene un bloque de la tarea principal.

Se puede especificar una tarea dentro de un archivo.

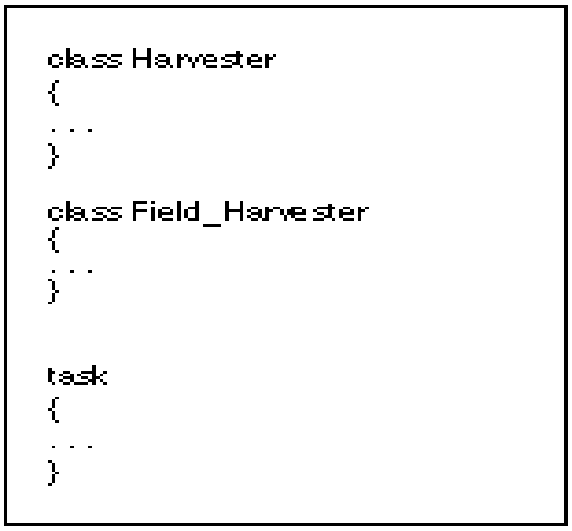


Figura 105 Definición estructural de las clases Harvester() y field_Harvester()

```
import cosechador;
import Field_Harvester;

task
{       Field_Harvester Tony (2, 2, East, 0);
        Tony.harvestField();
        Tony.turnOff();
}
```

Las primeras dos líneas dicen que la instalación incluirá el contenido entero de Harvester.r y de Field_Harvester.r en este programa en lugar de las instrucciones de la importación.

Usar la importación le dice a la instalación que lea otra descripción, tal como Harvest.r, en cierto lugar de especificación.

4.12 EVITANDO ERRORES

Muchos principiantes piensan que todo esto de planear, analizar, ejecutar y simular los programas toma demasiado tiempo. Lo que realmente toma tiempo es estar corrigiendo errores.

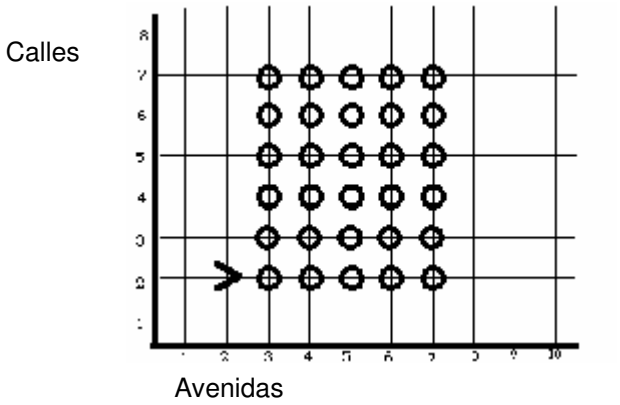
Estos errores se clasifican en dos amplias categorías: Los errores del planeamiento de ejecución y de intento, suceden cuando escribimos un programa sin un plan de bien pensado y podemos perder mucho tiempo de programación. Son generalmente difíciles de encontrar porque los segmentos grandes del programa pueden ser modificados o desechados. El planeamiento y el análisis cuidadoso del plan pueden ayudarnos a evitar errores del planeamiento. Los errores de programación léxicos y de sintaxis, suceden cuando escribimos realmente el programa. Si escribimos el programa entero sin la prueba de él, tendremos indudablemente muchos errores a corregir, algunos de los cuales pueden ser casos múltiples del mismo error.

Al escribir el programa en bloques se reduce el número total de errores introducidos en cualquier instante y pueden prevenir ocurrencias múltiples del mismo error. La técnica de refinamiento paso a paso es una herramienta que permite la planeación y el análisis, para poner los planes en ejecución de tal manera que permita conducir a un programa del robot que contenga un mínimo de errores.

