UNIVERSIDAD
REY JUAN CARLOS

INGENIERÍA TÉCNICA EN INFORMÁTICA DE SISTEMAS

Curso Académico 2009/2010

Proyecto de Fin de Carrera

Herramienta CASE para la descripción de modelos UML: EzUML

Autor: ANTONIO GARRO RETAMAL.

Tutor: ÁLVARO POLO VALDENEBRO.

Agradecimientos:

En primer lugar, me gustaría agradecer a mis padres, hermano y hermana el apoyo recibido y la fe depositada en mí durante estos años. A la paciencia de mi novia, a la que en muchas ocasiones he cambiado por horas de estudios.

Agradecer a todos los profesores de la escuela por sus enseñanzas en distintas materias que hacen que nos formen como ingenieros. En especial a Isidoro Hernán, por introducirme en el mundo de la programación.

Dar la gracias a Álvaro Polo, por ofrecerme la oportunidad de realizar el Proyecto de Fin de Carrera con él, y haber podido adquirir nuevos conocimientos y formarme como Ingeniero Técnico de Sistemas Informáticos.

También, mostrar mis agradecimientos a compañeros de trabajo y amigos que me han ayudado y han confiado en mí.

RESUMEN

EzUML es una herramienta CASE para la descripción de modelos UML utilizando un lenguaje textual como entrada y la producción de código como salida. Algunas características fundamentales son:

- Un lenguaje formal para describir diagramas, entidades, relaciones, comportamiento dinámico, patrones de diseño, etc.
- Generación de código. Traducción del lenguaje formal EzUML en código en dos de los lenguajes más populares: Java y C++.

ÍNDICE

1. INTRODUCCIÓN

El uso de herramientas CASE tradicionales hacen que muchos desarrolladores no se sientan cómodos a la hora de trabajar con ellas. Este hecho se debe a que las interfaces gráficas producen una ralentización en el trabajo. Por ejemplo, para crear un diagrama de clase con dos clases, los diseñadores deben realizar numerosas interacciones en combinación del uso del ratón y teclado. Para agregar una interfaz `IFoobar` la cual provee un método `doSomething(int a, flot b): char` y una clase `Foobar` empleando una herramienta CASE tradicional tendríamos que realizar los siguientes pasos:

1. Arrastrar y soltar la entidad interfaz del *toolbox* al lienzo. Seguidamente insertaremos el nombre de la interfaz como `IFoobar` en el cuadro de texto y pulsaremos aceptar.
2. Ahora tendremos que hacer click para agregar el método. Aparecerá una nueva ventana.
3. Ingresaremos `doSomething()` como nombre del método en el cuadro de texto que corresponda.
4. Seleccionamos `char` como valor de retorno en el *combobox*.
5. Debemos introducir el nombre del parámetro `a` y seleccionar el tipo `int` en un *combobox*.
6. Debemos introducir el nombre del parámetro `b` y seleccionar el tipo `float` en un *combobox*.
7. Hacemos click en el botón aceptar para guardar los cambios. La ventana en la que nos encontramos desaparece.
8. Volvemos a arrastrar del *toolbox* al lienzo una nueva clase. Insertamos su nombre como `Foobar` en el cuadro de texto, y seguidamente damos al botón aceptar.
9. Hacemos click en el *toolbox* para implementar la relación.
10. Pulsamos el botón izquierdo del ratón sobre `IFoobar` y arrastramos hasta `Foobar`.
11. Hacemos click con el botón izquierdo del ratón sobre `Foobar` y seleccionamos la aplicación desde el menú contextual `doSomething`.

Unos once pasos más o menos según la herramienta que se escoja, de los cuales pueden tomar varios segundos. Este ejemplo simple puede llevarnos unos dos minutos en realizarlo. Si ahora nosotros intentamos escribir el siguiente fragmento de código EzUML:

```
define class IFoobar as interface {
operations:
      doSomething (in a:  int, in b:  float): char;
};
define class Foobar {
relations:
      specializes IFoobar;
operations:
      doSomething (in a:  int, in b: float): char;
};
```

Este proyecto pretende hacer más fácil la vida del desarrollador, ya que con el anterior ejemplo se demuestra que con 10 líneas de código se puede realizar el mismo diagrama en menos tiempo. De este modo se consigue reducir tiempos, acabar con el estrés producido por la búsqueda de botones, cuadros de texto, elementos del *toolbox* o *combobox*.

1.1 HERRAMIENTAS WYSIWYG

WYSIWIG es el acrónimo de *What You See Is What You Get* ("lo que ves es lo que obtienes"). Las herramientas WYSIWYG se aplican a procesadores de texto y a otro tipo de editores con formato, los cuales permiten escribir un documento viendo el resultado final directamente. Algunas herramientas CASE de tipo WYSIWIG son Rational Rose, ArgoUML, Bouml, etc.

Ventajas:

- Es más intuitivo y fácil de manejar.
- Tiempo de aprendizaje es menor respecto al aprendizaje de un lenguaje de programación.

- Ves el resultado final de los diagramas UML.

Inconvenientes:

- Aumento del tiempo de trabajo por la labor de uso de ratón y teclado en busca de los elementos requeridos.
- No es portable. Depende de un editor WYSIWYG concreto.
- Corrección de código al ser traducido en otro lenguaje por el editor WYSIWYG. Esto significa que el responsable directo de la traducción es el editor, aunque las directivas hubieran sido impartidas por el diseñador.

La motivación de EzUML ha venido a consecuencia de los inconvenientes producidos por las herramientas CASE de tipo WYSIWYG. Las ventajas producidas que ofrece una representación textual son numerosas. Vamos a discutir algunas de ellas, como son:

- El ahorro de tiempo respecto a la búsqueda de elementos del editor. Con lo que nos permite una mayor concentración en la información del modelo para poder transmitirlo en el diseño final.
- Es portable, debido a que un lenguaje de programación puede ser escrito en cualquier editor de texto.
- La corrección de código no debe ser una preocupación, la traducción de código está basada en la responsabilidad del usuario al escribir su código.
- Los cambios producidos del código escrito en el editor de texto pueden gestionarse mediante herramientas de control de versiones. En cambio los métodos de representación binarios no se pueden gestionar, porque es posible que se pueda dar una combinación de bytes que coincida con alguna de las variables que utiliza el propio gestor, de tal forma, que se produciría una modificación del contenido y se corrompería. Además, el sistema de cálculo que se utiliza para obtener diferencias no está diseñado para trabajar con información binaria.

1.1.2 ARQUITECTURA DIRIGIDA POR MODELOS

MDA, *Model Driven Architecture*, se trata de un framework de desarrollo de software que define una nueva forma de construir software en la que se usan modelos del sistema, a distintos niveles de abstracción, para guiar todo el proceso de desarrollo, desde el análisis y diseño hasta el mantenimiento del sistema y su integración con futuros sistemas.

MDA pretende separar, por un lado, la especificación de las operaciones y datos de un sistema, y por el otro, los detalles de la plataforma en la que el sistema será construido.

Los principales objetivos de MDA son mejorar:

- La productividad.
- La portabilidad.
- La interoperabilidad.
- La reutilización de sistemas.

El proceso de desarrollo de software con MDA es:

- Construcción de un Modelo Independiente de la Plataforma (PIM), un modelo de alto nivel independiente de cualquier tecnología o plataforma.
- Transformación del modelo anterior en Modelos Específicos de Plataforma (PSM), los cuales describen el sistema de acuerdo una tecnología de implementación determinada.
- Generación de código a partir de PSM.

Por eso los modelos que estén escritos en un lenguaje bien definido, suponen tener asociadas una sintaxis y una semántica bien definidas, permitiendo la interpretación automática por parte de transformadores o compiladores de modelos, siendo de gran importancia para MDA.

1.2 OTRAS APROXIMACIONES

UML (Unified Modeling Language), es un lenguaje para la visualización, especificación, construcción, y documentación de los artefactos de los sistemas de software.

Otras aproximaciones, es la serialización de modelos de UML. La serialización de un objeto consiste en obtener una secuencia de bytes que represente el estado del objeto. Esta secuencia se puede utilizar para enviarse a través de la red o para guardarla en un fichero para su uso posterior. En definitiva, el objetivo final es recomponer el objeto original.

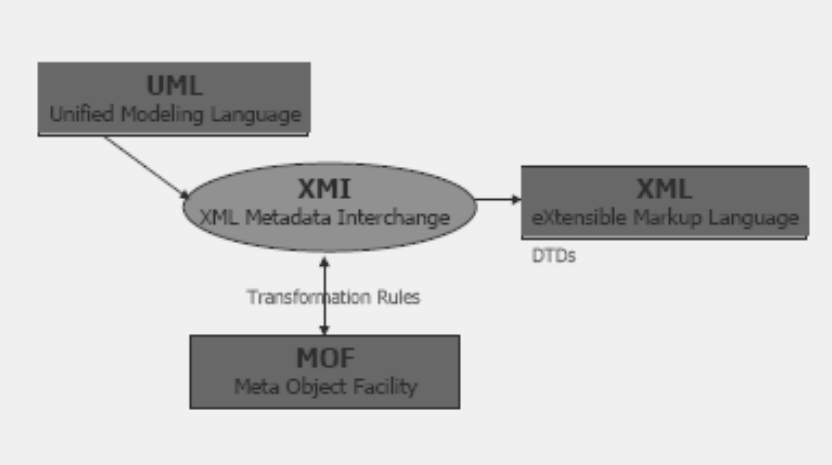
XMI es un estándar del Object Management Group (OMG) que sirve para el intercambio de información vía XML. Puede ser utiliza por todos los metadatos de los metamodelos que estén expresados en MOF. Su uso más frecuente es el intercambio de modelos UML, es decir, para la serialización de metamodelos.

Desde la perspectiva de modelado de OMG, los datos se dividieron en modelos abstractos y modelos concretos. Los modelos abstractos representan la información semántica, mientras que los modelos concretos representan diagramas visuales. Los modelos abstractos son instancias de lenguaje de modelado basado en MOF (Meta Object Facility) como puede ser UML. Para los diagramas se utiliza el XMI.

El propósito de XMI es facilitar el intercambio de los metadatos entre las herramientas de modelado UML y los repositorios de metadatos MOF en entornos heterogéneos.

MOF es un estándar que define un conjunto de constructores que pueden ser usados para definir lenguajes de modelado.

En conclusión, XMI es un formato de intercambio de metadatos independientes de cualquier plataforma. Se crea una nueva forma de transferir metadatos entre repositorios MOF. La serialización que se aplicó al principio fue sobre modelos y metamodelos de UML, más tarde se aplicaría a modelos y metamodelos MOF en XML. Además XMI, proporciona interoperabilidad entre herramientas.



La vía que utiliza XMI es el lenguaje XML (Extensible Markup Language) que permite describir y organizar la información de manera que resulte fácilmente comprensible tanto para las personas cómo para las máquinas. XML es un metalenguaje que está pensado para la interacción entre máquinas.

Tanto XMI como XML son lenguajes de etiquetado, con lo que no llegan aproximarse al lenguaje natural. A las personas, normalmente, nos gustan los lenguajes de programación de alto nivel, que sean lo más parecido al lenguaje humano. De esta forma el programador tendrá mayores facilidades a la hora de programar. Esta ventaja se junta con la de un aprendizaje más rápido de la sintaxis del lenguaje que el de otro tipo de lenguajes.

