3. TECNICAS AVANZADAS PARA ROBOTS

Objetivo: En este capítulo se presentará un número de tópicos bastante innovadores. Primero se presentará el concepto de recursión. Recursión vs procesos repetitivos que, permitirán que los robots ejecuten una secuencia de instrucciones más de una vez. También se establecerá la relación formal entre recursión y repetición. Se estudiarán dos nuevas instrucciones que darán a los robots la capacidad de solucionar algunos problemas de manipulación de pitos, incluyendo computación numérica. Se aplicará la nueva programación orientada a objetos y se verán algunas aplicaciones que serán aprendidas en este tema. Lo anterior puede parecer extraño al comienzo, pero después de explicar varios ejemplos se modificará la situación.

3.1. INTRODUCCIÓN A LA RECURSIÓN

Haciendo un análisis a los procesos repetitivos, ahora se examinará una nueva forma de que un robot repita una acción. La técnica se llama recursión. La palabra significa simplemente recurrir o repetir. Cuando un lenguaje de programación permite definiciones recursivas o recursión significa que un método puede llamarse a sí mismo durante la ejecución del robot. Lo anterior puede parecer extraño al comienzo, pero después de explicar varios ejemplos la confianza retornará como si fuera la ejecución de un robot en proceso repetitivo while. Se notará que la recursión es exactamente otra estructura de control como puede parecer inicialmente, se entenderá fácilmente como una repetición. Está es otra herramienta para adicionar a la colección. Existe el ejemplo de un robot que se mueve de un punto a otro, conociendo exactamente qué distancia se tiene que mover.

Hay muchas situaciones, sin embargo en la cual esto no es verdadero, por ejemplo: La condición inicial considera un robot llamado Kristin colocado en el origen (1,1) mirando al este y hay un pito en cualquier esquina a lo largo de la calle 1. El método deseable debe hacer que Kristin busque el pito, lo recoja y retorne al origen.

La instrucción de llamado al procedimiento sería:

```
Kristin.BuscaPito();
```

Si se sabe que el pito está entre la calle 1 y la calle 23 se escribiría un programa con procesos repetitivos. Se sabe que no hay más qué decir, ¿En un viaje de 15 bloques (bloques de 5 esquinas por ejemplo), se puede encontrar una solución? ¿Cómo? ¿Pero si no conocemos la distancia?. Pensemos inicialmente en la clase encontrar_pito().

```
class Encontrar_Pito: Robot
{
  void encuentraPito( );
  void regresa( );
  void Buscapito( );
}
 void Encontrar_Pito::BuscarPito ( ){
            EncuentraPito( );
                        pickBeeper( );
                        turnLeft( );
                        turnLeft( );
                        Retornar( );
            }
```

Esto efectivamente hace la tarea, se tendrá que escribir otros dos nuevos métodos EncuentraPito() y Retornar(). Ahora se analizará el procedimiento EncuentraPito( ). Se sabe que hay un pito en alguna de las esquinas sobre la calle 1. También se sabe que Kristin está sobre la calle 1 mirando al Este.

Puede ser que Kristin esté ya sobre el pito en una esquina, en este caso no hay nada qué hacer. Sin embargo, se puede comenzar a escribir:

```
void EncuentraPito()
{       if (! nextToABeeper())
        {
                ...
        }
}
```

Terminará correctamente en el caso de que Kristin esté sobre un pito en la esquina. Supóngase que el pito está sobre alguna otra esquina, entonces Kristin necesitará moverse rumbo a otra y comprobar otras esquinas.

```
  void EncuentraPito()
{       if (! NextToABeeper())
        {       move();
                ???
        }
}
```

Bien, ¿Qué necesita hacer Kristin después de moverse? Note que Kristin está en una esquina diferente a la esquina original de donde partió, comprobó y encontró ausencia de pitos. Por consiguiente, se puede concluir que un pito está ahora (después de moverse) ya sea en la localización actual de Kristin o más hacia el este de él. Pero ésta es exactamente la misma situación (relativamente) en la que Kristin está cuando comenzó está tarea (condición inicial). Por consiguiente, se concluye que lo que Kristin necesita hacer ahora exactamente es EncontrarPito() una vez más.

```
void EncuentrePito()
{       if (! nextToABeeper())
        {       move();
                EncuentrePito();
        }
}
```

Obsérvese que EncuentrePito() ha sido definido completamente, pero ha sido definido llamando además es EncuentrePito() dentro de su procedimiento. ¿Qué puede estar haciendo? Bien, supóngase que necesitamos vaciar un océano con un balde. Al ejecutar el procedimiento OceanoVacío(), inicialmente se pregunta si el océano está vacío. Si es así está hecho el trabajo, en caso contrario, se saca un balde de agua justamente del océano y llamamos el procedimiento Oceanovacío().

Es importante pensar que semejante definición recibe el nombre de definición recursiva, y se define un proceso en sus propios términos cada vez menos complejo. Se define un proceso en términos de algo más simple o una versión más pequeña de sí misma. Aquí se define EncuentraPito() en cualquier caso sea que Kristin esté ya en la esquina del pito o no; se usa move(); y EncuentraPito() así Kristin este en cualquier lugar. Es necesario conocer también dónde se ejecuta tal instrucción y dónde efectivamente se encuentra un pito en el camino de karel, de otra manera después de todo movimiento la reejecución de EncuentraPito() encontrará que la prueba es verdadera, generando sin embargo otra reejecución de EncuentraPito() a menos que termine.

El otro método retorna el procedimiento regresar() en forma similar, excepto para la prueba. Aquí sin embargo se conoce que hay un muro al oeste. Si se garantiza que el robot está mirando al Oeste, el siguiente método es válido.

```
void Retornar()
{       if (frontIsClear())
        {       move();
                Retornar() ;
        }
}
```

La programación con recursión es muy potente y algunas veces está sujeta a error.

