SISTEMA EXPERTO EN

LA SOLUCION DE TANGRAMAS

Francisco Javier Noriega Romero

Universidad Nacional Autónoma de México

Tesis de Maestría en Ciencias de la Computación

Dra. Guillermina Yankelevich, directora

Dra. Hanna Oktaba, coasesora

México, 1991

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

INTRODUCCION 1

I. SMALLTALK: UN LENGUAJE VISUAL 5

1. Mecanismos de abstracción de control 12

1.1 Control a nivel de unidades 13
1.2 Recursividad 14
1.3 Control a nivel de instrucción 15
1.4 Iteración sobre las colecciones 17
1.5 El método VALUE de la clase BLOCK 18
1.6 Conclusión 18

2. Mecanismos de abstracción de datos 20

2.1 Tipos y Algebras 22
2.2 Búsquedas glotonas 23
2.3 Polimorfismo 26
2.4 Un APUNTE sobre APUNTADORES 28

3. Mecanismos de abstracción por encapsulación .. 29

3.1 Los objetos 30
3.2 El problema de la herencia 30
3.3 Conclusión 31

4. Mecanismos de abstracción en arquitecturas de tipo paralelo 33

4.1 La programación concurrente 33
4.2 Paralelismo en Smalltalk. El proyecto Mushroom 37
4.3 Objetos Activos 40
4.4 Conclusión 41

5. Conclusión del Capítulo 42

II. ESPECIFICACION FORMAL DEL JUEGO 43

1. Presentación del juego 45

2. Especificación Formal 51

Diagramas 52
Especificación de Tangramas 56
Juego Exhaustivo 61

3. Comentario sobre la Especificación 69

Especificación del Juego 78

4. Justificación del Método Propuesto para la Determinación del Polígono Resultante 83

4.1 Terminología 83
4.2 Definiciones 85
4.3 Método 87

5. Conclusión del capítulo 94

III. IMPLEMENTACION 96

1. Quetzalc nuevo comenzamos 98

2. La clase Tangrama 101

3. La clase Pieza 104

4. Otras clases del simulador 106

5. Conclusión del capítulo 107

IV. UN HIPOTETICO EXPERTO ARTIFICIAL 108

V. CONCLUSION GENERAL 113

BIBLIOGRAFIA 117
APENDICE A: LISTADOS DEL PROGRAMA 124
APENDICE B: ESPECIFICACION ALGEBRAICA 173

INTRODUCCION

Un hombre corre, tropieza y arroja al suelo un mosaico de arcilla que se parte en siete tanes, de donde surgió el tangrama que utilizaría para describir al mundo, según cuenta la leyenda:

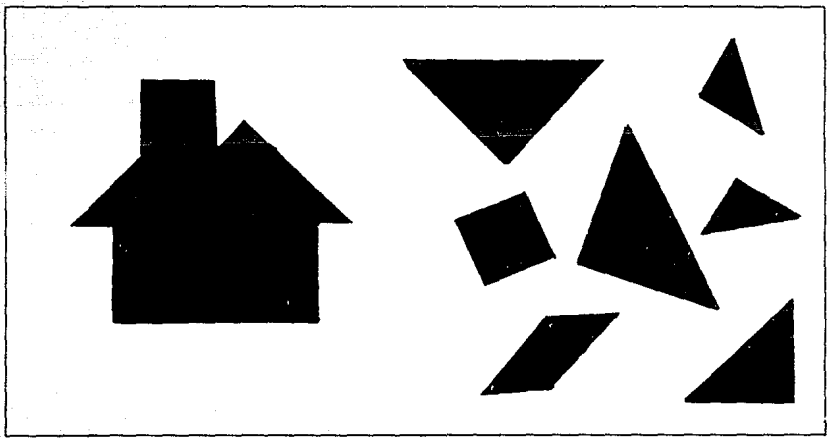


Figura 1: Un ejemplo de tangrama y los siete tanes del juego.

El objetivo general de esta tesis consiste en realizar una aproximación al estudio de los lenguajes visuales desde las Ciencias de la Computación y desde la Inteligencia Artificial, presentando un ejercicio sobre el juego de tangramas.

La programación orientada a objetos funge como eje central y su análisis, también es un objetivo particular del proyecto.

Pretendemos mostrar a través de la programación de un simulador del juego de tangramas, la aplicación de los conceptos teóricos relativos a este tipo de programación, y con ello, verificar su utilidad como una metodología para el desarrollo de "software".

La especificación algebraica del tangrama, de ser factible, permitiría ver la estructura formal del juego.

Como último objetivo planteamos el esquema de trabajo para implementar un sistema experto, cuya programación íntegra supera el alcance de esta tesis.

Yankelevich[1] propone que el tangrama es un material idóneo para el estudio de la abstracción intelectual. El tangrama es una escritura con imágenes y la investigación de Yankelevich es pilar de nuestro estudio.

Arnheim[2], en el año de sesenta y nueve publica un libro, ya clásico, titulado: "Visual Thinking" donde analiza diferentes aspectos cognitivos de la percepción visual y destina un capítulo a la abstracción intelectual.

Shi-Kuo Chang[3] edita el trabajo de más de treinta autores, sobre lenguajes visuales y programación visual publicado en el año de noventa. El editor trabaja en el laboratorio de computación visual de la universidad de Pittsburg.

Hégron[4] publica en ochenta y cinco su libro sobre síntesis de imágenes. Las operaciones sobre polígonos convexos forman parte del conjunto de algoritmos del libro. Weiler[5] publica en ochenta un artículo sobre representaciones gráficas de polígonos cóncavos con ciclos múltiples.

Oktaba[6], de México, presenta las especificaciones algebraicas de tipos de datos básicos y estructurados. Mallgren[7] publica un libro sobre especificaciones algebraicas de gráficos.

Negrete[8], de México, propone que el tangrama puede ser entendido como una demostración analógica de teoremas geométricos simples.

Deutsch y Hayes[9] en el año de setenta y dos plantean una solución heurística al juego de tangramas. Se podrían seguir dos aproximaciones para abordar el problema, dicen los autores: la primera, es una solución combinatoria donde se ensaya empatando las siete piezas del tangrama hasta lograr llegar a la solución; la segunda, proviene de la programación heurística y consiste en probar particiones tentativas sobre un contorno dado como problema.

Las dos posturas planteadas en el excelente artículo de Deutsch y Hayes se podrían refrasear como una aproximación sintética y otra analítica. "Resuelven" el tangrama en la segunda forma, trazando líneas de extensión sobre sus vértices y estableciendo algunas heurísticas para converger más rápido en el análisis de la figura.

Elegimos la otra cara de la moneda para intentar la construcción del simulador. Entender al juego como un problema de síntesis de imágenes es ver al tangrama como un lenguaje visual.

En el primer capítulo, una excerpta sobre la programación orientada a objetos, discutimos algunos aspectos conceptuales del lenguaje de programación SmallTalk. Los lenguajes visuales de programación proveen de nuevos elementos para el tratamiento de los lenguajes icónicos.

Enfocamos nuestro análisis desde tres perspectivas complementarias: desde los lenguajes de programación, desde la Inteligencia Artificial, y planteamos algunos indicios para un tratamiento formal de los lenguajes icónicos.

Durante el segundo capítulo presentamos la especificación formal, algebraica, del juego.

La implementación en Smalltalk de un programa de juegos que resuelve tangramas de tres piezas es analizada en el tercer capítulo.

El cuarto es una propuesta somera acerca de la construcción del sistema experto. Listamos clases y métodos de nuestra aplicación y los hemos incluido como apéndice.

Tres años de trabajo se conjugan en esta síntesis, que ponemos a consideración del lector y de nuestro honorable jurado.

I. SMALLTALK: UN LENGUAJE VISUAL

Todo y más se ha dicho ya acerca de la programación orientada a objetos (POO) y también de Smalltalk*. Goldberg(10) lo ha descrito como un lenguaje visual enfatizando el valor de su interfaz gráfica; sin embargo, estamos convencidos de que esta idea tiene un trasfondo que lo lleva ser un lenguaje útil en la programación de modelos conceptuales sobre algunos aspectos de la percepción visual. Se pretende en este capítulo expresar claramente las razones fundamento de tal afirmación.

La POO surge, al igual que todo el conocimiento científico, coherente con los resultados del trabajo en diferentes disciplinas. Para introducir la excerpta se exponen algunos de los conceptos relacionados con dicha filosofía de programación, manifiestos en diferentes ramas de las Ciencias de la Computación y en la Inteligencia Artificial (IA). Establecer un punto de comparación con Simula -primer lenguaje orientado a objetos- que es el progenitor de Smalltalk, posiblemente permita delinear los conceptos más importantes de este paradigma.

La crisis de "software" con que bautiza Dijkstra(11) a la situación ocasionada por el surgimiento de computadoras poderosas construidas con componentes electrónicos miniaturizados que posibilitaban al programador la implantación de ideas con las que "...antes ni siquiera habría soñado "; es la cuna donde nace Simula y más tarde Smalltalk.

Los trabajos de Polya(12) publicados en 1945, sobre heurísticas en resolución de problemas matemáticos fueron una "primera inspiración" para el estudio de la analogía, según Hall. El punto central en esta heurística es utilizar lo que ya se conoce. Algunas de las preguntas que se hacen son: ¿haz visto este problema antes?, ¿haz visto este problema en alguna forma ligeramente distinta?

---

*Nos referiremos a SMALLTALK V como SMALLTALK, durante todo el ensayo, a menos que se indique lo contrario. SMALLTALK V es una versión de SMALLTALK que corre en computadoras personales.

Esta idea, donde la solución de un problema se puede pensar en términos de la solución de otro similar al que se complementa o añaden ciertas propiedades, nos refiere al concepto de HERENCIA -no sólo de propiedades, sino también de heurísticas o métodos (procedimental)- que es uno de los conceptos fundamentales de la POO. Está implícito el concepto de clases subsumidas en un árbol jerárquico: clases superiores de problemas generales, subdivididos en problemas particulares que ocupan niveles inferiores de clasificación.

Uno de los primeros genes de la POO, aún importante dentro de los lenguajes de programación, es la subrrutina. Un objeto se puede entender como un refinamiento de este concepto: no es una abstracción de programa (procedimental) únicamente; a diferencia de la subrrutina, posee un estado, un conjunto de propiedades que tienen una persistencia mayor que las variables de las subrrutinas. Programas y datos se encuentran en una relación cercana: encapsulados.

Hacia los inicios de la década de los setentas, se hablaba de una programación sin "GOTO´S" y en general de evitar las instrucciones dinámicas no acotadas. La metodología de programación consistía en dividir el problema inicial en subproblemas elementales y la solución se encontraría mediante la superposición de las soluciones parciales del problema. La POO es una programación estructurada, pero no sólo en su algorítmica, también se estructuran los atributos del problema.

La siguiente cita[13] compara los módulos de programa con una forma más amplia de encapsulación que exige una programación diferente a efectos de lograr mayor expresividad en las aplicaciones: "la encapsulación se relacionaba en el pasado con la descomposición de un problema y la interferencia mínima entre módulos de programa. Ahora, si se pretende explotar la reusabilidad de los objetos, el polimorfismo, la herencia entre clases, los tipos parametrizados, etc;

se deben organizar las aplicaciones de manera que sean consistentes con estas posibilidades. Se han desarrollado técnicas de descomposición de problemas en componentes relativamente independientes, pero aún carecemos de una metodología para ver las aplicaciones en términos de objetos". Resultan atractivos estos nuevos recursos durante el diseño y la implantación de programas que sean utilizados como modelos conceptuales del intelecto.

La ENCAPSULACION, además de implicar una relación estrecha entre programas y datos, refleja la propiedad -por lo menos en SmallTalk- de que el estado del objeto no es accesible ni puede ser alterado ("HIDING"), sino sólo a través de MENSAJES (llamadas) a METODOS (procedimientos) propios del objeto. Facilita una programación en la que un objeto se ve como un módulo que puede ser reemplazado por otro equivalente sin que esto afecte la programación de los otros módulos. En este sentido, la POO es afín a la intención de la programación funcional y lógica (declarativa) de evitar los efectos laterales, con vistas a lograr una programación "demostrable".

Se han reportado[14] intentos de programación de módulos que no sean dependientes de un tipo de datos particular, sino que sean genéricos, o reutilizables para un conjunto determinado de tipos de datos. La parametrización de tipos es una característica importante de la POO, íntimamente relacionada con la ENCAPSULACION.

Uno de los objetivos del grupo de investigación de Xerox en Palo Alto al desarrollar Smalltalk '80 era brindar al programador todo el "poder" de la computación, un lenguaje de programación que sólo requiriera conocimientos elementales de computación para poder utilizarlo[15].

Este interés es la respuesta a un sentimiento de la época que expresa claramente la siguiente cita[16]: "A medida que ocurre el desarrollo cada vez más intenso del hombre con la computadora, la relación dolorosa de la programación con unos y ceros se transforma en menos dolorosa conforme aparecen los superlenguajes generando la aspiración a una relación más humana con las máquinas nuevas; se hace deseable que las máquinas "piensen" (bajo la implícita esperanza de que lo puedan hacer)"

SmallTalk, sin embargo, no logra totalmente su objetivo. Efectivamente, es una programación "más humana", más intuitiva; pero no es un lenguaje dirigido a usuarios finales. La gran cantidad de objetos que conforman su máquina virtual obligan al programador a tener que memorizar mucha información. Si a esto se agrega que fue un lenguaje diseñado como intérprete y de muy alto nivel, el resultado es que los programas en Smalltalk no corren ... "caminan" (lo cual no descarta su gran valor en la programación de prototipos).

Minsky y Pappert[17] en su teoría de estructuras proponen que la inteligencia es el resultado de una sociedad de agentes organizados jerárquicamente con funciones sencillas y que se comunican entre ellos sin la necesidad de tener un lenguaje común.

Un sistema orientado a objetos se ve como una sociedad de este tipo, pero sí existe -al menos en Smalltalk- un lenguaje común entre ellos. Los lenguajes con objetos activos eliminarían la restricción de envío de mensajes[18].

Los marcos de Minsky[19] son una herramienta para la representación del conocimiento, el trabajo citado se refiere específicamente a la representación de escenas visuales. Son estructuras jerárquicas en donde un marco tiene un conjunto de atributos.

Este interés es la respuesta a un sentimiento de la época que expresa claramente la siguiente cita[16] : "A medida que ocurre el desarrollo cada vez más intenso del hombre con la computadora, la relación dolorosa de la programación con unos y ceros se transforma en menos dolorosa conforme aparecen los superlenguajes generando la aspiración a una relación más humana con las máquinas nuevas; se hace deseable que las máquinas "piensen" (bajo la implícita esperanza de que lo puedan hacer)"

SmallTalk, sin embargo, no logra totalmente su objetivo. Efectivamente, es una programación "más humana", más intuitiva; pero no es un lenguaje dirigido a usuarios finales. La gran cantidad de objetos que conforman su máquina virtual obligan al programador a tener que memorizar mucha información. Si a esto se agrega que fue un lenguaje diseñado como intérprete y de muy alto nivel, el resultado es que los programas en Smalltalk no corren ... "caminan" (lo cual no descarta su gran valor en la programación de prototipos).

Minsky y Pappert[17] en su teoría de estructuras proponen que la inteligencia es el resultado de una sociedad de agentes organizados jerárquicamente con funciones sencillas y que se comunican entre ellos sin la necesidad de tener un lenguaje común.

Un sistema orientado a objetos se ve como una sociedad de este tipo, pero sí existe -al menos en Smalltalk- un lenguaje común basado en mensajes entre ellos. Los lenguajes con objetos activos eliminarían la restricción de envío de mensajes[18] .

Los marcos de Minsky[19] son una herramienta para la representación del conocimiento, el trabajo citado se refiere específicamente a la representación de escenas visuales. Son estructuras jerárquicas en donde un marco tiene un conjunto de atributos.

Cada marco está relacionado con otros sub-marcos de tal forma que el concepto de CLASE está incluido dentro de esta representación.El contenido procedimental es mínimo, ésta es una de las razones por las cuales no son una programación orientada a objetos. No obstante, son un material de gran interés para la IA, y son también un análogo a la POO.

Bic y Gilbert comentan el trabajo en bases de datos: "Smith y Smith introducen dos conceptos importantes en el desarrollo de modelos de base de datos: agregación y generalización (1977), el primero permite que una relación entre dos objetos sea vista como un objeto agregado; al mismo tiempo que las propiedades de los objetos individuales se ignoren, implicando que la agregación es una forma de abstracción. Por ejemplo, 'empleado' podría verse como la agregación de dos objetos de menor nivel tales como nombre, dirección, salario y así sucesivamente.

El segundo concepto -generalización- apunta al modelo de objetos ordenados jerárquicamente que constituyen una empresa. Esta es una abstracción que permite que una clase de objetos sean vistos como un objeto típico o genérico de esta clase. Permite que atributos con un valor común para todos los miembros de la clase sean grabados en el objeto genérico, en vez de ser copiados repetidamente en los niveles inferiores. Por ejemplo, el hecho de que todas las secretarias saben escribir a máquina, podría ser guardado en el objeto genérico secretaria y heredado por cada individuo que pertenezca a esta clase. El concepto de generalización no distingue entre los atributos heredables por los individuos y aquellos que se aplican al conjunto como un todo"(28) .

Ambas abstracciones aparecen continuamente en la POO. En Smalltalk, por ejemplo, la agregación se programa utilizando las variables de instancia. Los atributos que cada objeto de la clase debe poseer son variables de instancia que se definen en la declaración de la clase, estas variables son a su vez otros objetos.

Las variables de clase permiten declarar propiedades genéricas a todos los objetos que pertenezcan a ella, tampoco distingue entre los atributos heredables y aquellos que se aplican al conjunto como un todo. Los mecanismos de herencia a través de las variables, dan mayor expresividad a estos conceptos. Recuérdese que no se heredan propiedades exclusivamente, también los métodos son heredables.

Hasta aquí hemos descrito algunas de las aportaciones a las Ciencias de la Computación y a la IA, paralelas al desarrollo de la POO. A continuación se pretende resumir estos conceptos presentando a Simula, paráfrasis de Oktaba[21] .

Simula 62, 64 y 67 fueron desarrollados en Noruega con el fin de simular sistemas reales; un interés importante a finales de los cuarentas y durante la década de los cincuentas.

La construcción de un modelo para la simulación se realizaba mediante un conjunto de procesos concurrentes que cooperaban entre sí (Smalltalk tiene muy pocos elementos para el manejo de la concurrencia).

Simula utiliza los conceptos de clase, objeto, variables de clase (que tienen un significado diferente a las variables de clase en Smalltalk), asociación de objetos a variables, atributos de una clase, acceso a los atributos (existe un modo protegido al correr las aplicaciones que permite la encapsulación), subclases (herencia), procedimientos virtuales (POLIMORFISMO ad hoc, este concepto es muy importante dentro de la programación orientada a objetos y se detallará más adelante).

En la definición de las clases se pueden incluir parámetros formales (en Smalltalk no) instanciados durante la llamada al procedimiento "new", que permite la creación de un objeto nuevo (el método "new" también existe en Smalltalk).

En Smalltalk, una clase siempre es subclase de otra y tiene una sola superclase (no hay herencia múltiple). Toda la programación en Smalltalk se realiza dentro de un árbol de clases cuya raíz es la clase OBJECT.

Las variables de clase son apuntadores a los objetos de una clase en Simula. La asignación entre dos cualesquiera variables de apuntador ocasiona que ambas apunten al mismo objeto.

Los objetos, al igual que en Smalltalk, se crean dinámicamente y preservan sus estados.Los procedimientos virtuales (métodos en Smalltalk) se pueden redefinir en las subclases ("overloading").

A continuación presentamos la clasificación propuesta por Wegner[22] para distinguir los diferentes tipos de lenguajes, que usualmente se confunden:

"Un lenguaje basado en objetos es un lenguaje basado en clases o clásico si cualquier objeto debe pertenecer a una clase. Un lenguaje basado en clases es un lenguaje orientado a objetos si un mecanismo de herencia puede definir gradualmente jerarquías de clases."

"Mi gato es orientado a objetos" es el título que Roger King[23] da a su artículo en el cual describe cómo el término orientado a objetos se ha sobreutilizado:"Tengo un gato que se llama Trash. En el clima político actual, si yo quisiera venderlo (al menos a un científico en computación), no debería enfatizar que es amable con las personas y autosuficiente, que pasa la vida cazando ratones. Mejor debería argumentar que es orientado a objetos...lo siento, Trash. No se puede dar una definición exacta de modelaje orientado a objetos...así que los modelos semánticos y mi gato tendrán que encontrar otro camino hacia la fama".

1. MECANISMOS DE ABSTRACCION DE CONTROL

En el área de lenguajes de programación suele utilizarse el término de abstracción para denotar al hecho de haber despojado a un objeto, o concepto, de los elementos que no son significativos, sin los cuales preserva su estructura (abstracción sintética).

Los lenguajes de programación tradicionalmente se estudian y comparan mediante dos formas de abstracción: abstracción de tipos, que está relacionada a los datos del programa; y abstracción de control, que se refiere a la secuencia de ejecución de uno o más programas. Por supuesto que este análisis es factible para máquinas de arquitectura Von Newman, donde existe el concepto de programa y por tanto el de lenguaje de programación; pierden sentido en otras arquitecturas, por ejemplo en redes neuronales.

Las operaciones que la Máquina de Turing puede realizar son de lectura, escritura, movimiento de la cabeza y brinco condicional. Las tres primeras son operaciones mas bien relacionadas con el tratamiento de datos y el acceso a memoria; la última provee los elementos básicos de control.

Ghezzi[24] señala que los dos mecanismos fundamentales de control de flujo a nivel de instrucción son secuenciación y ramificación. Distingue el control a nivel de instrucción, del control a nivel de unidades.

Cuando el programador cuenta con estos dos únicos medios (ramificación y secuenciación) escribir un programa constituye una tarea bastante compleja aun en el caso de simples rutinas.

Desde los primeros microprocesadores se incluyó en su conjunto de instrucciones el brinco a subrrutina para el control de unidades. La secuencia normal se interrumpe temporalmente durante la ejecución, y se reanuda después de ejecutar cierto grupo de instrucciones espacialmente separadas del programa.

Estos tres elementos se han combinado para formar estructuras de mayor nivel que faciliten el trabajo del programador. El "GOTO" es una de las primeras instrucciones de control que aparecen en los lenguajes de programación. Dijkstra[25] criticó severamente su uso puesto que produce un distanciamiento entre el texto del programa y su ejecución. El "GOTO" hace de los programas un revoltijo ("mess") según el autor. Otros han defendido su uso controlado. Prosigue la contienda.

FORTRAN presenta la estructura DO...CONTINUE para la programación de ciclos. En ALGOL y LISP se pueden programar ciclos con condiciones de salida (de tipo WHILE), y condicionales estructurados (de tipo IF...THEN...ELSE).

A nivel de programa, FORTRAN introduce la subrrutina; LISP , las llamadas recursivas de funciones como estructura básica de control; SIMULA, el concepto de corrutinas, que se ejecutan simultáneamente alternándose el control, en máquinas que poseen un solo procesador.

1.1 CONTROL A NIVEL DE UNIDADES

El método es la unidad operativa de Smalltalk. La programación de un sistema se realiza dentro de un árbol de clases y subclases. Consiste en definir ramas nuevas del árbol y enriquecer las preexistentes, agregando métodos y redefiniendo clases.

Cada clase tiene asociado un conjunto de métodos y viceversa. El control de programas se lleva a cabo a través de mensajes, que se envían diferentes objetos y que activan a los métodos correspondientes.

Todos los objetos en Smalltalk son pasivos puesto que su única forma de activación es mediante un mensaje que se envía al objeto, o a la clase. Cuando los mensajes se envían a un objeto, los que se ejecutan se llaman métodos de instancia; cuando se envían a la clase, se llaman métodos de clase. Los últimos se utilizan poco, generalmente sirven para crear e inicializar objetos nuevos, instancias de la clase.

La herencia permite ejecutar un método definido en una clase superior si no está definido en la clase del receptor.

La estructura sintáctica de los métodos consta de una primera línea que es el nombre propio del método; una línea (opcional) para declaración de variables temporales, que existen sólo durante el tiempo en que está activo el método; líneas de programación: ya sea un mensaje que se envía a un objeto receptor, o bien una instrucción para la asignación de un objeto a una variable; una o varias líneas de retorno de control, que siempre regresan un objeto (^ es el símbolo que se utiliza como retorno). Más adelante se presentan algunos ejemplos.

Como ya se había mencionado, un sistema orientado a objetos es una sociedad de objetos organizados jerárquicamente que se comunican entre sí a través de mensajes.

1.2 RECURSIVIDAD

Bajo el esquema anterior, las llamadas recursivas de funciones o procedimientos no existen. La recursividad como mecanismo de control, consiste en mandar un mensaje -que activa al método que se define- a un objeto de la misma clase.

La variable self es utilizada, dentro de un método, para referirse al objeto receptor. Es posible distinguir dos acepciones de la recursividad ligeramente distintas: los métodos que envían su propio mensaje selector al objeto self; y los que lo envían a cualquier otro objeto de la clase. Un ejemplo del segundo caso es el método factorial, en la clase de los enteros positivos:

```
factorial
  (self = 1) ifTrue:  [ ^1]
             ifFalse: [  ^(self - 1) factorial *
                self]
```

Otra variable especial para el control de envío de mensajes es super*. LaLonde y Pugh[26] explican el significado de super: sirve para referirse al receptor del método, cuando se utiliza super, la búsqueda del método seleccionado comienza en la superclase (el padre en el árbol de clases) de la que contiene la definición del método. Lo cual no es siempre lo mismo que comenzar la búsqueda en la superclase de self. Supongamos, por ejemplo, la siguiente jerarquía de clases: Poligono es padre de Cuadrilatero, y Cuadrilatero es padre de Rectangulo; además, el método perimetro está definido en la clase Poligono. Al enviar el mensaje perimetro a la variable especial super, dentro de un método que se define en Rectangulo, la búsqueda de perimetro no se inicia en el padre de Rectangulo (ergo: Cuadrilatero) sino en el padre de Poligono

Si bien, la recursividad no juega un rol primario como lo hace en LISP o PROLOG; sí ocupa un lugar importante en esta forma de programación.

1.3 CONTROL A NIVEL DE INSTRUCCION.

Para conservar la propiedad de que cada sentencia es un mensaje que se envía a un objeto o clase, la sintaxis de las estructuras de control adquiere un matiz distinto al de la mayor parte de los lenguajes de programación. El siguiente ejemplo se refiere al cálculo del perímetro de un polígono:

---

* Se evita el uso de acentos en las palabras reservadas, métodos, clases y variables, puesto que no son permitidas por el compilador.

```
perimetro
    | suma, i |
    suma := 0.
    i := 1.
    [i > self ultimaArista] whileFalse:
        [suma := suma + (self aristaNum: i) longitud.
        i := i + 1].
^suma
```

Los objetos de la clase Block son secuencias de instrucciones que se encierran entre paréntesis cuadrados (similar a la sintaxis de LOGO).

Para interpretar el mensaje whileFalse: anterior, en primer lugar se distingue que el receptor es un bloque; se evalúa el bloque [i> ultimaArista]; si el resultado es el objeto true, regresa el control;si el resultado fue el objeto false, entonces se evalúa el bloque que está a la derecha del mensaje whileFalse: ([suma:= ...]), y se envía recursivamente el mensaje whileFalse: al bloque receptor [i> ultimaArista](27) .

La sintaxis y la semántica del mensaje "IF" es muy similar y se ilustra con una versión recursiva del método que calcula el perímetro:

```
perimetro: i
  (i > self ultimaArista)
    ifTrue:  [^0]
    ifFalse: [^(self aristaNum: i) longitud + (self
perimetro: (i+1)) ].
```

La estructura equivalente al DO...CONTINUE de FORTRAN (FOR en PASCAL) es el mensaje que se envía a la clase de los enteros:

```
perimetro
  |suma|
  suma := 0.
  1 to: self ultimaArista by: 1 do:
     [:i | suma := suma + (self aristaNum: i)
     longitud ].
^suma
```

timesRepeat: (REPEAT en LOGO) realiza una iteración similar pero no incrementa automáticamente ninguna variable:

```
perimetro
    |suma i|
    suma := 0.
    i := 1.
    self ultimaArista timesRepeat:
        [suma := suma + (self aristaNum: i) longitud.
        i := i + 1].
```

1.4 ITERACION SOBRE LAS COLECCIONES

Collection es la clase abstracta (no tiene instancias) que reúne a todos las clases de grupos de objetos: arreglos, conjuntos, "streams", etc. Análogos a los tipos estructurados de otros lenguajes.

La función MAPCAR de LISP quizá halla sido la fuente de inspiración para el conjunto de mensajes que iteran sobre las colecciones de Smalltalk.

Una versión parecida a MAPCAR es el método collect:, evalúa un bloque al que pasa como parámetro el primer elemento de la colección receptora, vuelve a evaluar el bloque con el segundo elemento de la colección, y así sucesivamente. Regresa la colección que se formó por el resultado de cada evaluación del bloque.

select: y reject:, seleccionan o rechazan los elementos de una colección siguiendo un criterio de decisión dado (todos los que cumplan con cierta condición).

Algunos de estos métodos se pueden programar en forma recursiva del segundo tipo que señalamos, donde el objeto receptor es el siguiente de la "lista", como métodos recursivos a la cola.

Existen una gama amplia de métodos que iteran sobre las colecciones en SmallTalk.

1.5 El METODO value DE LA CLASE Block.

Pensar en programas que escriben programas no es una idea nueva en computación. Sin embargo, sigue siendo una característica deseable de cualquier lenguaje de programación.

Las funciones EVAL y APPLY de LISP, proveen mecanismos para la selección dinámica de evaluación de funciones. EVAL recibe como parámetro una expresión simbólica y regresa el resultado de la EVALuación de dicha expresión. APPLY recibe como parámetros el nombre de una función y su lista de parámetros; y aplica la función.

En SmallTalk se pueden definir nuevos mecanismos de abstracción de control haciendo uso de un elemento análogo a la función EVAL : el método value de la clase Block.

Cuando se envía el mensaje value a un objeto de esta clase, el bloque envía cada uno de los mensajes y regresa como resultado el objeto producto de la última instrucción.

Cuando decíamos que "el bloque se evalúa", al describir la semántica del mensaje whileFalse:, nos referíamos al método value. Algunos autores[28] ya han descrito esta propiedad.

1.6 CONCLUSION

La diversidad de conceptos y su múltiple relación son un impedimento para el aprendizaje de este lenguaje.

Smalltalk incorpora elementos de muchos otros lenguajes de programación. Sin duda, hay una gran influencia de la programación LISP en las estructuras de control de flujo en SmallTalk.

La creación y ejecución dinámica de código, permiten la programación de métodos que realicen tareas generales, de mayor alcance.

Estas características son de gran valor en la programación de expertos artificiales. La programación de métodos generales inscrita en un árbol de clases, posibilita la creación de modelos para la abstracción intelectual de tipo sintético.

2. MECANISMOS DE ABSTRACCION DE DATOS

Los trabajos que existen sobre el tema forman un conjunto amplio que cubre desde sus aspectos formales hasta los más concretos de la programación.

Tipo en lenguajes de programación se podría entender como un conjunto de datos a los que se asocia un conjunto de operaciones.

Se han realizado en computación diferentes representaciones que van desde el bitio (dato elemental) y su agrupación en "bytes", hasta sofisticadas estructuras en los lenguajes de alto nivel.

Son ampliamente conocidos los llamados tipos básicos: enteros, caracteres, racionales, etc. El programador los utiliza como un modelo sin importar cuál sea su representación binaria en la computadora; en este sentido se dice que son abstracciones de datos.

Si la "definición" de tipo se re-escribe en términos de "... un conjunto de datos o tipos ..." entonces incluirá nuevos elementos de órdenes superiores. En un nivel inmediato al de los tipos básicos se encuentran los estructurados, por ejemplo: un arreglo de enteros, un registro, o un conjunto de caracteres.

Ciertos lenguajes cuentan con un número fijo de tipos (como FORTRAN); en otros, existen algunos predefinidos en el lenguaje y es posible definir nuevos tipos (como PASCAL).

LISP es un caso que merece especial atención. Existen sólo dos tipos en el lenguaje: los átomos o tipo básico; y las listas, el único tipo estructurado. Con un conjunto restringido de operaciones se construye todo el lenguaje. Esta simplicidad, asociada al carácter simbólico de sus tipos, hizo de LISP una herramienta idónea para el tratamiento de problemas que antes no habían sido resueltos en computadoras.

Para Cardelli y Wegner[29] promovedores del uso de tipos en lenguajes de programación, el tipo puede ser entendido como una vestimenta o armadura que protege a la representación sin tipos subyacente, de un uso inadecuado y arbitrario.

"Los lenguajes de programación en los cuales el tipo de cualquier expresión puede ser determinado mediante un análisis estático del programa se dice que es estáticamente tipificado. La tipificación estática es una propiedad útil, pero el requisito de que todas las variables y expresiones sean acotadas dentro de un tipo, desde el momento de la compilación, algunas veces es demasiado restrictivo. Podría ser reemplazado por el requisito más ligero de garantizar que todas las expresiones sean de tipos consistentes a pesar de que el tipo en sí mismo pueda ser estáticamente desconocido; esto generalmente se puede lograr introduciendo pruebas de tipos durante el tiempo de la ejecución. Los lenguajes en que todas las expresiones son de tipos consistentes se llaman lenguajes fuertemente tipificados ("strong typed"). Si un lenguaje es fuertemente tipificado, su compilador puede garantizar que el programa se ejecutará sin provocar errores de tipos".[30]

En la programación orientada a objetos, el mecanismo fundamental de abstracción de datos es la clase. Su definición en Smalltalk consiste en determinar los atributos de las entidades que pertenecen a ella, y las operaciones que estas entidades pueden realizar. La POO re-significa la noción de tipo bajo el nombre de clase, las diferencias que por esta razón ocurren son motivo de discusión en esta sección.

2.1 TIPOS Y ALGEBRAS

Deutsch[31] define tipo: "Un tipo es una caracterización precisa de las propiedades estructurales o conductuales que una colección de entidades (actual o potencial) comparten. Una instancia de un tipo es una entidad que posee las propiedades características del tipo".

Wegner[32] escribe que históricamente se introdujeron como una forma de clasificación de colecciones de datos por sus propiedades comunes.

Según Zilles[33] se debe recuperar esa noción de tipo. Un mismo elemento puede pertenecer a varios tipos, que son colecciones de datos capaces de realizar un conjunto dado de operaciones, la inclusión de subtipos dentro de tipos no debe estar preespecificada necesariamente. El autor define álgebra como un conjunto de tipos y sus conjuntos respectivos de operaciones cerradas con respecto a los tipos que pertenecen a dicha álgebra.

En este marco, continúa el autor, un operador genérico actúa sobre varias álgebras. Recibe un parámetro algebraico. Si se conoce la semántica abstracta de las operaciones requeridas, es factible probar la correctez del operador genérico, del cual se sabe, entonces, que funcionará apropiadamente para cualquier álgebra que satisfaga la semántica abstracta.

Por ejemplo, la operación de ordenamiento ("SORT") debe estar parametrizada por una operación que especifica cuál es la relación de orden que se ha de utilizar; además se debe indicar que el operador actúa sobre colecciones de objetos que tienen una operación de selección y otra de intercambio de lugar entre objetos, dentro de la estructura de la colección.

En Smalltalk se encuentra predefinido el "SORT". El método sort: to: de la clase SortedCollection, ordena una colección de objetos. La estructura de la colección es el receptor del método, las operaciones de selección (basicAt:) y actualización (basicAt: put:) son dos primitivos del lenguaje heredados de la clase Object. El intercambio se realiza dentro del método utilizando una variable temporal de paso. La única variable de instancia propia de la clase es del tipo bloque (sortBlock), mediante la cual se parametriza la relación de orden que debe haber entre los elementos de la colección receptora.

Un ejemplo similar al de Zilles, de un nivel de abstracción equivalente a "SORT", se podría llamar CLASIFICA. El parámetro algebraico lo forman una colección de objetos y un criterio de clasificación entre ellos, establecido como una relación entre uno o varios atributos de los objetos (p. ej. color y tono en una clasificación de imágenes). El resultado de CLASIFICAr es una colección bidimensional (clases versus objetos) donde cada objeto aparece dentro de las clases a que pertenezca.

Dos álgebras son equivalentes[34] si los tipos que las componen son equivalentes (i.e. los conjuntos son isomorfos) y existe una correspondencia entre operaciones, tal que ellas se comporten en la misma forma.

2.2 BUSQUEDAS GLOTONAS

Problemas cuya solución se encuentra dentro de un espacio cerrado de búsqueda, definido mediante un conjunto de estados posibles; han sido estudiados por la IA.

Algunos problemas de juegos utilizan además, funciones heurísticas que permiten podar el árbol definitorio del espacio de búsqueda.

En el campo de la Teoría de Algoritmos, se ha utilizado el término "glotón"[35] para agrupar a todos aquellos que resuelven problemas de optimización -problemas en los que alguna cantidad debe ser minimizada o maximizada- siguiendo una estrategia mediante la cual se eligen opciones que son localmente óptimas en cada paso, con la esperanza de encontrar un óptimo global. Esta técnica de diseño de algoritmos es comparable con otras como la recursión [36] : "Divide y Conquistarás". Utilizando la recursión se puede resolver un problema partiéndolo en instancias más simples de él mismo.

Los algoritmos para resolver problemas en un espacio cerrado parten de un estado inicial y su objetivo consiste en llegar a un estado meta o solución. El "costo" de alcanzar el estado solución, desde un estado cualquiera, generalmente es un parámetro cuantificable. Un glotón minimiza esta "distancia" cada vez que se mueve de un estado a otro.

¿Cómo formular un método glotón en Smalltalk?: combinando ambas técnicas (recursión y "glotonería") en un método genérico definido para la clase de los glotones (subclase de los problemas-en-espacios-cerrados).

La propiedad glotona del método que se propone es una función heurística (GLOTONA) cuyo resultado probablemente sea el estado máximo elegido entre un conjunto de estados posibles, conjunto parámetro de GLOTONA. Esta operación es similar a "SORT": está parametrizada por una relación de orden, entre estados en este caso.

El método que se expone a continuación realiza una búsqueda llamada "primero el mejor" ("best-first") que, según Rich[37] , fue planteada inicialmente por Hart en sesenta y ocho.