1.3 PROCESO DE DESARROLLO

El desarrollo del proyecto ha seguido un proceso estructurado siguiendo una metodología con la ayuda de Trac.

Trac es un gestor de proyectos de software. Permite enlazar información entre una base de datos de errores de software, un sistema de control de versiones (Subversion), y la gestión documental colaborativa basada en wiki.

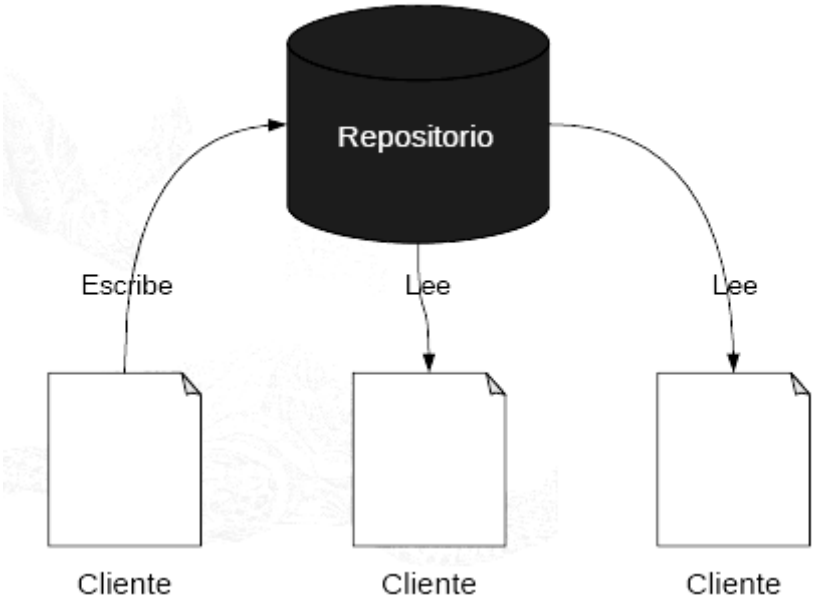
Algunas características más importantes de Trac son:

- **Gestión de tareas e incidencias (sistema de tickets):** tiene como finalidad gestionar las tareas del proyecto, reportes de errores y asuntos

de soporte técnico de software. Normalmente un ticket es asignado a una persona que debe resolverlo o reasignarlo. Los tickets se pueden editar, comentar, asignar, dar una prioridad y ser discutidos.

- **Roadmap:** se usa para comentar el estado actual en el que se encuentra el desarrollo de un software. Con está práctica se consigue dar una visión de hacia adónde apunta a llegar un software. Además contiene una vista sobre el sistema de tickets, el cual ayuda a planificar y gestionar el desarrollo de un proyecto, siguiendo una secuencia de actividades.
- **Gestión documental colaborativa basada en wiki:** en Trac se incluye un motor interno de wiki, usado para texto y documentación de todo el sistema. En las páginas de la wiki, de los tickets y de los mensajes de log se utiliza el formato WikiFormatting, permitiendo dar al texto un formato y enlaces a todos los módulos de Trac. Su objetivo principal es la sencillez en la edición de texto y alentar a las personas a la aportación de contenido de texto para los proyectos.
- **Timeline:** es una vista histórica del proyecto detallada en un informe. Se listan todos los eventos en orden cronológico con una descripción de cada uno de éstos con la persona responsable del cambio realizado. También muestra los cambios producidos en las páginas de la wiki, así como la creación, resolución de tickets, cambios en el código y las distintas etapas que han sido completadas.
- **Browse Source:** su uso es para poder navegar por el repositorio de código fuente del proyecto. De esta forma podemos observar los distintos directorios en las distintas etapas de desarrollo, así como el contenido de los archivos de cada uno de los directorios. Además podemos acceder a las distintas revisiones de cada archivo.
- **Sistema de control de versiones**: Trac usa Subversión, es un sistema de control de versiones centralizado para compartir información. Subversión gestiona archivos y directorios, y los cambios producidos a través del tiempo, pudiendo volver a recrear un proyecto desde cualquier momento en su historia. La información guardada es almacenada en un repositorio en forma de árbol con una jerarquía de directorios y archivos.

En la siguiente figura se muestra las acciones que puede realizar los clientes sobre los archivos del repositorio.



Subversion recuerda todos los cambios que se realizan en el repositorio, así como los cambios producidos en el árbol de directorios. Además provee la habilidad de leer estados anteriores del sistema de archivos.

El objetivo de un sistema de control de versiones es el de permitir editar de forma colaborativa y compartir la información.

1.4 COMUNIDAD MORFEO

Es una comunidad de desarrollo, en el que participan multitud de entidades, de las cuales TID (Telefónica Investigación y Desarrollo) la encabeza. Cuyo objetivo principal es abaratar costes en el desarrollo de plataformas de software distribuidas. Se pretende seguir un modelo de negocio, que es el modelo de software libre. Con este modelo de desarrollo de software se garantiza que el código de los programas sea abierto, es decir, sea gratuitamente accesible al público. De esta forma se consigue que se fomente la participación en el desarrollo de aplicaciones, y así tener menores costes tanto de mantenimiento como desarrollo.

El desarrollo de EzUML se encuentra dentro de la Comunidad Morfeo, siguiendo así una "metodología ágil" (http://trac.morfeo-project.org/ezuml/wiki).

Los sistemas de control de versiones ayudan a controlar las distintas versiones del código fuente, así como las distintas especializaciones realizadas. Un sistema de control de versiones nos debe proporcionar:

- Un mecanismo de almacenaje para los elementos que se gestionen.
- Poder realizar los distintos cambios sobre los distintos elementos que se encuentren almacenados.
- Debe tener un registro histórico de las acciones realizadas por cada elemento almacenado. Así como poder volver a versiones anteriores de los elementos introducido

  Normalmente, se anotan los distintos cambios que se realizan entre dos versiones en un informe, para así facilitar mejor su seguimiento y cambios producidos.

- Los sistemas de control de versiones deben de disponer de un repositorio, que será donde se gestione el conjunto de información del sistema y contendrá el historial de todos los elementos gestionados.

EzUML sigue el modelo de desarrollo de software de la comunidad, el modelo de software libre, de modo que tiene un cierto interés industrial. EzUML tiene su propio *website* (http://trac.morfeo-project.org/ezuml/wiki), sus herramientas colaborativas, el sistema de reporte de *bugs*, etc., y toda su documentación escrita en inglés, signo evidente de la importancia de éste proyecto. El modelo de software libre garantiza que el código de los programas sea accesible al público y totalmente gratuito, obteniendo así, una mayor participación tanto en el desarrollo de aplicaciones como a su uso por parte del público.

2. ANÁLISIS

2.1 ANÁLISIS DEL PROBLEMA

Tenemos la necesidad de expresar UML en forma de texto, debido a los inconvenientes producidos por las herramientas CASE de tipo WYSIWYG, comentados en el capítulo de HERRAMIENTAS WYSIWYG.

Para poder expresar UML en texto lo realizamos primero en un lenguaje formal, una gramática independiente del contexto.

Las gramáticas independientes del contexto (GIC) permiten describir los lenguajes de programación. Además, estas gramáticas son bastante simples con lo que nos permite diseñar eficientes algoritmos de análisis sintácticos, que para una cadena de caracteres dada determinen como puede ser generada desde la gramática.

2.2 DISEÑO DEL LENGUAJE

La especificación del lenguaje EzUML se ha realizado en notación BNF (Backus-Naur Form).

Una GIC está formada por una 4-tupla:

1. Un conjunto de finito de símbolos terminales.
2. Un conjunto finito de símbolos no terminales.
3. Un conjunto finito de producciones.
4. Símbolo inicial.

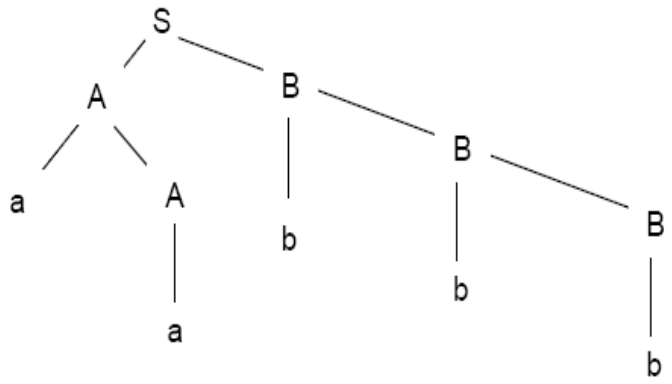
Donde las producciones son de la siguiente forma: `A□w`, donde `A` es un no terminal y `w` es una cadena que puede contener terminales y no terminales. A continuación mostramos un ejemplo de GIC.

```
S→ AB

A→ aA | a

B→ bB | b
```

Las GIC se pueden derivar y representar de forma gráfica en un árbol de derivación.

Un árbol de derivación nos permite mostrar gráficamente cómo se puede derivar cualquier cadena de un lenguaje a partir del símbolo inicial de una gramática que genera ese lenguaje

Seguidamente realizaremos del anterior ejemplo su árbol de derivación para la cadena `aabbb`.



Los símbolos terminales se encuentran entre paréntesis angulares ("<", ">"), y las producciones tienen el siguiente formato:

```
<lado_izquierdo> ::= <lado_derecho>
```

Donde `<lado_izquierdo>` es un no terminal, y `<lado_derecho>` es una expresión que contiene secuencias de símbolos, tanto terminales como no terminales, y pueden estar separados por la barra vertical "`|`", que indica una opción. Aquellos símbolos que no aparezcan en lado izquierdo, pero si en lado derecho son los terminales.

A continuación se mostrará la especificación del lenguaje EzUML, cuya función es la de traducir la expresividad de UML en forma de declaraciones textuales en vez de diagramas gráficos.

EzUML está constituido por los siguientes elementos básicos:

- La entrada de EzUML está formada por una lista de sentencias.

- Una sentencia puede ser una declaración, una acción de generación de código, una acción de dibujo, o una elección de lenguaje predefinido.
- Una declaración podrá ser una declaración de estereotipo, una declaración de clase, una declaración del paquete o una declaración de vista de clase.

```
<ezuml input> ::= <statement list>
<statement list> ::= <statement>  ; <statement list> | λ
<statement> ::= <declaration> | <drawing action> | <code
generation action> | <lang setup>
<declaration> ::= <stereotype declaration> | <class declaration>
| <package declaration> | <class view declaration>
```

La vista de clases de UML se utiliza para proporcionar vistas estáticas y dinámicas de las entidades que forman un sistema de software. Referente al diseño orientado objeto, estas entidades son los paquetes, clases, los objetos y sus relaciones.

Para la declaración de estereotipos de UML utilizamos la siguiente sintaxis:

```
<stereotype declaration> ::= define stereotype <identifier>
<stereotype application>
<stereotype application> ::=  applicable to <entity list> | λ
<entity list> ::= <entity> | <entity> , <entity list>
<entity> ::= package | class | interface | attribute | operation
| relation
```

Esta sentencia declara un nuevo estereotipo dado un nombre como `<identifier>` `<stereotype application>` permite definir un conjunto limitado de entidades a las que se le puede aplicar ese estereotipo. Si el estereotipo no se aplica a ninguna entidad, entonces este podrá aplicarse a cualquier entidad.

