UNIVERSIDAD DE LAS CIENCIAS INFORMÁTICAS

FACULTAD 3

Entorno de Desarrollo Integrado para el lenguaje ensamblador

Trabajo de Diploma para optar por el título de
Ingeniero en Ciencias Informáticas

Autora:
Anabel González Valiente

Tutor:
Ing. René Queizán Pérez

Co-Tutora:
MSc. Yalice Gámez Batista

La Habana, junio de 2015
"Año 57 de la Revolución"

La programación en bajo nivel es buena para el alma del programador

John Carmack - cofundador de id Software

DECLARACIÓN DE AUTORÍA

Declaro ser la autora de la presente tesis y reconozco a la Universidad de las Ciencias Informáticas los derechos patrimoniales de la misma, con carácter exclusivo.

Para que así conste firmo la presente a los ____ días del mes de _____ del año _____.

_____

Anabel González Valiente

Autora

_____ _____

Ing. René Queizán Pérez MSc. Yalice Gámez Batista

Tutor Co-Tutora

DATOS DE CONTACTO

Datos de tutor:

Nombre: Ing. René Queizán Pérez

Correo electrónico: rqueizan@uci.cu

Datos de Co-Tutor:

Nombre: MSc. Yalice Gámez Batista

Correo electrónico: yaliceg@uci.cu

Datos del Autor:

Nombre: Anabel González Valiente

Correo electrónico: avaliente@estudiantes.uci.cu

AGRADECIMIENTOS

Agradezco:

A mi familia por el apoyo que me brindó para alcanzar esta meta, principalmente a mi mamá y mi papá, por haber hecho de mí la mujer que soy.

A pipo y mima por haberme criado como una hija y darme todo el amor del mundo.

A mis tíos, Ariel, Raúl, Jorge, Héctor y Marita por apoyarme en todo.

A mis primos, Arielito, Melissa y Yaima por ser como hermanos para mí.

A los familiares que ya no están pero que influyeron en mi educación y me ayudaron a llegar hasta aquí: primo Jorge Luis, tía Barbarita, abuelo Héctor, abuela María, abuela Nena, tía Alejandrina y Chaca.

A mi hermano Alejandro por estar juntos desde pequeños.

A mis compañeros de grupo, por ser la familia que somos.

A mis amiguis Laura, Lorena, Arian y César por ser como mis hermanos, por los momentos inolvidables que pasamos juntos y por apoyarme en las situaciones difíciles.

A mis amigos Armando, Neri, Magda y Jose. A mi amigo y profesor de Matemáticas Mesa por todas las dudas que me aclaró y por su apoyo.

A mis compañeros y profesores del preuniversitario.

Un agradecimiento a todos los profesores que me formaron como profesional, especialmente a Yulierki, Yoenry, Bolivar, Maikel Navarro, Lytyet, Rosalina, Ana Cecilia, Dailien, Erlys, Reyder, Leandro y Yenin.

A los niños de mis ojos Lucas y Angélica.

A mi co-tutora Yalice por su paciencia y por todo el apoyo brindado.

Al tribunal por todas sus recomendaciones.

A la Revolución por darme la oportunidad de estudiar en una universidad como esta.

Un agradecimiento especial a mi novio y tutor por todo el amor que me da y por la inmensa ayuda que me ha brindado en la realización de este trabajo, por estar ahí para aclarar mis dudas, por estar ahí cuando más lo necesito, por quererme, por …, en fin gracias por existir.

Finalmente agradecer a todo aquel que de una forma u otra ha colaborado para hacer este sueño realidad.

DEDICATORIA

Dedico este trabajo a toda mi familia

RESUMEN

En la Universidad de las Ciencias Informáticas, en el segundo año se imparte la asignatura de Arquitectura de Computadoras. Esta asignatura tiene entre sus objetivos que los estudiantes sean capaces de implementar subrutinas de mediana y baja complejidad en lenguaje ensamblador. Sin embargo las herramientas que actualmente se utilizan atentan en alguna medida contra el desarrollo de esta habilidad, debido a que carecen de una interfaz gráfica y de otras facilidades que son características de las herramientas que ellos comúnmente utilizan. El presente trabajo de diploma tiene como finalidad desarrollar un entorno de desarrollo integrado (IDE por sus siglas en inglés) que facilite los procesos de codificación, compilación y ejecución del código en lenguaje ensamblador para los estudiantes de segundo año de la Universidad de las Ciencias Informáticas. La herramienta está diseñada para los estudiantes y profesores de la asignatura de Arquitectura de Computadoras. La herramienta soporta el repertorio de instrucciones básico del lenguaje ensamblador y brinda las facilidades de mostrar los errores en tiempo de codificación, permite autocompletar código, resalta la sintaxis del lenguaje e integra la compilación y ejecución de código. Se establecieron los referentes teóricos de los IDE y las fases del proceso de compilación. El desarrollo de la herramienta estuvo guiado por las pautas de la metodología de desarrollo de software XP. Para garantizar la calidad del producto se aplicaron métricas y pruebas, que sirvieron para verificar el correcto funcionamiento de la herramienta y para validar que la herramienta cumple con las expectativas del cliente.

Palabras claves: compilación, ensamblador, entornos de desarrollo integrado, FASM

ABSTRACT

In the second year of the University of Informatics Sciences, the course "Computer Architecture" is taught. This course's objectives is that students are able to implement subroutines medium and low complexity in assembly language. But the tools currently used to some extent attempt against the development of this skill, because they lack a graphical interface and other facilities that are characteristic of the tools they commonly use. This dissertation aims to develop an integrated development environment that facilitates the process of coding, compiling and running the code in assembly language for second year students at the University of Informatics Sciences. The tool is designed for students and teachers of the subject Computer Architecture. The tool supports the basic set of assembly language instructions and provIDE the facilities of display errors in coding time, allows autocomplete code, language syntax highlights and integrates the compilation and code execution. The theoretical framework of the IDE and the phases of the compilation process were established. The development of the tool was guided by the patterns of development methodology XP software. To ensure product quality metrics and tests was applied, which also served to verify the correct operation of the tool, and validate that meets customer expectations.

Keywords: assembly, compilation, FASM, integrated development environment

ÍNDICE DE CONTENIDO

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

INTRODUCCIÓN

El lenguaje ensamblador fue desarrollado en la década de los 60. Su importancia radica principalmente en que trabaja directamente con el microprocesador, dándole al programador la capacidad de realizar tareas muy técnicas que serían difíciles de implementar en un lenguaje de alto nivel (Dart, y otros, 2008). De ahí que la carrera de Informática en las universidades cubanas asuma la enseñanza del lenguaje ensamblador.

La programación en ensamblador se caracteriza por ser de alta complejidad para los estudiantes debido a que constituye un cambio de filosofía, en no sólo la programación sino también en la filosofía y estructura del pensamiento lógico. Sin embargo es de suma importancia su impartición pues permite un mayor entendimiento del funcionamiento interno de las computadoras.

La Universidad de las Ciencias Informáticas (UCI), desde sus inicios, incorporó la enseñanza del lenguaje ensamblador en las asignaturas de Máquinas Computadoras I y II, que luego evolucionaron a la asignatura de Arquitectura de Computadoras, todas del segundo año de la carrera. La asignatura de Arquitectura de Computadoras tiene dentro de sus objetivos que los estudiantes sean capaces de implementar subrutinas de mediana y baja complejidad en lenguaje ensamblador.

En el año 2011, como parte de la estrategia de migración hacia el software libre adoptada por el país y por la universidad, se comienza a trabajar en la asignatura con herramientas libres y sobre el sistema operativo Linux. Esto determinó que se restringieran las herramientas a utilizar. Actualmente las herramientas que se utilizan en la asignatura dificultan la correcta asimilación del contenido por parte de los estudiantes. Ellas son: un editor de texto donde se escribe el código, el compilador FASM que funciona en Linux a nivel de consola, y finalmente la máquina virtual Qemu para la ejecución del programa. Todas ellas se trabajan a nivel de comandos en consola, lo que provoca rechazo en los estudiantes, acostumbrados a utilizar herramientas como un entorno de desarrollo integrado (IDE por sus siglas en inglés) con grandes facilidades de interfaz gráfica.

Partiendo de la problemática antes descrita, se identifica como **problema de la investigación:** Los estudiantes de segundo año de la carrera Ingeniería en Ciencias Informáticas presentan dificultades en la programación en ensamblador debido a lo engorroso de los procesos de codificación, compilación y ejecución del código.

El **objeto de estudio** lo constituye: El proceso de desarrollo de software de los IDE y el **campo de acción**: El proceso de desarrollo de software de los IDE para el lenguaje ensamblador.

Para darle solución al problema planteado se persigue como **objetivo general:** Desarrollar un IDE que facilite los procesos de codificación, compilación y ejecución del código en lenguaje

ensamblador, para los estudiantes de segundo año de la carrera Ingeniería en Ciencias Informáticas.

Para darle cumplimiento al objetivo general se definen los siguientes **objetivos específicos**:

1. Establecer los referentes teóricos de los IDE.
2. Obtener el modelo de diseño del IDE atendiendo a las regularidades identificadas en los referentes teóricos y su contextualización al problema identificado.
3. Implementar el IDE atendiendo al diseño realizado de manera tal que facilite los procesos de codificación, compilación y ejecución del código en lenguaje ensamblador.
4. Validar la solución propuesta.

Para lograr la realización de los objetivos específicos se trazaron las siguientes **tareas de la investigación:**

1. Análisis del estado del arte de los IDE para ensamblador
2. Selección de las herramientas y tecnologías a utilizar para el desarrollo del IDE
3. Definición de las funcionalidades del sistema
4. Obtención del modelo de diseño del sistema
5. Selección de los estándares de codificación
6. Implementación de las funcionalidades
7. Diseño de Casos de Pruebas
8. Validación del sistema

Con la realización del presente trabajo de diploma se pretende obtener un IDE, que facilite los procesos de codificación, compilación y ejecución del código en lenguaje ensamblador, para los estudiantes de segundo año de la carrera Ingeniería en Ciencias Informáticas. Y se tiene como **idea a defender**: El desarrollo de un IDE facilitará los procesos de codificación, compilación y ejecución del código en lenguaje ensamblador.

Se hace uso en la presente investigación de los siguientes métodos científicos:

Métodos teóricos:

- **Histórico-lógico:** se realiza un análisis de cómo han evolucionado los diferentes IDE, obteniendo una tendencia de las características de los compiladores para su incorporación en la propuesta.
- **Analítico-sintético:** se utiliza durante el estudio y análisis de las bibliografías consultadas para asumir posiciones y determinar los fundamentos en relación con el objeto investigado.

- **Modelación:** se modela el desarrollo de la solución a partir de los artefactos que establece la metodología de desarrollo.

Métodos empíricos:

- **Medición:** se utiliza en las métricas aplicadas para validar los requisitos, el código y en las pruebas realizadas al sistema para verificar y validar su correcto funcionamiento.
- **Encuesta:** se utiliza para constatar la necesidad de una nueva herramienta y para validar el objetivo general de la investigación.