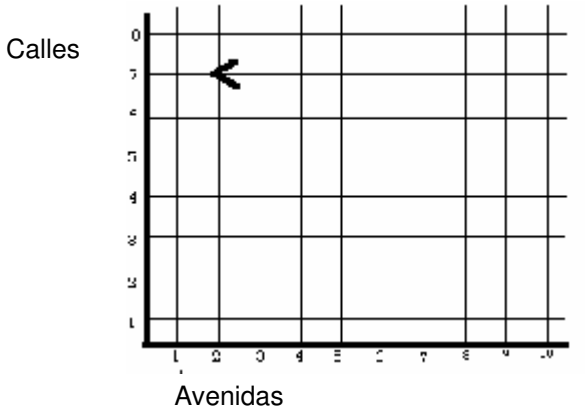
4.13 MODIFICACIONES FUTURAS

En el capítulo anterior, se dijo que es necesario escribir programas que sean fáciles de leer, de entender, de eliminar errores y de modificar.

El mundo del robot puede ser modificado fácilmente para que los programas existentes se mantengan en el robot. Puede ser mucho más simple y toma menos tiempo para modificar un programa existente, para realizar una tarea, para escribir totalmente un nuevo programa. A continuación se presentan dos situaciones que diferencian la tarea del Cosechador().



Estado inicial



Estado final

Figura 106 Estados inicial y final tarea del cosechador

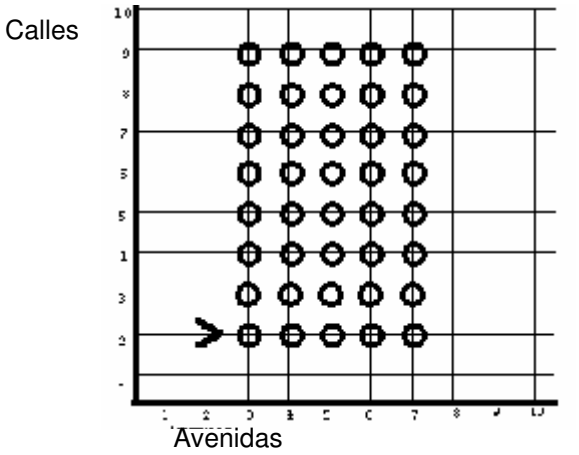


Figura. 107 Tarea del cosechador con más filas

Es difícil modificar nuestra clase del Cosechador() y el programa que la contiene. El problema es fácil de solucionar. ¿Qué pasa si agregamos dos nuevas líneas al bloque original de la tarea principal para solucionar la nueva tarea? No necesitamos ningún cambio a la clase del Cosechador() por sí mismo.

El uso de nuevas instrucciones permite encontrar rápidamente dónde se necesita realizar el cambio.

Realizamos un cambio simple al harvestOneRow como sigue,

```
void harvestOneRow()
{
        harvestCorner();
        move();
        harvestCorner();
        move();
        harvestCorner();
        move();
        harvestCorner();
        move();
        harvestCorner();
        move();                 // Adiciones estas dos
        harvestCorner();        // Nuevos mensajes
}
```

Este cambio a la clase del Cosechador() está muy bien a condición de que no necesitamos solucionar el problema original en el futuro. Es ciertamente ventajoso dejar la clase Cosechador() sin cambios y crear una nueva clase, Long_Harvester() que contenga esta instrucción modificada del harvestOneRow().

```
class Long_Harvester extends Harvester
{
        void  harvestOneRow()
        {
                super.harvestOneRow();          // Ejecute la instrucción
```

```
                move();                 // Adiciones estas dos
                harvestCorner();        // Nuevos mensajes
        }
}
```

El uso de nuevas instrucciones simplifica la tarea de encontrar y fijar errores. Esto es cierto si las instrucciones son cortas y pueden ser fácilmente entendidas.

Si el robot marca una vuelta incorrecta e intenta recoger un pito en el lugar incorrecto. Utilizando nuevas instrucciones para escribir el programa, y que cada nueva instrucción realice una tarea específica. La posición ForNextHarvest() y un sistema de controles de las tareas relacionadas con harvestTwoRows() nos permite determinar generalmente la localización probable del error.

4.14 PROGRAMAR SIN NUEVAS INSTRUCCIONES

A continuación se presenta un programa que procura solucionar el problema de los pitos, planteado inicialmente con sólo instrucciones primitivas.