Las búsquedas en amplitud y profundidad son casos atípicos del método glotón, cuando la relación de orden está dada en función de la posición del estado en el árbol de búsqueda.

```
busca
 "Método que realiza una búsqueda
 glotona en un espacio cerrado"
 |ensayo|
 ensayo := self gloton.
 (ensayo = nil)
    ifTrue:  [^ 'no hay solución']
    ifFalse: [ensayo estaVisitado].
 (ensayo esSolucion)
    ifTrue:  [^ ensayo camino]
    ifFalse: [self saca: ensayo.
              self mete: (ensayo hijos).
              self busca ].
```

La colección de estados está encapsulada dentro del objeto self. El cual reconoce el mensaje de selección y control de búsqueda gloton; y los mensajes constructores de la colección de estados mete: y saca:

Existe otro grupo de mensajes enviados a los estados y no a la colección. El mensaje esSolucion prueba si el ensayo es el resultado esperado o no. El estaVisitado y ruta actualizan las colecciones visitados y camino, respectivamente. Estas colecciones son dos variables de instancia también encapsuladas en el objeto self. Negrete(38) en su discusión sobre el problema de caníbales y misioneros utiliza una lista para almacenar los estados visitados. El método hijos regresa una colección de estados que son los hijos sin visitar del estado actual.

El código es compacto y permite observar la estructura abstracta de la búsqueda propuesta, gracias a las cualidades de encapsualción y ocultamiento propias del paradigma, que permiten distanciar el código, de las estructuras de datos.

El resultado es un método genérico, polimorfo, aplicable a un conjunto amplio de problemas.

2.3 POLIMORFISMO

Las funciones polimórficas son aquellas que pueden ser evaluadas por -o que regresan valores de- diferentes tipos. El operador de división (/) en FORTRAN es uno de los ejemplos más simples de polimorfismo. Realiza una división entera cuando el divisor es entero; real, cuando el divisor es real (polimorfismo ad hoc).

Hablar de polimorfismo en un lenguaje sin tipos como Smalltalk resulta paradójico. Esto se debe a que la noción de tipo ha ido adquiriendo diferentes significados y el término "lenguaje sin tipos" ("type-free") se refiere a las pruebas de consistencia en el momento de la compilación, mas que a la posible existencia de alguna organización de los objetos.

En la POO los objetos se encuentran organizados jerárquicamente. Las clases reúnen a los objetos que poseen características descriptivas y operativas, comunes.

Cardelli y Wegner[39] afinan la clasificación de polimorfismo propuesta originalmente por Strachey en el año de sesenta y siete:

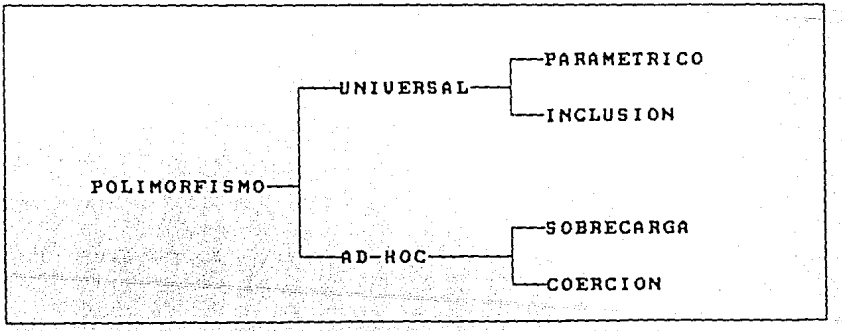


FIGURA 2: La clasificación de polimorfismo de Cardelli y Wegner.

El polimorfismo universal se identifica cuando las unidades de programa (métodos) tienen el mismo significado para cada elemento del conjunto de tipos válidos.

El polimorfismo ad-hoc, en cambio, se refiere a unidades de programa cuyo significado puede ser distinto para cada elemento del conjunto de tipos válidos. Por ejemplo, el significado del mensaje gloton en el algoritmo descrito, dependerá de cuál sea el objeto receptor, incluso podría no ser una función de maximización de una cantidad sino cualquier otro tipo de selección de estados, por ello decíamos que los algoritmos de búsqueda en profundidad y amplitud son casos ATIPICOS del glotón.

Existen dos tipos de polimorfismo universal: el paramétrico, ejemplificado en ML; y el polimorfismo por inclusión, propio de los lenguajes orientados a objetos.

En el polimorfismo paramétrico, el tipo es un parámetro de la función. El ejemplo de Wegner[40] se refiere a la función LENGTH: dada una lista, regresa el número de elementos que la componen. Al llamarla, se determina si es una lista de enteros, o caracteres, etc.

El polimorfismo por inclusión, también llamado polimorfismo por herencia, se da cuando un método es aplicable a todos los objetos de una clase y de todas las sub-clases, sub-sub-clases, etc. Por ejemplo, la fórmula de cálculo del perímetro de un polígono es válida tanto para cuadriláteros, como para un triángulo escaleno.

Aunque ambas clases se refieren a una especialización de tipos[41], ésta es diferente; una vez que se ha instanciado el tipo en la función paramétrica, no se puede re-instanciar.

Por el contrario, la herencia de métodos es multinivel.

El polimorfismo paramétrico actúa sobre un CONJUNTO de tipos; el de inclusión, sobre un ARBOL de clases.

Sobrecarga y coerción son los dos casos de polimorfismo ad-hoc o aparente. Coerción se da convirtiendo el tipo original, en el tipo requerido, antes de realizar la operación polimórfica. La división en FORTRAN se puede interpretar como una coerción de tipos.

Un ejemplo de sobrecarga es la doble significación que tiene el operador '+' en algunos lenguajes; tanto operador de concatenación entre cadenas, como operador de suma entre cantidades. Todos los métodos en SmallTalk se pueden sobrecargar (redefinir) en varias clases.

3.3 Un APUNTE sobre APUNTADORES

Uno de los mecanismos más primitivos y poderosos de abstracción de datos es el APUNTADOR. El direccionamiento indirecto es la herramienta que poseen la mayor parte de los microprocesadores para separar los datos de su representación interna.

En SmallTalk todas las referencias a los objetos se llevan a cabo por medio de apuntadores que el lenguaje controla implícitamente. La asignación es una igualación de apuntadores, las dos variables apuntan al mismo objeto. Una de las decisiones al diseñar SmallTalk fue elegir un microprocesador dotado con un grupo amplio de operaciones con modo de direccionamiento indirecto.

En otros lenguajes -C, por ejemplo- el manejo de apuntadores es totalmente explícito y el programador es responsable de utilizarlos en forma adecuada y flexible.

3. MECANISMOS DE ABSTRACCION POR ENCAPSULACION

Hemos analizado hasta este punto dos formas de abstracción, las abstracciones de control de flujo de programas y las abstracciones de datos. La interrelación entre ambos mecanismos origina una tercera forma: las abstracciones por encapsulación.

El concepto de encapsulación también ha cambiado desde los inicios de las Ciencias de la Computación. Para introducir esta sección, seguimos el enfoque de Oktaba (42) sobre la evolución de estas ideas.

El objetivo original consistió en formar cápsulas de programa para evitar la repetición de código (subrrutinas). Se concibió después como una forma estructurante de programas (funciones y procedimientos).

El paso de parámetros, siendo el medio de comunicación entre cápsulas, adquirió gran importancia en los años setentas. Se formularon hipótesis que apuntaban a las llamadas por nombre para ser las más utilizadas en un futuro, puesto que estas llamadas depositan el resultado de la función en algún parámetro actual.

Los módulos aparecen más tarde en lenguaje de programación como una forma de abstracción en el paradigma imperativo. Los módulos de MODULA y los paquetes de ADA son ejemplos representativos de esta forma de encapsulación. Los módulos permiten abstracciones de nivel superior a las abstracciones facilitadas por subprogramas o funciones. Dotan al programador de un nuevo elemento para el análisis de sistemas y han servido como un modelo programable en algunos estudios sobre especificaciones formales(43)

Los módulos (en MODULA) se construyen en dos unidades distintas: la primera es definitoria porque declara la interfaz de conexión con los otros módulos; la segunda contiene la programación misma ("module implementation").

Un tema de investigación en los últimos años versa acerca de la generación de lenguajes formales para describir la semántica de estas unidades; la definición de la interfaz no implica necesariamente explicitar la semántica de la cápsula. El módulo levanta una barrera entre "el qué" y "el cómo" tantas veces deseada por los investigadores.

Encapsular programas y datos es la tercera acepción que se da al término, en ella, a un conjunto de datos se asocia un conjunto de operaciones.

Las cápsulas parametrizadas con tipos (polimorfismo paramétrico) para la programación de unidades genéricas, también son una forma de encapsulación.

3.1 LOS OBJETOS

El objeto posee un estado y puede realizar cualquier método del conjunto de métodos asociado a la clase que pertenece. Encapsulación usualmente se refiere -en la POO- a la propiedad adicional de que el estado permanece oculto ("HIDING") y sólo es accesible mediante mensajes enviados al objeto mismo.

Simplificar el mantenimiento, y por ende, reducir la posibilidad de error, son una consecuencia directa de la encapsulación. Un método puede ser reemplazado por otro totalmente distinto, siempre y cuando conserve su estructura tanto en la llamada, como en el retorno.

3.2 EL PROBLEMA DE LA HERENCIA

Si bien, el concepto de objeto está estrechamente relacionado con el concepto de tipo de datos abstractos, existen algunas diferencias. En este punto analizaremos la relación entre encapsulación y herencia.

Alan Snyder[44] analiza el problema de la herencia de variables de instancia. Cuando se altera la definición de la clase desechando, por ejemplo, una variable de instancia, se provocan errores en métodos (definidos para las sub-clases) que utilizan dicha variable. La herencia de variables de instancia en SmallTalk viola la encapsulación de los objetos.

Otro problema que analiza el autor[45] se presenta cuando alguna operación heredada no puede ser ejecutada por alguna subclase, en estos casos debería existir algún mecanismo para excluir dichas operaciones. Desde nuestro punto de vista, esta exclusión sería también útil para las variables de instancia.

En lenguajes de herencia múltiple, el hecho de que una clase deje de ser subclase de otra, puede generar indirectamente problemas en las clases inferiores [46] .

La posibilidad de agregar métodos dinámicamente también puede ocasionar problemas.

3.3 CONCLUSION

Uno de los objetivos centrales de la encapsulación como mecanismo de abstracción, se presenta en la siguiente cita: "La repetitividad obscurece el objetivo de un programa y hace que el mantenimiento sea molesto. En vez de esto, se busca una forma para expresar similitudes y enfatizar las diferencias que hay entre dos segmentos de código, ahí surge la necesidad de abstracción"[47] .

El distanciamiento entre las estructuras de datos, y los programas que las usan; y la agrupación jérárquica de los objetos, pueden lograr el efecto contrario: des-encapsular la representación del conocimiento, de la máquina de inferencia. Característica mínima necesaria, según Negrete[48] , para la construcción de expertos artificiales:

"Quizá el postulado principal y más firme que podría encontrarse para generar una teoría de sistemas expertos es el de la separabilidad del conocimiento como representación, del mecanismo de inferencia que lo usa...

... el efecto de encapsular la experticia dentro de la representación y después diseminarla en esa forma ... es prácticamente torturador del conocimiento en manos de los así llamados ingenieros del conocimiento (inquisidores en este caso)".

4. MECANISMOS DE ABSTRACCION EN ARQUITECTURAS DE TIPO PARALELO

En las arquitecturas secuenciales los eventos ocurren uno tras otro, en una sola dimensión temporal. Por el contrario, en arquitecturas de tipo paralelo, dos o más procesos se ejecutan simultáneamente. Es por esta razón que el tiempo adquiere gran importancia en el paralelismo. Los estudios sobre este tema son vastos y constituyen una rama importante en las Ciencias de la Computación.

Uno de los primeros motivos para el desarrollo de estas arquitecturas fue el alto costo de los dispositivos electrónicos. Compartir recursos mejoraría significativamente la economía de los centros de cómputo.

Los eventos temporizados y los recursos compartidos son dos nuevas formas de abstracción que inducen a una tercera: el no-determinismo.

4.1 LA PROGRAMACION CONCURRENTE

Las corrutinas de SIMULA-67 fueron un antecedente para el surgimiento de los lenguajes concurrentes.

La instrucción de control "RESUME", desactiva la corrutina en ejecución y despierta a otra. El control se alterna, en un conjunto de corrutinas que se ejecutan concurrentemente en máquinas con un sólo procesador. El juego de cartas[49] es un ejemplo tradicional para visualizar el efecto de las corrutinas. Un jugador tira su baraja y pasa el control al siguiente, que realizará una tarea similar; el ciclo se repite hasta encontrar un ganador.

Probablemente la persistencia de los objetos en Simula facilitó la programación de este mecanismo de control en el lenguaje.

En el año de sesenta y ocho se presenta ALGOL un lenguaje concurrente que cuenta con instrucciones para la sincronización de procesos(50) .

Se avanzaba a la par en la formulación de algoritmos y su prueba formal para evitar el problema del bloqueo mutuo y garantizar la exclusión mutua de procesos en secciones críticas (porciones de programa que deben ser ejecutados por un sólo proceso a la vez).

Oktaba(51) presenta las soluciones de exclusión mutua con espera activa, la solución de Dijkstra-68 y la de Peterson-81. La validez de todas ellas está condicionada a la propiedad de "interlock", por la cual se puede afirmar que en cada momento se ejecuta una sola operación sobre la memoria compartida.

Oktaba describe el problema de la siguiente manera(52) :

"1. En cada momento sólo un proceso puede estar en la sección crítica.

2. Un proceso entra inmediatamente en la sección crítica cuando ésta está desocupada.

3. Un proceso no puede ser suspendido indefinidamente cuando quiere entrar en la sección crítica."

Continúa la autora exponiendo el trabajo de Eisenberg y Mc Guire-72, sobre la condición de justicia, la cual garantiza que un proceso que intente entrar a la sección crítica, podrá lograrlo en el transcurso de un número finito de turnos. La solución famosa (Dijkstra,68) para la exclusión mutua utilizando semáforos surgió como resultado de su profunda crítica a las soluciones con espera activa(53) .

Los semáforos son variables que pueden tener valores enteros y sobre las cuales se pueden realizar las operaciones atómicas -operaciones que no pueden ejecutar dos o más procesos simultáneamente- de incremento V(s) y decremento P(s).

GHEZZI[54] describe la semántica de estas operaciones:

```
"   P(s): si s > 0
          entonces s := s-1
          si no suspende el proceso actual
    V(s): si hay algún proceso suspendido
          entonces despierta el proceso
          si no s := s+1 ...
```

...los semáforos tienen (1) un estructura de datos asociada donde se graba la identidad de los procesos suspendidos, y (2) alguna política de selección para despertar al proceso indicado cuando es requerido por la operación V(s)".

Los problemas analizados tienen su ejemplo correspondiente, así: la exclusión mutua ha sido abordada en el problema de productores-consumidores; el bloqueo mutuo, en el de los cinco filósofos; y el problema de lectores-escritores ha servido para ejemplificar el acceso concurrente durante la lectura, y exclusivo del recurso, durante la escritura.

Steigerwald y Nelson[55] presentan la programación de un simulador en Smalltak-80, para el problema de productores-consumidores, que funciona en máquinas de un solo procesador. El punto central de la simulación consiste en provocar cierto nivel de no-determinismo mediante el uso de las clases Delay y Random. Cabe mencionar que estas dos clases no existen en Smalltak-V.

Oktaba[56] cita a Hoare y Brinch Hansen diciendo que los semáforos son un mecanismo poderoso en programación concurrente pero son poco estructurados por las siguientes razones:

* Las mismas operaciones sirven para expresar dos conceptos diferentes: exclusión mutua y sincronización.

* No se expresa sintácticamente cuales recursos están protegidos por los semáforos.

* Una equivocación al usar operaciones sobre semáforos (intercambio de P y V) no puede ser descubierta al nivel de la compilación y puede causar graves errores al momento de la ejecución.

Para superar estos obstáculos se proponen las regiones críticas y las regiones críticas condicionales por Hoare y otra propuesta por Brinch Hansen en el año de setenta y dos. El concepto de regiones críticas condicionales fue introducido por Brinch y Hansen en lenguaje Edison(57) .

La semántica de la región crítica condicional la expresa claramente la siguiente cita(58) "... With R When B do S; donde R es el recurso compartido, B, una expresión booleana; y S, una instrucción. La semántica de esta instrucción es un poco complicada: la expresión B se evalúa indivisiblemente (con otras regiones del recurso R), si ésta se cumple, entonces se ejecuta la instucción como una sección crítica; en caso contrario, el proceso suspende sus acciones hasta que se cumpla la condición B, liberando el recurso".

Hoare en 74, y Brinch Hansen en 75 proponen encapsular el recurso compartido con sus operaciones en un tipo de datos denominado monitor, cuyas operaciones son atómicas, es decir, la exclusión mutua de los procesos que manipulan las estructuras de datos está garantizada por el lenguaje mismo(59) (60) . La sincronización condicional se logra con las operaciones para suspender un proceso, denominada wait (en la notación de Hoare) y delay (en la notación de Brinch Hansen); y la operación para reanudar un proceso denominada signal y continue, en las notaciones de Hoare y Brinch Hansen, respectivamente.

La versión de Smalltak concurrente incluye un nuevo tipo de objetos atómicos (monitores); sólo un proceso a la vez puede ejecutar cualquier método asociado al objeto. Los mensajes enviados a un objeto atómico se ejecutan serialmente en forma de PEPS ("FIFO")[61] .

4.2 PARALELISMO EN SMALLTAK. EL PROYECTO MUSHROOM

En esta sección analizaremos el trabajo de HOPKINS y WOLCZKO[62] .

Comienzan afirmando que el modelo de objetos discretos que se comunican entre sí ha sido considerado como una base útil para la explotación del paralelismo.

Existen tres modelos diferentes dentro de los lenguajes orientados a objetos. Smalltalk es representante de aquéllos en los cuales los mensajes son llamadas a procedimientos, la dirección de la llamada (y el objeto receptor) se determina dinámicamente. Lenguajes como POOL-T en donde los objetos son módulos de programa ejecutables concurrentemente forman el segundo grupo.

Finalmente, en los lenguajes de tipo ACTORES, el envío de un mensaje activa a un objeto (un ACTOR), sin embargo el objeto transmisor del mansaje continúa sus acciones; en estos lenguajes puede haber un gran número de objetos activos simultáneamente. En los lenguajes actores existe un mecanismo de delegación en donde cada objeto tiene su propia funcionalidad, pero tiene información acerca de otros objetos sobre los cuales pueda delegar responsabiliades cuando es incapaz de responder a un mensaje.

El proyecto MUSHROOM tenía por objetivo construir un sistema orientado a objetos interactivo y distribuido.

Reportan avances en el diseño de un lenguaje orientado a objetos que incluye delegación, herencia múltiple y recursos compartidos por varios usuarios, entre otras características.

En vista de la expresividad limitada para la solución de problemas en paralelo en SmallTalk, señalan los autores, y debido a la escasez de mecanismos de bajo nivel (semáforos y procesos únicamente), se hace necesario desarrollar nuevas abstracciones que enriquezcan el lenguaje.

Proponen dos nuevas clases: una, que permite la evaluación tardía ("lazy evaluation") de mensajes; y otra función de evaluación ansiosa ("eager evaluation"), que se programa dentro de la nueva clase Future. Veamos el siguiente ejemplo:

```
answer := [10 factorial] futureValue.
    .
    .
    .
Transcript show: (answer printString).
```

La primera instrucción asigna a la variable answer un objeto nuevo de la clase Future y lanza un proceso en paralelo para el cálculo del factorial, continúa con la secuencia de mensajes hasta llegar a una en la cual se requiere el valor de answer, el método espera hasta que se deposita en answer el resultado de diez factorial.

Lazy esta diseñada para diferir la ejecución de cierto método. Ninguna de las dos construcciones protegen al programador de efectos laterales indeseables. Future permite el desarrollo de operadores más complejos, por ejemplo el de suma en paralelo.

La ingeniosa implantación de Future está basada en el hecho de que un método suspende temporalmente su ejecución cuando algún objeto no entiende el mensaje que se le envía, e invoca al método doesNotUnderstand:, éste último puede reactivar ("resume") al método anteriormente suspendido.

El control de la concurrencia se lleva cabo redefiniendo el método doesNotUnderstand: para la clase Future. Cada objeto de esta clase posee un semáforo que se utiliza para la sincronización de los procesos. doesNotUnderstand: lanza el nuevo proceso (que se encuentra en una variable de tipo bloque), reactiva al método suspendido y pone al semáforo en estado de espera. Cuando el proceso termina, regresa el valor del semáforo al estado de siga ("signal").

Algunos problemas impiden el uso, transparente al programador, de Future. Se hace necesaria la programación de un método de toque, llamado touch, dentro de Future para esperar a que el resultado del proceso o valor prometido, se reasigne a la variable correspondiente (answer en el ejemplo)

Otro problema mencionado se presenta debido al uso de objetos de tipo Future que consumen memoria y tiempo de procesador aun en los casos en que el valor prometido no se requiera en ningún momento posterior. Un sistema ideal debería realizar alguna actividad colectora de procesos-basura.

Presentan una clase reguladora cuyo objeto es limitar la carga de procesos que un grupo de objetos-Future suministran. Cada uno de estos objetos se relaciona a un regulador que cuenta el número de procesos por ejecutar; cuando el regulador alcanza un límite predefinido, el siguiente objeto Future no puede lanzar un nuevo proceso, razón por la cual debe evaluar el código secuencialmente.

Por último, manifiestan su interés por integrar al proyecto funciones heurísticas que permitan la migración dinámica de objetos desde un procesador hacia otro.

4.3 OBJETOS ACTIVOS

Uno de los primeros estudios reportados sobre objetos activos se debe a Dahl, Dijkstra y Hoare en el año de setenta y dos. Existen diferentes interpretaciones acerca de qué es un objeto activo, en general se refiere a la noción de objetos inteligentes con cierto grado de control autónomo dentro de una organización asíncrona. Los objetos activos son agentes independientes y fuente de conocimiento. La distinción operativa entre objetos pasivos y activos, radica en el hecho de que mientras los objetos pasivos necesitan recibir un mensaje de activación; los activos, en cambio, pueden realizar diferentes acciones sin recibir mensaje alguno(63) .

Actores es la familia de lenguajes que incorporan inicialmente el concepto de objetos activos. La formalización de actores se produjo en un laboratorio de Inteligencia Artificial. Ellis(64) resume las propiedades básicas de un sistema actor:

"1. Un actor se comunica con otro mediante un sistema de mensajes transferidos en forma de un sistema de repartición de correos.

2. Cada actor está formado por una dirección de correo y una conducta ... la conducta de un actor especifica los tipos de mensajes que el actor recibirá y las operaciones realizadas una vez que se han recibido los mensajes.

3. Transmisor y receptor proceden en paralelo asíncronamente durante y después de la transferencia de mensajes..."

La propuesta inicial de actores fue un modelo conceptual que cimentó algunas de las bases para el procesamiento distribuido(65) . Está basado en el hecho de la independencia funcional de los objetos; además, aprovecha la historicidad ("history-sensitive") de los objetos, característica que no poseen los lenguajes funcionales(66) .

Los actores conservan la encapsulación. La herencia entre actores se limita a que el recién creado herede la conducta de su progenitor, pero esta conducta puede cambiar dinámicamente[67] .

Argus, Emerald y Kno son ejemplos de sistemas basados en objetos activos. "Algunos sistemas de objetos activos permiten tanto la ejecución simultánea de métodos dentro de un objeto, como la ejecución simultánea en diferentes objetos"[68] .

Los autores concluyen diciendo que la POO, tal como la conocemos hoy en día, está "cerca de su lecho de muerte", no obstante, ven enormes oportunidades a los sistemas de objetos activos[69] .

4.4 CONCLUSION

El paralelismo indudablemente enriquece los sistemas de aplicación. Sin embargo, produce un nuevo "dolor de cabeza" al programador durante la depuración de programas. Dijkstra[78] en su carta acerca del "GOTO", diserta sobre la dificultad intelectual para la comprensión de las relaciones dinámicas. La especificación formal de programas, afirma OKTABA[71] es una herramienta valiosa para la programación de sistemas sobre arquitecturas en paralelo.

Cabe mencionar, que se han realizado numerosos estudios durante estos últimos años sobre métodos formales para la descripción de la interacción entre el hombre y la máquina. El indeterminismo es uno de los problemas frecuentemente analizados en la búsqueda de mejores interfaces de usuario.

Un ejemplo de arquitecturas de tipo paralelo en Inteligencia Artificial es el conjunto de sistemas expertos que comparten una estructura de datos (un pizarrón) sobre la cual leen o escriben algún tipo de conocimiento; el pizarrón central es un canal de información para la intercomunicación de expertos artificiales[72].

Por último, coincidimos en que los objetos activos serán los futuros sistemas orientados a objetos.

5. CONCLUSION DEL CAPITULO

Los estudios de Pappert plasmados en el lenguaje LOGO; el lenguaje SIMULA-67 de los noruegos Dahl, Myhrhaug y Wygaard; y la conjunción de estos conceptos en SMALLTALK, ha sido la base para el desarrollo de los nuevos lenguajes visuales de programación.

Palotes, ventanas y juegos de dragones y princesas, son sólo efectos poco importantes, que han tenido la programación basada en objetos junto a la alta tecnología del "hardware".

Un área dentro de la cual estos lenguajes resultan ser herramientas idóneas, es en el estudio de los lenguajes pictográficos: "Iconos, pinturas y gráficas simbólicas son consideradas algunas veces como ´palabras´ o ´bloques´ que son colocados en un espacio de dos o tres dimensiones siguiendo algunas reglas de construcción predefinidas para crear expresiones y procedimientos en un lenguaje de programación icónico (o gráfico)"[73] .

II. ESPECIFICACION FORMAL DEL JUEGO

II. ESPECIFICACION FORMAL DEL JUEGO

El fracaso de proyectos de ingeniería de "software" se debe, en la mayor parte de los casos, a especificaciones inadecuadas. Al finalizar el proyecto surgen inumerables expectativas que no cumplen los sistemas y cuyas adaptaciones incrementan enormemente los costos y el tiempo de realización. La especificación es un elemento decisivo durante la etapa de planeación de los sistemas.

Diversas disciplinas han desarrollado y utilizan diferentes técnicas de especificación, que van desde las más informales o ambiguas, hasta las especificaciones formales. Un elemento común a todas es que extraen la información relevante de un medio concreto, destacando sólo aquellos puntos significativos en el contexto en el que se trabaja.

Una de las interpretaciones que se ha dado al concepto de especificación dentro de ingeniería de "software" es para referirse al contrato que celebran el usuario y el programador, donde se establecen las tareas que realiza el sistema. El término tiene muchas otras acepciones en esta disciplina.

Ghezzi(74) agrupa las especificaciones en dos categorías: operacionales y descriptivas. Las primeras son técnicas procedimentales de especificación: diagramas de flujo, máquinas de estado finito y redes de Petri, sirven al autor para ejemplificar este tipo de especificaciones. Otra forma de especificar un sistema es describiendo o declarando sus propiedades: diagramas de entidad relación, la lógica de primer orden y las especificaciones algebraicas, entre otras.

El formalismo matemático subyacente a las especificaciones algebraicas, continúa el autor, es el álgebra. "Las especificaciones algebraicas definen un sistema como un álgebra heterogénea, es decir, una colección de diferentes conjuntos sobre los cuales se definen algunas operaciones".

Nuestro objetivo principal en este capítulo es realizar una especificación algebraica del objeto tangrama, que nos sirva como ejercicio en esta técnica de especificación. Esperamos, además, poder mostrar algunas propiedades intrínsecas importantes del juego.

El curso de Oktaba[75] sobre especificaciones formales nos sirvió como punto de partida para la elaboración de este capítulo. Otra referencia importante fue el libro de Mallgren[76] sobre especificaciones formales de gráficos.

El capítulo se estructura en cuatro secciones que indican un orden metodológico: comenzamos por dar una aproximación intuitiva del problema; presentamos la especificación algebraica y un comentario general a ella; continuamos justificando el método que proponemos para la suma o unión de contornos de polígonos que es una antesala a la formulación de una prueba de correctez de este punto central en la especificación; concluimos el capítulo destacando algunos aspectos inherentes al tangrama.

2.1 PRESENTACION DEL JUEGO

El objetivo del juego consiste en descubrir una partición del tangrama propuesto como problema, en sus siete piezas elementales llamadas tanes.

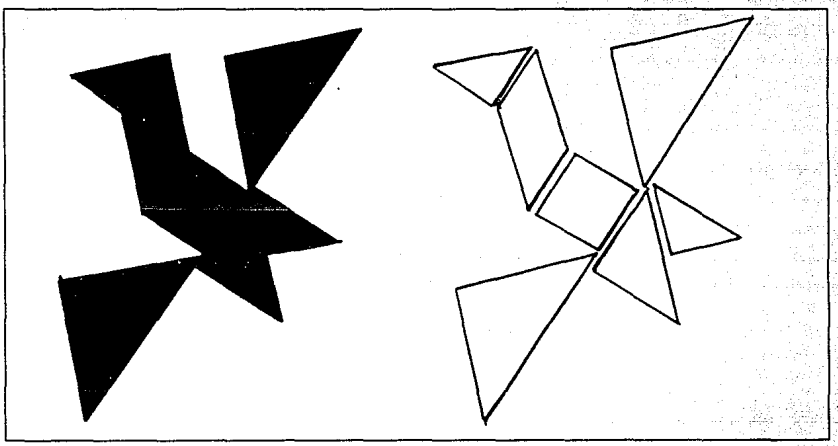


FIGURA 3: A la izquierda, un tangrama problema; a la derecha, su solución en términos de los siete tanes.

Los tanes generalmente son construidos de algún material como madera o plástico; no son superficies totalmente planas, tienen un grosor de dos o tres milímetros. Esto hace que no se puedan encimar uno sobre otro. La solución de un problema siempre involucra a los siete tanes.

Podemos ver en la figura que los tanes son polígonos de tres o cuatro lados cuyos ángulos son múltiplos de cuarenta y cinco grados. Todos los tanes son polígonos convexos y tienen dos caras, el juego permite levantar los tanes de la mesa y voltearlos.

La mecánica del juego consiste en ir retirando los tanes uno por uno hasta llegar a siete para formar un tangrama completo. Los tanes se toman del conjunto sin repetición y siguiendo eventualmente algún criterio de selección, una relación de orden. Es conveniente entender al conjunto de tanes como un conjunto ordenado.

A medida que la persona va colocando los tanes resulta indispensable, también, la capacidad de retirar alguno de los que habían sido previamente colocados sobre la mesa.

El juego termina cuando se ha logrado construir un tangrama completo que sea equivalente al tangrama propuesto como problema. Deben compararse ambos tangramas a fin de terminar o continuar con otro ensayo.

De esta manera, el juego se realiza en una secuencia de quitar y poner, comparando los resultados parciales contra el patrón o problema, hasta llegar a resolver el tangrama.

El tangrama completo, formado por los siete tanes, es una superficie conexa, es decir, ningún tangrama tiene dos o más superficies fracturadas o disconexas. Es posible encontrar tangramas con ciclos interiores, en este caso existe un contorno externo y uno o más contornos internos (Figura 4).

Los tangramas con agujeros son poco frecuentes. Otra clase de tangramas de los cuales sí existen numerosos ejemplos son aquellos que presentan uniones de tipo vértice-arista o vértice-vértice (figura 5).

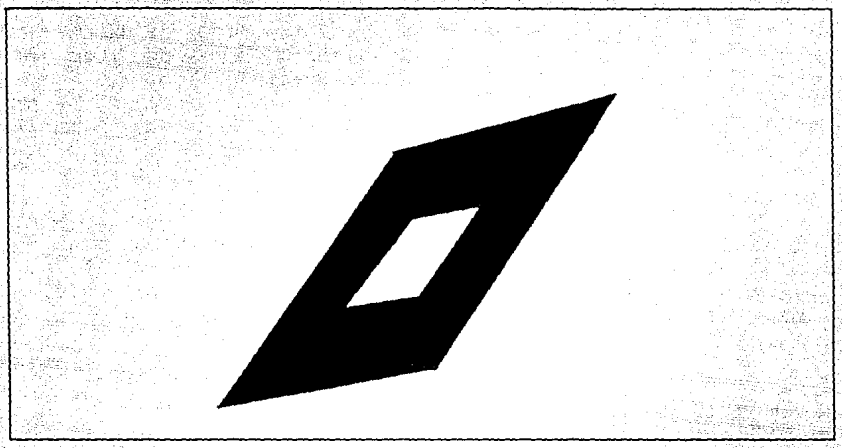


FIGURA 4: Los tangramas que tienen ciclos interiores exigen una representación más compleja durante la especificación.

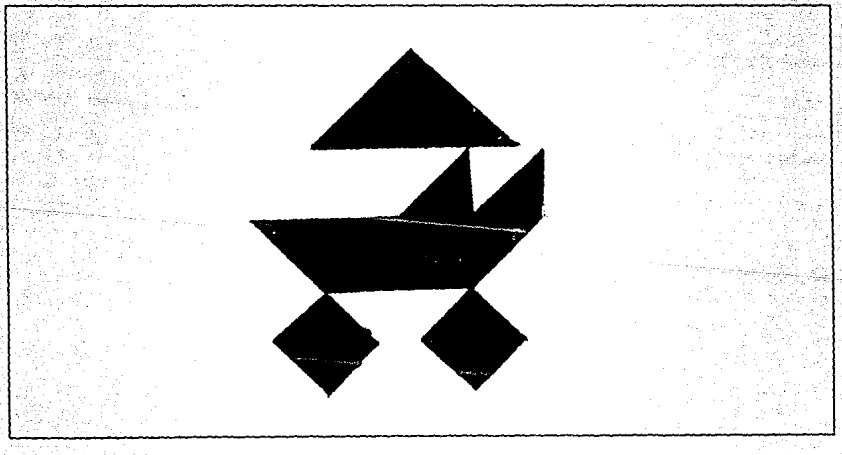


FIGURA 5: Los tangramas con uniones por el vértice también exigen representaciones más complejas durante la especificación.

La combinación de los siete tanes para formar un sinúmero de figuras nos ha llevado a pensar en el tangrama como un objeto sobre el cual se pueden ejecutar un conjunto de operaciones.Este objeto físico tiene ciertas propiedades que debemos conservar durante la especificación y programación del modelo.

Durante el proceso de especificación es necesario hacer algunas consideraciones que posibiliten o bien simplifiquen la especificación del sistema físico. En el tangrama, establecemos algunas restricciones para delimitar el problema y lograr una especificación clara y precisa.

Podemos incialmente pensar en él como si se tratara de un polígono: una superficie cerrada, delimitada por un contorno de aristas. Los tangramas con ciclos internos exigirían además la representación de estos ciclos. Son pocos los ejemplos de tangramas con agujeros y, además, podría haber tangramas con más de un agujero. Los algoritmos de solución se complicarían significativamente y sus estructuras de datos también. Para nuestro propósito incluir esta clase de tangramas no ofrece beneficios importantes.

Los tangramas que presentan uniones por el vértice también deben ser representados en forma distinta, tampoco incluiremos este tipo de tangramas dentro de nuestra especificación.

La reflexión de tanes o tangramas no es difícil de especificar, sin embargo, tampoco brinda ningún aporte significativo a nuestro estudio, por ello la excluimos de nuestro análisis.

El conjunto de ensayos posibles es no numerable: cada tan se desliza sobre cualquier otro en un trayecto continuo. La simulación del juego en la computadora exige resolver este problema de infinitud.

Es factible encontrar diferentes soluciones que delimiten al conjunto de ensayos posibles. Nuestra aproximación considera un número fijo de puntos de contacto sobre cada arista de cualquier tan o tangrama.

Una solución alternativa podría ser, por ejemplo, dado un punto inicial sobre el perímetro de un tan o tangrama (correspondiente al ensayo inicial), continuar con el siguiente ensayo cuyo punto de contacto sería determinado recorriendo una longitud constante sobre el perímetro del polígono. Véase la siguiente figura:

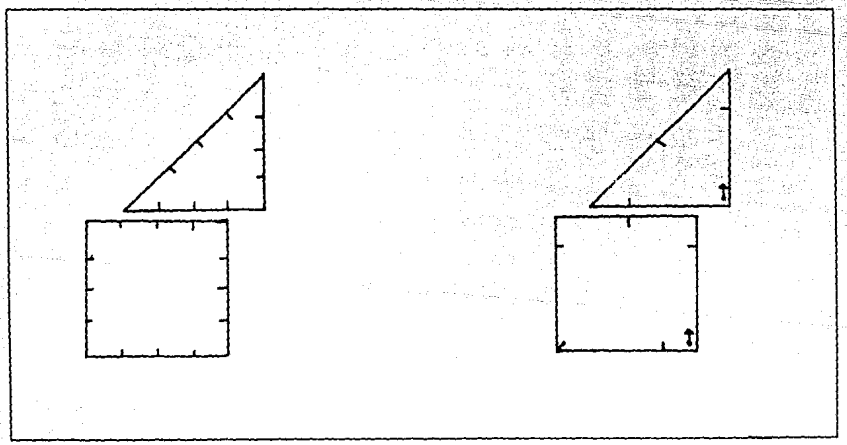


FIGURA 6: Dos posibles interpretaciones que resuelven el problema de un número infinito de ensayos. A la izquierda, se consideran tres puntos de contacto en cada arista; a la derecha, cada ensayo se determina recorriendo una longitud constante partiendo de un punto inicial.

En este punto, podemos dar una definición inductiva del objeto tangrama:

TANGRAMA DEF.

i) Cualquiera de los siete tanes es un tangrama.
ii) La unión de dos tangramas es un tangrama.
iii) Ninguna otra cosa es un tangrama.

Decimos, además, que los tanes deben ser retirados del conjunto sin repetición.

En esta aproximación constructivista, podemos plantear que existen tres operaciones necesarias y suficientes sobre el objeto en cuestión que permitirían simular el juego:

1. La unión o suma de dos tangramas, cuyo resultado es un tangrama.

2. La comparación de dos tangramas para determinar si son iguales.

3. La desunión o resta de un tangrama a otro.

2.2 ESPECIFICACION FORMAL

La sintaxis de nuestra especificación sigue el estándar ASL ("Algebraic Specification Language"), cuya descripción se puede encontrar en el libro de Horebeck[77]

Dividimos la presentación en tres partes: primera, la especificación del objeto tangrama; segunda, las especificaciones de todos los tipos necesarios para construir al tangrama, que corresponden a operaciones menos abstractas (se incluyen como apéndice); tercera, es un ejemplo de la especificación del juego que realiza una búsqueda exhaustiva. Cada una de estas secciones se presenta en orden "Top-Down", es decir, iniciamos presentando los tipos más abstractos y continuamos hacia los menos abstractos.