El documento se encuentra estructurado de la siguiente forma:

Capítulo 1. Fundamentación teórica: se establecen los referentes teóricos relacionados con los IDE y el proceso de compilación. Se realiza un análisis de los IDE existentes para programar en lenguaje ensamblador. Se realiza el análisis y selección de la metodología, herramientas y tecnologías que mejor se adecuen a la solución propuesta.

Capítulo 2. Propuesta de solución: a partir de la metodología seleccionada, refleja cada una de las etapas del desarrollo de la solución. Se explican y documentan cada uno de los artefactos generados y que dan soporte a la especificación de sus funcionalidades, la arquitectura, el modelo de diseño, así como el uso de patrones y estándares de codificación.

Capítulo 3. Validación de la solución propuesta: se analizan y seleccionan las técnicas, pruebas y métricas a utilizar para la validación de la solución propuesta, en cada una de las etapas del desarrollo. Se documentan las pruebas realizadas y se analizan los resultados obtenidos.

CAPÍTULO 1: FUNDAMENTACIÓN TEÓRICA

El presente establece los referentes teóricos relacionados con los IDE y el proceso de compilación. Se realiza un análisis de los IDE existentes para programar en lenguaje ensamblador. Finalmente se realiza el análisis y selección de la metodología, herramientas y tecnologías que mejor se adecuen a la solución propuesta.

1.1 Antecedentes

En la especialidad de Ingeniería Informática, es de gran importancia el dominio de los elementos de hardware de un computador. En la carrera de Ingeniería en Ciencias Informáticas esto se corresponde con una de las salidas del modelo del profesional, la de implantador y soporte técnico. Se espera que los egresados de esta especialidad sean capaces de ejercer como profesionales altamente calificados, y preparados para enfrentar los desafíos que la vida laboral les impone. Entre ellos están los problemas técnicos.

Para un mejor entendimiento del principio de funcionamiento de las computadoras desde el punto de vista técnico, es de vital importancia el dominio de un lenguaje de bajo nivel, ensamblador. Es por ello que en la UCI, desde sus inicios, se incorporó en el segundo año, la enseñanza del lenguaje ensamblador en las asignaturas de Máquinas Computadoras I y II, que luego evolucionaron a la asignatura de Arquitectura de Computadoras. La asignatura de Arquitectura de Computadoras tiene dentro de sus objetivos que los estudiantes sean capaces de implementar subrutinas de mediana y baja complejidad en lenguaje ensamblador.

En el año 2011, como parte de la estrategia de migración hacia el software libre adoptada por el país y por la universidad, se comienza a trabajar en la asignatura con herramientas libres y sobre el sistema operativo Linux. Esto determinó que se restringieran las herramientas a utilizar. Actualmente las herramientas que se utilizan en la asignatura dificultan la correcta asimilación del contenido por parte de los estudiantes. Ellas son: un editor de texto donde se escribe el código, el compilador FASM que funciona en Linux a nivel de consola, y finalmente la máquina virtual Qemu para la ejecución del programa. Todas ellas se trabajan a nivel de comandos en consola lo que provoca rechazo en los estudiantes, acostumbrados a utilizar herramientas con un IDE con grandes facilidades de interfaz gráfica.

Como parte de la constatación de los inconvenientes del uso de estas herramientas en la impartición de la asignatura en la Facultad 3, se aplicó una encuesta a un total de trece profesionales, que han impartido estas asignaturas y con elevada experiencia en este tema. El promedio de años de experiencia en la programación en ensamblador de los encuestados es de aproximadamente 9 años

y el nivel de conocimiento es 8. Los indicadores a los que se les evaluó el impacto sobre la asimilación del lenguaje ensamblador fueron (ver Anexo 1):

1. Se realiza sobre la consola de Linux
2. Es preciso dominar los comandos para la compilación y ejecución
3. El compilador sólo detecta un error por compilación
4. Sólo se da información de la línea del error
5. No hay distinción de sintaxis
6. No existe autocompletamiento de código
7. Codificación, compilación y ejecución del código engorrosos

Los resultados arrojados se muestran en la siguiente figura (ver Figura 1):

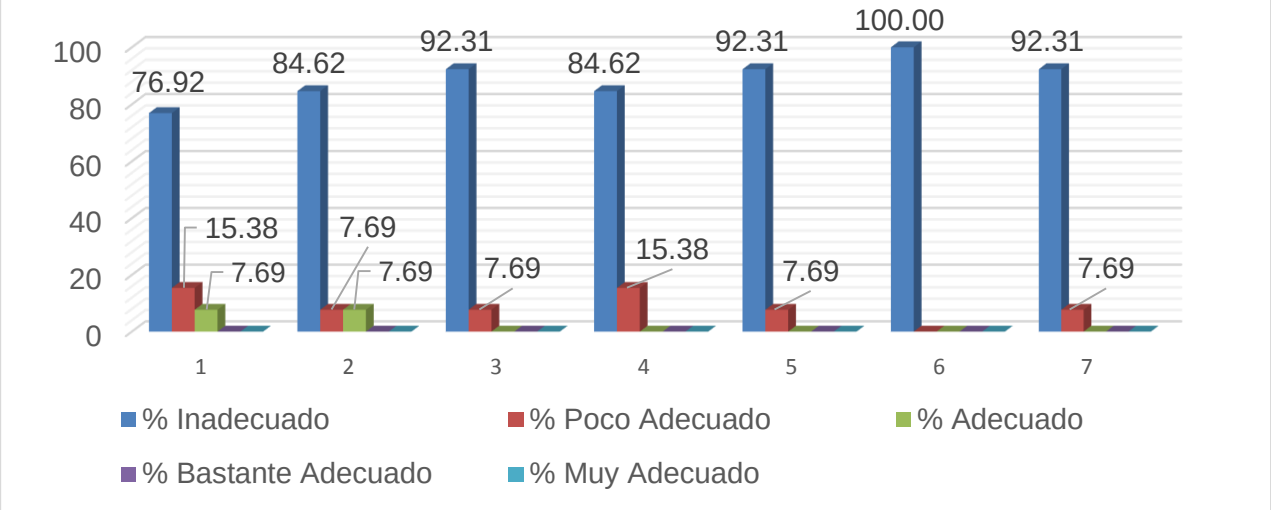

Figura 1: Resultados de la encuesta para valorar la necesidad de una nueva herramienta

Como se pudo constatar las herramientas actualmente utilizadas presentan aspectos que no favorecen un desarrollo adecuado de la habilidad de implementar en lenguaje ensamblador. De estos aspectos los que más influyen negativamente son el 6, el 3, el 5 y el 7, relacionados con las facilidades gráficas que brindan los IDE que usualmente los estudiantes utilizan, y a las que están acostumbrados. Por lo que se evidencia la necesidad de un sistema que dé respuestas a estas inconformidades e incorpore nuevas funcionalidades que puedan ser identificadas por el cliente.

1.2 Análisis de las soluciones existentes

En la actualidad se cuenta con diversos IDE para programar en ensamblador. Dentro de los más utilizados se encuentran:

Flat Assembler: es un rápido y eficiente auto ensamblador de 32 bits disponible para DOS, Windows y Linux. Actualmente soporta arquitectura de 32 y 64 bits, también soporta instrucciones

avanzadas como MMX, 3DNow!, SSE, SSE4, AVX, AVX2 y XOP. Puede producir una salida en los siguientes formatos: binario, MZ, PE, COFF o ELF. Incluye el potente y fácil uso de macros. Hace varias corridas para optimizar el código (Grysztar, 2015). Esta herramienta tiene como limitaciones que no destaca la sintaxis del lenguaje, sólo posee interfaz gráfica para el sistema operativo Windows, no muestra los errores en tiempo de codificación, en tiempo de compilación notifica un solo error, no autocompleta el código, y está orientado a trabajar con las APIs del sistema (Grysztar, 2010). A continuación se muestra una imagen donde se evidencian algunas de estas deficiencias (ver Figura 2).

Figura 2: IDE Flat Assembler

Assembler IDE: es un entorno de desarrollo para código ensamblador, cuyo propósito es automatizar al máximo este proceso, integrando el editor de código, el depurador y el desensamblador. Entre las posibilidades que ofrece el entorno Assembler IDE, destacan la depuración de código, el editor de código, el desensamblador y la compilación con NASM, TASM, MASM y FASM. Esta herramienta tiene como limitaciones que no destaca la sintaxis del lenguaje, sólo se ejecuta en el sistema operativo Windows, no muestra los errores en tiempo de codificación, en tiempo de compilación notifica un solo error, no autocompleta el código, y está orientado a trabajar con las APIs del sistema (Cnarte Ojeda, 2009). A continuación se muestra una imagen donde se evidencian algunas de estas deficiencias (ver Figura 3).

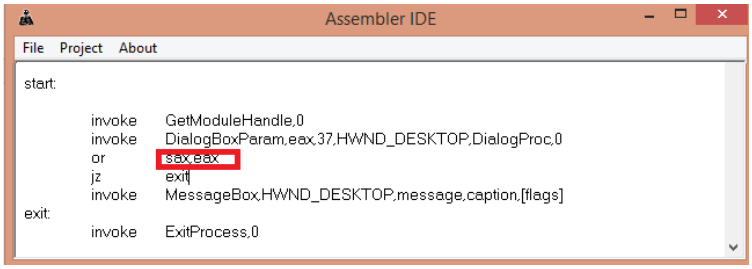

Figura 3: IDE Assembler IDE

Easy Code: es un entorno visual de programación en ensamblador hecho para generar aplicaciones Windows de 32 bits. La interfaz de Easy Code, muy parecida a la de Visual Basic, permite programar todo tipo de aplicaciones en ensamblador (archivos ejecutables, bibliotecas dinámicas y estáticas, archivos objeto COFF, aplicaciones de consola y drivers) de una manera rápida y fácil (Balderas Ortigosa, 2011). Cuenta con coloreado de sintaxis, funciones de depuración, constructor de interfaces gráficas de usuario, no requiere instalación y posee autocompletamiento, pero este no es inteligente. No muestra errores en tiempo de codificación (Found, 2015). A continuación se muestra una imagen, donde se evidencian las deficiencias antes mencionadas (ver Figura 4).