3.2. AMPLIACIÓN A LA RECURSIÓN

Dado el problema, un robot tiene que limpiar todos los pitos de cualquier esquina, se puede fácilmente escribir un proceso repetitivo como aparece a continuación:

```
class Aspiradora: Robot
{
  void LimpiaEsquina();
}
  void  LimpiaEsquina()
            {           while ( nextToABeeper() )
                        {           pickBeeper();
                        }
            }
            ...
}
```

Esto soluciona correctamente el problema y se conoce como solución iterativa ("iterativa" significa una repetición de algún tipo, while or loop, explicada en el capitulo anterior). Contrasta la anterior solución con la siguiente solución recursiva.

```
void LimpiaEsquina()
{           if ( nextToABeeper() )
            {           pickBeeper();
                        LimpiaEsquina();
            }
}
```

La diferencia entre estos dos métodos es muy sutil. El primer método, que usa el repetitivo while, es llamado una vez y nunca el foco del robot sale del proceso repetitivo hasta que haya finalizado, recogiendo cero o más pitos (depende del número inicial en la esquina).¿Qué pasa en la función recursiva, LimpiaEsquina()? Se analizará cuidadosamente.

Si el robot está inicialmente en una esquina vacía cuando el mensaje es enviado, no pasa nada (similar al repetitivo con while). Si es una esquina con un pito cuando el mensaje es enviado, La condición del if es verdadera, así que el robot ejecuta una instrucción pickBeeper() (vacía la esquina). Entonces hace un segundo llamado de LimpiaEsquina(), teniendo en cuenta donde fue la primera llamada. Cuando LimpiaEsquina() es llamado la segunda vez, la condición de if es falsa, no ocurre nada y el robot retorna al primer llamado.

[Invocación Inicial]

```
void LimpiaEsquina()
{       if ( nextToABeeper() )
        {       pickBeeper();
                LimpiaEsquina(); // <--  Este es la segunda llamada de LimpiaEsquina.
        }                                       /. El robot retorna a este punto cuando
}                                               // el llamado finaliza la ejecución.
```

Segunda Invocación

```
void LimpiaEsquina()
{       if ( nextToABeeper() )          // <-- Esto es ahora falso
        {       pickBeeper();
                LimpiaEsquina();
        }
}
```

Cada llamado resulta en invocaciones separadas (Ejemplarizacion) de la instrucción LimpiaEsquina(). El robot puede completamente ejecutar cada ejemplo, siempre recordando dónde fue el ejemplo previo, así que retorna allí cuando finaliza.

El proceso para escribir instrucciones del robot recursivas, es muy similar a escribir procesos repetitivos.

Fase 1: Consideremos la condición de parada (Llamado también caso base)-- ¿Cuál es el caso más simple en que el problema puede ser solucionado? En el problema de LimpiarEsquina(), el caso más simple, o base, es el caso cuando ya está la esquina vacía.

Fase 2: ¿Qué tiene que hacer el robot en el caso base? En este ejemplo no tiene qué hacer nada.

Fase 3: Encontrar la forma de solucionar una parte sencilla del problema completo que no es el caso base. Esto es llamado "reducción del problema en el caso general ". En el problema de LimpiaEsquina(), El caso general es cuando el robot está en la esquina con uno o más pitos y la reducción es recoger un pito.

Fase 4: Estar seguro que la reducción guía es caso base. Una vez más en el ejemplo anterior del LimpiaEsquina(), por elegir un pito a la vez, el robot puede eventualmente limpiar la esquina de pitos, independiente del número originalmente presente.

Comparación y contraste entre iteración y recursión.

Un proceso repetitivo iterativo completa cada iteración antes de comenzar la siguiente.

Un método tipo recursivo comienza una nueva instancia antes de completar el que rige en el momento. En el momento que ocurre la instancia actual es temporalmente suspendida, en espera de completar una nueva instancia.

Desde luego, está nueva instancia puede no completarse antes de generar propiamente otra. Cada instancia sucesiva tiene que ser completa y a su vez la última debe ser la primera instancia.

Aunque cada instancia recursiva se supone hace algo (frecuentemente mínimo) hacia adelante en el caso base, no se usan repeticiones para controlar los

llamados recursivos. Así, generalmente se ve un if o un if/else en el cuerpo del nuevo método recursivo, pero no un while.

Se propone utilizar recursión para mover un robot denominado Karel hacia un pito. ¿Cómo puede hacerse esto? Sigamos las fases presentadas anteriormente:

¿Cuál es el caso base?. Karel está sobre el pito.
¿Qué tiene que hacer el Robot en el caso base? nada.
¿Cuál es el caso general? El robot no está sobre el pito.
¿Cuál es la reducción? Moverse hacia el pito y hacer un llamado recursivo.
¿La reducción conduce hacia la terminación? Sí, se asume que el pito está directamente enfrente del robot, la distancia es mínima de un bloque en cada llamada recursiva.

La implementación final será:

```
void EncuentraPito()
{       if ( ! nextToABeeper() )
        {       move();
                EncuentraPito();
        }
}
```

Note que este problema puede también ser solucionado fácilmente, usando la instrucción repetitiva while. Analicemos un problema que no es fácil de solucionar, usando la instrucción repetitiva while. Consideremos una mina de pitos perdidos, ¿La esquina con un número grande de pitos? Imagínese escribir el siguiente método en una búsqueda por la mina. Un robot llamado karel camina hacia el oriente, desde una localización actual hasta encontrar los pitos. La mina de pitos perdidos está justamente al norte de la intersección, a una distancia igual al número de movimientos que karel hace para llegar a una posición actual de un pito. Se escribirá el nuevo método EncontrarMina().

No es fácil solucionar el problema con la instrucción repetitiva porque no tenemos alguna forma apropiada de recordar cuántas intersecciones han sido recorridas. Puede aparecer probablemente un esquema de rastreo de pitos muy complejo, se analizará una solución recursiva bastante directa. Una vez más las respuestas a las preguntas.

¿Cuál es el caso base? Karel está sobre el pito.
¿Qué tiene que hacer Karel en el caso base? TurnLeft() (Ahora karel estará orientado hacia el Norte).
¿Cuál es el caso general? Karel no está sobre un pito.
¿Cuál es la reducción? Mover hacia adelante un bloque, haciendo un llamado recursivo y karel ejecuta un segundo movimiento después de un llamado recursivo. Este segundo movimiento será ejecutado en todas las instancias, pero en el caso base, motivará que karel haga muchos movimientos hacia el norte, después del caso base hasta llegar nuevamente a obtener el caso base.

