



**UNIVERSIDAD DE CASTILLA-LA MANCHA**  
**ESCUELA SUPERIOR DE INFORMÁTICA**

**INGENIERÍA**  
**EN INFORMÁTICA**

**PROYECTO FIN DE CARRERA**

‘Editor gráfico de diagramas de clases basado en trazos naturales’

Alumno: García Martín-Mantero, Yaiza

**Enero, 2010**



**UNIVERSIDAD DE CASTILLA-LA MANCHA**

**ESCUELA SUPERIOR DE INFORMÁTICA**

Tecnologías y Sistemas de la Información

**PROYECTO FIN DE CARRERA**

‘Editor gráfico de diagramas de clases basado en trazos naturales’

Autor: García Martín-Mantero, Yaiza

Director: Jurado Monroy, Francisco

Tutora Académica: Molina Díaz, Ana Isabel

**Enero, 2010**

**TRIBUNAL:**

**Presidente:**

**Vocal 1:**

**Vocal 2:**

**Secretario:**

**FECHA DE DEFENSA:**

**CALIFICACIÓN:**

**PRESIDENTE**

**VOCAL 1**

**VOCAL 2**

**SECRETARIO**

Fdo.:

Fdo.:

Fdo.:

Fdo.:

## **Resumen**

Dibujar a mano siempre ha sido un medio importante de comunicación y de resolución de problemas tanto para diseñadores como para ingenieros. Se trata de una forma familiar, eficiente y natural de expresar ideas, particularmente en las primeras fases de diseño de software. Desafortunadamente, a la hora de comenzar a escribir código, estos bocetos se dejan atrás.

Los diagramas de clases, son los más comunes en cuanto a modelado de sistemas orientados a objetos se refiere, ya que se emplean durante el proceso de análisis y diseño de los sistemas.

Las herramientas tradicionales CASE (*Computer Automated Software Engineering*), dan al usuario potentes características de edición, e incluso permiten generar automáticamente esqueletos de código a partir de los Diagramas de Clases. Este tipo de herramientas son muy útiles en el diseño de un sistema, sin embargo, obligan al usuario a aprender a usar el software CASE, en lugar de dibujar los diagramas UML como lo harían sobre papel.

Por lo anterior, en este proyecto se propone una aplicación capaz de reconocer los diagramas de clase dibujados a mano alzada sobre una Tablet PC.

## ***Agradecimientos***

A Paco Jurado y a Ana Isabel Molina por la confianza que han depositado en mí.

A Beryl Plimmer por la información ofrecida.

A todos los que me han ayudado en algún momento del proceso.

## Contenido

ÍNDICE DE ILUSTRACIONES.....	vii
ÍNDICE DE TABLAS.....	x
1. INTRODUCCIÓN.....	1
1.1. JUSTIFICACIÓN DEL PROYECTO.....	1
1.2. ESTRUCTURA DEL DOCUMENTO.....	2
1.3. TERMINOLOGÍA.....	3
2. OBJETIVOS DEL PROYECTO.....	4
2.1. HIPÓTESIS DE TRABAJO.....	4
2.2. OBJETIVOS GENERALES.....	4
2.3. OBJETIVOS ESPECÍFICOS.....	5
2.4. LIMITACIONES Y CONDICIONANTES.....	5
3. ANTECEDENTES, ESTADO DE LA CUESTIÓN.....	6
3.1. DISPOSITIVOS PARA INTERACCIÓN NATURAL: TABLET PC.....	7
3.2. DESARROLLO DE APLICACIONES PARA TABLET PC.....	8
3.3. TABLET PC SDK.....	10
3.4. LENGUAJE C#.....	12
3.5. LENGUAJE UML.....	13
3.6. RECONOCIMIENTO DE TRAZOS.....	16
3.6.1. Problemas en el reconocimiento de trazos.....	16
3.6.2. Comprendiendo trazos.....	16
3.6.3. Aplicaciones basadas en el Reconocimiento de Trazos.....	18
3.6.4. Comparativa sobre las aplicaciones basadas en el reconocimiento de diagramas de clases.....	26
3.7. PROCESO.....	29
3.7.1. División de trazos en texto y figuras.....	29
3.7.2. Localizar esquinas.....	30

4.	METODOLOGÍA DE TRABAJO.....	33
4.1.	ESPECIFICACIÓN DE REQUISITOS.....	33
4.2.	FASE DE ANÁLISIS .....	33
4.2.1.	Diagrama de Casos de Uso .....	33
4.2.2.	Priorización de Casos de Uso.....	34
4.2.3.	Descripción de Casos de Uso.....	35
4.3.	FASE DE DISEÑO.....	35
4.3.1.	Diagramas de secuencia .....	35
4.3.2.	Arquitectura Multicapa .....	39
4.4.	DETALLES DE IMPLEMENTACIÓN .....	41
4.4.1.	Visión General .....	41
4.4.2.	Dividir los trazos en texto y figuras.....	42
4.4.3.	Encontrar esquinas .....	47
4.4.4.	Dividir polilíneas en líneas.....	55
4.4.5.	Clasificar líneas .....	56
4.4.6.	Buscar clases .....	57
4.4.7.	Asignar texto a clase.....	62
4.4.8.	Buscar relaciones.....	65
4.4.9.	Crear archivo .chico .....	70
4.4.10.	Borrado.....	73
5.	RESULTADOS .....	76
5.1.	MANUAL DE USUARIO .....	76
5.1.1.	Restricciones .....	83
5.2.	USABILIDAD DE LA APLICACIÓN.....	89
5.3.	COMPARATIVA CON LAS APLICACIONES EXISTENTES .....	92
5.4.	CUMPLIMIENTO DE OBJETIVOS.....	94
6.	CONCLUSIONES Y PROPUESTAS DE MEJORA.....	95

6.1. CONCLUSIONES .....	95
6.2. PROPUESTAS DE MEJORA .....	96
7. REFERENCIAS BIBLIOGRÁFICAS .....	98
ANEXO I.....	103
ANEXO II .....	104
ANEXO III .....	117



## ÍNDICE DE ILUSTRACIONES

Fig. 3.1. Flujo de las <i>APIs</i> del <i>Tablet PC SDK</i> en una aplicación. ....	12
Fig. 3.2. Asociación.....	14
Fig. 3.3. Asociación con navegabilidad.....	14
Fig. 3.4. Herencia. ....	15
Fig. 3.5. Agregación. ....	15
Fig. 3.6. Composición. ....	15
Fig. 3.7. Modo de trazos naturales de Tahuti.....	19
Fig. 3.8. Modo de formales naturales de Tahuti. ....	20
Fig. 3.9. Teclado virtual de Tahuti. ....	21
Fig. 3.10. Teclado virtual de Ideogramic UML.....	22
Fig. 3.11. Conversión de una clase a trazo formal en SketchUML. ....	23
Fig. 3.12. Segmentación en el sistema de reconocimiento de diagramas de Lank. ....	24
Fig. 3.13. Corrección de errores tras la segmentación en el sistema de reconocimiento de diagramas de UML de Lank. ....	25
Fig. 3.14. Reconocimiento de texto en SUMLOW.....	26
Fig. 3.15. Árbol de clasificación de los trazos según Patel .....	30
Fig. 3.16. <i>Stroke</i> original (a) y <i>Stroke</i> con los puntos espaciados una misma distancia (b).31	
Fig. 3.17. Correspondencia entre las esquinas y las distancias menores.....	32
Fig. 4.1. Diagrama de casos de uso.....	34
Fig. 4.2. Diagrama de secuencia de Dibujar.....	36
Fig. 4.3. Diagrama de secuencia de Borrar.....	37
Fig. 4.4. Diagrama de secuencia de Exportar. ....	38
Fig. 4.5. Estudio de técnicas de <i>Divider</i> según Patel .....	43
Fig. 4.6. Esquinas encontradas por la alternativa de la librería. ....	48
Fig. 4.7. Esquinas encontradas por la alternativa de Wolin y Hammond. ....	49
Fig. 4.8. Partes de una clase.....	58

Fig. 4.9. Numeración de los <i>strokes</i> de una clase .....	63
Fig. 4.10. <i>Gesture</i> para borrar .....	73
Fig. 5.1. Dibujar con trazos separados. ....	77
Fig. 5.2. Dibujar con polilíneas: Primer ejemplo. ....	77
Fig. 5.3. Dibujar con polilíneas: Segundo ejemplo. ....	78
Fig. 5.4. Dibujar con polilíneas: Tercer ejemplo. ....	78
Fig. 5.5. Borrado de trazos.....	79
Fig. 5.6. Archivo->Exportar. ....	79
Fig. 5.7. Cuadro de diálogo para guardar el archivo .chico.....	80
Fig. 5.8. Advertencia de que se debe introducir una ruta para el archivo. ....	80
Fig. 5.9. Indicativo de que el archivo se ha creado.....	81
Fig. 5.10. Archivo guardado. ....	81
Fig. 5.11. Código generado en el archivo .chico.....	82
Fig. 5.12. Diagrama importado en el entorno desarrollado por el grupo CHICO. ....	82
Fig. 5.13. Trazos divididos, incorrecto.....	83
Fig. 5.14. Trazos enteros, correcto.....	84
Fig. 5.15. Relaciones a esquinas, incorrecto.....	84
Fig. 5.16. Relaciones a otra parte de las clases que no sean las esquinas, correcto.....	85
Fig. 5.17. Texto fuera de las clases, incorrecto.....	85
Fig. 5.18. Texto en el apartado que le corresponde, correcto. ....	86
Fig. 5.19. Más información aparte del nombre de la clase, los atributos y los métodos, incorrecto. ....	86
Fig. 5.20. Sólo el nombre de la clase, los atributos y métodos, correcto. ....	87
Fig. 5.21. Dibujar primero el nombre de la clase y segundo la línea de separación con los atributos, incorrecto.....	87
Fig. 5.22. Trazos repasados, incorrecto.....	88
Fig. 5.23. Trazos limpios, correcto.....	88

Fig. 5.24. Controles de la aplicación.....	89
Fig. 5.25. Trazos naturales.....	90
Fig. 5.26. Trazos formales. ....	90
Fig. 5.27. Zona editable.....	91
Fig. 5.28. Zona no editable. ....	91
Fig. III.1. Versión errónea de <i>GetCorners</i> .....	117
Fig. III.2. Versión correcta de <i>GetCorners</i> .....	118
Fig. III.3. Versión errónea de <i>PostProcessCorners</i> . ....	119
Fig. III.4. Versión correcta de <i>PostProcessCorners</i> . ....	120
Fig. III.5. Versión errónea de <i>HalfwayCorner</i> .....	120
Fig. III.6. Versión correcta de <i>HalfwayCorner</i> .....	121
Fig. III.7. Versión errónea de <i>IsLine</i> . ....	121
Fig. III.8. Versión correcta de <i>IsLine</i> . ....	121
Fig. III.9. Versión errónea de <i>PathDistance</i> . ....	122
Fig. III.10. Versión correcta de <i>PathDistance</i> . ....	122

## ÍNDICE DE TABLAS

Tabla 3.1. Comparativa de aplicaciones de reconocimiento de diagramas de UML.....	28
Tabla 5.1. Comparativa de las aplicaciones de reconocimiento de diagramas de UML incluyendo la desarrollada. ....	93
Tabla I.1. <i>Gestures</i> empleadas en Ideogramic UML. ....	103

# 1. INTRODUCCIÓN

En este capítulo se va a realizar una breve introducción al presente proyecto fin de carrera, justificando el por qué realizar un proyecto de estas características, la estructura que va a presentar, y los términos no traducidos y empleados a lo largo de la documentación del mismo.

## 1.1. JUSTIFICACIÓN DEL PROYECTO

Dibujar a mano alzada siempre ha sido un medio importante de comunicación y de resolución de problemas tanto para diseñadores como para ingenieros. Se trata de una forma familiar, eficiente y natural de expresar ideas, particularmente en las primeras fases de diseño de software. Desafortunadamente, a la hora de comenzar a escribir código, estos bocetos se dejan atrás [Sezgin et al., 2001; Hammond y Davis, 2002].

El Lenguaje Unificado de Modelado (en adelante UML, por sus siglas en inglés, *Unified Modeling Language*) [Booch et al., 1999; Larman, 1999], es un lenguaje visual de propósito general empleado para especificar, visualizar, construir y documentar sistemas por medio de conceptos orientados a objetos. El lenguaje UML consta de varias partes, entre las que destaca, por considerarse como los cimientos del lenguaje, la vista estática. Dicha vista, describe la estructura de los objetos del sistema sin entrar en detalles del comportamiento dinámico del mismo, y se representa mediante diagramas de clases.

Un Diagrama de Clases [Booch et al., 1998], describe gráficamente la estructura de un sistema mostrando sus clases, atributos y las relaciones entre ellas. Este tipo de diagrama se emplea durante el proceso de análisis y diseño de los sistemas. Se trata de los diagramas más comunes en cuanto a modelado de sistemas orientados a objetos se refiere.

Las herramientas tradicionales CASE (*Computer Automated Software Engineering*), como Rational Rose, dan al usuario potentes características de edición, e incluso permiten generar automáticamente esqueletos de código a partir de los Diagramas de Clases. Este tipo de herramientas son muy útiles en el diseño de un sistema, sin embargo, presentan dos inconvenientes [Lank et al., 2000]:

1. Los desarrolladores trabajan la mayor parte del tiempo de manera independiente en ordenadores separados.
2. Los desarrolladores se ven obligados a aprender a usar el software CASE, en lugar de dibujar los diagramas UML como lo harían sobre papel.

La idea de la que parte el presente proyecto es la de crear un entorno de reconocimiento de diagramas UML realizados a mano alzada, de tal forma que se solucionen los problemas comentados anteriormente. Por un lado, se permitirá la interacción entre usuarios, ya que los diagramas se dibujarán sobre una Tablet PC o una pizarra digital, pudiendo participar todos en el mismo diseño. Y por otro lado, no se necesitaría aprender ninguna herramienta software, ya que el usuario dibujaría los diagramas de la misma forma que lo haría sobre una hoja de papel.

Con todo lo anterior se establece el objetivo principal del presente proyecto: *Crear un sistema de edición y reconocimiento de Diagramas de Clases UML basado en trazos naturales.*

## 1.2. ESTRUCTURA DEL DOCUMENTO

Este documento se va a estructurar de la siguiente manera

- Capítulo 1.- Introducción

Introducción al tema, justificación de la realización del proyecto, estructura y terminología empleada en el mismo.

- Capítulo 2.- Objetivos del Proyecto

Descripción del problema a resolver, así como el entorno de trabajo, explicando los objetivos que se pretende obtener y estableciendo las limitaciones y los condicionantes considerados en la resolución del problema.

- Capítulo 3.- Antecedentes, estado de la cuestión

Conocimientos obtenidos en la búsqueda bibliográfica.

- Capítulo 4.- Metodología de trabajo

Metodología empleada y descripción de las distintas fases del desarrollo del proyecto: análisis, diseño e implementación.

- Capítulo 5.- Resultados

Resultados obtenidos tras haber aplicado la metodología citada en el apartado anterior.

- Capítulo 6.- Conclusiones y propuestas de mejoras

Conclusiones obtenidas en la realización del proyecto, así como propuestas de futuro para posibles mejoras.

- Capítulo 7.- Referencias Bibliográficas

Listado de referencias bibliográficas consultadas en la realización del proyecto.

### 1.3. TERMINOLOGÍA

Puesto que toda la bibliografía consultada está en inglés, se ha intentado hacer las traducciones lo más coherentemente posible, sin embargo existen términos que se usarán en inglés en el presente documento. Dichos términos quedan definidos a continuación:

- *Stroke*. Se entiende por *stroke* cualquier trazo realizado con el dispositivo de entrada, es decir, el tramo de tinta digital dibujado que se crea desde que se apoya el lápiz en la pantalla hasta que se levanta.
- *Gesture*. Se trata de un trazo al que se le ha asignado una función o un significado concreto. Para que sea correctamente reconocido e interpretado debe dibujarse tal y cómo se especifique tanto en forma como en orientación.

## 2. OBJETIVOS DEL PROYECTO

En este capítulo, se enuncia en primer lugar la hipótesis de trabajo del presente proyecto fin de carrera, continuando con una explicación detallada de los objetivos generales y específicos que deberán alcanzarse. Para terminar, se indicarán las limitaciones y condicionantes que deberán cumplirse para llevar a cabo la realización del mencionado proyecto.

### 2.1. HIPÓTESIS DE TRABAJO

Con lo introducido en el capítulo anterior, se establece la hipótesis de trabajo del presente proyecto fin de carrera del siguiente modo: *Es posible crear un sistema de edición y reconocimiento de Diagramas de Clases UML basado en trazos naturales.*

Para alcanzar su consecución, en los siguientes apartados se definirán los objetivos generales y específicos establecidos.

### 2.2. OBJETIVOS GENERALES

El objetivo general de este proyecto es la *construcción de un sistema software que ayude a los Ingenieros Informáticos a modelar Diagramas de Clases a través de trazos naturales*, según el estándar UML que es el estudiado en la mayoría de Escuelas de Informática en la actualidad.

El sistema reconocerá los trazos del usuario, los procesará y los interpretará de forma adecuada. Es importante recalcar que, toda la aplicación se podrá gestionar con el movimiento gestual del usuario, mediante un dispositivo apuntador tipo lápiz y sobre pantalla táctil.

La aplicación permitirá la exportación de los modelos en formato XML para que posteriormente pueda importarse en una aplicación desarrollada por el grupo CHICO de la Escuela Superior de Informática de Ciudad Real de modo que pueda verse su aplicación directa dentro del proceso de desarrollo software.



### 2.3. OBJETIVOS ESPECÍFICOS

A continuación se enumeran los objetivos específicos a cubrir en este proyecto:

- Estudio de las técnicas de *reconocimiento de trazos*.
- Búsqueda de soluciones y/o librerías que den soporte total o parcial a la problemática del reconocimiento de trazos.
- Estudio detallado del dominio en el que se aplicará la interacción basada en trazos naturales: *Diagramas de Clases UML*. Identificación de *elementos*, *relaciones* y *atributos* que componen dichos diagramas [Booch et al., 1999].
- Realizar una interfaz de usuario usable, de interacción fácil e intuitiva, de tal manera que se asemeje lo más posible a una hoja de papel.

### 2.4. LIMITACIONES Y CONDICIONANTES

La aplicación está diseñada para ser ejecutada en una Tablet PC o pizarra digital, pero al estar programada en un ordenador convencional, pueden utilizarse como dispositivos de entrada tanto el ratón como una tableta digital.

La implementación de la aplicación se ha llevado a cabo en un ordenador portátil en el entorno de programación Visual Studio 2008, en lenguaje C#, por lo que se debe disponer del Framework 2.0 para su correcto funcionamiento.

### 3. ANTECEDENTES, ESTADO DE LA CUESTIÓN

En este capítulo se van a desarrollar los conocimientos adquiridos durante la búsqueda bibliográfica. En él se definirá qué se entiende por Tablet PC, para poder explicar posteriormente cada uno de los aspectos que la caracterizan. Entre ellas, cabe mencionar que en el desarrollo de aplicaciones para Tablet PC, deben tenerse en cuenta dos nuevos factores: el uso de un *lápiz* como entrada y el empleo de *tinta digital*. Por esta razón, se explican en el segundo apartado las pautas que deben seguirse para conseguir que una aplicación desarrollada para este tipo de dispositivos no presente errores de usabilidad. Así, en el tercer apartado se hace una breve introducción al kit de desarrollo empleado en la implementación del presente proyecto, y continuando con detalles de implementación, se realiza en el cuarto apartado una breve introducción al lenguaje utilizado para implementar la aplicación.

Una vez realizada la introducción tecnológica, el capítulo se adentrará a mostrar el dominio concreto de aplicación, de manera que se describirán las características gráficas de los Diagramas de Clase UML, ya que van a ser éstos diagramas los que la aplicación desarrollada deberá reconocer.

Tras esto, comenzarán a explicarse los conocimientos adquiridos sobre reconocimiento de trazos. En primer lugar se explica la problemática existente en el desarrollo de aplicaciones de reconocimiento de trazos. En segundo lugar, se exponen, señalando sus características más notables, las técnicas de reconocimiento de trazos existentes. A continuación, y en este mismo apartado, se lleva a cabo un estudio de las aplicaciones existentes en este campo, haciendo especial hincapié en las basadas en el reconocimiento de trazos de Diagramas de Clase UML. Para finalizar con el apartado, se hace una comparativa de éstas últimas aplicaciones mencionadas.

Por último, se explican las técnicas encontradas que permiten llevar a cabo dos puntos en el proceso del desarrollo de la aplicación: la división de los trazos en texto y figuras, y la localización de esquinas en los trazos.

### 3.1. DISPOSITIVOS PARA INTERACCIÓN NATURAL: TABLET PC

Son varios los dispositivos que permiten realizar una interacción a mano alzada con el sistema. Tablet PC, pantallas táctiles, pizarras digitales, tabletas digitales, etc. son algunos de los ejemplos. Sin embargo, los principios en los que se fundamentan todos ellos son los mismos: reconocimiento del trazo realizado por el usuario. Por ello, centraremos nuestra explicación y análisis en el dispositivo conocido como Tablet PC, cuyo uso resulta más extendido, haciendo saber al lector que varios de los principios aplicados a éste pueden ser extendidos al resto de dispositivos que permitan interacción natural.

Existen distintas formas de describir y definir una Tablet PC, entre las cuales, la más completa se enuncia a continuación:

*“Una Tablet PC es un ordenador fino, autónomo y de propósito general con una pantalla interactiva integrada. Generalmente, dicha pantalla es de gran tamaño y acepta un lápiz especial (stylus o pen) como dispositivo de entrada”* [Jarret y Su, 2003]

Cuando se crean aplicaciones para Tablet PC, se encuentran elementos ya conocidos, como almacenamiento en un disco duro, restricciones de memoria y una pantalla. Sin embargo, existen componentes específicos de una Tablet PC que necesitan tenerse en cuenta a la hora de desarrollar una aplicación para este tipo de dispositivos [Crooks, 2004].

Las características esenciales que hacen que un ordenador en particular sea una Tablet PC, se detallan a continuación [Jarret y Su, 2003; Crooks, 2004]:

- *Tamaño y forma:* Es relativamente delgada y presenta una gran pantalla (del tamaño de una hoja de papel o mayor).
- *Peso:* En general, los ordenadores portátiles deben ser ligeros; en el caso de las Tablet PC, debido a sus escenarios de uso, su peso es un factor de gran importancia.

- *Entrada mediante lápiz:* Es cierto que aceptar un lápiz como entrada no es una característica esencial en todas las Tablet PC's, pero la mayoría de los usos serios de las *tablets*, lo requieren. El lápiz resuelve dos de los inconvenientes de los ordenadores portátiles:
  - *Entrada natural:* La gente está familiarizada con el uso de lápices para escribir. Para mucha gente, el lápiz es más fácil de usar que otros dispositivos como los ratones. El uso del lápiz además, permite dar soporte en una serie de escenarios que serían difícilmente practicables con otras metodologías de entrada. Algunos de estos escenarios son la escritura a mano y el dibujo a mano alzada.
  - *Contexto social:* Hay mucha gente que se distrae con el sonido de las teclas o que no ve lo que tiene enfrente por tener la pantalla del ordenador delante. Esto es un problema que se da habitualmente en las reuniones, y que se salva gracias al empleo de las Tablet PC.
- *Tinta digital:* Es una de las características únicas de las *tablets*. Cuando se escribe con el lápiz en una Tablet PC, se trabaja con información en tiempo real, en forma de tinta digital.
- *Autonomía:* Pueden funcionar por completo sin la asistencia de otro ordenador.
- *Propósito General:* Una Tablet PC puede emplearse en cualquiera de los campos en que pueda hacerlo un ordenador personal.

### 3.2. DESARROLLO DE APLICACIONES PARA TABLET PC

Aunque la mayoría de los conocimientos del desarrollo de aplicaciones pueden aplicarse en el mundo del Tablet PC, es necesario poner especial atención en los aspectos relativos a la usabilidad de las aplicaciones [Jarret y Su, 2003], dado que en el desarrollo de aplicaciones para Tablet PC, se introducen dos nuevos factores: el uso de los *lápices* como dispositivos de entrada y la *tinta digital*.

Un equipo de Microsoft realizó entre 1999 y 2002 una serie de estudios para determinar las guías o pautas a seguir para diseñar aplicaciones para este tipo de dispositivos de forma que no presentasen errores de usabilidad. Estas guías se pueden dividir en tres áreas:

- *Usabilidad de la Tablet PC:* Los participantes en el estudio determinaron principalmente dos ventajas de las *tablets* frente a los ordenadores:
  - Se elimina la barrera social en muchos sitios de trabajo, sobre todo en las reuniones (*meetings*) debido a su portabilidad.
  - Son más fáciles de usar en espacios reducidos o incluso de pie gracias al empleo del lápiz como entrada.

El estudio también reveló que los participantes preferían *tablets* con teclado ya que en algunas situaciones la usabilidad mejoraría con éste.

- *Usabilidad de la aplicación:* En cuanto al diseño de las aplicaciones, el estudio reveló los siguientes resultados:
  - Es preferible dejar a un lado los diseños de interfaces de usuario basados en entrada para lápiz (como poner la barra de herramientas en la parte baja de la aplicación en lugar de en la parte superior) y continuar con los tradicionales ya que las aplicaciones diseñadas de esta segunda forma son más consistentes y mejoran la productividad del usuario.
  - Los usuarios prefieren que sus anotaciones permanezcan como tinta digital en lugar de convertirse a texto. Los estudios llevados a cabo por Microsoft no son los únicos que llegan a esta conclusión, ya que en [Hammond y Davis, 2002; Alvarado y Davis, 2004] también se especifica que los usuarios prefieren trabajar con sus trazos porque se distraen con la transformación a trazos formales.
  - Se deben distinguir de alguna forma en la aplicación, las zonas donde se puede introducir tinta digital y en las que no. El problema es que no existe una recomendación oficial para hacer las zonas de entrada de tinta digital visualmente distintas.

- Las aplicaciones no deben trabajar en píxeles en cuanto a interfaces de usuario se refiere, ya que a medida que aumenta el tamaño de la pantalla, los controles aparecen más pequeños.
- Debe tenerse en cuenta la modalidad, es decir, los distintos modos o estados en que la aplicación acepta entradas. Por ejemplo, en el modo “lápiz” la entrada es tratada de forma distinta a como se haría en el modo “borrar”. La modalidad disminuye la eficiencia de la aplicación, por lo que lo ideal sería trabajar con el lápiz de forma natural, sin tener que especificar mediante los distintos modos lo que se desea hacer en cada momento.
- *Usabilidad del lápiz de entrada:* Conseguir focalizar los puntos de interacción entre el lápiz de entrada y la pantalla táctil, es uno de los mayores retos que se presentan en este campo. Los estudios de Microsoft han revelado que los usuarios sólo pueden pulsar sobre áreas mayores de 5 milímetros de tamaño.

### 3.3. TABLET PC SDK

En esta sección se mostrará de forma resumida los principales objetos, elementos y utilidades aportadas por la librería *Tablet PC SDK* y que han sido empleadas en la elaboración del presente proyecto. No se pretende con esta sección mostrar en detalle las particularidades de la tecnología de implementación usada, que podrán ser consultadas por el lector en [Jarret y Su, 2003].

El kit de desarrollo de aplicaciones para Tablet PC es un conjunto de APIs (*Application Programming Interface*) que se pueden dividir en tres grupos, cada uno de los cuales proporciona una porción esencial de funcionalidad para lápiz y tinta [Jarret y Su, 2003]:

- *Tablet Input API (Pen API).* Maneja las características específicas del lápiz de entrada, sus botones, así como aspectos relacionados con la captura de tinta digital y las *gestures* provenientes del movimiento del lápiz.

Para capturar la tinta se emplean dos objetos: *InkCollector* e *InkOverlay*. El primero se utiliza para capturar la tinta que dibuja el usuario, y el segundo es un supertipo del primero, es decir, posee toda su funcionalidad y además añade la posibilidad de borrar y seleccionar la tinta capturada.

- *Ink Data Managed API (Ink API)*. Se encarga de manipular y almacenar la tinta digital captada por la *Tablet Input API*.

En este caso los objetos más destacados son: *Ink* y *Stroke*. El primero contiene toda la tinta capturada, además de los métodos para añadir y eliminar los *strokes* de él. En el objeto *Stroke* se almacena la información correspondiente a un trazo de tinta digital.

- *Ink Recognition API (Recognition API)*. Se encarga de interpretar de forma inteligente la tinta digital.

Aquí, el objeto más importante es *Recognizer*, que representa una máquina de reconocimiento en particular.