Examine el programa y haga las mismas preguntas que acabamos de exponer:

¿Dónde cambiaríamos el programa para solucionar la primera situación modificada? ¿ dónde cambiaríamos el programa para solucionar la segunda situación modificada? - suponga que Marcos da una vuelta incorrecta mientras que recoge los pitos. ¿Dónde miraríamos para corregir el error?

```
task
{       ur_Robot Marco = new ur_Robot(2, 2, East, 0);
        Marco.move();
        Marco.pickBeeper();
        Marco.move();
        Marco.pickBeeper();
        Marco.move();
        Marco.pickBeeper();
```

```
Marco.move();
Marco.pickBeeper();
Marco.move();
Marco.pickBeeper();
Marco.turnLeft();
Marco.move();
Marco.turnLeft();
Marco.pickBeeper();
Marco.move();
Marco.pickBeeper();
Marco.move();
Marco.pickBeeper();
Marco.move();
Marco.pickBeeper();
Marco.move();
Marco.pickBeeper();
Marco.turnRight();
Marco.move();
Marco.turnRight();
Marco.pickBeeper();
Marco.move();
Marco.pickBeeper();
Marco.move();
Marco.pickBeeper();
Marco.move();
Marco.pickBeeper();
Marco.move();
Marco.pickBeeper();
Marco.turnRight();
Marco.move();
Marco.turnLeft ();
Marco.pickBeeper();
Marco.move();
Marco.pickBeeper();
Marco.move();
Marco.pickBeeper();
Marco.move();
Marco.pickBeeper();
Marco.move();
Marco.pickBeeper();
```

```
        Marco.turnLeft();
        Marco.move();
        Marco.turnLeft();
        Marco.pickBeeper();
        Marco.move();
        Marco.pickBeeper();
        Marco.move();
        Marco.pickBeeper();
        Marco.move();
        Marco.pickBeeper();
        Marco.move();
        Marco.pickBeeper();
        Marco.move();
        Marco.turnOff();
}
```

Las listas largas de mensajes como éstas, pueden solucionar correctamente un problema pero son muy difíciles de leer y de entender, además, son muy difíciles para eliminar errores y para modificarse.

4.15 ESCRIBIR PROGRAMAS COMPRENSIBLES

En la programación de Karel, el éxito está en su correcta escritura de los programas para lograr su comprensibilidad, ya que de ésta depende la claridad para buscar la solución de los problemas. Si un programa es comprensible, es fácil eliminar sus errores. ¿Qué se necesita para que un programa sea fácil de entender?

Para que un programa sea fácil de entender se necesitan dos criterios:

Primero, que el programa este compuesto por bloques simples, fáciles y comprensibles. Que cada parte del programa sea entendible.

Segundo, que el programa este dividido en bloques pequeños y fáciles de entender. Debemos cerciorarnos de nombrar las nuevas instrucciones

correctamente. Estos nombre proporcionan una descripción (posiblemente la única descripción) de las ordenes que ejecuta la instrucción.

El lenguaje de programación del robot permite elegir cualquier nombre de la instrucción, pero con esta libertad viene la responsabilidad de seleccionar nombres exactos y descriptivos. Porque es más fácil verificar o eliminar errores de un programa que contenga nuevas instrucciones.

Las nuevas instrucciones pueden ser probadas independientemente. Al escribir un programa debemos simular y verificar cada instrucción inmediatamente hasta que sea correcta. Entonces podemos olvidarnos cómo trabaja la instrucción y sólo recordar que hace la instrucción. El recordar debe ser fácil, si nombramos la instrucción correctamente. Esto es más fácil, si la instrucción ejecuta solamente una orden.

Las nuevas instrucciones imponen una nueva estructura a nuestros programas, entonces, podemos utilizar esta estructura para encontrar la solución del problema. Para eliminar los errores de un programa, primero debemos encontrar las nuevas instrucciones que están funcionando incorrectamente.

Un fenómeno psicológico interesante, está relacionado con el mecanismo de definición de la instrucción del robot, con el razonamiento humano. El cerebro humano puede centrarse en una cantidad de información limitada en cualquier momento. La capacidad que tiene el robot de no hacer caso de los detalles que no son relevantes es una gran ayuda para programar la escritura y eliminar errores.

La mayoría de los programadores principiantes tienen la tendencia a escribir las definiciones demasiado grandes en la instrucción. Lo correcto es escribir pequeñas definiciones e ir nombrando las instrucciones, a cambio de escribir grandes definiciones.
Escribir programas comprensibles con nuevas instrucciones y usar la técnica del refinamiento paso a paso, reduce el número de errores y la cantidad de tiempo para la programación del robot de escritura.