En el siguiente cuadro se muestra la aplicación correspondiente a cada entidad.

Entidad	Descripción
`package`	Aplicado a un paquete
`class`	Aplicado a una clase
`interface`	Aplicado a una interfaz
`attribute`	Aplicado a un atributo de clase
`operation`	Aplicado a una operación de clase
`relation`	Aplicado a clases

Algunos ejemplos de uso a la hora de la declaración de estereotipo son:

```
define stereotype Readable;
define stereotype RemoteObject applicable to class, interface;
define stereotype DatabaseMapped applicable to attribute;
```

Para la declaración de clases se utiliza la siguiente sintaxis:

```
<class declaration> ::= define class <identifier> <applied
stereotypes> { <class sections> }
<applied stereotypes> ::= as <identifier list> | λ
<identifier list> ::= <identifier> | <identifier> , <identifier
list>
<class sections> ::= <class section> <class sections> | λ
<class section> ::= <attributes section> | <operations section>
| <relations section>
```

En la anterior declaración se define una nueva clase identificada por `<identifier>`.

El no terminal `<applied stereotypes>` indica que estereotipos se aplican a la clase y el otro no terminal `<class sections>`, se utiliza para definir los atributos, las operaciones y las relaciones de la clase.

Para la definición de los atributos de una clase se dispone de la siguiente forma:

```
<attributes section> ::= attributes : <attributes declarations>
<attribute declarations> ::= <attribute declaration> ;
<attribute declarations> | λ
<attribute declaration> ::= <scope> <static modifier>
<identifier> <applied type> <applied stereotypes>
```

```
<scope> ::= public | private | protected | package | λ
<static modifier> ::= static | λ
<applied type> ::= : <identifier> | λ
```

La sección de la declaración de atributos dentro de una clase comienza con `attributes:`, seguido de un conjunto de declaraciones de atributos. Cada uno de éstos está formado por los siguientes elementos que se mostrarán a continuación.

- `<scope>` Indica el ámbito del atributo. Si no se da, por defecto se aplica `public`.
- `<static modifier>` Indica si esta el atributo es una extensión de la clase o una instancia de la clase.
- `<applied type>` Indica el tipo del atributo. Si no se da, entonces no tiene asociado ningún tipo.
- `<applied stereotypes>` Indica que estereotipos se aplican al atributo. Para ver su sintaxis con más detalle se muestra en la declaración de clase.

Un ejemplo de la sintaxis de definición de atributos podría ser:

```
define class Person {
attributes:
   birthplace;
   package static adultage: integer;
   private name: string;
   public children: Person as Vector;
};
```

Las operaciones de una clase las definimos de la siguiente forma:

```
<operation section> ::= operations: <operation declarations>
<operation declarations> ::= <operation declaration> <operation
declarations> | λ
<operation declaration> ::= <scope> <static modifier>
<identifier> ( <argument declarations> ) <applied type> <applied
stereotypes> ;
<argument declarations> ::= <argument declaration list> | λ
```

```
<argument declaration list> ::= <argument declaration> |
<argument declaration> , <argument declaration list>
<argument declaration> ::= <argument mode> <identifier> <applied
type>
<argument mode> ::= in | out | inout | λ
```

Para la anterior sección, el ejemplo de operación sería:

```
define class Person {
operations:
      public sendMail(in msg: string): void;
      public hire(inout job: Job): void;
      protected static lifespan(): uint;
      kill(in method);
};
```

Las relaciones de clase se dan entre la clase dada y otras clases. A continuación mostraremos la sintaxis para las relaciones de clase.

```
<relations section> ::= relations : <relation list>
<relation list> ::= <relation> ; <relation list> | λ
<relation> ::= <unidir association> | <bidir association> |
<specialization>
<unidir association> ::= <association type> with <multiplicity>
<identifier>
<bidir association> ::= <multiplicity> bidir <association type>
with <multiplicity> <identifier>
<association type> ::= association | aggregation | composition
<specialization> ::= specializes <identifier>
```

Un ejemplo de relación de clase podría ser el siguiente:

```
define class Person {
  relations:
    specializes NSObject;
    [0..1] bidir aggregation with [0..1] NSApplication;
    composition with [0..1] NSSet;
```

```
};
```

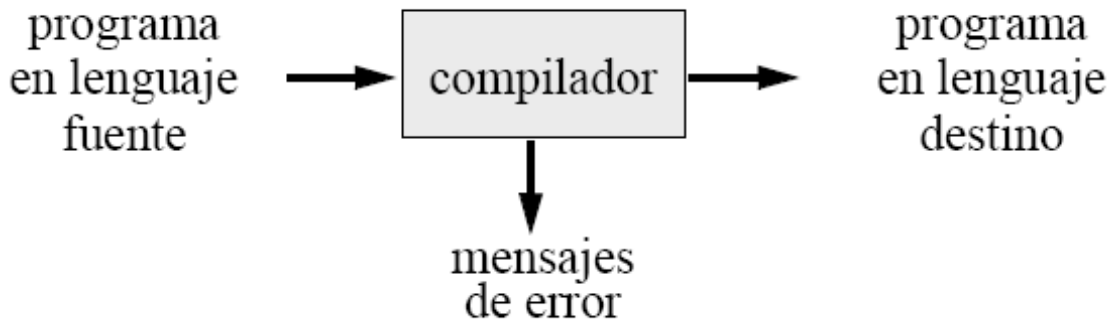
Las sentencias de una acción de generación de código o una elección de lenguaje predefinido se declaran de la siguiente manera:

```
<code generation action> ::=  generate code <language>
<language setup> ::= use <language>
<language> ::= java | cpp
```

La sentencia `drawing action` no se ha declarado todavía. Esta declaración se realizará en un futuro para así conseguir la generación de diagramas de UML.

2.3 DISEÑO DEL COMPILADOR

Un compilador es un programa que lee un programa escrito en un lenguaje, y lo traduce a un programa equivalente en otro lenguaje.



Es de gran importancia mencionar, que durante el proceso de traducción, el compilador debe informar al usuario de los errores que contenga el programa fuente.

Los primeros compiladores aparecieron en la década de 1950. Éstos eran costosos de implementar, pero hoy en día existen técnicas sistemáticas para la construcción de compiladores.

En la actualidad, existe una gran variedad de compiladores para los distintos lenguajes fuente y lenguajes destino.

2.3.1 MODELO DE ANÁLISIS Y SÍNTESIS DE LA COMPILACIÓN

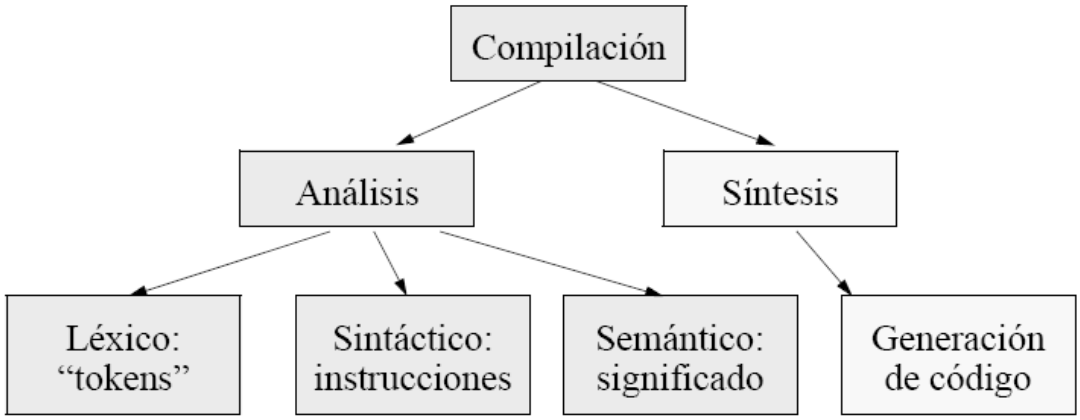
En el proceso de compilación hay dos etapas:

Etapa de análisis: divide el programa fuente en sus elementos constituyentes y crea una representación intermedia del mismo. Hay tres tipos, el análisis léxico que separa cada componente del programa, "token"; el análisis sintáctico, el cual separa cada sentencia del lenguaje, agrupando varios componentes léxicos o "tokens"; y el análisis semántico, donde se encarga de detectar la validez semántica de las sentencias aceptadas por el analizador sintáctico y recoger información para las siguientes fases del compilador.

Etapa de síntesis: construye el programa objeto o destino a partir de una descripción en un lenguaje de representación intermedia.

En la etapa de análisis se determinan las operaciones, las cuales implican al programa fuente y se registran en una estructura de tipo árbol.

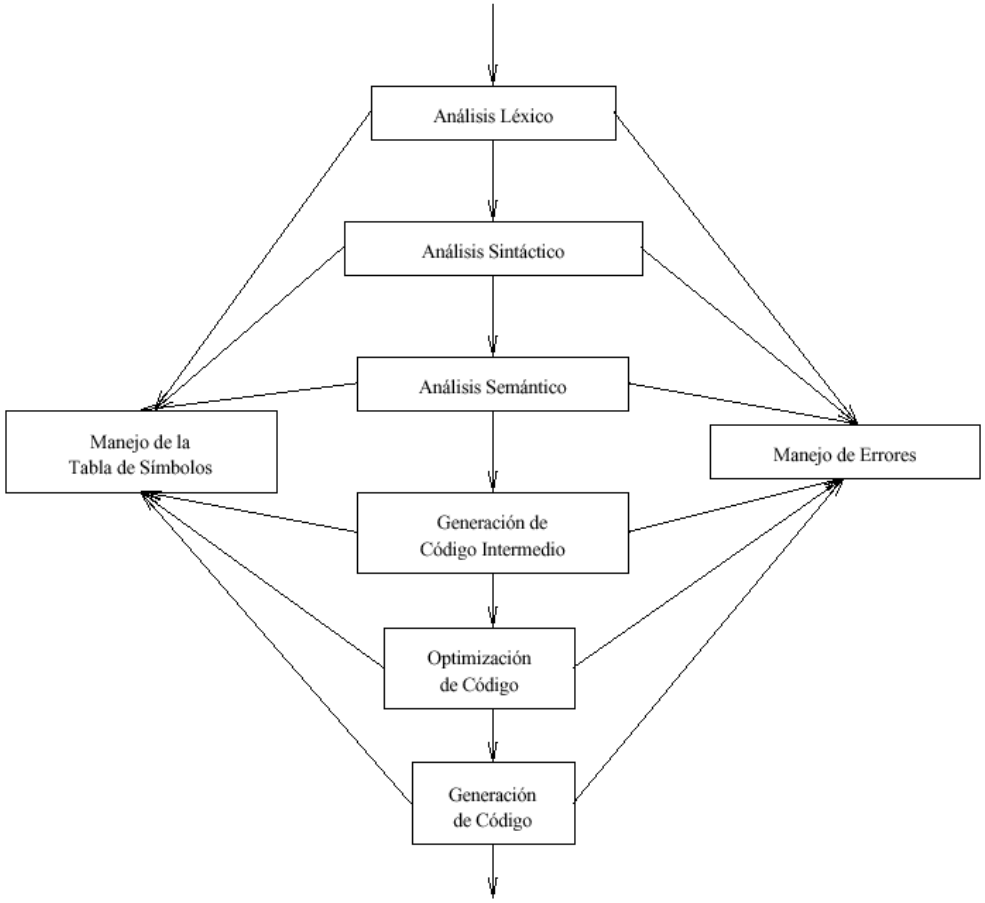
En el siguiente esquema se puede ver las distintas etapas de la compilación:



El proceso de generación de código se realizará en dos de los lenguajes más populares, como son Java y C++.