¿La reducción produce la terminación? Si, se asume que el pito está directamente enfrente de Karel.

El método completo queda así:

```
void  EncontrarMina()
{           if ( nextToABeeper() )
            {           turnLeft();
            }
            else
            {           move();
                        EncontrarMina();
                        move();
            }
}
```

¿Cuántos turnLeft() son ejecutados? ¿Cuántos movimientos? ¿Cuántos llamados a EncuentraMina()?

Una forma de pensar en una solución recursiva y sus respectivos llamados, en términos de la especificación del propio método se explicrá a continuación: En este caso, la especificación es que cuando un robot ejecuta está instrucción caminará determinado números de etapas, hasta encontrar un pito. Supongase k etapas, giro a la izquierda y caminará k etapas adicionales. Adicionalmente se supone que inicialmente el robot viaja N etapas hasta llegar al pito (k = N). Cuando examinamos el método anterior, la cláusula else tiene primero un mensaje de **move()**. Esto significa que el robot está ahora a N-1 pasos del pito, por consiguiente para la especificación, el llamado recursivo (k = N-1) camina N-1 etapas propicia a la izquierda (turnLeft()), y camina N-1 paso más allá. Por consiguiente, se necesita proveer un mensaje de movimiento adicional (**move()**) después la recursión completa los N pasos.

Se tratará de solucionar un número de problemas y un buen paso de mirar la recursión que viene a ser un uso confortable de la iteración. Gran parte es debido a que la recursión requiere alguna intuición para ver la reducción correctamente, especialmente en problemas difíciles. Está inducción viene con la práctica, porque los problemas sencillos están diseñados en forma fácil.

3.3. RECURSIÓN DE COLA Y REPETICIÓN

Supóngase que se tiene un proceso repetitivo con while y deseamos reescribir el proceso no repetitivo, pero que haga la misma tarea. Consideremos el caso más sencillo, supóngase que el mecanismo de control más extremo en un método repetitivo. Tomemos como ejemplo el siguiente método:

```
void EncuentraPito()
{           while ( ! nextToABeeper())
            {           move();
            }
```

```
}
```

Por definición de la instrucción while lo anterior es lo mismo que:

```
void EncuentraPito()
{       if ( ! nextToABeeper())
        {       move();
                while ( ! nextToABeeper())
                {       move();          } EncuentraPito( )
                }
        }
}
```

Sin embargo, la instrucción while anidada en la última forma es exactamente el cuerpo de EncuentraPito() usando la definición inicial de EncuentraPito(), así que reescribiendo la segunda forma quedará:

```
void EncuentraPito()
{       if ( ! nextToABeeper())
        {       move();
                EncuentraPito();
        }
}
```

Así, que la primera forma, un while, es equivalente a la última forma en un programa recursivo. También notemos que se puede exactamente transformar el segundo en el primero, dado que son instrucciones equivalentes.

Se percibe finalmente que está es una forma especial de recursión, aunque después del paso de la recursión (EncuentraPito() anidado) no hay nada más que hacer en este método y por consiguiente hay que observar que las instrucciones del while son las mismas a las instrucciones del if. Está forma de recursión es llamada recursión de cola, puesto que el paso recursivo viene propiamente de la cola de la computadora.

Es posible probar formalmente que la recursión de cola es equivalente a un proceso repetitivo con while. Por lo tanto, se puede ver que un proceso repetitivo con while es exactamente una forma especial de recursión. Así nada puede hacer con un proceso repetitivo, usted, puede hacer un programa recursivo. Existen algunas teorías profundas en ciencia computacional (en matemática) que han sido desarrolladas para está observación especialmente en los lenguajes funcionales tales como CAML[1].

Existe una forma de contraste en el método de solución de EncuentraMina(), discutida anteriormente y que ciertamente fue recursiva, pero no recursiva de cola, porque el mensaje final **move()** antecede un mensaje recursivo.

3.4. BÚSQUEDA

Está sección introduce dos nuevos métodos llamados zigIzqUp() and zagDerDown(), que mueve un robot diagonalmente al noroeste y sureste respectivamente. Ambos de estos métodos están definidos usando sólo los métodos de ur_robot() como turnLeft(), pero se deriva un inmenso potencial conceptual con capacidad para pensar en términos de moverlo diagonalmente. Por razones que no llegan a ser claras en el ejercicio, la clase dentro de la cual colocamos estos métodos es matemático. Tenemos la clase robot como la clase padre.

La siguiente definición introduce clave de está sección zigIzqUp() and zagDerDown(). Este par de direcciones no son arbitrarias; si un robot se mueve a la izquierda y hacia arriba bastante tiempo, esto eventualmente alcanza el límite de la pared occidental. El mismo argumento posee el viaje hacia abajo y hacia la derecha, excepto que en este caso el robot eventualmente busca el límite de la pared sur.

[1] *www.inria.fr*

Las otras dos posibles direcciones carecen de propiedades útiles, un robot nunca encuentra paredes límites al viajar arriba y hacia la derecha, y no está seguro a cuál de las dos paredes límites llega primero, cuando viaja hacia abajo y a la izquierda.

El siguiente método define zigIzqUp() and zagDerDown().

```
class Matemático: Robot
{
void zigIzqUp( );
void zagDerDown( );

}
void Matemático:: zigIzqUp ( )
{           // Precondición: Orientado al oeste y el frente despejado
// Postcondición: Orientado al oeste
                        move( );
                        turnRight( );
                        move( );
                        turnLeft( );
            }

void Matemático:: zagDerDown( )
{           // Precondición: Orientado al sur y el frente despejado
// Postcondición: Orientado al sur
                        move( );
                        turnLeft( );
                        move( );
                        turnRight( );
            }
            ...
}
```

Se asume que se tiene un karel llamado Matemático. Obsérvese que no se parte de esos métodos de Heurísticos para moverse en una dirección dada. Para ejecutar zigIzqUp() correctamente, karel debe estar mirando al oeste; para ejecutar zagDerDown() correctamente, Karel tiene que estar mirando al sur.