A continuación (Véase Fig. 3.1 extraída de [Jarret y su, 2003]), puede verse el flujo seguido entre estas tres APIs cuando participan y colaboran en el contexto de una aplicación de edición y reconocimiento de trazos:

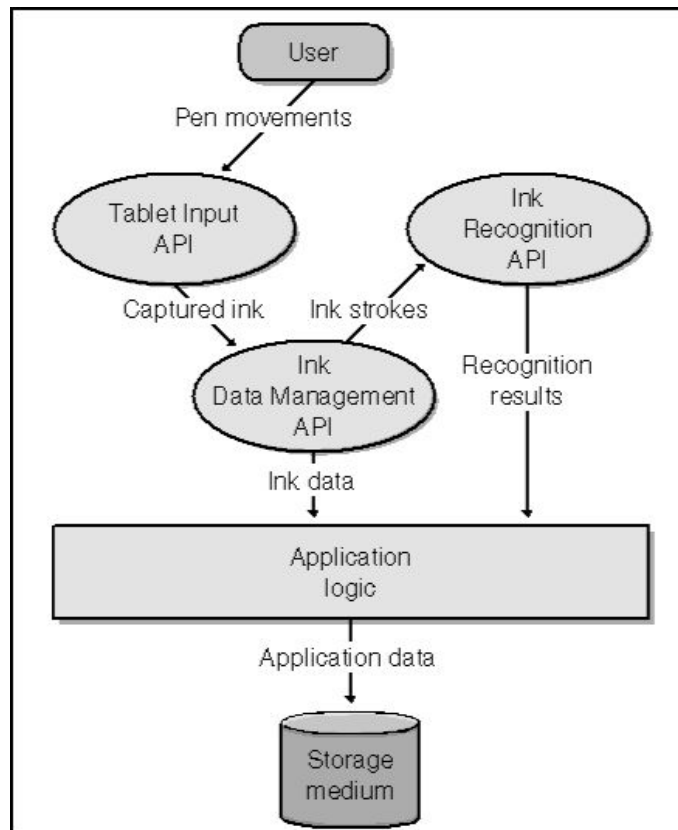


Fig. 3.1. Flujo de las APIs del *Tablet PC SDK* en una aplicación.

En este kit de desarrollo, también se incluyen dos controles que se encargan de integrar las funcionalidades de la tinta y el lápiz con la nueva aplicación. Dichos controles son los siguientes:

- *InkEdit*: Captura la tinta y la convierte opcionalmente en texto.
- *InkPicture*: Se encarga de capturar y mostrar tinta sobre imágenes.

### 3.4. LENGUAJE C#

C# es un lenguaje de programación de propósito general, orientado a objetos y con un sistema de tipos seguro [Albahari, y Albahari, 2007]. Se trata de un lenguaje de programación simple y moderno derivado de C y C++. El objetivo de este lenguaje es mejorar la productividad del programador. C# combina la alta productividad de Microsoft Visual Basic y la eficacia bruta de C++ [Archer, 2001]. El arquitecto principal del lenguaje desde su primera versión es Anders Hejlsberg (creador del Turbo Pascal y de la



arquitectura de Delphi). Se puede trabajar con C# en cualquier plataforma, pero está diseñado para trabajar con el Framework de Microsoft.NET [Albahari, y Albahari, 2007].

C# es una rica implementación del paradigma orientado a objetos, que incluye encapsulación, herencia y polimorfismo. Las características más distinguidas de C# desde una perspectiva orientada a objetos son:

- *Sistema de tipos unificado.* El bloque de construcción fundamental de C# es una encapsulación de datos y funciones llamada *tipo*. C# presenta un sistema de tipos unificado, donde todos los tipos comparten un tipo base común.
- *Clases e interfaces.* En el paradigma de la orientación a objetos puro, el único tipo de datos es la *clase*. En C#, existen otra clase de tipos, uno de los cuales es la *interfaz* (similar a las interfaces de Java). Una interfaz es como una clase, lo único, que es sólo la definición de un tipo, no una implementación. Es particularmente útil en escenarios que requieran múltiples herencias.
- *Propiedades, métodos y eventos.* En el paradigma de la orientación a objetos puro, todas las funciones son *métodos*. En C#, los métodos son sólo un tipo de *funciones miembro*, que incluyen *propiedades* y *eventos*. Las propiedades son funciones miembro que encapsulan una parte del estado del objeto, como el color de un botón o el texto de una etiqueta. Los eventos son funciones miembro que simplifican las actuaciones ante los cambios de estado de los objetos.

### 3.5. LENGUAJE UML

UML (“*Unified Modeling Language*” o Lenguaje Unificado de Modelado) es un lenguaje estandarizado por el *Object Management Group* (OMG) que se utiliza para diseñar, construir y documentar sistemas software mediante el paradigma orientado a objetos [Booch et al., 1999; Polo, 2008]. UML incluye un conjunto muy amplio de diagramas que se utilizan para representar la estructura y el comportamiento de sistemas software.

El diagrama de clases es uno de los diagramas más conocidos y utilizados de UML. Un diagrama de clases muestra la estructura de un sistema o subsistema desde el punto de vista de las clases que lo componen, así como de las relaciones que existen entre tales clases.

Gráficamente, una clase se representa mediante un rectángulo que tiene normalmente tres compartimentos: en el superior se escribe el nombre de la clase, en el central se escribe la lista de campos o atributos de la clase, y en el inferior se escribe la lista de operaciones de la clase.

Las relaciones más destacadas que pueden darse entre las distintas clases son las siguientes [Booch et al., 1999; Polo, 2008]:

- *Asociación*: Ocurre cuando una instancia conoce a otra instancia de la misma o de otra clase, pero ambas instancias pueden vivir separadamente y sin conocerse.

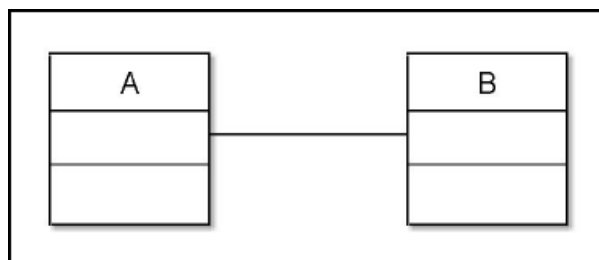


Fig. 3.2. Asociación.

- *Asociación con navegabilidad*: En este caso, sólo la instancia de la clase origen (A) tiene conocimiento de la existencia de la clase destino (B).

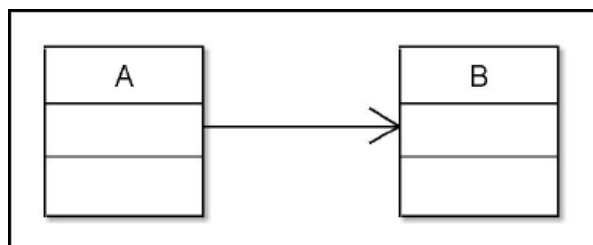


Fig. 3.3. Asociación con navegabilidad.

- *Herencia*: Indica que A es una subclase de B y por ello heredará sus métodos y atributos.

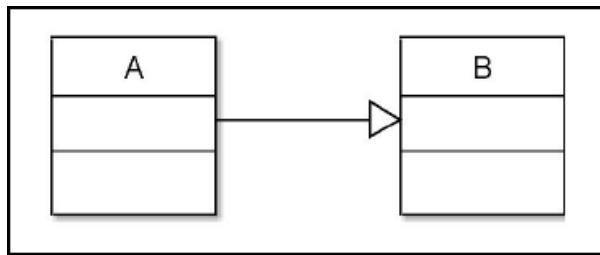


Fig. 3.4. Herencia.

- *Agregación*: Se da cuando dentro de una clase (B) se da una instancia de otra clase (A).

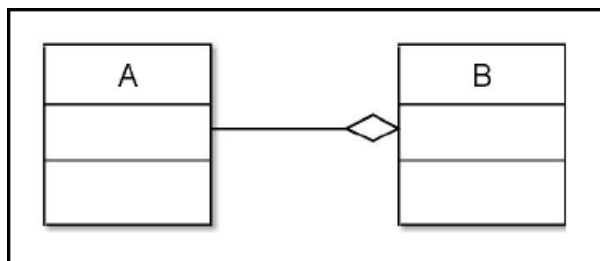


Fig. 3.5. Agregación.

- *Composición*: Se trata de una agregación “fuerte”, en el sentido de que la instancia contenida (A) nace después de la instancia continente (B) y muere antes.

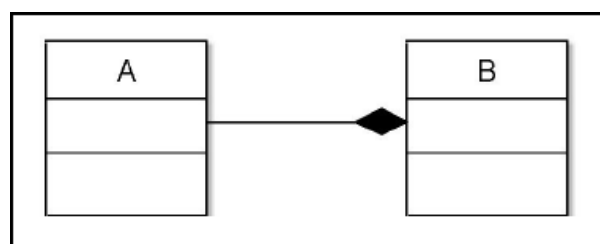


Fig. 3.6. Composición.

## 3.6. RECONOCIMIENTO DE TRAZOS

### 3.6.1. Problemas en el reconocimiento de trazos

Probablemente, la parte más importante a la hora de crear una aplicación que comprenda trazos, es tener un reconocedor flexible pero a la vez eficaz [Alvarado, 2000] ya que, generalmente, los sistemas que permiten una mayor flexibilidad son los que más errores cometen en el reconocimiento de trazos. Cuantos menos errores cometa el sistema, menos tiempo de trabajo perderá el usuario corrigiendo dichos errores. En la práctica, no se puede construir un sistema que no presente errores en el reconocimiento de trazos, pero sí se puede pretender que éste cometa los menos posibles.

### 3.6.2. Comprendiendo trazos

En 1963, Ivan Sutherland creó el sistema *Sketchpad* en un ordenador TX-2 en el MIT (Instituto Tecnológico de Massachusetts) [Sutherland, 1963]. Este sistema fue denominado como el primer ordenador de aplicaciones gráficas. Se creó antes de la invención del ratón, por lo que el dispositivo de entrada consistía en un lápiz luminoso. El usuario podía crear gráficos complicados en dos dimensiones a través de una serie de comandos de edición y comandos de gráficos primitivos. La luz del lápiz se usaba junto con un teclado que permitía a los usuarios crear gráficos primitivos, es decir, líneas y círculos, y editar mediante comandos como “Copiar”. Con ayuda del teclado, y empleando algunas restricciones adicionales, se podía modificar la geometría de las figuras en escena, pudiéndose crear estructuras complejas.

No es hasta 1991, de la mano de Dean Rubine [Rubine, 1991], cuando se crea la primera herramienta de reconocimiento de trazos, GRANDMA.

La disciplina del *reconocimiento de trazos* emplea modalidades y técnicas que permiten que las aplicaciones identifiquen las figuras que el usuario dibuja a mano. Cuando el usuario mueve el lápiz o cualquier otro dispositivo de entrada sobre una superficie digital, dicho movimiento se muestrea mediante datos espacio-temporales. Estos puntos representan los trazos dibujados por el usuario y pueden entonces analizarse. El

reconocimiento de trazos puede llevarse a cabo mediante una o más de las siguientes tres técnicas existentes [Dixon y Hammond, 2009]:

- *Reconocimiento basado en “gestures”*

Esta técnica depende de cómo se han dibujado los trazos, es decir, el orden y dirección de los trazos es esencial a la hora del reconocimiento [Rubine, 1991; Long et al., 1999; Long et al., 2001]. Se requiere un periodo de entrenamiento por el usuario para conocer y aprender las distintas *gestures*. El reconocimiento mediante *gestures* también depende del estilo que el usuario presente al dibujar, por lo que se requiere un periodo de entrenamiento similar para que el sistema aprenda cómo dibuja el usuario.

Las bases de esta técnica se encuentran en el trabajo de Rubine [Rubine, 1991]. En él se describe GRANDMA, una herramienta para reconocer los objetos procesando un trazo simple y determinando sus características. Éste método no rompe el trazo en segmentos o curvas, por lo que cada objeto debe ser dibujado mediante un trazo. Rubine propone trece características que pueden ser empleadas para describir cualquier forma dibujada mediante un trazo. Asimismo, propone dos técnicas para rechazar *gestures* falsas.

El trabajo de Rubine fue extendido posteriormente por Long et al. [Long et al., 2000], quienes determinaron un nuevo conjunto de características consistente en once de las propuestas por Rubine, más seis propias.

Ambos trabajos resultan de gran ayuda en el reconocimiento de *gestures* en dos dimensiones, pero al aplicarlos a los problemas naturales en el reconocimiento de diagramas a mano alzada, la precisión obtenida no es óptima [Paulson et al. 2008a].

Debido a las limitaciones impuestas por los sistemas de reconocimiento basados en *gestures*, los cambios más recientes en reconocimiento de trazos tienden a centrarse en las técnicas de reconocimiento mediante geometría [Paulson et al. 2008a].

- *Reconocimiento basado en visión por computador*

Esta técnica se basa en la apariencia de las formas y generalmente es inflexible ante las variaciones de cómo se dibuja [Oltmans, 2007; Wobbrock et al., 2007]. Se suele usar cuando el dibujo se ha realizado fuera del ordenador y se trae a él, como ocurre con una

hoja de papel escaneada. Al igual que otras técnicas de visión por computador, los trazos pueden analizarse pixel a pixel en lugar de hacerlo con los puntos muestreados.

- *Reconocimiento basado en geometría*

En este caso, las formas se definen por sus propiedades geométricas y por sus restricciones [Apte y Kimura, 1993; Sezgin et al. 2001; Hammond y Davis, 2005; Paulson y Hammond, 2008; Paulson et al., 2008a]. Por ejemplo, una flecha puede describirse como tres líneas, sus longitudes y los ángulos entre ellas.

Esta propuesta permite una mayor flexibilidad en el dibujo, ya que no se requiere que los trazos estén dibujados en un orden y dirección concretos. La desventaja de este tipo de reconocimiento es el numeroso uso de umbrales y jerarquías heurísticas que hacen más complicado analizar y optimizar el sistema [Paulson et al., 2008a].

De las tres aproximaciones anteriores, las más ampliamente usadas son las basadas en *gestures* y las basadas en geometría [Paulson et al., 2008a]. En las siguientes secciones se mostrarán algunas de las aplicaciones que emplean estas técnicas para pasar a analizar posteriormente los pros y contras que presentan cada una de ellas.

### 3.6.3. Aplicaciones basadas en el Reconocimiento de Trazos

Las aplicaciones basadas en el reconocimiento de trazos abarcan un amplio abanico de dominios: Matemáticas [La Viola y Zeleznik, 2004], Ingeniería eléctrica [Gennari et al., 2005; Alvarado y Davis, 2004], diseño de sistemas mecánicos [Alvarado, 2000], química [Ouyang y Davis, 2007], física [Alvarado, 2000], diseño de interfaces de usuario [Landay, 1996; Caetano et al., 2002] juegos educativos [Paulson et al., 2008b], e incluso fonética del Mandarín [Taele y Hammond, 2008; Taele y Hammond, 2010].

Dado que el interés del presente proyecto fin de carrera se centra en los Diagramas de Clases UML, se muestran algunas de las aplicaciones existentes en dicho área:

- *Tahuti*

Se trata de una herramienta de reconocimiento de trazos para diagramas de clases en UML [Hammond y Davis, 2002]. Los trazos que forman parte de la parte gráfica del diagrama se reconocen según su geometría, por lo que el usuario tiene la libertad de dibujar un mismo elemento de distintas formas.

Presenta dos formas de ver el diagrama, una con los trazos como tinta digital (Véase Fig. 3.7 extraída de [Hammond y Davis, 2002]) y otra con los trazos formales (Véase Fig. 3.8 extraída de [Hammond y Davis, 2002]) de tal forma que el usuario puede cambiar la vista para trabajar con el modo que desee.

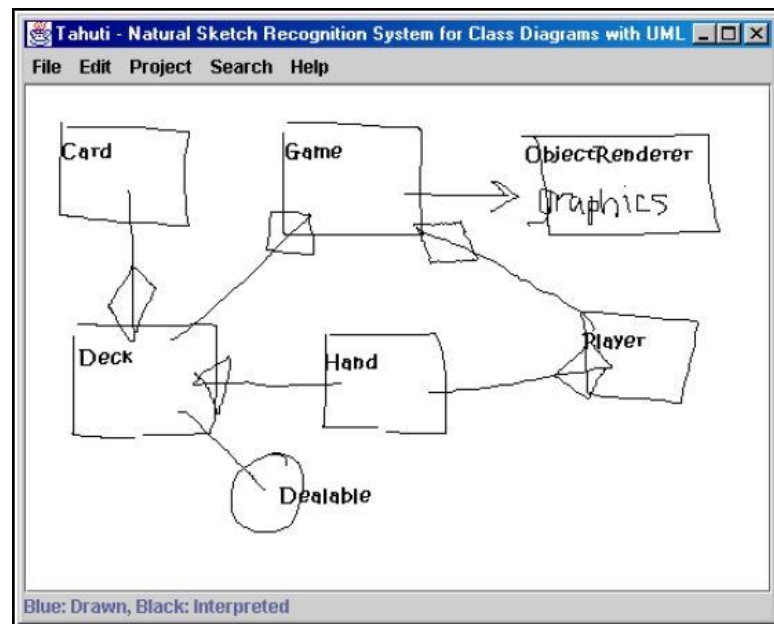


Fig. 3.7. Modo de trazos naturales de Tahuti.

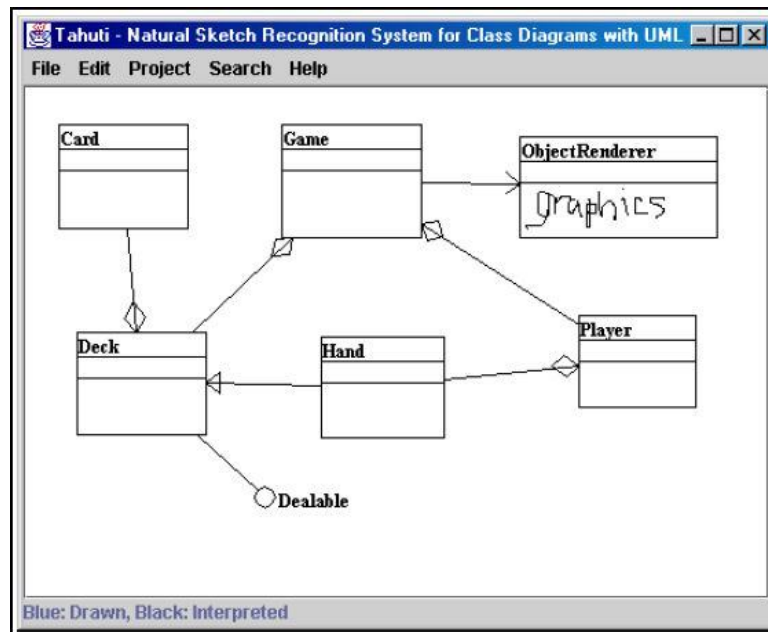


Fig. 3.8. Modo de formales naturales de Tahuti.

Aunque existan dos modos de visionado de los trazos, el instante en que se crean juega un papel importante en el reconocimiento ya que el pre procesado del trazo se lleva a cabo en cuanto termina de dibujarse y por tanto, sigue patrones temporales. Es decir, se tiene la libertad para dibujar el elemento de la manera que se desee siempre que éste se dibuje de seguido. Si por ejemplo se crea una asociación y posteriormente se observa que debería ser una herencia, no se puede pintar el triángulo correspondiente porque el cuerpo de la flecha y la cabeza no se han dibujado uno inmediatamente después del otro.

Mirando las dos imágenes anteriores, se observa que esta aplicación no presenta reconocimiento de texto manual. Para que el texto se reconozca como tal, debe introducirse desde un teclado virtual (Véase Fig. 3.9 extraída de [Hammond y Davis, 2002]). El teclado virtual aparece cuando la clase se reconoce como tal, es decir, en primer lugar debe dibujarse la clase, esperar a que se reconozca y, cuando esto haya ocurrido, hacer doble clic en la clase para que aparezca el teclado virtual y se pueda introducir el texto.



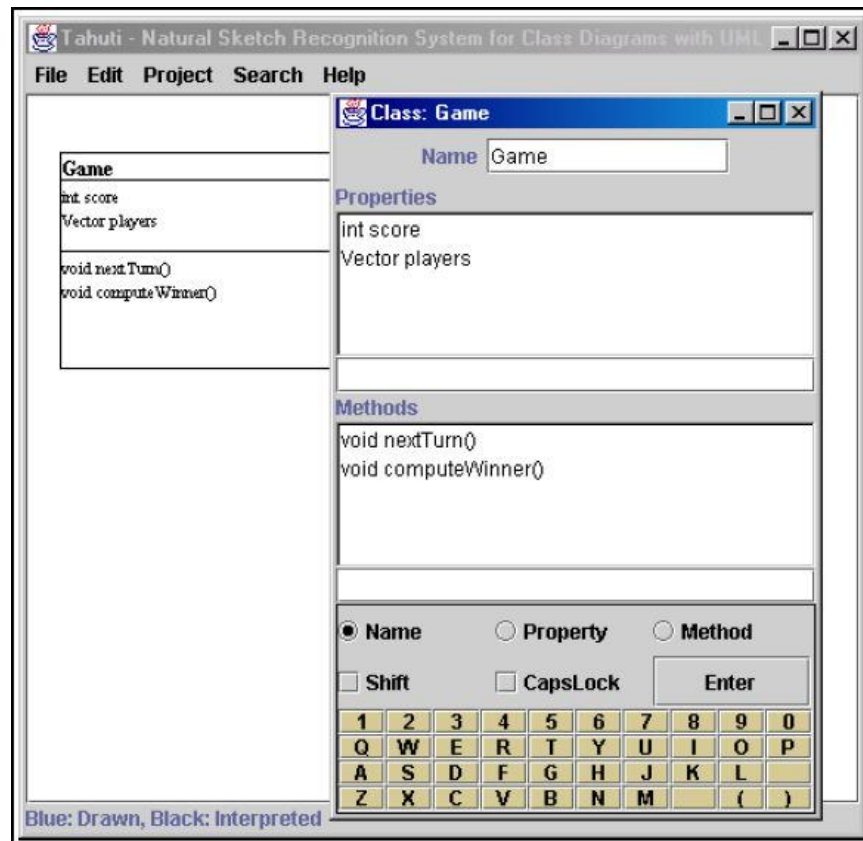


Fig. 3.9. Teclado virtual de Tahuti.

Esta aplicación permite el reconocimiento de relaciones de asociación, herencia y agregación.

- *Ideogramic UML*

Esta aplicación presenta distintos nombres según el año de los artículos que se consulten, así podemos encontrarla como Knight Tool [Damm et al., 2000a; Damm et al., 2000b] y como Ideogramic UML [Damm y Hansen, 2002; Hansen y Ratzer, 2002]. Se escoge el segundo nombre por ser el más reciente. Se trata de una aplicación de reconocimiento de diagramas de clases basada en *gestures* [Damm y Hansen, 2002; Hansen y Ratzer, 2002], es decir, se requiere que los elementos se dibujen con un solo trazo y de la forma indicada en el manual (Véase Anexo I). Obligar al usuario a dibujar de una forma concreta, limita su libertad, lo que conlleva a una pérdida de naturalidad a la hora de crear los diagramas.

Los trazos naturales se transforman en formales en cuanto terminan de dibujarse y se reconoce la *gesture* en cuestión. Con esto, se viola uno de los aspectos importantes de la

usabilidad de la aplicación (Véase Apartado 3.2) ya que según varios estudios [Hammond y Davis, 2002; Jarret y su, 2003; Alvarado y Davis, 2004] los usuarios prefieren trabajar con los trazos en tinta digital sin que se conviertan a trazos formales en mitad del diseño.

Presenta reconocimiento de texto siempre que este se introduzca, como ocurría con Tahuti, mediante un teclado virtual (Véase Fig. 3.10 extraída de [Damm et al., 2000b]).

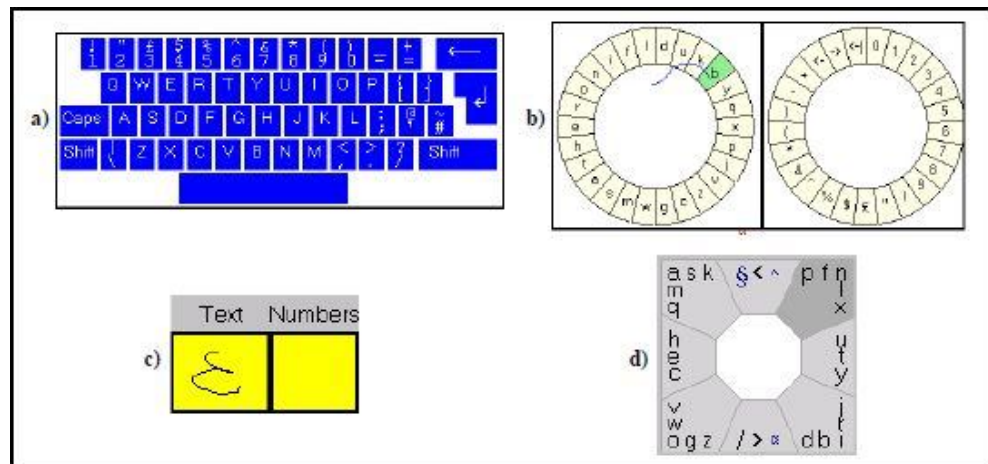


Fig. 3.10. Teclado virtual de Ideogramic UML

La aplicación posee dos modos de trabajo, el de dibujo, en el que no se reconocen los trazos, y el de UML, que interpreta los trazos dibujados como elementos de los diagramas de UML.

Se reconocen las relaciones de asociación, asociación unidireccional, herencia, agregación, composición y dependencia, siempre y cuando se dibujen de acuerdo al manual (Véase Anexo I).

- *SketchUML*

En esta aplicación [Qiu, 2007; Tenbergen et al., 2008; SketchUML] se combinan dos tipos de reconocimiento de trazo, el geométrico para reconocer las clases y el basado en *gestures* para reconocer el triángulo y el rombo que forman una relación de herencia y una de agregación respectivamente. Para que se reconozcan estas relaciones primero se debe dibujar una asociación entre las clases y posteriormente transformarla a la relación deseada dibujando la *gesture* correspondiente. Como se ha dicho anteriormente, siempre que se obligue al usuario a dibujar de una forma determinada, se está reduciendo su libertad.

La transformación de los trazos naturales a los formales se lleva a cabo en cuanto se termina de dibujar el elemento y se reconoce como parte de un diagrama de UML. Al igual que ocurría en Ideogramic, esto provoca que no se cumpla uno de los aspectos importantes de la usabilidad según los estudios llevados a cabo por, entre otros [Hammond y Davis, 2002; Alvarado, 2004], Microsoft [Jarret y su, 2003].

En su primera versión [Qiu, 2007] sólo se reconocían los nombres de las clases, mientras que en la segunda versión [Tenbergen et al., 2008, SketchUML] ya pueden incluirse atributos de las clases y métodos. Sin embargo, en ambos casos se debe crear en primer lugar la clase con el nombre (Véase Fig. 3.11 extraída de [Qiu, 2007]). En la segunda versión, los atributos y métodos se introducirán cuando la clase se haya reconocido y transformado como tal. El nombre de la clase puede escribirse antes o después del cuadrado que simboliza la clase, sin embargo, los atributos y métodos se introducen cuando la clase se ha transformado y se indica el hueco para ellos.

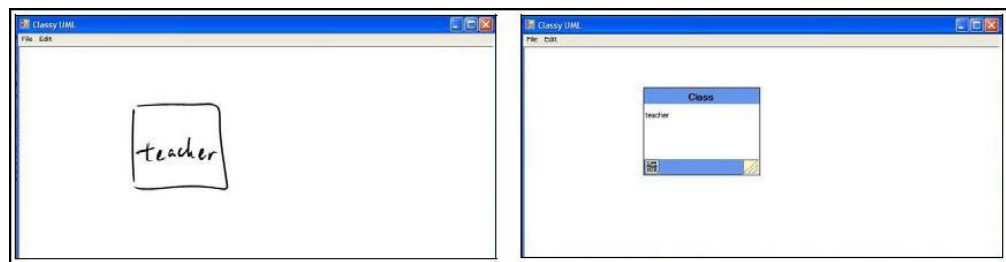


Fig. 3.11. Conversión de una clase a trazo formal en SketchUML.

La herramienta reconoce el texto manuscrito y las siguientes relaciones: asociación, herencia y agregación.

- *“An Interactive System for Recognizing Hand Drawn UML Diagrams”*

Edward Lank et al. [Lank et al., 2000], han construido un sistema de reconocimiento de UML que agrupa los trazos en primer lugar por el orden en que se dibujan, y en segundo por propiedades geométricas, principalmente por la longitud de los trazos. Como los trazos se agrupan por el orden en que se dibujan, se vuelve a tener el mismo problema que se tenía en Tahuti: No se permite añadir trazos a una clase o relación si no es en el instante de su creación.

En ocasiones, este algoritmo causa varios falsos positivos, por lo que hay que tener especial cuidado con algunos trazos, en concreto con los rombos de las agregaciones que a menudo se confunden con círculos.

Aunque el usuario puede dibujar de forma natural, no ocurre lo mismo con la edición. Existen dos modos de edición: segmentación y agrupación de trazos por tiempo. Ambos pueden producir errores, por lo que se permite al usuario corregir los resultados de la segmentación (Véanse Fig. 3.12. y Fig. 3.13 extraídas de [Lank et al., 2000]) para reducir la probabilidad de error en el segundo modo de edición.