La especificación del tipo punto es una adaptación de la que da Mallgren[78]. Los tipos pilas, listas y árboles, se tomaron de la especificación de Oktaba[79].

Nuestro algoritmo para la intersección de segmentos es una versión corregida y probada del que aparece en Hégron[68]. A partir de la biblioteca de métodos de Smalltalk[81] escribimos los algoritmos para la operación menorOIgual en puntos; y para las operaciones interseccionAux2 y seIntersectan -modificado para que incluya segmentos unidos por el vértice- en segmentos.

Presentamos a coninuación los diagramas de dependencia de tipos de toda la especificación, e inmediatamente después, la especificación del módulo más importante aunque damos un comentario introductorio al módulo tangrama, recomendamos leer el comentario detallado que se peresenta en la siguiente sección.

Diagrama General de Dependencia de tipos

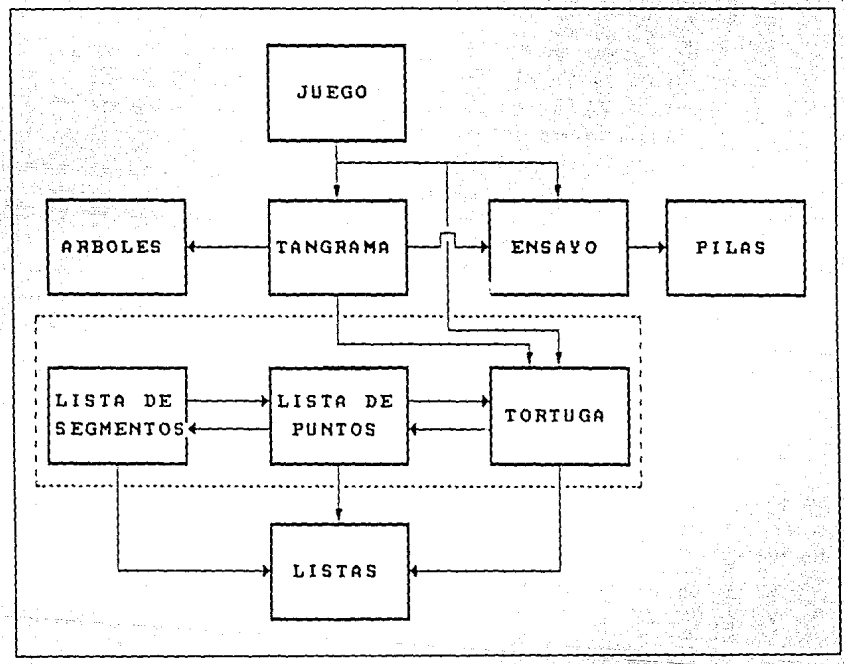


FIGURA 7: Gráfica de generación de tipos

Diagrama de Dependencia de Tipos Del Tangrama

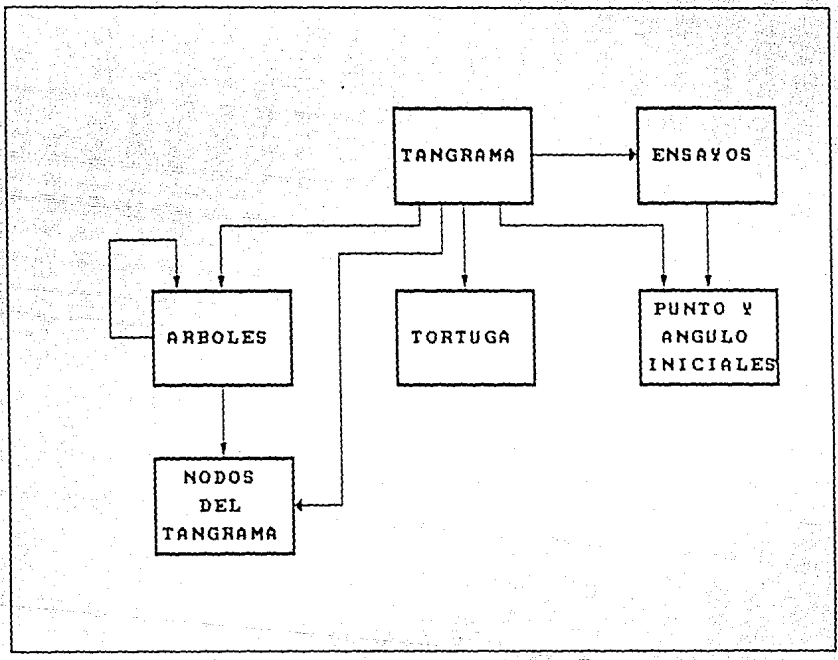


FIGURA 8: Gráfica de generación de tipos del objeto tangrama

Diagrama de Dependencia de Tipos del Juego

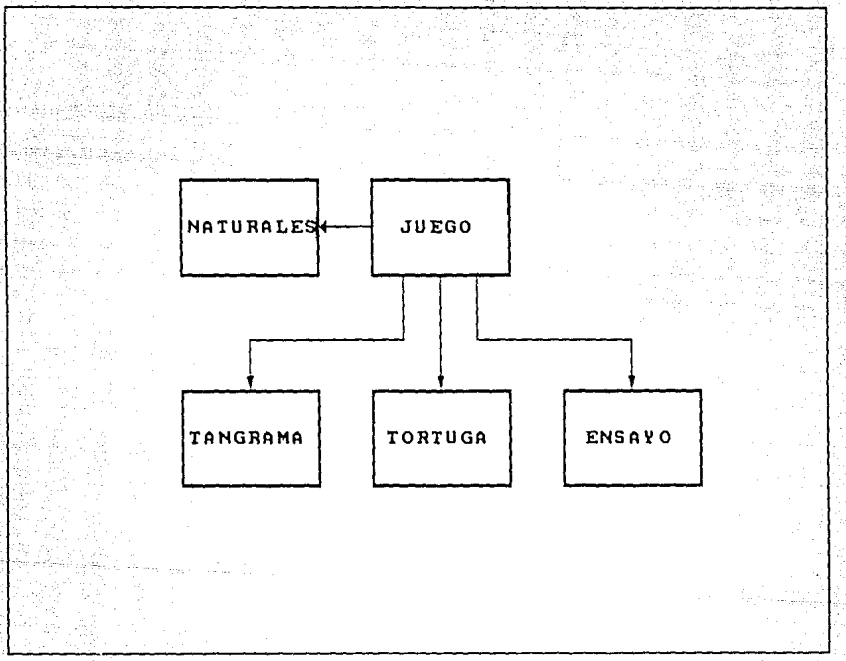


FIGURA 9: Gráfica de generación de tipos del Juego Exhaustivo

Diagrama de Dependencia de Tipos entre las Tres Representaciones del Tangrama

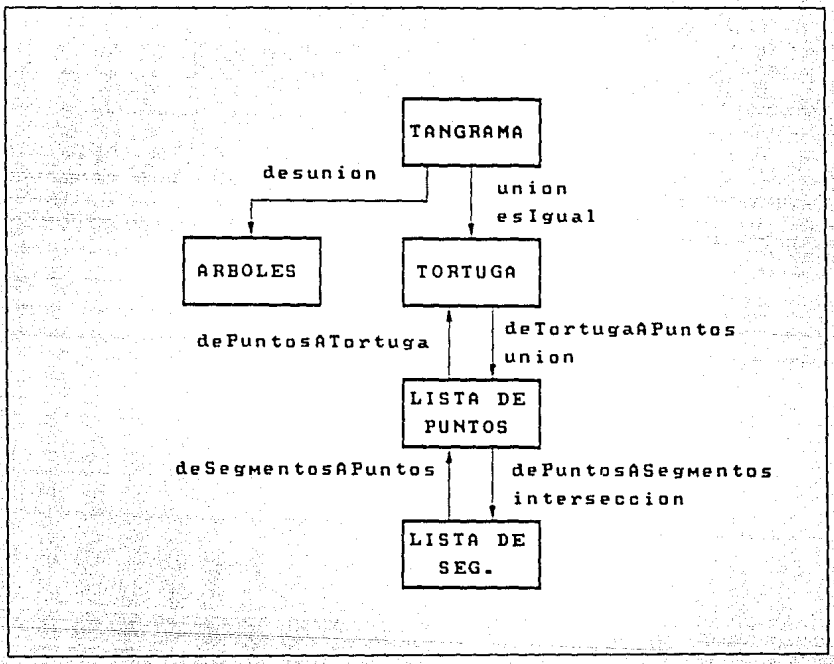


FIGURA 10: Dependencia de las tres representaciones del tangrama.

2.2.1. Especificación de Tangramas.

Como ya hemos mencionado, son tres las operaciones mínimas necesarias en el tangrama; son tres, también, las representaciones mínimas necesarias (o al menos más económicas) para especificar al tangrama. Es un objeto polimórfico que aparentemente no cae dentro de la clasificación de Wegner y Cardelli (capítulo 1).

Cabe mencionar, que Hearsay (1976), el primer sistema experto en reconocimiento de voz, piedra miliar de la IA, también utiliza representaciones múltiples de la voz.

La unión es una operación que se realiza bajo la representación del polígono utilizando una técnica cartesiana (ListaDePuntos); la técnica del lugar geométrico (Tortuga) se utiliza para crear la representación en la cual se resuelve la operación esIgual; la tercera y última es una representación sintáctica del objeto (ArbolBinario), en donde cada nodo contiene a la tortuga y un parámetro que indica la relación de posición que guardan los hijos entre sí.

```
-- Tangrama
-- Instanciación de la spec Tangrama

ESPEC Tangramas;

  IMPORTA tangrama,elemento,esNuevo,izq,der DE Tangrama;
          TODO DE Booleanos;
          coorInic, origen DE PtoYAngIniciales;
          ens DE Ensayos;
          tortuga, númEns TODO DE NodosTang;
  EXPORTA TODO DE  Tangramas;

  OPERACIONES
      esIgual:Tang*Tang -> Bool;
      unión:Tang*Tang*Ensayo -> Tang;
      desunión:Tang*Tang -> Tang;
VAR nodoT:NodoTang;t,t1,t2:Tangrama;ens:Ensayo;
```

```
AXIOMAS
ta1. esIgual(nuevo,t)= si esNuevo(t) entonces cierto si no falso
ta2. esIgual(tangrama(t1,nodoT,t2),t) =
     si esNuevo(t)
     entonces falso
     si no
        si esIgual(tortuga(nodoT),
                   tortuga(elemento(t))  )
        entonces cierto
        si no    falso

ta3. unión(nuevo,t,ens) = t
ta4. unión(tangrama(t1,nodoT,t2),t,ens)=
     si esNuevo(t)
     entonces tangrama(t1,nodoT,t2)
     si no
        tangrama(
          tangrama(t1,nodoT,t2),
          nodoTang(
            dePuntosATortuga(
              unión(
                deTortugaAPuntos(tortuga(nodoT),origen()),
                deTortugaAPuntos(
                  tortuga(elemento(t)),
                  coorInic(ens,tangrama(t1,nodoT,t2),t)
                )
              )
            ),
            ens
          ),
          t
        )

ta5. desunión(nuevo,t) = nuevo
ta6. desunión(tangrama(t1,nodoT,t2),t)=
     si esNuevo(t)
     entonces tangrama(t1,nodoT,t2)
     si no
        si esIgual(t1,t)
        entonces t1
        si no
            si esIgual(t2,t)
            entonces t2
            si no unión(desunión(t1,t),
                        desunión(t2,t),
                        númEns(nodoT)    )

FIN DE Tangramas;
```

```
Tangrama ES INSTANCIA ArbolesEsq;
  RENOMBRA Arbol COMO Tang;
  CON Elementos COMO NodosTang,
      Elem COMO NodoTang;
      arbolNulo COMO nuevo;
      esIgual COMO esIgual;
      creaNodo COMO tangrama;
FIN DE INSTANCIA ArbolesEsq;

ESPEC NodosTang;
  IMPORTA TODO DE Reales;
          TODO DE Booleanos;
          TODO DE TangramaComoTortuga;
          TODO DE Ensayos;
  EXPORTA TODO;
  GENERO NodoTang;
  OPERACIONES
    nuevo: -> NodoTang;
    nodoTang:Tortuga*Ensayo -> NodoTang;
    tortuga:NodoTang -> Tortuga;
    númEns:NodoTang  -> Ensayo;
VAR tor:Tortuga; ens:Ensayo;

AXIOMAS
nt1. tortuga(nuevo) = nil
nt2. tortuga(nodoTang(tor,ens)) = tor

nt3. númEns(nuevo) = nil
nt4. númEns(nodoTang(tor,ens)) = ens

FIN DE NodosTang;
```

```
ESQUEMA ArbolesEsq;
[
  PARAM Elementos;
    IMPORTA TODO DE Booleanos;
    EXPORTA TODO;
    GENERO Elem;
      OPERACION
         esIgual:Elem*Elem -> Bool;
    VAR e,e1,e2:Elem;
    AXIOMAS
      esIgual(e,e)=cierto
      esIgual(e,e1)=esIgual(e1,e)
      esIgual(e,e1) y esIgual(e1,e2) => esIgual(e,e2)
  FIN DE Elementos;
];
ESPEC Arboles;
   IMPORTA TODO DE Elementos;
           TODO DE Naturales;
   EXPORTA TODO;
   GENERO Arbol;
     OPERACIONES
       árbolNulo: -> Arbol;
       creaNodo:Arbol*Elem*Arbol -> Arbol;
       esNulo:Arbol -> Bool;
       izq:Arbol -> Arbol;
       der:Arbol -> Arbol;
       elemento:Arbol -> Elem;

       esHoja:Arbol -> Bool;
       numNodos:Arbol -> Bool;
       cortaHoja:Elem*Arbol -> Arbol;

VAR a,aIzq,aDer:Arbol; e,e1,e2:Elem;
AXIOMAS

ab1. esNulo(árbolNulo) = cierto
ab2. esNulo(creaNodo(aIzq,e,aDer)) = falso

ab3. izq(árbolNulo) = error
ab4. izq(creaNodo(aIzq,e,aDer)) = aIzq

ab5. der(árbolNulo) = error
ab6. der(creaNodo(aIzq,e,aDer)) = aDer

ab7. elemento(árbolNulo) = error
ab8. elemento(creaNodo(aIzq,e,aDer)) = e

ab9.  esHoja(árbolNulo) = falso
ab10. esHoja(creaNodo(aIzq,e,aDer)) =
      si esNulo(aIzq) & esNulo(aDer)
      entonces cierto
      si no falso
```

```
ab11. númNodos(árbolNulo) = 0
ab12. númNodos(creaNodo(aIzq,e,aDer)) = 1 + númNodos(aIzq) + númNodos(aDer)

ab13. cortaHoja(e,árbolNulo) = arbolNulo
ab14. cortaHoja(e,creaNodo(aIzq,e2,aDer)) =
      si esHoja(aIzq) & esIgual(e,elemento(aIzq))
      entonces creaNodo(árbolNulo,e2,aDer)
      si no
         si esHoja(aDer) & esIgual(e,elemento(aDer))
         entonces creaNodo(aIzq,e2,árbolNulo)
         si no  creaNodo(cortaHoja(e,aIzq),e2,cortaHoja(e,aDer))


FIN DE Arboles;

FIN DE ESQ Arboles;
```

2.2.2. Juego Exhaustivo.

```
ESQUEMA PilasEsq;
[
  PARAM Elementos;
    IMPORTA TODO DE Booleanos;
    EXPORTA TODO;
    GENERO Elem;
      OPERACION
         esIgual:Elem*Elem -> Bool;
    VAR e,e1,e2:Elem;
    AXIOMAS
      esIgual(e,e)=cierto
      esIgual(e,e1)=esIgual(e1,e)
      esIgual(e,e1) y esIgual(e1,e2) => esIgual(e,e2)
  FIN DE Elementos;
];
ESPEC Pilas;
   IMPORTA TODO DE Elementos;
           TODO DE Naturales;
   EXPORTA TODO;
   GENERO Pila;
     OPERACIONES
       pilaVacía: -> Pila;
       mete:Elem*Pila -> Pila;
       saca:Pila -> Pila;
       esVacía:Pila-> Bool;
       tope:Pila -> Elem;
VAR p:Pila; e:Elem;
AXIOMAS
pi1. saca(pilaVacía)  =  error
pi2. saca(mete(e,p))  =  p

pi3. esVacía(pilaVacía) = cierto
pi4. esVacía(mete(e,p)) = falso

pi5. tope(pilaVacía) = error
pi6. tope(mete(e,p)) = e

FIN DE Pilas;

FIN DE ESQ Pilas;

-- Pila de Naturales
-- Instanciación de la espec Pilas

INSTANCIA PilasEsq;
  RENOMBRA Pila COMO PilaDeNat;
  CON Elementos COMO Naturales,
      Elem COMO Nat;
FIN DE INSTANCIA PilasEsq;
```

```
ESPEC Ensayos;
  IMPORTA TODO DE Naturales;
          TODO DE Angulos;
          TODO DE Booleanos;
          TODO DE PtoYAngIniciales;
          TODO DE TangramaComoTortuga;
          TODO DE PilasDeNat;
  EXPORTA TODO;
  GENERO Ensayo;
  OPERACIONES
    nuevo: -> Ensayo;
    ensayo:PilaDeNat*Nat-> Ensayo;
    inicial: -> ptoYAngInicial;
    contLocal:Ensayo -> Nat;
    contGral:Ensayo -> PilaDeNat;
    siguiente:Ensayo*Tang -> Ensayo;
    esUltimo:Ensayo -> Bool;
    coorInic:Ensayo*Tortuga ->ptoYAngInicial;
    coorInicAux:Ensayo*Tortuga*Tortuga*Nat*Nat*Real*Real*Real*Real*Angulo*Nat;
    i:Ensayo -> Nat;
    j:Ensayo -> Nat;
    k:Ensayo -> Nat;
    l:Ensayo -> Nat;
    m:Ensayo -> Nat;
    n:Ensayo -> Nat;
VAR ens:Ensayo; a:PilaDeNat; b,k,númAri1,númAri2:Nat; alfa:Angulo; tor1,tor2:Tortuga; tan:
      seg1,seg2,x,y:Real;
AXIOMAS
en1. inicial= ensayo(
              mete(1,mete(1,mete(1,mete(1,mete(1,mete(1,vacía))))),
              1   )

en2. contGral(nuevo) = error --el ensayo debe ser inicializado desde el juego--
en3. contGral(a,b)   = a

en4. contLocal(nuevo) = error --el ensayo debe ser inicializado desde el juego--
en5. contLocal(a,b)   = b
```

```
en5. siguiente(nuevo,tan) = error --el ensayo debe ser inicializado desde juego-
en6. siguiente(ensayo(a,b),tan) =
      si esVacía(contGral(ensayo(a,b)))
      entonces nil ---último ensayo más uno ---
      si no
         si tope(contGral(ensayo(a,b))) < longitud(tortuga(elemento(der(tan))))*
                                          longitud(tortuga(elemento(izq(tan)))
         entonces
            ensayo(
              mete(tope(contGral(ensayo(a,b)))+1, saca(contGral(ensayo(a,b)))),
              contLocal(ensayo(a,b))
         si no
           ensayo(
             mete(
               1,
               contGral(
                 siguiente(
                   ensayo(
                     saca(contGral(ensayo(a,b))),
                     contLocal(ensayo(a,b))  ),
                   izq(tan)
                 )
               ),
             contLocal(ensayo(a,b))
           )

en7. esUltimo(nuevo) = error --el ensayo debe ser inicializado desde juego--
en8. esUltimo(ensayo(a,b),tan) =
     si siguiente(ensayo(a,b),tan) = nil
     entonces cierto
     si no falso

en9.  coorInic(nuevo,tor1,tor2) = error --debe ser inicializado desde el juego--
en10. coorInic(ensayo(a,b),tor1,tor2)=
        coorInicAux(
          ensayo(a,b),tor1,tor2,
          contLocal(ensayo(a,b))//(longitud(tor2)+2)//3),
          contLocal(ensayo(a,b))\\(longitud(tor2)+2)//3),
          contLocal(ensayo(a,b))//((longitud(tor2)+2)\\3+1)/4)*
                    longitud(
                      aristaNúm(
                        contLocal(ensayo(a,b))//(longitud(tor2)+2)//3),
                        tor1
                      )
                    ),
          contLocal(ensayo(a,b))//((longitud(tor2)+2)\\3+1)/4)*
                    longitud(
                      aristaNúm(
                        contLocal(ensayo(a,b))//(longitud(tor2)+2)//3),
                        tor2
                      )
                    ),
          0,0,0,1
        )
```

```
---Recorriendo el contorno del primer tangrama---
en11. coorInicAux(ensayo(a,b),tor1,tor2,númAri1,númAri2,seg1,seg2,x,y,alfa,k)=
        error
en12. coorInicAux(ensayo(a,b),tor1,tor2,númAri1,númAri2,seg1,seg2,x,y,alfa,k)=
     si  k  < numAristo1
     entonces
        coorInicAux(
          ensayo(a,b),sacaDer(tor1),tor2,númAri1,númAri2,seg1,seg2,
          x+cos(alfa)*longitud(der(tor1)),
          y+sen(alfa)*longitud(der(tor1)),
          alfa+angulo(der(tor1),
          k+1
        )
     si no
        coorInicAux2(
          ensayo(a,b),tor1,tor2,númAri1,númAri2,seg1,seg2,
          x+cos(alfa)*(seg1+seg2),
          y+sen(alfa)*(seg1+seg2),
          alfa,
          númAri2-1
        )
---Recorriendo el contorno del segundo tangrama---
en13. coorInicAux2(ensayo(a,b),tor1,tor2,númAri1,númAri2,seg1,seg2,x,y,alfa,k)=
        error
en14. coorInicAux2(ensayo(a,b),tor1,tor2,númAri1,númAri2,seg1,seg2,x,y,alfa,k)=
     si  k  > 0
     entonces
        coorInicAux2(
          ensayo(a,b),tor1,tor2,númAri1,númAri2,seg1,seg2,
          x+cos(angulo(aristaNúm(k,tor2)))*longitud(aristaNúm(k,tor2)),
          y+sen(angulo(aristaNúm(k,tor2)))*longitud(aristaNúm(k,tor2)),
          alfa-angulo(aristaNúm(k,tor2)),
          k-1
        )
     si no
        ptoYAngInicial(x,y,alfa)

en14. i(nuevo) = error --debe ser inicializado desde juego--
en15. i(ensayo(a,b))=
      ensayo(
        a,
        der(sacaDer(sacaDer(sacaDer(sacaDer(sacaDer(contGral(ensayo(a,b))))))))
      )

en16. j(nuevo) = error --debe ser inicializado desde juego--
en17. j(ensayo(a,b))=
      ensayo(
        a,
        der(sacaDer(sacaDer(sacaDer(sacaDer(contGral(ensayo(a,b)))))))
      )
```

```
en18. k(nuevo) = error --debe ser inicializado desde juego--
en19. k(ensayo(a,b))=
      ensayo(
        a,
        der(sacaDer(sacaDer(sacaDer(contGral(ensayo(a,b))))))
      )

en20. l(nuevo) = error --debe ser inicializado desde juego--
en21. l(ensayo(a,b))=
      ensayo(
        a,
        der(sacaDer(sacaDer(contGral(ensayo(a,b)))))
      )

en22. m(nuevo) = error --debe ser inicializado desde juego--
en23. m(ensayo(a,b))=
      ensayo(
        a,
        der(sacaDer(contGral(ensayo(a,b))))
      )

en24. n(nuevo) = error --debe ser inicializado desde juego--
en25. n(ensayo(a,b))=
      ensayo(
        a,
        der(contGral(ensayo(a,b)))
      )


FIN DE Ensayos;
```

```
ESPEC PtoYAngIniciales;
  IMPORTA TODO DE Reales;
          TODO DE Angulos;
  EXPORTA TODO;
  GENERO PtoYAngInicial;
  OPERACIONES
    nuevo: -> PtoYAngInicial;
    PtoYAngInicial;:Real*Real*Angulo -> PtoYAngInicial;
    origen: -> ptoYAngInicial;
VAR
AXIOMAS

pa1. origen= ptoYAngInicial(0,0,0)

FIN DE PtoYAngIniciales;
```

```
ESPEC JuegoExhaustivo;
  IMPORTA TODO DE Naturales;
          TODO DE Ensayos;
          TODO DE Tangramas;
          TODO DE Tortuga;
  EXPORTA TODO;
  GENERO PtoYAngInicial;
  OPERACIONES
    nuevo: -> Juego;
    creaJuego:Tortuga -> Juego;
    problema:Juego -> Tortuga;
    juega:Juego -> Tang;
    busca:Juego -> Tang;
    superUnión:Juego -> Tang;
    tan1:->Tang;
    tan2:->Tang;
    tan3:->Tang;
    tan4:->Tang;
    tan5:->Tang;
    tan6:->Tang;
    tan7:->Tang;
VAR  prob:Tortuga; j(juego); t:Tang; i,j,k,l,m,n:Nat
AXIOMAS
ju1. problema(nuevo)=tortuga(nueva)
ju2. problema(creaJuego(prob))=prob

ju3. juega(nuevo)= tangrama(nuevo,nodoTang(nuevo,nuevo),nuevo)
ju4. juega(creaJuego(prob))=
       busca(
         creaJuego(prob),
         ensayo(inicial),
         tangrama(
             nuevo,
             nodoTang(problema(creaJuego(prob)),nuevo),
             nuevo) )   )

ju3. busca(nuevo,ens,t)= tangrama(nuevo,nodoTang(nuevo,nuevo),nuevo)
ju4. busca(creaJuego(prob),ens,t)=
         si esIgual(t,superUnión(creaJuego(prob),ens)))
         entonces superUnión(creaJuego(prob),ens)
         si no
            si esUltimo(ens)
            entonces error --el problema es insoluble: no es tangrama--
            si no    busca(creaJuego(prob),siguiente(ens),t)

ju3. superUnión(nuevo,ens)= tangrama(nuevo,nodoTang(nueva,nuevo),nuevo)
ju4. superUnión(creaJuego(prob),ens)=
          unión(unión(unión(unión(unión(unión(tan1,tan2,n(ens)),
          tan3,m(ens)),tan4,l(ens)),tan5,k(ens)),tan6,j(ens)),
          tan7,i(ens))
```

```
ju5.  tan1 = tangrama(nuevo,nodoTang(trianGde,nuevo),nuevo)
ju6.  tan2 = tangrama(nuevo,nodoTang(trianGde,nuevo),nuevo)
ju7.  tan3 = tangrama(nuevo,nodoTang(trianMedian,nuevo),nuevo)
ju8.  tan4 = tangrama(nuevo,nodoTang(trianMedic^,nuevo),nuevo)
ju9.  tan5 = tangrama(nuevo,nodoTang(romboide,nuevo),nuevo)
ju10. tan6 = tangrama(nuevo,nodoTang(cuadrado,nuevo),nuevo)
ju11. tan7 = tangrama(nuevo,nodoTang(trianChico,nuevo),nuevo)

FIN DE JuegoExhaustivo;
```

2.3 COMENTARIO SOBRE LA ESPECIFICACION

Una vez que hemos descrito intuitivamente el juego de tangramas, pasaremos a discutir su especificación formal, comenzando por los tipos basicos y continuamos progroesivamente hacia tipos de mayor nivel de abstracción. Omitimos algunos puntos que no exigen explicación.

Naturales, reales y booleanos son los tipos de menor nivel y se importan desde la mayor parte de los módulos de esta especificación. El tipo Angulo, es un tipo que restringe los ángulos que se forman en cualquier vértice, a ser múltiplos de cuarenta y cinco grados.

Punto es el tipo elemental dentro de la especificación de gráficos. El punto es un par ordenado de reales, y sus operaciones regresan la ordenada y abcisa en el plano cartesiano y se definen también, las operaciones comparativas esIgual y esMenorOIgual.

Todo polígono debe cumplir con la condición de que ningún vértice sea colineal a sus vértices adyacentes; esColineal compara la pendiente formada entre un punto y el punto intermedio, contra la pendiente formada por el punto intermedio y el tercer punto. Si amoas son iguales, el resultado es cierto; si no, falso.

Segmentos, a su vez, se define como un par ordenado de puntos, sobre el cual existen las observadoras: origen y esquina que regresan el primer y el segundo elemento del par, respectivamente.

Una operación fundamental común a todos los métodos de construcción de polígonos es la intersección entre segmentos, según Weiler[82]. El algoritmo para la intersección de segmentos se presenta en el siguiente capítulo y es un algoritmo corregido del que presenta Hégron[83].

Intersección calcula el segmento resultante, la mayor parte de axiomas en Segmento se refieren a esta operación.

Así como puntos y segmentos se utilizan para la representación del tangrama en el plano cartesiano, Arista es el tipo básico en nuestra representación de polígonos como gráficas de tortuga, ambos tipos de representación son prácticamente imprescindibles para la realización de alguna tarea en específico.

El tipo Arista es otro par, cuyos argumentos son la longitud y el ángulo que debe girar la tortuga antes de trazar la siguente arista. También está definida la relación de igualdad entre aristas.

Los tres tipos, puntos, segmentos y aristas, podrian ser agrupados y presentarse como instancias enriquecidas del esquema de pares ordenados. Constituyen el núcleo sobre la cual se construyen las representaciones gráficas del tangrama.

Posponemos la discusión de los otros tipos básicos por el momento para explicar la especificación de los tipos estructurados que se utilizan en las representaciones simbolicas del polígono tangrámico.

Una estructura de tipo PEPS (primeras entradas, primeras salidas), que llamamos listas, es el esquema en el que se construyen estas representaciones. Las operaciones generadoras básicas son vacía y meteIzq, además, especificamos una generadora de extensión sacaDer que recorta la lista eliminando el elemento que entró primero. Este tipo está parametrizado precisamente por el tipo elemento que se instanciará más adelante.

numOcurr cuenta el número de veces que aparece un elemento en la lista; rotaDer mete por la izquierda, el elemento que se encontraba al extremo derecho, y lo saca de ahí, en otras palabras la operación produce un corrimento circular de la estructura hacia la derecha; extrae remueve de la lista todas las ocurrencias de un elemento determinado.

Cuando se instancía el tipo elementos como puntos, se obtiene el género lista de puntos que sirve para representar al polígono del tangrama como una nube de vértices en el plano cartesiano. Aquí se calcula el polígono resultante de la unión de dos polígonos.

La justificación del método que proponemos para determinar la unión forma parte de otra sección en este mismo capítulo. Los pasos a grosso modo son: determinar la intersección de los dos polígonos, garantizar que ésta forma un camino abierto, restarla de ambos polígonos, y concatenar sus contornos.

El cálculo de la intersección se hace como una coerción de tipos (polimorfismo ad-hoc). Se construye un nuevo tipo auxiliar que es el tipo listaDeSegmentos. Y se definen un par de operaciones que convierten de uno a otro tipo y viceversa.

Dadas las listas de vértices de los polígonos a unir, se convierten las representaciones a listas de segmentos, se calcula la interseccion en este tipo, y se regresa a la representación en puntos.

La intersección de polígonos resulta ser vacía cuando el ensayo es inválido, esto es, cuando los polígonos se ocultan uno detrás del otro, cuando los polígonos forman uniones por el vertice, o cuando no existe ningún punto de contacto entre ambos polígonos.

El procedimiento consiste en construir la lista Intersección, calculando la intersección de cada uno de los segmentos del polígono P1 con cada uno de los segmentos del polígono P2. Después se verifica que no halla uniones por el vértice, hayUniónPorElVértice busca que todos los segmentos-punto estén unidos a por lo menos un segmento de la intersección que no sea segmento-punto. Antes de regresar, intersección elimina de la lista todos los segmentos-punto, aplicando sacaPuntos.

Estas son todas las operaciones del tipo de datos lista de segmentos.

Regresando al problema de la unión en lista los puntos, encontramos que la última verificación que se hace sobre la intersección, para garantizar que sea un camino abierto, es contar los puntos libres de la lista, deben ser exactamente dos.

esDiscontinua regresa cierto si hay más, o menos de dos puntos libres. Un punto libre se encuentra 1 vez en la lista de la intersección, los puntos no libres se encuentran más de 1 vez. Así definimos esLibre, y apartir de ésta, libre1 y libre2.

Una vez que se ha comprobado que la intersección forma un camino abierto, el siguiente paso del método indica que debemos restar la intersección a cada uno de los polígonos.

Un punto delicado en la especificación de este paso, se refiere al hecho de que en la represetación del contorno como lista de puntos, dos vértices adyacentes siempre van uno tras otro en la lista de puntos; y se asume también que el primer punto de la lista, el extremo derecho, es adyacente al último, el extremo izquierdo. Al restar un camino abierto, de un polígono, el resultado es otro camino abierto (figura 11).

RestaYRota garantiza que los extremos del camino resultante sean el derecho e izquierdo de la lista del camino, siempre y cuando halla algún vértice de P2 que forme parte de la intersección(V1 y V2, en la figura 11)

Cuando algún polígono no tiene vértices no libres que forman parte de la intersección, entonces las rotaciones se hacen hasta encontrar al segundo punto colineal a los puntos libres de la intersección (figura 12)

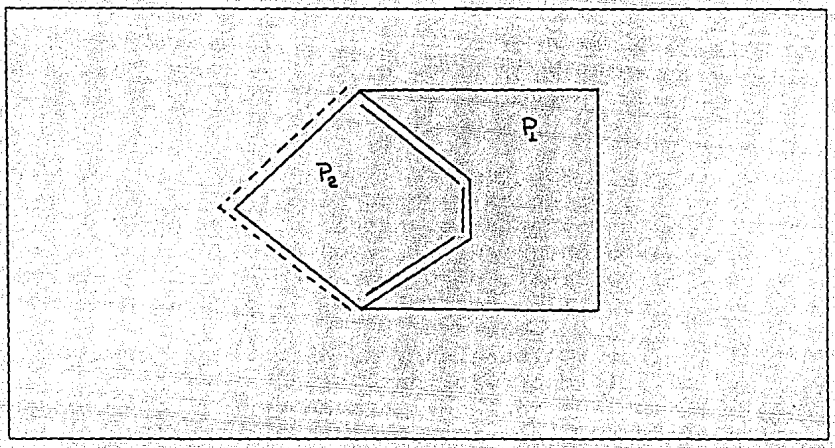


FIGURA 11: La línea con puntos indica el camino abierto resultante de restar la intersección (en líneas dobles) del polígono P2 (en líneas contínuas)

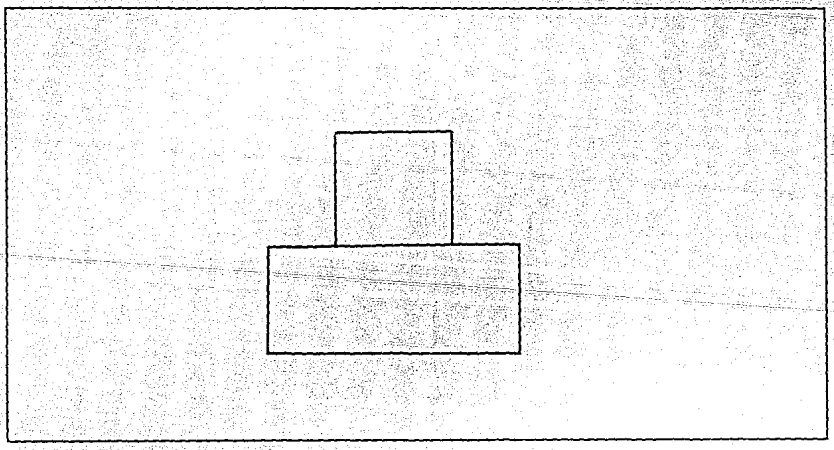


FIGURA 12: Tanto P1 como P2 no contienen puntos no libres en la intersección.El camino resultante sobre P2 debe ser: V1, V2, V3, V4 y sobre P1: V´2, V´3, V´4, V´1.

HayPuntosNoLibres determina si se trata de un caso como el de la figura 11, o como el de la figura 12; y de esta forma, el procedimiento siguido para rotar.

Si no hayPuntosNoLibres de un polígono P en la intersección y entonces se recorreHastaSegundoColineal; pero, si hayPuntosNoLibres, entonces restaYRotaAux rota la lista hasta encontrar el primero y saca todos esos vértices, al terminar, el primer elemento es el extremo del camino; el último de la lista, el otro extremo.

Por último sacaColineales y sacaRepetidos garantizan que en el polígono resultante ningún vértice sea colineal a sus dos vértices adyacentes, y que ningún vértice esté repetido, respectivamente. Esto ocurre sobre sobre los puntos libres, véase la siguiente figura:

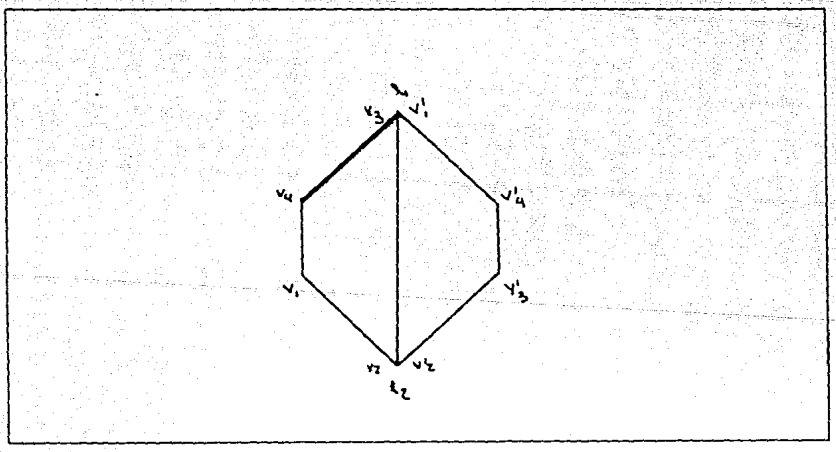


FIGURA 13: V3 es el segundo colineal de P1, a l1 y l2. Entonces el camino resultante sobre P1 es: V3, V4, V1, V2. V´2 es el segundo colineal de P2, el camimo resultante es: V´2, V´3, V´4, V´1. El polígono resultante es V3V4V2V´2V´3V´4V´1.V2=V´2 y V3=V´1, o sea, están repetidos.