Estos requisitos son llamados las precondiciones de los métodos. Se recuerda que una precondición de un método tiene que ser verdadera antes que el robot pueda correctamente ejecutar el método.

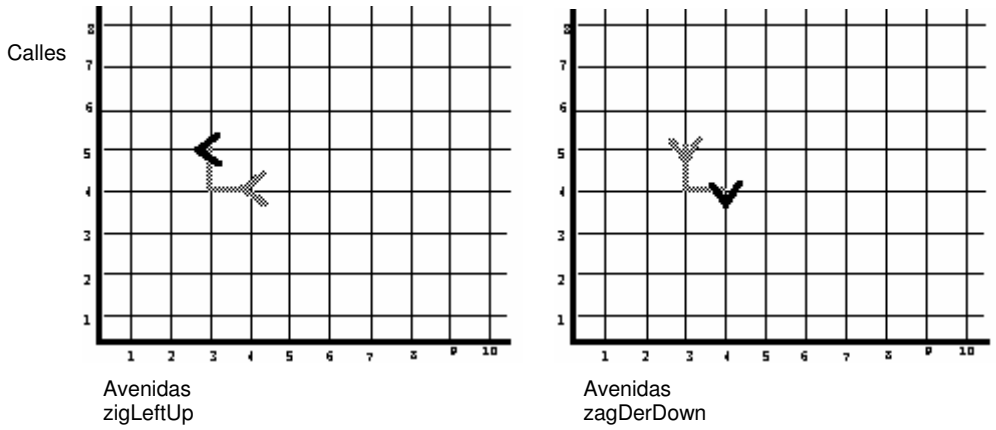

Figura 61. Precondicones para los Métodos ZipLeftUp() y zagDerDown()

Para este ejemplo la precondición direccional de zigIzqUp() es que karel esté mirando al Oeste; igualmente la precondición de zagDerDown() es que karel esté mirando al sur. Karel ejecuta estos métodos, cuando su precondición es satisfecha, como se ilustra en la figura 61.

Aquí hay una instrucción que es llena de terminología: La precondición direccional de zigIzqUp() y zagDerDown() es un invariante de cada instrucción ejecutada. Esto exactamente significa que si karel está mirando al oeste y ejecuta zigIzqUp(), el robot está aún mirando al oeste después que la instrucción ha terminado de ejecutarse. Está propiedad permite a karel ejecutar una secuencia de zigIzqUp() sin tener que restablecer su precondición direccional. Una instrucción similar mantiene a karel casi mirando al este para zagDerDown(). También observe que cada instrucción tiene que ser ejecutada sólo cuando karel encuentra el frente despejado. Está precondición no es un invariante de la instrucción, porque Karel tiene que estar a un bloque de la esquina donde el frente está bloqueado (e.g., Karel puede ejecutar zigIzqUp() mientras mira al oeste en la calle 4 y avenida 2).

El primer método principal que se escribe soluciona el problema de encontrar un pito que puede ser localizado en cualquier parte del micromundo, la tarea es escribir un método llamado EncuentraPito() en la que karel se posiciona en la misma esquina del pito. Existe una versión de este problema donde tanto karel como el pito están encerrados en un cuarto. Está nueva formulación tiene menos restricciones rigurosas. El pito está localizado en alguna esquina de la calle del micromundo de karel, y no hay muros en el micromundo. Por supuesto, los muros limitantes están siempre presentes.

Una sencilla solución puede emanar de su mente. En este ensayo, karel inicialmente está en el origen y mirando al Este. El robot entonces se mueve hacia el este por la calle primera ubicando un pito. Si karel encuentra un pito en la calle 1º, él ha culminado la tarea; si el pito no es encontrado en la calle 1º, karel se mueve para atrás hacia la pared del oeste, conmutando sobre la calle 2º, y continúa buscando a partir de allí. Karel repite está estrategia hasta encontrar el pito. Desafortunadamente existe una confusión que está implícita en la instrucción de búsqueda. No hay forma para que karel conozca que un pito no está en la calle 1º. No tiene importancia cuánto explora karel en la calle 1º, el robot nunca puede estar seguro que un pito no está más de un bloque hacia el este.

La percepción es como si karel y nosotros estuviéramos atrapados en una trampa imposible, pero existe una solución ingeniosa de nuestro problema.

Se puede prever la implicación de movimientos de zig-zag. Se necesita programar a karel para ejecutar radicalmente un tipo diferente de patrón de búsqueda; el procedimientos EncuentraPito() muestra un patrón de búsqueda.

```
void  EncuentraPito( )
{           VayalOrigen( );
            MirandoOeste( );
            while (  ! nextToABeeper( ) )
            {           if ( facingWest( ) )
                        {           zigMove( );
                        }
```

```
                    else
                    {       zagMove( );
                    }
            }
}
```

Este método de búsqueda aumenta la frontera de la búsqueda, similar a la forma como el agua puede fluir sobre el mundo de karel como si fuera una fuente desbordante en el origen. A grandes rasgos se puede ver a karel viajar de acá para allá diagonalmente en el margen de está ola de agua. Se debe estar seguro que es un patrón de búsqueda garantizado para encontrar un pito eventualmente, independiente de las localizaciones de los pitos en nuestra analogía, se necesita convencernos que el pito eventualmente se encuentra húmedo (ver figura 62). Se puede utilizar refinamiento paso a paso para escribir el método EncuentraPito() utilizando la estrategia de búsqueda con los métodos zigIzqUp() and zagDerDown().