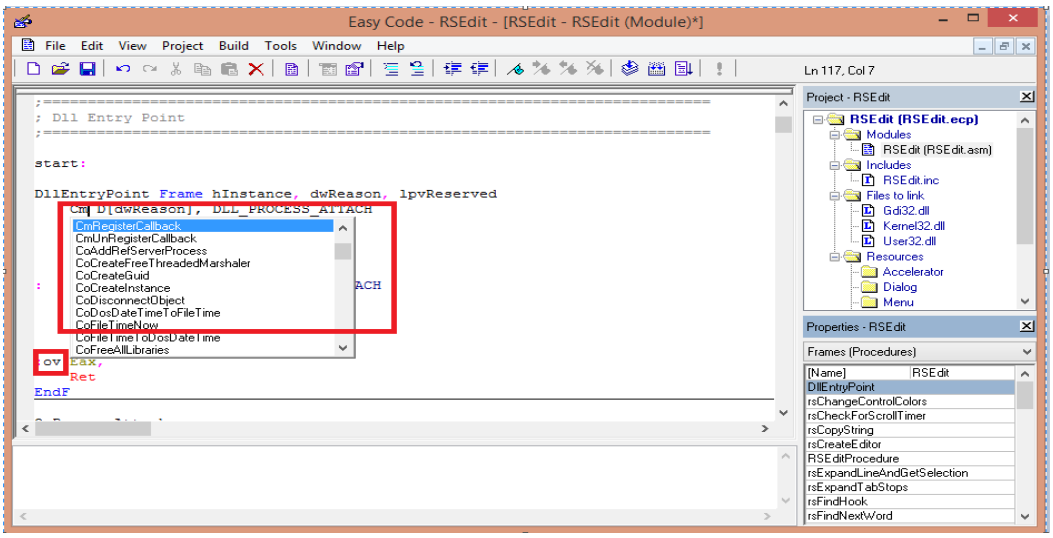

Figura 4: IDE Easy Code

Fresh IDE: es una continuación del proyecto FASM. Es perfectamente compatible con FASM. Sirve para generar código para cualquier plataforma que soporte FASM. Permite compilar, ejecutar y depurar aplicaciones con interfaz visual. Se ejecuta en Linux y Windows. Cuenta con coloreado de sintaxis. Posee autocompletamiento pero este no es inteligente (Found, 2015). A continuación se muestra una imagen, donde se evidencian las deficiencias antes mencionadas (ver Figura 5).

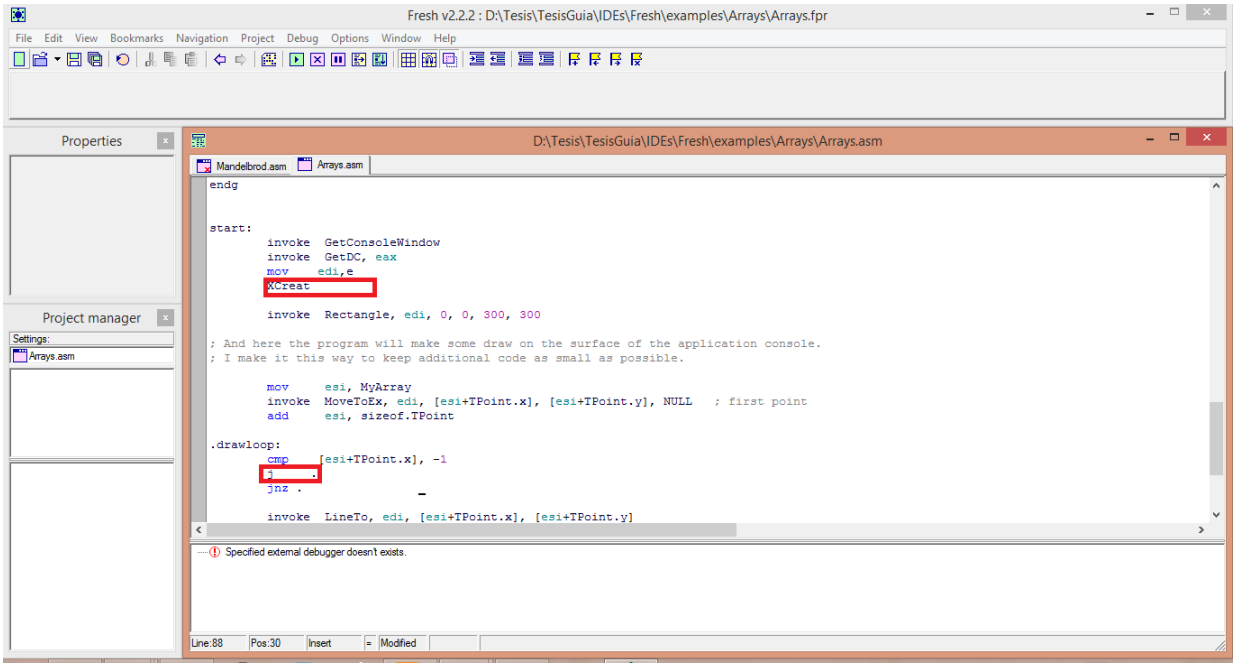

Figura 5: IDE Fresh IDE

Rad ASM: es un entorno de desarrollo para ensamblador de 32 bit para Windows. Tiene las siguientes características: resaltado de sintaxis, editor de recursos, comandos make, depuración integrada y personalización del IDE. Soporta los siguientes ensambladores: MASM, FASM, NASM, HLA y GoASM. Esta herramienta tiene como limitaciones que solo se ejecuta en Windows, no muestra los errores en tiempo de codificación, no autocompleta código y está orientada a usar las API del sistema operativo (Moreno, 2013). A continuación una imagen donde se evidencian algunas de estas desventajas (ver Figura 6).

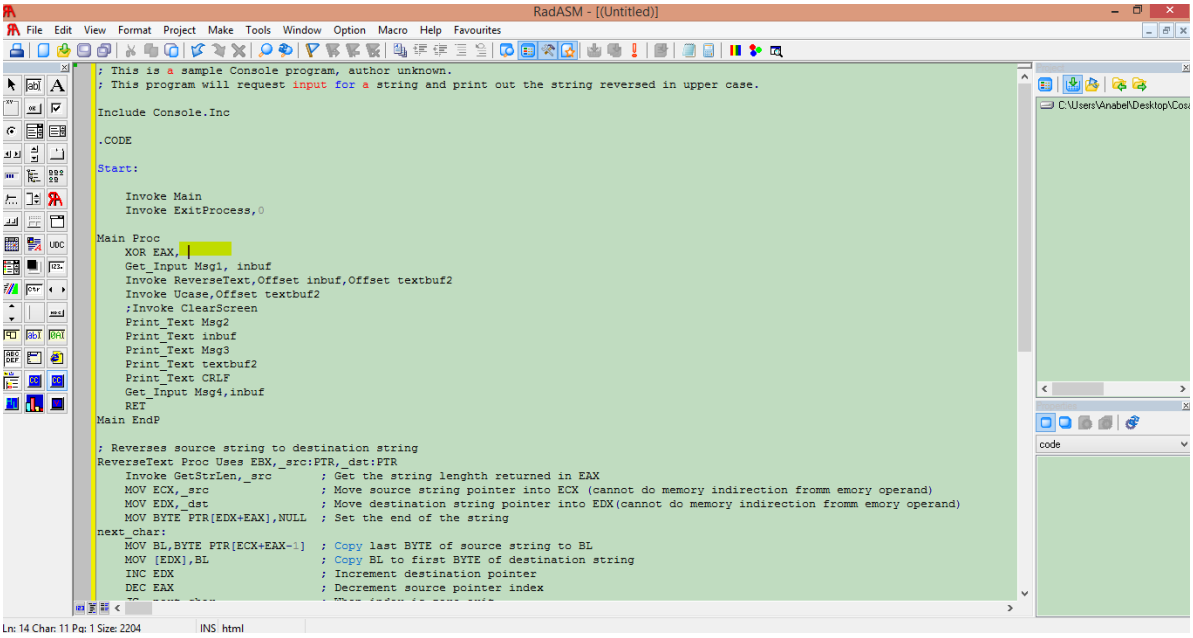

Figura 6: IDE RadASM

WinASM Studio: es un IDE para aplicaciones de 32 bits bajo Microsoft Windows y 16 bits bajo DOS, usando Microsoft Macro Assembler (MASM) y Flat Assembler (FASM). Soporta autocompletamiento para las funciones de las APIs de Windows y sus parámetros, lo que facilita el desarrollo rápido de aplicaciones (Navarrete, 2012). Es extensible mediante una completa interfaz de *plugins* (complemento), e incluye un poderoso editor visual. Posee una interfaz de usuario multilenguaje. Soporta los ensambladores MASM, POASM y FASM. Esta herramienta tiene como limitaciones que solo se ejecuta en Windows, no muestra los errores en tiempo de codificación, no resalta la sintaxis del lenguaje, no autocompleta código y está orientada a usar las APIs del sistema operativo (Cnarte Ojeda, 2009). A continuación una imagen donde se evidencian algunas de estas desventajas (ver Figura 7).

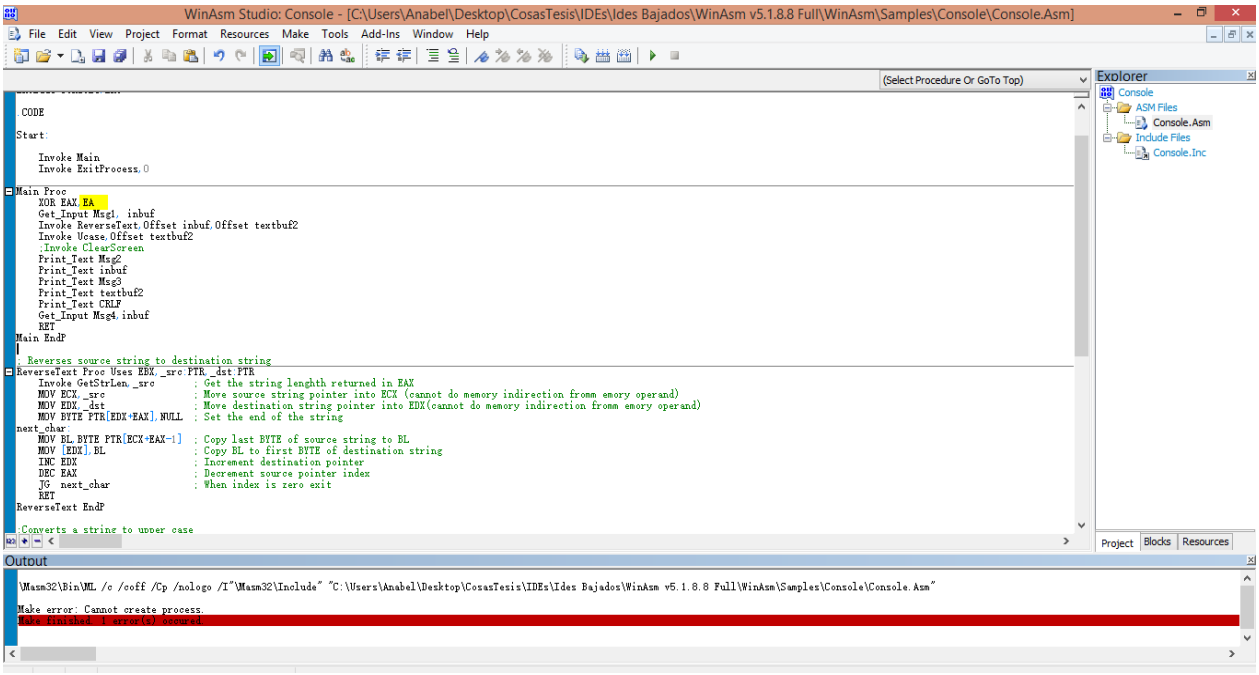

Figura 7: IDE WinASM Studio

SASM: es un IDE para programar en ensamblador que soporta FASM. Se ejecuta sobre el sistema operativo Linux. Esta herramienta posee las siguientes deficiencias: no resalta la sintaxis del lenguaje, está centrado en usar las APIs del sistema operativo, no muestra los errores en tiempo de codificación, y en la compilación muestra un solo error y no posee autocompletamiento de código (Manushin, 2015).