2.3.2 FASES DE UN COMPILADOR

Un compilador realiza su trabajo en fases. Cada una de estas fases realiza una transformación que será de utilidad para la siguiente fase, donde se puede ver en la siguiente figura.



En la imagen anterior, podemos apreciar que las tres primeras fases se corresponden con el análisis de un programa fuente y las tres últimas fases con la síntesis de un programa objeto. Por otro lado tenemos el manejo de la tabla de símbolos y de errores, que hará uso cada una de las fases.

La tabla de símbolos tiene como misión principal guardar en un área de memoria los distintos identificadores. También la tabla de símbolos puede ser inicializada con identificadores de un lenguaje específico. A medida que se van conociendo nuevos identificadores, se agregan a los registros de la tabla de símbolos.

Para la detección de errores en cada fase de compilación, pueden encontrarse errores y éstos deben ser manejados. De tal forma, que puedan seguir las demás fases sin que se vea interrumpido el proceso y así seguir identificando otros errores posibles.

Normalmente, el análisis semántico y sintáctico, detectan la mayoría de los errores. En el análisis semántico el compilador intentará detectar aquellas estructuras que tengan una estructura sintáctica correcta pero no tiene significado para la operación que se encuentre implicada. En nuestro caso, al ser un traductor, algunas de las fases de un compilador serán omitidas, como son la generación de código intermedio y la optimización de código.

3. DISEÑO

En el capítulo de diseño trataremos los temas de los bloques funcionales de la arquitectura básica del sistema, las entidades que conforman el sistema y el modelo dinámico del traductor.

3.1 ARQUITECTURA BÁSICA DEL SISTEMA

En la arquitectura básica del sistema tenemos que tener en cuenta los siguientes bloques funcionales:

- **Árbol de entidades.**

  Éste árbol es formado por las entidades: `class`, `attribute`, `operation`, `relation`, `stereotype`, así como otras que conforman el sistema. Cada una de estas entidades nos ayudará a describir el lenguaje EzUML. En el próximo apartado entraremos en más profundidad.

- **Tabla de símbolos.**

  Con esta tabla intentamos almacenar la información que necesitaremos sobre los identificadores de las entidades para así obtener toda la información relacionada a éstas. Dependiendo del lenguaje que utilicemos, se inicializará la tabla de símbolos con una serie de palabras reservadas respecto a los lenguajes Java y C+

+. Tenemos que tener en cuenta, antes de la inicialización, la tabla debe estar completamente vacía, para así no nos reporte errores de duplicación de nombres de entidades.

También nos apoyaremos en la tabla de símbolos para consultar los distintos identificadores de entidades y así servir de apoyo al análisis semántico.

- **Consola de errores.**

Durante la compilación de nuestro programa EzUML necesitamos que nos reporte información sobre si ha habido algún error o no. Si se produce algún error en el análisis, éste se notificará al exterior por medio de mensajes para así obtener una mayor información del error producido.

Podemos encontrarnos con errores sintácticos y errores semánticos, los cuales serán manejados por medio de una recuperación de error.

En el apartado del modelo dinámico del traductor se explicará con más detalle la recuperación de errores en el análisis sintáctico.

3.2 DISEÑO DE LAS ENTIDADES

En este apartado abordaremos las distintas entidades que encarnan el comportamiento del sistema y explicaremos la finalidad de cada una de ellas.

La clase `Entity`, es la clase base de las demás entidades. Se utiliza para saber el tipo de entidad que nos llega a través de las clases extendidas y el nombre o identificador de la entidad. La clase `Entity` es usada para rellenar la tabla de símbolos.

Podemos ver la declaración de la clase `Entity` y sus funciones, como el método constructor `Entity()`, que registra el nombre y el tipo de entidad, o la función `getEntityType()`, que nos devuelve el tipo de entidad que es.

```
define class Entity
{
attributes:
```

```
      private name string;
      private entityType EntityType;

operations:
      public Entity(in nam: string, in type: EntityType);
      public setName(in nam: string): bool;
      public getName(): string;
      public getEntityType(): EntityType;
};
```

La clase `ClassStatement`, es utilizada para saber el tipo de sentencia que nos llega, es decir, si es una clase, o un estereotipo, o una acción de generación de código o una acción de dibujado, a través de la función `getStatementType()`.

```
define class ClassStatement
{
      operations:
           public ClassStatement(in name: string, in entType:
EntityType);
          public getStatementType(): ClassStatementType;

      relations:
           specializes Entity;
};
```

En la anterior sección, hay una función virtual (`getStatementType()`), esto es un concepto de polimorfismo de la programación orientada a objetos.

Hacemos uso del polimorfismo cuando deseamos definir una abstracción para englobar objetos de distintos tipos.

Las clases `GenerationCode`, `Drawing` y `UseLang` se utilizan para saber el tipo de sentencia que nos llega por medio de la función virtual, y así poder saber si es una acción de generación de código, de pintado o el tipo lenguaje usado. Con la utilización de métodos virtuales podemos preguntar al método virtual de la clase base y éste nos devolverá el tipo al que pertenece.

```
define class GenerationCode
{
     operations:
          public GenerationCode(in name: string);
          public getStatementType(): ClassStatementType;

     relations:
          specializes ClassStatement;
};

define class Drawing
{
     operations:
        public Drawing(in name: string);
        public getStatementType(): ClassStatementType;

     relations:
          specializes ClassStatement;
};

define class UseLang
{
     operations:
          public UseLang(in name: string);
          public getStatementType():ClassStatementType;

     relations:
          specializes ClassStatement;
};
```

La clase `Stereotype`, nos permite declarar estereotipos de UML. La función `addApplicableEntities` sirve para almacenar los distintos tipos de entidades que se le puede aplicar a un estereotipo. Podemos observar la declaración de la función virtual `getStatementType()`, que se detalló en la anterior sección, por los motivos de polimorfismo. La función o`perador isEqual()`, nos compara si dos estereotipos son iguales.

```
define class Stereotype
{
      attributes:
            private applicableEntity: EntityType as list;

      operations:
            public Stereotype(in name: string);
            public addApplicableEntities(in ent: EntityType as
list): void;
            public isEqual (in s: Stereotype): bool;
            public getStatementType():ClassStatementType;

      relations:
           specializes ClassStatement;
};
```

La clase `ClassMember`, esta clase nos vemos obligados a declararla por los distintos miembros que compone una clase de UML, como son los atributos, operaciones y relaciones. De esta forma podemos realizar polimorfismo de funciones y diferenciar por medio del método virtual `getMemberType()` cada uno de los miembros. La función `setScope()` asigna el ámbito del atributo y `getScope()` nos devuelve el ámbito del atributo (`public`, `private`, `packaged`, `protected`). La función `setStaticModifier` asigna por medio de un booleano si el atributo es una instancia de la clase, y la función `hasStaticModifier()` nos indica si tiene o no una instancia de una clase.

```
define class ClassMember
{
      attributes:
            private scope: ScopeType;
            private staticModifier: bool;

      operations:
            public ClassMember(in name: string, in entType:
EntityType);
```

```
            public setScope(in s: ScopeType): void;
            public getMemberType(): ClassMemberType;
            public setStaticModifier(in s: bool): void;
            public getScope(): ScopeType;
            public hasStaticModifier():bool;

      relations:
            specializes Entity;
};
```

Para la clase `Attribute`, contiene dos atributos privados, una lista de estereotipos y un puntero al tipo `Class`, que será el tipo del atributo de la clase UML en cuestión. A continuación se muestran las cabeceras de las funciones utilizadas para esta clase.

```
define class Attribute
{
      attributes:
            private stereotypes: Stereotype as list;
            private returnType: Class;

      operations:
            public Attribute(in name: string);
            public setReturnType(in c: Class): void;
            public getReturnType(): Class;
            public setStereotype(in s: Stereotype): void;
            public hasReturnType(in s: Class): bool;
            public getStereotypes(): Stereotype as list;
            public isEqual (in a: Attribute): bool;
            public getMemberType():ClassMemberType;

      relations:
            specializes ClassMember;
};
```

La clase `Operation`, se utiliza para la declaración de operaciones en las clases de UML. Los atributos que tiene esta clase son una lista de `ArgumentType`, es decir, una

lista de los argumentos que contiene una operación. Tenemos también una lista de estereotipos y el tipo que de la operación, `returnType`, que es un puntero a `Class`.

```
define class Operation
{
      attributes:
            private arguments: ArgumentType as list;
            private returnType: Class;
            private stereotypes: Stereotype as list;

      operations:
            public Operation(in name: string);
            public setReturnType(in c: Class): void;
            public hasReturnType(in s: Class): bool;
            public getReturnType(): Class;
            public setStereotype(in s: Stereotype): void;
            public getStereotypes(): Stereotype as list;
            public setArguments(in arg: ArgumentType as list):
void;
            public getArguments(): ArgumentType as list;
            public isEqual (in o: Operation): bool;
            public hasNameRepeated(in s: string): bool;
            public getMemberType(): ClassMemberType;

      relations:
            specializes ClassMember;
};
```

En la siguiente sección, se muestra la clase `Class`, designa a las clases de UML. Sus funciones principales son: añadir un estereotipo, `addStereotype()`; añadir una operación, `addOperation()`; añadir un atributo, `addAttribute()`; y añadir una relación a la clase, `addRelation()`. Otros métodos de gran importancia que nos ayudarán a la generación de código son `getAllStereotypes()`, `getAllOperations()`, `getAllAttributes()`, `getAllRelations()`.

```
define class Class
```

```
{
      attributes:
            private stereotypes: Stereotype as list;
            private operations: Operation as list;
            private relations: Relation as list;
            private attributes: Attribute as list;

      operations:
           public Class(in name: string, in entType: EntityType);
            public addStereotype(in s: Stereotype): void;
            public hasStereotype(in s: Stereotype): bool;
            public removeStereotype(in s: Stereotype): void;
            public getAllStereotypes(): Stereotype as list;
            public addOperation(in o: Operation): void;
            public hasOperation(in o: Operation): bool;
            public removeOperation(in o: Operation): void;
            public getAllOperations(): Operation as list;
            public addAttribute(in a: Attribute a): void;
            public hasAttribute(in a: Attribute): bool;
            public removeAttribute(const Attribute &a): void;
            public getAllAttributes(): Attribute as list;
            public isEqual (in c: Class): bool;
            public addRelation(in r: Relation): void;
            public removeRelation(in r: Relation): void;
            public getAllRelations(): Relation as list;
            public getStatementType(): ClassStatementType;

      relations:
            specializes ClassStatement;
};
```