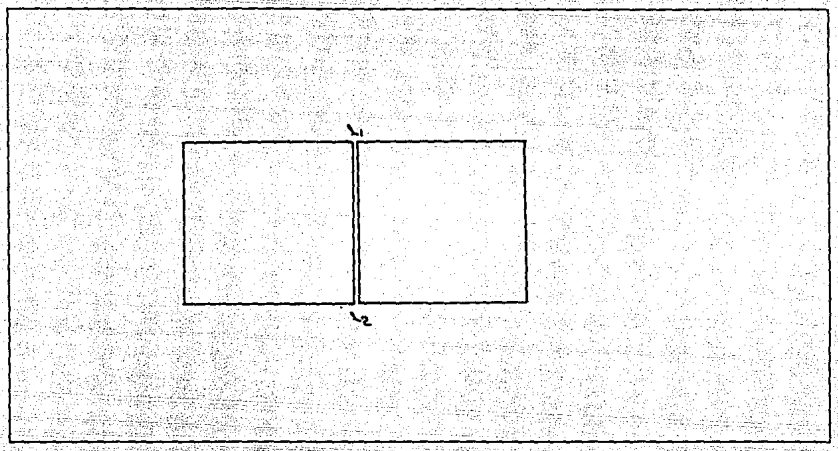


FIGURA 14: Muestra un caso en que l1 y l2 son dos vértices colineales en el polígono resultante.

Como se puede ver, la operación más compleja en la especificación del juego es la unión de contornos. La última operación sobre la lista de puntos, es para convertir del tipo lista de puntos, al tipo tortuga (o lista de aristas). A nivel de especificación basta con calcular la longitud del segmento entre dos puntos, utilizando el teorema de Pitágoras; también calcular el ángulo, y acumularlo para el cálculo del siguiente par de puntos.

En el siguiente capítulo veremos que es necesario, durante la implementación, que al realizar esta conversión, se actualizen algunas variables para conservar la orientación en el despliegue.

La representación del tangrama como gráfica de tortuga se hace en una lista de aristas.Esta representación es la representación simbólica central en nuestra especificación.

La operación que se realiza sobre esta representación es la comparación de los contornos de dos tangramas para determinar si son iguales o no. Intuitivamente, se puede afirmar que dos tangramas son iguales aunque se encuentren en diferentes posiciones sobre la mesa (por rotaciones o traslaciones).

Para realizar la operación de igualdad, se define primero una operación de identidad de contornos; un contorno es idéntico a otro si las tortugas de ambos son idénticas, es decir, sí las listas de aristas son exactamente iguales. Se podría decir que dos tangramas son iguales si la tortuga de uno es igual a una o más rotaciones de la tortuga del otro.

sonIguales compara ambas tortugas para ver si sonIdénticas, sí son idénticas entonces son iguales; pero si no, rota la primera tortuga a la derecha y vuelve a probar si son idénticas, repite este procedimiento hasta volver al primer intento y, si esto ocurre entonces el resultado es falso, no son iguales.

Las operaciones de despliegue, que no especificamos, se contruyen a partir de la tortuga.

La última representación encapsulada en el objeto tangrama, es una representación jerárquica del objeto. Gran parte de los programas de juegos, incluyen en su diseño, algoritmos de "Back-Tracking", en donde se debe poder regresar a un estado anterior.

Así mismo, algunos sistemas expertos son capaces de explicar cúal fue el camino que se siguió para llegar a una conclusión o solución determinada. Un esquema de árbol binario se utiliza para esta tercera representación.

Al igual que listas, está parametrizada por un elemento o nodo del árbol. La semántica de estas operaciones resulta clara de la especificación per se. La operación cortaHoja fue especificada pero no se utiliza en ninguna operación.

El esquema de árboles se instancía para dar origen al tipo tangrama; ahí elemento se intancía como nodoTang. Abrimos un paréntesis para explicar este tipo básico.

nodoTang es un par ordenado, cuyo primer elemento es la tortuga del tangrama, y el segundo, es un ensayo, que no es otra cosa sino una expresión de la relación de posición que guardan los dos tangramas hijos en el árbol para llegar al tangrama padre, y poder reconstruir la representación de tortuga. Este tipo consta de sólo dos observadoras que regresan el primer y segundo argumentos, respectivamente.

Un tangrama es por lo tanto, un árbol binario, cuyos nodos almacenan la representación de tortuga y un parámetro que indica cuál es la relación de posición que guardan los hijos.

Las tres operaciones del tangrama son: unión, esIgual y desunión. La unión de dos tangramas es el tangrama resultante cuyos hijos son los dos parámetros de la unión, y cuyo nodo está compuesto por la unión de los polígonos expresada como una gráfica tortuga y por el ensayo.

De aquí, se define tan como una hoja del árbol. Cada uno de los siete tanes es una constante del tipo tangrama.

esIgual se calcula directamente sobre la representación de tortuga.

desunión compara cada una de las ramas con el tangrama que se pretende desligar ; si éste es igual a cualquiera de los hijos entonces regresa como resultado el hijo complementario; pero, si ninguno de los hijos es igual, entonces se hace una llamada recursiva para intentar descubrir a este tangrama y reconstruyen el nodo al regresar de la recursividad.

Especificación del Juego.

Una vez que hemos construido al objeto tangrama. Definimos a sus siete componentes elementales como hojas del árbol tangrámico cuyas gráficas de tortuga están en el nodo del árbol, así el juego consta de siete piezas o tanes.

El objetivo es claro en este punto:

Juego Definición.

Dada la representación de tortuga de un polígono cualquiera, que llamamos problema; juego es el mecanismo por el cual se determina una partición del problema en sus tangramas elementales (tanes), en un número finito de ensayos; o bien, declara que el problema no tiene solución.

Conjunto de Ensayos Definición.

Ensayos es un conjunto ordenado y finito, mediante el cual se define un recorrido dentro del espacio de búsqueda del juego. Cada elemento del conjunto, que corresponde a un estado posible, define la relación de posición que guardan los n tanes entre sí, tambien la que guardan los hijos en un tangrama en particular.

Estado Definición.

Un estado, es el tangrama resultante de la composición de uniones de los n tanes tomados de dos en dos.

En la especificación formal, se puede ver que la operación juega inicia la busqueda de la solución de un problema propuesto. Debido a que la operación de igualdad se define en el tangrama como una operación entre dos tangramas, juega crea el tangrama correspondiente al polígono problema.

busca realiza la búsqueda desde el ensayo inicial hasta el ensayo cuyo tangrama sea igual al tangrama problema, en ese caso, regresa la descomposición del problema en sus siete tanes.

Si ello no ocurre y se alcanza el último ensayo entonces termina la busqueda exhaustiva en una condición de error, puesto que el problema no es un tangrama que se pueda resolver bajo esta definición de juego.

El módulo de ensayos se especifica como un par ordenado donde el primer elemento del par es una pila de naturales que sirve como un contador general de ensayos, aquí se controla la relación de posición de los siete tanes; el segundo elemento del par, controla la relación de posición de los dos tangramas actuales en que se realiza la operación de unión, también es un natural. Comenzamos por aclarar la semántica de este último elemento.

En la introducción de este capítulo mencionamos el problema de infinitud que habíamos de resolver antes de la especificación y por supuesto de la implementación. El tangrama es un objeto cuyas aristas son continuas y existe un número infinito de puntos entre cada vértice.

Durante la unión, dos tangramas ponen en contacto algún punto de una de sus aristas con un punto del otro tangrama.

Restringimos el número de ensayos declarando que existen sólo tres puntos de contacto en cada arista y un ensayo en cada uno de estos puntos del primer tangrama con uno del otro tangrama. Véase la siguiente figura:

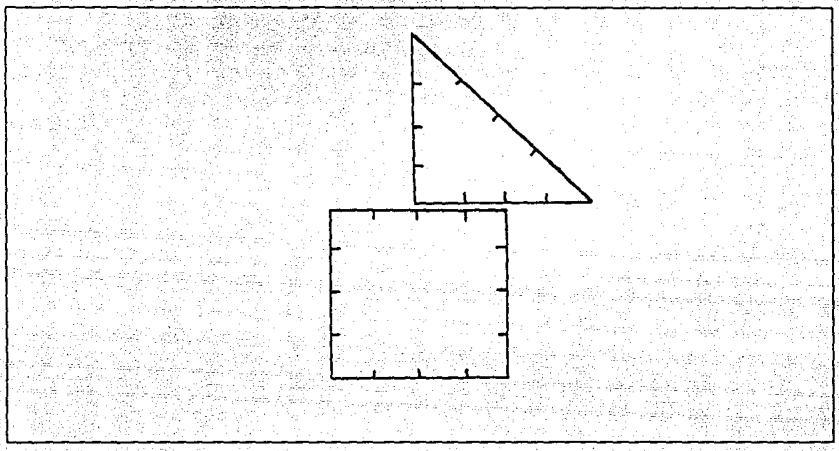


FIGURA 15: muestra la forma en que se unen las aristas en tres puntos equidistantes de contacto.

El contador local es un natural que indica cúales son los puntos de contacto entre el tangrama 1 y 2:

El ensayo 1 corresponde al par (1,1); el 2, al par (1,2); hasta formar la siguiente secuencia:

((1,1),(1,2),...,(1,m),(2,1),(2,2),...,(2,m),...,(n,m))

donde n es el número de puntos de contacto en el tangrama 1; y m, el del segundo tangrama.

Cuando traducimos de la representación de tortuga hacia la representación de vértices en el plano cartesiano, debemos conocer el par de coordenadas y la orientación de la tortuga para comenzar a calcular la posición de los vértices del segundo tangrama en el plano.

origen es una constante que regresa la triada (0,0,0) refiriéndose a la abcisa, ordenada y orientación de la tortuga, respectivamente. Los axiomas de coorInic, declaran la forma en que se debe de encontrar el punto y orientación inicial para calcular la representación como una nube de vértices del segundo tangrama con respecto del primero, en un ensayo determinado.

Los axiomas decodifican el número de ensayo para obtener la pareja de puntos de contacto sobre cada tangrama; después, partiendo del origen, se recorre cada arista hasta llegar al punto de contacto del segundo tangrama. De ahí, continúan sobre el segundo tangrama en orden descendente, es decir, hacia la primera arista del segundo tangrama y en sentido opuesto al que se recorría, el tangrama 1:

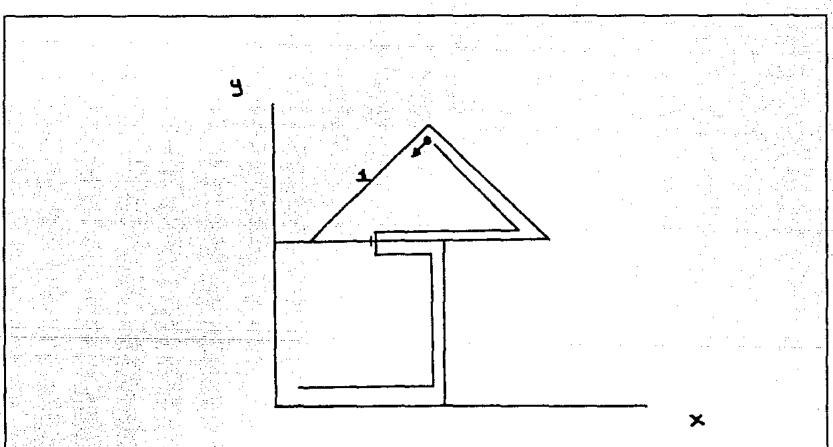


FIGURA 16: Muestra el recorrido a seguir para determinar el punto inicial y la orientación que se debe hacer antes de localizar cualquier vértice en el segundo tangrama.

La pila de conteo general, está formada por seis contadores locales, uno para cada nivel de profundidad en el árbol tangrámico. Para incrementar el contador general se busca incrementar el contador de primer nivel, el menos significativo, si ya había alcanzado su limite superior, entonces se reinicia en el valor 1, y recursivamente incrementa el contador local del siguiente nivel.

Cuando todos los contadores alcanzaron el nivel superior, se han agotado todos los ensayos de nuestra búsqueda exhaustiva.

Las operaciones i,j,k,l,m,n observan los contadores locales de cada nivel.

Por último, hacemos incapié en que la especificación de los módulos juego y ensayos sólo ilustran una posibilidad de trabajo con nuestro objeto de interés: el tangrama.

2.4 JUSTIFICACION DEL METODO PROPUESTO PARA LA DETERMINACION DEL POLIGONO RESULTANTE

La operación constructora básica del objeto tangrama es la unión de tangramas, como ya hemos visto. El problema que ahora nos ocupa es menos abstracto dentro de nuestra especificación y quizá, el punto más delicado tanto aquí como en la implementación, se trata de la operación de suma o unión de los contornos de dos tangramas.

No encontramos reportado en la literatura ningún método de unión de polígonos cóncavos que satisfaciera los requisitos del juego.

La operación unión de contornos se realiza en el tipo cola de puntos que representa al polígono como una secuencia de sus vértices. Describiremos informalmente la semántica de la unión y esperamos que esto sirva como justificación del método que proponemos para realizar esta operación.

Terminología

Algunos de los conceptos que introducimos en este punto forman parte de la teoría de gráficas y han sido presentados por diversos autores [1,2,3,4].
Seguiremos la notación y terminología de Mc Hugh [5].

Una gráfica G(V,A) consiste de un conjunto V de elementos llamados vértices y un conjunto A de pares no-ordenados de miembros de V llamados aristas. En nuestro caso particular, interesa la interpretación cartesiana del concepto de gráficas, por tal razón, un vértice es un punto en el plano y una arista es un segmento de recta que une a dos vértices.

La gráfica formada por un sólo vértice se llama gráfica trivial. El conjunto de vértices se denota como V(G) y el de aristas, como A(G). Se dice que un vértice u en V(G) es adyacente al vértice v en V(G) si (u,v) es una arista en A(G). La arista entre el par de vértices u y v se denota como (u,v). u y v son puntos terminales de la arista (u,v) y elementos de V(G).

Se dice que la arista (u,v) es incidente sobre u y sobre v. Un vértice que no tiene aristas incidentes se llama vértice aislado, mientras que un par de aristas que inciden sobre un mismo vértice son aristas adyacentes.

El grado de un vértice v, denotado como gdo(v), es el número de aristas incidentes sobre v. Se dice que una gráfica G(V,A) es regular de grado n si todos los vértices en V(G) son de grado n.

Una subgráfica S de la gráfica G(V,A) es una gráfica S(V',A') tal que V' está contenido en V y A' está contenido en A, y los puntos terminales de A' son elementos de V' también. Se utiliza la notación G - v, cuando v está en V(G), para denotar la subgráfica inducida de G con V - (v).

Así mismo, si V' es un subcojnunto de V(G) , entonces G-V' denota la subgráfica inducida de G con V-V'. Se utiliza la notación G - (u,v), donde (u,v) está en A(G), para denotar a la subgráfica G(V,A-((u,v))). Si se agrega una arista nueva (u,v) a G(V,E) donde u y v son elementos de V(G), se obtiene la gráfica G(V,A U ((u,v))), lo cual se denota como G(V,A) U (u,v).

Se define camino abierto desde un vértice u en G hacia un vértice v en G como una secuencia alternada de vértices y aristas,

v1,a1,v2,a2,...,ak-1,vk,

Donde v1 = u, vk = v, todos los vértices y aristas en la secuencia son distintos, los vértices sucesivos vi y vi+1 son puntos terminales de las arista intermedia ai.

Si relajamos la definición anterior, y permitimos que p1 y p2 coincidan, llamaremos ciclo al camino cerrado resultante.

Llamamos puntos libres o extremos a los puntos que se encuentran el final de un camino abierto.

Se dice que dos vértices u, v en una gráfica están conectados si y sólo si existe un camino de u a v. Se dice que G es conexa si y sólo si cualquier par de vértices en G están conectados.

Definiciones

POLIGONO

Polígono es una gráfica P(V,A) conexa, regular de grado dos, tal que A(p) es un conjunto no vacío de segmentos en el plano cartesiano y V(p) un conjunto de puntos, donde ningún par de segmentos de A(P) se intersectan, ni existen tampoco dos segmentos adyacentes en A(P) que sean colineales.

INTERSECCION DE SEGMENTOS

La intersección de dos segmentos o aristas s1 int* s2, es el segmento formado por el conjunto de puntos que pertenecen tanto a s1 como a s2.

---

* Utilizaremos abrebiaturas subrayadas en vez de los símbolos formales que regularmente se utilizan.

Si las aristas no se tocan entonces s1 int s2 = fi, el segmento vacío; si la intersección es un solo punto v, entonces el segmento-punto resultante es (v,v)

INTERSECCION DE POLIGONOS

La intersección de dos polígonos P1 int P2 es la gráfica I(V,A) tal que A(I) es el conjunto formado por la intersección de cada una de las aristas o sementos en P1 contra cada arista en P2:

A(I) = ((u,v); (u,v) = s1 int s2 para toda a1 está A(p1), para toda a1 esta A(p2))

OCLUSION

Dos polígonos P1, P2 se ocluyen o superponen si existe cualquier par de semgentos se intersectan en un punto que no es vértice:

(u1,v1) int (u2,v2) = (w,w) tal que w no está V(p1), w no está V(p2), (u1,v1) esta A(p1),(u2,v2) esta A(p2)

UNION POR EL VERTICE

Dos polígonos P1, P2 están unidos por el vértice si existe (w,w) está P1 int P2, w es un vértice aislado de V(P1 int P2).

RESTA DE SEGMENTOS

La resta o diferencia entre dos segmentos s1 - s2 es el conjunto de segmentos formado por todos los puntos tales que u esta ((V(P1) or V(P2)) y u esta (s1 y s2) or u no esta (s1 y s2)

RESTA DE POLIGONOS

La resta entre dos polígonos P1 - P2 es la gráfica cuyo conjunto de aristas está dado como:

A(P1-P2) = (s; s esta s1-s2, para todo s1 esta A(p1), para todo s2 esta A(p2))

UNION DE POLIGONOS

La unión de polígonos P1 unión P2 = G(V,A) donde

A= (s; s esta (P1-P2) or s esta (P2-P1))

Método para la Determinación del Polígono Resultante de la Unión de dos Polígonos.

El método que proponemos se formula en dos pasos:

i) Probar que el ensayo es válido.

ii)Si es válido, determinar el polígono; si no continuar con el siguiente ensayo.

Paso i) Probar que el ensayo es válido, esto es, que la gráfica resultante de P1 unión P2 es un polígono. (propiedad de cerradura de la unión)

Esta prueba se puede formular en el siguiente enunciado:

ENUNCIADO 1:

La unión de dos polígonos P1 unión P2 es un polígono Pr si y sólo si P1 int P2 forma un camino abierto y para toda (u,v), (v,w) esta A(P1 union P2), dos segmentos colineales, se substituyen por un solo segmento (u,w) en A(P1 union P2)

JUSTIFICACION

Por el enunciado 1 basta mostrar que la intersección forma un camino abierto para validar el ensayo. Sin embargo, demostrar la correctez del enunciado no es trivial.

La primera parte de la demostración garantiza una condición de suficiencia:

Si P1 int P2 forma un camino abierto y se sustituyen las aristas colineales, entonces Pr es un polígono.

Los puntos extremos del camino l1 y l2 son puntos tanto de P1 como de P2, además existen dos caminos diferentes que unen a dos puntos cualesquiera en un polígono; entonces existe un camino l1,...,l2 por P1 que no incluye puntos de P2; y existe también otro camino l1,...l2 por P2 que no incluye puntos de P1 (excepto a l1 y l2 por supuesto). Pr es el ciclo formado por ambos caminos.

Nótese que esta afirmación es equivalente a restar la intersección de ambos polígonos, tal como se definió la unión.

El complemento de la justificación es más difícil de probar:

Si Pr es un polígono entonces forma un camino abierto P1 int P2

O se podría reexpresar como:

Si P1 int P2 no forma un camino abierto, entonces Pr no es un polígono.

P1 int P2 no es un camino abierto si:

i) P1 int P2 es la gráfica vacía.

ii) P1 int P2 es la gráfica trivial.

iii) P1 int P2 forma algún ciclo.

iv) P1 int P2 es conexa y tiene algún vértice de grado distinto de 2 que no sean los puntos extremos.

v) P1 int P2 es disconexa.

caso i) si P1 int P2 es la gráfica vacía, entonces Pr no es un polígono.

La resta de P1-P2 = P1 y P2-P1 = P2 puesto que los polígonos no se tocan. Por la definición de unión Pr = P1 unión P2 tal que A(P1 union P2) = A(P1) union A(P2). La gráfica resultante tiene dos ciclos: P1 y P2. Un polígono es un sólo ciclo.

caso ii) si P1 int P2 es la gráfica trivial entonces Pr no es un polígono

La demostración es igual que en el caso i) (Véase la definición de resta de segmentos)

caso iii) si P1 int P2 forma algún ciclo entonces Pr no es un polígono.

Los puntos de la intersección son puntos tanto de P1 como de P2, si la intersección forma un ciclo, entonces ese mismo ciclo está en P1 y P2; como P1 y P2 son polígonos entonces P1 int P2 = P1 = P2 y P1 union P2 = fi, la gráfica vacía. La gráfica vacía no es un polígono por definición.

caso iv) si P1 int P2 es conexa y tiene algún vértice de grado distinto de 2 que no sean los puntos extremos, entonces Pr no es un polígono.

Todos los vértices de la intersección excepto los puntos libres l1, l2 son vértices tanto de P1 como de P2. Lo cual contradice el que todo polígono es una gráfica regular de grado dos.

caso v) si P1 int P2 es disconexa entonces Pr no es un polígono.

Si P1 int P2 es disconexa y contiene uno o más puntos aislados, estos puntos aislados generan vértices de grado cuatro en la unión, lo que contradice la definición de polígono. (véase la definición de resta de segmentos)

Si P1 int P2 es disconexa y no contiene puntos aislados entonces P1-P2 es diconexa y P2-P1 también.

Supongamos que P1-P2 tiene dos caminos abiertos y P2-P1 también.

Existen cuatro puntos extremos l1,l2,l3,l4. Supongamos que P1 es un ciclo definido como:

l1,...,l2,...,l3,...,l4,...l1

entonces P2 es un ciclo definido como:

l1,...,l2,...,l3,...,l4,...l1

o P2´

l1,...,l2,...,l4,...,l3,...l1

Si P1 int P2 es:

l1,...,l2 l3,...,l4

Entonces P1 union P2:

l1,...,l4,...,l1 l2,...,l3,...,l2

La unión forma dos ciclos, entonces no es polígono.

El caso de P1 union P2´. P2´ no es polígono puesto que hay algún par de segmentos que se intersectan:

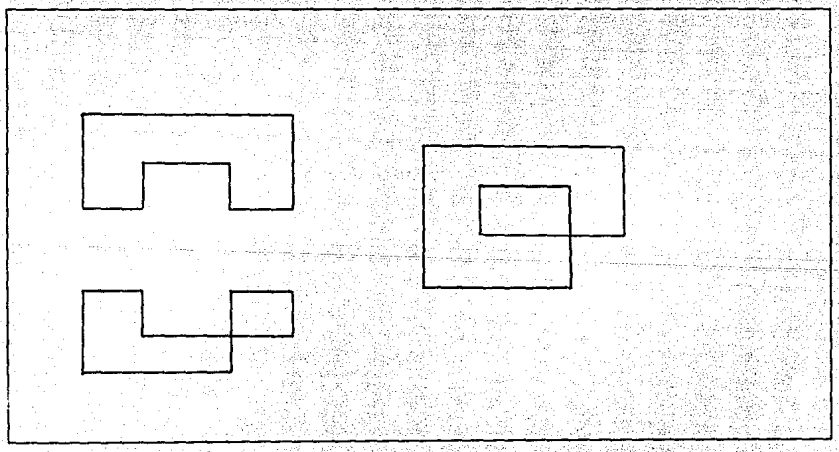


FIGURA 17: A la izquierda P1, P2´; a la derecha, la unión de ambas gráficas

Aunque esta última prueba es poco clara, sí nos da una idea intuitiva de qué sucede en el caso de intersecciones que forman dos caminos abiertos. Falta probar también el caso de intersecciones que forman n caminos abiertos, suponemos que debería hacerse por inducción.

PASO ii) Si es válido el ensayo, entonces determinar el polígono.

La definición de unión de polígonos dice que el polígono resultante está formado por todos los sementos resultado de restar P!-P2 y P2-P1.

La operación de resta de segmentos es una operación costosa y la de polígonos lo es aún más. Por ello, el método que seguimos es equivalente y se plantea:

ENUNCIADO 2:

Si P1 union P2 es un polígono Pr, entonces A(Pr) es el conjunto formado por los segmentos (u,v) tales que

i) u,v no esta en P1 int P2 y (u,v) está en A(P1) o (u,v) está en A(P2)

ii) u es adyacente a algún vértice de la intersección y v también es adyacente al mismo vértice de la intersección. u está en V(P1) y v está en V(P2)

iii) u es colineal a (l1,l2), v es l1 donde distancia(u,l1) < dist(u,l2). u está en V(P1) y v está en V(P2)

Se puede ver por las definiciones de unión de polígonos, resta de polígonos y resta de segmentos que todos los segmentos de la intersección no forman parte de la unión, y únicamente los dos puntos extremos l1, l2 son vértices de la unión.

Así mismo, todos los vértices que no forman parte de la intersección si son parte de la unión.

En el caso de las aristas ocurre algo similar. las aristas cuyos vértices no forman parte de la intersección, deben formar parte de la unión. Cuando un extremo del segmento es parte de la intersección y el otro no, entonces se forma una arista con los vértices de ambos polígonos que son adyacentes a la intersección o son los extremos de ésta. Cuando la intersección está formada por dos únicos puntos libres, un sólo segmento (l1,l2), entonces es necesario hacer la prueba de colinealidad y distancia para determinar las dos aristas resultantes.

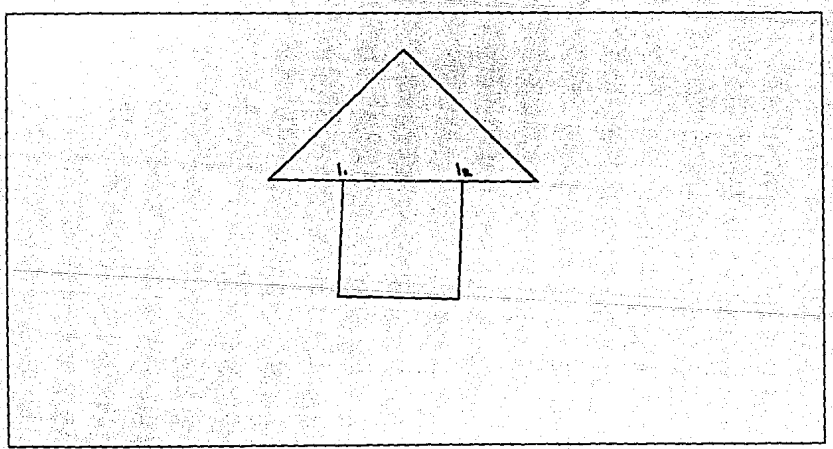


FIGURA 18: Cuando la intersección está formada por dos únicos puntos, es necesario hacer la prueba de colinealidad y distancia

2.5 CONCLUSION DEL CAPITULO

Velarde(89) dice que es frecuente encontrar problemas de gráficas en dondes es útil tener funciones que convierten una representación gráfica en otra sobre la cual es más fácil ejecutar ciertas operaciones, y una vez realizadas, regresar a la representación original.

El tangrama no es la excepción, por el contrario, pensamos que es un muy buen ejemplo de los beneficios que se pueden obtener mediante esta coerción de tipos gráficos.

La especidicación algebraic a nos peremite ver dentro de una sola cápsula (el módulo tangrama) represe4ntaciones múltiples que coexisten y cooperan en la descripción de nuestro objeto de interés.

Como ya hemos mencionado, las tres operaciones del tangrama son su constructora básica (la unión), una observadora (la igualdad) y una "destructora" de extensión (la desunión). A cada una de ellas corresponde una representación: dentro del plano cartesiano, fuera del plan y una representación jerárquica, respectivamente.

Abelson(98) nos dice: "una gran diferencia entre la geometría de tortuga y una geometría de coordenadas descansa sobre la noción de las propiedades intrínsecas de la figura. Una propiedad intrínseca es aquella que depende sólo de la figura en cuestión, no en la relación de las figuras con un marco de referencia. El hecho de que un rectángulo tiene cuatro ángulos iguales es intrínseco al rectángulo. Pero el hecho de que un rectángulo en particular tiene dos lados verticales es extrínseco, puesto que se requiere un marco de

referencia para determinar cuál es la dirección "vertical". Las tortugas prefieren descripciones intrínsecas de las figuras".

La operación de igualdad entre dos tangramas tiene que ver sólo con sus propiedades intrínsecas y no es necesario un sistema de coordenadas externo para poder realizar esa comparación.

La unión, por el contrario, es una operación que está parametricada por los dos tangramas y la relación de posición que guarda uno con respecto al otro; en esta operación es prácticamente indispensable tener un marco externo de referencia.

En el siguiente capítulo exponemos nuestro intento, y fracaso, al tratar de resolver el problema de la unión de polígonos en la representación de tortuga; la razón fundamental se debe a otra propiedad que señala Abelson$_{(91)}$ sobre las tortugas referente a que éstas son representaciones locales; la tortuga sólo vé una pequeña posición del plano que rodea su posición actual. Intentar determinar si dos tangramas se ocluyen o superponen nos fue imposible en la representación de tortuga.

La tercera es una representación evolutiva del tangrama, en ella se "recuerda" el camino seguido para construir al objeto, por ello, preguntas como: ¿Cuál es la solución encontrada?, ¿Qué tangrama resulta al extraer o desunir un tangrama de otro? o ¿Cuáles son los tanes que conforman un determinado tangrama?; sólo pueden ser contestadas desde esta representación.

III. IMPLEMENTACION

III. IMPLEMENTACION

En los dos dos primeros capítulos planteamos los fundamentos técnicos de la materia en cuestión; en este capítulo nos avocamos al problema concreto de la programación de un simulador del juego de tangramas.

Cuando nos preguntábamos al inicio del proyecto, cuál sería el lenguaje más adecuado para programar el simulador, las razones no eran suficientemente claras; a medida que avanzamos, resultó evidente la elección: SMALLTALK.

Por un lado, el tangrama es un objeto que para ser simulado en la computadora, se requiere de un lenguaje de programación gráfico.Muchos lenguajes de programación poseen algunas herramientas para manejo del despliegue gráfico, pero no por ello son lenguajes visuales de programación(como dijimos en el primer capítulo).

En segundo término, buscábamos que la diferencia entre la especificación y la implementación fuera mínima. Smalltalk es uno de los lenguajes en que se encuentra incluído el concepto de tipo de datos abstractos, acorde a las especificaciones algebraicas.

En tercer lugar, tenemos que encontrar un lenguaje que facilite la programación de un experto artificial cuya estrategia de solución sea la abstracción.Como veremos en el siguiente capítulo, el concepto de clase es muy útil para entender la abstracción.Esta es la causa principal por la cual decidimos utilizar Smalltalk.

Se encuentran diferencias significativas entre los programas y la especificación, puesto que esta última va un paso adelante de la programación.Algunos "algoritmos" y sus "estructuras de datos" son más claros y sintéticos en el capítulo anterior, lo cual se puede atribuir a la especificación per se, y sobre todo, a una mejor comprensión del problema.

Recordamos, e interpretamos, el texto de Fairley(92)en que alude a métodos de desarrollo de "software" donde la especificación de requisitos, y la implementación, forman un ciclo repetido hasta aproximarse al objetivo, generalmente inalcanzable, de cumplir los "deseos" del "cliente".Aunque teóricamente sea posible realizar una especificación formal completa, correcta, clara y precisa partiendo del problema concreto; nuestra experiencia es mucho más pragmática y modesta en este sentido.

Sólo nos resta, invitar al lector a compartir el capítulo más emotivo de nuestra disertación.

3.1 Quetzalc nuevo comenzamos.

En alguna ventana de trabajo del ambiente de Smalltalk, tecleamos el mensaje comenzamos enviándolo a una instancia de la clase Quetzalc.El monitor se cubre ahora con cuatro ventanas, unidas bajo un título común que dice: QUETZALCOATL.Este es el nombre de nuestro futuro experto artificial, que dentro del contexto de esta tesis, no obedece a ninguna razón particular.

La ventana superior izquierda, es una ventana de texto dedicada a comentar el avance del sistema en la solución del problema, algunos sistemas expertos realizan actividades de esta índole.Hasta el momento aparece la numeración de ensayo en curso, y por último la descripción del ensayo solución; nos ha servido también, para la depuración ("debugging") del simulador.

A su derecha, se puede ver una imagen gráfica del conjunto de tareas que intervienen en el problema.Nuestra primera aproximación consistió en realizar una búsqueda exhaustiva de dos tanes.En la siguiente fase, implementamos la misma búsqueda para el problema de tres tanes, que ya es un problema "isomorfo" al de siete piezas; esta afirmación la discutiremos más adelante.

En la ventana inferior derecha, aparece la imagen del tangrama problema.

Por último, en la ventana inferior izquierda, se observa una imagen animada del desarrollo del juego, que muestra en cada cuadro un ensayo, y en él una última viñeta del ensayo solución.

A las ventanas de texto en Smalltalk, se puede asociar on objeto método, cuya ejecución regresa un objeto de la clase Menu (véase método abrecon:, de la clase Quetzalcoatl en el apéndice).

Este método difiere su ejecución, hasta recibir un mensaje del ratón (o del teclado); en cuanto recibe el mensaje, aparece en la ventana de texto un menú "pop up".Al menú de nuestro sistema, lo crea un método que se denomina menuPrincipal, y tiene seis opciones que describimos a continuación:

1) ¿cuántas piezas?, ejecuta el método tipoDeJuego que sirve para actualizar una variable de instancia llamada tipoJuego que determina el número de piezas que se usarán: dos o tres; 2) ¿qué piezas?, sirve para que el usuario defina el conjunto de tareas que ha de utilizar; 3) ¿cuál problema?, se solicita al usuario mediante un prompt -igual que en las otras opciones- que seleccione el número de problema que el simulador resolverá; 4) Jugar, proporciona el disparo inicial que desencadena la "balacera", o en otras palabras, el mensaje que activa al simulador para que se inicie el juego; 5) Prueba, es una opción de utilería para el programador, que facilita la prueba de los programas; 6) Salir, ejecuta el método fin, que termina la sesión cerrando la ventana QUETZALCOATL.

Dentro de la clase Quetzalc, además existen otros métodos para la inicialización de cada una de las ventanas, creación de nuevos Quetzalcóatls, etc. que se incluyen como apéndice.

Cabe mencionar, que a este nivel de profundidad, la programación tiene un carácter declarativo:Las ventanas, aparecen como tableros de control, sobre los cuales se presionan botones que activan objetos; el método que presenta esta característica más claramente es abreCor:.

La expresión Quetzalc nuevo comenzamos, crea un objeto quetzalc; se podía haber elegido enviar el mensaje Comenzamos, directamente a la clase Quetzalc; preferi-

mos tomar el primer camino, para ubicar al simulador en los indicios de la programación concurrente.Diferentes instancias de Quetzalc podían cooperar simultáneamente en la búsqueda de la solución del problema, las variables de la clase servirían como canal de comunicación entre estos objetos.

De la clase Quetzalc penden la mayor parte de las otras clases de nuestro sistema:

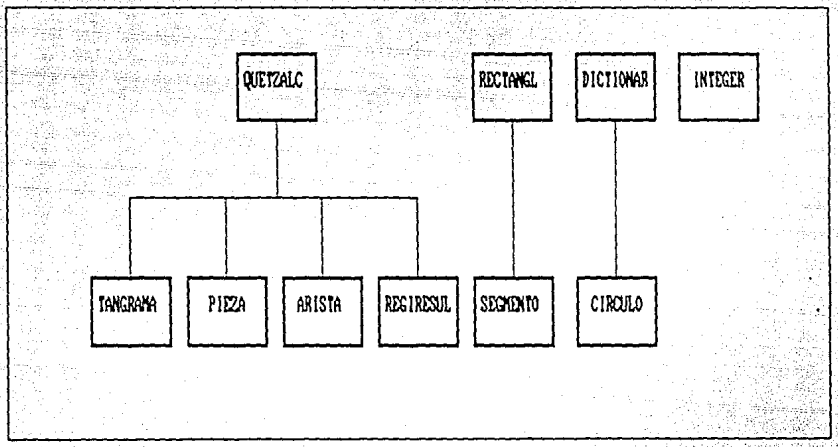


FIGURA 19: Diagrama general de clases.

Es necesario reestructurar este árbol para mejorar la modularidad del sistema, proponemos la siguiente jerarquía. (véase figura 20)

Así, la función de Quetzalc sería reunir a las clases más importantes del simulador, y crear nuevas instancias de si mismo.La función para definir el espacio, es delegada a la subclase Ambiente; y las operaciones propias del juego, a la subclase del mismo nombre.

Bajo este esquema existen aún diferencias con los módulos de la especificación, nuestra intención es sólo mostrar como Quetzalc, en la implementación actual, mezcla funciones que debían ser aisladas.

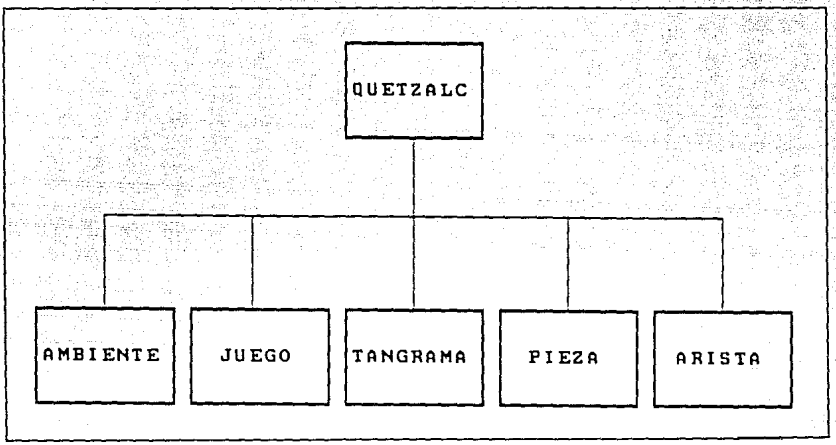


FIGURA 20: Diagrama general de clases. Sólo se muestran las subclases de Quetzalc.