A continuación se hace un resumen de las características de interés que presentan los IDE antes analizados (ver Tabla 1):

Tabla 1: Resumen de las características de los IDE existentes para el lenguaje ensamblador

	Flat Assembler	Assembler IDE	Easy Code	Fresh IDE	Rad ASM	WinASM Studio	SASM
Coloreado de sintaxis	No	No	Si	Si	No	No	No
Ejecución en Linux	No	No	No	Se emula	No	No	Si
Centrado a usar las API del sistema operativo	Si	Si	Si	Si	Si	Si	Si
Mostrar errores en tiempo de codificación	No	No	No	No	No	No	No
Autocompletamiento Inteligente	No	No	No	No	No	No	No

Una vez analizadas las características de los IDE existentes para ensamblador, se concluye que ninguno de ellos da respuesta al problema planteado, pues en su mayoría se ejecutan en Windows. Los que se ejecutan en Linux presentan las desventajas de no autocompletar código y no mostrar los errores en tiempo de codificación.

1.3 Conceptos asociados al dominio del problema

A continuación se exponen los conceptos fundamentales relacionados con los IDE, principalmente los compiladores y las fases del proceso de compilación, puesto que el compilador es un componente fundamental de un IDE.

1.3.1 Proceso de compilación

Un **compilador** es un programa informático que traduce un programa escrito en un lenguaje de programación a otro lenguaje de programación, generando un programa equivalente que la máquina será capaz de interpretar. Usualmente el segundo lenguaje es lenguaje de máquina, pero también puede ser un código intermedio, o simplemente texto. Este proceso de traducción se conoce como **compilación** (Laborda, y otros, 1985)**.**

El **proceso de compilación** divide su labor en dos etapas: una que analiza la entrada y genera estructuras intermedias, y otra que sintetiza la salida a partir de dichas estructuras (Gálvez Rojas, y otros, 2005). A continuación se muestra una imagen de las etapas del proceso de compilación (ver Figura 8).

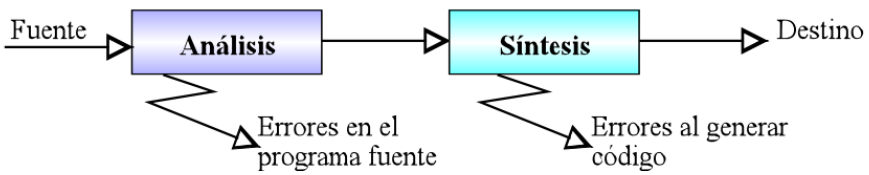

Figura 8: Etapas del proceso de compilación **(Acevedo Martínez, 2011)**

La etapa de análisis tiene como objetivos (Gálvez Rojas, y otros, 2005) **:**

- Controlar la corrección del programa fuente.
- Generar las estructuras necesarias para comenzar la etapa de síntesis.

Para llevar a cabo estos objetivos, la etapa de análisis se compone de las fases: análisis lexicográfico, análisis sintáctico y análisis semántico. A continuación se describen las fases antes mencionadas (Gálvez Rojas, y otros, 2005):

- **Análisis lexicográfico:** divide el programa fuente en los componentes básicos del lenguaje a compilar. Cada componente básico es una subsecuencia de caracteres del programa fuente, y pertenece a una categoría gramatical: números, identificadores de usuario (variables, constantes, tipos, nombres de procedimientos,…), palabras reservadas, signos de puntuación, etc.

  - **Funciones del analizador léxico:**

  El analizador léxico es la primera fase de un compilador. Su principal función consiste en leer los caracteres de entrada y elaborar como salida una secuencia de componentes léxicos que utiliza el analizador sintáctico para hacer el análisis.

  Ante un pedido de un componente léxico, el analizador léxico lee una secuencia de caracteres del código fuente del programa hasta identificar un componente léxico, el cual es retornado como respuesta al pedido del analizador sintáctico. A continuación se muestra la interrelación que existe entre el analizador léxico y el sintáctico (ver Figura 9):

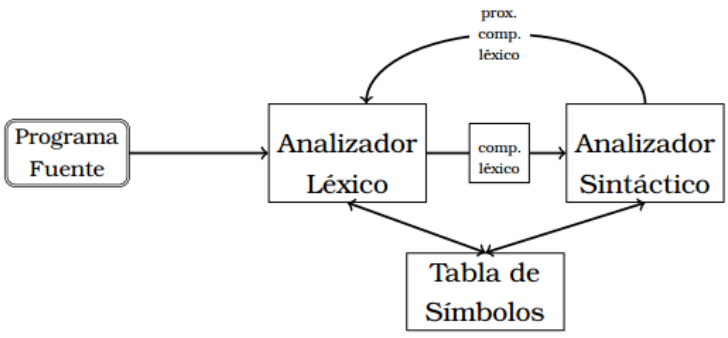

Figura 9: Interrelación entre el analizador léxico y el analizador sintáctico **(Acevedo Martínez, 2011)**

- **Análisis sintáctico:** es la fase del analizador que se encarga de chequear la secuencia de componentes léxicos que representa al texto de entrada, en base a una gramática dada. En caso de que el programa de entrada sea válido, suministra el árbol sintáctico que lo reconoce en base a una representación computacional. Este árbol es el punto de partida de la fase

posterior de la etapa de análisis: el analizador semántico. A continuación se muestra la interrelación que existe entre el analizador sintáctico y el analizador semántico (ver Figura 10):

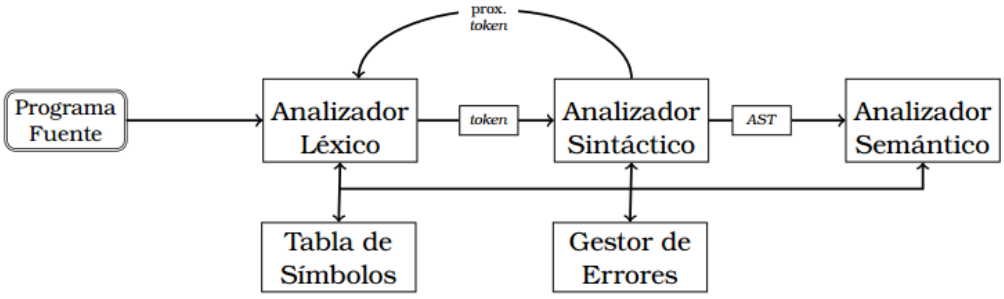

Figura 10: Interrelación entre los analizadores sintáctico y semántico **(Acevedo Martínez, 2011)**

- **Análisis semántico:** comprueba que el programa fuente respeta las directrices del lenguaje que se compila (todo lo relacionado con el significado): chequeo de tipos, rangos de valores, existencias de variables, etc. Es la fase encargada de detectar la validez semántica de las sentencias aceptadas por el analizador sintáctico.

En todas las fases descritas anteriormente está presente la **Tabla de Símbolos**, la cual consiste en una estructura de datos de alto rendimiento que almacena toda la información necesaria sobre los identificadores de usuario (Gálvez Rojas, y otros, 2005).

La tabla de símbolos almacena la información que en cada momento se necesita sobre las variables del programa; información tal como: nombre, tipo, dirección de localización en memoria, tamaño, etc. Una adecuada y eficaz gestión de la tabla de símbolos es muy importante, ya que su manipulación consume gran parte del tiempo de compilación. De ahí que su eficiencia sea crítica (Gálvez Rojas, y otros, 2005).

Otro de los elementos que se maneja en todas la fases del proceso de compilación es la **Gestión e información de errores**. Este componente se encarga de gestionar e informar los errores que tenga el código fuente, y un buen compilador debe ayudar al programador a localizar e identificar los errores. El analizador léxico detecta errores cuando los caracteres que restan de la entrada no forman un token[1] válido en el lenguaje. El analizador sintáctico se encarga de detectar las cadenas que no cumplen con las reglas estructurales del lenguaje. Durante la fase de análisis semántico el

[1]Un *token* es un par formado por un nombre de símbolo y un atributo opcional valor. El nombre del símbolo es una categoría que representa una especie de unidad léxica, por ejemplo, una palabra clave concreta, o una secuencia de caracteres de entrada que denota un identificador.

compilador trata de detectar estructuras con una sintaxis correcta pero incorrecta desde el punto de vista semántico de acuerdo a las operaciones involucradas (Gálvez Rojas, y otros, 2005).

1.3.2 Entorno de desarrollo integrado (IDE)

Un IDE es un programa compuesto por un conjunto de herramientas que proveen facilidades a los programadores para agilizar el proceso de desarrollo de software. Consta de un editor de código, un compilador, un depurador y un constructor de interfaz gráfica (Bell, y otros, 2003), (Matellan Olivera, 2004), (Rouse, 2007). Antiguamente los programadores tenían que editar los ficheros, salvarlos, correr el compilador, construir la aplicación y ejecutarla a través de un depurador. Actualmente un IDE integra en una única aplicación: editor, compilador, el depurador y herramientas para la administración de los proyectos. Todas estas posibilidades favorecen el aumento de la productividad del programador (Bolton, 2014).

Características que puede tener un IDE (Bolton, 2014)**:**

- Multiplataforma
- Soporte para diversos lenguajes de programación
- Integración con sistemas de control de versiones
- Reconocimiento de sintaxis
- Extensiones y componentes para el IDE
- Integración con marcos de trabajo populares
- Depurador
- Importar y exportar proyectos.
- Múltiples idiomas
- Manual de usuario y ayuda

1.4 Metodologías, lenguajes y herramientas de desarrollo

La elección de la metodología, lenguaje y herramientas de desarrollo es un proceso minucioso ya que de ello depende la calidad del desarrollo y del producto final. Para la elección se debe tener en cuenta las características del equipo de desarrollo y del producto en cuestión. El equipo de desarrollo es pequeño ya que está compuesto por un integrante y el cliente está en constante comunicación con el equipo de desarrollo. Se cuenta con poco tiempo para el desarrollo del sistema, máximo cinco meses. El sistema a desarrollar es pequeño y con pocos requisitos funcionales.

1.4.1 Metodología de desarrollo

La metodología de desarrollo de software guía el proceso de desarrollo para alcanzar la satisfacción del cliente y del equipo de desarrollo. Se debe seleccionar la metodología que mejor se adapte al

equipo de trabajo y al sistema a desarrollar. A continuación se hace un análisis para la selección de la metodología de desarrollo de software.

Una metodología es un conjunto integrado de técnicas y métodos que permite abordar de forma homogénea y abierta cada una de las actividades del ciclo de vida de un proyecto de desarrollo. Es un proceso de software detallado y completo (INTECO, 2009).

Según la filosofía de desarrollo se pueden clasificar las metodologías en dos grupos. Las metodologías tradicionales, que se basan en una fuerte planificación durante todo el desarrollo, y las metodologías ágiles, en las que el desarrollo de software es incremental, cooperativo, sencillo y adaptado (INTECO, 2009).