4.16 UNA TAREA PARA DOS ROBOTS

En esta sección comencemos con una tarea para dos robots que no nos restringe a usar un solo robot. Podemos tener tantos robots como tantas tareas se quieran programar.

Los robots pueden comunicarse de maneras, como órdenes reciba en su programación. Por ejemplo está es una tarea simple para dos robots. Karel se encuentra en la calle 3ª con avenida 1ª, tiene un pito, y está orientado al este. Carlos está en los revestimientos del origen del este. Karel debe llevar el pito a Carlos y ponerlo abajo. Carlos debe entonces tomarlo y llevarlo a la calle 1ª con avenida 3ª. El pito se debe colocar en esta esquina.

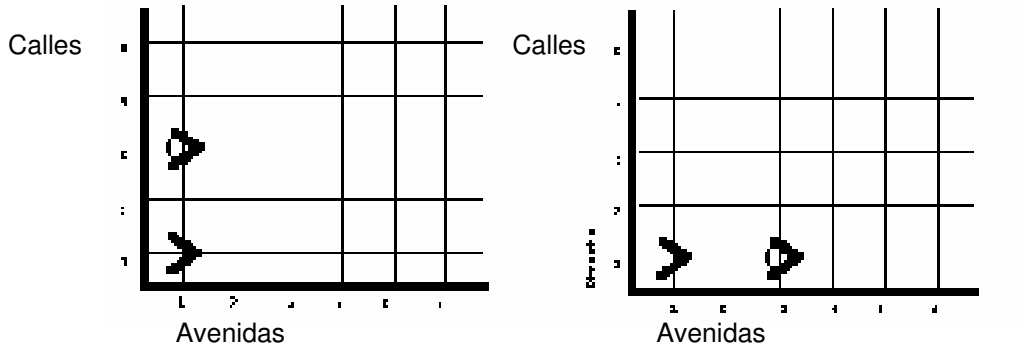


Figura. 108 Posibles estados inicial y final para una tarea con dos robots

```
task
{ur_Robot Karel ur_Robot(3, 1, East, 0);
ur_Robot Carl  ur_Robot(1, 1, East, 0);

Karel.pickBeeper();
Karel.turnLeft();
Karel.turnLeft();
Karel.turnLeft();
Karel.move();
Karel.move();
Karel.putBeeper();
```

```
Carl.pickBeeper();
Carl.move();
Carl.move();
Carl.putBeeper();
Karel.turnOff();
Carl.turnOff();
}
```

Resulta fácil solucionar problemas con la ayuda de un equipo o grupo de robots, lo importante es el uso de las nuevas instrucciones que se deben emplear, por ejemplo un mensaje tipo HarvestTwoRows() sería el apropiado para este tipo de tareas (Véase el problema del cosechador).

La solución del programa será la siguiente:

```
import Harvester;

task
{       Harvester Karel = new Harvester(2, 2, East, 0);
        Harvester Kristin = new Harvester(4, 2, East, 0);
        Harvester Matt = new Harvester(6, 2, East, 0);
        Karel.move();
        Karel.harvestTwoRows();
        Karel.turnOff();
        Kristin.move();
        Kristin.harvestTwoRows();
        Kristin.turnOff();
        Matt.move();
        Matt.harvestTwoRows();
        Matt.turnOff();
}
```

El problema del cosechador también se puede solucionar con seis robots, una manera aún más interesante de hacer esto, es dejar a un robot que coordine las acciones de los otros. Para que este proceso funcione, necesitamos dos clases de robot diferentes. Una clase de robots será llamada un Coreógrafo, porque dirige los otros, que son los robots corrientes y los otros robots son los

estándares de la edición ur_Robot(). El truco aquí es que el Coreógrafo instalará a los otros robots y después garantizando que atenderán las acciones del Coreógrafo.

El Coreógrafo necesita saber los nombres de los otros dos robots, así que damos nombres a estos robots privados del Coreógrafo. Pero como no se ha visto esta característica en el lenguaje de programación del robot. Entonces declaramos el robot momentos antes del bloque de la tarea principal, es permitido definir nombres del robot dentro de una nueva clase. Los robots declarados están disponibles, los ayudantes de la clase de robot ya están declarados, pero no se pueden utilizar por otros robots o en el bloque de la tarea principal.