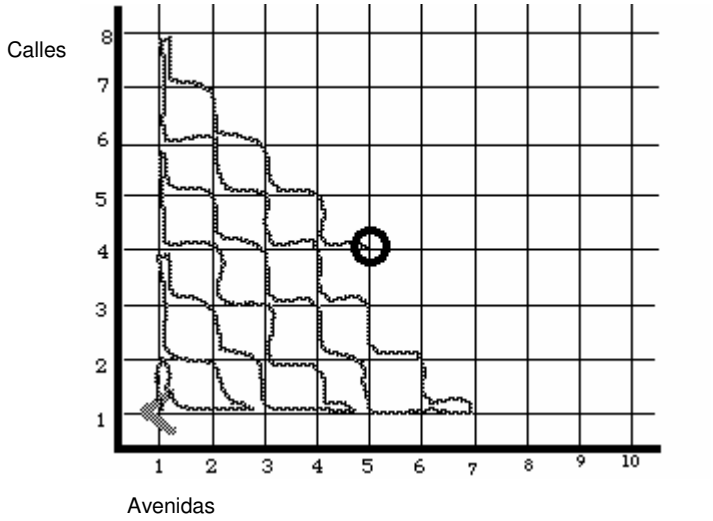

Figura 62. Aumento de la frontera de la búsqueda

El método EncuentraPito() comienza moviendo a karel al origen y mirando al oeste. (Decimos cómo escribir la instrucción mirando al oeste). Estos mensajes establecen la precondición direccional para zigIzqUp(). El ciclo while propuesto propone mover a Karel hasta encontrar un pito, y es correcto si el ciclo repetitivo eventualmente termina. La condición if, el cual es anidado en el cuerpo del ciclo, determinando en qué dirección karel ha girado y continúa

moviéndose a lo largo de la diagonal en la misma dirección. Se continúa refinando paso a paso, escribiendo los métodos zigMove() y zagMove().

```
void  zigMove()
{           // Precondición: Orientado al Oeste
            if ( frontIsClear( ) )
            {           zigIzqUp( );
            }
            else
            {           avanceProxDiagonal( );
            }
}
```

```
void  zagMove( )
{           // Precondición: Orientado al sur
            if ( frontIsClear( ) )
            {           zagDerDown( );
            }
            else
            {           avanceProxDiagonal( );
            }
}
```

Los métodos de desplazamiento zigMove() y zagMove() operan similarmente; sin embargo, se discutirá sólo zigMove(). El método zigMove() mueble robot diagonalmente hasta próxima esquina. De otra forma, el robot es bloqueado por el muro límite occidental y puede avanzar hacia el norte hasta la próxima diagonal. Ahora se escribirá el método que avanza karel hasta la próxima diagonal.

```
void  avanceProxDiagonal( )
 {          if ( facingWest( ) )
             {          facingNorth( );
             }
            else
             {          facingEast( );
```

```
            }
            move( );
            turnAround( );
 }
```

El método avanceProxDiagonal() comienza con Karel orientado hacia fuera del origen; girando en una dirección diferente, dependiendo tanto del robot como del zig o zag. En ambos casos, Karel se mueve en una esquina en forma más rápida del origen y gira al rededor. Si karel ha hecho zig en la diagonal actual, después de ejecutar avanceProxDiagonal(), el robot es posicionado para continuar con zag en la próxima diagonal, y viceversa.

Obsérvese que cuando karel ejecuta un método de zigIzqUp() o un método de zagDerDown(), tiene que visitar dos esquinas; la primera es visitada temporalmente, y la segunda es una esquina diagonal a la esquina del inicio de karel. Cuando se piensa acerca de estos métodos, se debe ignorar la esquina intermedia y exactamente recordar que está instrucción mueve diagonalmente a karel. También se advierte que una visita temporal a la esquina es garantía de no tener un pito, porque es parte de una oleada frontal que karel visita mientras estuvo pasando por la anterior diagonal.

La ejecución de EncuentraPito() por Karel, ilustra en la situación presentada anteriormente y lo conecta con la operación. Intente obtener una retribución de cómo todas estás instrucciones, se ajustan en conjunto para acoplar la tarea particularmente y así estará cerrada la atención al método avanceProxDiagonal(). Implemente EncuentraPito() en la situación donde el pito está en el origen y la situación donde el pito está próximo al límite del muro.

3.6 HACIENDO ARITMÉTICA

Una de las cosas que los computadores hacen bien es manipular los números. Los robots pueden estar enseñando aritmética se notará a continuación: Una forma de representar números en el mundo del robot es el uso de los pitos. Se

puede representar el número 32 colocando 32 pitos en una esquina, pero puede presentarse más sofisticado. Supóngase que se representan los diferentes dígitos de un número multidígitos separadamente. Consecuentemente para representar el número 5132 se puede utilizar la 2º calle como un ejemplo de un lugar para colocar el número, se colocan 5 pitos en la calle 2º , avenida 1º, 1 pito en la 2º con 2º , 3 pitos en la 2º con 3º , y 2 pitos en la 2º con 4º. Se puede escribir en la columna completa el número como se ilustra a continuación:

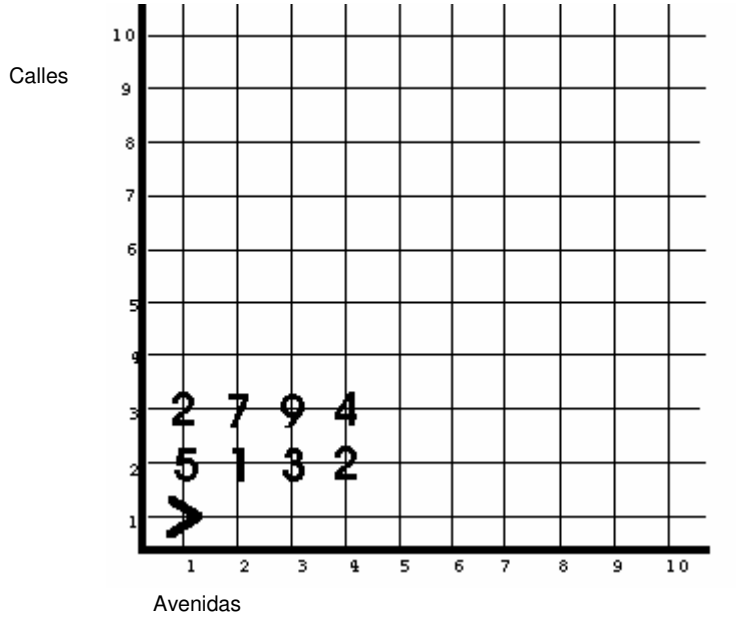

Figura 63 Abstracción de sumar en el mundo de Karel

Se comienza con un simple caso, y exactamente se adiciona una columna de números de simples dígitos. Veamos la figura siguiente como un ejemplo. Supóngase que iniciamos con un robot sumador en la calle 1º con una columna de números representados por pitos al norte de este. En cada esquina habrá entre 1 y 9 pitos. Esperamos escribir en la calle 1º el número decimal que representa la suma de la columna. Entonces se puede "llevar" si el número total de pitos es más de 9 se asume que no inicia en la avenida 1º.