Las metodologías tradicionales centran su atención en llevar una documentación exhaustiva de todo el proyecto y en cumplir con un plan de proyecto, definido todo esto, en la fase inicial del desarrollo del proyecto. Se focalizan en la documentación, planificación y procesos (INTECO, 2009).

Las metodologías ágiles ponen de relevancia que la capacidad de respuesta a un cambio es más importante que el seguimiento estricto de un plan (INTECO, 2009).

Se opta por la utilización de un enfoque ágil debido a que el tiempo para la realización del sistema es corto, el cliente está disponible a tiempo completo, y el sistema a desarrollar es pequeño y cuenta con pocos requisitos funcionales.

Una vez seleccionado el enfoque, corresponde seleccionar la metodología. Para la selección de la metodología se analizan varios exponentes de la misma, Programación Extrema (XP), Scrum y Crystal, por ser de las más conocidas y populares. A continuación se muestra una tabla comparativa entre estás metodologías (ver Tabla 2).

Tabla 2: Comparación entre las principales metodologías ágiles **(Letelier, y otros, 2006)**.

Indicadores	**XP**	**Scrum**	**Crystal**
Participación del cliente	Activa, retroalimentación continua entre el cliente y el equipo de desarrollo.	En cada iteración	Activa
Pruebas	Ejecutadas constantemente ante cada modificación del sistema.	En cada iteración	Diariamente
Documentación	Se sustituye la documentación por la comunicación.	Poca	Poca
Centra la atención	Programadores	Gestión del proyecto	Programadores
Flexible para el equipo	Poco	Poco	Poco

Teniendo en cuenta la comparación anterior, se establece como metodología para guiar el desarrollo de la solución XP, por las siguientes razones:

- El cliente está en constante comunicación con el equipo de desarrollo.
- Las pruebas de unidad se realizan de manera periódica, lo que garantiza el correcto funcionamiento de la solución.
- El sistema es pequeño y con pocos requisitos funcionales, por lo que no es necesario generar mucha documentación.
- El equipo de desarrollo posee conocimientos de la metodología.
- Es fácil de aplicar.

Programación Extrema:

XP es una metodología ágil centrada en potenciar las relaciones interpersonales como clave para el éxito en el desarrollo de software, promoviendo el trabajo en equipo, preocupándose por el aprendizaje de los desarrolladores, y propiciando un buen clima de trabajo. Se basa en la realimentación continua entre el cliente y el equipo de desarrollo, comunicación fluida entre todos los participantes, simplicidad en las soluciones implementadas y coraje para enfrentar los cambios. Se define como especialmente adecuada para proyectos con requisitos imprecisos y muy cambiantes, y donde existe un alto riesgo técnico (INTECO, 2009).

La metodología XP define como artefacto primordial las **historias de usuario** que son la base del software a desarrollar. Estas historias son escritas por el cliente y describen las interrelaciones entre los usuarios y el sistema, y generalmente son complementadas con otro tipo de descripción. A partir de ellas y de la arquitectura que se utilizará, se planifican las entregas, así como los objetivos de cada una y las iteraciones con que contará (Alitimón, y otros, 2014).

Dentro de sus ventajas, se puede decir que la programación es más organizada, los programadores estarán satisfechos y la tasa de error disminuirá.

La Programación Extrema asume que la planificación nunca será perfecta y que los requisitos pueden cambiar a lo largo del ciclo de vida del software, a medida que varíen las necesidades del negocio.

1.4.2 Lenguaje de programación

El lenguaje de programación es un idioma artificial diseñado para expresar computaciones que pueden ser llevadas a cabo por máquinas como las computadoras. Pueden usarse para crear programas que controlen el comportamiento físico y lógico de una máquina, para expresar algoritmos con precisión (Ferrer Muñoz, y otros, 2013).

La elección del lenguaje de programación está dada por las características del producto a desarrollar. El lenguaje debe ser interpretado para garantizar que la herramienta sea portable. Dentro de los lenguajes interpretados más utilizados están Java y C#. Java posee las siguientes limitantes: las listas genéricas no están correctamente implementadas, no permite especificar parámetros opcionales , no dispone de *properties* (propiedades) lo cual influye en la legibilidad del código, no dispone de indexadores, carece de inicializadores de colección, carece de inicialización de objetos, no soporta parámetros de salida, y el consumo de recursos es elevado. Teniendo en cuenta las desventajas antes mencionadas, se decide usar como lenguaje de programación para implementar la solución C#, agregando que es un lenguaje orientado a objetos simple, elegante y con seguridad en el tratamiento de tipos. Posee gran robustez, gracias a la recolección de elementos no utilizados y a la seguridad en el tratamiento de tipos. Además, el equipo de desarrollo tiene experiencia en su utilización.

Características de C# (Microsoft, 2010)**:**

- C# es un lenguaje de programación simple pero eficaz, diseñado para escribir aplicaciones empresariales.
- El lenguaje C# es una evolución de los lenguajes C y C++. Utiliza muchas de las características de C++ en las áreas de instrucciones, expresiones y operadores.
- C# presenta considerables mejoras e innovaciones en áreas como seguridad de tipos, control de versiones, eventos y recolección de elementos no utilizados (liberación de memoria).
- C# proporciona acceso a los tipos de API más comunes: .NET Framework, COM, Automatización y estilo C. Además, admite el modo no seguro en el que se pueden utilizar punteros para manipular memoria que no se encuentra bajo el control del recolector de elementos no utilizados.

1.4.3 IDE para el desarrollo de la solución

Teniendo en cuenta que el lenguaje seleccionado para programar el sistema es C#, se procede a seleccionar el IDE para desarrollar la solución. Para definirlo se tuvieron en cuenta los IDE más usados para programar en C#. A continuación se analizan las características distintivas de cada uno de ellos:

MonoDevelop: es un IDE libre y gratuito, diseñado principalmente para programar en C# y otros lenguajes .NET. Permite desarrollar aplicaciones de escritorio y web en Linux, Windows y Mac OS. Dentro de las características más importantes se pueden citar (MonoDevelop, 2015):

- **Multiplataforma:** se ejecuta en Linux, Windows y Mac OS X.

- **Editor de texto avanzado:** brinda autocompletamiento de código para el lenguaje C#, brinda plantillas de código y permite plegar el código.
- **Personalizable:** es completamente personalizable, permite cambiar los atajos de teclado definidos y las herramientas externas.
- **Soporte para diversos lenguajes:** soporta los siguientes lenguajes: C#, F#, Visual Basic .NET, C/C++ y Vala.
- **Diseñador visual GTK#:** permite construir fácilmente aplicaciones GTK#.
- **Depurador integrado**: para depurar aplicaciones Mono y nativas.
- **Incorpora otras herramientas:** incorpora herramientas que permiten controlar el código y realizar pruebas de unidad.

SharpDevelop: es un IDE gratuito para aplicaciones .Net Framework. Dentro de las características más importantes se pueden citar (SharpDevelop, 2014):

- Soporta aplicaciones escritas en los lenguajes C, C#, Visual Basic.NET, Boo, ASP.NET, ADO.NET, XML y HTML.
- Proporciona todas las características demandadas para un entorno de programación de Windows, como el autocompletado de código, plantillas de proyecto, depurador integrado o diseñador de formularios, entre otras.
- SharpDevelop es compatible con Visual Studio Express y Visual Studio 2005, emplea el mismo tipo de formato para los ficheros de proyecto y código fuente.
- Resalta la sintaxis de los lenguajes C#, C, HTML, ASP, ASP.NET, VBScript, VB.NET y XML.

Visual Studio Community 2013: es una versión libre de Visual Studio. Se puede usar en un ambiente docente, para la investigación académica, o para contribuir a proyectos de código abierto. Incluye todas las funcionalidades de Visual Studio Professional 2013, diseñado y optimizado para los desarrolladores individuales, estudiantes, colaboradores de código abierto y los equipos pequeños (Microsoft, 2015). Dentro de las características más importantes se pueden señalar (Microsoft, 2015):

- **Entorno de desarrollo:** se centra en crear valor y realizar el trabajo con un entorno de desarrollo limpio, rápido y con un gran potencial.
    - Pantalla informativa CodeLens[2]
    - Entorno muy completo
    - Desarrollo de aplicaciones empresariales

[2] Es una característica que muestra información sobre el código (autor, cambios efectuados, resultados de las pruebas de unidad, referencias, etc) directamente en el editor de código.

- **Compatibilidad con plataformas:** compila aplicaciones dirigidas a plataformas de Microsoft, así como aplicaciones web, móviles, servicios web y servicios en la nube para diferentes dispositivos.
    - Aplicaciones Windows
    - Aplicaciones web y servicios en la nube
    - Aplicaciones de producción
- **Desarrollo de software ágil:** adopta gradualmente los procedimientos ágiles recomendados que mejor se adapten al equipo y al plan, administra y supervisa el progreso de diferentes equipos y registros de trabajos pendientes.
    - Administración ágil de tareas
    - Gráficos de evolución
- **Colaboración en equipo:** conecta el equipo de desarrollo, las partes interesadas y los usuarios finales a través de herramientas integradas que favorecen la colaboración.
    - Salones de equipo
    - Administración de comentarios
    - Repositorios basados en Git
- **Depuración y diagnóstico:** identifica y soluciona problemas que impiden que la aplicación se ejecute correctamente, independientemente de la plataforma.
    - Depurador avanzado
    - Browser Link
    - IntelliTrace
- **Herramientas para pruebas de software:** herramientas para pruebas avanzadas que garantizan la calidad a lo largo del ciclo de vida de una aplicación, lo que permite obtener software de alta calidad.
    - Administración de casos de prueba
    - Pruebas de rendimiento web
    - Pruebas de carga
- **Manejo de versiones finales:** configura, planea, aprueba e implementa las aplicaciones para cualquier entorno con herramientas que reducen el ciclo de desarrollo y mejoran el proceso de entrega.
    - Administra una secuencia coherente
    - Define procesos de producción de versiones

Una vez realizado el análisis de los IDE se optó por el Visual Studio Community 2013 teniendo en cuenta que es gratuito, posee una alta eficiencia en la comprobación y autocompletamiento del código, permite diseñar formularios de forma sencilla e intuitiva. Además brinda soporte para la realización de las pruebas de software y herramientas para aplicar métricas al código.

Adicionalmente es el IDE en el que el equipo de desarrollo tiene mayor experiencia lo que tributa a minimizar el tiempo de desarrollo del software por este concepto.

1.4.4 Control de versiones

El control de versiones es el proceso de almacenar cambios de un proyecto de desarrollo. La utilización de un control de versiones tiene gran importancia debido a que permite viajar en la línea de tiempo del proyecto. Conservan todas las versiones que se generen de un proyecto a través del tiempo y lleva el control del autor de cada cambio.

La utilización de un control de versiones brinda grandes beneficios, tales como:

- Cualquier versión almacenada puede ser recuperada para visualizarla, compararla con otras versiones o modificarla.
- Se puede observar las diferencias entre las versiones almacenadas.
- Pueden trabajar en el mismo proyecto diferentes desarrolladores, sin riesgo de que se pierda información.
- Permite dividir los proyectos en varias ramas y después fusionarlas en una única versión.