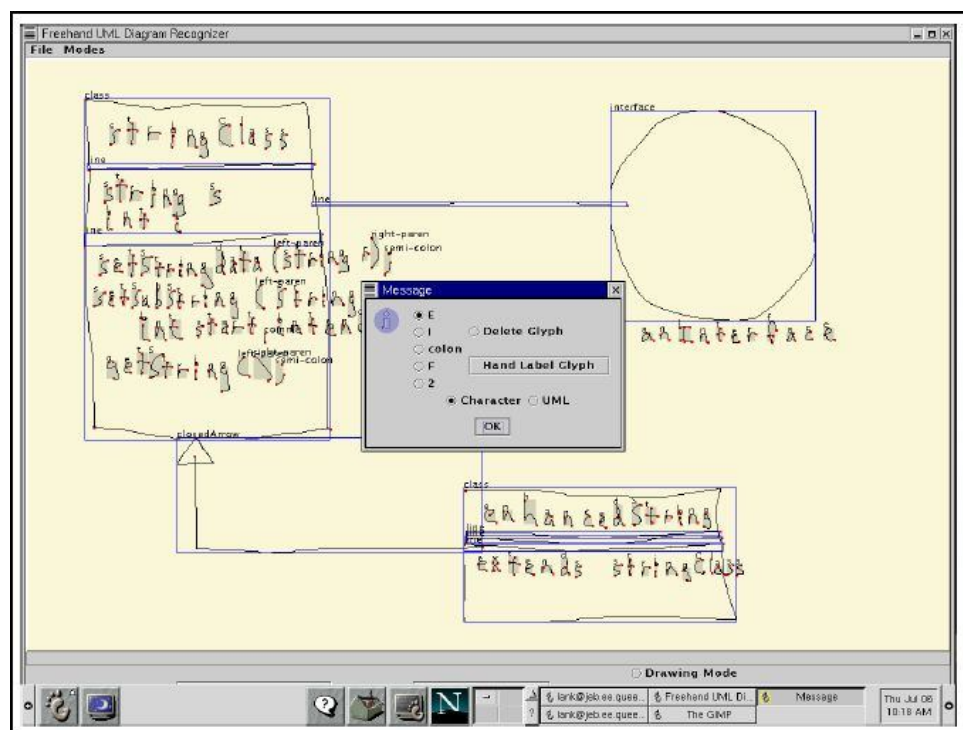


Fig. 3.12. Segmentación en el sistema de reconocimiento de diagramas de Lank.



Fig. 3.13. Corrección de errores tras la segmentación en el sistema de reconocimiento de diagramas de UML de Lank.

Reconoce entrada de texto manual y las relaciones de asociación, asociación unidireccional y herencia.

- *SUMLOW*

En esta herramienta [Plimmer y Grundy, 2005] el reconocimiento de trazos se lleva a cabo fijándose en las características geométricas de los mismos. Una vez más, como ocurría con Tahuti y con el sistema de Lank, se siguen patrones temporales para identificar las formas, con el problema que esto conlleva.

Se reconoce la entrada de texto manual siempre y cuando se escriba en las zonas indicadas para ello. A continuación se explica que significa la frase anterior:

En primer lugar, el usuario dibuja una clase, ésta se reconoce y aparece una línea de puntos indicando dónde debe escribirse para que el texto sea reconocido (Véase Fig. 3.14 extraída de [Plimmer y Grundy, 2005]). Al obligar al usuario a escribir en la zona y momento indicados, se está reduciendo su libertad a la hora de llevar a cabo el diagrama deseado.

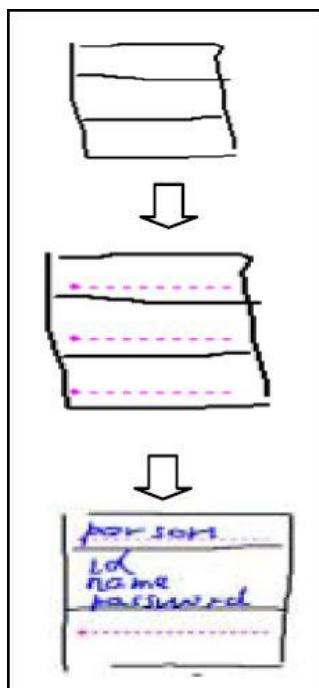


Fig. 3.14. Reconocimiento de texto en SUMLOW.

En esta aplicación, el usuario indica cuándo quiere que los trazos naturales se transformen a formales, pudiéndose perder información en la transformación [Plimmer y Grundy, 2005].

En este caso, únicamente se reconocen dos tipos de relaciones; asociaciones unidireccionales y herencias.

#### 3.6.4. Comparativa sobre las aplicaciones basadas en el reconocimiento de diagramas de clases

A continuación (Ver Tabla 3.1) se muestran a modo de resumen las características más relevantes de las distintas aplicaciones de reconocimiento de diagramas de clases en UML.

En primer lugar se indica el tipo de reconocimiento en que se basa cada una de las aplicaciones. Las basadas total o parcialmente en *gestures*, son las que delimitan la libertad de dibujo del usuario en mayor medida. Las basadas en geometría permiten una mayor flexibilidad en el diseño, pero al estar regidas por patrones temporales, cometen errores si alguno de los trazos de un componente no se dibuja de manera consecutiva al resto.

Observando el momento del reconocimiento puede verse que todas las aplicaciones lo realizan al terminar de dibujar el trazo. Algunas porque se hace necesario para poder reconocer las *gestures* y el resto porque, como se ha comentado anteriormente se basan en patrones temporales.

Otro aspecto a tener en cuenta es el momento en que transforman los trazos a mano alzada en trazos formales, ya que si se hace inmediatamente como ocurre con Ideogramic UML y SketchUML, se pierde uno de los factores importantes de la usabilidad de la aplicación, ya que esta conversión en mitad del diseño, distrae al usuario.

Cuanto mayor sea el número de relaciones reconocidas, mayor variedad de diagramas podrán diseñarse. Por ahora, como puede verse en la tabla, la aplicación que más relaciones reconoce es la basada íntegramente en *gestures*. Esto es así porque cuanto mayor número de limitaciones se pongan para que el usuario lleve a cabo el diseño, más fácil será el reconocimiento de los componentes, y por tanto, mayor será la variedad de elementos que la aplicación reconozca.

Para terminar, puede verse como la única aplicación que permite una libertad total de escritura es el *Interactive System for Recognizing Hand Drawns Diagrams*.

	<b>Tipo de Reconocimiento</b>	<b>Momento del Reconocimiento</b>	<b>Transformación a trazos formales</b>	<b>Relaciones</b>	<b>Texto</b>
<b>Tahuti</b>	Basado en Geometría Patrones Temporales	Al terminar el trazo	Al cambiar de vista	Asociación Unidireccional, Herencia, Agregación	Teclado virtual
<b>Ideographic</b>	Basado en <i>Gestures</i>	Al terminar el trazo	Al terminar el trazo	Asociación Unidireccional, Asociación, Herencia, Agregación, Dependencia, Composición	Teclado virtual
<b>SketchUML</b>	Geometría-> Clases <i>Gestures</i> -> Triángulo y Diamante	Al terminar el objeto	Al terminar el trazo	Asociación, Herencia, Agregación	Manual en la zona indicada
<b>Interactive System for Recognizing Hand Drawn UML Diagrams</b>	Basado en Geometría Patrones Temporales	Al terminar el trazo	Cuando indique el usuario	Asociación Unidireccional, Asociación, Herencia	Manual
<b>SUMLOW</b>	Basado en Geometría Patrones Temporales	Al terminar el trazo	Cuando indique el usuario	Asociación Unidireccional, Herencia	Manual en la zona indicada

Tabla 3.1. Comparativa de aplicaciones de reconocimiento de diagramas de UML



## 3.7. PROCESO

### 3.7.1. División de trazos en texto y figuras

El texto y las figuras son semánticamente diferentes y necesitan un tratamiento separado durante el proceso de reconocimiento [Patel et al., 2007]. Los caracteres dibujados a mano deben agruparse en palabras y frases, mientras que las figuras se identifican como componentes.

La mayoría de las herramientas para dibujar diagramas incluyen algún tipo de reconocimiento [Damm et al., 2000b], pero son muy pocas las que proveen de reconocimiento de figuras y texto. A continuación se detallan algunas de las principales técnicas:

- *Bishop*

Esta técnica de división de *strokes*, se basa en el uso discriminatorio de técnicas de aprendizaje automático (*machine learning*) [Bishop et al., 2004].

Se basa en el estudio de tres aproximaciones que aumentan su complejidad sucesivamente. Comienza considerando cada trazo por separado, del que se extraen un conjunto de características para entrenar al clasificador probabilístico. Este clasificador se basa en redes neuronales *feed-forward*. El modelo se aumenta incluyendo información temporal de forma que se pueda capturar la correlación entre las sucesivas catalogaciones. Por último, el modelo se extiende considerando la información extraída de los saltos entre *strokes* sucesivos.

Los resultados demuestran que el uso del contexto temporal mejora la clasificación en comparación con el estudio del *stroke* individual. Sin embargo, la clasificación teniendo en cuenta los saltos entre *strokes* no muestra mejoras tan claras.

- *Clase Divider de Microsoft*

Se trata de una clase de la librería *Microsoft.Ink* que permite analizar y dividir los *strokes* en texto y gráficos [MSDN].

- *Divider de Patel*

Patel [Patel et al., 2007] identifica 46 características que pueden emplearse para dar el soporte básico a la segmentación de la tinta digital. Estas 46 características se agrupan en siete categorías: tamaño, tiempo, intersecciones, curvatura, presión, valores de reconocimiento del sistema operativo y saltos entre trazos.

En el artículo se lleva a cabo un análisis de 1519 trazos dibujados por 26 personas distintas y se obtiene un árbol de clasificación de los trazos con dos tipos de hojas: figura o texto. Cada rama del árbol se decide según variables de clasificación (Véase Fig. 3.15 extraída de [Patel et al., 2007]).

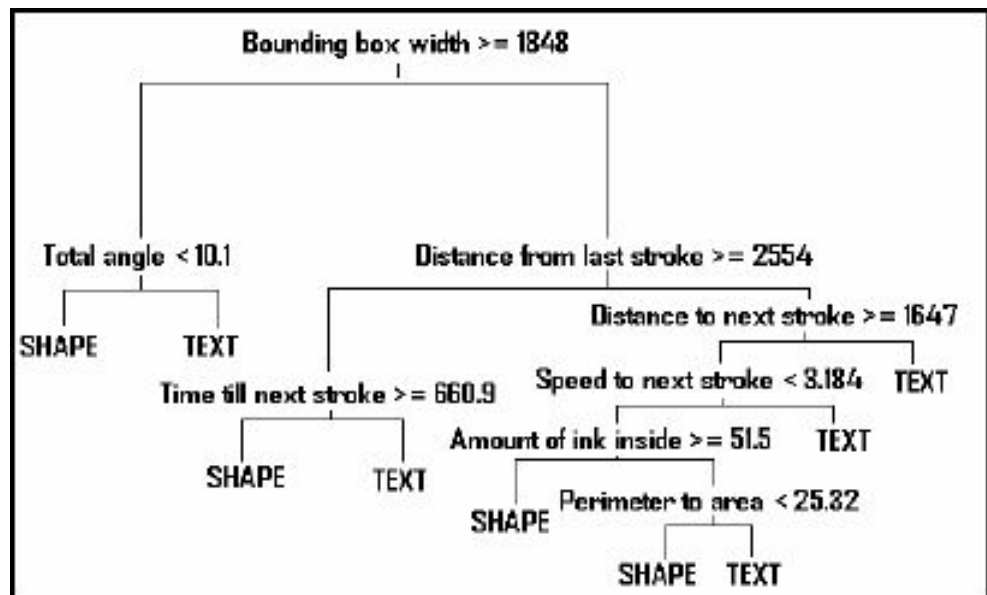


Fig. 3.15. Árbol de clasificación de los trazos según Patel

### 3.7.2. Localizar esquinas

Buscar las esquinas en los trazos, es una técnica que implica la división de un *stroke* en primitivas [Wolin y Hammond, 2008]. En los buscadores de esquinas en polilíneas (trazo formado por varias líneas) como es el caso del presente proyecto, se busca el mínimo conjunto de puntos de tal forma que si el trazo se rompe en esos puntos, las primitivas resultantes sean únicamente líneas. De entre los mecanismos y las técnicas que pueden emplearse, cabe destacar:

- *Propiedad PolylineCusps de la clase Stroke en Microsoft.Ink*

Se trata de una propiedad de la clase *Stroke* proporcionada por la librería *Microsoft.Ink*, que devuelve un array que contiene los índices de los vértices del *stroke* en cuestión [Jarret y Su, 2003; MSDN].

- *Sezgin*

En [Sezgin et al., 2001] se emplea la curvatura del trazo y la velocidad del lápiz para determinar las esquinas. Los puntos con gran curvatura son candidatos a esquinas, al igual que los puntos con baja velocidad. Una vez que se obtienen las esquinas candidatas, se elige la mejor opción siguiendo unas métricas. Este proceso se va repitiendo hasta obtener las esquinas reales que serán aquellas de las mejores de las candidatas que estén por debajo de un cierto umbral de error.

- *ShortStraw*

Wolin y Hammond [Wolin y Hammond, 2008] han encontrado una técnica de fácil comprensión e implementación para encontrar las esquinas en una poli línea.

Los pasos que se llevan a cabo son los siguientes:

- A partir del *stroke*, crear un array con los puntos de éste que estén separados una misma distancia (Véase Fig. 3.16 extraída de [Wolin y Hammond, 2008]).

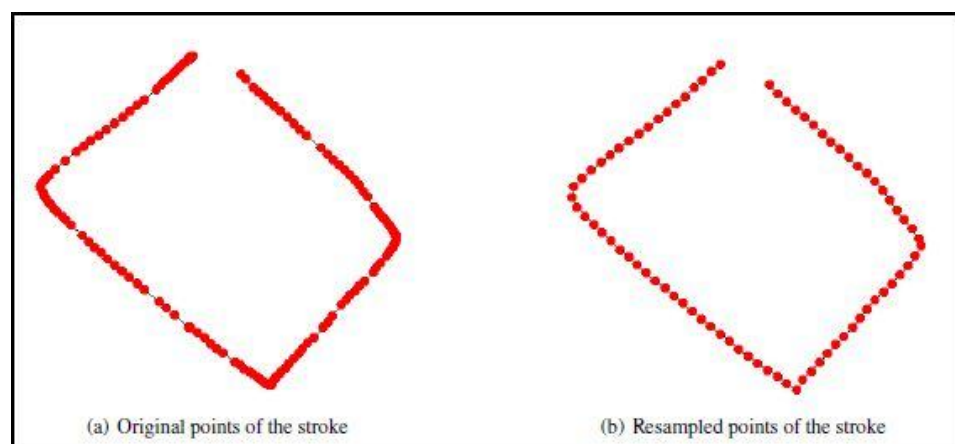


Fig. 3.16. *Stroke* original (a) y *Stroke* con los puntos espaciados una misma distancia (b).

- Se toma una constante (3 en este caso) y se calcula la distancia que hay desde un punto a tres puntos a su derecha y a tres a su izquierda.
- Las distancias menores son las que se corresponden con las esquinas (Véase Fig. 3.17 extraída de [Wolin y Hammond, 2008]).

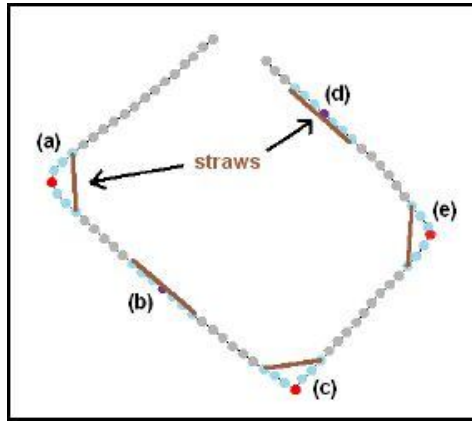


Fig. 3.17. Correspondencia entre las esquinas y las distancias menores.

## 4. METODOLOGÍA DE TRABAJO

En este capítulo se explica de forma detallada los pasos seguidos en el desarrollo del presente proyecto fin de carrera, empezando por un nivel más abstracto (diagramas de casos de uso y de secuencia) para finalizar en un nivel más concreto (implementación).

### 4.1. ESPECIFICACIÓN DE REQUISITOS

Se desea crear una aplicación de reconocimiento de trazos naturales, concretamente centrada en el reconocimiento de Diagramas de Clases. Esta aplicación presenta básicamente los siguientes dos requisitos:

- Debe crearse una herramienta que permita flexibilidad al usuario pero que a su vez no pierda robustez a la hora de llevar a cabo el reconocimiento de los trazos.
- Debe contar con un único modo para que el usuario tenga la sensación más parecida a estar trabajando sobre una hoja de papel.

### 4.2. FASE DE ANÁLISIS

#### 4.2.1. Diagrama de Casos de Uso

Un diagrama de casos de uso representa la vista funcional del sistema que se debe desarrollar [Polo, 2008]. Cada caso de uso representa un requisito del sistema. El diagrama muestra las relaciones entre los actores y los casos de uso.

Los actores representan aquellos elementos que se comunican con el sistema pero que no forman parte de él. En nuestro caso (Véase Fig. 4.1), tenemos dos actores; los usuarios del sistema que actúan con el sistema desencadenando la ejecución de operaciones y el sistema de archivos.

Puede verse que la mayoría de los casos de uso se incluyen en “Reconocer”. Eso es porque el comportamiento de todos ellos forma parte del reconocimiento.

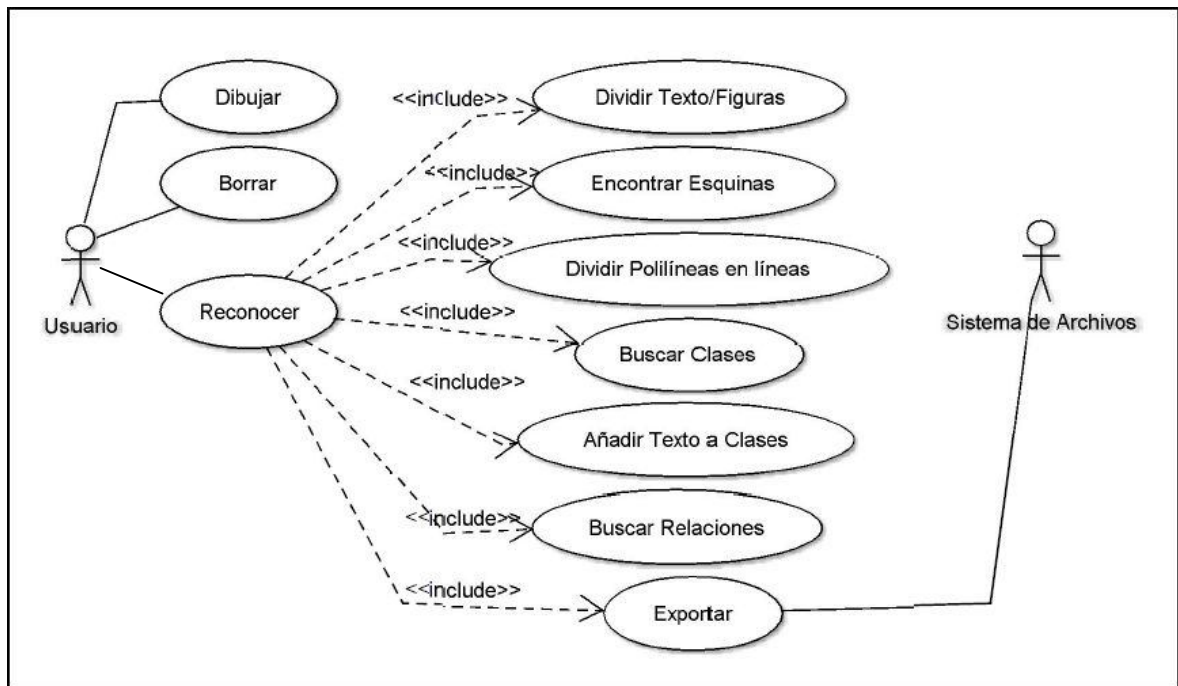


Fig. 4.1. Diagrama de casos de uso.

#### 4.2.2. Priorización de Casos de Uso

Para establecer el orden en que van a implementarse los casos de uso, debe hacerse una priorización de los mismos atendiendo a criterios como complejidad, necesidad de un caso de uso para implementar otro, etc.

Tanto para que el usuario dibuje como para que borre, no se necesita tener implementado ningún caso de uso con anterioridad, sin embargo, para que se lleve a cabo el reconocimiento de trazos, en primer lugar deben clasificarse los trazos en texto y figuras, después, se buscan las esquinas en los trazos que forman parte de las líneas. El siguiente paso es dividir las polilíneas en líneas cortando por las esquinas para poder buscar las clases. Una vez encontradas las clases, se busca el texto que forma parte de ellas. A continuación, se buscan las relaciones entre los trazos restantes y, por último, se crea un fichero con toda la información recopilada durante el proceso.

Con estas consideraciones, se plantea la siguiente ordenación:

- 1) Dibujar.
- 2) Borrar.

- 3) Reconocer
- 4) Dividir texto/figuras.
- 5) Encontrar esquinas.
- 6) Dividir polilíneas en líneas.
- 7) Buscar clases.
- 8) Añadir texto a clases.
- 9) Buscar relaciones.
- 10) Exportar

#### **4.2.3. Descripción de Casos de Uso**

Dado que se trata de un flujo secuencial, sin caminos alternativos, no se cree necesaria la descripción textual de los casos de uso ya que el requisito funcional que se está implementando no necesita mayor explicación.

### **4.3. FASE DE DISEÑO**

#### **4.3.1. Diagramas de secuencia**

Mediante los diagramas de secuencia se describen los flujos de eventos en términos de objetos y de mensajes pasados entre dichos objetos. Se suele construir un diagrama de secuencia por cada flujo de eventos. Como se ha comentado en el apartado anterior (Véase sección 4.2), se tienen tres flujos: dibujar, borrar y reconocer.

#### 4.3.1.1. Dibujar

Cuando el usuario desee dibujar, simplemente tendrá que hacerlo sobre el formulario (Véase Fig. 4.2), ya que al tratarse de una aplicación con un único modo, no hay que indicar al sistema lo que se desea hacer.

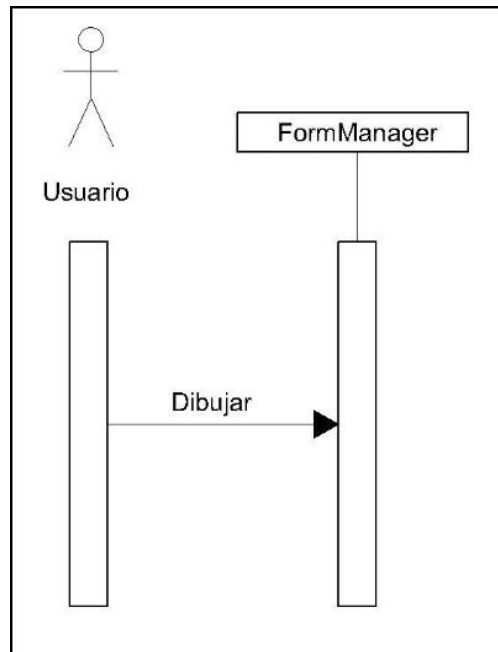


Fig. 4.2. Diagrama de secuencia de Dibujar.

#### 4.3.1.2. Borrar

Para borrar, tiene que dibujarse la *gesture* asignada a esta función (Véase Fig. 4.3). Cuando esto ocurra, el sistema reconoce la intención del usuario y hace una llamada para que se borren los trazos afectados por el tachado.



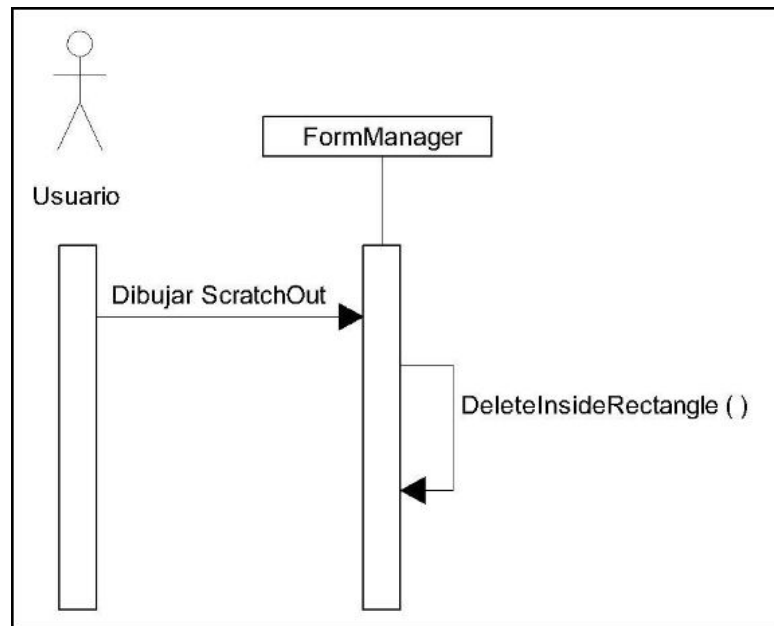


Fig. 4.3. Diagrama de secuencia de Borrar.

#### 4.3.1.3. Reconocer

Para que se lleve a cabo el reconocimiento del diagrama, el usuario debe pulsar la opción “Exportar” del menú archivo. Inmediatamente aparece un cuadro de diálogo para indicar dónde se desea guardar el archivo generado. A partir de aquí, se suceden secuencialmente todos los pasos indicados en el apartado 4.2, excepto el de borrar y dibujar (Véase Fig. 4.4).

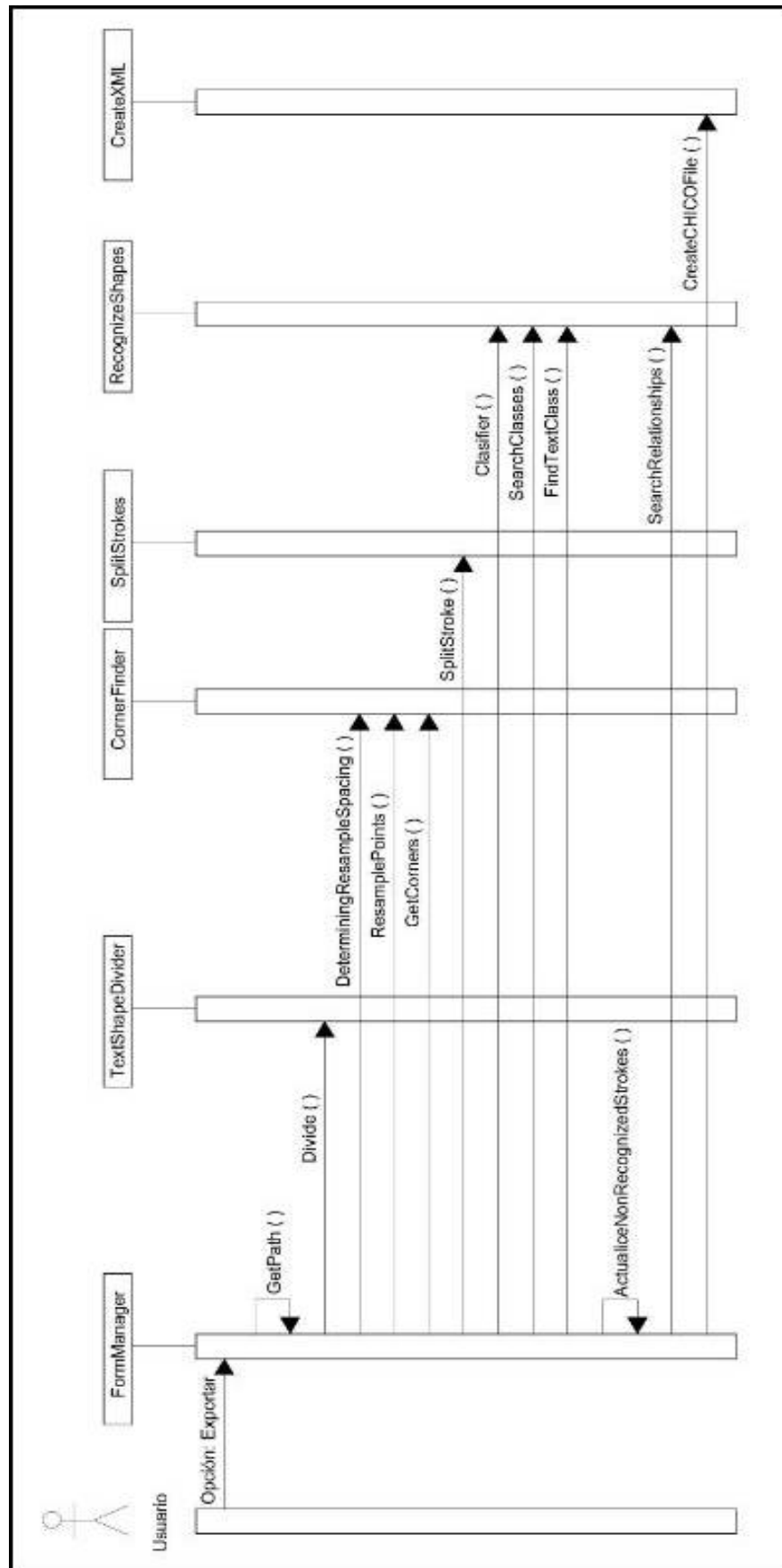


Fig. 4.4. Diagrama de secuencia de Exportar.