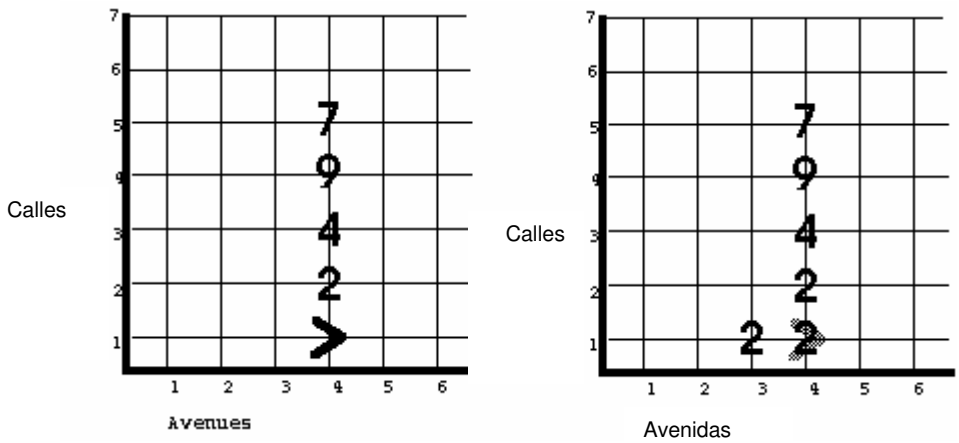

Figura 64 Invariante del problema sumar.

Nuestro robot sumador utiliza dos robots asistentes. El primero de estos chequea para ver si una lleva es necesaria y el segundo actualmente hace el lleva si es necesario. Estos robots necesitan ser creados y encontrar el robot sumador que los crea a ellos, se iniciará con una super clase(padre) que provee este método de encuentro. De todas las otras clases para este problema se derivan subclases de está clase base; Descubridor. Actualmente no creamos cualquier robot descubridor, sin embargo está clase es exactamente un método de lugar apropiado que puede ser común a otras clases.

```
class Finder : Robot
{
        void turnAround( ){...}
        void turnRight( ){...}

        void moveToRobot()   // Robot directamente  a la cabeza.
        {       while( ! nextToaRobot() )
                {       move();
                }
        }
        ...
}
```

```
class Carrier extends Finder
{
        void carryOne(){...}   // lleva"uno" para la próxima columna
}

class Checker extends Finder
{
        boolean enoughToCarry(){...}   // Requiere carrear significativamente
        . . .
}

class Adder extends Finder   // Siempre crea en la calle 1 mirando al norte
{
        Carrier Carry = new Carrier(1,1, East, infinity);
        Checker Check = new Checker(1,1, East, 0);

        void gatherHelpers(){...}
        void addColumn(){...}
        . . .
}

void gatherHelpers() // Adder class
{       Carry.findRobot( );
        Carry.turnLeft( );
        Check.findRobot( );
        Check.turnLeft();
}
```

Una vez que el robot sumador ha creado sus asistentes, se puede ejecutar el procedimiento addColumn(). Hacer esto es simplemente recoger todos los pitos nortes hasta encontrar una esquina vacía, retorna a la calle 1º, depositar todos los pitos allí y luego los asistentes terminarán la tarea.

```
void addColumn( ) // Orientado al  norte sobre la primera calle
{       move( );
        while(nextToABeeper( )
        {       pickBeeper( );
                if ( ! nextToABeeper( ) )
```

```
                    {       move( );
                    }
            }
            turnAround( );
            while( frontIsClear( ) )
            {       move( );
            }
            turnAround( );
            while ( anyBeepersInBeeperBag( ) )
            {       putbeeper( );
            }
            // Computa el cociente y el residuo.
            while( Check.enoughToCarry( )
            {       Carry.carryOne( );
            }
}
```

El acarreador es considerablemente simple. Advierte que administrar al acarreador, un número infinito de pitos en la bolsa así que no es posible que salga corriendo.

```
void carryOne( )  // Facing north on 1st Street. -- Carrier class
{       turnLeft( );
        move( );    // Nota:  Error: intento de salirse si trata de acarrear.
                            // desde la calle 1
        putBeeper( );
        turnAround( );
        move( );
        turnLeft( );
}
```

El robot verificador hace todo el trabajo interesante. Se tiene que determinar si hay 10 o más pitos en la esquina actual. Si hay, retorna verdadero, de otra manera falso. Se puede tantear para recoger 10 pitos para chequear esto. Sin embargo, se tiene que hacer más desde tener que llamar repetitivamente para ver cuántos múltiplos de 10 realmente hay. Sin embargo, se tiene que encontrar una forma de disponer de cada grupo de 1 pito antes de intentar para comprobar el próximo grupo de 10. Otra importante tarea adicional es encontrar al menos 10 pitos en un grupo que puedan salir de una esquina actual. Esto es llevar la cuenta para el primer dígito de la respuesta; una que no se acarrea.

```
boolean enoughToCarry( ) // Orientado al Norte sobre la primera calle. – Clase Checker
{       loop(10)
        {       if (nextToABeeper( ) )
                {       pickBeeper( );
                }
                else
                {       emptyBag( );
                        return false;
                }
        }
        // Se tienen que encontrar 10 pitos.
        move( );
        emptyBag( ); // lSalirse en la calle 2.
        turnAround( );
        move( );
        turnAround( );
        return true;
}
```

Para finalizar se necesita sólo adicionar el procedimiento emptyBag() en la clase Checker :

```
void emptyBag( )
{       while( anyBeepersInBeeperBag( ) )
        {       putBeeper( );
        }
}
```

Ahora estamos listos para afrontar el problema de la suma multicolumna. Observe que si se inicia a la derecha al final del valor de fila y exactamente en la trasparencia del sumador y de sus asistentes a la izquierda, después de adicionar una columna, los tres robots se posesionan en la próxima columna. Entonces se trabaja de derecha izquierda, computando la suma correcta porque acarreamos en la columna antes de que está columna sea sumada.