Esto es, porque los nombres del ayudante del robot serán privados en la nueva clase de robot. El Coreógrafo también necesitará eliminar todos los métodos del robot. Si ordenamos al Coreógrafo que se mueva, éste puede ordenar a los otros robots que se muevan también.

```
class Choreographer : ur_Robot
{
        ur_Robot Lisa(4,2,East,0);      // El primer robot ayudante
        ur_Robot Tony(6,2,East,0);      // El Segundo robot ayudante
        void harvest();
        void harvestARow();
        void harvestCorner();
        void move();
        void pickBeeper();
        void turnLeft();
        void turnOff();
}
```

Aquí es el bloque de tarea principal de nuestro programa.

```
task
{       Choreographer Karel(2, 2, East, 0);
        Karel.harvest();
```

```
        Karel.turnOff();
}
```

Aquí está la clase completa del Coreógrafo, con algunas anotaciones.

```
class Choreographer extends ur_Robot
{
        ur_Robot Lisa = new ur_Robot(4,2,East,0);          // El primer robot ayudante
        ur_Robot Tony = new ur_Robot(6,2,East,0);          // El Segundo robot ayudante
```

Harvest y harvestARow son similares como se ha visto anteriormente.

```
        void harvest()
        {
                harvestARow();
                turnLeft();
                move();
                turnLeft();
                harvestARow();
        }

        void harvestARow()
        {
                move();
                harvestCorner();
                move();
                harvestCorner();
                move();
                harvestCorner();
                move();
                harvestCorner();
                move();
                harvestCorner();
                move();
        }

        void harvestCorner()
        {
                pickBeeper();
```

```
}
```

La clave está en definir un robot Coreógrafo y el grupo de robots que trabará con él en la redefinición de métodos heredados. A manera de ejemplo se muestra un caso de movimiento único:

```
void  move()
{           super.move();
            Lisa.move();
            Tony.move();
}
```

En cada uno de los otros métodos, primero se ejecuta la instrucción heredada del mismo nombre y enseguida se envía el mensaje a los dos robot del ayudante.

```
        ... // similar al método move()
}
```

Cuando ordenamos que se mueva el robot del Coreógrafo (aquí Karel) primero ejecuta la instrucción heredada del movimiento y enseguida envía mensajes de movimiento a los dos robots ayudantes, que pertenecen a la clase ur_Robots(), sin afectar a ningún otro robot. Esto significa que siempre que Karel se mueva, cada uno de sus ayudantes también lo hará "automáticamente." Igualmente para las instrucciones pickBeeper() y turnleft(). Cuando un robot envía un mensaje a otro robot, el mensaje pasa del remitente a través del satélite al otro robot. El robot remitente debe entonces esperar que termine la instrucción enviada antes de reasumir su propia ejecución.

- **Diseño orientado a objetos.**

Anteriormente se aprendió una técnica útil para diseñar una sola clase, preguntando qué tareas son necesarias para esta y de diseñar las tareas complejas en tareas más simples. Ahora vamos a mirar este diseño desde una perspectiva más amplia. Se debe prever que podemos necesitar varias clases

de robot para realizar cierta tarea y estos robots pueden cooperar de cierta manera. Aquí se discute solamente las ediciones del diseño

En el mundo real, la construcción de casa es una tarea moderadamente compleja, ya que, es hecha generalmente por un equipo de constructores, cada uno con su propia especialidad. Quizás sea igual en el mundo del robot.

Si se mira algunos de nuestros ejemplos anteriores, se encontrará que hay realmente dos clases de instrucciones. La primera clase de instrucción, es el harvestTwoRows() en la clase Cosechadora que significa ser utilizada en el bloque de la tarea principal, la segunda clase es goToNextRow(), que tiene significado para ser utilizada internamente como parte de la descomposición del problema. Por ejemplo, la instrucción goToNextRow() es poco probable para ser utilizada excepto dentro de otras instrucciones.