### 4.3.2. Arquitectura Multicapa

La aplicación se ha implementado siguiendo una arquitectura de multicapa, es decir, se encuentra estructurada en capas, cada una de ellas con una serie de clases que comparten un objetivo. Generalmente, y así es el caso, una aplicación multicapa consta de al menos tres capas [Polo, 2008], siendo las más habituales las siguientes:

- *Presentación*. En esta capa se encuentran las ventanas que se encargan de mostrar la información y con las que interactúa el usuario. En este proyecto y en esta capa, se encuentra únicamente la clase *FormManager.cs*, encargada de ofrecer al usuario el espacio donde dibujar los diagramas y los controles para que éste determine cuándo reconocer su creación.
- *Dominio*. También llamada de Negocio o de Procesamiento, se trata de la capa en la que reside toda la lógica que describe los mecanismos para resolver el problema. En este caso, las clases que forman parte de esta capa son las siguientes:
  - *Class.cs*: Contiene la especificación de las clases que se van a reconocer, es decir, los trazos que la forman, el nombre de la clase, los atributos y los métodos.
  - *ConvexHulls.cs*: Contiene los métodos necesarios para llevar a cabo la envoltura convexa, es decir, para calcular en un espacio vectorial real el mínimo conjunto de puntos que dejan en su interior (envuelven) un conjunto de puntos determinado. Esta clase ha sido cedida por Beryl Plimmer de la Universidad de Auckland y sus métodos no han sufrido modificación, únicamente se han eliminado los que no eran necesarios para el desarrollo del presente proyecto.
  - *CornerFinder.cs*: Contiene los métodos necesarios para buscar las esquinas en los trazos que se le pasen.
  - *ExtraMath.cs*: Contiene funciones matemáticas implementadas, como la comprobación de si dos trazos son paralelos o la pendiente de un trazo.

- *Facade.cs*: Para facilitar una posible reutilización o modificación del código, en esta clase se incluyen todas las constantes y variables que se utilizan a lo largo de todo el proceso.
- *RecognizeShapes.cs*: Contiene los métodos necesarios para reconocer las formas (orientación de las líneas, clases, relaciones...).
- *Relation.cs*: Contiene la especificación de las relaciones que se van a reconocer, es decir, las clases que forman parte de la relación y su tipo.
- *SplitStrokes.cs*: Contiene los métodos necesarios para romper las polilíneas en líneas por las esquinas.
- *TextShapeDivider.cs*: Contiene el código necesario para separar los trazos que forman parte de las figuras de los que forman parte del texto. Este código también ha sido cedido por Beryl Plimmer, pero en esta ocasión sí se han tenido que llevar a cabo modificaciones en el código que se explicarán más adelante.
- *Persistencia*. En esta capa se incluyen las clases mediante las que se accede a los medios de almacenamiento. En este caso, se tiene una única clase, *CreateXML.cs*, que es la encargada de generar el archivo de texto con estructura de XML con la información obtenida en el reconocimiento del diagrama.

## 4.4. DETALLES DE IMPLEMENTACIÓN

A continuación se explican en detalle los distintos pasos llevados a cabo en la implementación. Esta información se divide en subapartados indicando en cada uno de ellos las alternativas estudiadas y el por qué de su rechazo o aceptación. En el caso de que la alternativa haya sido la elegida, se especifican los detalles de implementación.

### 4.4.1. Visión General

A grandes rasgos, el proceso de reconocimiento se lleva a cabo de la siguiente manera:

- 1) Lo primero es separar los trazos de texto de los que forman partes de figuras, ya que al tratarse de trazos distinta naturaleza, necesitan un tratamiento diferente. Este proceso se detalla en la sección 4.4.2.
- 2) A continuación, y trabajando con los trazos de figuras, se buscan las esquinas de cada uno de ellos. En el subapartado 4.4.3 se profundiza en los detalles de implementación.
- 3) Una vez que se tienen las esquinas, se dividen todas las polilíneas en líneas, cortando por las esquinas. Se puede ver en la sección 4.4.4 el proceso completo.
- 4) Lo siguiente es clasificar las líneas en horizontales, verticales y otras. Esta técnica se detalla en el punto 4.4.5.
- 5) A continuación, se buscan las clases que existan en el diagrama. Una clase la forman cuatro líneas horizontales y dos verticales, por lo que se buscan las cuatro líneas horizontales que tienen las mismas verticales más cercanas. En la sección 4.4.6 se profundiza en los detalles de implementación.
- 6) Una vez hecho esto, se buscan entre los *strokes* de texto los que pertenezcan a cada una de las clases, es decir, el texto que forma el nombre de la clase, los atributos y los métodos. Este proceso se detalla en el punto 4.4.7.

- 7) Al tener completas todas las clases, texto incluido, los *strokes* restantes son los que formarán parte de las relaciones, por lo que se pasa de trabajar únicamente con dichos trazos.
- 8) Con los *strokes* que aún no han sido reconocidos, se repiten los pasos 2 y 3. Esto es necesario porque en ocasiones algunas partes de las relaciones se reconocen como texto cuando no lo son y por tanto no se dividen en líneas hasta este momento.
- 9) Con todos los *strokes* no reconocidos convertidos en líneas, se buscan las relaciones. En la sección 4.4.8 se detalla la técnica empleada para el reconocimiento de las distintas relaciones.
- 10) Se crea el archivo *.chico* con la información recopilada estructurada en XML, de modo que pueda ser importada y tratada por una aplicación capaz de permitirnos continuar con el ciclo de desarrollo. Los detalles de implementación pueden verse en la sección 4.4.9.

#### 4.4.2. Dividir los trazos en texto y figuras

En este caso, las alternativas coinciden con las técnicas explicadas en la sección 3.7.1.

En primer lugar se barajó la posibilidad de utilizar la clase *Divider* de Microsoft. Esta alternativa tiene a su favor la facilidad de implementación, pero se decidió no trabajar con ella al observar el estudio llevado a cabo por Patel et al. [Patel et al., 2007]. En dicho estudio se analizan los resultados de tres tipos distintos de divisores de trazos aplicados a dos juegos distintos de datos (Véase Fig. 4.5 extraída de [Patel et al., 2007]). En ambos casos se observa como los peores resultados se obtienen con la clase *Divider* de Microsoft (*Microsoft* en el gráfico), y los mejores con el *Divider* de Patel (*New* en el gráfico). El tercer divisor se corresponde con el predecesor del de Patel, por lo que no se ha estudiado.

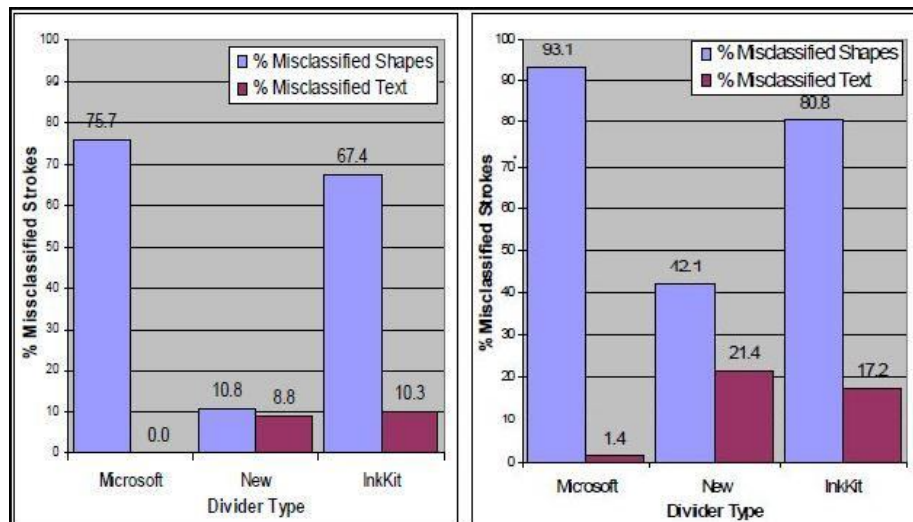


Fig. 4.5. Estudio de técnicas de *Divider* según Patel

Una vez descartada la clase *Divisor* de Microsoft, la elección de la alternativa a implementar se realiza entre el divisor de Bishop y el de Patel. La complejidad del primero y la disponibilidad del código del segundo (Véase ANEXO II) gracias a la colaboración de Beryl Plimmer (Universidad de Auckland) hace que se escoja éste último.

A pesar de disponer del código proporcionado por Plimmer, bajo su consentimiento y aprobación, éste no se emplea íntegramente en el presente proyecto fin de carrera, sino que se modifica y adapta a las necesidades del mismo para evitar sobrecargas de métodos innecesarias y para hacer un código más legible. Los únicos métodos que permanecen intactos son los que pertenecen a la clase *ConvexHulls.cs*.

Tras su adaptación a nuestra aplicación, el método central de este apartado se basa en el árbol de clasificación del artículo (Véase Fig. 3.15) y queda como sigue:

```
// Dependiendo de la posición del stroke se ejecuta un código u
// otro
public static int Divide(Stroke s)
{
    // Un único stroke
    if (OnlyOne(s))
        return DivideOnlyOne(s);
    else
    {
        // Primer stroke
        if (!ExistsPrev(s))
```

```

        return DivideFirst(s);
    else
    {
        // Último stroke
        if (!ExistsNext(s))
            return DivideLast(s);
        else
            // Strokes intermedios
            return DivideMedium(s);
    }
}

// Divider cuando se tiene un único stroke
public static int DivideOnlyOne(Stroke s)
{
    if (GetWidth(s) >= Facade.B_BOX_WIDTH)
        if (GetTotalAngle(s) < Facade.TOTAL_ANGLE)
            return Facade.SHAPE;
        else
            return Facade.TEXT;

    else
        if (GetInkInside(s) >= Facade.AMT_INK)
            return Facade.SHAPE;
        else
            if (GetPerimeterToArea(s) < Facade.PERIMETER)
                return Facade.SHAPE;
            else
                return Facade.TEXT;
}

// Divider cuando se tiene el primer stroke
public static int DivideFirst(Stroke s)
{
    if (GetWidth(s) >= Facade.B_BOX_WIDTH)
        if (GetTotalAngle(s) < Facade.TOTAL_ANGLE)
            return Facade.SHAPE;
        else
            return Facade.TEXT;
}

```



```

else
{
    double timeToNext = GetTimeToNext(s, GetNextStroke(s));
    double distanceToNext = GetDistanceBetweenStrokes(
        GetNextStroke(s), s);
    double speedToNext = distanceToNext / timeToNext;

    if (timeToNext > Facade.TIME_NEXT)
        return Facade.SHAPE;

    if (distanceToNext >= Facade.DIST_NEXT)
        if (speedToNext < Facade.SPEED_NEXT)
            if (GetInkInside(s) >= Facade.AMT_INK)
                return Facade.SHAPE;
            else
                if (GetPerimeterToArea(s) < Facade.PERIMETER)
                    return Facade.SHAPE;
                else
                    return Facade.TEXT;
            else
                return Facade.TEXT;
        else
            return Facade.TEXT;
    }
}

// Divider cuando se tiene un último stroke
public static int DivideLast(Stroke s)
{
    if (GetWidth(s) >= Facade.B_BOX_WIDTH)
        if (GetTotalAngle(s) < Facade.TOTAL_ANGLE)
            return Facade.SHAPE;
        else
            return Facade.TEXT;

    else
        if (GetDistanceBetweenStrokes(s, GetPrevStroke(s)) >=
            Facade.DIST_LAST)
            return Facade.SHAPE;
        else
            if (GetInkInside(s) >= Facade.AMT_INK)

```

```

        return Facade.SHAPE;
    else
        if (GetPerimeterToArea(s) < Facade.PERIMETER)
            return Facade.SHAPE;
        else
            return Facade.TEXT;
    }

    // Dividir cuando se tienen strokes intermedios
    public static int DivideMedium(Stroke s)
    {
        if (GetWidth(s) >= Facade.B_BOX_WIDTH)
            if (GetTotalAngle(s) < Facade.TOTAL_ANGLE)
                return Facade.SHAPE;
            else
                return Facade.TEXT;

        else
            if (GetDistanceBetweenStrokes(s, GetPrevStroke(s)) >=
                Facade.DIST_LAST)
                if (GetTimeToNext(s, GetNextStroke(s)) >=
                    Facade.TIME_NEXT)
                    return Facade.SHAPE;
                else
                    return Facade.TEXT;
            else
            {
                double distanceToNext = GetDistanceBetweenStrokes(
                    GetNextStroke(s), s);
                double timeToNext = GetTimeToNext(
                    s, GetNextStroke(s));
                double speedToNext = distanceToNext / timeToNext;

                if (distanceToNext >= Facade.DIST_NEXT)
                    if (speedToNext < Facade.SPEED_NEXT)
                        if (GetInkInside(s) >= Facade.AMT_INK)
                            return Facade.SHAPE;
                        else
                            if (GetPerimeterToArea(s) < Facade.PERIMETER)
                                return Facade.SHAPE;
                            else

```

```

        return Facade.TEXT;
    else
        return Facade.TEXT;
    else
        return Facade.TEXT;
    }
}

```

Además de cambiar el código por razones estéticas y de comprensión, se modifica el primer punto de decisión del árbol de clasificación (Véase Fig. 3.15) por la siguiente razón: Al dibujar las clases trazo a trazo, es decir, sin polilíneas, se observó que los trazos verticales eran reconocidos por el *Divider* como texto. El primer nodo del árbol mira el ancho del *BoundingBox* (mínimo rectángulo que engloba todo el trazo) del trazo. Las líneas horizontales y verticales presentan distinto ancho, por lo que se iban por ramas distintas. Para solucionar esto, se mira el ancho y alto del trazo y, con el mayor de ellos se hace la comparación en el primer nodo del árbol. El código resultante es el siguiente:

```

// Se mira el alto y ancho del stroke y se devuelve el mayor
public static int GetWidth(Stroke s)
{
    int width;
    int height;

    width = s.GetBoundingBox().Width;
    height = s.GetBoundingBox().Height;

    if (width > height)
        return width;
    else
        return height;
}

```

#### 4.4.3. Encontrar esquinas

Al igual que ocurre en el apartado anterior, las alternativas de esta parte del proceso ya se han comentado anteriormente (Véase sección 3.7.2).

En este caso no se disponía de estudios que revelasen la eficacia de cada una de las alternativas, por lo que se ha intentado encontrar un equilibrio entre la facilidad de programación y los resultados obtenidos. Debido a la complejidad de implementación se desestima el algoritmo propuesto por Sezgin et al. [Sezgin et al., 2001]. Los dos métodos restantes se implementan para ver qué resultados se obtienen.

Tras la implementación, como puede comprobarse observando las imágenes, los resultados obtenidos por el método de Wolin y Hammond (Véase Fig. 4.7) [Wolin y Hammond, 2008] son mejores que los obtenidos empleando la librería (Véase Fig. 4.6) [Jarret y Su, 2003; MSDN], a pesar de que las esquinas estén mejor dibujadas al aplicar esta técnica.

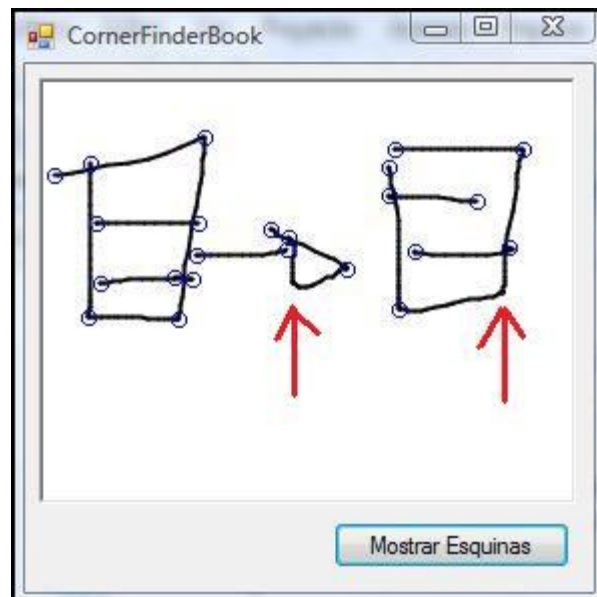


Fig. 4.6. Esquinas encontradas por la alternativa de la librería.

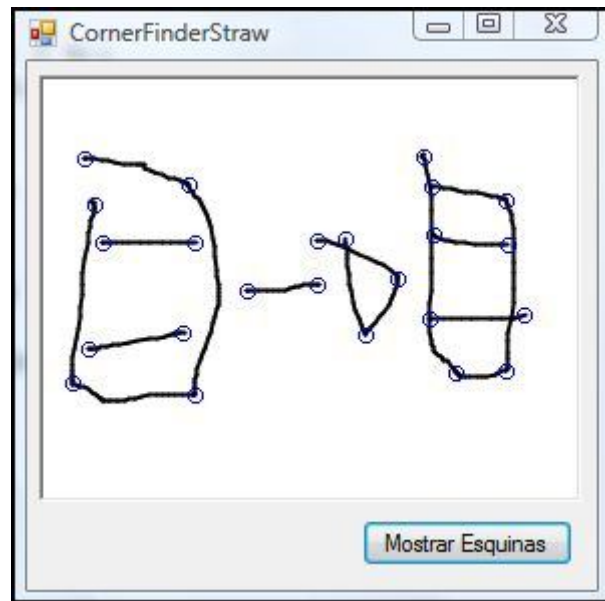


Fig. 4.7. Esquinas encontradas por la alternativa de Wolin y Hammond.

En esta parte se ha encontrado un grave problema al existir dos versiones del artículo de Wolin y Hammond [Wolin y Hammond, 2008], ya que en primer lugar el código se implementó siguiendo las indicaciones y el pseudocódigo de la versión con errores (Véase Anexo III), con lo que obviamente no se conseguía el resultado deseado. Finalmente se obtuvo y estudió el artículo que corregía las erratas (Véase Anexo III) y se pudo continuar con el trabajo. Aun con la versión corregida, se han llevado a cabo algunos cambios y ajustes en la implementación del pseudocódigo del algoritmo para el correcto funcionamiento del mismo con el fin de alcanzar el objetivo buscado. Concretamente, se ha bajado la constante de umbral de 0.95 a 0.85 a la hora de comprobar si un segmento forma una línea. Esto se ha hecho necesario porque en las relaciones no se tomaban como líneas, las que sí lo eran, señalándose las correspondientes esquinas que producían un reconocimiento incorrecto de las relaciones. También se ha eliminado un trozo de código que añadía esquinas que no se habían encontrado en una primera pasada. Esto no se hacía necesario puesto que las esquinas se reconocían perfectamente y se corría el riesgo de que se encontraran esquinas que no lo fueran.

Desde la clase *FormManager.cs*, se hacen las siguientes llamadas a los métodos:

```
dist_between_points =  
CornerFinder.DeterminingResampleSpacing(strokes_shape[i]);
```

```

resampled =
(List<Point>)CornerFinder.ResamplePoints(strokes_shape[i],
dist_between_points);

List<int> corners = (List<int>)CornerFinder.GetCorners(resampled);

```

El código implementado de las funciones se encuentra en la clase *CornerFinder.cs* y queda como sigue:

```

// Determina la distancia que debe existir entre los puntos
// muestreados
public static double DeterminingResampleSpacing(Stroke s)
{
    double dist_between_points;
    Point top_left = new Point();
    Point bottom_right = new Point(); ;

    // Se obtiene el BoundingBox del Stroke
    Rectangle r = s.GetBoundingBox();

    // Se asignan a top_left la esquina superior izquierda del
    // boundingbox,y a bottom_right la esquina inferior derecha.
    top_left.X = r.Left;
    top_left.Y = r.Top;

    bottom_right.X = r.Right;
    bottom_right.Y = r.Bottom;

    // Se calcula la longitud de la diagonal formada por las dos
    // esquinas calculadas
    double diagonal = ExtraMath.Distance(bottom_right, top_left);

    // Se calcula la distancia entre los puntos muestreados
    dist_between_points = diagonal / 40.0;

    return dist_between_points;
}

// Deja los puntos del stroke que esten espaciados en
// dist_between_points

```

```

public static List<Point> ResamplePoints(Stroke s, double
dist_between_points)
{
    // Se crea una lista con los puntos del stroke
    List<Point> points = new List<Point>(s.GetPoints());

    // Se añade el primer punto del stroke a la coleccion de
    // resampled
    List<Point> resampled = new List<Point>();
    resampled.Add(points[0]);

    // Se crea un nuevo punto que se utilizara mas adelante
    Point pt = new Point();

    // Se inicializa el marcador de distancia a 0
    double distance_holder = 0;

    // Se inicializa la distancia entre dos puntos
    double distance_points = 0;

    // Se recorren todos los puntos del stroke haciendo lo siguiente
    for (int i = 1; i < points.Count; i++)
    {
        // Se calcula la distancia Euclidea entre el punto actual y
        // el anterior
        distance_points = ExtraMath.Distance(points[i - 1],
points[i]);

        if ((distance_holder + distance_points) >=
dist_between_points)
        {

            // Se crea un punto nuevo (pt) localizado aproximadamente
            // a una distancia dist_between_points del último punto
            // muestreado
            pt.X = (int)(points[i - 1].X +
(((dist_between_points - distance_holder) /
distance_points) * (points[i].X - points[i - 1].X)));
            pt.Y = (int)(points[i - 1].Y +
(((dist_between_points - distance_holder) /
distance_points) * (points[i].Y - points[i - 1].Y)));

```

```

        // Se añade el punto pt a la lista de "resampled"
        resampled.Add(pt);

        // Se añade el punto pt al stroke, en la posición
        // anterior al punto en el que estamos
        points.Insert(i, pt);

        distance_holder = 0;
    }
    else
        distance_holder += distance_points;
    }
    return resampled;
}

// Busca los puntos de resampled que se corresponden con esquinas
public static List<int> GetCorners(List<Point> resampled)
{
    // Se crea la lista donde irán las esquinas. Almacenará un
    // conjunto de índices que referencian puntos. Por ejemplo,
    // corner(i)=j indica que el punto(j) es la i-esima esquina
    // encontrada
    List<int> corners = new List<int>();

    corners.Add(0);

    // Se crea una lista con las distancias entre dos puntos
    // separados W puntos del punto actual
    List<double> straws = new List<double>();

    for (int i = 0; i < Facade.W; i++)
    {
        straws.Add(ExtraMath.Distance(resampled[i + Facade.W],
            resampled[i]));
    }

    for (int i = Facade.W; i < (resampled.Count - Facade.W); i++)
    {
        straws.Add(ExtraMath.Distance(resampled[i - Facade.W],
            resampled[i + Facade.W]));
    }
}

```



```

}

for (int i = (resampled.Count - Facade.W); i < resampled.Count;
i++)
{
    straws.Add(ExtraMath.Distance(resampled[i - Facade.W],
    resampled[i]));
}

// Se calcula un umbral, threshold. Para ello, ordenamos la
// lista de straws y calculamos su mediana. Para calcular la
// mediana se necesita ordenar la lista. Se trabajara con una
// copia de la lista para no modificar la original
List<double> copy_straws = new List<double>(straws);
copy_straws.Sort();
int middle = copy_straws.Count / 2;
double median = (copy_straws.Count % 2 != 0) ?
(double)copy_straws[middle] :
((double)copy_straws[middle] + (double)copy_straws[middle - 1])
/ 2;
double threshold = median * 0.95;

// Ahora se recorre la lista de straws. Si la distancia es menor
// que el umbral, se considera esquina
double local_min;
int local_min_index;

for (int i = Facade.W; i < (resampled.Count - Facade.W); i++)
{
    if (straws[i] < threshold)
    {
        local_min = Facade.INFINITE;
        local_min_index = i;

        while (i < straws.Count && straws[i] < threshold)
        {
            if (straws[i] < local_min)
            {
                local_min = straws[i];
                local_min_index = i;
            }
        }
    }
}

```

```

        i++;
    }
    corners.Add(local_min_index);
}
}

// Se añade el ultimo indice a corners
corners.Add(resampled.Count - 1);
corners = PostProcessCorners(resampled, corners, straws);
return corners;
}

// Se vuelven a procesar las esquinas encontradas para eliminar
// falsos positivos
public static List<int> PostProcessCorners(List<Point> resampled,
List<int> corners, List<double> straws)
{
    int c1, c2 = 0;

    for (int i = 1; i < (corners.Count - 1); i++)
    {
        c1 = corners[i - 1];
        c2 = corners[i + 1];

        if (IsLine(resampled, c1, c2))
        {
            corners.RemoveAt(i);
            i -= 1;
        }
    }
    return corners;
}

// Determina si la parte del stroke indicada es una línea o no
public static bool IsLine(List<Point> resampled, int a, int b)
{
    double distance = (double)ExtraMath.Distance( resampled[a],
resampled[b]);
    double path_distance = (double)PathDistance(resampled, a, b);

    if ((distance / path_distance) > Facade.THRESHOLD)

```

```

        return true;
    else
        return false;
    }

    // Determina la distancia del tramo (no la distancia más corta)
    public static double PathDistance(List<Point> resampled, int a, int
b)
    {
        double distance = 0;

        for (int i = a; i < b; i++)
        {
            distance += (double)ExtraMath.Distance(resampled[i],
resampled[i + 1]);
        }

        return distance;
    }

```

#### 4.4.4. Dividir polilíneas en líneas

Para realizar esta labor no se ha necesitado estudiar distintas alternativas ya que la librería *Microsoft.Ink* proporciona el método *Split* de la clase *Stroke* [MSDN] con el que se divide el trazo por el sitio que se indique. Lo que se hace es pasar como índice para dividir el *stroke*, la posición de las esquinas, consiguiendo así dividir las polilíneas en líneas.

El código implementado es el siguiente:

```

// Para option = 0 => se trabaja con los strokes_shape
// Para option = 1 => con strokes_without_recognize
public static void SplitStroke(Stroke s, List<Point> resampled,
List<int> corners, int option)
{
    // Se crea el findex que se va a emplear en el split
    float findex = 0;

    // Se comprueba la posicion en que el punto del stroke coincide
    // con la segunda esquina (sólo hace falta mirar la segunda
    // esquina porque la primera es el

```

```

// primer punto del stroke, y a partir de la tercera
// pertenecerían al stroke nuevo, el obtenido al romper el
// trazo). Si la segunda esquina es el último punto, ya tenemos
// la línea, pero si no lo es, tendremos que ver que posición
// ocupa en el stroke para poder romperlo por ahí.

// Punto de la esquina (en resampled)
Point pt_corner = resampled[corners[1]];

// Se comprueba que no sea el último punto
if (!pt_corner.Equals(resampled[resampled.Count - 1]))
{

    findex = SplitStrokes.GetFIndex(pt_corner, s);

    // El stroke se rompe por ese punto
    Stroke new_stroke = s.Split(findex);

    // El nuevo stroke se añade a la lista de strokes
    FormManager.ink_overlay.Ink.Strokes.Add(new_stroke);

    if (option == 0)
        FormManager.strokes_shape.Add(new_stroke);
    else
        Facade.strokes_without_recognize.Add(new_stroke);
}
}

```

#### 4.4.5. Clasificar líneas

En este caso tampoco existen alternativas ya que la idea desde un principio fue dividir las líneas en horizontales, verticales y otras. Para ello hay que fijarse en la pendiente de las líneas. El código implementado es el siguiente:

```

public static void Clasifier(Stroke s)
{
    double slope = ExtraMath.Slope(s, 0);

    // Si pendiente = INFINITE => vertical_strokes

```

```

// Si pendiente<SLOPE_MAX => horizontal_strokes
// Si pendiente>1/SLOPE_MAX=> vertical_strokes
// En otro caso => other_strokes

if(slope == Facade.INFINITE)
{
    Facade.vertical_strokes.Add(s);
}
else
{
    if (slope < Facade.SLOPE_MAX)
    {
        Facade.horizontal_strokes.Add(s);
    }
    else
    {
        if (slope > (1 / Facade.SLOPE_MAX))
        {
            Facade.vertical_strokes.Add(s);
        }
        else
        {
            Facade.other_strokes.Add(s);
        }
    }
}
}

```

En este apartado, se hace necesario calcular las pendientes en valor absoluto, ya que no se debe dejar que la inclinación de la recta afecte a la decisión de clasificarla como vertical, horizontal u oblicua.

#### 4.4.6. Buscar clases

Una vez más no se tienen alternativas. En este caso se analiza la forma y la posición de las clases. Una clase siempre va a tener la misma orientación, es decir, no nos vamos a encontrar una clase girada (se supone que el usuario dibuja de forma coherente). Como requisito se especifica que para que una clase se reconozca como tal, debe dibujarse con

los distintos apartados para el nombre, los atributos y los métodos (Véase Fig. 4.8) y que, al menos, debe contener el nombre de la clase donde corresponde.

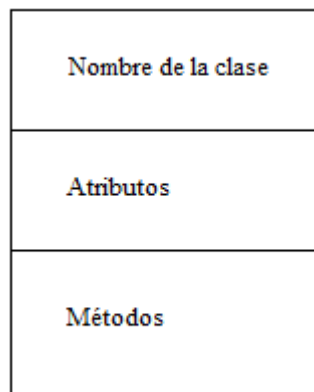


Fig. 4.8. Partes de una clase

Con lo anterior, nos encontramos con una clase cuando coincidan cuatro líneas horizontales con las mismas verticales a ambos lados.