Se necesitan dos métodos adicionales en la clase sumador: slideLeft() y addAll( ). El método slideLeft( ) es fácil.

```
void slideLeft( )

{       turnLeft( );
```

```
        move( );
        turnright( );
        carrier.turnLeft( );
        carrier.move( );
        carrier.turnRight( );
        checker.turnLeft( );
        checker.move( );
        checker.turnRight( );
}
```

¿Cómo addAll() conoce cuando se hace adición de columnas? Una forma es tener el número más a la izquierda iniciando él la segunda avenida, así que hay un salón para llevar cualquier valor de la columna más a la izquierda. El sumador entonces necesita comprobar para ver si en la segunda avenida antes de la adición. Esto requiere un nuevo predicado.

```
boolean onSecondAve( )  // Precondición:  Orientado al norte y no sobre la primera avenida.
{       turnLeft( );
        move( );
        if (frontIsClear)
        {       turnAround( );
                move( );
                turnLeft( );
                return False;
        }
        turnAround( );
        move( );
        turnLeft( );
        return True;
}
```

Se está ahora listo para escribir el método addAll() en la clase sumador.

```
void addAll( )
{       while( ! onSecondAve( ) )
        {       addColumn( );
                slideLeft( );
        }
```

```
        addColumn( );
}
```

3.7 POLIMORFISMO- ¿POR QUÉ ESCRIBIR MUCHOS PROGRAMAS CUANDO HAY QUE HACER UNO?

Polimorfismo significa literalmente en griego "muchas formas". En programación orientada objetos se refiere al hecho de que los mensajes enviados por objetos (robots) pueden ser interpretados diferente, y dependiendo de la clase de objeto (robot) que recibe el mensaje. La mejor forma de pensar en esto es recordar que el robot es autónomo en el micromundo. Se envía un mensaje y éste responde. Esto no llega a ser una unidad de control remoto para que el usuario directamente accione. Preferentemente, éste "Escucha" el mensaje enviado y éste responde de acuerdo con un diccionario interno. Recuerde que cada robot consulta su propio diccionario de instrucciones para seleccionar el método que utiliza para responder cualquier mensaje. La nueva versión de cualquier método, entonces, cambia el significado del mensaje para robots de la nueva clase.

Para ilustrar la consecuencia de esto, nos apropiamos de algo dramático, sin embargo, no es un ejemplo útil. Supóngase que tenemos las siguientes dos clases:

```
class Putter : Robot
{
        void move( )
        {       super.move( );
                if (anyBeepersInBeeperBag())
                {       putBeeper( );
                }
        }
}

class Getter : Robot
{
```

```
        void move( )
        {       super.move( );
                while (nextToABeeper( ))
                {       pickBeeper( );
                }
        }
}
```

Ambas clases sustituyen el método move(). De está manera el robot Putter pone pitos en las esquinas en que se mueve y el robot Getter recoge por las que se mueve. Supóngase ahora que utiliza estás dos clases en la siguiente tarea:

```
task
{       Putter  Lisa = new Putter(1, 1, East, infinity);
        Getter  Tony = new Getter(2, 1, East, 0);

        loop (10)
        {       Lisa.move( );
        }
        loop (10)
        {       Tony.move( );
        }
}
```

Si el micromundo contiene un pito en cada una de las primeras diez esquinas a lo largo de la calle 1º y también cada una de las primeras diez esquinas de la 2º calle entonces cuando la tarea es realizada habrá 2 pitos en cada uno de los primeros 10 bloques a lo largo de la 1º calle, y ninguno de los correspondientes bloques de la calle 2º. No hay nada sorpresivo acerca de esto, pero note que tanto Lisa y Tony responden al mismo mensaje en idénticas (relativas) situaciones.

El significado de polimorfismo es siempre profundo para esto, pero sin embargo. En efecto, los nombres que se utilicen para referirse al robot no están "Lanzamiento anterior" para que los robots propiamente, pero sólo por comodidad del usuario. Un robot dado puede ser referido por diferentes nombres, llamados alias.

Primero, se declara el nombre que se usa como alias. No se hace referencia a un nuevo robot aún.

Robot Karel; // "Karel" Se refiere a algún robot.

En segundo lugar, se requiere "asignar" un valor al nombre karel. En otras palabras se necesita especificar que el robot de nombre "Karel" es al que nos referimos. Se hace esto con una instrucción de asignación. Esto asume que el nombre Tony ya se refiere a un robot como si ya fuera parte del citado bloque de tarea principal.

Karel = Tony;

Esto establece a karel como un nombre alternativo(alias) para el robot también conocido como Tony. Se puede exactamente acomodarse al nombre de "Karel" refiriéndose al robot Lisa. Es muy importante notar, sin embargo, que un alias no se refiere a cualquier robot hasta que asignemos un valor al nombre.

Entonces enviando un mensaje de turnLeft() usando el nombre de karel envía un mensaje al mismo robot que es nombrado como Tony aludiéndolo, aunque ellos son el mismo robot.

Karel.turnLeft( );

Asumamos ahora que se considera la tarea revisada lentamente a continuación: Se usa la misma estructura y que el micromundo es como antes. La única diferencia es que nos referimos al robot usando el nombre de Karel en cada caso. Note que mientras tenemos tres nombres aquí, sólo tenemos dos robots.

```
task
{       Robot  Karel;
        Putter Lisa = new Putter(1, 1, East, 100);
        Getter Tony = new Getter(2, 1, East, 0);
```

```
        Karel = Lisa;                   // "Karel" refiere a  Lisa;
        loop (10)
        {       Karel.move( );          // Lisa pone abajo 10 pitos.
        }
        Karel = Tony;                   // "Karel" refiere a  Tony
        loop (10)
        {       Karel.move( );          // Tony sweeps ten blocks
        }
}
```

Así que no sólo puede tener mensajes idénticos (move()) refiriéndose a diferentes acciones; aún si la instrucción del mensaje es idéntica Karel.move(). Se pueden tener diferentes acciones. Advierta sin embargo, que, aún se están enviando mensajes de los dos diferentes robots, y que estos robots son de diferentes clases. Es aún posible formar diferentes cosas que sucedan en diferentes ejecuciones de la misma instrucción.