Se decide usar como herramienta para el control de versiones Git, debido a que se integra con el IDE seleccionado para implementar la solución (Visual Studio Community 2013). Git es un controlador de versiones libre y de código abierto, diseñado para manejar proyectos grandes y pequeños con velocidad y eficiencia.

Características de Git (Git, 2014)**:**

- Permite la creación de nuevas ramas y la unión de ramas.
- Es pequeño y rápido.
- Es distribuido.
- Es seguro.
- Es libre y de código abierto.

1.5 Conclusiones del capítulo

A partir de la encuesta realizada a trece profesionales de alta experiencia en la programación en ensamblador, se pudieron constatar los impactos negativos del uso de las herramientas definidas por la universidad, para el desarrollo de las habilidades de implementación en ensamblador de los estudiantes. Por estas razones, se hace necesario el desarrollo de un IDE que facilite la impartición de la asignatura, y la asimilación por parte de los estudiantes de los contenidos de la misma.

Teniendo en cuenta las limitaciones identificadas, se realizó un análisis de los IDE más utilizados, y se constató que ninguno da respuesta al problema planteado. Por lo que es necesario el desarrollo de un nuevo IDE que satisfaga estos requisitos. Además se definieron las funcionalidades de estos IDE a incorporar en el nuevo sistema.

Para dicho desarrollo se establecieron los fundamentos teóricos asociados a los compiladores y a los IDE, en función de identificar las tendencias actuales en este ámbito así como las funcionalidades y características a incorporar en la nueva herramienta.

Considerando las características del equipo de desarrollo y las particularidades de la solución que se pretende obtener, y en vista de que el tiempo con el que se cuenta para entregar la propuesta de solución es corto, quedó evidenciado que una metodología ágil es la más adecuada para esta investigación. Realizando un análisis de las metodologías ágiles más utilizadas se escogió XP, debido a los beneficios que brinda al ser fácil de estudiar y aplicar, y mantiene una estrecha comunicación con el cliente lo que posibilita la obtención de un producto con calidad.

Una vez determinada la metodología a utilizar, se definieron, como lenguaje a utilizar C# porque es robusto y permite la obtención de aplicaciones portables, como IDE Visual Studio Community 2013 por las facilidades que brinda y como control de versiones Git porque se integra con el IDE seleccionado para desarrollar el sistema.

CAPÍTULO 2: PROPUESTA DE SOLUCIÓN

Partiendo de la metodología seleccionada, el presente capítulo refleja cada una de las etapas del desarrollo de la solución. Se explican y documentan cada uno de los artefactos generados que dan soporte a la especificación de sus funcionalidades, la arquitectura, el modelo de diseño, así como el uso de patrones y estándares de codificación.

2.1 Descripción del sistema

La solución propuesta está diseñada para los estudiantes y profesores de la asignatura Arquitectura de Computadoras. El objetivo de esta herramienta es brindar un IDE para programar en lenguaje ensamblador, que brinde las facilidades de coloreo de sintaxis, autocompletamiento de código, detección de errores en tiempo de codificación y la integración de los procesos de compilación y ejecución del código. El IDE está centrado en el repertorio de instrucciones que se utiliza comúnmente en la asignatura de Arquitectura de Computadoras (ver Anexo 2).

2.2 Ingeniería de requisitos

La ingeniería de requisitos cobra vital importancia cuando se quiere que los sistemas informáticos se ajusten perfectamente a las necesidades del cliente. Los requisitos representan las características y funcionalidades que el sistema debe poseer.

2.2.1 Requisitos Funcionales

Los requisitos funcionales son declaraciones de los servicios que debe proporcionar el sistema, la forma en que debe reaccionar ante ciertas entradas y cómo se debe comportar en situaciones particulares (Sommerville, 2005). Se identificaron los siguientes requisitos funcionales:

- **RF-01** Abrir programa.
- **RF-02** Guardar programa.
- **RF-03** Realizar análisis léxico.
- **RF-04** Realizar análisis sintáctico.
- **RF-05** Realizar análisis semántico.
- **RF-06** Integrar la compilación y ejecución del código.
- **RF-07** Mostrar errores en tiempo de codificación.
- **RF-08** Colorear la sintaxis del lenguaje.
- **RF-09** Autocompletar código.

2.2.2 Requisitos No Funcionales

Los requisitos no funcionales son restricciones de los servicios o funciones ofrecidos por el sistema. Se aplican al sistema en su totalidad y no se refieren directamente a funciones específicas, sino a sus propiedades emergentes; la fiabilidad, el tiempo de respuesta y la capacidad de almacenamiento son algunas de ellas. Pueden estar dirigidos incluso al proceso de desarrollo, especificando herramientas o estándares de calidad a emplear en función de las necesidades del usuario (Sommerville, 2005). A continuación los requisitos no funcionales identificados para la propuesta de solución:

- **Requisitos de hardware:**
  - **RNF-01** Recursos tecnológicos
  1. 256 MB de RAM como mínimo
  2. Microprocesador de 1,66Ghz
  3. Espacio libre en el disco duro de 12 MB
- **Requisitos de software:**
  - **RNF-02** Entorno de ejecución

  Para la ejecución del IDE son indispensable las siguientes instalaciones:
  1. Para los usuarios Linux: Framework Mono
  2. Para los usuarios Windows: Framework .Net o Mono
  3. Compilador FASM
  4. Máquina virtual Qemu para Linux y Qemu Manager para Windows.
- **Requisitos de interfaz:**
  - **RNF-03** Interfaz de usuario
  1. La interfaz debe ser amigable, intuitiva y fácil de operar.

2.2.2.1 Especificación de requisitos funcionales

La especificación de requisitos establece la base para el acuerdo entre usuarios y desarrolladores de software, quedando definido el comportamiento deseado del producto (IEEE, 2004).

En la metodología XP las historias de usuario (HU) son el artefacto que se utiliza para especificar los requisitos funcionales del sistema. A continuación se describen los campos que se deben llenar en una HU y se muestran las HU obtenidas (ver Tablas de la 3 a la 11):

- **Número:** identificador de la HU
- **Nombre:** nombre que identifica a la HU
- **Usuario:** involucrados en la ejecución de la HU
- **Iteración asignada:** iteración en que se implementará la HU.

- **Prioridad en el negocio:** grado de prioridad que le asigna el cliente a la HU en dependencia del valor en el negocio. Los valores que puede tomar son alta, media o baja.
- **Riesgo en desarrollo:** grado de complejidad que le asigna el equipo de desarrollo a la HU luego de analizarla (alto, medio o bajo).
- **Puntos estimados:** estima el esfuerzo asociado a la implementación de la HU.
- **Puntos reales:** resultado del esfuerzo asociado a la implementación de la HU.
- **Descripción:** descripción sintetizada de la HU
- **Observaciones:** información de interés

Tabla 3: HU Abrir Programa.

Historia de usuario	
Número: 1	**Nombre:** Abrir programa
Usuario: estudiantes y profesores	**Iteración asignada:** 2
Prioridad en el negocio: alta	**Puntos estimados:** 3 días
Riesgo en desarrollo: bajo	**Puntos reales:** 3 días
Descripción: permite abrir un programa en el IDE.	
Observaciones:	

Tabla 4: HU Guardar Programa.

Historia de usuario	
Número: 2	**Nombre:** Guardar programa
Usuario: estudiantes y profesores	**Iteración asignada:** 2
Prioridad en el negocio: alta	**Puntos estimados:** 1 día
Riesgo en desarrollo: bajo	**Puntos reales:** 1 día
Descripción: permite guardar un programa.	
Observaciones:	

Tabla 5: HU Análisis Léxico

Historia de usuario	
Número: 3	**Nombre:** Análisis Léxico
Usuario: estudiantes y profesores	**Iteración asignada:** 1
Prioridad en el negocio: alta	**Puntos estimados:** 3 días
Riesgo en desarrollo: medio	**Puntos reales:** 3 días
Descripción: permite realizar el análisis léxico del programa, además de ir gestionando los errores léxicos.	
Observaciones:	

Tabla 6: HU Análisis Sintáctico

Historia de usuario	
Número: 4	**Nombre:** Análisis Sintáctico
Usuario: estudiantes y profesores	**Iteración asignada:** 1
Prioridad en el negocio: alta	**Puntos estimados:** 30 días
Riesgo en desarrollo: alto	**Puntos reales:** 30 días
Descripción: permite verificar que el programa cumple con las reglas gramaticales del lenguaje ensamblador, además gestiona los errores sintácticos.	
Observaciones:	

Tabla 7: HU Análisis Semántico

Historia de usuario	
Número: 5	**Nombre:** Análisis Semántico
Usuario: estudiantes y profesores	**Iteración asignada:** 1
Prioridad en el negocio: alta	**Puntos estimados:** 10 días
Riesgo en desarrollo: alto	**Puntos reales:** 10 días
Descripción: permite chequear que el programa cumple con las reglas semánticas del lenguaje ensamblador, además gestiona los errores semánticos.	
Observaciones:	

Tabla 8: HU Integración de la compilación y la ejecución del código

Historia de usuario	
Número: 6	**Nombre:** Integración de la compilación y la ejecución del código
Usuario: estudiantes y profesores	**Iteración asignada:** 2
Prioridad en el negocio: alta	**Puntos estimados:** 1 día
Riesgo en desarrollo: bajo	**Puntos reales:** 1 día
Descripción: permite integrar la compilación y ejecución del código en un mismo paso.	
Observaciones:	

Tabla 9: HU Mostrar errores

Historia de usuario	
Número: 7	**Nombre:** Mostrar errores
Usuario: estudiantes y profesores	**Iteración asignada:** 2
Prioridad en el negocio: alta	**Puntos estimados:** 1 día
Riesgo en desarrollo: bajo	**Puntos reales:** 1 día
Descripción: permite mostrar los errores en tiempo de codificación.	
Observaciones:	

Tabla 10: HU Colorear Sintaxis

Historia de usuario	
Número: 8	**Nombre:** Colorear Sintaxis
Usuario: estudiantes y profesores	**Iteración asignada:** 2
Prioridad en el negocio: alta	**Puntos estimados:** 3 días
Riesgo en desarrollo: medio	**Puntos reales:** 3 días
Descripción: permite resaltar la sintaxis del lenguaje ensamblador.	
Observaciones:	

Tabla 11: HU Autocompletar código

Historia de usuario	
Número: 9	**Nombre:** Autocompletar código
Usuario: estudiantes y profesores	**Iteración asignada:** 2
Prioridad en el negocio: alta	**Puntos estimados:** 7 días
Riesgo en desarrollo: medio	**Puntos reales:** 7 días
Descripción: permite autocompletar código de forma inteligente.	
Observaciones:	

2.3 Planificación

En esta etapa los clientes establecen la prioridad de las HU y los programadores estiman el esfuerzo necesario para implementar cada una de ellas. La estimación del esfuerzo utiliza como medida el punto. En la estimación realizada a continuación un punto equivale a un día de desarrollo (ver Tabla 12).