El código implementado es el siguiente:

```
// Busca las clases que hay
public static void SearchClasses(List<Stroke> vertical_strokes,
List<Stroke> horizontal_strokes)
{
    // s1 es el stroke que esta más cerca el primer punto del stroke
    // horizontal.
    // s2 es el stroke más cercano al último punto del stroke
    // horizontal
    Stroke s1 = null;
    Stroke s2 = null;

    // Se crea una matriz donde se almacena en cada posicion, el
    // stroke horizontal con los dos strokes verticales más cercanos
    // a él ( 3 strokes en total, el horizontal y los dos verticales
    // más cercanos).
    Stroke[,] nearby = new Stroke[3, horizontal_strokes.Count];

    for (int i = 0; i < horizontal_strokes.Count; i++)
    {
        Stroke h_s = horizontal_strokes[i];
```

```

NearbyStrokes(h_s, ref s1, ref s2, vertical_strokes, 0);

// Se almacena en el array de vecinos el stroke horizontal
// con los dos verticales más cercanos
nearby[0, i] = h_s;
nearby[1, i] = s1;
nearby[2, i] = s2;
}

// Una vez que se tienen los strokes horizontales con los
// verticales más cercanos se ve cuales de los horizontales
// tienen los mismos verticales cercanos. Cuando haya 4
// horizontales con los mismos 2 verticales, se forma una clase
// con los 6.

// Se crea una lista con los strokes horizontales con mismas
// verticales más cerca
List<Stroke> same_vertical = new List<Stroke>();

int index = 0;

for (int i = 0; i < horizontal_strokes.Count - 1; i++)
{

    // Se inicializan los strokes que van a crear la clase
    Stroke side1 = null;
    Stroke side2 = null;
    Stroke side3 = null;
    Stroke side4 = null;
    Stroke side5 = null;
    Stroke side6 = null;

    same_vertical.Add(nearby[0, i]);

    // El for no llega al último elemento, porque si ya no está
    // en la lista, es que no coincide con otro stroke, es decir,
    // no pertenece a la clase.

    for (int j = i + 1; j < horizontal_strokes.Count; j++)
    {

```

```

if (!same_vertical.Contains(nearby[0, j]))
{
    if ((nearby[1, i].Equals(nearby[1, j]) || nearby[1,
i].Equals(nearby[2, j]))
    && (nearby[2, i].Equals(nearby[1, j]) || nearby[2,
i].Equals(nearby[2, j])))
    {
        same_vertical.Add(nearby[0, j]);
    }
}
}

// Cuando se tengan cuatro horizontales en same_vertical,
// tenemos una clase. Para saber qué stroke está en qué
// posición de la clase hacemos lo siguiente:
// - si s1.X < s2.X => s1=side1 y s2=side2
//   else => s1=side2 y s2= side1
// - en same_vertical, se mira el que tenga mínima Y =>
// side3, el de máxima => side6.
// De los dos restantes se mira el de mayor Y => side5, y el
// restante => side 4.

if (same_vertical.Count == 4)
{
    if (nearby[1, i].GetPoint(0).X < nearby[2,
i].GetPoint(0).X)
    {
        side1 = nearby[1, i];
        side2 = nearby[2, i];
    }
    else
    {
        side1 = nearby[2, i];
        side2 = nearby[1, i];
    }

    Stroke stroke_y_max = null;
    Stroke stroke_y_min = null;

    for (int j = 0; j < same_vertical.Count; j++)
    {

```



```

    for (int k = j + 1; k < same_vertical.Count-1; k++)
    {
        if (same_vertical[j].GetPoint(0).Y >
            same_vertical[k].GetPoint(0).Y)
        {
            stroke_y_max = same_vertical[j];
        }
    }
}

side6 = stroke_y_max;
same_vertical.Remove(stroke_y_max);

for (int j = 0; j < same_vertical.Count; j++)
{
    for (int k = j + 1; k < same_vertical.Count-1; k++)
    {
        if (same_vertical[j].GetPoint(0).Y <
            same_vertical[k].GetPoint(0).Y)
        {
            stroke_y_min = same_vertical[j];
        }
    }
}

side3 = stroke_y_min;
same_vertical.Remove(stroke_y_min);

if (same_vertical[0].GetPoint(0).Y <
    same_vertical[1].GetPoint(0).Y)
{
    side4 = same_vertical[0];
    side5 = same_vertical[1];
}
else
{
    side4 = same_vertical[1];
    side5 = same_vertical[0];
}

```

```

        if (side1 != null && side2 != null && side3 != null &&
            side4 != null && side5 != null && side6 != null)
        {
            // Se añade la nueva clase con los strokes indicados a
            // la lista de clases
            Facade.classes.Insert(index, new Class(side1, side2,
            side3, side4, side5, side6));
            index++;
            Facade.strokes_recognized.Add(side1);
            Facade.strokes_recognized.Add(side2);
            Facade.strokes_recognized.Add(side3);
            Facade.strokes_recognized.Add(side4);
            Facade.strokes_recognized.Add(side5);
            Facade.strokes_recognized.Add(side6);
        }
    }

    for (int j = same_vertical.Count - 1; j >= 0; j--)
    {
        same_vertical.RemoveAt(j);
    }
}
}

```

#### 4.4.7. Asignar texto a clase

Aquí tampoco se tienen alternativas. Para identificar qué texto corresponde al nombre de la clase, a los atributos o a los métodos, se aprovecha la posición en que se guarda cada uno de los *strokes* de la clase. Así se tiene que si el texto está dentro del rectángulo formado por 1, 2,3 y 4, es el nombre de la clase, si lo está entre 1, 2,4 y 5 son los atributos, y si lo está entre 1,2,5 y 6, son los métodos (Véase Fig. 4.9).

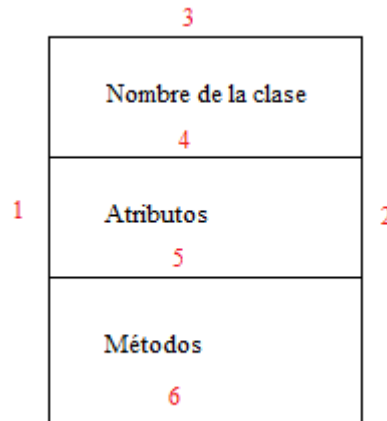


Fig. 4.9. Numeración de los *strokes* de una clase

La implementación es la siguiente:

```
// Busca el texto de las clases. Si la opcion es:
// - 0=>Nombre de la clase
// - 1=>Atributos
// - 2=>Metodos
public static String FindTextClass(Class c, Strokes strokes_text,
int option)
{
    Strokes reco_name = FormManager.ink_overlay.Ink.CreateStrokes();
    String name = null;

    // Se obtienen los puntos necesarios de la clase
    int s1_X = c.GetS1().GetPoint(0).X;
    int s2_X = c.GetS2().GetPoint(0).X;
    int s3_Y = 0;
    int s4_Y = 0;

    switch (option)
    {
        case 0:
            s3_Y = c.GetS3().GetPoint(0).Y;
            s4_Y = c.GetS4().GetPoint(0).Y;
            break;
        case 1:
            s3_Y = c.GetS4().GetPoint(0).Y;
```

```

        s4_Y = c.GetS5().GetPoint(0).Y;
        break;
    case 2:
        s3_Y = c.GetS5().GetPoint(0).Y;
        s4_Y = c.GetS6().GetPoint(0).Y;
        break;
    default:
        break;
}

// Se recorren los strokes reconocidos como texto, y los
// que esten entre s3 y s4 de la clase, forman el nombre, y es
// lo que se reconoce
for (int i = 0; i < strokes_text.Count; i++)
{
    Stroke s = strokes_text[i];

    // Se obtienen los puntos de la esquina superior izquierda
    // del stroke
    int s_X = s.GetBoundingBox().X;
    int s_Y = s.GetBoundingBox().Y;

    // Si el stroke esta entre los lados 1,2,3 y 4 de la clase,
    // pertenece al nombre de la clase
    if ((s1_X < s_X) && (s2_X > s_X) && (s3_Y < s_Y) && (s4_Y >
s_Y))
    {
        reco_name.Add(s);
        Facade.strokes_recognized.Add(s);
    }
}

name = reco_name.ToString();

return name;
}

```

#### 4.4.8. Buscar relaciones

En este caso existen varias alternativas cuyos diferentes pros y contras se analizaron antes de decantarse por una u otra. Se procede a explicarlas a continuación:

- *Alternativa 1*

En primer lugar se miró la posibilidad de emplear *gestures* en el reconocimiento de las relaciones. Esto tiene a su favor la facilidad de implementación, pero en contra que la libertad del usuario a la hora de dibujar se ve mermada.

- *Alternativa 2*

Como segunda opción, se pensó en separar los trazos por longitud en largos y cortos y para reconocer las relaciones mirar lo siguiente:

- Una asociación ( ----- ) está compuesta por un trazo largo.
- Una asociación unidireccional ( -----> ) la forman un trazo largo y dos cortos.
- Una herencia ( -----|> ) la forman un trazo largo y tres cortos.
- Una agregación ( -----<> ) la forman un trazo largo y cuatro cortos.

Pero esta alternativa no funcionaría en todos los casos, ya que por ejemplo, si sólo existiesen asociaciones en un diagrama de clases, la separación en trazos largos y cortos sería incorrecta y como consecuencia de ello se haría un reconocimiento erróneo del diagrama.

- *Alternativa 3*

En tercer lugar se pensó en introducir el factor tiempo, pero una vez más se limitaba la libertad del usuario al tener que dibujar las relaciones en orden. Esto se puede ver con el siguiente ejemplo: si se dibujase una asociación y al finalizar el diagrama se observase que debería haberse dibujado una herencia, no se puede pintar la cabeza de la herencia sobre la asociación, ya que al no haberse dibujado en orden, no se reconocería correctamente.

- *Alternativa 4*

Como cuarta alternativa, se barajó la idea de obligar al usuario a pintar la relación de forma que existiese intersección entre la cabeza de la relación y el cuerpo, pero también se desestimó por limitar la libertad del usuario.

- *Alternativa 5*

La quinta opción fue establecer que la cabeza de la relación debía dibujarse con un solo trazo para poder reconocerlas posteriormente mirando las esquinas de cada una de ellas de la siguiente forma:

- Una asociación ( ----- ) no tiene cabeza.
- La cabeza de una relación de asociación unidireccional ( -----> ) presenta tres esquinas (punto inicial, punto final y la esquina propiamente dicha).
- La cabeza de una relación de herencia ( -----|> ) presenta cuatro esquinas.
- La cabeza de una relación de agregación ( -----<> ) tiene cinco esquinas.

Esta alternativa también tuvo que eliminarse por las siguientes razones:

- Limita la libertad el usuario.
- Por tamaño, las cabezas de las relaciones se tratan como texto al pasar por el divisor de texto y figuras, pero puede ocurrir que en algún caso este tamaño sea mayor y se trate como figuras. Si esto ocurriese, los trazos se romperían en líneas, por lo que se perderían las esquinas y por tanto no se podrían reconocer las relaciones como es debido.

- *Alternativa Adoptada*

Como alternativa final, se optó por lo siguiente: Los trazos se ordenan de mayor a menor tamaño. Se coge el primer *stroke* (el más largo) y se busca el *stroke* que tenga más cercano (se le llama “n”). Si la distancia de “n” al *stroke* largo es menor que un cierto porcentaje (en este caso se ha probado con el 50%) de la longitud total de éste, “n” forma parte de una relación con el *stroke* largo, si no es así, se tiene una asociación. A continuación se miran los *strokes* más cercanos a “n” (nn1 y nn2). Si nn1 y nn2 son

paralelos, se trata una agregación. Si no son paralelos, y alguno de los dos coincide con el *stroke* largo, se tiene una asociación unidireccional, y si no coinciden, se tiene una herencia.

El código implementado es el siguiente:

```
// Determina la relacion en funcion de las puntas
// ----- (Ningun stroke en la punta)
// -----> (Se mira el stroke de la punta más cercano al
// resto de la flecha, y se miran los strokes más cercanos a dicho
// stroke. Si uno es la línea larga de la flecha se trata de una
// asociación direccionada)
// -----<> (Los strokes más cercanos al stroke más cercano a
// la línea larga,son paralelos)
// -----|> (Los strokes más cercanos al stroke más cercano
// a la línea larga, no son paralelos)
public static void SearchRelationships()
{
    // Se ordenan los strokes de menor a mayor longitud
    Facade.strokes_without_recognize.Sort(
CompareStrokesByLength);

    // Se le da la vuelta para tenerlos de mayor a menor longitud
    Facade.strokes_without_recognize.Reverse();

    int i = Facade.strokes_without_recognize.Count;
    do
    {
        // Se crean los strokes n1 y n2, correspondientes a los
        // strokes más cercanos al stroke s (el primero de la lista
        // de los strokes_without_recognize)
        Stroke n1 = null;
        Stroke n2 = null;

        // Será el stroke más cercano a "s"
        Stroke n = null;

        // Se crean los strokes nn1 y nn2, correspondientes a los
        // strokes más cercanos al stroke n
        Stroke nn1 = null;
```

```

Stroke nn2 = null;

// Se mira el primer stroke. Si es el único, se trata de una
// asociación (----)
Stroke s = Facade.strokes_without_recognize[0];

if (i == 1)
{
    CreateAssociation(s);
    i -= 1;
}
else
{
    // Se miran los strokes mas cercanos a los extremos del
    // primer stroke (n1 para el primer extremo y n2 para el
    // segundo)
    NearbyStrokes(s, ref n1, ref n2,
        Facade.strokes_without_recognize, 1);

    // Se comprueba que estén a menos de un porcentaje de la
    // longitud // del stroke inicial. Si es así, pertenecen a la
    // relación. En caso contrario se trata de una asociación
    // (----)( se crea la asociación y se elimina el stroke de
    // la lista de no reconocidos)
    bool belongs_n1 = BelongsToRelation(s, n1, 0);
    bool belongs_n2 = BelongsToRelation(s, n2,
        s.GetPoints().Length - 1);

    if (belongs_n1 == false && belongs_n2 == false)
    {
        CreateAssociation(s);
        i -= 1;
    }
    else
    {
        // Si no tenemos una asociación, se miran los strokes
        // más cercanos a los dos extremos del stroke que
        // estaba más cerca del inicial.
        if (belongs_n1 == true)
            n = n1;
        if (belongs_n2 == true)

```



```

n = n2;

// Se miran los strokes mas cercanos a los extremos de
// "n" (nn1 para el primer extremo y nn2 para el
// segundo)
NearbyStrokes(n, ref nn1, ref nn2,
Facade.strokes_without_recognize, 0);

// Si esos dos strokes son paralelos: ----<> (Se crea
// relación y se eliminan strokes de no reconocidos)
if (ExtraMath.AreParallels(nn1, nn2))
{
    CreateAgregation(s, n, nn1, nn2);
    i -= 5;
}
else
{
    // Si no son paralelos y si nn1 o nn2 coinciden con
    // s: ---->
    if ( FormManager.CompareStrokes(s, nn1) ||
        FormManager.CompareStrokes(s, nn2))
    {
        CreateAssociationDirectional(s, n, nn1, nn2);
        i -= 3;
    }
    else
    {
        // Si no son paralelos y ni nn1 ni nn2 coinciden
        // con s: ----|> (Crear relación y eliminar
        // strokes)
        CreateGeneralization(s, n, nn1, nn2);
        i -= 4;
    }
}
}
}
} while (i > 0) ;
}

```

En este apartado, y a diferencia del proceso desarrollado en el apartado 4.4.5, las pendientes de las rectas se calculan sin emplear el valor absoluto y multiplicándolas por

100. Esto se hace así porque los valores de las pendientes se redondeaban y no se obtenían los resultados deseados.

#### 4.4.9. Crear archivo .chico

Para que la información recopilada en el reconocimiento del diagrama pueda ser importada e interpretada por herramientas CASE que permitan continuar con el resto del ciclo de desarrollo de un producto software, se optó por elegir una herramienta desarrollada en el seno del grupo de investigación CHICO integrada a modo de *plug-in* para la plataforma Eclipse (<http://www.eclipse.org>). Para ello, el editor a mano alzada debe permitir crear un fichero de estructurado en XML donde se indicarán las clases de las que consta y las relaciones existentes entre ellas.

El código que genera dicho archivo es el siguiente:

```
public static void CreateCHICOFile()
{
    // El archivo se va a guardar en la ruta indicada en el cuadro
    // de diálogo de guardar
    using (StreamWriter stream_writer = new
    StreamWriter(Facade.path))
    {
        // Se escribe la cabecera
        stream_writer.Write("<?xml version=\"1.0\" encoding=\"utf-
        8\"?>\n");

        stream_writer.Write("<com.chico.chico:Model
        xmi:version=\"2.0\" xmlns:xmi=\"http://www.omg.org/XMI\"
        xmlns:xsi=\"http://www.w3.org/2001/XMLSchema-instance\"
        xmlns:Domain_Package=\"Domain_Package\"
        xmlns:com.chico.chico=\"com.chico.chico\">\n");

        // Se recorre la lista de clases para ir escribiendo el
        // contenido de cada una en el archivo
        for (int i = 0; i < Facade.classes.Count; i++)
        {
            Class c = Facade.classes[i];
```

```

        String name = c.GetName();
        stream_writer.Write("\t<itsClasses name =" + "\"" + name
+ "\"" + ">\n");

        if (c.GetAttributes() != null)
        {

            for (int j = 0; j < c.GetAttributes().Length; j++)
            {
                String attribute =
                c.GetAttributes().GetValue(j).ToString();
                stream_writer.Write("\t\t<itsAttributes name =" +
                "\"" + attribute + "\"" + ">\n");
            }
        }

        if (c.GetMethods() != null)
        {
            for (int j = 0; j < c.GetMethods().Length; j++)
            {
                String met = c.GetMethods().GetValue(j).ToString();
                stream_writer.Write("\t\t<itsMethods name =" + "\"" +
                met + "\"" + ">\n");
            }
        }
        stream_writer.Write("\t</itsClasses>\n");
    }
}

```

// Se recorre la lista de relaciones para ir escribiendo el contenido de cada

// una en el archivo

```

for (int i = 0; i < Facade.relations.Count; i++)
{
    Relation r = Facade.relations[i];
    String source = "//@itsClasses." +
RecognizeShapes.SearchIndexClass(r.GetStartClass());
    String target = "//@itsClasses." +
RecognizeShapes.SearchIndexClass(r.GetEndClass());
    String relation_type = "";

    switch (r.GetType())

```

```

        {
            case 0:
                relation_type =
                    "Domain_Package:Bidirectional_Association";
                break;
            case 1:
                relation_type = "Domain_Package:Association";
                break;
            case 2:
                relation_type = "Domain_Package:Generalization";
                break;
            case 3:
                relation_type = "Domain_Package:Aggregation";
                break;
        }
        stream_writer.Write("\t<itsRelationships xsi:type=" +
            "\"" + relation_type + "\"");
        stream_writer.Write(" source=");
        stream_writer.Write "\"" + source + "\"");
        stream_writer.Write(" target=");
        stream_writer.Write "\"" + target + "\"");
        stream_writer.Write("/>\n");
    }
    stream_writer.Write("</com.chico.chico:Model>");
}
}

```

La ruta para guardar el archivo se obtiene del cuadro de diálogo que aparece para señalar la ruta deseada. El código se muestra a continuación:

```

private void GetPath()
{
    String file_filter = "CHICO files (*.chico)|*.chico| " +
        "All files (*.*)|*.*";

    SaveFileDialog save_dialog = new SaveFileDialog();

    // Inicializa y muestra el cuadro de diálogo
    save_dialog.Filter = file_filter;

    if (save_dialog.ShowDialog() == DialogResult.OK)

```

```

    {
        Facade.path = save_dialog.FileName;
    }
}

```

#### 4.4.10. Borrado

El borrado es la única acción que se implementa mediante *gestures*, y aun así se trata de una forma natural para borrar como es el tachado (Véase Fig. 4.10). El único requisito para que se reconozca este trazo como la acción para borrar es que tenga al menos tres movimientos horizontales de ida y vuelta.



Fig. 4.10. *Gesture* para borrar

El código implementado para activar el reconocimiento de la *gesture* es el siguiente:

```

// Se crea un objeto InkOverlay y se pone en el modo ink-and-
// gesture. Se coge este modo y no gesture-only porque este ultimo
// necesita que transcurra un tiempo para reconocer la gesture.
ink_overlay = new InkOverlay(pnlInput.Handle);
ink_overlay.CollectionMode = CollectionMode.InkAndGesture;

// Se indica al InkOverlay las gestures que queremos reconocer
ink_overlay.SetGestureStatus(ApplicationGesture.Scratchout, true);

// Se conecta el manejador de eventos con las gesture y los stroke
ink_overlay.Gesture += new
InkCollectorGestureEventHandler(inkOverlay_Gesture);

ink_overlay.Stroke += new
InkCollectorStrokeEventHandler(inkOverlay_Stroke);

```

Cuando la *gesture* es reconocida, se ejecuta el código que borrará los trazos deseados, incluido el del propio tachado. La implementación es la siguiente:

```

void inkOverlay_Gesture(object sender, InkCollectorGestureEventArgs
e)

```

```

{
    // Si se esta seguro del gesture a un nivel de confianza fuerte
    // (Strong) o intermedio (intermediate) borramos. En caso
    // contrario, no se hace nada.
    if ((e.Gestures[0].Confidence == RecognitionConfidence.Strong)
        ||
        (e.Gestures[0].Confidence ==
        RecognitionConfidence.Intermediate))
    {
        switch (e.Gestures[0].Id)
        {
            case ApplicationGesture.Scratchout:
                // Borramos
                // Se obtiene el BoundingBox del Scratchout
                Rectangle r = e.Strokes.GetBoundingBox();

                // Se recorre la coleccion de strokes
                foreach (Stroke s in ink_overlay.Ink.Strokes)
                {
                    // Se mira si cada stroke esta incluido en el
                    // BoundingBox de Scratchout y si es asi se borra
                    DeleteInsideRectangle(s, r, e);
                }
                break;
            default:
                // No se hace nada
                break;
        }
    }

    // Se asegura de que el stroke se ha borrado
    e.Cancel = false;
    pnlInput.Invalidate();
    throw new NotImplementedException();
}

private void DeleteInsideRectangle(Stroke s, Rectangle r,
InkCollectorGestureEventArgs e)
{
    // Se buscan los strokes que tengan al menos un 60% en el
    // boundingbox

```

```

    Strokes borrarStrokes =
    ink_overlay.Ink.HitTest(r, 60);

    // Si se encuentra algo, se borran los strokes afectados
    if (borrarStrokes.Count > 0)
    {
        // El stroke del scratchout no se maneja mas
        e.Cancel = true;

        // Se borran los strokes afectados
        ink_overlay.Ink.DeleteStrokes(borrarStrokes);

        // Se borra el stroke del scratchout
        ink_overlay.Ink.DeleteStrokes(e.Strokes);

        Refresh();
    }
}

```

## 5. RESULTADOS

En este capítulo se explica el manual de usuario así como las restricciones que éste deberá cumplir al dibujar. Estas restricciones se derivan de las distintas decisiones tomadas a lo largo del desarrollo del presente proyecto.

Aquí también se observan los criterios de usabilidad del punto 3.2 que se han cumplido.

El capítulo termina con una comparativa entre las aplicaciones de reconocimiento de trazos de Diagramas de Clases existentes y la desarrollada.

### 5.1. MANUAL DE USUARIO

Dado que se trata de una aplicación que permite interacción natural, no se requiere ningún conocimiento especial para comenzar a trabajar con ella, simplemente debe abrirse y empezar a dibujar como si de una hoja de papel se tratase.

El usuario puede dibujar de forma natural, salvo pequeñas excepciones explicadas en el subapartado 5.1.1. Existen múltiples formas de combinar los trazos ya que se permite dibujar tanto con trazos separados (Véase Fig. 5.1) como con trazos seguidos o polilíneas (Véanse Fig. 5.2, Fig. 5.3 y Fig. 5.4). A continuación, se explican las figuras a las que se hace referencia.

En la siguiente imagen (Véase Fig. 5.1), se ve un ejemplo de cómo quedaría una clase dibujada con trazos separados. Se resaltan las esquinas para que se observe que no existe continuidad en los trazos, sino que, efectivamente, cada pared de la clase es dibujada con una única línea.



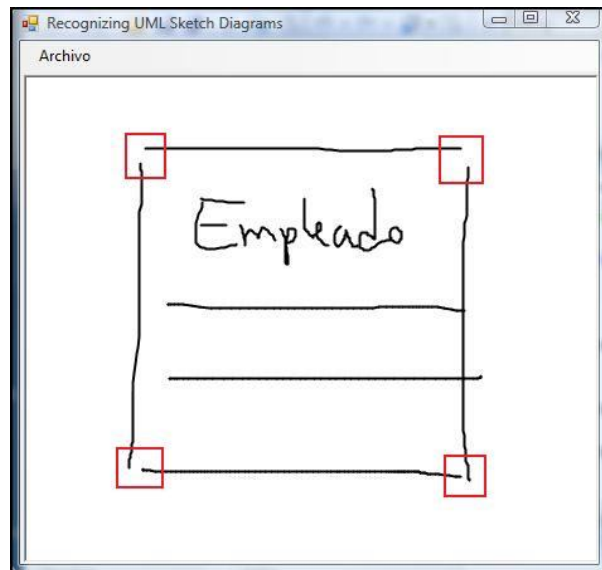


Fig. 5.1. Dibujar con trazos separados.

A continuación (Véase Fig. 5.2), se muestra el primer ejemplo de clase dibujada con polilíneas. Puede verse como el marco de la clase se ha creado con dos trazos. El primero de ellos abarca el lateral izquierdo de la clase, la parte superior y el lateral derecho, mientras que el segundo trazo, completa la clase en la zona inferior.

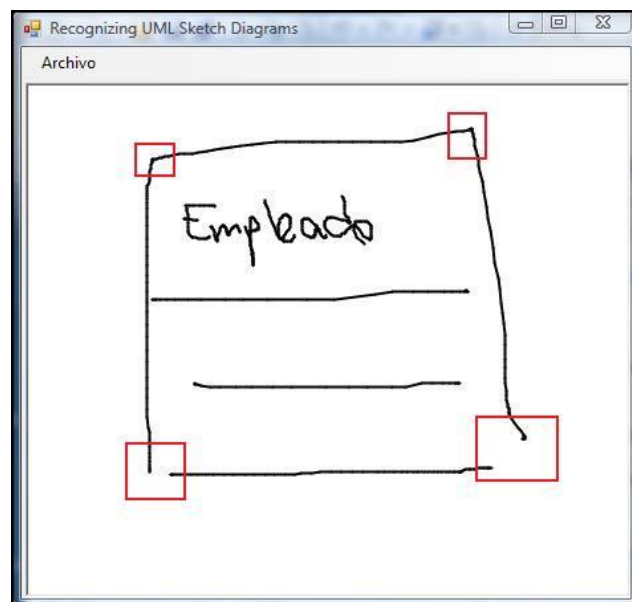


Fig. 5.2. Dibujar con polilíneas: Primer ejemplo.

En el caso siguiente (Véase Fig. 5.3), el borde de la clase se dibuja con dos trazos. Uno que engloba la parte izquierda y la inferior, y el otro que cierra la clase por la derecha y por la parte superior.

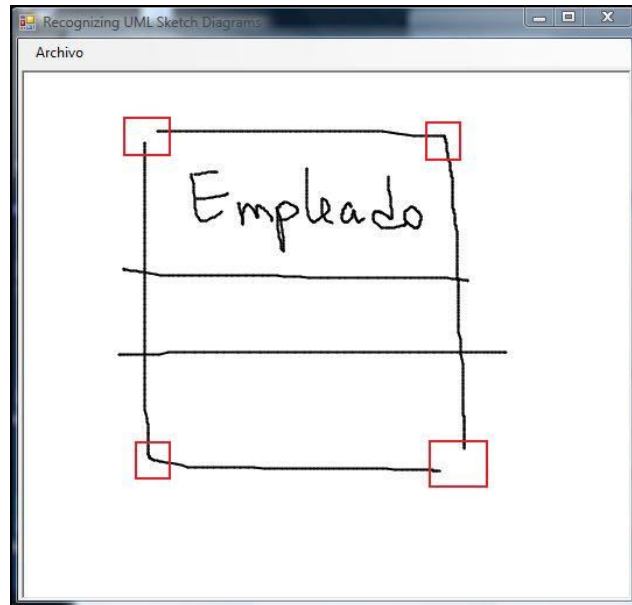


Fig. 5.3. Dibujar con polilíneas: Segundo ejemplo.

Por último (Véase Fig. 5.4), se dibuja el contorno de la clase con un único trazo que comienza y termina en la esquina superior izquierda.