Consideremos lo siguiente:

```
task
{       Robot  Karel;
        Putter Lisa(1, 1, East, 100);
        Getter Tony(2, 1, East, 0);

        loop (10)
        {       Karel = Lisa;                   // "Karel" refiere a Lisa;
                loop (2)
                {       Karel.move( );          // ??
                        Karel = Tony;           // "Karel" refiere a Tony

                }
        }
}
```

Note que el mensaje move() es enviado 20 veces, pero 10 veces es enviado a Lisa, y 10 veces a Tony, alternando nuevamente. La calle 1º alcanza pitos extras y la calle 2º los atrapa rápidamente.

3.8 CONCLUSIÓN

Finalmente, se espera hacer la siguiente pregunta: ¿Cuándo es apropiado diseñar una nueva clase y cuándo se modificará o adicionará una clase de robot existente?.

Una completa respuesta a la pregunta está más allá del alcance de este documento, porque hay muchas cosas que deben considerarse en la decisión, pero se pueden dar algunas directrices aquí si en su juicio una clase de robot contiene errores u omisiones, para que todos entiendan las modificaciones. Aquí las omisiones significan que hay algún método (acciones o predicados) que es necesario completar la funcionalidad básica de la clase o hacer que el robot lo haga en la clase o en la clase que fue diseñado.

De otra manera, si tenemos una clase útil, y necesitamos una funcionalidad adicional, especialmente más funcional que lo previsto por la clase donde ya se tiene construida una nueva clase como una subclase proporcionada apropiadamente. De tal forma, cuando se necesita un robot con capacidades originales puede usarse una clase original y cuando se necesita una nueva funcionalidad puede usarse una nueva. Algunas veces la escogencia es hecha porque encontramos que muchos de los métodos de algunas clases son la elección, es hecha porque encontramos que muchos de los métodos de alguna clase son exactamente las que esperamos, pero uno o dos métodos pueden ser más útiles en el nuevo problema si ellos son modificados o ampliados.

3.9 PROBLEMAS PROPUESTOS

Los siguientes problemas usan métodos de recursión, búsqueda y aritmética discutidos en el capítulo. Algunos de los problemas a continuación usan combinación de los métodos zigIzqUp() y zagDerDown(), o variantes sencillos de estos. Cada problema es difícil de solucionar pero una vez que el plan es descubierto (a través probablemente de un "aha de experiencia"), el programa

que implementa la solución no será asimismo difícil de escribir. Se puede asumir también que no hay sección de muros en el micromundo.

Finalmente podrá asumirse que el robot, karel comienza sin pitos en su bolsa a menos que se diga otra cosa. No hacer cualquier suposición acerca de la esquina de inicio de karel u orientación dada, a menos que se especifique en el problema.

1. Karel se ha graduado en alfombrar por etapas. Karel tiene que alfombrar un salón completamente encerrado sólo un pito puede ser localizado en cada esquina. El salón puede ser de cualquier tamaño y de cualquier forma. La figura muestra un posible piso. El área gris no debe ser alfombrada por karel. Karel puede comenzar en cualquier lugar del salón y estar en cualquier orientación.

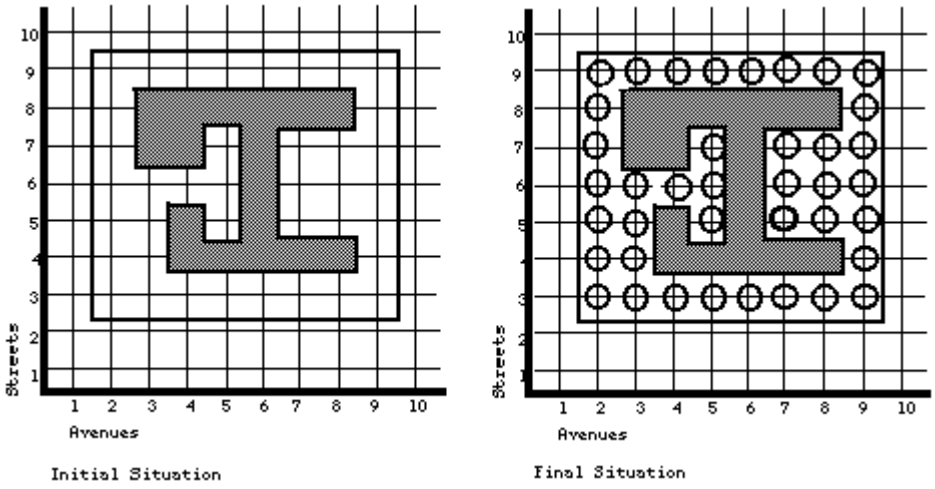

Figura 65 Estado inicial y final del problema “Alfombrar”

2. Reescriba tanto zigIzqUp() y zagDerDown() así que automáticamente satisfagan las precondiciones de orientación.

3. Asuma que hay un pito en la calle 1º y avenida Nº. Programe a karel para encontrarlo y llevarlo a la calle Nº y avenida 1º.

4. Asuma que hay un pito en la calle Sº y avenida Aº y que karel tiene dos pitos en la bolsa. Programe a karel para poner los pitos de la bolsa en la calle 1º y avenida Aº y calle Sº y avenida 1º.

El pito original tiene que quedarse en la esquina en que comenzó.

5. Si usted disfruta de la ciencia computacional y eventualmente ha tomado un curso en teoría de la computación, gratamente evoque los días que usted, gastó programando karel y trate de solucionar el siguiente problema, pruebe que karel con la ayuda de algunos pitos es equivalente a la máquina de Turing. Hint. Use la equivalencia entre la máquina de Turing y un autómata 2-contador.