Tabla 12: Duración de las HU

Número	Historia de Usuario	Puntos de estimación (días)
1	Abrir programa	3
2	Guardar programa	1
3	Análisis Léxico	3
4	Análisis Sintáctico	30
5	Análisis Semántico	10
6	Integración de la compilación y la ejecución del código	1
7	Mostrar errores	1
8	Colorear Sintaxis	3
9	Autocompletar código	7

2.3.1 Plan de iteraciones

Una vez identificadas las HU y de realizada una estimación del esfuerzo requerido para la implementación de cada una, se realiza el plan de iteraciones. Este plan especifica las HU que serán implementadas en cada iteración del desarrollo del sistema. Se determinaron dos iteraciones para la realización del sistema.

- **Iteración 1:** esta iteración tiene como objetivo realizar la fase de análisis del IDE, para ello se implementan las HU: 3, 4 y 5.
- **Iteración 2:** esta iteración tiene como objetivo implementar todo lo relacionado con las facilidades de interfaz gráfica del IDE, para ello se implementan las HU: 1, 2, 6, 7, 8 y 9.

2.3.2 Plan de duración de las iteraciones

Este plan muestra las HU en el orden que se implementan en las iteraciones y muestra el tiempo de cada iteración (ver Tabla 13).

Tabla 13: Plan de duración de las iteraciones

Iteración	Historias de Usuario	Duración de la iteración (días)
1	1. Análisis Léxico 2. Análisis Sintáctico 3. Análisis Semántico	43
2	1. Abrir Programa. 2. Guardar Programa. 3. Integración de la compilación y la ejecución del código. 4. Mostrar errores. 5. Colorear Sintaxis. 6. Autocompletar código.	16

2.3.3 Plan de entregas

El plan de entregas específica las fechas de inicio y fin de cada iteración y las HU que deben estar terminadas al finalizar la misma (ver Tabla 14).

Tabla 14: Plan de entregas

Iteración	Historias de Usuario a entregar	Fecha de Inicio	Fecha Fin
1	HU 3, 4 y 5	15 de marzo de 2015	27 de abril de 2015
2	HU 1, 2, 6, 7, 8 y 9	28 de abril de 2015	13 de mayo de 2015

2.4 Arquitectura de la solución

La arquitectura de un sistema es un marco conceptual completo que describe su forma y estructura (sus componentes y la manera en que se integran) (Pressman, 2005). Para lograr la calidad del software es necesario establecer desde el inicio una arquitectura que garantice robustez y escalabilidad.

El diseño arquitectónico del sistema está basado en el **patrón arquitectónico Tuberías y Filtros** (ver Figura 11). Este patrón se aplica cuando los datos de entrada se transforman en datos de salida mediante una serie de componentes para el cálculo o la manipulación. Una estructura tuberías y filtros tiene un conjunto de componentes denominados filtros, conectados por tuberías que transmiten datos de un componente al siguiente. Los filtros están diseñados para esperar la entrada de datos con cierta forma y producir una salida de una forma específica. Sin embargo, no es necesario que los filtros conozcan el funcionamiento de los filtros vecinos (Pressman, 2005). Los

compiladores son el ejemplo clásico de este patrón, de ahí su selección como patrón arquitectónico para la solución.

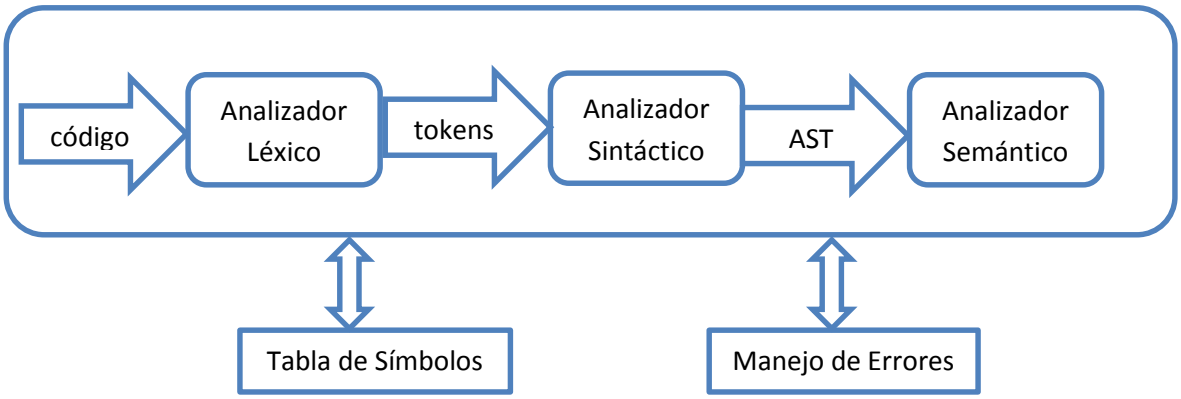

Figura 11: Arquitectura de la solución

En la herramienta los filtros son: Analizador Léxico, Analizador Sintáctico y Analizador Semántico, y las tuberías son el mecanismo para comunicar un filtro con el siguiente. El filtro que comunica el Analizador Léxico y el Analizador Sintáctico es el método que devuelve un token, y el filtro que comunica el Analizador Sintáctico y el Analizador Semántico es el método que devuelve el AST. La Tabla de Símbolos y el Manejo de Errores son componentes que utilizan todos los filtros para realizar sus responsabilidades. A continuación se describen los componentes del sistema y las clases que los componen:

Analizador Léxico: encargado de tomar como entrada el programa fuente y convertirlo en la secuencia de tokens. Está compuesto por las siguientes clases:

- **Kind.cs:** es un enumerativo que denota el tipo de token definido.
- **Lexer.cs:** es la clase encarga de realizar el análisis léxico, su función principal es devolver el listado de tokens y para ello se auxilia del resto de las clases del Analizador Léxico.
- **Location.cs:** tiene como propósito almacenar la línea en la que aparece el token en el código fuente. Es muy útil para ubicar con precisión los errores.
- **Reader.cs:** es la encargada de manipular el fichero fuente, es la clase que debe mantener un contador de la línea actual en el código fuente del último carácter retornado, así como tomaºr un carácter cada vez que le sea requerido por el Lexer.
- **State.cs:** es un enumerativo que representa estados intermediarios de un token.
- **Token.cs:** es la clase que se encarga de representar un token.

Tabla de símbolos: encargada de almacenar información sobre los nombres de los identificadores usados en el programa fuente. Está compuesto por las siguientes clases:

- **Symbol.cs:** es la clase encargada de representar un símbolo.

- **SymbolsTable.cs:** es la clase que se encarga de gestionar los símbolos almacenados en la tabla de símbolos.

Analizador Sintáctico: es el encargado de comprobar la sintaxis del programa fuente y construye el árbol de sintaxis abstracta. Está compuesto por la siguiente clase:

- **Parser.cs:** es la clase encargada de realizar el análisis sintáctico y de generar el AST.

Analizador Semántico: utiliza información del árbol sintáctico y de la tabla de símbolos para comprobar la validez semántica del programa. Está compuesto por la siguiente clase:

- **Checker.cs:** es la clase encargada de chequear las reglas semánticas.

Manejo de errores: encargado de manejar los errores detectados en la etapa de análisis. Está compuesto por las siguientes clases:

- **Msg.cs:** clase encargada de almacenar los mensajes de error.
- **HandledException.cs:** es una excepción personalizada, para llevar el control de las excepciones lanzadas por el IDE.
- **ErrorManager.cs:** clase encargada de almacenar todos los errores que se detectan en el programa fuente.

2.5 Modelo de diseño

La metodología XP en lugar de utilizar diagramas de clases para representar el diseño utiliza las tarjetas CRC (Clase- Responsabilidad- Colaboración). Cada tarjeta representa una clase del sistema y define sus responsabilidades (lo que conoce y realiza) y las colaboraciones (clases con las que se relaciona para llevar a cabo sus responsabilidades) (Pressman, 2005). A continuación las tarjetas CRC correspondientes a las clases principales del IDE (ver Tablas de la 15 a la 20):

Tabla 15: Tarjeta CRC Lexer.cs

Nombre de la clase: Lexer.cs	
Responsabilidades:	**Colaboradores:**
- Devolver el listado de tokens. - Devolver un token.	- Reader.cs - Token.cs - ErrorManager.cs

Tabla 16: Tarjeta CRC SymbolsTable.cs

Nombre de la clase: SymbolsTable.cs	
Responsabilidades:	**Colaboradores:**
- Adicionar símbolo. - Modificar símbolo.	- Symbol.cs

Tabla 17: Tarjeta CRC Parser.cs

Nombre de la clase: Parser.cs	
Responsabilidades:	**Colaboradores:**
- Verificar el cumplimiento de la sintaxis del lenguaje. - Generar el AST.	- Lexer.cs - ErrorManager.cs - SymbolsTable.cs

Tabla 18: Tarjeta CRC Checker.cs

Nombre de la clase: Checker.cs	
Responsabilidades:	**Colaboradores:**
-Verificar que el programa sea correcto semánticamente.	- AST.cs - ErrorManager.cs - SymbolsTable.cs

Tabla 19: Tarjeta CRC ErrorManager.cs

Nombre de la clase: ErrorManager.cs	
Responsabilidades:	**Colaboradores:**
- Almacenar los errores que contiene el programa fuente.	- Error.cs

Tabla 20: Tarjeta CRC Compiler.cs

Nombre de la clase: Compiler.cs	
Responsabilidades:	**Colaboradores:**
- Llevar a cabo la compilación del código.	- Lexer.cs - Parser.cs - ErrorManager.cs

2.6 Patrones de diseño

Un patrón es una descripción de un problema y la solución, a la que se da un nombre, y que se puede aplicar a nuevos contextos. Idealmente proporciona consejos sobre el modo de aplicarlo en varias circunstancias, y considera los puntos fuertes y compromisos (Larman, 2003).

Los patrones de diseño constituyen soluciones genéricas probadas a problemas comunes del desarrollo de software. Proporcionan flexibilidad, elegancia y reutilización. Para lograr un buen diseño se hace uso de los siguientes patrones:

- **Patrones GRASP (General Responsibility Assignment Software Patterns)**

Los patrones GRASP describen los principios fundamentales del diseño de objetos y la asignación
de responsabilidades, expresados como patrones (Larman, 2003).

Experto en Información (Experto): consiste en asignar una responsabilidad al experto en información (la clase que tiene la información necesaria para realizar la responsabilidad) (Larman, 2003). Aplicando este patrón se obtiene un diseño con mayor cohesión y la información se mantiene encapsulada disminuyendo el acoplamiento.

Este patrón se evidencia en todas las clases del sistema, porque cada responsabilidad fue asignada a la clase que maneja la información para llevarla a cabo. Un ejemplo concreto se evidencia en la clase Reader.cs, que es la responsable de conformar los tokens porque es la que contiene los caracteres del programa fuente. A continuación se muestra la imagen de la clase Reader.cs, donde el atributo source es el que contiene los caracteres del programa fuente y el método Token() se encarga de conformar los tokens agrupando los caracteres del programa fuente (ver Figura 12).