3.2 La clase Tangrama.

Describiremos en primer lugar a las variables de instancia de la clase Tangrama.Pieza1 y Pieza2 son instancias de la clase Pieza, son dos polígonos en su representación de tortuga.Nótese que en la especificación, estas variables corresponderían a los dos hijos en el árbol -tangrámico, y que ahí son del tipo tangrama; un tipo definido recursivamente.Aunque esto no es una diferencia significativa; si es una solución más elegante en la especificación.

puntoUnion1 y puntoUnion2, son los parámetros que indican la relación de posición entre pieza1 y pieza2, es el ensayo decodificado como los puntos de unión entre ambas piezas.

piezaTang es la tortuga del tangrama y formaTang es una representación del polígono como una matriz de bitios. Esta cuarta representación no fue especificada puesto que se utiliza sólo como parte de la interfaz con el usuario.giraInic es una variable que se utiliza para conservar la orientación del tangrama en la pantalla y producir una Sensación de Continuidad en los ensayos, al usuario que está viendo el desarrollo del juego.

Como parte de los métodos estándar en Smalltalk, se incluyen una serie de operaciones lógicas entre formas. Uno de nuestros intentos fracasados, fue tratar de resolver el problema de la unión de polígonos bajo esta representación.Es posible e imprescindible construir la representación como forma partiendo de la tortuga, pero no su inversa.

Intentamos formular un algoritmo de segmentación que recorre la matriz de bitios y construye paso a paso las aristas del polígono; pero sólo obteníamos gráficas aproximadas de la imagen.No descartamos aún la posibilidad de reutilizar esta técnica, que a pesar de ser inexacta, probablemente sea mucho más eficiente en el sistema experto.

Las tres variables restantes, xCoor, yCoor y giroTan corresponden a la tríada coorInic en la especificación. Se utilizan para construír el tangrama como una nube de vértices, que no forma parte del estado del objeto; es temporal y sólo se construye para resolver el problema de la unión.

La operación SonIguales en la especificación corresponde a su complemento esdiferente:pero escencialmente el método es el mismo en ambos casos.

El prototipo de sólo dos tanes construye la unión sobre las tortugas de los tangramas hijos.esto es factible puesto que todos los tanes son polígonos conexos, y en ese caso, la instersección siempre está formada por un solo segmento:

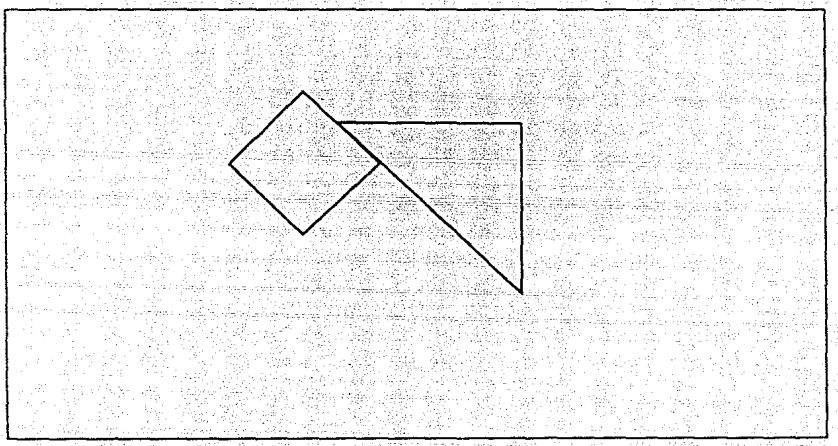


FIGURA 21: Las intersecciones de polígonos convexos involucran una sola arista de cada polígono.

El método consiste en recorrer Pieza1 hasta llegar a la arista de unión; ahí seguir por Pieza2 hasta regresar y continuar por Pieza1.En la arista de la unión se resuelven los problemas singulares de colinealidad y resta de segmentos (combinación: en la clase Tangrama del apéndice).

Nótese además, que el problema de oclusión no existe en el caso de los polígonos convexos; pero determinar la oclusión entre dos polígonos cóncavos, parece ser que es imposible fuera del plano cartesiano.

Weiler[93] afirma que los métodos para determinar unión de polígonos calculan intersecciones entre segmentos.

Las representaciones que usa Weiler, son representaciones jerárquicas (de polígonos con ciclos múltiples) que utilizan sistemas de coordenadas externas; en estas jerarquías, la unión se construye en forma inmediata; pero probar que dos contornos sean diferentes, es prácticamente imposible; construir la gráfica de tortuga también parece ser un problema bastante complejo.La gran mayoría de trabajos sobre polígonos, plantean algoritmos para polígonos convexos.

Los problemas de cuatro, cinco, seis y siete piezas son problemas de polígonos cóncavos, en donde lo único que cambia es que el espacio de búsqueda es enorme.

El método para reconstruir el polígono resultante en el plano cartesiano, es una versión anteriror al de la especificación.Analiza once casos distintos de problemas particulares.Los métodos restantes son muy similares al a especificación.

3.3 La clase Pieza.

Esta clase es el análogo a la lista de aristas en la especificación, usan sólo dos variables de instancia, una que mide la longitud de la "lista" denominada totVert y otra que es un diccionario de aristas, este diccionario se controla como si se tratara de un arreglo, las llaves de selección son los números de cada arista.

Aquí se definen los sistemas mediante cinco métodos. También es en esta clase donde se programan las operaciones que convierten la operación que calcula la matriz de bitios, y las que convierten de Tortuga a Puntos y de Puntos a Tortuga.

En estas dos últimas operaciones hubo que resolver un problema técnico, referente a una limitación absurda del lenguaje.Las operaciones con reales sólo se pueden hacer en Smalltalk, si la máquina tiene un coprocesador matemático.La mayor parte de las computadoras personales no tienen este componenete electrónico integrado.

Las funciones trigonométricas seno,coseno y arcotangente no pueden ser ejecutadas sin el coprocesador.Tuvimos que reprogramar esas funciones para adaptarlas al tipo racional, esto fue posible gracias a que el tangrama sólo posee ángulos de cuarenta y cinco grados.Por esta razón a los métodos polígono: y vertices:, que corresponden a las operaciones mencionadas, se hace un ajuste para lograr el efecto deseado.

Nos gustaría hacer otra aclaración sobre el lenguaje antes de continuar nuestra discusión.Se ha reportado en la literatura que Smalltalk es un lenguaje sin tipos ("typeless"), encontramos que esta afirmación es prácticamente cierta, a excepción de una prueba de consistencia de tipos que realiza el compilador, prohibiendo la asignación a variables que son parámetros del método, es decir, que dentro de un método no se puede cambiar el tipo de la clase asociada, a ningún parámetro.

Envía el mensaje "can´t assign to" cuando ocurre esta asignación.Incrementar un contador, o asignar un valor a cualquier variable entera, que es un parámetro del método, situación que se presenta con cierta regularidad durante la programación de casi cualquier sistema.

Nos disculpamos por la falta de haber declarado un método trapezio en vez de romboide y prometemos corregir éste en la próxima versión del simulador.

3.4 Otras clases de simulador.

La correspondencia entre las clases arista, y segmento y punto y sus tipos en la especificación es casi directa y no exige más explicación que los comentarios de los métodos.

Debido a que la clase Diccionario es sobre la cual construimos los tipos de datos estructurados, y no sobre una clase lista como se especificó; entonces tuvimos problemas para poder contender contra la estructura de estos diccionarios.Resolvimos los problemas de una manera bastante artificial, agregando una clase Circular que enriquece a la clase Dictionary.

En Circular definimos el procedimiento que regresa la clase del elemento anterior en el diccionario; el anterior al principio es el último; y también el método siguienteDe:.Programamos otros métodos auxiliares a los de reconstrucción de polígonos resultante de la unión. Esta clase deberá desaparecer y se debe incluír una nueva clase con las operaciones sobre listas definidas en la especificación.

RegiResul también es otra clase auxiliar al método de la unión, que probablemente deba desaparecer.La intersección se construye sobre un diccionario de regiResul.

La clase Integer estándar de Smalltalk, fue enriquecida con los métodos de decodificación del ensayo; y las del seno, coseno y arcotangente.

3.5 CONCLUSION DEL CAPITULO

Uno de los primeros obstáculos que ha de librar el programador novato, para poder realizar un sistema en Smalltalk, es entender como se estructura la interfaz lógica del lenguaje.El ambiente está inmerso en ventanas,"streams", despachadores, etc., que no son tarea fácil de entender.

A medida que fuimos avanzando,el ambiente se transforma y contamos con una herramienta muy útil para la programación de nuestro sistema.El despliegue gráfico de los tipos de datos, nos permite ver, en el sentido estricto de la palabra, el reflejo nítido de cada método programado; podríamos afirmar que el simulador constituye un puente, entre los objetos gráficos de Pappert y los objetos psicomotrices de Yankelevich.

Deutsch y Hayes$_{(94)}$ en su solución heurística al juego de tangramas, sitúan la discusión en un plano conceptual, explicando las estrategias generales de solución y de ahí pasan a analizar las estructuras de datos (y a definir el programa); pero no exponen ningún programa o algoritmo detallado del problema; no dicen cual es es el lenguaje de programación que utilizaron,no aclaran cual es el grado de avance de su implementación.

Hasta este momento hemos aprovechado algunas de las vantajas del lenguaje, como son la encapsulación y persistencia de los objetos, y la interfaz gráfica entre otras.No obstante, el concepto de jerarquías de clases se ve sub-utilizado en el simulador; creemos que estas cualidades serán fundamentales durante la programación de la máquina de inferencia del experto artificial.

IV. UN HIPOTETICO EXPERTO ARTIFICIAL

La imagen radiográfica de un corazón latiente en una sala de hemodinamia, se transduce y transmite hacia el interior de una computadora para su análisis. Este es un ejemplo de la aplicación que ha tenido el procesamiento digital de imágenes en la medicina actual. Pero, ¿cuáles han sido las aportaciones de la Inteligencia Artificial (IA) en el campo de las imágenes?

A finales de la década de los cincuentas, se consolida como una disciplina científica que descubre una nueva dimensión en el estudio de la imagen: deja a un lado el análisis numérico arduo y se avoca a su análisis simbólico. El intento de construir un programa que tradujera del ruso al inglés(95), fue uno de los motivos para el desarrollo de sistemas que interactúan en lenguaje natural. Eliza, desarrollado en MIT en el año de sesenta y tres, es un programa de este tipo que simula el trabajo de un psiquiatra.

Dos años después se publica Dendral, el primer sistema experto. Fue escrito para realizar análisis espectrográficos. En sesenta y nueve, Nilsson presenta un robot ambulante equipado con una cámara de televisión y detectores de choque.(95)

Gran parte de las investigaciones en Inteligencia Artificial, en una segunda etapa, se han dirigido hacia la búsqueda de representaciones del conocimiento -y de la imagen en particular- más cercanas a la realidad.

En este primer nivel de análisis de la imagen, los marcos de Minsky son uno de los casos más importantes. Se proponen como una representación de escenas visuales, factibles de ser programadas por computadora.

La imagen forma una estructura de tipo jerárquico. De esta manera la representación de un cuarto comedor, por ejemplo, está compuesta por las representaciones, en sub-marcos, de cada uno de los objetos que se encuentran en el cuarto; así mismo, la representación del objeto lámpara estará formada a su vez por otro conjunto de sub-sub-marcos.

En paralelo a este desarrollo surgen los lenguajes orientados a objetos que son una herramienta para el tratamiento de estas representaciones utilizando computadoras.

Las escenas reconstruidas pueden ser aprovechadas por programas que realicen inferencias. Es importante conservar sólo las propiedades significativas para la descripción de la escena dentro de alguna aplicación dada(97).

Patterson, expone cuáles han sido los enfoques en este segundo nivel, relacionado con la interpretación de la imagen. Ubica en un extremo los múltiples trabajos que hay sobre reconocimiento de patrones que, según el autor, son exclusivamente métodos de CLASIFICACION de los objetos dentro de una escena. En el extremo opuesto, sería deseable producir descripciones detalladas y proveer de una interpretación acerca de la formación, propósito, intensión y expectativa de los objetos en la escena; sin embargo esto queda fuera de los alcances actuales de la Inteligencia Artificial.

En un espacio intermedio, podemos ubicar a los programas que estableciendo algunas hipótesis preliminares sobre los posibles objetos partícipes en la escena, y siguiendo algún sistema de razonamiento, realizan interpretaciones plausibles de la imagen.

Un interés de la psicología ha sido conocer la capacidad intelectual de la persona para determinar, por ejemplo, cuáles son los caminos que facilitan su desarrollo.

Por otra parte, una preocupación continua en la IA es entender cómo se estructura el intelecto; y su corolario, la cuestión de si es posible programar algunas de estas funciones en una computadora.

Pruebas como la de Rorschach, que están enfocadas a establecer un perfil clínico psiquiátrico del individuo, a través de su interpretación de imágenes abstractas; quedan prácticamente fuera del campo de trabajo de la IA debido a su enorme complejidad. Pensar en computadoras que experimenten emociones es, todavía, una idea que pertenece a un futuro remoto.

Sin embargo, existen ejemplos dentro de la psicología, más alentadores para el investigador en IA. Algunas preguntas de la prueba WYSC, que se formulan en los términos de: ¿En qué se parece ... a ...?, nos remiten a un problema estudiado por matemáticos y psicólogos desde hace mucho tiempo, nos referimos a la analogía.

Trabajos recientes sobre el razonamiento analógico(98,99) muestran diversos ensayos que se refieren a este campo dentro de la IA.

Son de particular interés, los estudios sobre analogías gráficas. Dos figuras geométricas se comparan para encontrar semejanzas y diferencias, tanto en sus propiedades ópticas (figurativas), como en su construcción (operación).

Las arquitecturas de razonamiento análogico están fundamentadas en la suposición de que el hombre utiliza su experiencia anterior para la solución de problemas cotidianos: ¿En qué se parece esta nueva escena a otra conocida con anterioridad?, ¿qué conocimiento se puede extraer acerca de la nueva, por el adquirido en la anterior?; se podría formular una tercera pregunta: ¿Es posible determinar una nueva clase de escenas de la cual ambas son instancias?. Las dos primeras se refieren al razonamiento analógico, la tercera al razonamiento inductivo y a la abstracción.

La mecánica del razonamiento analógico(108) es la siguiente:

1) RECONOCIMIENTO de un problema como el análogo a otro.

2) ELABORACION de una correspondencia entre los elementos de ambos problemas y posibles inferencias en algún contexto de uso, incluyendo justificación, reparación y extensión de esta correspondencia.

3) CONSOLIDACION de los resultados de la analogía para que puedan ser reinstanciados en otros contextos.

Nuestra investigación se refiere a la construcción de un sistema experto que resuelva problemas geométricos siguiendo un razonamiento analógico.

Existe un compromiso entre la expresividad del problema seleccionado para un estudio de esta índole, y la dificultad para materializar estas ideas en un programa de computadoras que procure un análisis riguroso de estos conceptos.

La primera tarea fue escoger un paradigma de representación para estas figuras en la computadora; se decidió representar al tangrama como un "objeto",

siguiendo la pauta marcada por otros sistemas de visión por computadora que utilizan modelos jerárquicos de representación de la imagen.

El segundo paso fue la construcción de un programa que tratara al problema dentro de un espacio cerrado de búsqueda y lo resolviera ensayando todas las posibles alternativas de solución. Esta técnica se aplica en problemas de juegos como el ajedrez (en donde no se ansayan todas las posibilidades).

Se pretende dotarlo de la habilidad para responder preguntas como ¿en qué se parece un tangrama cuadrado a un tangrama romboide?, ¿qué elementos en común guardan las soluciones de ambos tangramas? para que genere o re-descubra la clase de los tangramas cuadriláteros.

Negrete[181] comenta que la abstracción está relacionada al concepto de clase. No es simplemente colocar a un objeto dentro de la clase que le corresponde, como sería el caso de los programas de reconocimiento de patrones que realizan una actividad de clasificación. La abstracción, señala el autor, consiste en generar una nueva clase donde se puedan agrupar un conjunto de objetos.

Este problema se vislumbra tan difícil como atractivo para el programador, pensamos que la creación dinámica de clases en SmallTalk (que no hemos experimentado) podrá ser muy útil durante la programación de la máquina inferencial del sistema experto.

La elaboración de este experto eventualmente servirá como un sistema de referencia para un estudio sobre el proceso de abstracción intelectual.

V. CONCLUSION GENERAL

Hemos podido observar durante la especificación formal del tangrama, cómo a este nivel fue posible modularizar la aplicación puesto que las características relevantes esenciales de los objetos en cuetión, emergen y destacan claramente, esto se debe en parte a la síntesis que lleva implícita la especificación, lo que permite al ingeniero de "software" trabajar con un lenguaje conciso para desarrollar un esquema sólido sobre el que se construye la implementación.

Durante muchos años se han desarrollado diversas técnicas de especificación que van desde las informales, hasta las especificaciones formales. Parece aún lejano el día en que los proyectos de "software" exijan la realización de especificaciones formales previas a la implementación; por otro lado, un alto porcentaje del costo final de las aplicaciones es reflejo de fallas, ambiguedades e incompletez en la especificación original.

Pensamos que el desarrollo de nuevas técnicas de especificación seguirá siendo un área muy importante de investigación en los próximos años. Coincidimos con la opinión de Ghezzi[182] en que el esfuerzo intelectual invertido al realizar especificaciones algebraicas es muy importante en la formación del ingeniero de "software".

La característica declarativa de las especificaciones formales, nos llevaría a pensar que sería conveniente utilizar un lenguaje funcional o uno de la programación lógica para la implementación de este proyecto. Esta programación sería un reflejo nítido de la especificación.

Las especificaciones más abstractas son claras y concisas; por el contario, las especificaciones de más bajo nivel son extensas y confusas, el código del programa es más entendible en la mayor parte de los casos; evitar la asignación y la secuenciación en estas operaciones durante la especificación condujo a la segmentación múltiple de axiomas que, a fin de cuentas, no es otra cosa que una secuenciación y asignación disimuladas e ineficientes.

Los algoritmos de este juego gráfico en las operaciones de bajo nivel incluyen un gran número de cálculos numéricos simples que probablemente constituyen el intríngulis de la especificación.

La mayoría de los lenguajes de programación ofrecen algunos mecanismos para aboradr el problema; LISP y PROLOG, por ejemplo, incluyen la asignación como un agregado. Por otra parte, la naturaleza híbrida de SmallTalk y en general de los lenguajes orientados a objetos, en el sentido de que son lenguajes declarativos y procedurales, pudo asimilar los beneficios de una programación sintética en los niveles de operaciones más abstractas y aprovechar, también, algunas ventajas de la programación imperativa, en los procedimientos de bajo nivel.

Después de haber realizado la especificación formal del tangrama y del juego, desde una aproximación sintética de la imagen o más específicamente de síntesis de polígonos. Podemos afirmar que el objeto tangrama puede ser entendido como un sistema formal; y desde aquí, el juego es un demostrador natural de teormeas.

En esta comparación , vemos al conjunto de tanes como nuestro conjunto de axiomas sobre el cual se aplican repetidamente las reglas de inferencia que son la unión

y la desunión para establecer nuevos teoremas que se comparan con el "postulado" del teorema a demostrar, hasta encontrar una secuencia repetida de inferencias sobre los axiomas que demuestren el teorema-tangrama.

Negrete(183), ha observado, además, la naturaleza analógica de este tipo de demostraciones. En nuestros axiomas analógicos se aplican una serie de reglas de inferencia, que no son las de la lógica, son reglas de inferencia analógica (i.e. en el plano).

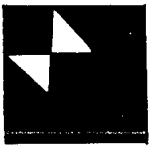
Este trabajo constituye un apoyo riguroso al de Yankelevich, específicamente sobre la utilidad pedagógica del tangrama.

La relación que existe entre la especificación y la implementación nos permite reflexionar sobre la importancia de la programación orientada a objetos como una herramienta de diseño en ingeniería de "software". El programador está obligado a desarrollar las aplicaciones en SmallTalk dentro de ese esquema de clases y subclases, está obligado a pensar en términos de los objetos, de sus propiedades y de sus funciones. A este nivel de abstracción, es donde la programación adquiere un carácter declarativo, acorde con las especificaciones descriptivas.

Yankelevich habla del tangrama como elemento de una escritura con imágenes; reportes en el campo de los lenguajes visuales(184) proponen especificaciones semiformales de lenguajes bidimensionales, la numeración maya y egipcia, por ejemplo, han sido tomadas como material de estudio en estos reportes, otros trabajos proponen también formalizaciones de otros lenguajes visuales(185).

El material de estudio es amplio y hemos analizado trabajos que van desde las epsecificaciones formales de interfaces de usuario(186), hasta investigaciones mexicanas(187), que provienen de antropólogos y semiotas sobre los lenguajes pictográficos en los códices mexicanos precolombinos.

Es sorprendente la profundidad del tangrama como objeto del conocimiento: las imágenes tangrámicas son imágenes psicomotrices que el niño se inicia operando sobre la mesa (Yankelevich) en el proceso de demostración analógica de teoremas geométricos simples (Negrete); son imágenes animadas en el interior del niño y en la computadora; son elementos gráficos de representación múltiple en el sistema nervioso central y múltiple en nuestro sistema orientado a objetos que resuelve tangramas de tres piezas; son un sistema formal en el niño y en la especificación algebraica que nosostros hacemos; y probablemente sean, también, instancias de clases dinámicamente generadas (abstracciones) en el niño y en un experto artificial en la solución de tangramas.



BIBLIOGRAFIA

BIBLIOGRAFIA

1. Yankelevich, Guillermina.- "La cifra en el lenguaje de las imágenes".- VI Congreso Mexicano de Psicología.- México,1990.

2. Arnheim, Rudolf.- El pensamiento visual.-EUDEBA, cuarta edición en español.- Argentina, 1985.- 343 pp.

3. Chang, Shi-Kuo, editor.- Visual Languages and Visual Programming.- Plenum Press.- EUA,1990.- 340 pp.

4. Hégron, Gerard.- Image Synthesis.- MIT press.- EUA,1988.- 216 pp.

5. Weiler, Kevin.- "Polygon Comparison Using a Graph Representation".- Computer Graphics.- ACM press.- vol. 14, núm. 3.- EUA, julio, 1980.- pp. 10-19.

6. Oktaba, Hanna.- Curso sobre estructuras de datos.- IIMAS, UNAM.- México, 1989.

7. Mallgren, William.- Formal Specification of Interactive Graphics Programming Languages.- MIT press, ACM.- Inglaterra, 1983.- 265 pp.

8. Negrete, José.- "Demostración analógica en Inteligencia Artificial".- Primer Coloquio de Ciencias Cognitivas.- agosto,1990.- Sao Paulo, Brasil.

9. Deutsch, E.S. y Hayes, K.C.- "A Heuristic Solution to the Tangram Puzzle".- Machine Intelligence.- Edinburgh Univertisty Press.- vol. 7 .- Inglaterra, 1972.- pp. 205-240.

10. Goldberg, Adele y Robson, David.-Smalltalk 80. The Langaae and its Implementation.- Addison Wesley.- primera edición.- EUA,1983.- p. viii (prefacio).

11. Dijkstra, Edgster W.-"The Humble Programmer" .- Communications of the ACM.- vol.15.- USA, 1972.- P.12

12. Hall, Rogers P.- "Computational Approaches to Analogical Reasoning: A Comparative Analysis".-Artificial Intelligence.- Elsevier Science Publisher.-vol. 39,mayo.-Holanda,1989.- p. 40

13. Tsichritzis, D.G.-"Directions in Object Oriented Research".- Object Oriented Concepts,Databases, and Applications.- editor King, Nom.- ACM press.- EUA,1989 .- p.526

14. Oktaba, Hanna y Berber, René.- "Crafting Reusable Software in Modula-2" .- BYTE .- Vol. 12, núm. 10,sept.- EUA,87.- pp.123-128

15. Krasner, Glenn.- SmallTalk 80. Bits of History Words of Advice.- Addison Wesley.- EUA, 1983.- p. iii(prefacio).

16. Negrete, José.- "Sabios artificiales".-Información Científica y Tecnológica.-CONACYT.- vol. 5, núm. 109, oct. 85.- México, 1985.- p. 22

17. Lara, Rolando.- Cibernética del cerebro.-LIMUSA.- México,1987.

18. Ellis, Clarena y Gibbs, Simon.-"Active Objects: Realities and Possibilities".- Object Oriented Concepts, Databases, and Applications.-Editado por Kim, Won et.al.- ACM press.- EUA, 1989.- pp. 561-572.

19. Minsky, Marvin.- "A Framework for Representing Knowledge".- The Psichology of Computer Vision.- Mc. Graw Hill.- EUA, 1975.- pp. 211-277

20. Bic, Lubomir y Gilbert.-"Learning from AI: New Trends in Database Technology".- Computer.- oct,86.- p. 48

21. Oktaba, Hanna.- Curso sobre Lenguajes de Programación.- comunicación personal.- IIMAS,UNAM.- México,1990

22. Wegner, Peter.- "Learning the Language".-Byte.- vol. 14, núm. 3, mar. 89.- EUA.- p. 247.

23. King, Roger.- "My Cat is Object Oriented".- Object Oriented Concepts, Databases, and Applications.-Editado por Kim, Won et.al.- ACM press.- EUA, 1989.- pp. 24-28.

24. Ghezzi,Carlo y Mehdi,Jazayeri.- Programming Language Concepts 2/E.- John Willey & Sons.- EUA, 1982.-p. 26

25. Dijkstra, Edgser.- "GO TO Statement Considered Harmful".- Communications of the ACM.- vol. 11, núm. 3, marzo.- EUA,1968.- pp. 146-148

26. LaLonde,Wilf y Pugh,John.-Inside SmallTalk.- Prentice Hall.- vol. 1.- EUA, 1990.- p. 57

27. Ibidem.- p. 35

28. Ibidem.- p. 39

29. Cardelli, Luca y Wegner,Peter.- "On Understanding Types, Data Abstraction, and Polimorphism".- ACM Computing Surveys.- Vol. 17, Núm. 4, diciembre.- EUA,1985.- p. 474

30. Idem.

31. Zilles, Stephen.- "Types, Algebras, and Modelling".-On conceptual Modelling.- ???.- p. 442

32. Wegner, Peter.- "The Object Oriented Classification Paradigm".- Research Directions in Object Oriented Programming.- MIT Press, Series in Computer Systems.- EUA,1987.- p. 492

33. Zilles, Stephen.- "Types, Algebras, and Modelling".- op. cit.- p. 447

34. Idem.

35. Baase, Sara.- Computer Algorithms. Introduction to Design and Analysis.- segunda edición.- Addison Wesley.- EUA,1988.-p.157

36. Ibidem.- p. 232

37. Rich, Elaine.- Artificial Intelligence.- Mc. Graw Hill.- EUA, 1983.- p. 78

38. Negrete José.- Inteligencia Aunque Sea Artificial.- Editorial Limusa.- México, 1990 .- p. ???

39. Cardelli, Luca y Wegner, Peter.- "On Understanding Types, Data Abstraction, and Polimorphism".- op. cit.- pp. 475-479

40. Zilles, Stephen.- "Types, Algebras, and Modelling".- op. cit.- p. 507

41. Idem.

42. Oktaba, Hanna.- Curso sobre Lenguajes de Programación.- comunicación personal.- IIMAS,UNAM.- México,1990.

43. Oktaba, Hanna y Berber, René.- "Crafting Reusable Software in Modula-2" .- op. cit.- pp.123-128

44. Snyder, Alan.- "Inheritance and the Development of Encapsulated Software Components".- Research Directions in Object Oriented Programming.- MIT Press.- editores Shriver, Bruce y Wegner, Peter.- EUA,1987.- p.168

45. Ibidem.- p. 172.

46. Ibidem.- p. 177.

47. Drossopoulou, Sophia, et. al.- Module and Type Systems.- Imperial College of Science and Technology.- Inglaterra, febrero, 1988.- p.4

48. Negrete, José.- Inteligencia Experimental en Computadoras.- Limusa.- México, 1988.- p.47

49. Ghezzi, Carlo y Mehdi, Jazayeri.- Programming Language Concepts 2/E.- op. cit.- p. 202

50. Ibidem.- p. 205

51. Oktaba, Hanna.- "Programación Concurrente (primera parte)".- Comunicaciones técnicas.- IIMAS, UNAM.- no. 86, serie azul.- México, 1985

52. Ibidem.- p. 13

53. Ibidem.- p. 31

54. Ghezzi,Carlo y Mehdi,Jazayeri.- Programming Language Concepts 2/E.- op. cit.- p. 205

55. Steigerwald, Robert y Nelson, Michael.- "Concurrent Programming in Smalltalk-80".- Sigplan Notices.- vol. 25, núm. 8, agosto.- EUA,1990.- pp. 27-36

56. Oktaba, Hanna.- "Programación Concurrente (segunda parte)".- Comunicaciones técnicas.- IIMAS, UNAM.- no. 100, serie azul.- México, 1987.- p. 1

57. Ibidem.- p. 2

58. Idem.

59. Ghezzi,Carlo y Mehdi,Jazayeri.- Programming Language Concepts 2/E.- op. cit.- p. 208

60. Oktaba, Hanna.- Curso sobre Lenguajes de Programación.- comunicación personal.- IIMAS,UNAM.- México,1990.

61. Hopkins, T.P. y Wolczko, M.I.- "Writing Concurrent Oject-Oriented Programs Using Smalltalk-80".-The Computer Journal.- Cambridge University Press.- vol. 32, núm. 4.- Inglaterra, 1989.- p. 342

62. Ibidem.- pp. 341-350

63. Ellis, Clarena y Gibbs, Simon.-"Active Objects: Realities and Possibilities".- op. cti.- p. 562

64. Ibidem.- p. 563

65. Agha, Gul y Hewitt, Carl.- "Actors: A Conceptual Foundation for Concurrent Object-Oriented Programming".- Research Directions in Object Oriented Programming.- MIT Press, Series in Computer Systems.- EUA, 1987.- p. 56 ???

66. Ibidem.- p. 70

67. Ibidem.- p. 55

68. Ellis, Clarena y Gibbs, Simon.-"Active Objects: Realities and Possibilities".- op. cti.- p. 566

69. Ibidem.- p. 571.

70. Dijkstra, Edgser.- "GO TO Statement Considered Harmful".- op. cit.- pp. 146-148.

71. Oktaba, Hanna.- Curso sobre Lenguajes de Programación.- comunicación personal.- IIMAS, UNAM.- México, 1990

72. Shaw, Mary.- "Larger Scale Systems Require Higher-Level Abstractions".- ACM, ???.- EUA, 1989.- p. 144.

73. Clarisse, Olivier y Shi-Kuo Chang.- "VICON. A Visual Icon Manager".- Visual Languages.- Shi-Kuo Chang, editor.- Plenum press.- EUA, 1986.- p. 153.

74. Ghezzi, Carlo.- Software Engineering.- Primera Escuela Internacional de Invierno.- Morelia, México, 1990.- p. 9

75. Oktaba, Hanna.- Curso sobre estructuras de datos.- IIMAS, UNAM.- México, 1989.

76. Mallgren, William.- op. cit.

77. Horebeck, Ivo Van, et. al.-Algebraic Specifications in Software Engineering.- Springer Verlag.- EUA, 1989.

78. Mallgren, William.- op. cit.- p. 205

79. Oktaba, Hanna.- Curso sobre estructuras de datos.- IIMAS, UNAM.- México, 1989.

80. Hégron, Gerard.- op. cit.- pp. 185-186

81. Sin autor.- SMALLTALK V.- DigiTalk Inc.- EUA, 1986

82. Weiler, Kevin.- op. cit.- p. 12

83. Hégron, Gerard.- op. cit.- pp. 185-186

84. McHugh, James.- Algorithmic Graph Theory.- Prentice Hall.- Inglaterra, 1990.- 327 pp.

85. Harary, Frank.- Graph Theory.- Addison Wesley.- EUA, 1969.- 274 pp.

86. Flament, Claude.- Teoría de Grafos y Estructuras de Grupo.- Editorial Tecnos.- España, 1963.- 144 pp.

87. Korfhage, Robert.- Discrete Computational Structures.- Academic Press.- colección Computer Science and Applied Mathematics.- EUA, 1974.- 381 pp.

88. McHugh, James.- Ibidem.- pp. 1-3

89. Velarde, Carlos.- Curso sobre matemáticas finitas.- IIMAS, UNAM.- México, 1989.

90. Abelson, Harold y DiSessa, Andrea.- Turtle Geometry.- MIT press, series in Artificial Intelligence.- EUA, 1980.- p. 13.

91. Ibidem.- p.14

92. Fairley, Richard.- Ingeniería de Software.- Mc. Graw Hill.- México, 1985.- pp.54-55

93. Weiler, Kevin.- op. cit.- p. 12

94. Deutsch, E.S. y Hayes, K.C.- op. cit.

95. Oktaba, Hanna.- Curso sobre Lenguajes de Programación.- comunicación personal.- IIMAS,UNAM.- México,1990

96. Ward,Patrick.- "Historia de la Inteligencia Artificial".- material fotográfico.- sin publicar.

97. Patterson, Dawn W.- Introduction to Artificial Intelligence and Expert Systems.- Prentice Hall.- EUA,1990.- p. 314

98. Negrete, José.- "Demostración Analógica en Inteligencia Artificial".- Primer Coloquio sobre Ciencias Cognitivas, agosto, 1990. Sao Paulo, Brasil.

99. Hall, Rogers.- "Computacional Approaches to Analogial Reasoning: a Comparative Analysis".- Artificial Intelligence.- Elsevier Science Publishers B. V.- Holanda, 1989.- pp. 39-120.

100. Ibidem.- p. 43

101. Negrete, José.- comunicación personal.- Laboratorio de BioMatemáticas.- IIB, UNAM.- México, 1990

102. Ghezzi, Carlo.-Conferencia sobre Ingeniería de Software.- comunicación personal.- Primera Escuela Internacional de Invierno.- Morelia, México, 1990.- p. 9

103. Negrete, José.- comunicación personal.- Laboratorio de BioMatemáticas.- IIB, UNAM.- México, 1990

104. Kapolka III, Anthony.- "A Study in Natural Visual Language, Numbers and Dates".- p. 45-69.

105. Mallgren, William.- op. cit.

106. Harrison, M y Himbley H, editores.- Formal Methods in Human-Computer Interaction.- Cambridge Series in Human-Computer Interaction.- ...