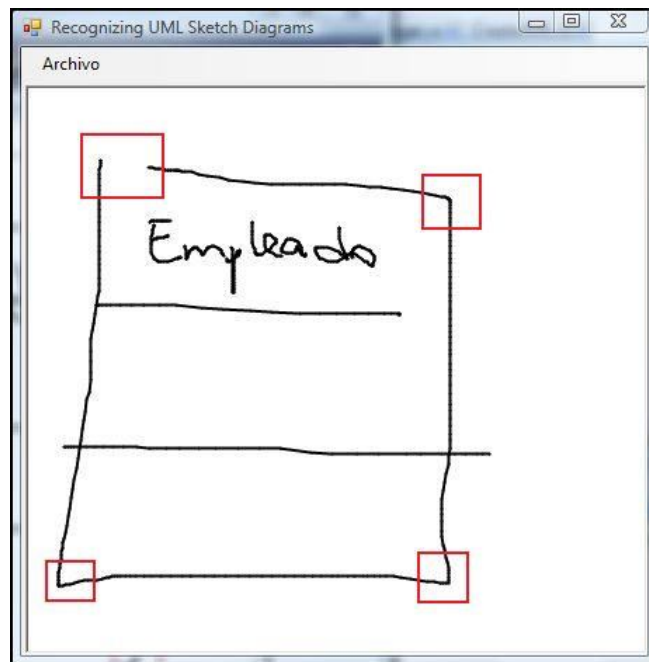


Fig. 5.4. Dibujar con polilíneas: Tercer ejemplo.

En cualquier momento del diseño, si el usuario dibuja la *gesture* asociada al borrado, inmediatamente se borrarán los trazos afectados por ésta (Véase Fig. 5.5). Es importante que el trazo de tachado cubra el que se desea borrar en prácticamente su totalidad, ya que sólo se borrarán los trazos que se vean cubiertos en, por lo menos, un 60% de su totalidad. Esto es así para evitar eliminar trazos que no se quieran suprimir.

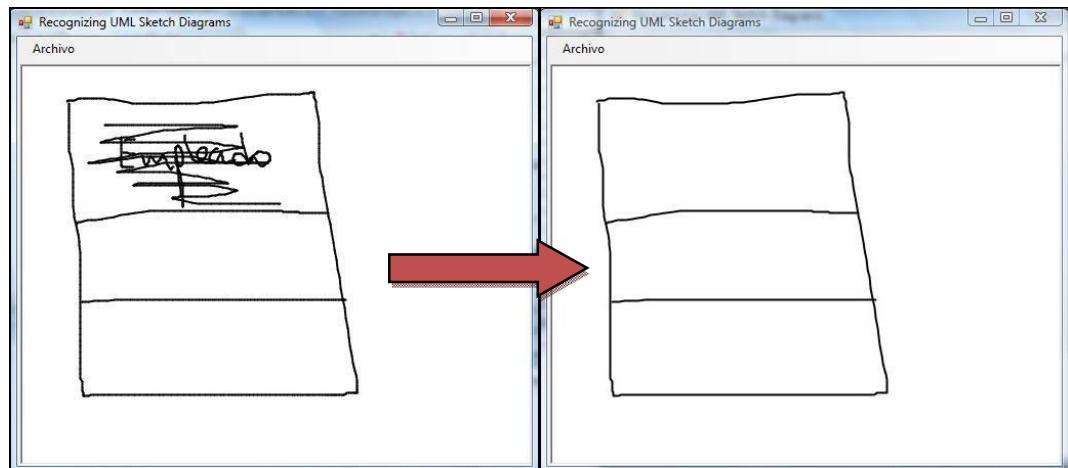


Fig. 5.5. Borrado de trazos.

Una vez terminado el diagrama, debe pulsarse la opción Exportar en el menú archivo (Véase Fig. 5.6).

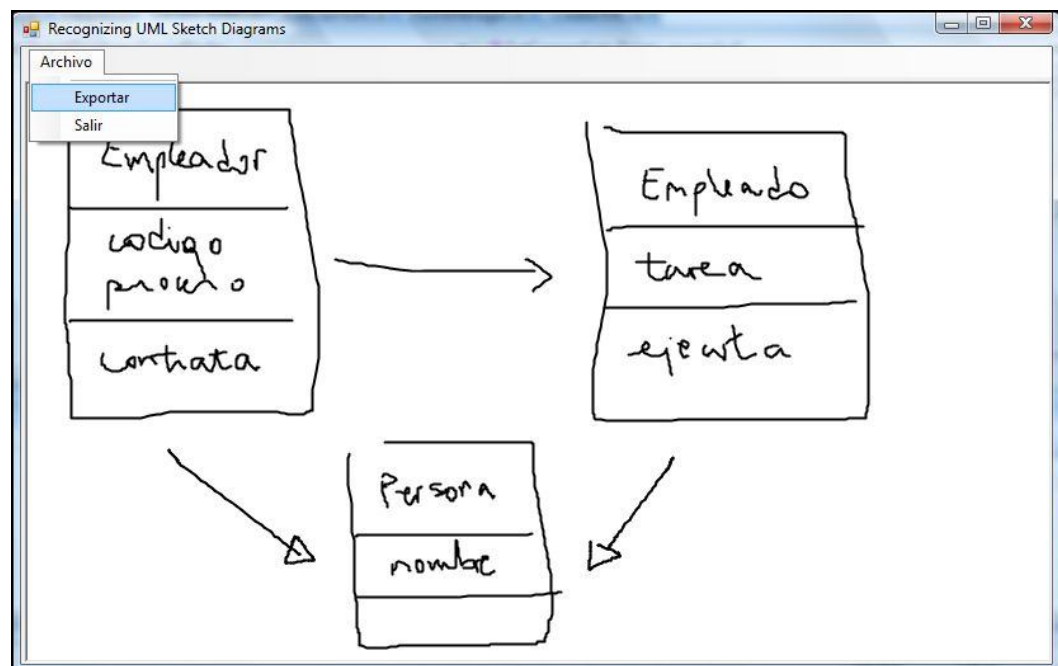


Fig. 5.6. Archivo->Exportar.

Inmediatamente aparece el cuadro de diálogo para seleccionar la carpeta donde se desea guardar el archivo y escribir el nombre que se le quiere dar (Véase Fig. 5.7).

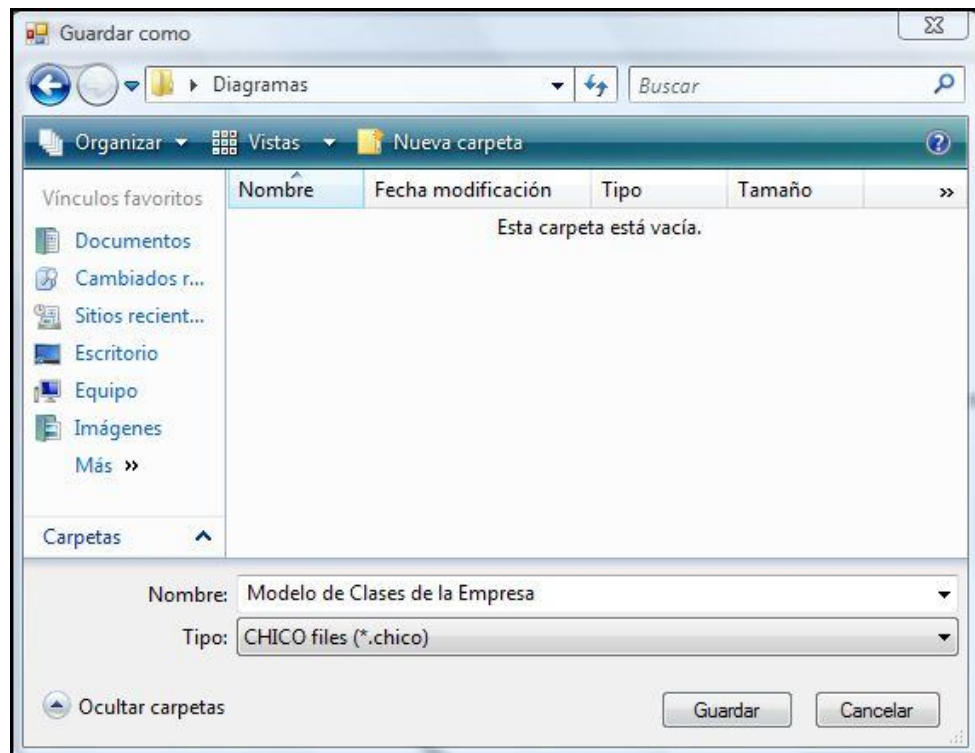


Fig. 5.7. Cuadro de diálogo para guardar el archivo .chico.

En caso de no introducir ruta, el reconocimiento del diagrama no se llevará a cabo y se mostrará un mensaje de advertencia (Véase Fig. 5.8).

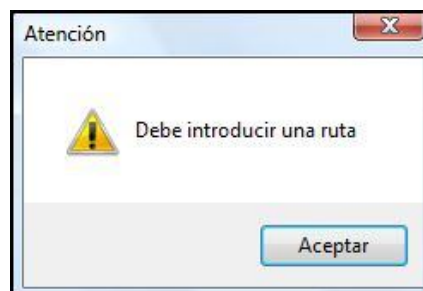


Fig. 5.8. Advertencia de que se debe introducir una ruta para el archivo.

Si se introduce la ruta, el reconocimiento se lleva a cabo. Cuando el proceso se complete se mostrará un mensaje indicando que el archivo se ha exportado (Véase Fig. 5.9).



Fig. 5.9. Indicativo de que el archivo se ha creado.

Una vez creado, el archivo se guarda en la carpeta indicada (Véase Fig. 5.10).

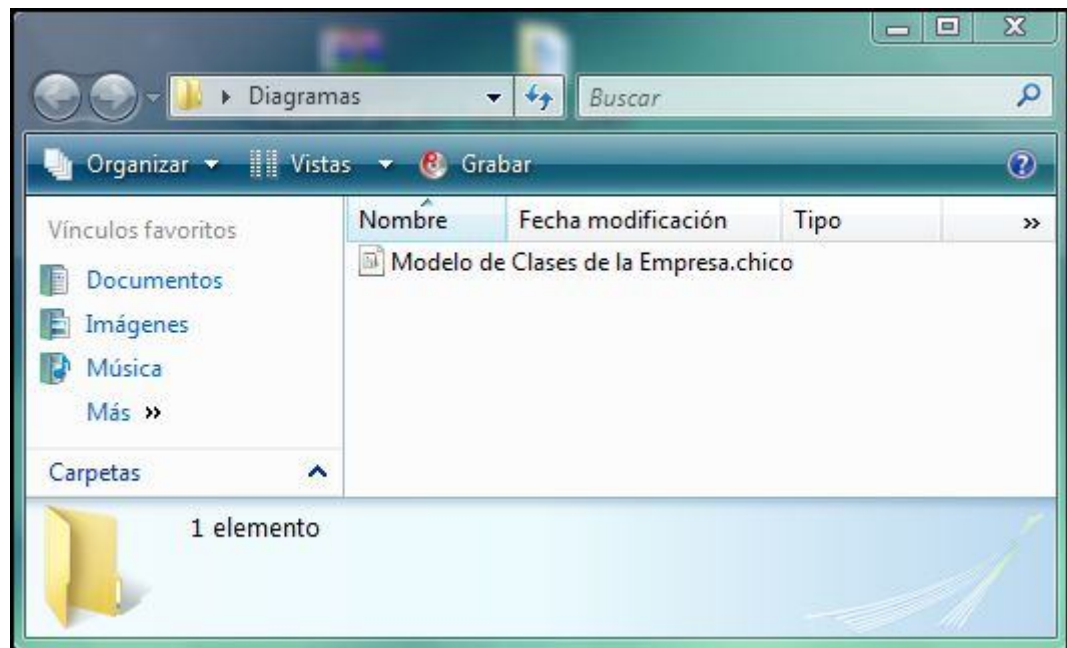


Fig. 5.10. Archivo guardado.

Si se abre el archivo, puede verse el código generado (Véase Fig. 5.11):

```

<?xml version="1.0" encoding="utf-8"?>
<com.chico.chico:Model xmi:version="2.0"
xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:Domain_Package="Domain_Package"
xmlns:com.chico.chico="com.chico.chico">
  <itsClasses name="Empleador">
    <itsAttributes name="codigo"/>
    <itsAttributes name="proceso"/>
    <itsMethods name="contrata"/>
  </itsClasses>
  <itsClasses name="Empleado">
    <itsAttributes name="tarea"/>
    <itsMethods name="ejecuta"/>
  </itsClasses>
  <itsClasses name="Persona">
    <itsAttributes name="nombre"/>
  </itsClasses>
  <itsRelationships xsi:type="Domain_Package:Generalization"
source="//@itsClasses.0" target="//@itsClasses.2"/>
  <itsRelationships xsi:type="Domain_Package:Association"
source="//@itsClasses.0" target="//@itsClasses.1"/>
  <itsRelationships xsi:type="Domain_Package:Generalization"
source="//@itsClasses.1" target="//@itsClasses.2"/>
</com.chico.chico:Model>

```

Fig. 5.11. Código generado en el archivo .chico.

Este archivo .chico, es el que se importa en la aplicación desarrollada por el grupo CHICO. En la imagen siguiente (Véase Fig. 5.12) puede verse la integración del diagrama en el entorno mencionado. Notar que la herramienta del grupo CHICO dispone de *autolayout*, por lo que es normal que las clases no se muestren en la misma posición en que se dibujaron.

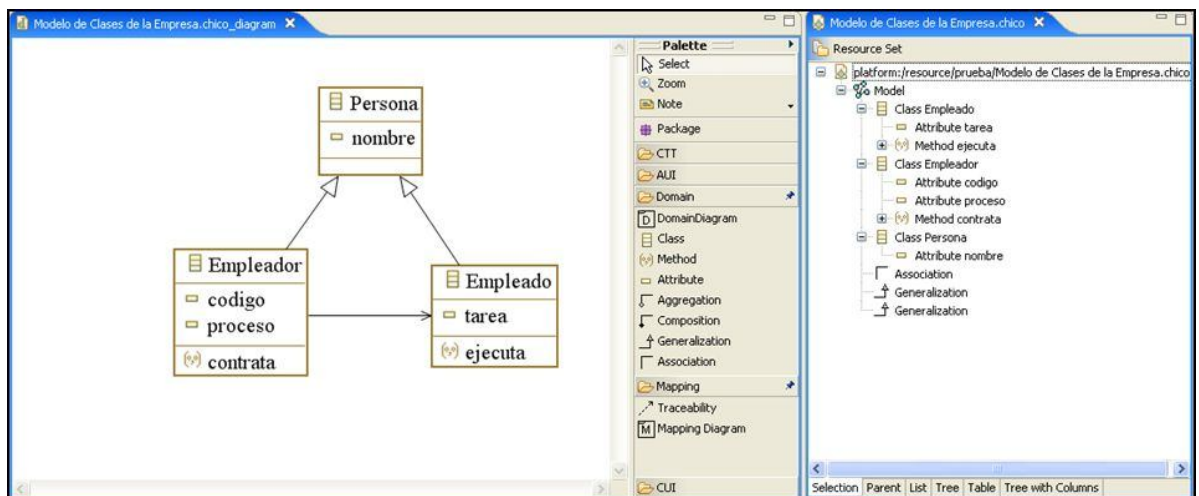


Fig. 5.12. Diagrama importado en el entorno desarrollado por el grupo CHICO.

### 5.1.1. Restricciones

Como se ha comentado en apartados anteriores (Véase sección 3.6.1), la dificultad de diseñar este tipo de aplicaciones reside en crear una aplicación flexible y a la vez robusta, por lo que generalmente el usuario debe seguir algunas pautas, que por otro lado resultan lógicas, para que el reconocimiento se lleve a cabo sin problemas. Este caso no es una excepción, por lo que el usuario deberá tener en cuenta lo siguiente a la hora de dibujar:

- Un trazo que sea una recta, no puede pintarse en varios trazos. Por ejemplo, si se está dibujando una asociación, no puede dibujarse en varias partes, sino que tiene que debe crearse en un solo trazo (Véanse Fig. 5.13 y Fig. 5.14).

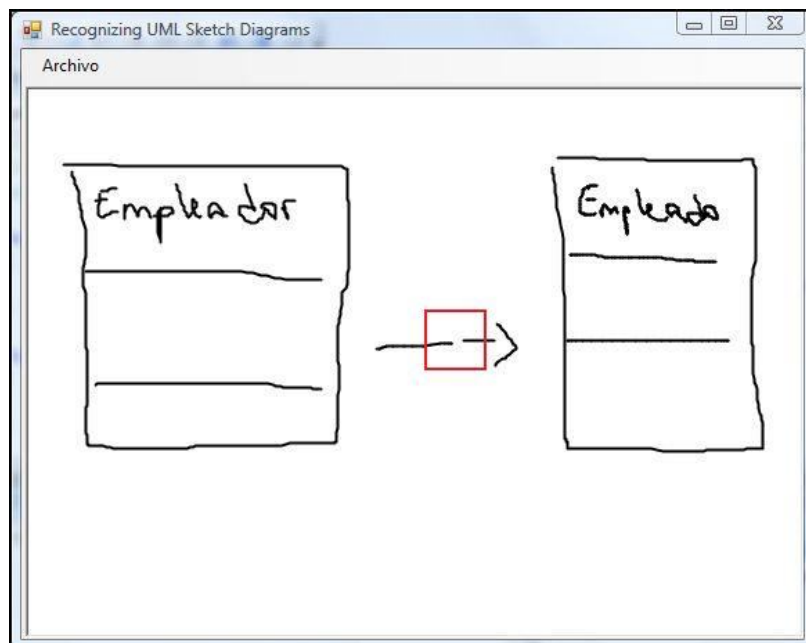


Fig. 5.13. Trazos divididos, incorrecto.

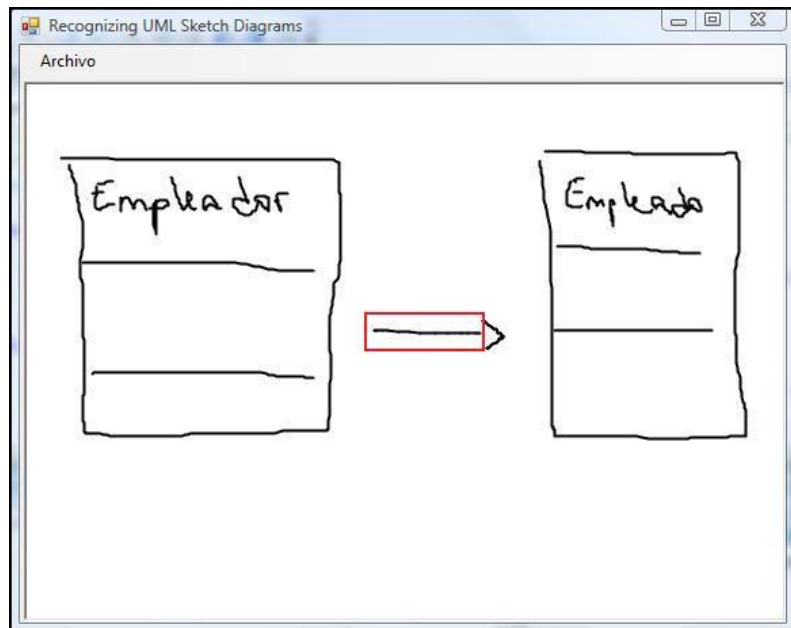


Fig. 5.14. Trazos enteros, correcto.

- Es preferible que las relaciones no se dibujen llegando o saliendo de las esquinas de las clases, ya que podría ocurrir que el cuerpo de la flecha se confunda con alguna línea de la clase y ésta no se reconozca como tal (Véanse Fig. 5.15 y Fig. 5.16).

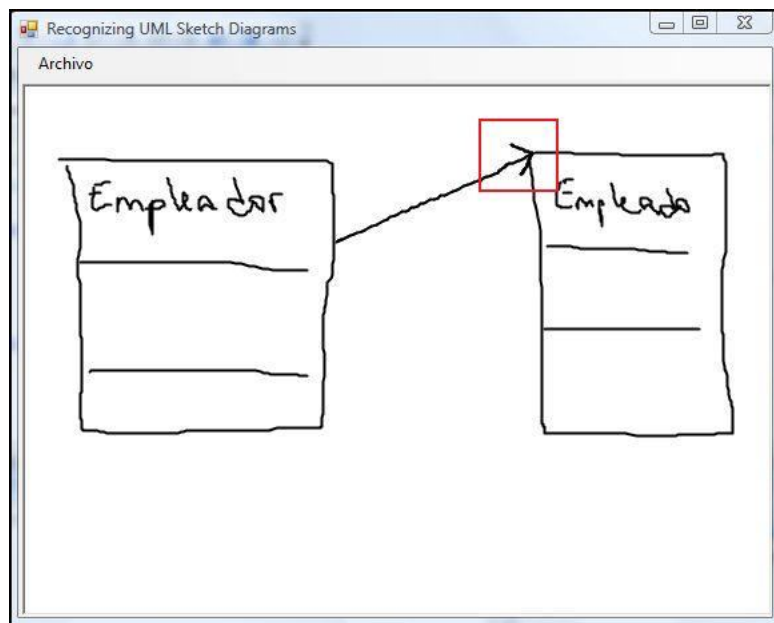


Fig. 5.15. Relaciones a esquinas, incorrecto.



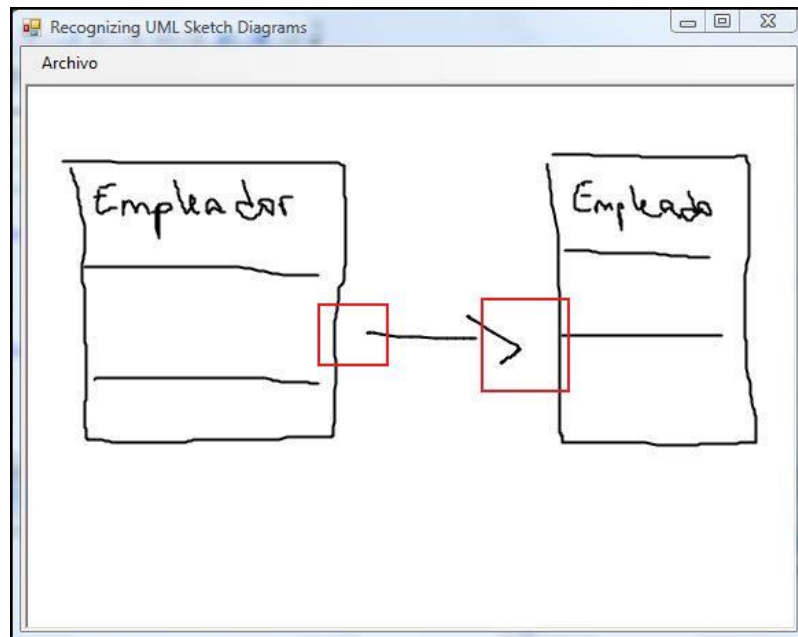


Fig. 5.16. Relaciones a otra parte de las clases que no sean las esquinas, correcto.

- El texto de las clases debe ir dentro de ésta (Véanse Fig. 5.17 y Fig. 5.18).

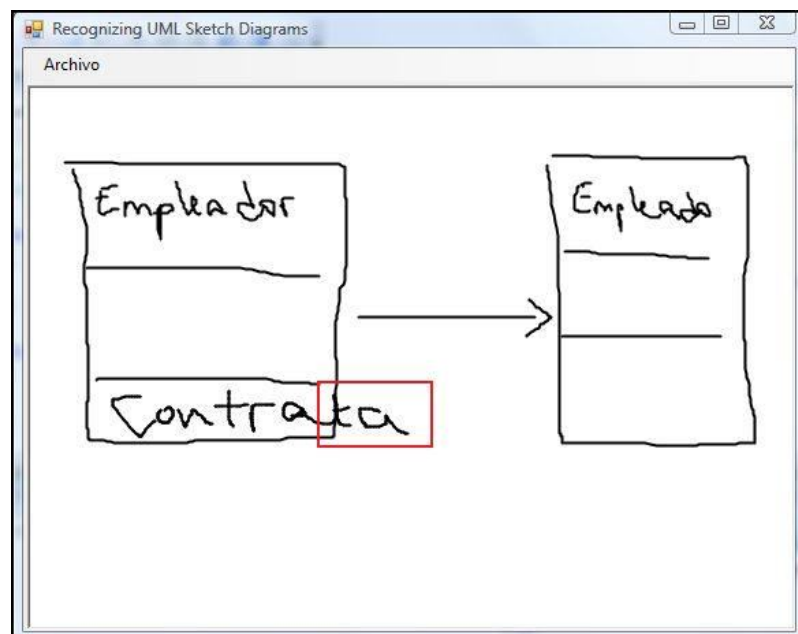


Fig. 5.17. Texto fuera de las clases, incorrecto.

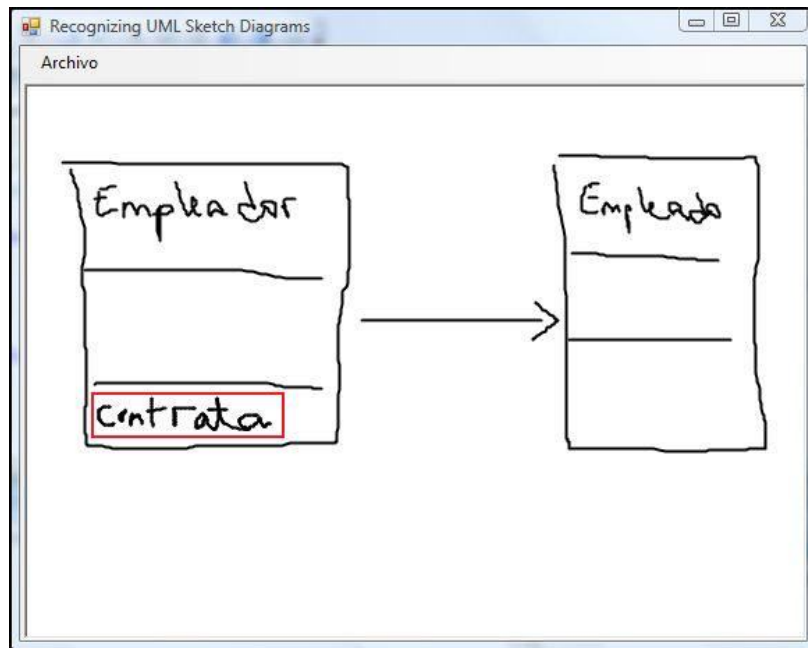


Fig. 5.18. Texto en el apartado que le corresponde, correcto.

- El sistema sólo reconoce el nombre de los métodos y atributos de las clases, no el tipo al que pertenecen ni, en el caso de los métodos, los parámetros pasados y los devueltos (Véanse Fig. 5.19 y Fig. 5.20).

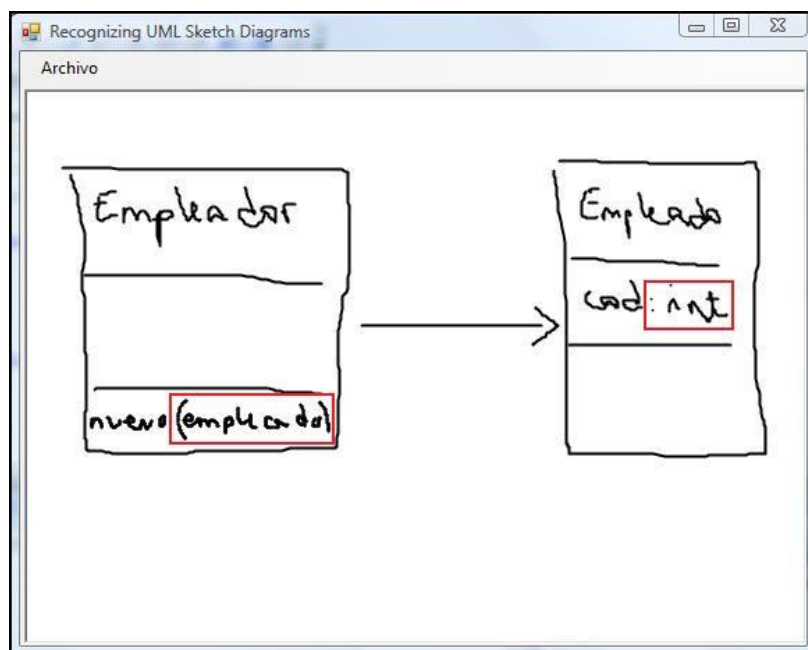


Fig. 5.19. Más información aparte del nombre de la clase, los atributos y los métodos, incorrecto.

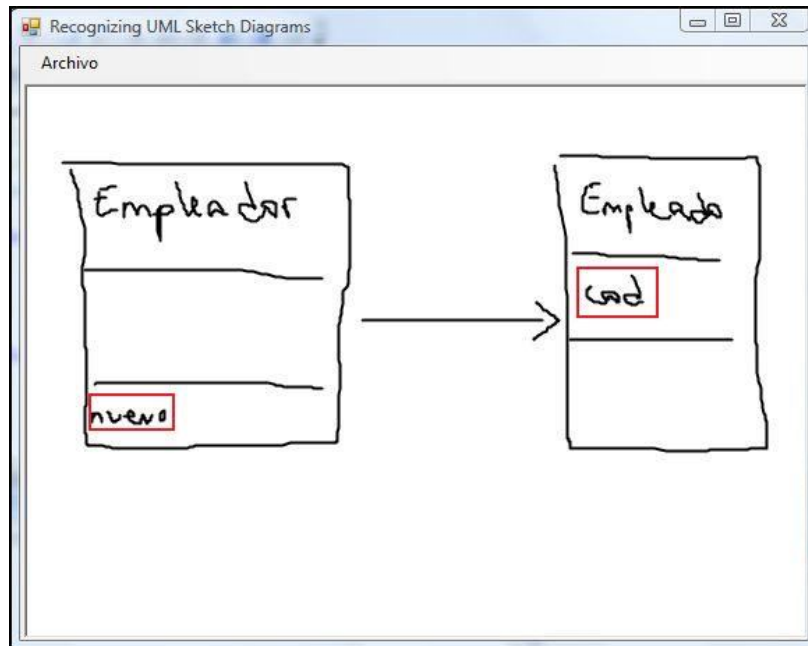


Fig. 5.20. Sólo el nombre de la clase, los atributos y métodos, correcto.