Para las relaciones de clase UML tendremos la clase `Relation`, como clase base y sus derivadas, `Specialization`, `Unidirectional` y `Bidirectional`. Como se puede observar en la siguiente sección, hay dos funciones virtuales, `getAssociationType()` y `getMultiplicity()`. Estas funciones son necesarias

para saber el tipo de relación (especialización, asociación unidireccional, asociación bidireccional) y la multiplicidad de ésta.

```
define class Relation
{
      attributes:
            private type: RelationType;
            private typeA: AssociationType;
            private multiplicityType: MultiplicityType as list;

      operations:
            public Relation(in name: string, in typ:
RelationType);
            public setRelationClass(in c1: Class, in c2: Class):
void;
            public setRelationType(in t: RelationType): void;
            public getRelationType():RelationType;
            public isEqual (in r: Relation): bool;
            public getMemberType(): ClassMemberType;
            public getAssociationType(): AssociationType;
            public getMultiplicity(): MultiplicityType as list;
            public ~Relation();

      relations:
           specializes ClassMember;
};

define class Specialization
{
      attributes:
            private type: RelationType;
            private typeA: AssociationType;
            private multiplicity: MultiplicityType as list;

      operations:
```

```
            public Specialization(in name: string);
            public getMultiplicity(): MultiplicityType as list;
            getAssociationType(): AssociationType;

      relations:
            specializes Relation;
};

define class UniDirectional
{
      attributes:
            private typeA: AssociationType;
            private multiplicity: MultiplicityType as list;

      operations:
            public UniDirectional(in name: string, in typA:
AssociationType);
            public setMultiplicity(in m: MultiplicityType): void;
            public getMultiplicity(): MultiplicityType as list;
            public getAssociationType():AssociationType;

      relations:
            specializes Relation;
};

define class BiDirectional
{
      attributes::
            private typeA: AssociationType;
            private multiplicity: MultiplicityType as list;

      operations:
            public BiDirectional(in name: string, in typA:
AssociationType);
            public setMultiplicityA(MultiplicityType m): void;
```

```
            public setMultiplicityB(in m: MultiplicityType):
void;
            public getMultiplicity(): MultiplicityType as list;
            public getAssociationType(): AssociationType;

      relations:
            specializes Relation;
};
```

A continuación, presentamos las clases que nos ayudan a la generación de código Java o C++. Pero antes tenemos que tener en cuenta unos factores en las relaciones de clase UML.

Al generar código, todas las asociaciones que van de A a B, las cuales no son de herencia, se han de resolver de la forma que A debe mantener un atributo del tipo referencia a B, es decir, en C++ es un puntero y en Java una referencia a objeto. Además, si la relación es unidireccional, A debe mantener la referencia a B, pero B no la mantiene hacia A, es decir, se puede navegar de A a B pero no viceversa.

Para el caso de la multiplicidad, en las asociaciones, existen dos multiplicidades, una en cada extremo de la relación. Esta multiplicidad nos indica cuantos objetos de la otra clase están asociados. Si es `0..1`, la asociación sólo puede ser con 0 o 1 objeto. Si es `*`, la asociación puede ser con múltiples objetos. En el caso de que la relación sea bidireccional, se puede especificar la multiplicidad en la dirección opuesta también. Además, si no se especifica nada, la multiplicidad se considera `0..1`.

Cuando la multiplicidad es `0..1`, se representa por medio de una referencia al objeto de la clase asociada. Si es mayor de 1 la multiplicidad, no es suficiente una referencia al objeto, para este caso, emplearemos una lista enlazada para representa el conjunto de asociaciones.

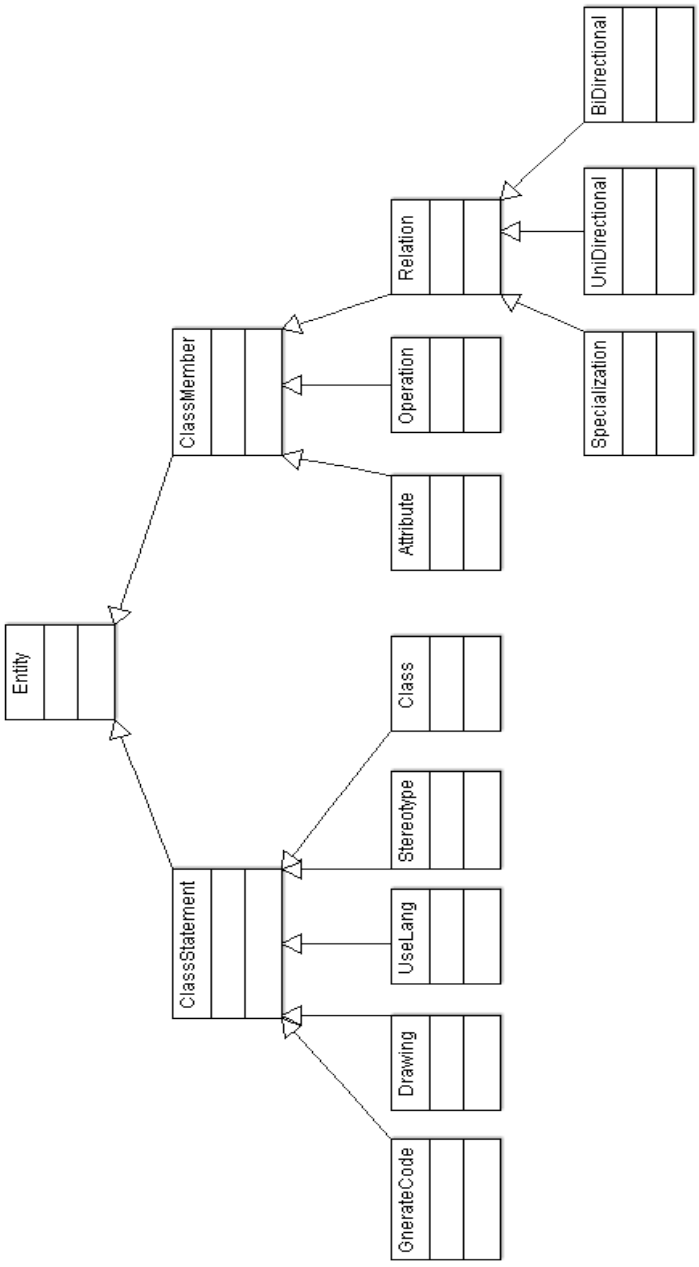
```
define class GenerateCode
{
      attributes:
            private langType: LangType;
```

```
     operations:
          public GenerateCode(in type: LangType);
          public setLangType(in type: LangType): void;
          public getLangType(): LangType;
};

define class ClassGenerateJava
{
     operations:
          public ClassGenerateJava(in type: LangType);
          public generateCodeJava(in classStat: ClassStatement
as list): void;
          public InitializeSymTableJava(in sym_table: Entity as
map): Entity as map;

     relations:
          specializes GenerateCode;
};

define class ClassGenerateCPP
{
     public:
          public ClassGenerateCPP(in type: LangType);
          public generateCodeCPP(in classStat: ClassStatement
as list): void;
          public InitializeSymTableCPP(in sym_table: Entity as
map): Entity as map;

     relations:
          specializes GenerateCode;
};
```

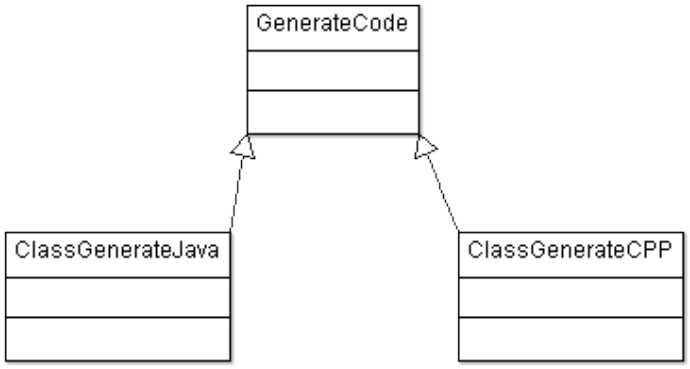
Las funciones que llevan a cabo la generación de código C++ y Java son `generateCodeCPP()` y `generateCodeJava()` respectivamente. Hay otras dos funciones que son de gran importancia, `InitializeSymTableCPP()` e

`InitializeSymTableJava()`, las cuales nos inicializarán la tabla de símbolos según el lenguaje que utilicemos en el código de EzUML con palabras reservadas.

Para dar una visión más global del diseño de entidades se mostrará a continuación el siguiente diagrama de clases:



A continuación mostraremos el diagrama de clases para la generación de código:



3.3 MODELO DINÁMICO DEL TRADUCTOR

En este apartado se abordará el tema del modelo que sigue nuestro traductor. Se mostrará un diagrama de secuencia del mismo. Trataremos de la generación de entidades, dónde y cuándo se realiza los análisis semántico, y cuando hacemos consultas a la tabla de símbolos, etc.

Para la realización de la traducción, nos hace falta un archivo donde estarán sentencias de código EzUML. A partir de ahora, el traductor empezará analizar. Si el analizador no encuentra nada raro, éste seguirá analizando, subiendo por el árbol de entidades, sino nos informará con un mensaje de error por la consola.

Si el analizador encuentra un identificador se comprobará:

1. que exista en la tabla de símbolos.
2. que represente lo que debe representar en el lugar que se encuentre.

Para el apartado 1 tenemos dos casos que se deben contemplar:

1.1 si estoy declarando algo, entonces el identificador que se ha encontrado esta identificando a lo que se ha declarado.

En este caso debemos comprobar en la tabla de símbolos de que existe, en caso afirmativo se mandará un mensaje de error por duplicación de identificador.

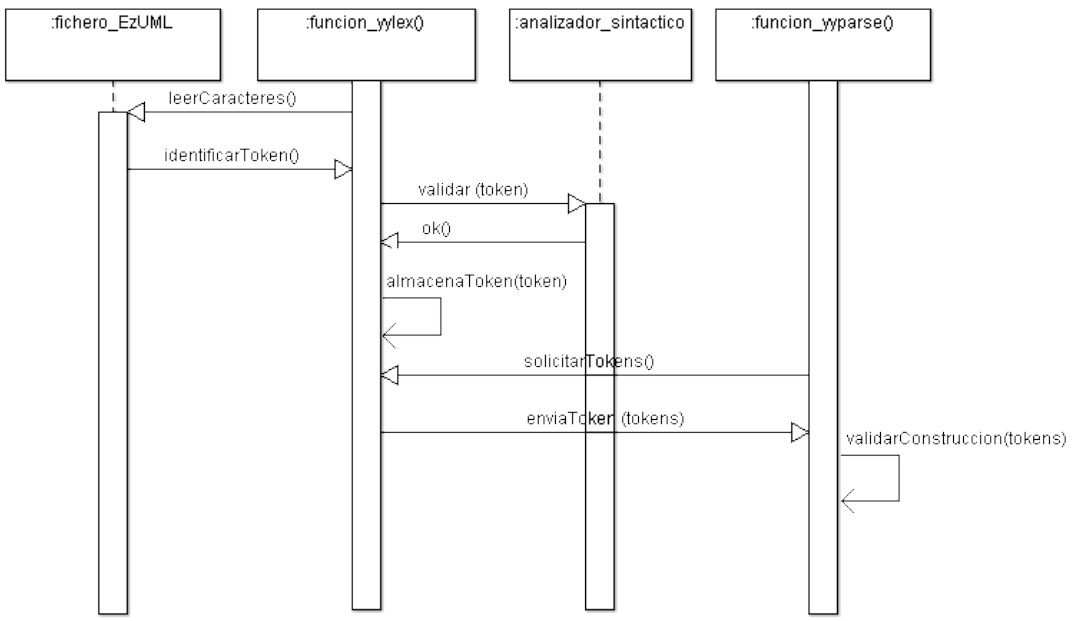
1.2 Si se esta declarando algo, y se ha encontrado un identificador que referencia a otra cosa.