107. Mohar, Luz.- La escritura en el México Antiguo.- Plaza Valdés, editores.- México, 1990.- 357 pp.

108. Pascual, Arturo.- Iconografía Arqueológica del Tajín.- Fondo de Cultura Económica.- México, 1990.- 318 pp.

```
Object subclass: #Quetzalc
  instanceVariableNames:
    'pieza1 pieza2 pieza3 tangProblema strEntrada apuVenTexto apuVenPiezas apuVenSolucic
lema venPrincipal tangSolucion streamTexto tipoJuego '
  classVariableNames: ''
  poolDictionaries:
    'CharacterConstants ' !

!Quetzalc class methods !

nueva
   "Regresa una pieza,tangrama o uni"n nueva"
^super new inicializa!

nuevo
   "Regresa una pieza,tangrama o uni"n nueva"
^super new inicializa! !


!Quetzalc methods !

abreCon: unString
   "Abre e inicializa la ventana del medio ambiente"
   strEntrada := unString.
   venPrincipal := TopPane new label: 'Q u e t z a l c " a t l'.
   venPrincipal
      addSubpane:
         (apuVenTexto := TextPane new  menu: #menuPrincipal;
                           model: self;
                           name: #iniVenTexto;
                           framingRatio: ( 0 @ 0 extent: 1/2 @ (1/2))  );
      addSubpane:
         (apuVenSolucion := GraphPane new  model: self;
                              name: #iniVenSolucion:;
                              framingRatio: ( 0 @ (1/2) extent: 1/2 @ (1/2))   );
      addSubpane:
         (apuVenPiezas := GraphPane new  model: self;
                           name: #iniVenPiezas:;
                           framingRatio: ( 1/2 @ 0 extent: ((1/2) @ (1/2)))   );
      addSubpane:
         (apuVenProblema := GraphPane new  model: self;
                              name: #iniVenProblema:;
                              framingRatio: ( 1/2 @ (1/2) extent: (1/2 @ (1/2)))  ).
   venPrincipal reframe: Display boundingBox .
   streamTexto := apuVenTexto dispatcher.
^venPrincipal dispatcher openWindow scheduleWindow!

comenzamos
   "Mtodo principal, que prepara el medio ambiente"
^Quetzalc nuevo abreCon:
'Soluci"n de un tangrama de dos piezas
convexas, sin reflexi"n, y sin contacto
por los vrtices'!
```

```
fin
   "Cierra la ventana principal"
   venPrincipal close.
^nil!

inicializa
   "inicializa todas las variables de instancia de Quetzalc"atl"
   |unTangrama otroTangrama|
   CursorManager execute change.
   pieza1 := Pieza nueva trianguloChico.
   pieza2 := Pieza nueva cuadrado.

   unTangrama := Tangrama nuevo.
   otroTangrama := Tangrama nuevo.
   unTangrama piezaTang: pieza1.
   otroTangrama piezaTang: pieza2.
   tangProblema := Tangrama nuevo combinacion: 1 de: unTangrama
      y: otroTangrama y: streamTexto y: apuVenSolucion.
   tipoJuego = 2.
   tangSolucion := Tangrama nuevo.
   strEntrada := ''.
   CursorManager normal change.
^self!

iniGraf: unRect
   "inicializa el fondo de las subventanas de gr ficas
   (manual Smalltalk V p.130)"
   |unaForma|
   unaForma := Form width: unRect width height: unRect height.
   unaForma displayAt: unRect origin.
^unaForma!

iniVenPiezas: otroRect
   "Inicializa la subventana de las piezas elegidas"
   |unaForma otraForma unRect|
   CursorManager execute change.
   self iniGraf: otroRect.
   unRect := apuVenPiezas frame.
   unaForma  := pieza1 creaForma: (unRect expandBy: (1//2)) giro: 0.
   otraForma := Form width: unRect width height: unRect height.
   otraForma copy: (Rectangle origin: (0 @ 0 ) extent: unaForma extent)
             from: unaForma to: (0 @ 0) rule: Form andRule.
   unaForma := pieza2 creaForma: (unRect expandBy: (1//2)) giro: 0.
   otraForma copy: (Rectangle origin: (0 @ 0 ) extent: unaForma extent)
     from: unaForma to: (unRect width // 2 @ 0) rule: Form andRule.
   (tipoJuego = 3) ifTrue:
      [unaForma := pieza3 creaForma: (unRect expandBy: (1//2)) giro: 0.
      otraForma copy: (Rectangle origin: (0 @ 0 ) extent: unaForma extent)
        from: unaForma to: ( 0 @ (unRect height // 2)) rule: Form andRule].
  apuVenPiezas form: otraForma.
```

```
    apuVenPiezas showWindow.
    apuVenPiezas update.
    CursorManager normal change.
^unaForma!

iniVenProblema: unRect
    "Inicializa la subventana del problema seleccionado"
    |unaForma  otroRect escalax escalay |
    CursorManager execute change.
    self iniGraf: unRect.
    unaForma := tangProblema formaTang.
    "otroRect := apuVenProblema frame.
    unaForma := tangProblema piezaTang creaForma: otroRect."
    "(otroRect width > otroRect height) ifTrue:
       [escalay := otroRect height // unaForma boundingBox height.
        escalax := otroRect height // unaForma boundingBox width]
    ifFalse:
       [escalay := otroRect width // unaForma boundingBox height.
        escalax := otroRect width // unaForma boundingBox width]."

  " unaForma magnify: unaForma boundingBox by: (escalax @ escalay). "
    apuVenProblema form: unaForma.
    apuVenProblema showWindow.
    apuVenProblema update.
    CursorManager normal change.
^unaForma!

iniVenSolucion: unRect
    "Inicializa la subventana de la solucion encontrada"
    |unaForma otraForma otroRect|
    CursorManager execute change.
    self iniGraf: unRect.
    "otroRect := apuVenSolucion frame.
    unaForma := tangSolucion piezaTang creaForma: otroRect.
    unaForma offset: ((otroRect width/2) @ (otroRect height/2)).
    otraForma := Form width: otroRect width height: otroRect height."
    unaForma := tangSolucion formaTang.
    apuVenSolucion form: unaForma.
    apuVenSolucion showWindow.
    apuVenSolucion update.
    CursorManager normal change.
^unaForma!

iniVenTexto
    "Escribe la bienvenida en la ventana de texto"
^strEntrada!

juega
    "Resuelve el tangrama problema con las piezas seleccionadas"
    |unTangrama otroTangrama tercerTangrama|
    CursorManager execute change.
```

```
    unTangrama := Tangrama nuevo.
    otroTangrama := Tangrama nuevo.
    unTangrama piezaTang: pieza1.
    otroTangrama piezaTang: pieza2.
    (tipoJuego = 3) ifTrue:
        [tercerTangrama := Tangrama nuevo.
        tercerTangrama piezaTang: pieza3.
        self resuelveCon: unTangrama y: otroTangrama y: tercerTangrama]
    ifFalse:
        [self resuelveCon: unTangrama y: otroTangrama].
    CursorManager normal change.
^tangSolucion!

menuPrincipal
    "Responde con un 'pane menu' de la subventana de control en la
     ventana principal"
^Menu
    labels: ('cu ntasPiezas\piezas\problema\jugar\prueba\salir') withCrs
    lines: #(2)
    selectors: #(
        tipoDeJuego selePiezas seleProblema juega prueba fin)!

prueba

 "metodo temporal para probar poligono"
 |unTangrama unaPieza otraPieza dicVert|
 otraPieza := Pieza nueva.
 unTangrama := Tangrama nuevo.
 unaPieza := Pieza nueva prueba3.
 otraPieza dicAristas: unaPieza dicAristas.
 1 to: unaPieza totVert do:
    [:i | (otraPieza dicAristas at: i ) longArista printOn: streamTexto.
     (otraPieza dicAristas at: i ) anguloSalida printOn: streamTexto.
     Lf printOn: streamTexto. ].
 "dicVert := unaPieza vertices: -45 y: -6 y: -8/(7071/10000) y: streamTexto.
 otraPieza poligono: dicVert y: (unaPieza totVert) y: streamTexto.  "!

pruebaPoligono

 "metodo temporal para probar poligono"
 |unTangrama unaPieza otraPieza dicVert unPunto|
 otraPieza := Pieza nueva.
 unTangrama := Tangrama nuevo.
 unaPieza := Pieza nueva prueba3.
 unPunto := Point new.
 dicVert := unaPieza vertices: -45 y: -6 y: -8/(7071/10000) y: unPunto y: streamTexto.
 otraPieza poligono: dicVert y: (unaPieza totVert) y: streamTexto.!

pruebaVertices

 "metodo temporal para probar vrtices"
```

```
 !unTangrama unaPieza !

 unTangrama := Tangrama nuevo.
 unaPieza := Pieza nueva prueba1.
 unTangrama pieza1: unaPieza.
 unaPieza := Pieza nueva prueba3.
 unTangrama pieza2: unaPieza.
 unTangrama puntoUnion1: 20.
 unTangrama puntoUnion2: 8.
 unaPieza := Pieza nueva prueba3.
 unTangrama piezaTang: unaPieza.
 unTangrama iniCoor: streamTexto y: apuVenSolucion.
 "unTangrama vertices: streamTexto."
 unTangrama pieza2 vertices: (unTangrama giroInic) y: (unTangrama xCoor)
                                        y: (unTangrama yCoor) y: streamTexto.!

resuelveCon:  unTangrama y: otroTangrama
   "resuelve el problema de un Tangrama a partir de otros dos
    actualiza el tangrama soluci"n, va mostrando en la pantalla
    cada uno de los ensayos"
   !i ensayo ultimaCombinacion!
   i := 1.
   ultimaCombinacion :=  unTangrama piezaTang totVert * (otroTangrama
   piezaTang totVert)*9.
   'ensayo numero ' printOn: streamTexto.
   Lf printOn: streamTexto.
   ensayo := Tangrama nuevo union: 1 de: unTangrama y: otroTangrama
      y: streamTexto y: apuVenSolucion.
   tangSolucion := ensayo.
   self iniVenSolucion: apuVenSolucion frame.
   [(ensayo esDiferenteA: tangProblema y: streamTexto) and: [i  <  ultimaCombinacion]]
      whileTrue: [ i := i + 1.
      ensayo union: i de: unTangrama y: otroTangrama y: streamTexto y: apuVenSolucion.
      tangSolucion := ensayo.
      self iniVenSolucion: apuVenSolucion frame. ].

   i < ultimaCombinacion  ifFalse:
      [ ensayo := Tangrama nuevo.
      tangSolucion := ensayo.
      self iniVenSolucion: apuVenSolucion frame].
   Lf printOn: streamTexto. 'fin' printOn: streamTexto.
^tangSolucion!

resuelveCon:  unTangrama y: otroTangrama y: tercerTangrama
   "resuelve el problema de un Tangrama a partir de otros tres
    actualiza el tangrama soluci"n, va mostrando en la pantalla
    cada uno de los ensayos"
   !i j ensayo ultComb1 ultComb2 cuartoTangrama!
   i := 0.
   ultComb1 := unTangrama piezaTang totVert *
               (otroTangrama piezaTang totVert) *
```

```
            (tercerTangrama piezaTang totVert) *  27.
   ultComb2 := (otroTangrama piezaTang totVert) *
               (tercerTangrama piezaTang totVert) *  9.
ensayo := Tangrama nuevo.

   [i  <  ultComb1] whileTrue:
      [ i := i + 1.
      ensayo union: i de: unTangrama y: otroTangrama y: streamTexto y: apuVenSolucion.
      j:=0.
      cuartoTangrama := Tangrama nuevo.
      [( cuartoTangrama esDiferenteA: tangProblema y: streamTexto) and:
      [j  < ultComb2 ]]
      whileTrue: [ j := j + 1.
         cuartoTangrama union: j de: ensayo y: tercerTangrama y:
         streamTexto y: apuVenSolucion.
         tangSolucion := cuartoTangrama.
         self iniVenSolucion: apuVenSolucion frame
      ]. (j < ultComb2) ifTrue: [i := ultComb1 + 999].
      ].

   (i = (ultComb1 + 999)) ifFalse:
      [ ensayo := Tangrama nuevo.
      tangSolucion := ensayo.
      self iniVenSolucion: apuVenSolucion frame].
   Lf printOn: streamTexto. 'fin' printOn: streamTexto.
^tangSolucion!

selePiezas
   "Pregunta al usuario cu les son las piezas del tangrama y
    actualiza las variables de instancia respectivas, tambin la
    imagen en la ventana"
    | unString strPieza1 strPieza2 strPieza3|
    strPieza1 := Prompter promptWithBlanks: '(Con cu l pieza?'
                 default: ''.
    unString := strPieza1 asUpperCase trimBlanks.
    [unString = 'TRIANGULO CHICO'  or:
       [unString = 'TRIANGULO MEDIANO' or:
       [unString = 'TRIANGULO GRANDE' or:
       [unString = 'TRAPEZIO' or:
       [unString = 'CUADRADO' ]]]]]
          whileFalse: [strPieza1 := Prompter promptWithBlanks: '(Con cu l?'
                                    default: ''.
             unString := strPieza1 asUpperCase trimBlanks].
    strPieza1 := unString.
    strPieza2 := Prompter promptWithBlanks: '(Con cu l otra pieza?'
                 default: ''.
    unString := strPieza2 asUpperCase trimBlanks.
    [unString = 'TRIANGULO CHICO'  or:
    [unString = 'TRIANGULO MEDIANO' or:
    [unString = 'TRIANGULO GRANDE' or:
    [unString = 'TRAPEZIO' or:
.pa [unString = 'CUADRADO' ]]]]] whileFalse:



      [strPieza2 := Prompter promptWithBlanks: '(Con cu l?'
```

```
    unString := strPieza3 asUpperCase trimBlanks.
    [unString = 'TRIANGULO CHICO' or:
    [unString = 'TRIANGULO MEDIANO' or:
    [unString = 'TRIANGULO GRANDE' or:
    [unString = 'TRAPEZIO' or:
    [unString = 'CUADRADO' ]]]]] whileFalse:
       [strPieza3 := Prompter promptWithBlanks: '(Con cu l?'
       default: strPieza3.
       unString := strPieza3 asUpperCase trimBlanks].
    strPieza3 := unString].

    (strPieza1='TRIANGULO CHICO' ) ifTrue: [pieza1:=Pieza nueva trianguloChico].
    (strPieza1='TRIANGULO MEDIANO') ifTrue: [pieza1:=Pieza nueva trianguloMediano].
    (strPieza1='TRIANGULO GRANDE' ) ifTrue: [pieza1:=Pieza nueva trianguloGrande].
    (strPieza1='TRAPEZIO'         ) ifTrue:  [pieza1:=Pieza nueva trapezio].
    (strPieza1='CUADRADO'         ) ifTrue:  [pieza1:=Pieza nueva cuadrado].

    (strPieza2='TRIANGULO CHICO' ) ifTrue: [pieza2:=Pieza nueva trianguloChico].
    (strPieza2='TRIANGULO MEDIANO') ifTrue: [pieza2:=Pieza nueva trianguloMediano].
    (strPieza2='TRIANGULO GRANDE' ) ifTrue: [pieza2:=Pieza nueva trianguloGrande].
    (strPieza2='TRAPEZIO'         ) ifTrue:  [pieza2:=Pieza nueva trapezio].
    (strPieza2='CUADRADO'         ) ifTrue:  [pieza2:=Pieza nueva cuadrado].

    (tipoJuego = 3) ifTrue:
    [(strPieza3='TRIANGULO CHICO' ) ifTrue: [pieza3:=Pieza nueva trianguloChico].
    (strPieza3='TRIANGULO MEDIANO') ifTrue:  [pieza3:=Pieza nueva trianguloMediano].
    (strPieza3='TRIANGULO GRANDE' ) ifTrue:  [pieza3:=Pieza nueva trianguloGrande].
    (strPieza3='TRAPEZIO'         ) ifTrue:   [pieza3:=Pieza nueva trapezio].
    (strPieza3='CUADRADO'         ) ifTrue:   [pieza3:=Pieza nueva cuadrado]].

    CursorManager execute change.
    self iniVenPiezas: apuVenPiezas frame.
    CursorManager normal change.
^self!

seleProblema
    "Pregunta al usuario cu l es el n#mero de problema y actualiza:
    la variable tangProblema y, tambin, la  imagen en la ventana"
    !unString unEntero otroEntero tercerEntero numMaximo unTangrama
    otroTangrama tercerTangrama!
    numMaximo := pieza1 totVert * 3.
    unString := Prompter prompt: '(Cu l es el punto de uni"n de la pieza 1?'
                default: ''.
.paunEntero := unString asInteger.

    [numMaximo < unEntero] whileTrue:
       [ unString := Prompter prompt: unString , ' es demasiado grande. Dame otro punto.
       default: ''.
       unEntero := unString asInteger].
    numMaximo := pieza2 totVert * 3.
    unString := Prompter prompt: '(Cu l es el punto de uni"n de la pieza 2?'
                default: ''.
    otroEntero := unString asInteger.
    [numMaximo < otroEntero] whileTrue:
       [ unString := Prompter prompt: unString , ' es demasiado grande. Dame otro punto.
       default: ''.
       otroEntero := unString asInteger].
    (tipoJuego = 3) ifTrue:
       [ unString := Prompter prompt: '(Cu l es el numero de combinaci"n con  la pieza 3'
                default: ''
```

```
    'problema numero ' printOn: streamTexto.
    LF printOn: streamTexto.
    otroEntero := (unEntero - 1) * (pieza2 totVert * 3) + otroEntero.
    tercerTangrama := Tangrama nuevo.
    tangProblema := tercerTangrama union: otroEntero
                        de: unTangrama y: otroTangrama y: streamTexto y: apuVenSolucion.
    (tipoJuego = 3) ifTrue:
      [tercerTangrama := Tangrama nuevo.
      tercerTangrama piezaTang: pieza3.
      tangProblema :=  Tangrama nuevo union: tercerEntero
                    de: tangProblema  y: tercerTangrama y: streamTexto
                    y: apuVenSolucion].
    "apuVenProblema update."
    self iniVenProblema: apuVenProblema frame.
    CursorManager normal change.
^self!
    |
tipoDeJuego
    "Seleccion el numero de piezas con el que se desea jugar"
    |numMaximo unEntero unString|
    numMaximo := 3.
    unEntero := 0.
    [(unEntero <= 1) or: [unEntero > 3]] whileTrue:
      [unString := Prompter prompt: 'Con cu ntas piezas jugamos?'
                    default: '3'.
       unEntero := unString asInteger].
    tipoJuego := unEntero.
^tipoJuego! !
```

```
Quetzalc subclass: #Tangrama
  instanceVariableNames:
    'pieza1 pieza2 puntoUnion1 puntoUnion2 piezaTang formaTang giroInic xCoor yCoor gir
  classVariableNames: ''
  poolDictionaries:
    'CharacterConstants ' !

!Tangrama class methods ! !


!Tangrama methods !

auxCombin2: segmento1A y: segmento2A y: unaArista y: arista1 y: arista2
y: unPunto y: numArista1 y: numArista2 y: segmento1 y: segmento2
   |i siguiente otraArista unAngulo otroAngulo|
      i := unPunto x .
      siguiente := unPunto y.
       segmento1A = segmento2A ifTrue:
            [unAngulo :=(piezaTang dicAristas at: ( i  -  1 )) anguloSalida.
            otroAngulo := arista2 anguloSalida.
            ((unAngulo \\ 90 = 0) and: [ otroAngulo \\ 90 = 0])  ifTrue:
               [numArista1 = pieza1 totVert ifTrue:
                  [otraArista := Arista nueva.
                  otraArista longArista: (( (piezaTang dicAristas at: 1)
                  longArista) +  ((piezaTang dicAristas at: (i - 1))
                  longArista)   ).
                  otraArista anguloSalida: (piezaTang dicAristas at: 1)
                  anguloSalida.
                  piezaTang dicAristas at: 1 put: otraArista.
                  piezaTang dicAristas removeKey: (i - 1 ).
                  i := i  -  1.  ]
               ifFalse:
                  [otraArista := Arista nueva.
                  otraArista longArista: arista1 longArista.
                  otraArista anguloSalida: ((pieza2 dicAristas at: numArista2)
                  anguloSalida + arista1 anguloSalida - 180).
                  piezaTang dicAristas at: (i - 1) put: otraArista.
                  siguiente := numArista1 + 2]. ]
            ifFalse:
               [otraArista := Arista nueva.
               otraArista anguloSalida:  (((piezaTang dicAristas at: (i - 1))
               anguloSalida) + ((arista1 anguloSalida)  -  180)  ).
               otraArista longArista: (piezaTang dicAristas at: (i - 1))  longArista.
               piezaTang dicAristas at: (i - 1) put: otraArista.
               siguiente := numArista1 + 1].  ].

      segmento1A < segmento2A ifTrue:
         [otraArista := Arista nueva.
         otraArista longArista: (segmento2A  -  segmento1A).
         otraArista anguloSalida: (arista1 anguloSalida  -  180).
         piezaTang dicAristas at: i put: otraArista.
         i := i + 1.
         siguiente := numArista1 + 1.    ]
```

```
    "Pasando la union, sobre pieza1"
       [siguiente > (pieza1 totVert)] whileFalse:
          [piezaTang dicAristas at: i put: (pieza1 dicAristas at: siguiente).
          i := i + 1.
          siguiente := siguiente + 1].
       (numArista1 = 1 and: [ segmento1 = segmento2 ]) ifTrue:
          [arista2 anguloSalida = ((piezaTang dicAristas at: (i - 1))
          anguloSalida) ifTrue:
            " [unaArista anguloSalida: ((pieza1 dicAristas at: (numArista1+1))
             anguloSalida).
             unaArista longArista: (((pieza1 dicAristas at: (numArista1+1))
             longArista) + ((piezaTang dicAristas at: (i - 1)) longArista)  ).
             piezaTang dicAristas at:  1 put: unaArista.]"
            [unaArista anguloSalida: (piezaTang dicAristas at: 1)  anguloSalida.
            unaArista longArista: ((piezaTang dicAristas at: 1) longArista +
            (piezaTang dicAristas at: (i - 1)) longArista).
            piezaTang dicAristas at: 1 put: unaArista.
            piezaTang dicAristas removeKey: (i - 1).
            i := i - 1]
          ifFalse:
             [otraArista := Arista nueva.
             otraArista anguloSalida: (((piezaTang dicAristas at: (i - 1))
             anguloSalida) + ((arista2 anguloSalida)  -  180)  ).
             otraArista longArista: (piezaTang dicAristas at: (i - 1))
             longArista.
             piezaTang dicAristas at: (i - 1) put: otraArista.  ].  ].
     unPunto x: i.
      unPunto y: siguiente.
^self!

auxCombina: segmento1 y: segmento2 y: unaArista y: arista1 y: arista2
y: unPunto y: numArista1 y: numArista2
   |i siguiente otraArista unAngulo otroAngulo|
      i := unPunto x.
      siguiente := unPunto y.
      segmento1 > segmento2 ifTrue:
         [unaArista longArista: (segmento1  -  segmento2).
         unaArista anguloSalida: (arista2 anguloSalida  -  180).
         piezaTang dicAristas at: i put: unaArista.
         i := i + 1   ].

      segmento1 = segmento2 ifTrue:
         [numArista1 > 1 ifTrue:
            [unAngulo :=(piezaTang dicAristas at: ( i  -  1 )) anguloSalida.
            otroAngulo := arista2 anguloSalida.
            ((unAngulo \\ 90 = 0) and: [ otroAngulo \\ 90 = 0])
            ifTrue:
               [unaArista longArista: (((pieza2 dicAristas at: siguiente)
               longArista) - ((piezaTang dicAristas at: (i - 1)) longArista)  ).
               unaArista anguloSalida: ((pieza2 dicAristas at: siguiente)
```

```
                anguloSalida).
                piezaTang dicAristas at: (i - 1) put: unaArista.
                siguiente = (pieza2 totVert) ifTrue:
                   [siguiente := 1]
                ifFalse:
                   [siguiente := siguiente + 1]]
             ifFalse:
                [otraArista := Arista nueva.
                otraArista longArista: (piezaTang dicAristas at: (i - 1)) longArista.
                otraArista anguloSalida: ( ((piezaTang dicAristas at: (i - 1))
                  anguloSalida) + ((arista2 anguloSalida) - 180) ).
                piezaTang dicAristas at: (i - 1) put: otraArista ]]
          ifFalse:
             [self giroInic: (arista2 anguloSalida + self giroInic + 180) ]].

       segmento1 < segmento2 ifTrue:
          [ i > 1 ifTrue:
             [otraArista := Arista nueva.
             otraArista longArista: (piezaTang dicAristas at: (i - 1)) longArista.
             otraArista anguloSalida: ((piezaTang dicAristas at: (i - 1)) anguloSalida
             piezaTang dicAristas at: ( i - 1 ) put: otraArista]
          ifFalse:
             [self giroInic: (180 + self giroInic)].
          unaArista longArista: (segmento2 - segmento1).
          unaArista anguloSalida: (arista2 anguloSalida).
          piezaTang dicAristas at: i put: unaArista.
          i := i + 1 ].
       unPunto x: i.
       unPunto y: siguiente.
^self!

auxEsDiferenteA: unTangrama por: i y: j con: ultimaArista
   "Ayuda a probar que dos tangramas son distintos"
   !umbral umbral2 k l unaArista otraArista numAriIgu!
   umbral := j - 1.
   umbral2 := i - 1.
   k:=1.
   l:=1.
   numAriIgu := 1.
   unaArista := self piezaTang dicAristas at: (umbral + k).
   otraArista := unTangrama piezaTang dicAristas at: (umbral2 + l ).
   [(numAriIgu = ultimaArista) or: [unaArista esDiferenteA: otraArista] ]
   whileFalse:
      [unaArista:= (self piezaTang dicAristas at: (umbral + k)).
      otraArista:= (unTangrama piezaTang dicAristas at: (umbral2 + l)).
      (unaArista esDiferenteA: otraArista) ifFalse:
         [numAriIgu := numAriIgu + 1].
      ((umbral + k) >= ultimaArista) ifFalse:
         [k := k + 1]
      ifTrue:
```

```
            [k := 1.
            umbral := 0].
        ((umbral2 + 1) >= ultimaArista) ifFalse:
            [l := l + 1]
        ifTrue:
            [l := 1.
            umbral2 := 0] ].
    (unaArista esDiferenteA: otraArista)
    ifTrue:
        [^false]
    ifFalse:
        [^true].!

combinacion: iesima de: unTangrama y: otroTangrama y: streamTexto y: apuVenSolucion
    "Genera un tangrama a partir de otros dos. Los dos tangramas
    tienen vertices concavos"
    |i arista1 arista2 unaArista segmento1 segmento2  numArista1
        numArista2 siguiente segmento1A segmento2A unPunto|
    "Inicializa las variables del tangrama resultado de la combinacion"
    pieza1 := unTangrama piezaTang.
    pieza2 := otroTangrama piezaTang.
    puntoUnion1 := (iesima union1De: (unTangrama piezaTang totVert)
      y: (otroTangrama piezaTang totVert)   ).
    puntoUnion2 := (iesima union2De: (unTangrama piezaTang totVert)
      y: (otroTangrama piezaTang totVert)   ).
    giroInic := 0.
    (streamTexto = nil) ifFalse:
        [puntoUnion1 printOn: streamTexto.
        Space printOn: streamTexto.
        puntoUnion2 printOn: streamTexto.
        Lf printOn: streamTexto].
    (puntoUnion1 > 0) ifTrue:
        ["Aristas anteriores a la union"
        numArista1 := (puntoUnion1 + 2)//3.
        numArista2 := (puntoUnion2 + 2)//3.
        numArista2 = (pieza2 totVert) ifTrue:
            [ siguiente := 1]
        ifFalse:
            [ siguiente := numArista2+1].
        i := 1.
        [ i < numArista1 ] whileTrue:
            [ piezaTang dicAristas at: i put: (pieza1 dicAristas at: i).
            i := i + 1 ].

        "La union. Primera parte."
        arista1   := pieza1 dicAristas at: i.
        arista2   := pieza2 dicAristas at: numArista2.
        segmento1 := (arista1 longArista) * (((  puntoUnion1 + 2)\\3 + 1)/4).
        segmento2 := (arista2 longArista) * (((4   -   ((puntoUnion2 + 2)\\3) - 1))/4).
        unaArista  := Arista nueva.
.pa     unPunto := Point new.
```

```
"Pasando la union, sobre pieza2"
[siguiente = numArista2] whileFalse:
  [siguiente = numArista2 ifFalse:
      [piezaTang dicAristas at: i put:
      (pieza2 dicAristas at: siguiente).
      i := i + 1.
      siguiente := siguiente + 1.
      siguiente = (pieza2 totVert + 1) ifTrue: [siguiente:=1]. ].    ].

"Al otro lado de  la union. Segunda parte."
segmento1A := arista1 longArista  -   segmento1.
segmento2A := arista2 longArista  -   segmento2.
segmento1A > segmento2A ifTrue:
   [ unaArista := Arista nueva.

   unaArista anguloSalida: ((piezaTang dicAristas at: (i - 1))
   anguloSalida  -  180).
   unaArista longArista: (piezaTang dicAristas at: (i - 1))
   longArista.
   piezaTang dicAristas at: (i - 1) put: unaArista.
   unaArista := Arista nueva.
   unaArista longArista: (segmento1A  -  segmento2A).
   unaArista anguloSalida: (arista1 anguloSalida).
   piezaTang dicAristas at: i put: unaArista.
   i := i + 1.
   siguiente := numArista1 + 1.  ].

unPunto x: i.
unPunto y: siguiente.
self auxCombin2: segmento1A y: segmento2A y: unaArista y: arista1 y:
arista2 y: unPunto y: numArista1 y: numArista2 y: segmento1
y: segmento2.
i := unPunto x .
siguiente := unPunto y.


(numArista1 = 1 and: [ segmento1 < segmento2 ]) ifTrue:
   [unaArista := Arista nueva.
   unaArista longArista:  (piezaTang dicAristas at: (i - 1)) longArista.
   unaArista anguloSalida: (((piezaTang dicAristas at: (i - 1))
   anguloSalida)  -  180).
   piezaTang dicAristas at: (i - 1) put: unaArista ].
```

```
      piezaTang totVert: (i - 1).
      (apuVenSolucion = nil) ifFalse:
         [formaTang := self creaForma: (apuVenSolucion frame) giro: giroInic]]
  ifFalse:
      [ ^Tangrama nuevo]. "error"
^self!

creaForma: unRect
   "regresa la forma del tangrama de tamaSo unRect"
^self piezaTang creaForma: unRect giro: giroTan!

discont: dicSeg y: libres y: streamTexto
 "Regresa true si hay alguna discontinuidad en la union (dicseg)"
 |unBool regiResu regiRes2 unPunto origen cont i total arista|
 unBool := false.
 arista := false.
 i:= 1.
 cont := 0.
 total := dicSeg size.
(total > 1) ifFalse:
    [unBool := true]
 ifTrue:
    [regiResu := dicSeg at:i.
    regiResu repe: 1.
    origen := regiResu union origin.
    (regiResu tipo = 'segmento') ifTrue:
       [arista := true].
    [i <= total] whileTrue:
       [i:= i + 1.
       (i > total) ifFalse:
          [regiRes2 := dicSeg at: i.
          (regiRes2 tipo = 'segmento') ifTrue: [arista := true].]
       ifTrue:
          [regiRes2 := RegiResul nuevo.

       ].
       (regiRes2 union origin = origen) ifFalse:
          [(((regiResu repe + 1) \\ 2) = 0) ifFalse:
             [(arista = true) ifFalse:
                [i := total + 1.  ]
             ifTrue:
                [cont := cont + 1.
                (cont=1) ifTrue: [libres x:  (i - 1)].
                (cont=2) ifTrue: [libres y: (i - 1)].
                (cont>2) ifTrue: [i := total + 1].
                regiResu := regiRes2.
                origen := regiResu union origin.
                regiResu repe: 1.
                (regiResu tipo = 'segmento') ifTrue:
                   [arista := true]
```

```
                ifFalse:
                    [arista := false.
                    ].
                ]
            ].]
        ifTrue:
            [regiResu repe: (regiResu repe + 1).
        ].
    ].
    (cont = 2) ifFalse: [unBool := true].
].
^unBool!

esDiferenteA: unTangrama y: streamTexto
    "prueba si dos tangramas son distintos"
    |unaArista otraArista i j ultimaArista umbral iguales inutil |

    "Primero hay que probar que tengan el mismo n#mero de aristas"
    ultimaArista := self piezaTang totVert.
    (ultimaArista = (unTangrama piezaTang totVert)) ifFalse:
        [^true]
    ifTrue: [ "Buscar una arista igual a la primera del 'receiver' "
        iguales := false.
        j := 1.
        unaArista := self piezaTang dicAristas at: j.
        [(j > ultimaArista) or: [iguales]] whileFalse:
            [i:= 1.
            otraArista := unTangrama piezaTang dicAristas at: i.
            [(unaArista esIgualA: otraArista) or: [i>ultimaArista]] whileFalse:
                [ otraArista := unTangrama piezaTang dicAristas at: i.
                i:= i + 1 ].
            (i = 1)  ifTrue:
                [i := 2].
            (i > ultimaArista) ifFalse:
                [ iguales := self auxEsDiferenteA: unTangrama por: (i - 1) y: j
                con: ultimaArista ].
            j:=j+1].
        ^iguales not ].!

esIgualA: unTangrama
    "prueba si dos tangramas son iguales o no"
^(self esDiferenteA: unTangrama) ifTrue: [False] ifFalse: [True]!

formaTang
    "regresa la matriz de bitios del tangrama"
^formaTang!

formaTang: unaForma
    "asigna al tangrama una nueva forma"
  formaTang := unaForma.
^self!
```

```
generaPoligono: giro1 y: streamTexto
  " Actualiza la variable piezaTang a base de la pieza1, la pieza2
    y el origen de la pieza2 (xCoor, yCoor, GiroInic)"

  |dicSeg dicVert1 dicVert2  dicVert3 totVert1 totVert2 segmento1 segmento2 regiResul
  k libres unPunto unaColec paso paso2|
  k := 0.
  dicVert1 := pieza1 vertices: 0 y: 0 y: 0 y: streamTexto.
  dicVert2 := pieza2 vertices: giroInic y: xCoor y: yCoor y: streamTexto.
  totVert1 := pieza1 totVert.
  totVert2 := pieza2 totVert.
  segmento1 := Segmento new.
  segmento2 := Segmento new.
  dicSeg := Dictionary new.
  libres := Point new.
  1 to: totVert1 do:
    [:i |segmento1 origin: (dicVert1 at: i) corner: (dicVert1 siguienteDe: i).
    1 to: totVert2  do:
      [:j |
      segmento2 origin: (dicVert2 at: j) corner: (dicVert2 siguienteDe: j).
      regiResul := segmento1 intersecta: segmento2 y: streamTexto.
      (regiResul = true) ifTrue:
         [^nil] "ensayo invalido; los tangramas se enciman"
      ifFalse:
         [(regiResul = nil) ifTrue: [] "Los segementos no se tocan"
         ifFalse:
            [regiResul pieza1: i pieza2: j.
              k := k + 1.
              dicSeg at: k put: regiResul.
              (regiResul tipo = 'segmento') ifTrue:
                 [k := k + 1.
                 paso := RegiResul nuevo.
                 paso2 := Segmento nuevo.
                 paso2 origin: (regiResul union corner) corner: (regiResul union origin);
                 paso union: paso2.
                 paso tipo: 'segmento'.
                 paso pieza1: i pieza2: j.
                 dicSeg at: k put: paso.
              ].]]]].
   unaColec := dicSeg values asOrderedCollection.
   dicSeg := unaColec asSortedCollection:
      [:i :j |  i  union origin <=
                j union origin  ].
   (self discont: dicSeg y: libres y: streamTexto) ifTrue:
      [^nil]. "ensayo invalido; discontinuidad en la interseccion de ambos tangramas"
   dicVert3 := self reconst: dicSeg y: libres y: dicVert1 y: dicVert2 y: streamTexto.
   "actualizar piezaTang con el resultado de poligono"
   unPunto := Point new.
   piezaTang poligono: dicVert3 y: dicVert3 size y: unPunto y: streamTexto.
```

```
   giroTan := giro1 + unPunto x.  "giro inicial del tangrama"
^true!

giroInic
^giroTan!

giroInic: unAngulo
 giroInic := unAngulo.
^giroTan!

giroTan
^giroTan!

inicializa
   "    "
   pieza1 := Pieza nueva.
   pieza2 := Pieza nueva.
   puntoUnion1 := 0.
   puntoUnion2 := 0.
   piezaTang := Pieza nueva.
   formaTang := Form white.
   giroInic := 0.
   xCoor := 0.
   yCoor := 0.
   giroTan := 0.
^self!

iniCoor: streamTexto y: apuVenSolucion
   "Actualiza las coordenadas de origen del tangrama"
   |x y i numArista1 numArista2 long arista angulo dicVert totVert
    arista1 arista2 segmento1 segmento2 unPunto|

   numArista1 := (puntoUnion1 + 2) //3.
   numArista2 := (puntoUnion2 + 2) //3.
   arista1    :=   pieza1 dicAristas at: numArista1.
   arista2    :=   pieza2 dicAristas at: numArista2.
   segmento1 := (arista1 longArista) * ((( puntoUnion1 + 2)\\3 + 1)/4).
   segmento2 := (arista2 longArista) * ((((puntoUnion2 + 2)\\3) +1)/4).
   (puntoUnion1 > 0 )ifFalse:
       [^Dictionary new] "error"
   ifTrue:
       ["buscar las coordenadas del primer vrtice"
       i := 1.
       x := 0.
       y := 0.
       angulo := 0.
       [ i < numArista1] whileTrue:
         [arista :=  pieza1 dicAristas at: i.
         long := arista longArista.
         x:= x + (long * angulo cos).
```

```
            y:= y + (long * angulo sin).
            i:= i + 1.
            angulo := angulo + arista anguloSalida].
         x:= x + ((segmento1 + segmento2)*(angulo cos)).
         y:= y + ((segmento1 + segmento2)*(angulo sin)).
         i := numArista2 - 1.

         [i > 0 ] whileTrue:
            [arista := pieza2 dicAristas at: i.
            long := arista longArista.
            angulo := angulo - arista anguloSalida.
            x:= x + (long * angulo cos).
            y:= y + (long * angulo sin).
            i:= i - 1].
         "x, y contienen las coordenadas del origen"
         angulo := angulo + 180.

      "asignar los valores a las variables de self"
         xCoor := x.
         yCoor := y.
         giroInic := angulo].
^self!

pieza1
^pieza1!

pieza1: unaPieza
   pieza1 := unaPieza.
^self!

pieza2
^pieza2!

pieza2: unaPieza
   pieza2 := unaPieza.
^self!

piezaTang
^piezaTang!

piezaTang: unaPieza
   piezaTang := unaPieza.
^self!

puntoUnion1
^puntoUnion1!

puntoUnion1: unEntero
   puntoUnion1 := unEntero.
^self!
```

```
puntoUnion2
^puntoUnion2!

puntoUnion2: unEntero
   puntoUnion := unEntero.
^self!

reconst: dicSeg y: libres y: dicVert1 y: dicVert2 y: streamTexto
"Regresa un diccionario de vertices del poligono resultantre de la union de
 dicVert1 y dicVert2"
|dicVert3 conjSeg2 l1 l2 caso borrado numArista1 numArista2 coli col2|
conjSeg2 := OrderedCollection new.
l1 := Point new.
l2 := Point new.
numArista1 := (puntoUnion1 + 2)//3.  "parametrizar el tres"
numArista2 := (puntoUnion2 + 2)//3.
dicSeg do: [:regResul | conjSeg2 add: (regResul union origin)].
conjSeg2 := conjSeg2 asSet.
dicVert3 := Circular nuevo.
l1 x: (dicSeg at: libres x) union origin x.
l1 y: (dicSeg at: libres x) union origin y.
l2 x: (dicSeg at: libres y) union origin x.
l2 y: (dicSeg at: libres y) union origin y.
conjSeg2 remove: l1.
conjSeg2 remove: l2.
dicVert1 removeAll: ( dicVert1 reject: [:vert1 | (conjSeg2 includes: vert1) ifTrue:
   [false] ifFalse: [true]]).
dicVert2 removeAll: ( dicVert2 reject: [:vert1 | (conjSeg2 includes: vert1) ifTrue:
   [false] ifFalse: [true]]).
caso := dicVert1 casos: dicVert2 con: l1 y: l2 y: streamTexto.
borrado := dicVert2 elementos.
borrado := dicVert1 elementos.
(caso = 1) ifTrue: "no hay puntos libres repetidos, en ambas piezas"
   [(borrado ~= -1) ifTrue:       "si no hay borrados"
      [dicVert3 desde: (numArista1 + 1) hasta: numArista1 por: dicVert1.
      dicVert3  desde: (numArista2 + 1) hasta: numArista2 por: dicVert2.]
   ifFalse:     "si hay borrados"
      [(borrado ~= 0) ifTrue:      "si no todos estan borrados"
         [dicVert3  desde:  0 hasta: -1 por: dicVert1.
      ].
      dicVert3  desde: 0 hasta: -1 por: dicVert2.
   ].
].
(caso = 2) ifTrue:  "un punto libre (l1) esta repetido en las dos piezas"
   [(borrado = -1) ifTrue:       "si no hay borrados"
      [dicVert3 desde: (numArista1 + 1) hasta: numArista1 por: dicVert1.
      ((dicVert3 at: 1) = l1) ifTrue:
         [dicVert3 := dicVert3 removeKey: 1.
         coli := 1]
```

```
      ifFalse
         [col  := dicVert3 size.
         dicV rt3 := dicVert3 removeKey: dicVert3 size.
      ].
      dicVert   desde: (numArista2 + 1) hasta: numArista2 por: dicVert2.
      (coli = 1) ifTrue: [coli := dicVert3 size].]
   ifFalse:    "si hay borrados"
      [(borrado ~= 0) ifTrue:     "si no todos estan borrados"
         [dic ert3  desde:  0 hasta: -1 por: dicVert1.
         coli := di.,ert3 size.
         dicV rt3  desde:  1 hasta: -1 por: dicVert2.]
      ifFalse
         [dic ert3  desde: 0 hasta: -1 por: dicVert2.
         coli := 0. "no puede haber colinealidad"
      ].
   ].
   dicVert3 :  dicVert3 colin: coli y: 0.
].
(caso = 11) i True:  "los dos puntos libres estan repetidos en ambas piezas"
   [(borrado  -1) ifTrue:       "si no hay borrados"
      [dicVer 3 desde: (numArista1 + 2) hasta: (numArista1 - 1) por: dicVert1.
      coli := dicVert3 size + 1.
      dicVert   desde: (numArista2 + 1) hasta: numArista2 por: dicVert2.
      col2 := dicVert3 size]
   ifFalse:    "si hay borrados"
      [dicVer 3  desde:  0 hasta: -1 por: dicVert1.
      coli := 1.
      col2 := dicVert3 size.
      dicVert   desde: 1 hasta: -1 por: dicVert2.
      dicVert  := dicVert3 removeKey: dicVert3 size.
   ].
   dicVert3 :  dicVert3  colin: coli y: col2.
].
^dicVert3!

union: iesima de: unTangrama y: otroTangrama y: streamTexto y: apuVenSolucion
   "A partir de dos tangramas construye la union
    y actualiza self"
   |giro1|
   puntoUnion1 := (iesima union1De: (unTangrama piezaTang totVert)
     y: (otroTangrama piezaTang totVert)   ).
   puntoUnion2 := (iesima union2De: (unTangrama piezaTang totVert)
     y: (otroTangrama piezaTang totVert)   ).
   pieza1 := unTangrama piezaTang.
   pieza2 := otroTangrama piezaTang.