- No puede dibujarse la línea de separación entre el nombre de la clase y los atributos inmediatamente después de escribir el nombre de la misma (Véase Fig. 5.21).

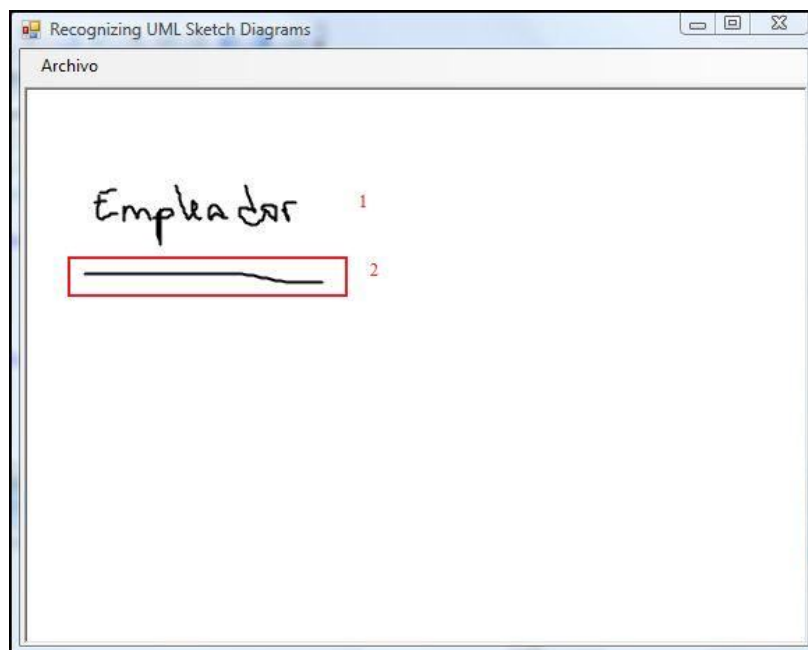


Fig. 5.21. Dibujar primero el nombre de la clase y segundo la línea de separación con los atributos, incorrecto.

- Los trazos deben ser limpios, es decir, no pueden repasarse (Véanse Fig. 5.22 y Fig. 5.23)

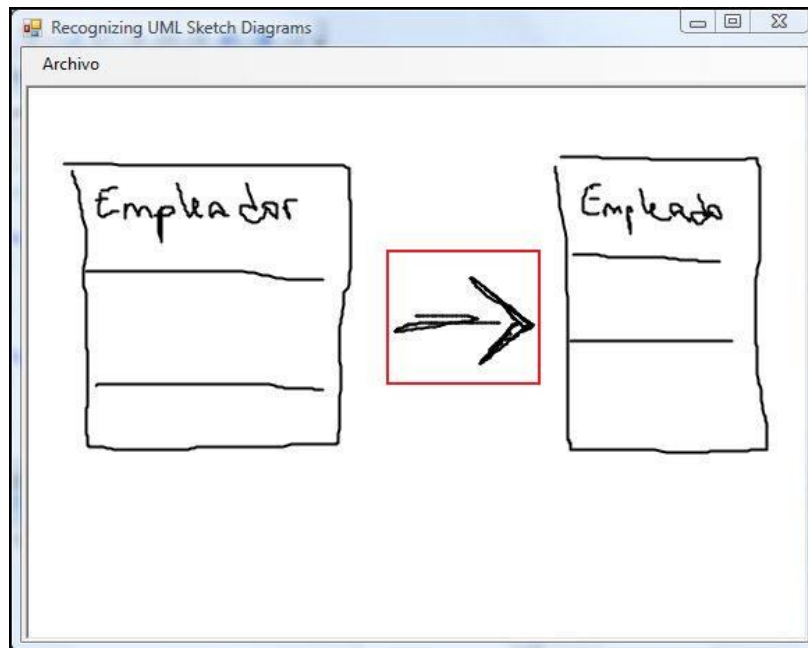


Fig. 5.22. Trazos repasados, incorrecto.

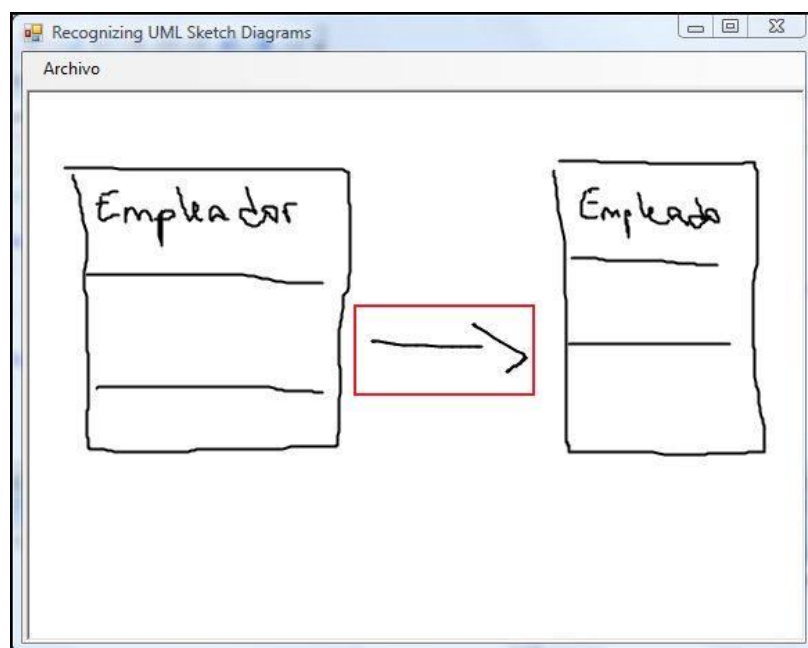


Fig. 5.23. Trazos limpios, correcto.

## 5.2. USABILIDAD DE LA APLICACIÓN

La aplicación desarrollada cumple con todos los criterios de usabilidad que recomienda el equipo de Microsoft [Jarret y Su, 2003]. Aunque estos criterios se han explicado en el apartado 3.2, se vuelven a comentar a continuación, pero en este caso, aplicados a nuestra herramienta:

- Es preferible dejar a un lado los diseños de interfaces de usuario basados en entrada para lápiz y continuar con los tradicionales. La aplicación (Véase Fig. 5.24) sigue el diseño normal de las interfaces de usuario, con el menú en la parte superior.

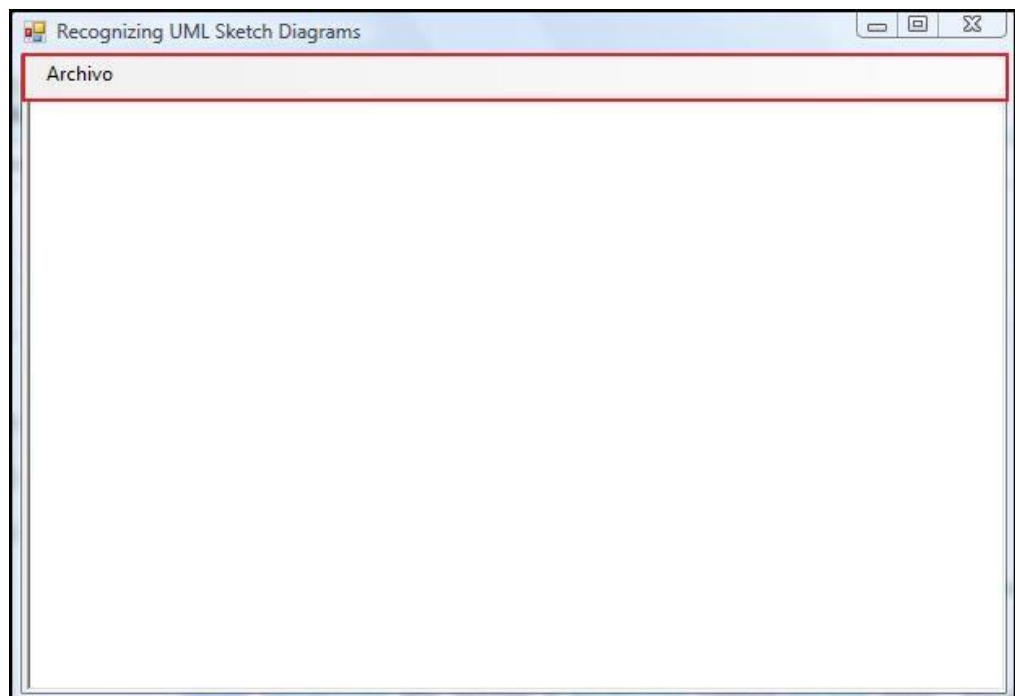


Fig. 5.24. Controles de la aplicación.

- Los usuarios prefieren que sus anotaciones permanezcan como tinta digital en lugar de convertirse a texto. En todo momento el usuario trabaja con sus trazos (Véase Fig. 5.25), dejando la transformación en trazos formales a la herramienta desarrollada por el grupo CHICO (Véase Fig. 5.26).

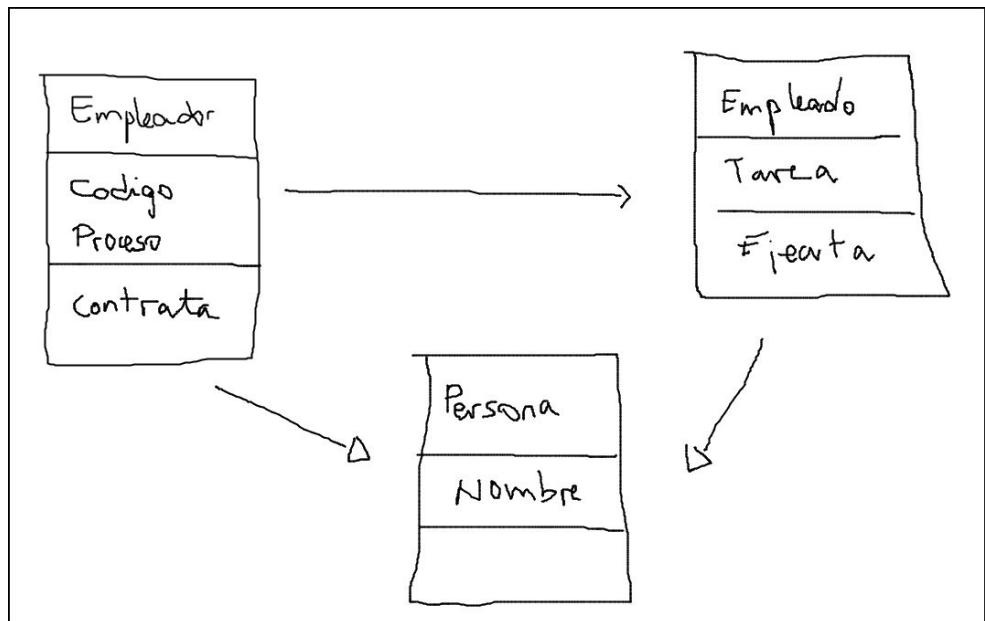


Fig. 5.25. Trazos naturales.

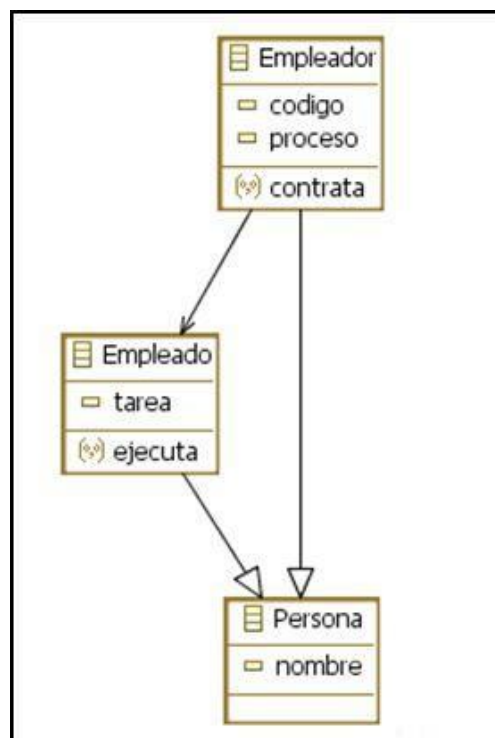


Fig. 5.26. Trazos formales.

- Se deben distinguir de alguna forma en la aplicación, las zonas donde se puede introducir tinta digital y en las que no. En la herramienta desarrollada el puntero cambia de forma dependiendo de la zona en que se mueva. Para las zonas editables, el puntero toma forma de punto (Véase Fig. 5.27) y para las no editables mantiene su forma natural de flecha (Véase Fig. 5.28).

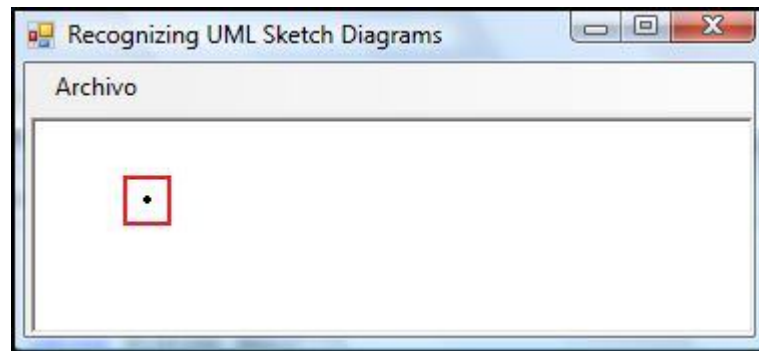


Fig. 5.27. Zona editable.

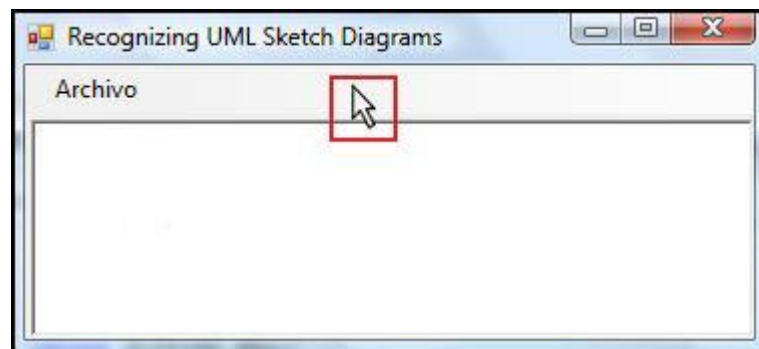


Fig. 5.28. Zona no editable.

- Las aplicaciones no deben trabajar en píxeles. Para cumplir esta condición, se ha añadido la siguiente línea de código:

```
AutoScaleDimensions = new.SizeF(
    AutoScaleDimensions.Width/(AutoScaleDimensions.Width/4F),
    AutoScaleDimensions.Height/(AutoScaleDimensions.Height/8F));
```

Con esto, se consigue que la aplicación no trabaje en píxeles, sino que se escale de acuerdo a las características de la pantalla.

- La modalidad disminuye la eficiencia de la aplicación, por lo que lo ideal sería trabajar con el lápiz de forma natural, sin tener que especificar mediante los distintos modos lo que se desea hacer en cada momento. La herramienta desarrollada dispone de un único modo, el usuario no debe especificar qué desea hacer en cada momento, simplemente tiene que hacerlo.

### 5.3. COMPARATIVA CON LAS APLICACIONES EXISTENTES

A continuación (Véase Tabla 5.1) se muestra una comparativa con las herramientas existentes de reconocimiento de diagramas de clases a la que se ha añadido la desarrollada. En ella se resaltan los mejores resultados.

Puede observarse cómo en todos los casos excepto en uno (relaciones reconocidas), la aplicación desarrollada en el presente proyecto fin de carrera obtiene los mejores resultados (buscando siempre la naturalidad a la hora de dibujar y una buena usabilidad). Incluso el apartado en que no se obtienen los mejores resultados es relativo, ya que únicamente se ve superada por Ideogramic UML, que, como se ha explicado con anterioridad, obliga al usuario a dibujar de forma determinada por emplear reconocimiento mediante *gestures*. Si miramos las aplicaciones basadas en geometría, la nuestra también obtendría los mejores resultados en cuanto a la variedad de relaciones que se reconocen.



	Tipo de Reconocimiento	Momento del Reconocimiento	Transformación a trazos formales	Relaciones	Texto
<b>Tahuti</b>	Basado en Geometría	Al terminar el trazo	Al cambiar de vista	Asociación Unidireccional, Herencia, Agregación	Teclado virtual
<b>Ideogramic</b>	Basado en <i>Gestures</i>	Al terminar el trazo	Al terminar el trazo	Asociación Unidireccional, Asociación, Herencia, Agregación, Composición, Dependencia	Teclado virtual
<b>SketchUML</b>	Geometría-> Clases <i>Gestures</i> -> Triángulo y Diamante	Al terminar el objeto	Al terminar el trazo	Asociación, Herencia, Agregación	Manual en la zona indicada
<b>Interactive System for Recognizing Hand Drawn UML Diagrams</b>	Basado en Geometría Patrones Temporales	Al terminar el trazo	Cuando indique el usuario	Asociación, Unidireccional, Asociación, Herencia	Manual
<b>SUMLOW</b>	Basado en Geometría Patrones Temporales	Al terminar el trazo	Cuando indique el usuario	Asociación Unidireccional, Herencia	Manual en la zona indicada
<b>PFC</b>	Basado en Geometría	Cuando indique el usuario	Al exportarlo, en la otra herramienta	Asociación Unidireccional, Asociación, Herencia, Agregación	Manual

Tabla 5.1. Comparativa de las aplicaciones de reconocimiento de diagramas de UML incluyendo la desarrollada.

## 5.4. CUMPLIMIENTO DE OBJETIVOS

A continuación se comprueba cómo se han conseguido alcanzar todos los objetivos planteados al comienzo del documento:

- Estudio de las técnicas de reconocimiento de trazos: Estas técnicas se han estudiado y se ha escogido para realizar el proyecto la que más libertad de dibujo permitía al usuario (técnica de reconocimiento basada en geometría).
- Búsqueda de soluciones y/o librerías que den soporte total o parcial a la problemática del reconocimiento de trazos: Se ha llevado a cabo un amplio estudio de cómo solucionar cada una de las funcionalidades de la aplicación, así como de la librería que permite trabajar con tinta digital.
- Estudio del dominio de los diagramas de clases: Era una de las bases para poder desarrollar la aplicación, por lo que se ha cumplido el objetivo.
- Realización de una interfaz de usuario usable, de interacción fácil e intuitiva: Se han cumplido todos los puntos que el equipo de Microsoft indica para conseguir una aplicación usable.
- Crear una aplicación que permita al usuario dibujar de forma natural: Aunque existen algunas limitaciones a la hora de dibujar, éstas son mínimas e intuitivas, por lo que el usuario casi puede dibujar de forma natural.

Ello permite dar como alcanzada la hipótesis de trabajo establecida al principio de éste documento, estableciendo que: *Es posible crear un sistema de edición y reconocimiento de Diagramas de Clases UML basado en trazos naturales.*

## 6. CONCLUSIONES Y PROPUESTAS DE MEJORA

En este capítulo se explican las conclusiones obtenidas al finalizar el presente proyecto fin de carrera, así como las posibles mejoras futuras que pueden llevarse a cabo en la aplicación desarrollada.

### 6.1. CONCLUSIONES

Como se ha comentado en el capítulo anterior, se han cubierto y satisfecho los objetivos planteados al inicio del proyecto, pero llegar a este punto no ha sido sencillo.

En primer lugar, al comienzo del proyecto, no se tenía ningún conocimiento sobre el funcionamiento de las Tablet PC, ni sobre las librerías existentes para desarrollar este tipo de aplicaciones, haciéndose necesario familiarizarse con las mismas.

Una vez sentadas las bases sobre las Tablet PC, la búsqueda de información se centró en los tipos de reconocimiento para conocer las ventajas e inconvenientes de cada uno de ellos con el fin de poder desarrollar una aplicación flexible pero robusta a la vez.

A continuación, se estudiaron las distintas herramientas de reconocimiento de diagramas de clases existentes, viendo los puntos fuertes y débiles de cada una de ellas para determinar las características que debería tener una aplicación que permitiese la mayor libertad de dibujo posible al usuario.

Ya con las características de la aplicación elegidas, se llevó a cabo un amplio estudio sobre las posibilidades para implementar cada una de las funcionalidades de ésta. Una vez más, la elección de la implementación de cada una de las partes en las que se divide el proyecto, se hizo buscando que el usuario pudiese dibujar de la forma más natural posible.

A pesar de disponer de una parte del código amablemente cedida por Beryl Plimmer), éste no se ha copiado íntegramente, sino que se ha corregido y adaptado para alcanzar los objetivos que buscaba la aplicación así como para facilitar su lectura y evitar sobrecargas innecesarias de los métodos.

De igual modo, tuvo que cambiarse una parte de la implementación indicada por el pseudocódigo del artículo de Wolin y Hammond [Wolin y Hammond, 2008], ya que tal y como estaba no reconocía todas las esquinas del modo que debiera. Pero éste no fue el mayor contratiempo encontrado con este artículo, sino la existencia de dos versiones distintas del mismo. En un principio se trabajó, sin saberlo, con el artículo erróneo, con todo el retraso en la realización del proyecto que eso conlleva.

A pesar de haber tomado las decisiones que se han considerado mejores para desarrollar una herramienta de reconocimiento flexible y robusta, ésta no está libre de cometer errores, ya que no sólo entra en juego el código implementado, sino que también tiene un papel importante el ruido que puede añadir el usuario a los trazos, así como el reconocedor de texto proporcionado por el kit de desarrollo de aplicaciones para Tablet PC.

Una vez que se consiguió reconocer los diagramas de clases, se pensó que sería una buena idea integrar la aplicación desarrollada con alguna herramienta CASE para obtener un diagrama formal. Es más, teniendo el diagrama integrado en una herramienta de este tipo, podría generarse automáticamente el esqueleto del código a seguir.

Para finalizar, decir que se ha conseguido implementar una aplicación de reconocimiento de diagramas de clases flexible y robusta en la medida de lo posible, que reúne las mejores características de las aplicaciones existentes en cuanto a la libertad que se concede al usuario se refiere. Además de esto, permite la integración con otra herramienta para transformar los trazos en un diagrama formal.

## 6.2. PROPUESTAS DE MEJORA

Aunque la aplicación cumple con los objetivos propuestos existen algunas mejoras que podrían llevarse a cabo, tanto para mejorar su eficiencia como para ampliar su ámbito de uso. Estas mejoras son las siguientes:

- Actualmente la aplicación presenta un orden de complejidad cuadrático debido a las búsquedas de las líneas más cercanas, ya que se miran todas las líneas y todos

los puntos de éstas. Si se buscan soluciones con un orden de complejidad menor, la velocidad de respuesta de la aplicación mejorará considerablemente.

- Puede pensarse la opción de guardar los diagramas realizados como imágenes. Esta opción podría tener como objetivo, entre otros, el pedagógico. Un ejemplo sería que los alumnos realizan sus diagramas de clase, los guardan como imagen y el profesor puede abrirlos y pintar sobre ellos para corregirlos en clase.
- La aplicación exporta a un tipo de archivos que sólo es legible por la aplicación desarrollada por el grupo CHICO. Podría pensarse en exportarlo como XMI (*XML Metadata Interchange*), de forma que pueda ser interpretado por herramientas como Rational Rose.
- Puede añadirse a la herramienta un algoritmo de suavizado de trazos o incluso la posibilidad de transformar en la misma los trazos naturales por los formales.
- Cuanto más se trabaje en estos campos, mejores serán los resultados obtenidos. Beryl Plimmer nos ha comentado que están trabajando en un nuevo *Divider* para mejorar la clasificación de los trazos en texto y figuras. Se propone por tanto, y cuando se termine, emplear este nuevo *Divider* para solventar las restricciones del presente proyecto derivadas del empleo del actual *Divider*, como puede ser el no poder dibujar la línea de separación del nombre de la clase y los atributos inmediatamente después de escribir el nombre de ésta.

## 7. REFERENCIAS BIBLIOGRÁFICAS

- [Albahari y Albahari, 2007] Albahari, J.; Albahari, B. 2007. *C# 3.0 In a Nutshell. A Desktop Quick Reference*. O'Reilly.
- [Alvarado, 2000] Alvarado, C. 2000. *A natural sketching environment: Bringing the computer into early stages of mechanical design*. Master's thesis, Massachusetts Institute of Technology, 2000.
- [Alvarado y Davis, 2004] Alvarado, C.; Davis, R. 2004. SketchREAD: A Multi-Domain Sketch Recognition Engine. En "UIST '04, páginas 24-27".
- [Apte y Kimura, 1993] Apte, A.; Vo, V.; Kimura, T. D. 1993. *Recognizing multistroke geometric shapes: An experimental evaluation*. En "UIST, páginas 121-128".
- [Archer, 2001] Archer, T. 2001. *C# A Fondo*. Mc Graw Hill.
- [Bishop et al., 2004] Bishop, C. M.; Svensén, M.; Hinton, G. E. 2004. *Distinguishing Text from Graphics in On-Line Handwritten Ink*. En "Proceedings of the 9<sup>th</sup> Int'I Workshop on Frontiers on Handwriting Recognition".
- [Booch et al., 1998] Booch, G.; Rumbaugh, J.; Jacobson, I. 1998. *The Unified Modeling Language User Guide*. Addison-Wesley.
- [Booch et al., 1999] Booch, G.; Rumbaugh, J.; Jacobson, I. 1999. *The Unified Modeling Language Reference Manual*. Addison-Wesley.
- [Caetano et al., 2002] Caetano, A.; Goulart, N; Fonseca, M.; Jorge, J. 2002. *JavaSketchIt: Issues in Sketching the Look of User Interfaces*. American Association for Artificial Intelligence.
- [Croocks, 2004] Crooks, C. E. 2004. *Developing Tablet PC Applications*. Charles River Media.
- [Damm et al., 2000a] Damm, C. H.; Hansen, K. M; Thomsen. 2000a. *Tool Support for Cooperative Object-Oriented Design: Gesture Based Modeling on an Electronic Whiteboard*. En "Proceedings of Computer Human Interaction".

- [Damm et al., 2000b] Damm, C. H.; Hansen, K. M.; Thomsen, M.; Tyrsted, M. 2000b. *Creative Object-Oriented Modelling: Support for Intuition, Flexibility, and Collaboration in CASE Tools*. En ECOOP 2000, páginas 27-43.
- [Damm y Hansen, 2002] Damm, C. H.; Hansen, K. M. 2002. *Ideogramic*. <http://www.ideogramic.com>
- [Dixon y Hammond, 2009] Dixon, D.; Hammond, T. 2009. *A Methodology Using Assistive Sketch Recognition for Improving a Person's Ability to Draw*. Texas A&M University Thesis.
- [Genari et al., 2005] Gennari, L.; Kara, L. B.; Stahovich, T. F. 2005. *Combining geometry and domain knowledge to interpret hand-drawn diagrams*. En "Computers and Graphics: Special Issue on Pen-Based User Interfaces".
- [Hammond y Davis, 2002] Hammond, T.; Davis, R. 2002. *Tahuti: A Geometrical Sketch Recognition System for UML Class Diagrams*. En "Proceedings of 2002 AAAI Spring Symposium on Sketch Understanding".
- [Hammond y Davis, 2005] Hammond, T.; Davis, R. 2005. *LADDER, a sketching language for user interface developers*. Elsevier
- [Hansen y Ratzer, 2002] Hansen, K. M.; Ratzer, A. V. 2002. *Tool Support for Collaborative Teaching and Learning of Object-Oriented Modeling*. En "ITiCSE '02"
- [Jarret y Su, 2003] Jarret, R.; Su, P. 2003. *Building Tablet PC Applications*. Microsoft Press.
- [La Viola y Zeleznik, 2004] La Viola, J.; Zeleznik, R. 2004. *MathPad<sup>2</sup>: A System for the Creation and Exploration of Mathematical Sketches*. En "ACM Transactions on Graphics (Proceedings of SIGGRAPH 2004)", páginas 432-440"
- [Landay, 1996] Landay, J. A. 1996. *SILK: Sketching Interfaces Like Krazy*. En "CHI '96", páginas 13-18".
- [Lank et al., 2000] Lank, E.; Thorley, J.S.; Chen, S. J.-S. 2000. *An Interactive System for Recognizing Hand Drawn UML Diagrams*. En "Proceedings for CASCON 2000".

[Larman, 1999] Larman, C. 1999. *UML y Patrones. Introducción al análisis y diseño orientado a objetos*. Prentice Hall.