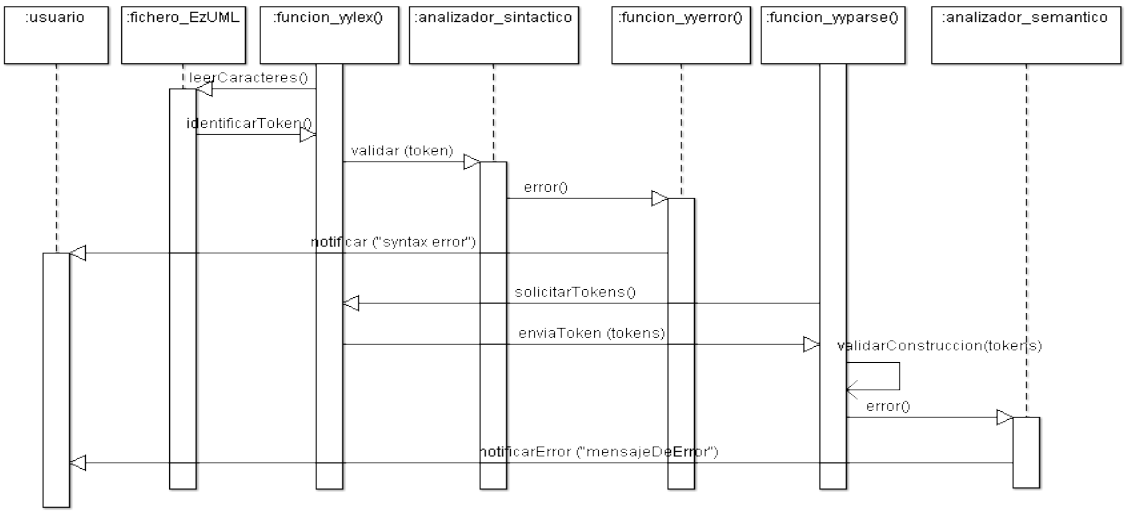
Ahora debemos de comprobar de que exista, de no ser así, damos un error de lo que hemos declarado no existe.

Y para el caso 2, también tendremos otro caso más:

2.1 se comprobará que la entidad que lleve ese identificador tiene el mismo tipo de entidad de la que estamos esperando a recibir. Por ejemplo, si se trata en algún lugar de especificar el tipo de un atributo, el identificador se tendrá que corresponder con una entidad del tipo `Attribute`.

A continuación mostramos el diagrama de secuencia del funcionamiento del traductor. El primer diagrama muestra el funcionamiento de la construcción válida de una sentencia. Y el segundo cuando no es válida una construcción.





En el anterior diagrama podemos observar la notificación del mensaje de "`syntax error`". Esta notificación se usa para la recuperación de errores sintácticos.

Utilizamos la recuperación de errores para que permita continuar al análisis después de que haya ocurrido un error. Así conseguimos mejorar la productividad del programador a la hora de programar, compilar y corregir.

4. IMPLEMENTACIÓN

En este capítulo se hablará de la implementación de EzUML, donde trataremos algunos temas como las herramientas que han sido necesarias para este proyecto, sin entrar en demasiados detalles. Mencionaremos los distintos ficheros que conforman nuestro sistema. Y por último, abordaremos los distintos tipos de la STL empleados, como el uso de flex y yacc, y de la importancia del uso de make.

4.1 HERRAMIENTAS UTILIZADAS

Para la implementación de EzUML hemos elegido C++, debido a la utilización de otras herramientas como son Lex y Yacc y su soporte a la programación orientado a objetos. Estas dos herramientas nos ayudan a generar automáticamente los analizadores lexico y sintáctico, respectivamente. Tanto Lex como Yacc generan código C/C++.

Las herramientas utilizadas en el desarrollo de EzUML son las siguientes:

- **g++ (GCC) 4.24**. Compilador de GNU para C/C++.
- **bison 2.3**. Generador de analizadores sintácticos de GNU.
- **flex 2.5.34**. Generador de analizadores léxicos de GNU.
- **make 3.81**. Herramienta de compilación de proyectos de GNU.
- **subversion 1.5.1**.Sistema de control de versiones.

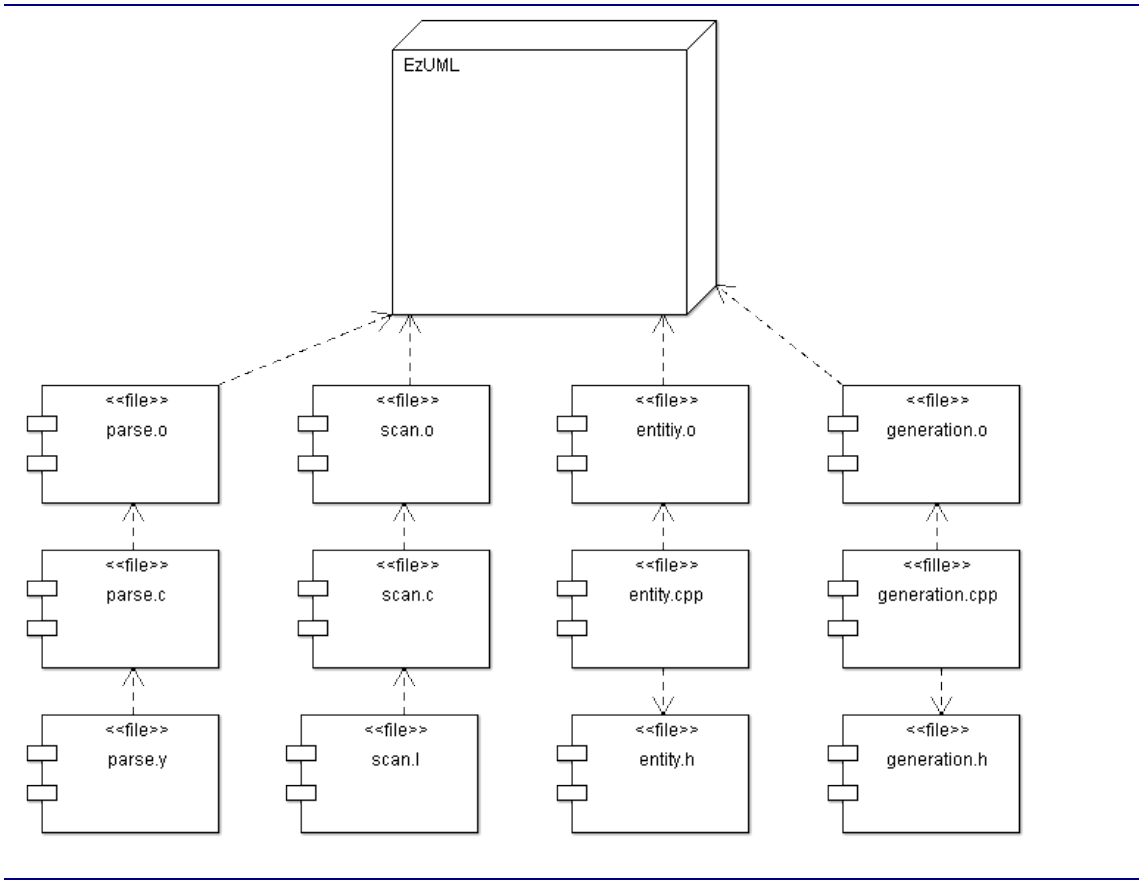
4.2 FICHEROS DEL SISTEMA

El sistema de EzUML lo conforman los siguientes ficheros:

- **entity.h.** En este archivo se encuentran las distintas clases de entidades de UML y las cabeceras de las funciones.
- **entity.cpp.** Se encuentra el código de las distintas clases de entidades de UML declaradas en entity.h.
- **generation.h.** En este archivo se encuentra las distintas clases para la generación de código en los lenguajes Java y C++ con sus respectivas funciones.
- **generation.cpp.** Encontramos el código de las funciones de las clases de generación de código en Java y C++.
- **parse.y.** Archivo que contiene el analizador sintáctico para el lenguaje EzUML.
- **scan.l.** Archivo que contiene los tokens del lenguaje EzUML para el análisis léxico.
- **Makefile.** Archivo que nos ayuda a la construcción del proyecto EzUML.

4.2.1 DIAGRAMA DE DESPLIEGUE

En el siguiente diagrama de despliegue podemos observar los distintos ficheros que necesita EzUML para su funcionamiento.



4.3 TIPOS DE LA STL EMPLEADOS

En este apartado hablaremos sobre los distintos tipos empleados de la STL (Standar Template Library) de C++, facilitándonos la implementación de contenedores como son vectores, listas enlazadas, etc.

La clase `list` tiene una estructura genérica de listas enlazadas. Su orden de complejidad es O(1) en la inserción, y en la búsqueda O(n). Se ha utilizado para almacenar estereotipos, tipos de entidad, argumentos de una operación, atributos,

operaciones de una clase, identificadores, relaciones entre clases, clases de `ClassMember` y clases de `ClassStatement`.

La clase `map` nos permite asociar una clave a un dato. El `map` toma dos parámetros, el tipo de la clave y el tipo de dato. La complejidad para la búsqueda y la inserción es O(log(n)). En el proyecto, se hace uso de `map` para la tabla de símbolos. Donde el primer parámetro que toma `map` para nuestra tabla de símbolo es el identificador de la entidad y el segundo parámetro es el tipo de entidad.

El tipo `string` se ha utilizado para los identificadores y la multiplicidad de las relaciones entre clase.

Y por último, la clase `fstream`, es una interfaz que nos proporciona la lectura y escritura de datos a partir de archivos de entrada o de salida. Esta interfaz, nos ha sido útil para los archivos de salida a la hora de generar código en Java o C++ por medio de la sentencia `generate code java` o `generate code cpp`.

4.4 USO DE FLEX Y YACC

Para una mayor compresión haremos una breve explicación del funcionamiento de flex y yacc a continuación para así tener unos conceptos básicos del tema.

4.4.1 FLEX

flex es una herramienta para generar escáneres, programas que reconocen patrones léxicos en un texto. flex nos permite leer ficheros de la entrada, o la entrada estándar si no se le ha indicado ningún nombre de fichero. Nos generará código fuente en C, a partir de una serie de especificaciones escritas en lenguaje lex. El código C generado contiene una función llamada `yylex()`, que localiza cadenas en la entrada que se ajustan a cada uno de los patrones léxicos especificados en el código fuente lex y realizando las acciones asociadas a cada patrón.

Un programa fuente de lex tiene el siguiente aspecto:

```
<sección de declaraciones>
```

```
%%
<sección de reglas>
%%
<sección de rutinas>
```

Donde la sección de declaraciones incluye declaraciones de variables, constantes y definiciones regulares. Por ejemplo:

```
letra      [A-Za-z]
```

La sección de reglas especifica los patrones a reconocer y las acciones asociadas a éstos, muy similar a la que utiliza `awk`:

```
patron     {acciones en C}
```

Y por último la sección de rutinas permite definir funciones auxiliares en C, incluyendo la función `main()`. Por defecto lex nos proporciona un `main()` que llama a la función `yylex()`.