La primera clase de instrucción tiene significado para ser pública y define de una cierta manera qué se piensa hacer con el robot. Si pensamos en un robot como servidor, entonces su cliente (el bloque de la tarea principal, o quizás otro robot) le enviará un mensaje con una de sus instrucciones "públicas". El cliente es quién solicita un servicio de un servidor. El robot Cosechador proporciona un servicio de cosecha. Su cliente solicita ese servicio. El lugar para comenzar el diseño de las instrucciones está con los servicios públicos y los servidores que los proporcionan. El robot propiamente servidor, ejecutará otras instrucciones para proporcionar el servicio. Podemos utilizar el refinamiento sucesivo para ayudar a diseñar las otras instrucciones que ayudan al servidor a realizar su servicio.

La manera más fácil de construir una casa es contratar los servicios de un especialista, que montará un equipo apropiado para construir nuestra casa.

Consideremos como robot contratista para construir la casa, el trabajo se considera de alguna manera hecho. El cliente no se preocupa cómo el contratista realiza la construcción mientras el resultado sea aceptable.

Note que nosotros hemos dado el diseño preliminar para una clase, la clase del contratista, con un método público: buildHouse(). Con esta metodología se ve la necesidad de descubrir nuevos métodos y otras clases de instrucciones.

También, sabemos que necesitaremos probablemente un solo robot de la clase contratista. Vamos a nombrarlo Kristin, para tener un nombre de referencia. La idea es examinar la tarea desde el punto de vista de Kristin. ¿Qué necesita hacer Kristin para construir una casa?. Una posibilidad de colocar todos los pitos apropiadamente, la otra forma es utilizar robots especialistas para construir cada una de las partes de la casa, por ejemplo: las paredes, la azotea, las puertas y las ventanas. Si quisiéramos que las paredes fueran hechas de ladrillo, debemos contar con los servicios de uno o más robots especializados para tal fin.

Las puertas y las ventanas se podían construir por un par de robots de carpinteria, y la azotea construirla por un robot techador. El contratista, Kristin necesita reunir a este equipo. Nosotros necesitamos otro método para esto, el equipo podría ser llamado por el cliente o comenzar la construcción. Kristin también necesita poder conseguir el equipo para empezar la obra.

Necesitamos un nuevo método gatherTeam() del contratista. ocupándose, de los trabajos más pequeños. El robot albañil debe responder a un mensaje de buildWall(). El contratista debe indicar al albañil donde deben ser construidas las paredes. De igual forma, el robot techador debe responder al mensaje makeRoof(), y el robot carpintero debe conocer los mensajes de makeDoor() y de makeWindow(). Puede ser que vayamos un paso más lejos con el techador y decidamos que sería provechoso hacer los dos aguilones de la azotea por separado. También quisiéramos que un techador pudiera hacer makeLeftGable() y a makeRightGable().

```
class Mason : ur_Robot
 {
   void buildWall();
 }
```

```
            void Mason :: buildWall()
            {

            }

class Carpintero : ur_Robot
  {
     void makeWindow();
     void makeDoor();
  }
            void Carpintero:: makeWindow()
            {

            }

            void Carpintero::makeDoor()
            {

            }


class Roofer : ur_Robot
 {
     void makeRoof();
     void makeLeftGable();
     void makeRightGable();
 }
            void  Roofer::makeRoof()
             {
                        ...
             }
            void Roofer:: makeLeftGable()
            {
                        ...
            }
            void Roofer:: makeRightGable()
            {
            }
```

El equipo de constructores está definido por el contratista, quien conoce sus nombres para evitar equivocaciones en las órdenes y sea él quien diga a los constructores que tiene que hacer cada uno y no el cliente decir que se tiene que hacer.

En la clase del contratista se debe declarar a estos robots con nombres privados y no con nombres globales, ya que, esto podría crear confusión.

Una declaración de la clase contratista sería la siguiente:

```
class Contratista :  ur_Robot
{
        Mason Ken_the_Mason = new Mason(1,1,E,??);
        Roofer  Sue_the_Roofer = new Roofer(...);
        Carpenter Linda_the_Carpenter = new Carpenter(...);
        Carpenter Steve_the_Carpenter = new Carpenter(...);

        void gatherTeam()// Call prior to first move.
        {
                ...// mensajes aquí en la posicion inicial del grupo.
        }

        void buildHouse()
        {
                ...// mensajes aquí para los 4 trabajadores..

        }
}

task
{       Contractor Kristin = new Contractor(1, 1, East, 0);
        ...
        Kristin.buildHouse();
        ...
        Kristin.turnOff();
}
```

Observe que el contratista es un servidor. Su cliente es el bloque de la tarea principal. Pero note también, que Kristin es un cliente de los cuatro ayudantes puesto que proporcionan los servicios (servicios de la pared a Kristin).