[Long et al., 1999] Long, Jr. A.C.; Landay, J.A.; Rowe, L.A. 1999. *Implications for a gesture design tool*. En “Human Factors in Computing Systems (SIGCHI Proceedings), páginas 40-47”. ACM Press.

[Long et al., 2000] Long, Jr. A.C.; Landay, J.A.; Rowe, L.A, Michiels, J. 2000. *Visual similarity of pen gestures*. En “ACM Conference on Human Factors in Computing Systems, CHI Letters, páginas 360-367”.

[Long et al., 2001] Long, Jr. A.C.; Landay, J.A.; Rowe, L.A. 2001. *Those Look Similar! Issues in Automating Gesture Design Advice*.

[MSDN] MSDN. <http://msdn.microsoft.com/en-us/library/default.aspx>

[Oltmans, 2007] Oltmans, M. 2007. *Envisioning sketch recognition: a local feature based approach to recognizing informal sketches*. Ph.D. dissertation, Massachusetts Institute of Technology.

[Ouyang y Davis, 2007] Ouyang, T. Y.; Davis, R. 2007. Recognition of Hand-Drawn Chemical Diagrams. En “Proceedings AAAI 2007”.

[Patel et al., 2007] Patel, R.; Plimmer, B.; Grundy, J.; Ihaka. R. 2007. *Ink Features for Diagram Recognition*. SBIM. California, IEEE.

[Paulson et al., 2008a] Paulson, B.; Rajan, P.; Davalos, P.; Gutierrez-Osuna, R.; Hammond, T. 2008a. *What!?! No Rubine Features?: Using Geometric-based Features to Produce Normalized Confidence Values for Sketch Recognition*. En “VL/HCC Workshop: Sketch Tools for Diagramming”.

[Paulson y Hammond, 2008] Paulson, B.; Hammond, T. 2008. *PaleoSketch: accurate primitive sketch recognition and beautification*. En “Proc. of 13th International Conference on Intelligent User Interfaces (IUI), páginas 1-10”.

[Paulson et al., 2008b] Paulson, B.; Eoff, B.; Wolin, A.; Johnston, J.; Hammond, T. 2008b. *Sketch-Based Educational Games: "Drawing" Kids Away from Traditional Interfaces*. En “7th International Conference on Interaction Design and Children Posters”.



[Plimmer y Grundy, 2005] Plimmer, B.; Grundy, J. 2005. *Beautifying Sketching-based Design Tool Content: Issues and Experiences*. En “6<sup>th</sup> Australasian User Interface Conference (AUIC2005)”.

[Polo, 2008] Polo, M. 2008. *Apuntes de Ingeniería del Software II*.

[Qiu, 2007] Qiu, L. 2007. *SketchUML: The Design of a Sketch-based Tool for UML Class Diagrams*. En “Proceedings of World Conference on Educational Multimedia, Hypermedia and Telecommunications, páginas 986-994”

[Rubine, 1991] Rubine, D. 1991. *Specifying gestures by example*. En “Computer Graphics, páginas 329-337”. ACM SIGGRAPH, Addison Wesley.

[Sezgin y Stahovich, 2001] Sezgin, T. M.; Stahovich, T.; Davis, R. 2001. *Sketch Based Interfaces: Early Processing for Sketch Understanding*. En “The Proceedings of 2001 Perceptive User Interfaces Workshop (PUI’01), páginas 1-8”.

[SketchUML] SketchUML. <http://sketchuml.tenbergen.org/>

[Sutherland, 1963] Sutherland, I. B. 1963. *Sketchpad, a man-machine graphical communication system*. En “Proceedings of the Spring Joint Computer Conference, páginas 329–346”.

[Taele y Hammond, 2008] Taele, P. Hammond, T. 2008. *A Geometric-based Sketch Recognition Approach for Handwritten Mandarin Phonetic Symbols I*. En “Visual Languages and Computation, VLC 2008 , páginas 270-275”.

[Taele y Hammond, 2010] Taele, P. Hammond, T. 2010. *LAMPS: A Sketch Recognition-Based Teaching Tool for Mandarin Phonetic Symbols I*. En “Visual Languages and Computation, VLC 2010”.

[Tenbergen et al., 2008] Tenbergen, B.; Grieshaber, C.; Lazzaro, L.; Buck, R. 2008. *Sketch UML: A Tablet PC-Based E-Learning Tool for UML Syntax Using a Minimalistic Interface*. En “Quest Proceedinds 2008, páginas 43-52”.

[Wobbrock, 2007] Wobbrock, J. O.; Wilson, A. D.; Li, Y. 2007. *Gestures without libraries, toolkits or training: a \$1 recognizer for user interface prototypes*. En “Proc. of

*the 20<sup>th</sup> Annual ACM Symposium on User Interface Software and Technology (UIST)(páginas 159-168)”*

[Wolin y Hammond, 2008] Wolin. A.; Hammond, T. 2008. *ShortStraw: A Simple and Effective Corner Finder for Polylines*. En “Eurographics 4th Annual Workshop on Sketch-Based Interfaces and Modeling”

## ANEXO I

En la siguiente tabla (Véase Tabla I.1) se muestran las *gestures* reconocidas por Ideogramic UML como elementos de los diagramas de clase de UML.


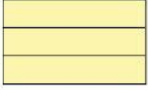

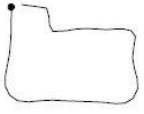
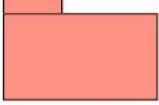











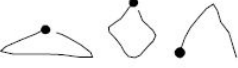
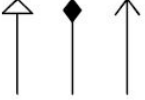
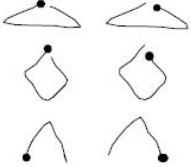

Gesture	Effect	Variations
	Class 	
	Package 	
	Comment 	
	Relationship 	
	Move or resize You may use right mouse button to move or resize elements directly	
	Delete	
	Relationship decorations 	
	Selection You may also use control + left mouse click to select elements.	

Tabla I.1. *Gestures* empleadas en Ideogramic UML.

## ANEXO II

A continuación se muestra el código cedido por Beryl Plimmer de la Universidad de Auckland. Se divide en dos clases: *RachelsDivider.cs* y *ConvexHulls.cs*. De la segunda clase, sólo se muestran los métodos empleados en el proyecto debido a la extensión de la misma.

### *RachelsDivider.cs*

```
class RachelsDivider
{
    private const int SHAPE = 0;
    private const int TEXT = 1;

    private const double bBoxWidth = 1847.5;
    private const double totalAngle = 10.09914;
    private const double distanceLast = 2553.684;
    private const double distanceNext = 1646.576;
    private const double timeNext = 660.8878;
    private const double speedNext = 3.18416;
    private const double amtInk = 51.5;
    private const double perimeter = 25.31871;

    public Guid theTimeGuid = new Guid("{DBAEA5AB-C085-4d0c-8A45-18DD146D1D26}");
    Ink myInk;

    public RachelsDivider(Ink myInk)
    {
        this.myInk = myInk;
    }

    //when there is only one stroke
    public int Divide(Stroke currentS)
    {
        if (GetWidth(currentS) >= bBoxWidth)
            if (GetTotalAngle(currentS) < totalAngle)
                return SHAPE;
            else
                return TEXT;
        //else
    }
```

```

        //if (GetIntersections(currentS) > 3) //different to mine
        //    return SHAPE;
        //else
    else
        if (GetInkInside(currentS) >= amtInk)
            return SHAPE;
        else
            if (GetPerimeterToArea(currentS) < perimeter)
                return SHAPE;
            else
                return TEXT;
    }

    //for the first and last stroke
    public int Divide(Stroke currentS, Stroke otherS, int
PrevOrNext)
    {
        if (PrevOrNext == -1) // previous stroke available
            if (GetWidth(currentS) >= bBoxWidth)
                if (GetTotalAngle(currentS) < totalAngle)
                    return SHAPE;
                else
                    return TEXT;
            else
                //if (GetIntersections(currentS) > 3) //diff to mine
                //    return SHAPE;
                //else
                if (GetDistanceBetweenStrokes(currentS, otherS) >=
distanceLast) //diff to mine
                    return SHAPE;
                else
                    //return TEXT;
                    if (GetInkInside(currentS) >= amtInk)
                        return SHAPE;
                    else
                        if (GetPerimeterToArea(currentS) < perimeter)
                            return SHAPE;
                        else
                            return TEXT;
        else //next available
            if (GetWidth(currentS) >= bBoxWidth)

```

```

        if (GetTotalAngle(currentS) < totalAngle)
            return SHAPE;
        else
            return TEXT;
    else // no distance from last stroke
    {
        double timeToNext = GetTimeToNext(currentS, otherS);
        double distanceToNext =
            GetDistanceBetweenStrokes(otherS, currentS);
        double speedToNext = distanceToNext / timeToNext;

        if (timeToNext > timeNext)
            return SHAPE;
        //else
        //    return TEXT;
        if (distanceToNext >= distanceNext)
        {
            if (speedToNext < speedNext)
            {
                if (GetInkInside(currentS) >= amtInk)
                    return SHAPE;
                else
                {
                    if (GetPerimeterToArea(currentS) <
                        perimeter)
                        return SHAPE;
                    else
                        return TEXT;
                }
            }
            else
                return TEXT;
        }
        else
            return TEXT;
    }
}

//for the middle strokes
public int Divide(Stroke currentS, Stroke prevS, Stroke nextS)
{
    if (GetWidth(currentS) >= bBoxWidth)
        if (GetTotalAngle(currentS) < totalAngle)
            return SHAPE;
        else
            return TEXT;
    else

```

```

        //if (GetIntersections(currentS) > 3)
        //    return SHAPE;
        //else
        if (GetDistanceBetweenStrokes(currentS, prevS) >=
distanceLast)
            if (GetTimeToNext(currentS, nextS) >= timeNext)
                return SHAPE;
            else
                return TEXT;
        else
        {
            double distanceToNext =
GetDistanceBetweenStrokes(nextS, currentS);
            double timeToNext = GetTimeToNext(currentS, nextS);
            double speedToNext = distanceToNext / timeToNext;

            if (distanceToNext >= distanceNext)
                if (speedToNext < speedNext)
                    if (GetInkInside(currentS) >= amtInk)
                        return SHAPE;
                    else
                        if (GetPerimeterToArea(currentS) <
perimeter)
                            return SHAPE;
                        else
                            return TEXT;
                    else
                        return TEXT;
                else
                    return TEXT;
            return TEXT;
        }
    }

private int GetWidth(Stroke s)
{
    return s.GetBoundingBox().Width;
}

private double GetTotalAngle(Stroke s)
{
    Point p1, p2, p3;

```

```

p1 = p2 = p3 = new Point(0, 0);
double temp;

double angleSum = 0;
p1 = s.GetPoint(0);
if (s.GetPoints().Length > 1)
    p2 = s.GetPoint(1);

for (int i = 1; i < s.GetPoints().Length - 1; i++)
{
    p3 = s.GetPoint(i + 1);
    temp = (((p3.X - p2.X) * (p2.Y - p1.Y)) -
            ((p2.X - p1.X) * (p3.Y - p2.Y))) / (double)
            (((p3.X - p2.X) * (p2.X - p1.X)) +
            ((p3.Y - p2.Y) * (p2.Y - p1.Y))));
    temp = Math.Atan(temp);
    if (Double.IsNaN(temp))
    {
        temp = 0;
    }

    angleSum += temp;
    p1 = p2;
    p2 = p3;
}
return angleSum;
}

private double GetInkInside(Stroke s)
{
    double numIntersections = 0;
    int numPoints = s.GetPoints().Length;
    System.Drawing.Rectangle rect1 = s.GetBoundingBox();
    rect1.Inflate(rect1.Width / 5, rect1.Height / 5);

    StrokeIntersection[] intersections1 =
s.GetRectangleIntersections(rect1);
foreach (StrokeIntersection intersection in intersections1)
{
    int beginIndex = (intersection.BeginIndex != -1 ?
        (int)intersection.BeginIndex : 0);

```



```

        int endIndex = (intersection.EndIndex != -1 ?
        (int)intersection.EndIndex : numPoints - 1);

        numIntersections += (endIndex - beginIndex + 1);
    }
    return numIntersections;
}

private double GetDistanceBetweenStrokes(Stroke currentS, Stroke
prevS)
{
    Point p1 = currentS.GetPoint(0);
    Point p2 = prevS.GetPoint(prevS.GetPoints().Length - 1);
    return (Distance(p1, p2));
}

public double GetTimeToNext(Stroke curr, Stroke next)
{
    double startTime = 0;
    double endTime = 0;

    #region first get the start time of the next stroke
    if (next.ExtendedProperties.DoesPropertyExist(theTimeGuid))
    {
        // Get the raw data out of this stroke's extended
        // properties list, using the previously defined
        // Guid as a key to the required extended property.
        long theLong =
        (long)next.ExtendedProperties[theTimeGuid].Data;
        // Then turn it (as a FileTime) into a time string.
        startTime =
        DateTime.FromFileTime(theLong).TimeOfDay.TotalMilliseconds;
    }
    #endregion

    #region now get the end time of this stroke
    if (curr.ExtendedProperties.DoesPropertyExist(theTimeGuid))
    {
        long theLong =
        (long)curr.ExtendedProperties[theTimeGuid].Data;
        endTime =

```

```

        DateTime.FromFileTime(theLong).TimeOfDay.TotalMilliseconds;
    }

    bool fFound = false;
    //from Tablet PC Platform SDK: Ink Data Management, Part II
    //ch 6 code
    for (int j = 0; j < curr.PacketDescription.Length; j++)
    {
        if (curr.PacketDescription[j] ==
            PacketProperty.TimerTick)
        {
            fFound = true;
            break;
        }
    }
    List<int> tempTicks = new List<int>();
    if (fFound)
        tempTicks.AddRange(curr.GetPacketValuesByProperty(
            PacketProperty.TimerTick));

    if (tempTicks.Count > 0)
    {
        endTime = endTime + tempTicks[tempTicks.Count - 1] *
            0.0001;
    }
    #endregion
    return (startTime - endTime);
}

private double GetPerimeterToArea(Stroke s)
{
    ConvexHulls convexHull = new ConvexHulls(s.GetPoints());
    double v = convexHull.GetConvexArea() /
        convexHull.GetConvexPerimeter();
    return v;
}

private double Distance(Point p1, Point p2)
{
    Double asqr = Math.Pow((p1.X - p2.X), 2.0);
    double bsqr = Math.Pow((p1.Y - p2.Y), 2.0);

```

```

        return Math.Sqrt(asqr + bsqr);
    }

private double GetGestureLengthBBoxSize(Stroke stroke)
{
    System.Drawing.Point[] points = stroke.GetPoints();
    System.Drawing.Point point1, point2;

    #region Feature 3: Length of the Gesture

    // Calculate the maximum and minimum points. Could use the
    // bounding box of the stroke, but using the points
    // themselves. Could also calculate when iterating later
    // for the angles.

    double totalGestureLength = 0;
    double minX, maxX, minY, maxY;

    point1 = points[0];

    minX = point1.X;
    maxX = point1.X;
    minY = point1.Y;
    maxY = point1.Y;
    for (int i = 1; i < points.Length; i++)
    {
        if (i < points.Length - 1)
        {
            point2 = points[i + 1];

            totalGestureLength +=
                ExtraMath.ComputeDistance(point1, point2);

            point1 = point2;
        }

        // Max / Min Points
        if (points[i].X < minX)
        {
            minX = points[i].X;
        }
    }
}

```

```

        else
            if (points[i].X > maxX)
            {
                maxX = points[i].X;
            }

            if (points[i].Y < minY)
            {
                minY = points[i].Y;
            }
            else
            {
                if (points[i].Y > maxY)
                {
                    maxY = points[i].Y;
                }
            }
        }

        double boundingBoxSize = (ExtraMath.ComputeDistance(minX,
minY, maxX, maxY));
        #endregion

        #region Feature 8: Total Gesture Length
        return totalGestureLength / boundingBoxSize;
        #endregion
    }

    private int GetXPolylineCusps(Stroke stroke)
    {
        return stroke.PolylineCusps.Length;
    }
}

```

### *ConvexHulls.cs*

```

public class ConvexHulls
{
    /// <summary>
    /// The amount that defines a different angle when calculating
    /// the smallest
    /// enclosing boundingBox.
    /// </summary>

```

```

private const double RECT_ROTATE_ANGLE = Math.PI / 8;
private Point[] convexPoints;
private double convexPerimeter = 0;
private double convexArea = 0;

// Creates a convex hull from the specified points.
public ConvexHulls(Point[] points)
{
    convexPerimeter = 0;
    convexArea = 0;
    convexPoints = CalculateConvexHull(points);
}

/// <summary>
/// Uses the Andrew variant of the Graham formula for
/// calculating
/// convex hulls.
/// </summary>
/// <param name="points"></param>
/// <returns></returns>
public Point[] CalculateConvexHull(Point[] points)
{
    if (points.Length < 3)
    {
        return points;
    }

    // TODO: Implement a quicksort algorithm, and find cut off
    // point for switching algorithms.
    points = BubblesortPoints(points);

    List<Point> upper = new List<Point>();

    upper.Add(points[0]);
    upper.Add(points[1]);

    for (int i = 2; i < points.Length; i++)
    {
        upper.Add(points[i]);
        while (upper.Count > 2 && !ExtraMath.IsRightTurn(
            upper[upper.Count - 3], upper[upper.Count - 2],

```

```

        upper[upper.Count - 1]))
    {
        upper.RemoveAt(upper.Count - 2);
    }
}

List<Point> lower = new List<Point>();
lower.Add(points[points.Length - 1]);
lower.Add(points[points.Length - 2]);

for (int i = points.Length - 3; i >= 0; i--)
{
    lower.Add(points[i]);
    while (lower.Count > 2 && !ExtraMath.IsRightTurn(
        lower[lower.Count - 3], lower[lower.Count - 2],
        lower[lower.Count - 1]))
    {
        lower.RemoveAt(lower.Count - 2);
    }
}

Point[] retPoints = new Point[upper.Count + lower.Count];
for (int i = 0; i < upper.Count; i++)
{
    retPoints[i] = upper[i];
}
for (int i = 0; i < lower.Count; i++)
{
    retPoints[i + upper.Count] = lower[i];
}

return retPoints;
}

protected Point[] BubblesortPoints(Point[] points)
{
    points = (Point[])points.Clone();
    for (int i = 0; i < points.Length; i++)
    {
        Point p = points[i];
        int pointIndex = i;
        for (int q = i + 1; q < points.Length; q++)
        {

```

```

        if (points[q].X < p.X || (points[q].X == p.X &&
points[q].Y < p.Y))
        {
            p = points[q];
            pointIndex = q;
        }
    }

    points[pointIndex] = points[i];
    points[i] = p;
}
return points;
}

/// <summary>
/// Calculated the first time it is asked for and cached after
/// that.
/// </summary>
/// <returns></returns>
public double GetConvexPerimeter()
{
    if (this.convexPerimeter == 0)
    {
        double perim = 0;
        for (int i = 0; i < convexPoints.Length - 1; i++)
        {
            perim += ExtraMath.Distance(convexPoints[i],
convexPoints[i + 1]);
        }
        convexPerimeter = perim;
    }
    return convexPerimeter;
}

/// <summary>
/// Returns the area of the convex hull, calculated as the sum
/// of the triangles.
/// </summary>
/// <returns></returns>
public double GetConvexArea()
{

```

```

        if (this.convexArea == 0)
        {
            double area = 0;
            Point point1 = convexPoints[0];
            for (int i = 1; i < convexPoints.Length - 1; i++)
            {
                area += ExtraMath.CalculateTriangularArea(point1,
                    convexPoints[i], convexPoints[i + 1]);
            }
            convexArea = area;
        }
        return convexArea;
    }
}

```



## ANEXO III

En el presente Anexo, se muestran los pseudocódigos del artículo de Wolin y Hammond [Wolin y Hammond, 2008]. Se indica para cada método corregido, la versión errónea y la correcta, resaltando los cambios que se han realizado:

### *GetCorners*

```
GET-CORNERS(points)
1  corners  $\leftarrow \emptyset$ 
2  APPEND(corners, 0)
3  W  $\leftarrow 3$ 
4  for i  $\leftarrow W$  to  $|points| - W$  do
5    strawsi  $\leftarrow$  DISTANCE(pointsi-W, pointsi+W)
6    t  $\leftarrow$  MEDIAN(straws)  $\times 0.95$ 
7    for i  $\leftarrow W$  to  $|points| - W$  do
8      if strawsi < t then
9        localMin  $\leftarrow +\infty$ 
10       localMinIndex  $\leftarrow i$ 
11       while i <  $|straws|$  and strawsi < t do
12         if strawsi < localMin then
13           localMin  $\leftarrow$  strawsi
14           localMinIndex  $\leftarrow i$ 
15         i  $\leftarrow i + 1$ 
16       APPEND(corners i)
17  APPEND(corners,  $|points|$ )
18  corners  $\leftarrow$  POST-PROCESS-
    CORNERS(corners, straws)
19  return corners
```

Fig. III.1. Versión errónea de *GetCorners*.

<b>Algorithm 4:</b> GET-CORNERS( <i>points</i> )	
<b>Input:</b> A series of resampled points	
<b>Output:</b> The resampled points that correspond to corners	
1	<i>corners</i> $\leftarrow \emptyset$
2	APPEND( <i>corners</i> , 0)
3	<i>W</i> $\leftarrow 3$
4	<b>for</b> <i>i</i> $\leftarrow W$ <b>to</b>   <i>points</i>   − <i>W</i> <b>do</b>
5	<i>straws</i> <sub><i>i</i></sub> $\leftarrow$ DISTANCE( <i>points</i> <sub><i>i</i>−<i>W</i></sub> , <i>points</i> <sub><i>i</i>+<i>W</i></sub> )
6	<i>t</i> $\leftarrow$ MEDIAN( <i>straws</i> ) $\times$ 0.95
7	<b>for</b> <i>i</i> $\leftarrow W$ <b>to</b>   <i>points</i>   − <i>W</i> <b>do</b>
8	<b>if</b> <i>straws</i> <sub><i>i</i></sub> < <i>t</i> <b>then</b>
9	<i>localMin</i> $\leftarrow +\infty$
10	<i>localMinIndex</i> $\leftarrow i$
11	<b>while</b> <i>i</i> <   <i>straws</i>   & <i>straws</i> <sub><i>i</i></sub> < <i>t</i> <b>do</b>
12	<b>if</b> <i>straws</i> <sub><i>i</i></sub> < <i>localMin</i> <b>then</b>
13	<i>localMin</i> $\leftarrow$ <i>straws</i> <sub><i>i</i></sub>
14	<i>localMinIndex</i> $\leftarrow i$
15	<i>i</i> $\leftarrow i + 1$
16	APPEND( <i>corners</i> , <span style="border: 1px solid red;">localMinIndex</span> )
17	APPEND( <i>corners</i> ,   <i>points</i>  )
18	<i>corners</i> $\leftarrow$ POST-PROCESS-CORNERS( <i>points</i> , <i>corners</i> , <i>straws</i> )
19	<b>return</b> <i>corners</i>

Fig. III.2. Versión correcta de *GetCorners*.

## *PostProcessCorners*

```
POST-PROCESS-CORNERS(points, corners, straws)
1  do
2    continue ← TRUE
3    for i ← 1 to |corners| do
4      c1 ← cornersi-1
5      c2 ← cornersi
6      if ¬IS-LINE(points, c1, c2) then
7        newCorner ← HALFWAY-
          CORNER(straws, c1, c2)
8        INSERT(corners, i, newCorner)
9        continue ← FALSE
10   while ¬continue
11   for i ← 1 to |corners| - 1 do
12     c1 ← cornersi-1
13     c2 ← cornersi+1
14     if IS-LINE(points, c1, c2) then
15       REMOVE(corners, i)
16       i ← i - 1
17   return corners
```

Fig. III.3. Versión errónea de *PostProcessCorners*.

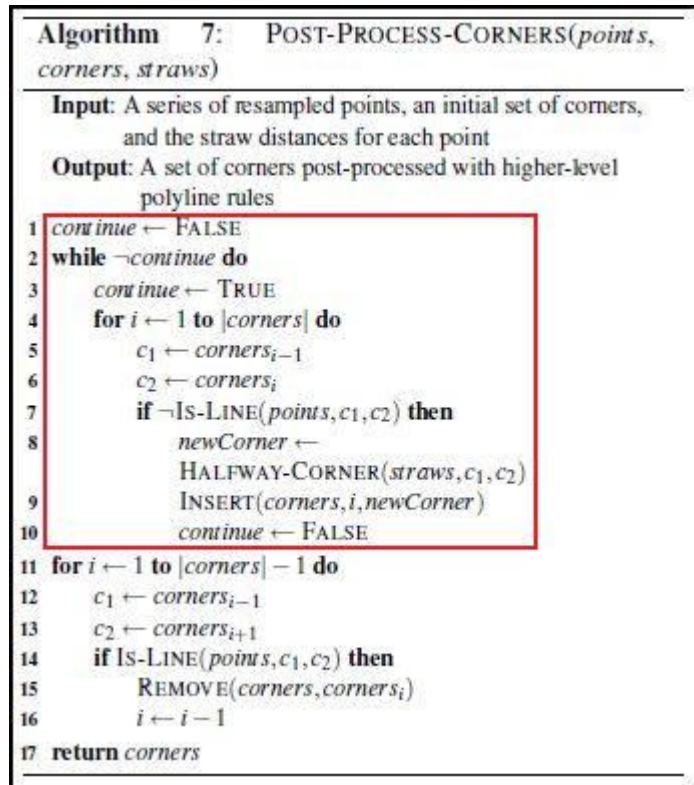


Fig. III.4. Versión correcta de *PostProcessCorners*.

### *HalfwayCorner*

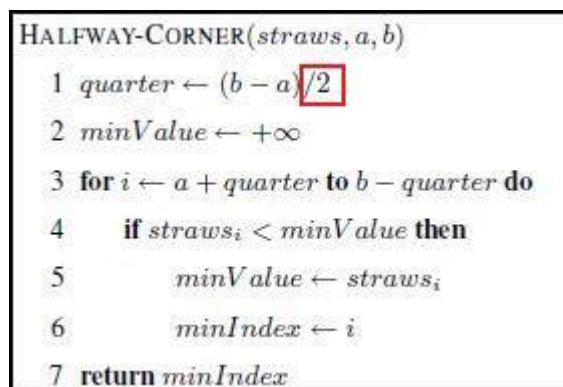


Fig. III.5. Versión errónea de *HalfwayCorner*.

**Algorithm 8:** HALFWAY-CORNER(*straws*, *a*, *b*)

**Input:** The straw distances for each point, two point indices *a* and *b*

**Output:** A possible corner between the points at *a* and *b*

```

1 quarter  $\leftarrow (b - a) / 4$ 
2 minValue  $\leftarrow +\infty$ 
3 for i  $\leftarrow a + \textit{quarter}$  to b - quarter do
4   if strawsi < minValue then
5     minValue  $\leftarrow \textit{straws}_i$ 
6     minIndex  $\leftarrow i$ 
7 return minIndex

```

Fig. III.6. Versión correcta de *HalfwayCorner*.

### *IsLine*

IS-LINE(*points*, *a*, *b*)

```

1 threshold  $\leftarrow 0.95$ 
2 distance  $\leftarrow \text{DISTANCE}(\textit{points}_a, \textit{points}_b)$ 
3 pathDistance  $\leftarrow \text{PATH-DISTANCE}(\textit{points}_a, \textit{points}_b)$ 
4 if distance / pathDistance < threshold then
5   return TRUE
6 else
7   return FALSE

```

Fig. III.7. Versión errónea de *IsLine*.

**Algorithm 9:** IS-LINE(*points*, *a*, *b*)

**Input:** A series of points and two indices, *a* and *b*

**Output:** A boolean for whether or not the stroke segment between points at *a* and *b* is a line

```

1 threshold  $\leftarrow 0.95$ 
2 distance  $\leftarrow \text{DISTANCE}(\textit{points}_a, \textit{points}_b)$ 
3 pathDistance  $\leftarrow \text{PATH-DISTANCE}(\textit{points}, a, b)$ 
4 if distance / pathDistance > threshold then
5   return TRUE
6 else
7   return FALSE

```

Fig. III.8. Versión correcta de *IsLine*.

## *PathDistance*

```
PATH-DISTANCE(points, a, b)  
1  $d \leftarrow 0$   
2 for  $i \leftarrow a$  to  $b - 1$  do  
3    $d \leftarrow d + \text{DISTANCE}(\text{points}_i, \text{points}_{i+1})$   
4 return  $d$ 
```

Fig. III.9. Versión errónea de *PathDistance*.

<b>Algorithm 5:</b> PATH-DISTANCE( <i>points</i> , <i>a</i> , <i>b</i> )
<b>Input:</b> A series of points and two indices, <i>a</i> and <i>b</i>
<b>Output:</b> The path (stroke segment) distance between the points at <i>a</i> and <i>b</i>
1 $d \leftarrow 0$
2 for $i \leftarrow a$ to $b$ do
3 $d \leftarrow d + \text{DISTANCE}(\text{points}_i, \text{points}_{i+1})$
4 return $d$

Fig. III.10. Versión correcta de *PathDistance*.