A continuación mostramos un trozo de código del fichero scan.l de EzUML escrito en flex.

```
code        {
                return CODE_TK;
            }
java        {
                return JAVA_TK;
            }
cpp         {
                return CPP_TK;
            }
association {
                return ASSOCIATION_SIMPLE_TK;
            }
aggregation {
                return AGGREGATION_TK;
            }
```

```
composition {
                return COMPOSITION_TK;
            }
bidir       {
                return BIDIR_TK;
            }
specializes {
                return SPECIALIZES_TK;
            }
[_a-zA-Z][_a-zA-Z0-9]*  {
                            yylval.string=strdup(yytext);
                            return IDENTIFIER;
                        }
```

4.4.2 YACC

Como hemos visto el análisis léxico realiza la tarea de reconocer los elementos de un lenguaje uno a uno. yacc se centra en el análisis sintáctico para saber si un fichero de entrada respeta las reglas de una gramática.

yacc es un generador de analizadores sintácticos. A partir de un fichero fuente en yacc, se genera un fichero en C que contiene el analizador sintáctico. Para que pueda funcionar el analizador sintáctico es necesario un analizador léxico. Es decir, el fichero fuente en C que ha generado yacc tiene una llamada denominada yyparse() que realiza llamadas a la función yylex() que deberá estar definida y nos devolverá el tipo de lexema encontrado, para así comprobar si la construcción es válida según la gramática. También se deberá encontrar la función yyerror(), que será invocada cuando el analizador sintáctico encuentre un símbolo que no encaje con la gramática.

Un programa fuente de yacc es parecido a uno de lex, con la salvedad de que la sección de reglas contiene reglas de gramáticas en vez de expresiones regulares.

```
<sección de declaraciones>
%%
```

```
<sección de reglas>
%%
<sección de rutinas>
```

En la sección de declaraciones se definen los símbolos terminales de la gramática mediante la directriz `%token`.

La sección de reglas contendrá la grámatica. En el lado izquierdo de la regla aparece un no terminal y a su derecha una combinación terminales y no terminales. Toda regla deberá incluir una acción en C.

```
no_terminal: combinación_de_terminales_y_no_terminales
           {Acciones en C}
```

En la sección de rutinas se incluirán las funciones `main()`, `yyles()` e `yyerror()`.

También se pueden devolver valores en la parte izquierda de una regla utilizando unas variables especiales de yacc a la vez que se aplica una regla de derivación. Esto nos permite mover valores de forma ascendente, a la vez que se realiza el análisis de una expresión. Estas variables nos permiten almacenar un valor semántico en cada uno de los símbolos de cada regla. Las variables que utiliza yacc son:

- `$$`, representa la parte izquierda de una regla.
- `$1`, `$2`, …, `$n`, representan los valores asociados a los símbolos de parte derecha de una regla. Donde `$n` es la posición n en la parte derecha de la regla.

Vamos a ver como interacciona flex y yacc en el siguiente ejemplo de una suma de enteros:

```
linea: exp_entera
{
     printf("%d\n", $1);
}
;
exp_entera: TOK_ENTERO
```

```
{
      $$ = $1;
}
|exp_entera '+' exp_entera
{
      $$ = $1 + $3;
}
```

Si tenemos como entrada `9 + 3`, la función `yylex()` le de la entrada el `9` porque concuerda con `TOK_ENTERO`, y le pasa la información a la función `yyparse()` mediante el valor de retorno `yylex()` y la variable `yyval`.

La variable `yylval` se establece al valor definido por la macro YYSTYPE. En el ejemplo anterior, para implementar el comportamiento esperado por yacc, definiríamos la macro YYSTYPE como `TOK_ENTERO` y la regla Lex:

```
[0-9]+
{
      yylval = atoi(yytext);
      return TOK_ENTERO;
}
```

Ahora `yyparse()` realiza una operación de desplazamiento apilando el símbolo y su correspondiente valor. Después `yyparse()` realiza una operación de reducción en la regla `exp_entera: TOK_ENTERO`. Esta reducción consta de los siguientes pasos.

1. se desapilan los símbolos de la parte derecha de la regla con sus valores semánticos correspondientes.
2. se ejecuta la acción `$$= $1;`.
3. Se vuelve apilar el símbolo el símbolo de la parte izquierda con su valor semántico, quedando `exp_entera= 9`.

Seguidamente `yylex()` le de la entrada el simbolo `+`, y le pasa la información a `yyparse()` por medio del valor de retorno de `yylex()`. La función `yyparse()`

realiza una función de desplazamiento apilando el símbolo +, pero sin ningún valor semántico.

Después nos encontramos con el `3`, `yylex()` lee de la entrada el `3` y éste le pasa la información a `yyparse()` mediante el valor de retorno de `yylex()` y la variable `yyval`. Se realiza por medio `yyparse()` una operación de desplazamiento y se apila el símbolo y su valor correspondiente.

En la regla `exp_entera: TOK_ENTERO` se produce una operación de reducción por parte de `yyparse()`, al igual que en los tres pasos anteriores. Después se ejecuta la acción asociada a la regla `$$= $1 + $3`, y se apila el símbolo de la parte izquierda con su valor semántico, en este caso `11`.

Por último, `yyparse()` realiza una operación de reducción de la regla `linea: exp_entera`. Se vuelve a desafilar los símbolos de la parte de la derecha de la regla junto con sus valores semánticos (`11`). Y se ejecuta la acción asociada a la regla `printf("%d\n", $1)`.

A continuación entraremos en detalles para explicar el funcionamiento de la recuperación de errores y la notificación de errores de yacc.

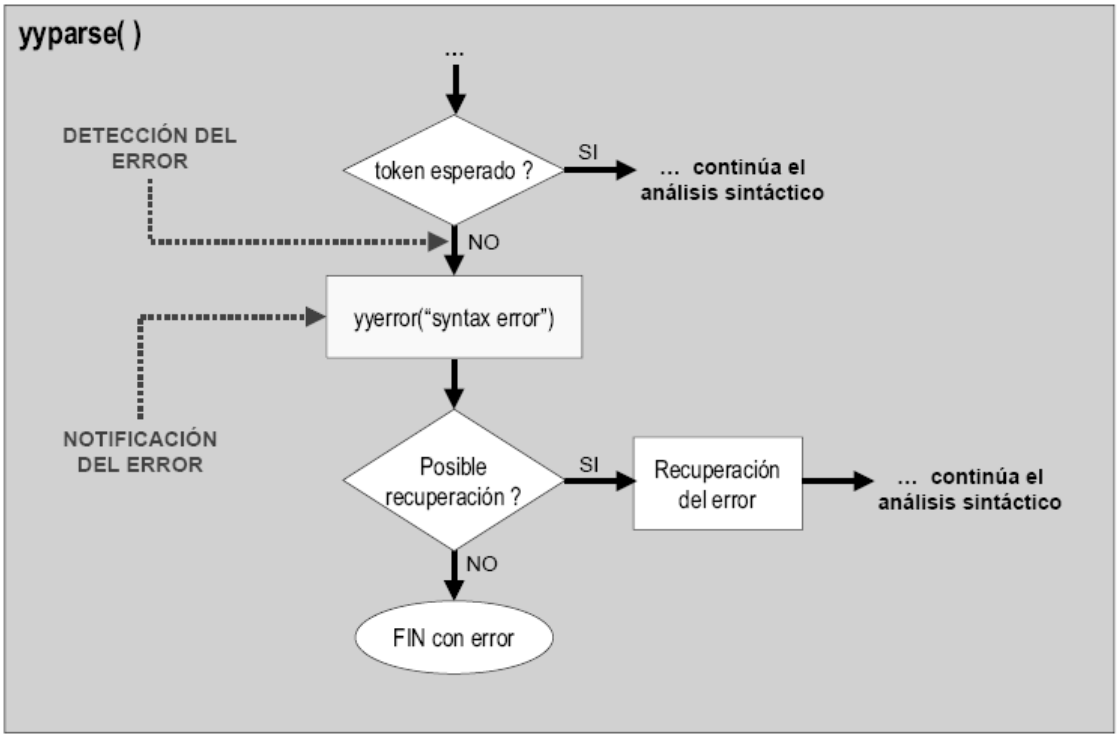
Si el error producido es durante el análisis sintáctico, la función `yyparse()` recibe de la función `yylex()` del analizador léxico un token que no puede satisfacer ninguna producción de la gramática. Cuando detecta este tipo de error la función `yyparse()`, ésta invoca la función `yyerror()` para que sea notificado.

La declaración de la función `yerror()` es la siguiente:

```
void yyerror(const char *s)
{
     fprintf(stderr, "Line %d: %s: unexpected '%s'\n", yylineno,
s, yytext);
}
```

Como podemos observar el mensaje saldrá por la salida estándar de error, mostrándonos la línea, el mensaje genérico de "syntax error" y un mensaje de la cadena de caracteres involucrada antes de producirse el error.

Una vez finalizada la función yyerror(), la función yyparse() intenta la recuperación del error, si la recuperación no es posible, yyparse() nos devolverá un 1.



Si se produce el error durante el análisis semántico, este se notificará por medio del objeto `cerr`, que representa el flujo de error estándar. La llamada de estas notificaciones ha sido por medio de macros. Se ha utilizado macros para una mayor legibilidad del código a la hora de leer la llamada de la macro.

A continuación mostramos las distintas macros utilizadas para el informe de errores semánticos.

```
#define REPEATED_ENTITY_ERROR(s) (cerr<<"Line: "<<yylineno<<":
error: '"<<s<<"' alredeady declared."<<endl)
#define NOT_FOUND_ENTITY_ERROR(s) (cerr<<"Line: "<<yylineno<<":
error: '"<<s<<"' not found."<<endl)
```

```
#define REPEATED_ENTITY_CLASS_ERROR(s) (cerr<<"Line:
"<<yylineno<<": error: '"<<s<<"' alredeady declared as
class."<<endl)
#define NOT_CORRESPONDED_CLASS_ERROR(s) (cerr<<"Line:
"<<yylineno<<": error: '"<<s<<"' not corresponded with a
class."<<endl)
#define NOT_CORRESPONDED_STEREOTYPE_ERROR(s) (cerr<<"Line:
"<<yylineno<<": error: '"<<s<<"' not corresponded with a
stereotype."<<endl)
#define NOT_CORRESPONDED_MEMBERTYPE_ERROR(s) (cerr<<"Line:
"<<yylineno<<": error: '"<<s<<"' not corresponded with a member
of a class."<<endl)
#define APPLICABLE_TO_ERROR(s) (cerr<<"Line: "<<yylineno<<":
error: stereotype '"<<s<<"' cannot be applied to class."<<endl)
#define DUPLICATED_GENERATE_CODE (cerr<<"Line: "<<yylineno<<":
error: duplicated generate code."<<endl)
#define NOT_USE_LANG_SETUP (cerr<<"Line: 1: error: not declared
use language (java or cpp)."<<endl)
#define ERROR_GENERATE_CODE (cerr<<"Line: "<<yylineno<<": error:
not generated code."<<endl)
```

En la anterior sección se puede apreciar que la información del error nos mostrará siempre la línea donde se ha producido y en algunos casos una cadena de caracteres de donde reporta el error, y seguidamente una información más detallada del tipo de error.