   self iniCoor: streamTexto y: apuVenSolucion.
   pieza1 := Pieza nueva.
   pieza1 copia: unTangrama piezaTang.
   pieza2 := Pieza nueva.
   pieza2 copia: otroTangrama piezaTang.
   giro1 := unTangrama giroTan.
```

```
   ((self generaPoligono: giro1 y: streamTexto) = nil) ifTrue:
      [self inicializa] "ensayo invalido"
   ifFalse:
      [ (apuVenSolucion = nil) ifFalse:
         [formaTang := self creaForma: (apuVenSolucion frame)]].
^self!

vertices: streamTexto
   "Regresa la representaci"n del tangrama en trminos de un diccionario de vrtices"
 ^piezaTang vertices: giroInic y: xCoor y: yCoor y: streamTexto!

xCoor
^xCoor!

yCoor
^yCoor! !
a: otroTangrama piezaTang.
   giro1 := unTangrama giroTan.
   ((self generaPoligono: giro1 y: streamTexto) = nil) ifTrue:
      [self inicializa] "ensayo invalido"
   ifFalse:
      [ (apuVenSolucion = nil) ifFalse:
         [formaTang := self creaForma: (apuVenSolucion frame)]].
^self!

vertices: streamTexto
   "Regresa la representaci"n del tangrama en trminos de un diccionario de vrtices"
 ^piezaTang vertices: giroInic y: xCoor y: yCoor
```

```
Quetzalc subclass: #Pieza
  instanceVariableNames:
    'dicAristas totVert '
  classVariableNames: ''
  poolDictionaries:
    'CharacterConstants ' !

!Pieza class methods ! !


!Pieza methods !

anguloSalida: iesimo
   "regresa la amplitud del isimo  ngulo de salida de la pieza receptora"
^self dicAristas at: iesimo anguloSalida!

c:.:a: otraPieza
 self dicAristas: otraPieza dicAristas.
 totVert := otraPieza totVert.
^self!

creaForma: unRect giro: giroInic
   "regresa la forma de la pieza de tama$o unRect"
   !unaForma otraForma angulo long unaPluma escala unPunto xMax xMin
   yMax yMin otroRect!
   unaForma := Form width: unRect width height: unRect height.
   totVert = 0 ifTrue:
      [^unaForma].
   unaPluma := Pen new: unaForma.
  xMin :=  unaPluma location x.
   yMin :=  unaPluma location y.
   xMax :=  unaPluma location x.
   yMax :=  unaPluma location y.
   unaPluma up;
      home;
      down;
      black.
   angulo := 90 + giroInic.
   1 to: totVert by: 1 do: [:i |
     long := (dicAristas at: i) longArista .
     unaPluma turn: angulo;
        go: (10 * long).
     xMin > unaPluma location x ifTrue: [ xMin := unaPluma location x ].
     yMin > unaPluma location y ifTrue: [ yMin := unaPluma location y ].
     xMax < unaPluma location x ifTrue: [ xMax := unaPluma location x ].
     yMax < unaPluma location y ifTrue: [ yMax := unaPluma location y ].
     angulo:=(dicAristas at: i) anguloSalida].
     otroRect := Rectangle origin: (xMin @  yMin) extent:
        ((xMax - xMin  + 1 ) @ (yMax - yMin + 1 )).
    " otraForma := Form new width: (xMax - xMin + 1 ) height: (yMax - yMin + 1 )
        initialByte: 0.
     otraForma copy: otroRect from: unaForma to: (0 @ 0)
```

```
                    rule: Form orRule. "

   ((unaForma byteValueAtX: 0 Y: 0)  =  0) ifTrue:
     [unaPluma fillAt: (0 @ 0)]
   ifFalse:
     [ ((unaForma byteValueAtX: 1 Y: 0)  =  0) ifTrue:
        [unaPluma fillAt: (1 @ 0)]
     ifFalse:
        [ ((unaForma byteValueAtX: 0 Y: 1)  =  0) ifTrue:
          [unaPluma fillAt: (0 @ 1)]
        ifFalse:
          [unaPluma fillAt: (1 @ 1)] ] ].
   unaForma reverse.
   unaPluma up;
      direction: 270;
      home;
      down;
      black.
   angulo := 90 + giroInic.
   1 to: totVert by: 1 do: [:i |
     long := (dicAristas at: i) longArista .
     unaPluma turn: angulo;
        go: (10 * long).
        angulo:=(dicAristas at: i) anguloSalida].

   "unaForma copy: otroRect from: otraForma to: (1 @ 1) rule: Form orRule."

   "otroRect := Rectangle origin: (xMin @  yMin) extent:
        ((xMax - xMin  + 1 ) @ (yMax - yMin + 1 )). "
     otraForma := Form new width: (xMax - xMin + 1 ) height: (yMax-yMin + 1 )
        initialByte: 0.
     otraForma copy: otroRect from: unaForma to: ( 0 @ 0 ) rule: Form orRule.
     ((unRect height // (yMax - yMin + 1 )) > (unRect width //
        (xMax - xMin + 1 ))) ifTrue:
        [escala := unRect width // otroRect width ]
     ifFalse:
        [escala := unRect height // otroRect height ].
     "otraForma := otraForma magnify: otraForma boundingBox by: (escala @ escala)."
^otraForma!

cuadrado
   "regresa el cuadrado est ndar"
   |unaArista|
   totVert :=4.
   1 to: 4 by: 1 do: [:i|   unaArista := Arista nueva.
   unaArista longArista: ((14041/10000)*5/2).
   unaArista anguloSalida: (180+90).
   dicAristas at: i put: unaArista].
^self!

dicAristas
```

```
^dicAristas!

dicAristas: unDiccionario
 |i salida unaArista otraArista|
 i := 1.
 salida := true.
 dicAristas := Dictionary new.
 [salida] whileTrue:
    [unaArista := Arista nueva.
    otraArista := unDiccionario at: i ifAbsent: [salida := false].
    (salida) ifTrue:
      [unaArista copia: otraArista.
      dicAristas at: i put: unaArista.
      i := i + 1]].
^dicAristas!

inicializa
   "Genera un diccionario vac!o en dicAristas y totVert igual a cero"
   dicAristas := Dictionary new.
   totVert := 0.
^ self!

longArista: iesima
   "regresa la longitud de la isima arista de la pieza receptora"
^self dicAristas at: iesima longArista!

poligono: dicVert y: numVert y: unPunto y: streamTexto
   "Traduce de una representacion de Vertices a una grafica de tortuga (inversa de
   vertices:)
   Actualiza el Diccionario de Aristas y el total de vertices, que son las dos variable
   de instancia de pieza
   Regresa la representacion  de la pieza en trminos de un diccionario de aristas"
   |x1 y1  x2 y2 deltaX deltaY i  angAnt angulo long arista dicAris unPunto|
   i := 1.
   dicAristas := Dictionary new.
   totVert := numVert.
   angulo := 0.
   angAnt := 0.

   "    'Las vertices son:' printOn: streamTexto.
      i := 1.
      [i > totVert ] whileFalse:
         [ (dicVert at: i)  printOn: streamTexto.
         Lf printOn: streamTexto.
         i := i + 1].
    i:=1.  Lf printOn: streamTexto.  "
   x1 := (dicVert at: i) x.
   y1 := (dicVert at: i) y.
   i:= 2.
   [ i > (totVert + 1) ] whileFalse:
      [arista := Arista nueva.
```

```
        (i <= totVert) ifTrue:
            [x2 := (dicVert at: i) x.
            y2 := (dicVert at: i) y]
        ifFalse:
            [x2 := (dicVert at: 1) x.
            y2 := (dicVert at: 1) y].
        deltaX := x2 - x1.
        deltaY := y2 - y1.
        angulo := deltaY arcTan: deltaX.
        (angulo sin = 0) ifTrue:
            [long := deltaX abs. ]
        ifFalse:
            [long := (deltaY / angulo sin) abs.
           "(long = ((long * 10000) // 10000)) ifFalse:
                [long := long//(7071/10000) * (7071/10000)]"].
        "long := ((long  * 100000 + (1/2))//1)/100000."
        arista longArista: long.
        arista anguloSalida: (angulo - angAnt).
        angAnt := angulo.
        dicAristas at: (i - 1) put: arista.
        y1 := y2.
        x1 := x2.
        i := i + 1].
        angulo :=  (dicAristas at: 1) anguloSalida.
        unPunto x: angulo.
        2 to: totVert do:
          [:i | (dicAristas at: (i - 1)) anguloSalida: (dicAristas at: i) anguloSalida].
        (dicAristas at: totVert) anguloSalida: (angulo - angAnt).
        'Las aristas son:' printOn: streamTexto.
        i := 1.
        [i > totVert ] whileFalse:
            [ (dicAristas at: i) longArista printOn: streamTexto.
              Space printOn: streamTexto.
              (dicAristas at: i) anguloSalida \\ 360 printOn: streamTexto.
            Lf printOn: streamTexto.
            i := i + 1].

^dicAristas!

prueba1
    "regresa el cuadrado est ndar"
    |unaArista|
    totVert := 8.
    unaArista := Arista nueva.
    unaArista longArista: 4.
    unaArista anguloSalida: (180+90).
    dicAristas at: 1 put: unaArista.
    unaArista := Arista nueva.
    unaArista longArista: 4.
    unaArista anguloSalida: 90.
    dicAristas at: 2 put: unaArista.
.paunaArista := Arista nueva.




    unaArista longArista: 4
```

```
    unaArista := Arista nueva.
    unaArista longArista: 4.
    unaArista anguloSalida: (180+90).
    dicAristas at: 5 put: unaArista.
    unaArista := Arista nueva.
    unaArista longArista: 8.
    unaArista anguloSalida: (180+90).
    dicAristas at: 6 put: unaArista.
    unaArista := Arista nueva.
    unaArista longArista: 12.
    unaArista anguloSalida: (180+90).
    dicAristas at: 7 put: unaArista.
    unaArista := Arista nueva.
    unaArista longArista: 8.
    unaArista anguloSalida: (180+90).
    dicAristas at: 8 put: unaArista.
^self!

prueba2
    "regresa el cuadrado est ndar"
    !unaArista!
    totVert :=4.
    1 to: 4 by: 1 do: [:i!   unaArista := Arista nueva.
    unaArista longArista: 4.
    unaArista anguloSalida: (180+90).
    dicAristas at: i put: unaArista].
^self!

prueba3
    "regresa el cuadrado est ndar"
    !unaArista!
    totVert :=3.
    unaArista := Arista nueva.
    unaArista longArista: 4.
    unaArista anguloSalida: (180+90).
    dicAristas at: 1 put: unaArista.
    unaArista := Arista nueva.
    unaArista longArista: 4.
    unaArista anguloSalida: (180+45).
    dicAristas at: 2 put: unaArista.
    unaArista := Arista nueva.
    unaArista longArista: 8 * (45 cos).
    unaArista anguloSalida: (180+45).
```

```
   dicAristas at: 3 put: unaArista.
^self!

totVert
^totVert!

totVert: unaFraccion
  totVert := unaFraccion.
^self!

trapezio
   "regresa el trapezio est ndar"
   !unaArista!
   totVert :=4.
   unaArista := Arista nueva.
   unaArista longArista: 5;
      anguloSalida: (180+45).
   unaAristas:=tAristatnuevaArista.
   unaArista longArista: ((14041/10000)*5/2);
      anguloSalida: (360 - 45).
   dicAristas at: 2 put: unaArista.
   unaArista := Arista nueva.
   unaArista longArista: 5;
      anguloSalida: (180 + 45).
   dicAristas at: 3 put: unaArista.
   unaArista := Arista nueva.
   unaArista longArista: ((14041/10000)*5/2);
      anguloSalida: (360 - 45).
   dicAristas at: 4 put: unaArista.
^self!

trianguloChico
   "regresa un triangulo chico est ndar"
   !unaArista!
   totVert :=3.
   unaArista := Arista nueva.
   unaArista longArista: 5;
      anguloSalida: (180+45).
   dicAristas at: 1 put: unaArista.
   unaArista := Arista nueva.
   unaArista longArista: ((14041/10000)*5/2);
      anguloSalida: (180+90).
   dicAristas at: 2 put: unaArista.
   unaArista := Arista nueva.
   unaArista longArista: ((14041/10000)*5/2);
      anguloSalida: (180+45).
   dicAristas at: 3 put: unaArista.
^self!

trianguloGrande
   "regresa un triangulo grande est ndar"
.pa!unaArista!



   totVert :=3.
   unaArista := Arista nueva
```

```
    dicAristas at: 2 put: unaArista.
    unaArista := Arista nueva.
    unaArista longArista: ((14041/10000)*5);
        anguloSalida: (180+45).
    dicAristas at: 3 put: unaArista.
^self!

trianguloMediano
    "regresa un triangulo mediano est ndar"
    |unaArista|
    totVert :=3.
    unaArista := Arista nueva.
    unaArista longArista: ((14041/10000)*5);
        anguloSalida: (180+45).
    dicAristas at: 1 put: unaArista.
    unaArista := Arista nueva.
    unaArista longArista: 5;
        anguloSalida: (180+90).
    dicAristas at: 2 put: unaArista.
    unaArista := Arista nueva.
    unaArista longArista: 5;
        anguloSalida: (180+45).
    dicAristas at: 3 put: unaArista.
^self!

vertices: unAngulo y: xx  y: yy y: streamTexto
    "Regresa la representaci"n de la pieza en trminos de un diccionario de puntos"
    |x y i  angulo long long2 arista dicVert unPunto|
    dicVert := Circular nuevo.
    dicVert totVert: totVert.
    i := 1.
    x := xx.
    y := yy.
    angulo := unAngulo.
    arista := Arista nueva.
    [ i > totVert ] whileFalse:
        [unPunto := Point new.
        unPunto x: x.
        unPunto y: y.
        dicVert at: i put: unPunto.
        arista :=   dicAristas at: i.
        long :=  arista longArista.
        "x:= x + (long * angulo cos).
```

```
        y:= y + (long * angulo sin)."

        long2 := ((long * angulo cos  * 100000 + (1/2))//1)/100000.
        x:= x + long2.
        long2 := ((long * angulo sin  * 100000 + (1/2))//1)/100000.
        y:= y + long2.

        angulo := angulo + arista anguloSalida.
        i:= i + 1].
^dicVert! !
```

```
Quetzalc subclass: #Arista
  instanceVariableNames:
    'longArista anguloSalida '
  classVariableNames: ''
  poolDictionaries: '' !

!Arista class methods ! !


!Arista methods !

anguloSalida
^anguloSalida!

anguloSalida: unaFraccion
   anguloSalida := unaFraccion.
^self!

copia: otraArista
 self longArista: otraArista longArista.
 self anguloSalida: otraArista anguloSalida.
^self!

esDiferenteA: unaArista
   "Prueba si dos aristas son iguales"
^(self esIgualA: unaArista) not!

esIgualA: unaArista
   "Prueba si dos aristas son iguales"
^((longArista  + (1/2)//1
    = (unaArista longArista  + (1/2)//1)) and: [(anguloSalida\\360) =
 ((unaArista anguloSalida)\\360)])!

inicializa
   "Genera un diccionario vac!o en dicAristas y totVert igual a cero"
   longArista := 0.
   anguloSalida := 0.
^ self!

longArista
^longArista!

longArista: unaFraccion
   longArista := unaFraccion.
^self! !
```

```
Rectangle subclass: #Segmento
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: '' !

!Segmento class methods !

nuevo
^super new! !


!Segmento methods !

intersecta: segmento2 y: streamTexto
 "Elaborado a partir del algoritmo presentado en: Hgron, Grard.- Image Synthesis,
  Elementary Algorithms.- MIT press.- EUA, 1988.- p. 185"
 "Regresa el segmento resultante construido a partir de la interseccion de los dos
  segmentos pasados como parametros (self y segmento2). Cuando los segmentos se
  intersectan en un punto que no es extremo, regresa true; cuando los segmentos no
  se tocan regresa nil."
 "Asume que no hay segmentos degenerados en punto"
 |unRect regiResul xba xcd ycd yba  xca yca t1 t2 m a b c d e temporal|
 regiResul := RegiResul nuevo.
 a := self origin.
 b := self corner.
 c := segmento2 origin.
 d := segmento2 corner.
 (b <= a) ifTrue:
    [temporal := a.
    a:= b.
    b:= temporal.
    self origin: a corner: b.
 ].
 (d <= c) ifTrue:
    [temporal := c.
    c:= d.
    d:= temporal.
    segmento2 origin: c corner: d.
 ].
 xba := self corner x - self origin x.
 yba := self corner y - self origin y.
 xcd := segmento2 origin x - segmento2 corner x.
 ycd := segmento2 origin y - segmento2 corner y.
 e   := xba * ycd - (xcd * yba).
 (e=0) ifTrue:
    ["paralelos o colineales"
    (yba = 0) ifFalse:
       [(xba = 0) ifFalse:
          [(a y - ((xba/yba)* a x) = (c y - ((xba/yba) * c x))) ifFalse:
            [^nil]  "paralelas, no se tocan"
          ifTrue:
             [(self intersects: segmento2) ifFalse:
```

```
                [^nil]          "colineales, no se tocan"
            ifTrue:
                [regiResul tipo: 'segmento'.
                regiResul union: (self intersect: segmento2).
                (regiResul union origin = regiResul union corner) ifTrue:
                    [regiResul tipo: 'punto'
                ].
            ].
          ].]
        ifTrue:
          [(b x = c x)  ifFalse:
              [^nil]  "paralelas, no se tocan"
            ifTrue:
              [(self intersects: segmento2) ifFalse:
                 [^nil]          "colineales, no se tocan"
              ifTrue:
                 [regiResul tipo: 'segmento'.
                 regiResul union: (self intersect: segmento2).
                 (regiResul union origin = regiResul union corner) ifTrue:
                    [regiResul tipo: 'punto'
                 ].
              ].
            ].
        ].]
    ifTrue:
      [(b y = c y)  ifFalse:
        [^nil]  "paralelas, no se tocan"
       ifTrue:
         [(self intersects: segmento2) ifFalse:
            [^nil]          "colineales, no se tocan"
          ifTrue:
            [regiResul tipo: 'segmento'.
            regiResul union: (self intersect: segmento2).
            (regiResul union origin = regiResul union corner) ifTrue:
               [regiResul tipo: 'punto'
            ].
          ].
       ].
    ].]
ifFalse:
   [xca := segmento2 origin x - self origin x.
   yca := segmento2 origin y - self origin y.
   t1 := xca * ycd - (xcd * yca)/e.
   t2 := xba * yca - (yba * xca)/e.
   ((t1 < 0) or: [ (t1 > 1) or: [(t2 < 0) or:[t2>1]]]) ifTrue:
      [^nil] "no hay interseccion; segmentos disjuntos"
   ifFalse:
      [ (((t1 = 0) or: [t1=1]) or: [(t2=0) or: [t2=1]]) ifFalse:
        [^true] "los segmentos se intersectan; los tangramas se enciman."
       ifTrue:
        [regiResul tipo: 'punto'.
```

```
        unRect := Rectangle new.
        (t1=0) ifTrue: [unRect origin: a corner: a].
        (t1=1) ifTrue: [unRect origin: b corner: b].
        (t2=0) ifTrue: [unRect origin: c corner: c].
        (t2=1) ifTrue: [unRect origin: d corner: d].
        regiResul union: unRect.
         ]]].
^regiResul!

intersects: aRectangle
        "Answer true if the receiver and aRectangle have
        any area in common, else answer false."
        "Modificacion al metodo de la clase Rectangle para que incluya unidos por el
        vertice"
    ^origin <= aRectangle corner
        and: [aRectangle origin <= corner]! !
in: b corner: b].
        (t2=0) ifTrue: [unRect origin: c corner: c].
        (t2=1) ifTrue: [unRect origin: d corner: d].
        regiResul union: unRect.
         ]]].
^regiResul!
```

```
Dictionary variableSubclass: #Circular
  instanceVariableNames:
    'primerElemento ultimoElemento totVert '
  classVariableNames: ''
  poolDictionaries: '' !

!Circular class methods !

nuevo
^super new inicializa.! !


!Circular methods !

anteriorA: unaLlave
"regresa el valor de la llave que le precede a unaLlave en self"
^self at: (self llaveAnteriorA: unaLlave).!

casos: dicVert2 con: l1 y: l2 y: streamTexto
|caso cas2 |
caso := 0.
cas2 := 0.
((self     keyAtValue: l1) = nil) ifFalse: [caso := caso + 1000].
((self     keyAtValue: l2) = nil) ifFalse: [caso := caso + 100].
((dicVert2 keyAtValue: l1) = nil) ifFalse: [caso := caso + 10].
((dicVert2 keyAtValue: l2) = nil) ifFalse: [caso := caso + 1].
(caso = 1111) ifTrue: [cas2 := 11].
(caso = 0111) ifTrue: [(l1 x: l2 x) y: l2 y.
                       cas2 := 2].
(caso = 1011) ifTrue: [cas2 := 2].
((caso = 0011)or:[(caso = 1001)or: [caso = 0110]])  ifTrue:
   [cas2 := 1].
(caso = 1101) ifTrue:
   [(l1 x: l2 x) y: l2 y.
   self intercambia: dicVert2.
   cas2 := 2].
(caso = 1110) ifTrue:
   [self intercambia: dicVert2.
   cas2 := 2].
(caso = 1100) ifTrue:
   [self intercambia: dicVert2.
   cas2 := 1].
(cas2 = 0) ifTrue:
   [cas2 := -1].
^cas2!

colin: col1 y: col2
"regresa self. Elimina los vertices en que halla colinealidad. Prueba en los vertcies
col1 y col2"
|punto1 punto2 vertice1 vertice2 vertice3 pend1 pend2 dicVert i|
dicVert := Circular nuevo.
```

```
((coli > col2) and: [col2 > 0]) ifTrue:
   [punto1 := col2.
   punto2 := coli.]
ifFalse:
   [punto1 := coli.
   punto2 := col2].
"colinealidad en el primer punto"
vertice2 := self at: punto1.
vertice1 := self anteriorA: punto1.
vertice3 := self siguienteDe: punto1.
pend1 := vertice2 x - vertice1 x.
(pend1 = 0) ifTrue:
   [pend1 := 'infinito']
ifFalse:
   [pend1 := (vertice2 y - vertice1 y) / pend1.
].
pend2 := vertice3 x - vertice2 x.
(pend2 = 0) ifTrue:
   [pend2 := 'infinito']
ifFalse:
   [pend2 := (vertice3 y - vertice2 y) / pend2.
].
(pend1 = pend2) ifTrue:
   [self removeKey: punto1.
].
"colinealidad en el segundo punto"
(punto2 > 0)  ifTrue:
   [vertice2 := self at: punto2.
   vertice1 := self anteriorA: punto2.
   vertice3 := self siguienteDe: punto2.
   pend1 := vertice2 x - vertice1 x.
   (pend1 = 0) ifTrue:
      [pend1 := 'infinito']
   ifFalse:
      [pend1 := (vertice2 y - vertice1 y) / pend1.
   ].
   pend2 := vertice3 x - vertice2 x.
   (pend2 = 0) ifTrue:
      [pend2 := 'infinito']
   ifFalse:
      [pend2 := (vertice3 y - vertice2 y) / pend2.
   ].
   (pend1 = pend2) ifTrue:
      [self removeKey: punto2.
   ].
].
"renumerar"
i := 1.
self do:[:vertice | dicVert at: i put: vertice.
   i := i + 1.
].
```

```
^dicVert!

desde: inicio hasta: fin por: dicVert
"agrega a self los vertices de dicVert comenzando por inicio y terminando en fin
si fin es -1 entonces comienza desde el primer elemento + inicio y termina en el ultimo
elemento"
|indice primero ultimo base|
indice := self size +1.
base := dicVert size.
(fin = -1) ifTrue:
   [primero := dicVert primerElemento + inicio \\ dicVert totVert.
   ultimo := dicVert ultimoElemento.]
ifFalse:
   [primero := inicio.
   ultimo := fin.
   (inicio = 0) ifTrue: [primero := base].
   (inicio > base ) ifTrue: [primero := 1].
   (fin = 0) ifTrue: [ultimo := base].
   (fin > base ) ifTrue: [ultimo := 1].
].
self at:indice put: (dicVert at: primero).
[primero = ultimo] whileFalse: [
   indice := indice +1.
   primero := dicVert llaveSiguienteDe: primero.
   self at: indice put: (dicVert at: primero).
].
^self!

elementos
|i llaves  base |
"actualiza las variables que indican la llave  del primer y ultimo elemento de self."
" regresa 0 si todos estan borrados; -1 si no hay borrados; en cualquier
otro caso regresa al ultimo elemento"
llaves := OrderedCollection new.
self keysDo: [ :llave | llaves addFirst: llave].
base := llaves size.
totVert := base.
(base = 0) ifTrue:
   [ultimoElemento := 0]   "todos borrados"
ifFalse:
   [primerElemento := 0.
   ultimoElemento := 0.
   i := 1.
   (llaves  includes: 1) ifFalse:  "si el primero esta borrado"
      [primerElemento := llaves at: 1.
      ultimoElemento := llaves at: base.]
   ifTrue:                         "si el primero no esta borrado"
      [ultimoElemento := 1.
      i := 0.
      llaves do: [ :llave |
         ((llave ~= (i +1)) and: [ i < 9999]) ifTrue:
```

```
            [ultimoElemento := i.
            i := 9999]
         ifFalse:
            [i := i + 1.
         ].
      ].
      (i >= 9999) ifTrue:
         [ primerElemento:= llaves at: (ultimoElemento +1)]
      ifFalse:
         [primerElemento := 1.
         (base = totVert) ifTrue:
            [ultimoElemento := -1] "no hay borrados"
         ifFalse:
            [ultimoElemento := base.
         ].
      ].
   ].
].
^ultimoElemento!

inicializa
  primerElemento := 0.
  ultimoElemento := 0.
  totVert := 0.
^self!

intercambia: dicVert2
| temporal |
temporal := self select: [:temp | true].
temporal keysDo: [ :i | self removeKey: i].
dicVert2 associationsDo: [ :i | self add: i].
self keysDo: [ :i | dicVert2 removeKey: i].
temporal associationsDo: [ :i | dicVert2 add: i].
^nil!

llaveAnteriorA: unaLlave
"regresa la llave que le precede a unaLLave en self"
|base llaves llaveAnt indice|
llaveAnt := nil.
base := self size.
llaves := OrderedCollection new.
self keysDo: [ :llave | llaves addFirst: llave].
indice := llaves indexOf: unaLlave.
(indice <= 1) ifTrue:
   [llaveAnt := base]
ifFalse:
   [llaveAnt := indice - 1.
].
^llaves at: llaveAnt.!

llaveSiguienteDe: unaLlave
"regresa la llave que le sigue a unaLLave en self"
```

```
!base llaves llaveSig indice!
llaveSig := nil.
base := self size.
llaves := OrderedCollection new.
self keysDo: [ :llave | llaves addFirst: llave].
indice := llaves indexOf: unaLlave.
(indice >= base ) ifTrue:
   [llaveSig := 1]
ifFalse:
   [llaveSig := indice + 1.
].
^llaves at: llaveSig.!

primerElemento
^primerElemento!

siguienteDe: unaLlave
"regresa el valor de la llave que sigue a unaLLave en self"
^self at: (self llaveSiguienteDe: unaLlave).!

totVert
^totVert!

totVert: unEntero
   totVert := unEntero.
^self!

ultimoElemento
^ultimoElemento! !
 := OrderedCollection new.
self keysDo: [ :llave | llaves addFirst: llave].
indice := llaves indexOf: unaLlave.
(indice >= base ) ifTrue:
   [llaveSig := 1]
ifFalse:
   [llaveSig := indice + 1.
].
^llaves at: llaveSig.!

primerElemento
^primerElemento!

siguienteDe: unaLlave
"regres
```

```
Quetzalc subclass: #RegiResul
  instanceVariableNames:
    'tipo union pieza1 pieza2 libre1 libre2 repe '
  classVariableNames: ''
  poolDictionaries: '' !

!RegiResul class methods !

nuevo
 ^super new inicializa! !


!RegiResul methods !

inicializa
  tipo := ''.
  pieza1 := 0.
  pieza2 := 0.
  union  := Rectangle new.
  libre1 := nil.
  libre2 := nil.
  repe := 0.!

libre1
^libre1!

libre1: unString
 libre1 := unString.
^self!

libre2
^libre2!

libre2: unString
 libre2 := unString.
^self!

pieza1: i pieza2: j
   pieza1 := i.
   pieza2 := j.
^self!

repe
^repe!

repe: unEntero
 repe := unEntero.
^self!

tipo
^tipo!

tipo: unString
```

```
 tipo := unString.
^self!

union
^union!

union: unRect
    union := unRect.
^self! !
```

```
Magnitude subclass: #Number
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: '' !

!Number class methods !

new: argumentIgnored
        "Answer an instance of the receiver.
         This method reports an error."
    ^self invalidMessage! !


!Number methods !

* aNumber
        "Answer the result of multiplying
         the receiver by aNumber."
    ^self implementedBySubclass!

+ aNumber
        "Answer the sum of the receiver and aNumber."
    ^self implementedBySubclass!

- aNumber
        "Answer the difference between
         the receiver and aNumber."
    ^self implementedBySubclass!

/ aNumber
        "Answer the result of dividing
         the receiver by aNumber."
    ^self implementedBySubclass!

// aNumber
        "Answer the integer result of dividing the
         receiver by aNumber with truncation
         towards negative infinity."
    ^self implementedBySubclass!

@ aNumber
        "Answer a point with the receiver as the
         x-coordinate and aNumber as the y-coordinate."
    ^Point new
        x: self;
        y: aNumber!

\\ aNumber
        "Answer the integer remainder after dividing
         the receiver by aNumber with truncation
         towards negative infinity."
    ^self implementedBySubclass!
```

```
obs
        "Answer the absolute value of the receiver."
    self < 0
        ifTrue: [^self negated].
    ^self!

arcCos
        "Answer the arc-cosine, an angle in
         radians, of the receiver."
    ^(Float pi / 2) - self arcSin!

arcSin
        "Answer the arc-sine, an angle in
         radians, of the receiver."
    (self > 1 or: [self < -1])
        ifTrue: [^self error: 'receiver of arcSin out of range'].
    self = 1
        ifTrue: [^Float pi / 2].
    self = -1
        ifTrue: [^(Float pi / 2) negated].
    ^(self / (1 - (self * self)) sqrt) arcTan!

arcTan
        "Answer the arc-tangent, an angle in
         radians, of the receiver."
    ^self asFloat arcTan!

arcTan:  ad
   !op unRacional otroEntero  resul!
   resul := nil. "error"
   op := self.
   ((op = ad) and: [ (op = 0) ]) ifTrue:
      [^resul].    "error"
   unRacional := op / ad.
   ((unRacional > (-1/2)) and: [unRacional < (1/2)])    ifTrue:
      [(ad > 0)  ifTrue:
         [resul := 0]
      ifFalse:
         [resul := 180]].
    ((unRacional < (-3/2)) or: [unRacional > (3/2)] )   ifTrue:
      [(op > 0)  ifTrue:
         [resul := 90]
      ifFalse:
         [resul := -90]].
   ((unRacional > (1/2)) and: [unRacional < (3/2)])    ifTrue:
      [(op > 0)  ifTrue:
         [resul := 45]
      ifFalse:
         [resul := -135]].
   ((unRacional < (-1/2)) and: [unRacional > (-3/2)])    ifTrue:
      [(op > 0)  ifTrue:
```

```
        [resul := 135]
      ifFalse:
        [resul := -45]].

^resul!

ceiling
        "Answer the integer nearest the
         receiver towards positive infinity."
    | anInteger |
    anInteger := self // 1.
        "truncates >> negative infinity"
    anInteger = self ifTrue: [^anInteger].
    ^anInteger + 1!

cos
    |unEntero otroEntero unRacional|
    unRacional := 7071/10000.
    unEntero := self \\ 180.
    otroEntero := self // 180.
    ((otroEntero \\ 2) = 0) ifTrue:
       [otroEntero := 1]
    ifFalse:
       [otroEntero := 1 negated].
    (unEntero =  0)  ifTrue: [^otroEntero*1].
    (unEntero = 45)  ifTrue: [^otroEntero*unRacional].
    (unEntero = 90)  ifTrue: [^0].
    (unEntero = 135) ifTrue: [^ (otroEntero*unRacional) negated].
^nil "error"!

cosres
        "Answer a Float which is the cosine of the receiver.
         The receiver is an angle measured in radians."
    ^self asFloat cos!

degreesToRadians
        "Answer the receiver converted
         from degrees to radians."
    ^self asFloat degreesToRadians!

denominator
        "Answer the denominator of the receiver.  Default
         is one which can be overridden by the subclasses."
    ^1!

even
        "Answer true if the integer part of
         the receiver is even, else answer false."
    ^self \\ 2 = 0!

exp
```

```
        "Answer a Float which is the
         exponential of the receiver."
    ^self asFloat exp!

floor
        "Answer the integer nearest the receiver
         truncating towards negative infinity."
    ^self // 1!

integerCos
        "Answer the integer cosine of the receiver
         angle, measured in degrees, scaled by 100."
    | q r |
    r := self rounded \\ 360.
    q := r // 90.
    r := r \\ 90 + 1.
    q = 0 ifTrue: [
        ^SinValues at: 92 - r].
    q = 1 ifTrue: [
        ^0 - (SinValues at: r)].
    q = 2 ifTrue: [
        ^0 - (SinValues at: 92 - r)].
    q = 3 ifTrue: [
        ^SinValues at: r]!

integerSin
        "Answer the integer sine of the receiver
         angle, measured in degrees, scaled by 100."
    | q r |
    r := self rounded \\ 360.
    q := r // 90.
    r := r \\ 90 + 1.
    q = 0 ifTrue: [
        ^SinValues at: r].
    q = 1 ifTrue: [
        ^SinValues at: 92 - r].
    q = 2 ifTrue: [
        ^0 - (SinValues at: r)].
    q = 3 ifTrue: [
        ^0 - (SinValues at: 92 - r)]!

ln
        "Answer a Float which is the
         natural log of the receiver."
    ^self asFloat ln!

log: aNumber
        "Answer a Float which is the log
         base aNumber of the receiver."
    ^self asFloat ln / aNumber asFloat ln!

negated
```

```
        "Answer the negation of the receiver."
    self implementedBySubclass!

negative
        "Answer true if the receiver is less
         than zero, else answer false."
    ^self < 0!

numerator
        "Answer the numerator of the receiver.  Default
         is the receiver which can be overridden by the
         subclasses."
    ^self!

odd
        "Answer true if the integer part of
         the receiver is odd, else answer false."
    ^self \\ 2 = 1!

positive
        "Answer true if the receiver is greater
         than or equal to zero, else answer false."
    ^self >= 0!

printFraction: numberFractionDigits
        "Answer a string, the ASCII representation
         of the receiver truncated to numberFractionDigits
         decimal places."
    | stream fraction integer |
    numberFractionDigits < 0
        ifTrue: [self error: 'Negative digit count'].
    stream := WriteStream on: (String new: 16).
    (integer := self // 1) printOn: stream.
    stream nextPut: $.
    fraction := self - integer.
    integer := 0.
    numberFractionDigits timesRepeat: [
        fraction := (fraction - integer) * 10.
        (integer := fraction // 1) printOn: stream].
    ^stream contents!

printOn: aStream
        "Append the ASCII representation of
         the receiver to aStream."
    ^self implementedBySubclass!

printRounded: numberFractionDigits
        "Answer a string, the ASCII representation
         of the receiver rounded to numberFractionDigits
         decimal places."
    | rounder |
    rounder := 1/(#(1 10 100 1000 10000
```

```
        100000 10000000 100000000)
        at: numberFractionDigits + 1).
    ^(self roundTo: rounder)
        printFraction: numberFractionDigits!

quo: aNumber
        "Answer the integer quotient
         with truncation toward zero."
    ^(self / aNumber) truncated!

radiansToDegrees
        "Answer the receiver converted
         from radians to degrees."
    ^self asFloat radiansToDegrees!

raisedTo: aNumber
        "Answer a Float which is the receiver
         raised to the power of aNumber."
    ^(aNumber * self ln) exp!

raisedToInteger: anInteger
        "Answer the receiver raised
         to the power of anInteger."
    | answer |
    (anInteger isKindOf: Integer)
        ifTrue: [
            answer := 1.
            anInteger abs timesRepeat: [
                answer := self * answer].
            anInteger < 0
                ifTrue:  [^1 / answer]
                ifFalse: [^answer]].
    self error: 'raisedToInteger needs integer power'!

reciprocal
        "Answer one divided by the receiver."
    ^self implementedBySubclass!

rem: aNumber
        "Answer the integer remainder after dividing
         the receiver by aNumber with truncation
         towards zero."
    ^self - ((self quo: aNumber) * aNumber)!

rounded
        "Answer the nearest integer to the receiver."
    ^self + self + self sign quo: 2!

roundTo: aNumber
        "Answer the receiver rounded to the
         nearest multiple of aNumber."
    ^self + (aNumber/2) truncateTo: aNumber!
```

```
sign
        "Answer 1 if the receiver is greater than zero,
         answer -1 if the receiver is less than zero,
         else answer zero."
    self strictlyPositive
        ifTrue: [^1].
    self negative
        ifTrue: [^-1].
    ^0!

sin
    |unEntero otroEntero unRacional resul|
    resul := nil.    "error"
    unRacional := 7071/10000.
    unEntero := self \\ 180.
    otroEntero := self // 180.
    ((otroEntero \\ 2) = 0) ifTrue:
       [otroEntero := 1]
    ifFalse:
       [otroEntero := 1 negated].
    (unEntero = 90)  ifTrue: [resul := otroEntero*1].
    (unEntero = 45)  ifTrue: [resul := otroEntero*unRacional].
    (unEntero = 0)   ifTrue: [resul := 0].
    (unEntero = 135) ifTrue: [resul :=  (otroEntero*unRacional)].
^resul!

sinres
        "Answer a Float which is the sine of the receiver.
         The receiver is an angle measured in radians."
    ^self asFloat sin!

sqrt
        "Answer a Float which is the
         square root of the receiver."
    ^self asFloat sqrt!

squared
        "Answer the receiver multiplied by the receiver."
    ^self * self!

storeOn: aStream
        "Append the ASCII representation of the
         receiver to aStream from which the
         receiver can be reconstructed."
    self printOn: aStream!

strictlyPositive
        "Answer true if the receiver is
         greater than zero, else answer false."
    ^self > 0!
```

```
tan
        "Answer a Float which is the tangent of
         the receiver.  The receiver is an angle
         measured in radians."
    ^self asFloat tan!

timesTwoPower: anInteger
        "Answer the result of multiplying the
         receiver by 2 to the exponent anInteger."
    ^self asFloat timesTwoPower: anInteger!

to: aNumber
        "Answer an Interval for the numbers between
         the receiver and the argument aNumber where
         each number is the previous number plus 1."
    ^Interval from: self to: aNumber!

to: sNumber by: iNumber
        "Answer an Interval for the numbers between
         the receiver and the argument sNumber where each
         number is the previous number plus the argument
         iNumber."
    ^Interval from: self to: sNumber by: iNumber!

to: sNumber by: iNumber do: aBlock
        "Evaluate the one argument block aBlock for the
         numbers between the receiver and the argument
         sNumber where each number is the previous number
         plus the argument iNumber."
    | index |
    index := self.
    iNumber > 0
        ifTrue: [
            [index <= sNumber] whileTrue: [
                aBlock value: index.
                index := index + iNumber]]
        ifFalse: [
            [sNumber <= index] whileTrue: [
                aBlock value: index.
                index := index + iNumber]]!

to: aNumber do: aBlock
        "Evaluate the one argument block aBlock for the
         numbers between the receiver and the argument
         aNumber where each number is the previous number
         plus 1."
    | index |
    index := self.
    [index <= aNumber]
        whileTrue: [
            aBlock value: index.
            index := index + 1]!
```

```
truncateTo: aNumber
        "Answer the receiver truncated (towards
         zero) to the nearest multiple of aNumber."
    ^self // aNumber * aNumber!

union1De: unEntero y: otroEntero
   "Calcula el n#mero del punto de contacto n#mero 1"
^(self//(otroEntero*3) > (unEntero*3)) ifTrue: [0] ifFalse:
    [(self\\(otroEntero*3)) = 0 ifTrue:  [self//(otroEntero*3)]
      ifFalse:  [self//(otroEntero*3)+1]  ]!

union2De: unEntero y: otroEntero
   "Calcula el n#mero del punto de contacto n#mero 2"
^(self//(otroEntero*3) > (unEntero*3)) ifTrue: [0] ifFalse:
 [(self\\(otroEntero*3)) = 0 ifTrue: [otroEntero*3]
 ifFalse:
[self\\(otroEntero*3)].  ].! !
ion1De: unEntero y: otroEntero
   "Calcula el n#mero del punto de contacto n#mero 1"
^(self//(otroEntero*3) > (unEntero*3)) ifTrue: [0] ifFalse:
    [(self\\(otroEntero*3)) = 0 ifTrue:  [self//(otroEntero*3)]
      ifFalse:  [self//(otroEntero*3)+1]  ]!

union2De: unEntero y:
```

```
truncateTo: aNumber
        "Answer the receiver truncated (towards
         zero) to the nearest multiple of aNumber."
    ^self // aNumber * aNumber!

union1De: unEntero y: otroEntero
   "Calcula el n#mero del punto de contacto n#mero 1"
^(self//(otroEntero*3) > (unEntero*3)) ifTrue: [0] ifFalse:
    [(self\\(otroEntero*3)) = 0 ifTrue:  [self//(otroEntero*3)]
      ifFalse:  [self//(otroEntero*3)+1]  ]!

union2De: unEntero y: otroEntero
   "Calcula el n#mero del punto de contacto n#mero 2"
^(self//(otroEntero*3) > (unEntero*3)) ifTrue: [0] ifFalse:
 [(self\\(otroEntero*3)) = 0 ifTrue: [otroEntero*3]
 ifFalse:
[self\\(otroEntero*3)].  ].! !
ion1De: unEntero y: otroEntero
   "Calcula el n#mero del punto de contacto n#mero 1"
^(self//(otroEntero*3) > (unEntero*3)) ifTrue: [0] ifFalse:
    [(self\\(otroEntero*3)) = 0 ifTrue:  [self//(otroEntero*3)]
      ifFalse:  [self//(otroEntero*3)+1]  ]!

union2De: unEntero y:
```