Este hecho es relativamente común en el mundo verdadero para que el cliente y el contratista estén de acuerdo

4.17 PROBLEMAS PROPUESTOS

Los problemas propuestos en esta sección, están definidos con nuevos métodos para Karel. Estos métodos están basados en las técnicas de la programación orientada a objetos, donde es necesario cumplir las nuevas técnicas de solución, como es la de refinamiento paso a paso para la definición de nuevas instrucciones. Además, se debe definir las secuencias dentro de una nueva instrucción.

Detectar y corregir los errores de sintaxis en los programas garantiza la solución correcta de los problemas. Simule la ejecución de karel en cada programa para verificar que la solución es correcta. Las tareas complejas se pueden solucionar con los programas que se dividen en pequeños problemas con instrucciones potentes.

1 Escriba las definiciones apropiadas para los nuevos métodos:

- MoveMile, recordando que las millas son 8 bloques de largo.
- El move_backward, que mueve Karel un bloque al revés, pero lo deja que hace frente a la misma dirección.
- Move_kilo_mile, que mueve Karel 1000 millas adelante.

2 Karel trabaja a veces como perno-pin-setter en un callejón del bowling. Escriba un programa que mande a Karel para transformar la situación inicial en el cuadro 3-5 en la situación final. Karel comienza esta tarea con diez pitos en su beeper-bolso. Cuadro Tarea Perno-Que fija

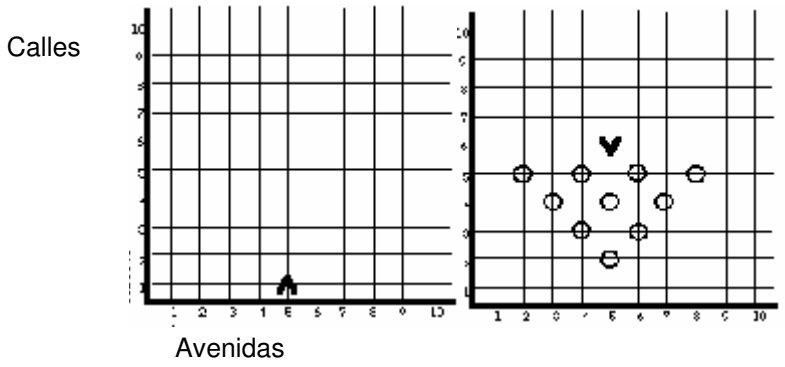


Figura. 109 Estados inicial y final del ejercicio 2

4.17.3 Escriba el programa del cosechador usando la técnica de refinamiento paso a paso.

4.17.4 La figura 110 ilustra un campo de pitos que Karel plantó una noche después de un juego del béisbol. Escriba un programa que coseche todos los pitos.

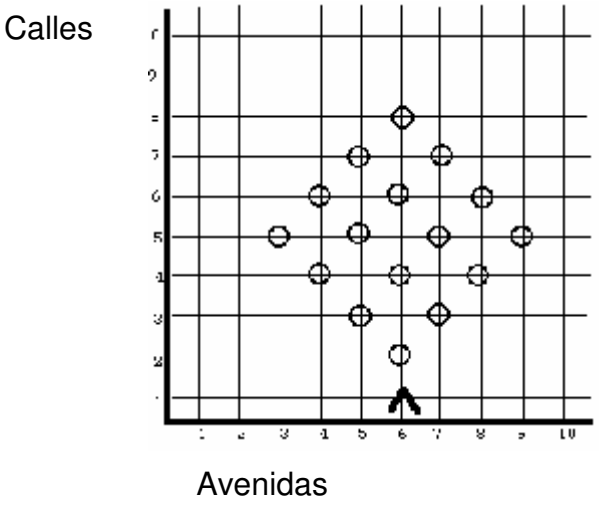


Figura. 110 Juego de Béisbol