Para la recuperación de estos errores se utiliza el token especial "`error`", el cual está reservado y definido para la gestión de errores.

El token "`error`" se utiliza para la construcción de nuevas reglas gramaticales, situándolo en posiciones concretas donde creamos que se pueda producir errores. Un ejemplo sería el siguiente:

```
sentencias: /*lambda*/
| sentencias ';'
| sentencias exp ';'
| sentencias error ';'
```

```
;
```

Si nos fijamos en la anterior sección, podemos ver el token "`error`" donde está posicionado en esa regla. La construcción será válida si encontramos un token "`error`", seguido de "`;`" después de "`sentencias`".

Se producirá la recuperación de errores cuando nos encontremos con algún error en el no terminal "`exp`". En este momento el analizador saca elementos de la pila de análisis hasta que se llegue a un estado en el que el token "`error`" sea legal, en nuestro ejemplo, "`sentencias`", y se avanza la entrada hasta que encontremos un token legal, que sería "`;`".

Como se ha comentado anteriormente, hay que situar el token "`error`" en determinadas posiciones de la regla en busca de los posibles errores. En nuestro caso, concretamente, situamos el token "`error`" después de haber almacenado información en la tabla de símbolos, para así reconocer elementos que necesitemos posteriormente.

Hay que tener en cuenta que en las reglas de recuperación de errores debemos de tener una acción semántica para esa producción, para así ascender el valor semántico del no terminal de la izquierda de la producción. En el caso de no dar una acción semántica se aplicará la regla por defecto `$$= $1`, dando lugar a valores semánticos no válidos y posibles fallos de segmentación.

4.5 USO DE MAKE

Los Makefiles son ficheros empleados por un programa llamado make, para automatizar tareas de construcción de software. A medida que los programas crecen en complejidad y tamaño, su construcción se vuelve más compleja, y es por esto el uso de esta herramienta. Además, los Makefiles nos ayudan a evitar los errores que puedan surgir y facilitan a otras personas la compilación de un código cuyas peculiaridades desconocen.

También hay que mencionar, que un programador sigue ciertos pasos como editar un fichero fuente, compilar y enlazar construyendo un ejecutable y depurar el resultado. Lo

importante de los Makefiles, es resolvernos que ficheros fuente han sido modificados para poder generarnos aquellos ficheros resultado que dependían de los primeros, ahorrándonos volver a compilar unidades que no han sufrido cambios.

Las reglas de un Makefiles tienen el siguiente formato:

```
destino : requisito ...
     comando
     ...
```

Donde `destino` es el nombre de un archivo a crear, un ejecutable o un archivo objeto `.o`. También puede ser el nombre de una tarea realizar, como puede ser hacer un `clean`, para eliminar archivos objeto y recomenzar una compilación desde el principio. Y `requisito` es el nombre de un archivo del cual depende el destino a crear. Hay que destacar que un `destino` puede depender de varios archivos `requisito`. Cuando el `destino` es el nombre de una tarea no hay requisitos, como es el caso de `clean`.

A continuación mostramos el Makefile del sistema:

```
CC = g++
YACC= yacc
FLEX= flex

all: ezuml

ezuml: parse.o scan.o entity.o generation.o
     $(CC) -o ezuml parse.o scan.o entity.o generation.o

entity.o: entity.cpp
     $(CC) -c entity.cpp

generation.o: generation.cpp
     $(CC) -c generation.cpp

parse.o: parse.c
```

```
	$(CC) -c parse.c

parse.c: parse.y
	$(YACC) -d -o parse.c parse.y

scan.o: scan.c
	$(CC) -c scan.c

scan.c: scan.l
	$(FLEX) -o scan.c scan.l

clean:
	rm -f parse.c scan.c parse.h ezuml *.o
```

Al principio del fichero podemos observar la declaración de las variables `CC`, `YACC` y `FLEX`. Podemos apreciar el destino `all`, que por medio de la invocación make all podemos realizar la compilación de nuestro sistema. Con la invocación de `make clean` que nos ayuda a limpiar las fuentes y los objetos.

5. CONCLUSIONES Y DESARROLLOS FUTUROS

Por ahora se han desarrollado los capítulos de análisis, diseño e implementación de EzUML. Ahora es el momento de evaluar el trabajo realizado y desarrollar en futuro la acción de gráficos de UML.

5.1 ÁNALISIS

En este punto se observó minuciosamente el problema que se planteaba en la construcción de un compilador aplicado al lenguaje UML. Hablamos de la necesidad de su creación con respecto a los problemas que nos plantean las herramientas WYSIWYG, y así poder facilitar la vida del desarrollador. Debido a este problema, se comienza con el desarrollo de EzUML cumpliendo con los requisitos y especificaciones que nos plantea el lenguaje UML.

Para poder afrontar mejor el problema de forma más fácil, se describe la especificación del lenguaje de EzUML y el funcionamiento de un compilador, así como sus distintas fases.

5.2 DISEÑO

En la fase de diseño nos basamos en los bloques funcionales que conforman nuestro sistema, como la tabla de símbolos, consola de errores, árbol de entidades, etc., para la arquitectura básica del sistema. Detallamos el modelo dinámico que sigue nuestro traductor, comentando dónde se generan las entidades, dónde y cuándo realizamos los análisis semánticos y cuándo debemos consultar a la tabla de símbolos. En este punto se ha tenido en cuenta posibles modificaciones futuras en el diseño para que afecten lo mínimo posible a la arquitectura básica del sistema.

5.3 IMPLEMENTACIÓN

En relación con la implementación, se abordó los distintos ficheros que componen nuestro sistema, así como las herramientas utilizadas para la realización de los mismos. La utilización de los tipos de la STL de C++ en los ficheros nos generó una gran ayuda a la hora de búsquedas e inserciones, tanto en listas, como en la tabla de símbolos de tipo clase `map`. Además, gracias a la ayuda de las herramientas flex y yacc podemos generar los reconocedores de análisis léxico y sintáctico, respectivamente.

En está fase a la hora de la implementación, nos encontramos con una serie de problemas, tales como la reducción de las producciones para ir subiendo el contenido hacia el nodo raíz. Cuando vamos subiendo o reduciendo las producciones hacia el nodo raíz, en algunos casos, tenemos el problema de querer acceder a algunas características de una clase. Para poder acceder a los objetos de la clase, debemos aplicar las técnicas de herencia y polimorfismo de funciones que nos ofrece C++. Otro problema fue donde situar el token `error`, debido a que es un trabajo minucioso y que debemos situarnos en la personalidad del usuario para saber donde puede ocasionar posibles errores a la hora de usar el lenguaje EzUML.

5.4 ESTADÍSTICAS GENERALES

A continuación mostraremos una estimación de costes, de duración y de número de desarrolladores a través del análisis del código fuente. Para dar una visión orientativa del código realizado para el proyecto EzUML, utilizaremos la herramienta SLOCCount.

- Líneas físicas totales de código fuente:

LENGUAJE DE PROGRAMACIÓN	Nº LÍNEAS DE CÓDIGO
C++	1.444 (55,48%)
FLEX	163 (38,25%)
YACC	995 (6,27%)
TOTAL	**2.602**

- Estimación de costes, de duración y de número de desarrolladores:
  - Estimación de esfuerzo de desarrollo para un único desarrollador: 6,55 meses (0,55 años).
  - Tiempo estimado de desarrollo: 5,11 meses (0.43 años).
  - Número medio de desarrolladores: 1,28
  - Costes económicos de desarrollo: 73.712$= 60.987,86 €

  Está estimación está basada en el modelo de desarrollo COCOMO, el cuál no resulta ser muy fiable, pero nos ayuda a hacernos una idea del esfuerzo realizado.

5.5 DESARROLLOS FUTUROS

En este momento, EzUML no cubre todas las especificaciones por el cual se comenzó a desarrollar. Por ahora, podemos introducir sentencias del lenguaje EzUML, compilar este lenguaje y que nos reporte los errores si ha habido alguno, y que nos genere automáticamente a partir de las sentencias introducidas código en el lenguaje C++ o Java.

Para poder cumplir con toda la funcionalidad de EzUML, nos faltaría la acción de dibujado de gráficos de UML, la cual es una de las características más importantes de esta herramienta CASE. En un futuro, se implementará la acción de dibujado y así pueda cumplir todas las funcionalidades EzUML, siendo así, una herramienta muy atractiva para programadores.

Para poder realizar la generación de diagramas UML, deberemos buscar nuevos requisitos en la fase de análisis para después utilizarlos en el diseño y posteriormente implementarlos.

Como punto final, cuando EzUML se encuentre totalmente disponible se publicará una primera versión. Esta primera versión será una versión beta, la cual deberá pasar un proceso de validación, de tal forma que podamos considerar estable, y también sea útil como una herramienta CASE.

5.6 CONCLUSIONES PERSONALES

Después de haber realizado y comentado el proyecto de EzUML, es el momento de evaluar todo lo aprendido en el desarrollo de éste mismo.

A lo largo del proyecto, he adquirido conocimientos sobre los lenguajes alto nivel ocasionado por el desarrollo de un compilador, así como, la obtención de herramientas y técnicas necesarias para este tipo de aplicaciones.

Que decir tiene, la ampliación de mis conocimientos y asentamiento de mis bases respecto al entorno de la orientación a objetos. Gracias al lenguaje C++, profundice más en el desarrollo orientado a objetos con las técnicas de polimorfismo, herencia de funciones y otras peculiaridades más.

Gracias a este proyecto, he conseguido mejorar mi formación y consolidando mis conocimientos aprendidos en la carrera.

6. BIBLIOGRAFÍA

1. Programación orientada a objetos con C++. Ceballos Sierra, Francisco Javier.
2. C/C++ curso de programación. Ceballos Sierra, Francisco Javier
3. Compiladores principios, técnicas y herramientas. Aho, Alfred V.
4. Lex and yacc. Levine, John R.
5. Manual de Flex. http://flex.sourceforge.net/manual/
6. Manual de Bison. http://www.gnu.org/software/bison/manual/bison.html
7. Otras páginas Web de interés del uso de flex y yacc:
    - http://arantxa.ii.uam.es/~mdlcruz/docencia/compiladores/2002_2003/compiladores_02_03_yacc_bison.pdf
    - http://epaperpress.com/lexandyacc/
    - http://www.medina-web.com/programas/documents/tutoriales/lex_yacc/index.html
8. Página Web de la asignatura Teoría de Autómatas y Lenguajes Formales de la Universidad de Valladolid. http://www.infor.uva.es/~mluisa/talf/
9. Página Web de la asignatura Procesadores del Lenguaje de la Universidad de Vigo:
    - http://ccia.ei.uvigo.es/docencia/PL/doc/flex.pdf
    - http://ccia.ei.uvigo.es/docencia/PL/bison.pdf
10. Manual de control de versiones con Subversion. http://svnbook.red-bean.com/nightly/es/svn-book.pdf
11. Tutorial del lenguaje C++. http://www.cplusplus.com/doc/tutorial/
12. Referencias del lenguaje C++. http://www.cppreference.com/
13. SLOCCount, David A. Wheeler. http://www.dwheeler.com/sloccount/
14. Basic COCOMO Model, National Aeronautics and Space Administration. http://www.freetutes.com/systemanalysis/sa3-cocomo.html