APENDICE B: ESPECIFICACION FORMAL

REPRESENTACIONES DE TANGRAMAS

```
ESPEC Puntos;
  IMPORTA TODO DE Reales;
          TODO DE Booleanos;
  EXPORTA TODO;
  GENERO Punto;
  OPERACIONES
    nuevo: -> Punto;
    punto:Real*Real -> Punto;
    x:Punto -> Real;
    y:Punto -> Real;
    esNuevo:Punto -> Bool;
    esIgual:Punto*Punto -> Bool;
    esMenorOIgual:Punto*Punto -> Bool;
    esColineal:Punto*Punto*Punto -> Bool;
VAR p,p2,p3:Punto; n,m:Real;
AXIOMAS
pt1. x(nuevo) = nil
pt2. x(punto(n,m)) = n

pt3. y(nuevo) = nil
pt4. y(punto(n,m)) = m

pt5. esNuevo(nuevo)  = cierto
pt6. esNuevo(punto(n,m)) = falso

pt7. esIgual(nuevo,p) = si esNuevo(p) entonces cierto si no falso
pt8. esIgual(punto(n,m),p2) =
     si x(punto(n,m)) = x(p2) & y(punto(n,m)) = y(p2)
     entonces cierto
     si no falso

pt9.  esMenorOIgual(nuevo,p) = si esNuevo(p) entonces cierto si no falso
pt10. esMenorOIgual(punto(n,m),p2) =
      si x(punto(n,m)) <= x(p2) & y(punto(n,m)) <= y(p2)
      entonces cierto
      si no falso

pt11. esColineal(nuevo,p2,p3) = falso
pt12. esColineal(punto(n,m),p2,p3) =
      si (y(p3)-y(p2))*(x(p2)-x(punto(n,m))) = (y(p2)-y(punto(n,m)))*(x(p3)-x(p2))
      entonces cierto
      si no falso

FIN DE Puntos;
```

```
ESPEC Segmentos;
  IMPORTA TODO DE Puntos;
          TODO DE Reales;
          TODO DE Booleanos;
  EXPORTA TODO;
  GENERO Seg;
  OPERACIONES
     nuevo: -> Seg;
     segmento:Punto*Punto -> Seg;
     origen:Seg -> Punto;
     esquina:Seg -> Punto;
     esNuevo:Seg -> Bool;
     esPunto:Seg -> Bool;
     intersección:Seg*Seg -> Seg;
     intersecciónAux1:Real*Real*Real*Real*Real*Real*Punto*Punto*Punto*Punto*Seg
                      *Seg -> Seg;
     intersecciónAux2:Seg*Seg -> Seg;
     seIntersectan:Seg*Seg -> Bool;
     xba:Seg -> Real;
     yba:Seg -> Real;
     xcd:Seg -> Real;
     ycd:Seg -> Real;
     xca:Seg*Seg -> Real;
     yca:Seg*Seg -> Real;
     e:Seg*Seg -> Real;
VAR s, s1, s2: Seg; p, p1, p2, a, b, c, d: Punto; xba,yba,xcd,ycd,e: Real;
AXIOMAS

se1. origen(nuevo) = nil
se2. origen(segmento(p1,p2)) = p1

se3. esquina(nuevo) = nil
se4. esquina(segmento(p1,p2)) = p2

se5. esNuevo(nuevo)    = cierto
se6. esNuevo(segmento(p1,p2)) = falso

se7. esPunto(nuevo) = falso
se8. esPunto(segmento(p1,p2)) =
     si esIgual(p1,p2)
     entonces cierto
     si no falso
---Regresa: nuevo, si no se tocan;seg, si se tocan(ya sea punto o  segmento);
--nil si se cruzan;error, si no son polígonos.
se9.  intersección(nuevo,s2) = nuevo
se10. intersección(segmento(p1,p2),s2) =
      si esPunto(segmento(p1,p2)) or esPunto(s2)
      entonces error --un polígono no debe tener segmentos degenerados en punto
      si no intersecciónAux1(xba(segmento(p1,p2)),yba(segmento(p1,p2)),
                  xca(segmento(p1,p2),s2),yca(segmento(p1,p2),s2),
                  xcd(s2),ycd(s2),
                  origen(segmento(p1,p2)),esquina(segmento(p1,p2)),
                  origen(s2),esquina(s2),
                  e(segmento(p1,p2),s2),segmento(p1,p2),s2)
```

```
se11. intersecciónAux1(xba,yba,xca,yca,xcd,ycd,a,b,c,d,e,nuevo,s2) =
      error  --condición inalcanzable
se12. intersecciónAux1(xba,yba,xca,yca,xcd,ycd,a,b,c,d,e,segmento(p1,p2),s2) =
      si e = 0 --paralelas o colineales
      entonces
         si yba <> 0
         entonces
            si xba <> 0
            entonces
               si  y(a) - ((xba/yba) * x(a))) <> y(c) - ((xba/yba) * x(c)))
               entonces nuevo --segmentos paralelos que  no se tocan--
               si no
                  si  not seIntersectan(segmento(p1,p2),s2)
                  entonces nuevo --segmentos colineales que no se tocan--
                  si no intersecciónAux2(segmento(p1,p2),s2) --segmento resultante--
            si no
               si  x(b) <> x(c)
               entonces nuevo --segmentos paralelos que no se tocan--
               si no
                  si  not seIntersectan(segmento(p1,p2),s2)
                  entonces  nuevo --segmentos colineales que no se tocan--
                  si no  intersecciónAux2(segmento(p1,p2),s2) --segmento resultante--
         si no
            si  y(b) <> y(c)
            entonces nuevo --segmentos paralelos que no se tocan--
            si no
               si  not seIntersectan(segmento(p1,p2),s2)
               entonces  nuevo --segmentos colineales que no se tocan--
               si no  intersecciónAux2(segmento(p1,p2),s2) --segmento resultante--
      si no --segmentos con pendiente diferente
         si ( 0 =< xca*ycd - xcd*yca/e =< 1) & ( 0 =< xba*yca - yba*xca/e =< 1)
         entonces
            si (xca*ycd - xcd*yca/e = 1) or (xca*ycd - xcd*yca/e =  0) or
               (xba*yca - yba*xca/e = 1) or (xba*yca - yba*xca/e =  0)
            entonces
               si (xca*ycd - xcd*yca/e = 0)
               entonces segmento(a,a)
               si no
                  si  (xca*ycd - xcd*yca/e = 1)
                  entonces segmento(b,b)
                  si no
                     si (xba*yca - yba*xca/e = 0)
                     entonces segmento(b,b)
                     si no
                        si (xba*yca - yba*xca/e = 1)
                        entonces segmento(b,b)
                        si no  error --condición inalcanzable--
            si no  nil --los segmentos se intersectan, los tangramas se encuman--
         si no nuevo --los segmentos no se intersectan--
```

```
se13. intersecciónAux2(nuevo,s2) = error
se14. intersecciónAux2(segmento(p1,p2),s2) =
      si esMenorOIgual(origen(segmento(p1,p2)),origen(s2))
      entonces
         si esMenorOIgual(esquina(segmento(p1,p2)),esquina(s2))
         entonces segmento(origen(s2),esquina(segmento(p1,p2)))
         si no segmento(origen(s2),esquina(s2))
      si no
         si esMenorOIgual(esquina(segmento(p1,p2)),esquina(s2))
         entonces segmento(origen(segmento(p1,p2)),esquina(segmento(p1,p2)))
         si no segmento(origen(segmento(p1,p2)),esquina(s2))

se15. seIntersectan(nuevo,s2) = falso
se16. seIntersectan(segmento(p1,p2),s2) =
      si esMenorOIgual(origen(segmento(p1,p2)),esquina(s2)) &
           esMenorOIgual(origen(s2),esquina((segmento(p1,p2)))
      entonces cierto
      si no falso

se17. xba(nuevo) = error
se18. xba(segmento(p1,p2)) =
      si esMenorOIgual(esquina(segmento(p1,p2)),origen(segmento(p1,p2)))
      entonces x(origen(segmento(p1,p2)))-x(esquina(segmento(p1,p2)))
      si no x(esquina(segmento(p1,p2)))-x(origen(segmento(p1,p2))

se19. yba(nuevo) = error
se20. yba(segmento(p1,p2)) =
      si esMenorOIgual(esquina(segmento(p1,p2)),origen(segmento(p1,p2)))
      entonces y(origen(segmento(p1,p2)))-y(esquina(segmento(p1,p2)))
      si no y(esquina(segmento(p1,p2)))-y(origen(segmento(p1,p2)))

se21. xcd(nuevo) = error
se22. xcd(segmento(p1,p2)) =
      si esMenorOIgual(esquina(segmento(p1,p2)),origen(segmento(p1,p2)))
      entonces x(esquina(segmento(p1,p2)))-x(origen(segmento(p1,p2)))
      si no x(origen(segmento(p1,p2)))-x(esquina(segmento(p1,p2)))

se23. ycd(nuevo) = error
se24. ycd(segmento(p1,p2)) =
      si esMenorOIgual(esquina(segmento(p1,p2)),origen(segmento(p1,p2)))
      entonces y(esquina(segmento(p1,p2)))-y(origen(segmento(p1,p2)))
      si no y(origen(segmento(p1,p2)))-y(esquina(segmento(p1,p2)))

se25. xca(nuevo,s2) = error
se26. xca(segmento(p1,p2),s2) =
      si esMenorOIgual(esquina(segmento(p1,p2)),origen(segmento(p1,p2)))
      entonces
         si esMenorOIgual(esquina(s2),origen(s2))
         entonces x(origen(s2))-x(origen(segmento(p1,p2)))
         si no x(esquina(s2))-x(origen(segmento(p1,p2)))
      si no
         si esMenorOIgual(esquina(s2),origen(s2))
         entonces x(origen(s2))-x(esquina(segmento(p1,p2)))
         si no x(esquina(s2))-x(esquina(segmento(p1,p2)))
```

```
se27. yca(nuevo,s2) = error
se28. yca(segmento(p1,p2),s2) =
      si esMenorOIgual(esquina(segmento(p1,p2)),origen(segmento(p1,p2)))
      entonces
         si esMenorOIgual(esquina(s2),origen(s2))
         entonces y(origen(s2))-y(origen(segmento(p1,p2)))
         si no y(esquina(s2))-y(origen(segmento(p1,p2)))
      si no
         si esMenorOIgual(esquina(s2),origen(s2))
         entonces y(origen(s2))-y(esquina(segmento(p1,p2)))
         si no y(esquina(s2))-y(esquina(segmento(p1,p2)))

se29. e(nuevo,s2) = error
se30. e(segmento(p1,p2),s2) = xba(segmento(p1,p2))*ycd(segmento(p1,p2))-
                              xcd(segmento(p1,p2))*yba(segmento(p1,p2))

FIN DE Segmentos;
```

Especificación Formal.

```
ESPEC Angulos;
  IMPORTA TODO DE Naturales;
          TODO DE Reales;
          TODO DE Booleanos;
  EXPORTA TODO;
  GENERO Angulo;
  OPERACIONES
    nuevo: -> Angulo;
    ángulo: Nat -> Angulo;
VAR  n:Nat;
AXIOMAS
an1. ángulo(n) =
       si n\\45 <> 0
       entonces error -todos los ángulos del juego son múltiplos de 45 grados--
       si no n
FIN DE Angulos;
```

```
ESPEC Aristas;
  IMPORTA TODO DE Angulos;
          TODO DE Reales;
          TODO DE Booleanos;
  EXPORTA TODO;
  GENERO Ari;
  OPERACIONES
    nueva: -> Ari;
    arista:Real*Angulo-> Ari;
    longitud:Ari-> Real;
    ángulo:Ari-> Angulo;
    esNueva:Ari-> Bool;
    esIgual:Ari*Ari-> Bool;
VAR a,a1,a2:Ari; n,m:Real; alfa:Angulo;
AXIOMAS

ar1. longitud(aristaNueva) = nil
ar2. longitud(arista(x,alfa)) = x

ar3. ángulo(aristaNueva) = nil
ar4. ángulo(arista(x,alfa)) = alfa

ar5. esNueva(aristaNueva)   = cierto
ar6. esNueva(arista(x,alfa)) = falso

ar7. esIgual(aristaNueva,a) = si esNueva(a) entonces cierto si no falso
ar8. esIgual(arista(x,alfa),a2) =
     si longitud(arista(x,alfa)) = longitud(a2) &
           ángulo(arista(x,alfa)) = ángulo(a2)
     entonces cierto
     si no falso

FIN DE Aristas;
```

Especificación Formal.

```
ESQUEMA ListasEsq;
[
  PARAM Elementos;
    IMPORTA TODO DE Booleanos;
    EXPORTA TODO;
    GENERO Elem;
      OPERACION
         esIgual:Elem*Elem -> Bool;
    VAR e,e1,e2:Elem;
    AXIOMAS
      esIgual(e,e)=cierto
      esIgual(e,e1)=esIgual(e1,e)
      esIgual(e,e1) y esIgual(e1,e2) => esIgual(e,e2)
  FIN DE Elementos;
];
ESPEC Listas;
   IMPORTA TODO DE Elementos;
           TODO DE Naturales;
   EXPORTA TODO;
   GENERO Lista;
     OPERACIONES
       vacía: -> Lista;
       meteIzq:Elem*Lista -> Lista;
       esVacía:Lista -> Bool;
       sacaDer:Lista -> Lista;
       izq:Lista -> Elem;
       der:Lista -> Elem;
       longitud:Lista -> Nat;
       miembro:Elem*Lista:Elem*Lista -> Bool;
       numOcurr:Elem*Lista -> Nat;
       rotaDer:Lista -> Lista;
       extraeElem:Lista -> Lista
       sacaRepetidos:Lista -> Lista;
       concatena:Lista*Lista -> Lista;
VAR l,l1,l2:Lista; e,e1,e2:Elem;
AXIOMAS

li1. esVacía(vacía) = cierto
li2. esVacía(meteIzq(e,l) =  falso

li3. sacaDer(vacía) = error --no se puede sacar nada de una lista vacía
li4. sacaDer(meteIzq(e,l) =
     si esVacía(l)  entonces vacía si no meteIzq(e,sacaDer(l))

li5. izq(vacía) = nil
li6. izq(meteIzq(e,l) =  e

li5. der(vacía) = nil
li6. der(meteIzq(e,l) =
     si esVacía(l)  entonces e si no sacaDer(l))
```

```
li7. longitud(vacía) = 0
li8. longitud(meteIzq(e,l)) =  longitud(l) + 1

li9.  miembro(e,vacía) = falso
li10. miembro(e1,meteIzq(e2,l))
      si esIgual(e1,e2)
      entonces cierto
      si no miembro(e1,l)

li11  numOcurr(e,vacía) = 0
li12. numOcurr(e1,meteIzq(e2,l)) =
      si esIgual(e1,e2)
      entonces numOcurr(e1,l) + 1
      si no numOcurr(e,l)

li13. rotaDer(vacía) = vacía
li14. rotaDer(meteIzq(e,l))= meteIzq(der(meteIzq(e,l)),sacaDer(meteIzq(e,l)))

li15. extraeElem(e,vacía) = vacía
li16. extraeElem(e1,meteIzq(e2,l)) =
      si esIgual(e1,e2)
      entonces extraeElem(e1,l)
      si no meteIzq(e2,extraeElem(e1,l))

li17. sacaRepetidos(vacía) = vacía
li18. sacaRepetidos(meteIzq(e,l)) =
      si miembro(e,l)
      entonces sacaRepetidos(e)
      si no   meteIzq(e,sacaRepetidos(l))

li19. concatena(vacía,l) = l
li20. concatena(meteIzq(e,l1),l2) = meteIzq(e,concatena(l1,l2))

FIN DE Listas;

FIN DE ESQ Listas;
```

```
-- Tangrama de Puntos
-- Instanciacion de la espec ListaPunt


INSTANCIA ListasEsq;
  RENOMBRA Lista COMO ListaPunt;
  CON Elementos COMO Puntos,
      Elem COMO Punto;
      esIgual COMO esIgual;
FIN DE INSTANCIA ListasEsq;



ESPEC TangramaComoListaDePuntos;

  IMPORTA TODO DE ListaPunt;(es el módulo o el género?)
          TODO DE Segmentos;
          TODO DE Puntos;
          TODO DE Booleanos;
          TODO DE Naturales;
  EXPORTA TODO DE (?) ListaPunt; (son las                de listas?)
          unión DE TangramaComoListaDePuntos;
  OPERACIONES
      esLibre:Punto*ListaPunt -> Bool;
      libre1:ListaPunt -> Punto;
      libre2:ListaPunt -> Punto;
      hayPuntosNoLibres:ListaPunt -> Bool;
      recorreHastaSegundoColineal:ListaPunt*ListaPunt -> ListaPunt;
      restaYRota:ListaPunt*ListaPunt -> ListaPunt;
      restaYRotaAux:ListaPunt*ListaPunt -> ListaPunt;

      dePuntosATortuga:ListaPunt -> Tortuga;
      dePuntosATortugaAux:ListaPunt*Punto*Angulo -> Tortuga;
      dePuntosASegmentos:ListaPunt*Punto -> ListaSeg;

      intersección:ListaPunt*ListaPunt -> ListaPunt;
      esDiscontinua:ListaPunt -> Bool;
      sacaRepetidos:ListaPunt -> ListaPunt;
      sacaColineales:ListaPunt -> ListaPunt;
      sacaColinealesAux:ListaPunt*Nat -> ListaPunt;
      unión:ListaPunt*ListaPunt -> ListaPunt;
VAR lp,lp1,lp2,inter:ListaPunt; k:Nat; p,p1,p2,der: Punto;
AXIOMAS

lp1. esLibre(e,vacía) = falso
lp2. esLibre(e1,meteIzq(e2,lp)) =
     si numOcurr(e1,meteIzq(e2,lp))\\2 = 1
     entonces cierto
     si no falso
```

```
lp3. hayPuntosNoLibres(vacía,inter) = falso
lp4. hayPuntosNoLibres(meteIzq(e,lp),inter)=
     si miembro(e,inter) & noesLibre(e,inter)
     entonces cierto
     si no hayPuntosNoLibres(lp,inter)

lp5. libre1(vacía) = nil --intersección discontinua--
lp6. libre1(meteIzq(e,lp)) =
     si esLibre(e,meteIzq(e,lp))
     entonces e
     si no libre1(extraeElem(e,lp))

lp7. libre2(vacía) = nil --intersección discontinua--
lp8. libre2(meteIzq(e,lp)) =
     si esLibre(e,meteIzq(e,lp))
     entonces libre1(extraeElem(e,lp))
     si no libre2(extraeElem(e,lp))

lp9.  recorreHastaSegundoColineal(vacía,inter,k) = vacía
lp10. recorreHastaSegundoColineal(meteIzq(e,lp1),inter,k) =
      si longitud(meteIzq(e,lp1)) < k
      entonces error --debe haber dos puntos colineales si no    --
                     --hay elementos nolibres en la intersección --
      si no
         si esColineal(der(meteIzq(e,lp1)),libre1(inter),libre2(inter))
         entonces
            si esColineal(der(rotaDer(meteIzq(e,lp1))),libre1(inter),
                 libre2(inter))
            entonces rotaDer(meteIzq(e,lp1))
            si no meteIzq(e,lp1)
         si no recorreHastaSegundoColineal(rotaDer(meteIzq(e,lp1)),inter,k+1)

lp11. restaYRota(vacía,inter) = vacía
lp12. restaYRota(meteIzq(e,lp1),inter) =
      si hayPuntosNoLibres(meteIzq(e,lp1),inter)
      entonces restaYRotaAux(sacaDer(meteIzq(e,lp1)),inter)
      si no  recorreHastaSegundoColineal(meteIzq(e,lp1),inter,0)

lp13. restaYRotaAux(vacía,inter) = vacía
lp14. restaYRotaAux(meteIzq(e,lp1),inter) =
      si hayPuntosNoLibres(meteIzq(e,lp1),inter)
      entonces
         si miembro(der(meteIzq(e,lp1)),inter) & noesLibre(der(meteIzq(e,lp1)),inter
         entonces  restaYRotaAux(sacaDer(meteIzq(e,lp1)),inter)
         si no     restaYRotaAux(rotaDer(meteIzq(e,lp1)),inter)
      si no  meteIzq(e,lp1)
```

```
1a7. dePuntosATortuga(vacía)= vacía
1a8. dePuntosATortuga(meteIzq(e,lp),inic) =
       rotaAngulo(dePuntosATortugaAux(meteIzq(e,lp),e,0))

1a7. dePuntosATortugaAux(vacía)= error --condición inalcanzable--
1a8. dePuntosATortugaAux(meteIzq(e1,lp),e2,alfa) =
     si esVacía(lp)
     entonces
       si y(e2) - y(e1) = 0
       entonces
         meteIzq(
           arista(
             abs(x(e2),
             0-alfa    ),
           dePuntosATortugaAux(sacaDer(MeteIzq(e,lp)),e2,0)
         )
       si no
         meteIzq(
           arista(
             abs(y(e2)-y(e1))/sen(arcTan((y(e2)-y(e1))/(x(e2)-x(e1)))),
             arcTan((y(e2)-y(e1))/(x(e2)-x(e1)))-alfa    ),
           dePuntosATortugaAux(
             sacaDer(MeteIzq(e,lp)),
             e2,
             arcTan((y(der(lp))-y(e1))/(x(der(lp))-x(e1)))
           )
         )

     si no
       si y(der(lp)) - y(e1) = 0
       entonces
         meteIzq(
           arista(
             abs(x(der(lp)),
             0-alfa    ),
           dePuntosATortugaAux(sacaDer(MeteIzq(e,lp)),e2,0)
         )
       si no
         meteIzq(
           arista(
             abs(y(der(lp))-y(e1))/sen(arcTan((y(der(lp))-y(e1))/(x(der(lp))-x(e1)))
             arcTan((y(der(lp))-y(e1))/(x(der(lp))-x(e1)))-alfa    ),
           dePuntosATortugaAux(
             sacaDer(MeteIzq(e,lp)),
             e2,
             arcTan((y(der(lp))-y(e1))/(x(der(lp))-x(e1)))
           )
         )
```

```
lp15. dePuntosASegmentos(vacía,e) = vacía
lp16. dePuntosASegmentos(meteIzq(e1,lp),izq) =
      si esVacía(lp)
      entonces meteIzq(segmento(e1,izq),vacía)
      si no    meteIzq(segmento(e1,izq(lp)),dePuntosASegmentos(lp,izq))

lp17. intersección(vacía,lp) = vacía
lp18. intersección(meteIzq(e,lp1),lp2)=
       deSegmentosAPuntos(intersección(dePuntosASegmentos(meteIzq(e,lp1),e),
                                       dePuntosASegmentos(lp2,Izq(lp2))))

lp19. esDiscontinua(vacía) = cierto --uniones por el vértice o disconexo--
lp20. esDiscontinua(meteIzq(e,lp)) =
      si esLibre(e,meteIzq(e,lp))
      entonces
         si libre2(extraeElem(e,lp)) <> nil
         entonces cierto
         si no falso
      si no
         si libre2(extraeElem(e,lp))  = nil
         entonces cierto
         si no falso

lp21. sacaColineales(vacía) = vacía
lp22. sacaColineales(meteIzq(e,lp)) =
      si longitud(sacaColinealesAux((meteIzq(e,lp)),0) <= 2
      entonces error --el polígono resultante de la unión de dos polígonos--
                     --debe ser vacío o tener más de dos vértices no coli --
                     --neales--
      si no sacaColinealesAux((meteIzq(e,lp),0)

lp23. sacaColinealesAux(vacía,k) = vacía
lp24. sacaColinealesAux(meteIzq(e,lp),k) =
      si k >= longitud(meteIzq(e,lp))
      entonces meteIzq(e,lp)
      si no
         si esColineal(e,der(lp),izq(lp))
         entonces sacaColinealesAux(lp,k)
         si no   sacaColinealesAux(rotaDer(meteIzq(e,lp)),k+1)

lp25. unión(vacía,lp) = c
lp26. unión(meteIzq(e,lp1),lp2) =
      si esDiscontinua(intersección(meteIzq(e,lp1),lp2))
      entonces nil --ensayo inválido--
      si no sacaColineales(sacaRepetidos(
              concatena(restaYRota(meteIzq(e,lp1),intersección(meteIzq(e,lp1),lp2)),
                        restaYRota(12,intersección(meteIzq(e,lp1),lp2)))  ))

FIN DE TangramaComoListaDePuntos;
```

Especificación Formal.

```
-- Tangrama de segmentos
-- Instanciacion de la espec ListaSeg

INSTANCIA ListasEsq;
  RENOMBRA Lista COMO ListaSeg;
  CON Elementos COMO Segmentos,
      Elem COMO Seg;
      esIgual COMO esIgual;
FIN DE INSTANCIA ListasEsq;


ESPEC TangramaComoListaDeSegmentos;

  IMPORTA TODO DE ListaSeg;
          TODO DE ListaPunt;
          TODO DE Segmentos;
          TODO DE Booleanos;
 EXPORTA TODO DE (?) ListaSeg;
          interseccion       DE TangramaComoListaDeSegmentos;
          deSegmentosAPuntos DE TangramaComoListaDeSegmentos;
  OPERACIONES
      deSegmentosAPuntos:ListaSeg -> ListaPunt;
      hayUniónPorElVérticeAux:ListaSeg -> Bool;
      hayUniónPorElVértice:ListaSeg -> Bool;
      sacaPuntos:ListaSeg -> ListaSeg;
      intersecciónAux:SE*ListaSeg -> ListaSeg;
      intersección:ListaSeg*ListaSeg -> ListaSeg;
  VAR ls,ls1,ls2:ListaSeg; s,s1,s2:Seg;
AXIOMAS

ls1. deSegmentosAPuntos(vacía) = vacía
ls2. deSegmentosAPuntos(meteIzq(s,ls))=
       meteIzq(origen(s),deSegmentosAPuntos(ls))

ls3. hayUniónPorElVérticeAux(s,vacía) = cierto
ls4. hayUniónPorElVérticeAux(s1,meteIzq(s2,ls))=
     si (esIgual(origen(s1),origen(s2)) or esIgual(origen(s1),esquina(s2)) )
           & (noesIgual(origen(s2),esquina(s2))

     entonces falso
     si no hayUniónPorElVérticeAux(s1,ls)

ls5. hayUniónPorElVértice(vacía) = falso
ls6. hayUniónPorElVértice(meteIzq(s,ls))=
     si esIgual(origen(s),esquina(s))
     entonces hayUniónPorElVérticeAux(s,ls) or hayUniónPorElVértice(ls)
     si no hayUniónPorElVértice(ls)

ls7. sacaPuntos(vacía) = vacía
ls8. sacaPuntos(meteIzq(s,ls))=
     si esIgual(origen(s),esquina(s))
     entonces sacaPuntos(ls)
     si no meteIzq(s,sacaPuntos(ls))
```

```
--regresa nil, si se cruzan; vacía, si no se tocan--
ls9.  intersecciónAux(s,vacía) = vacía
ls10. intersecciónAux(s1,meteIzq(s2,ls))=
      si intersección(s1,s2) = nil
      entonces nil "los tangramas se enciman"
      si no
         si esNuevo(intersección(s1,s2)) "los segmentos no se tocan"
         entonces intersecciónAux(s1,ls)
         si no meteIzq(intersección(s1,s2), intersecciónAux(s1,ls))

--regresa vacía, si se enciman, no se tocan o hay uniones por el vértice;--
--o sea, vacía cuando el ensayo es inválido                              --
ls11. intersección(vacía,ls) = vacía
ls12. intersección(meteIzq(s,ls1),ls2)=
      si intersecciónAux(s1,ls2) = nil
      entonces vacía "los tangramas se enciman"
      si no
         si hayUniónPorElVértice(
                  concatena(intersecciónAux(s,ls2), intersección(ls1,ls2)))
         entonces vacía "hay uniones por el vértice"
         si no sacaPuntos(concatena(intersecciónAux(s,ls2), intersección(ls1,ls2)))

FIN DE TangramaComoListaDeSegmentos;
```

```
-- Poligono como aristas
-- Instanciacion de la espec ListaAristas


INSTANCIA ListasEsq;
  RENOMBRA Lista COMO Tortuga;
  CON Elementos COMO Aristas,
      Elem COMO Arista;
      esIgual COMO esIgual;
FIN DE INSTANCIA ListasEsq;

ESPEC TangramaComoTortuga;

  IMPORTA TODO DE Tortuga;
          TODO DE Booleanos;
  EXPORTA TODO DE (?) ListaAristas;
          sonIguales DE TangramaComoTortuga;
  OPERACIONES
      sonIdénticas:Tortuga*Tortuga -> Bool;
      sonIgualesAux:Tortuga*Tortuga*Nat. -> Bool;
      sonIguales:Tortuga*Tortuga -> Bool;
      aristaNúm:Tortuga*Nat-> Arista;
      rotaAngulo:Tortuga -> Tortuga;
      rotaAnguloAux:Tortuga*Angulo*Nat->Tortuga;

      deTortugaAPuntos:Tortuga*PtoYAng->ListaPunt;

      cuadrado:->Tortuga
      romboide:->Tortuga
      trianChico:->Tortuga
      trianMedian:->Tortuga
      trianGde:->Tortuga

  VAR la,la1,la2:Tortuga; a,a1,a2:Aristas; k:Natural;p:Punto;inic:PtoYAng;
          lp:ListaPun;
AXIOMAS


la1. sonIdénticas(vacía,la) =
     si esVacía(la) entonces cierto si no falso
la2. sonIdénticas(meteIzq(a,la1),la2) =
     si esVacía(la2)
     entonces falso
     si no
        si esIgual(der(meteIzq(a,la1)),der(la2))
        entonces sonIdénticas(sacaDer(meteIzq(a,la1)),salaDer(la2))
        si no falso
```

```
la3. sonIgualesAux(vacía,la,k) = sonIdénticas(vacía,la)
la4. sonIgualesAux(meteIzq(a,la1),la2,k) =
     si (k < 0)
     entonces error --la llamada estuvo mal--
       si no
       si k >= longitud(meteIzq(a,la1))
       entonces falso
       si no
          si sonIdénticas(meteIzq(a,la1),la2)
          entonces cierto
          si no  sonIgualesAux(rotaDer(meteIzq(a,la1)),k+1)

la5. sonIguales(vacía,la) = si esVacía(la) entonces cierto si no falso
la6. sonIguales(meteIzq(a,la1),la2) = sonIgualesAux(meteIzq(a,la1),la2,0)

la5. aristaNúm(vacía,k) = error --el polígono no tiene tantas aristas--
la6. aristaNúm(meteIzq(a,la1),k) =
       si k =1
       entonces der(meteIzq(a,la1))
       si no aristaNúm(sacaDer(meteIzq(a,la1)),k-1)
---revisar---
la5. rotaAngulo(vacía) = vacía
la6. rotaAngulo(meteIzq(a,la1)) =
       rotaAnguloAux(meteIzq(a,la1),angulo(der(meteIzq(a,la1))),1)

la5. rotaAnguloAux(vacía,alfa,k) = vacía
la6. rotaAnguloAux(meteIzq(a,la1),alfa,k) =
      si k >= longitud(meteIzq(a,la1))
       entonces
           meteIzq(
             arista(
               longitud(der(meteIzq(a,la1))),
               alfa)
       si no
          rotaAnguloAux(meteIzq(
            arista(
              longitud(der(meteIzq(a,la1))),
              ángulo(izq(meteIzq(a,la1)))   ),
            la1
          ),alfa,k+1)
```

```
1a7. deTortugaAPuntos(vacía)= vacía
1a8. deTortugaAPuntos(meteIzq(a,la),inic) =
       meteIzq(
         punto(
           x(inic)+longitud(der(meteIzq(a,la)))*cos(angulo(inic)),
           y(inic)+longitud(der(meteIzq(a,la)))*sen(angulo(inic))    ),
          deTortugaApuntos(
           la,
           ptoYAng(
             x(inic)+longitud(der(meteIzq(a,la)))*cos(angulo(inic)),
             y(inic)+longitud(der(meteIzq(a,la)))*sen(angulo(inic)),
             angulo(inic) + angulo(der(meteIzq(a,la)))
           )
         )
       )

1a9. cuadrado = meteIzq(meteIzq(meteIzq(meteIzq(
                  arista(5*raíz(2)/2,ángulo(90))  )
                 arista(5*raíz(2)/2,ángulo(90))  )
                 arista(5*raíz(2)/2,ángulo(90))  )
                 arista(5*raíz(2)/2,ángulo(90))  )

1a10. romboide = meteIzq(meteIzq(meteIzq(meteIzq(
                 arista(5,ángulo(180+45))  )
                 arista(5*raíz(2)/2,ángulo(-45))  )
                 arista(5,ángulo(180+45))  )
                 arista(5*raíz(2)/2,ángulo(-45))  )

1a11. trianChico = meteIzq(meteIzq(meteIzq(
                   arista(5,ángulo(180+45)  )
                   arista(5*raíz(2)/2,ángulo(180+90))  )
                   arista(5*raíz(2)/2,ángulo(180+45))  )

1a12. trianMedian = meteIzq(meteIzq(meteIzq(
                    arista(5*raíz(2),ángulo(180+45)  )
                    arista(5,ángulo(180+90))  )
                    arista(5,ángulo(180+45))  )

1a13. trianGde = meteIzq(meteIzq(meteIzq(
                 arista(10,ángulo(180+45)  )
                 arista(5*raíz(2),ángulo(180+90))  )
                 arista(5*raíz(2),ángulo(180+45))  )


FIN DE TangramaComoTortuga;
```