Creador: asignar a la clase B la responsabilidad de crear una instancia de clase A si se cumple uno o más de los casos siguientes (Larman, 2003):

- B agrega objetos de A.
- B contiene objetos de A.
- B registra instancias de objetos de A.
- B utiliza más estrechamente objetos de A.
- B tiene los datos de inicialización que se pasarán a un objeto de A cuando sea creado (por tanto, B es un Experto con respecto a la creación de A).

B es un creador de los objetos A.

Uno de los beneficios del uso de este patrón es el bajo acoplamiento. El uso de este patrón se evidencia en la clase Reader, pues contiene todos los datos de inicialización que se pasarán cuando se crea un Token. En la Figura 12 se puede observar que la clase Reader tiene como atributos el lexema y la locación, que son los datos para inicializar un token (ver Figura 12).

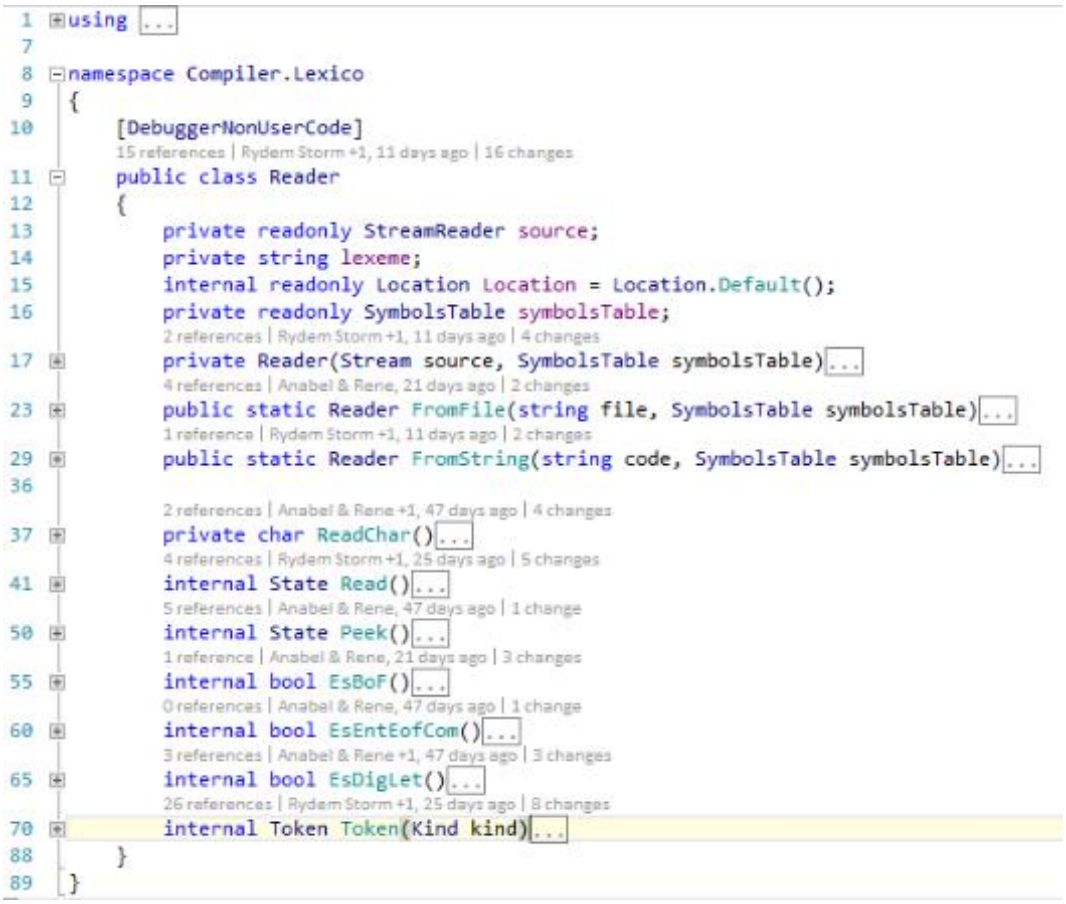

Figura 12: Ejemplo de los patrones experto y creador

Bajo Acoplamiento: el **acoplamiento** es una medida de la fuerza con que un elemento está conectado a, tiene conocimiento de, confía en, otros elementos. Un elemento con bajo (o débil) acoplamiento no depende de demasiados otros elementos (Larman, 2003). El uso de este patrón proporciona los siguientes beneficios: no afectan los cambios en otros componentes, fácil de entender de manera aislada y conveniente para reutilizar (Larman, 2003). Este patrón se refleja en la propia arquitectura del sistema, que se diseñó de forma tal que las dependencias sean solo las necesarias, además no existe herencias profundas. Además el sistema hace uso de los patrones experto y creador, y según Larman con el uso de estos patrones se garantiza el bajo acoplamiento (Larman, 2003).

Alta Cohesión: una clase tiene una responsabilidad moderada en un área funcional y colabora con otras clases para llevar a cabo las tareas (Larman, 2003). Cada elemento del diseño debe realizar una labor única dentro del sistema, no desempeñada por el resto de los elementos. El

sistema usa el patrón experto y según Larman el uso de este patrón aumenta la cohesión (Larman, 2003). Este patrón se refleja en la clase Compiler.cs que es la responsable de realizar la compilación, pero para desarrollar esta tarea necesita de la colaboración de las clases Lexer.cs, Parser.cs y Checker.cs fundamentalmente (ver Figura 13).

Controlador: asignar la responsabilidad de recibir o manejar un mensaje de evento del sistema a una clase que representa una de las siguientes opciones (Larman, 2003):

- Representa el sistema global, dispositivo o subsistema (controlador de fachada).
- Representa un escenario de caso de uso en el que tiene lugar el evento del sistema.

Es un intermediario entre la interfaz y las clases que la implementan de tal forma que es el que recibe los datos del usuario y el que los envía a las distintas clases según el método llamado. De esa forma la lógica del negocio está separada de la presentación. Este patrón se evidencia en la clase Compiler.cs ya que actúa como controladora del sistema, controlando el flujo de información y los eventos del sistema (ver Figura 13).

```
1   using ...
14
15  namespace Compiler
16  {
        8 references | Anabel & Rene +1, 12 days ago | 13 changes
17      public static class Compiler
18      {
19  #if linux
20  ...
22  #else
23          private static string qemu = @"C:\Program Files (x86)\QemuManager\qemu\qemu.exe";
24          private static string fasm = @"c:\Program Files (x86)\Fasm\FASM.EXE";
25  #endif
26          private static Regex file = new Regex(@"(?<file>.*) \[");
27          private static Regex line = new Regex(@"\[(?<line>\d+)\]");
28          private static Regex desc = new Regex(@"error: (?<desc>.*)\.");
            9 references | Rydem Storm, 30 days ago | 1 change
29          public static SymbolsTable SymbolsTable { get; private set; }
            6 references | Rydem Storm, 30 days ago | 1 change
30          public static Reader Reader { get; private set; }
            6 references | Rydem Storm, 30 days ago | 1 change
31          public static Lexer Lexer { get; private set; }
            2 references | Rydem Storm, 30 days ago | 1 change
32          public static Parser Parser { get; private set; }
            0 references | Rydem Storm, 30 days ago | 1 change
33          public static Checker Checker { get; private set; }
34
            1 reference | Rydem Storm, 24 days ago | 1 change
35          public static IEnumerable<Token> Load(string code)...
            2 references | Anabel & Rene +1, 14 days ago | 3 changes
42          public static IEnumerable<Token> LoadLine(string code)...
            1 reference | Rydem Storm, 16 days ago | 3 changes
49          public static void Chequear(string code)...
            3 references | Anabel & Rene +1, 12 days ago | 7 changes
59          public static bool Compilar(string code)...
            1 reference | Rydem Storm, 30 days ago | 1 change
94          private static void Borrar(string file)...
            1 reference | Rydem Storm, 30 days ago | 2 changes
99          public static bool Ejecutar(string code)...
            0 references | Anabel & Rene, 21 days ago | 1 change
118         public static string GetLexeme(this int index)...
122     }
123 }
```

Figura 13: Ejemplo de los patrones alta cohesión y controlador

- **Patrones GoF (Gang of Four)**

Los patrones GoF se clasifican en tres categorías según su propósito: creacionales, estructurales y de comportamiento (Gamma, y otros, 1994):

- **Creacionales:** abstraen el proceso de creación de instancias y ocultan los detalles de cómo los objetos son creados.
- **Estructurales:** se ocupan de cómo las clases y objetos se combinan para formar grandes estructuras y proporcionar nuevas funcionalidades.
- **De comportamiento:** ofrecen soluciones respecto a la interacción y responsabilidades entre clases y objetos, así como los algoritmos que encapsulan.

En la solución se utiliza el patrón **Visitor** (Visitante)**,** que es un tipo de patrón de comportamiento, para realizar el chequeo semántico del lenguaje. Este patrón es capaz de modelar distintas operaciones a llevar a cabo sobre una misma estructura de datos, permitiendo definir nuevas operaciones sin modificar la estructura del diseño (Gamma, y otros, 1994).

2.7 Codificación

En esta etapa se dividen las HU en Tareas de Ingeniería, con el objetivo de establecer las tareas necesarias para implementar cada HU. Es necesario establecer un estándar de codificación para lograr la homogeneidad y mantenibilidad del código.

2.7.1 Estándares de codificación

En aras de lograr que el código esté organizado, sea sencillo de entender, sea fácil de mantener y mantenga la homogeneidad en la implementación, se definió un conjunto de pautas a seguir durante la codificación.

- **Cabecera de archivo:** todos los archivos deben comenzar con una cabecera que especifique el autor y fecha de creación, adicionalmente se podrán incluir comentarios u otros datos de interés (ver Figura 14).

```
//
// Autor: Anabel González Valiente
// Fecha Creación: 08/03/2015
```

Figura 14: Cabecera de Archivo

- **Definición de clases:** las clases serán colocadas en archivos independientes que solo contendrán el código de la misma. El nombre de las clases iniciará con letras mayúsculas y de poseer más de una palabra, la primera letra de cada una deberá ser también mayúscula (ver Figura 15).

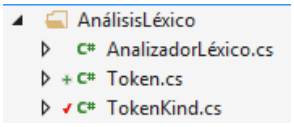

Figura 15: Definición de clases

- **Definición de variables y constantes:** los nombres de las variables y constantes deben ser descriptivos y concisos. No se usarán grandes frases ni abreviaciones. Los nombres de las variables iniciarán con letra minúscula y cada nueva palabra debe iniciar con mayúscula. Las constantes se escribirán en su totalidad con mayúsculas, usando un guion bajo para la separación de palabras (ver Figura 16).

```
private TokenKind kind;
private string lexeme;
private SourcePosition position;
private int entry;
```

Figura 16: Definición de variables y constantes

- **Definición de funciones:** los nombres de las funciones deben dar una idea del objetivo con el que fueron hechas. Los nombres iniciarán con letra mayúscula y cada nueva palabra debe iniciar con mayúscula (ver Figura 17).

```
private bool IsDigit()
{
    return char.IsDigit(currentChar);
}
```

Figura 17: Definición de Funciones

- **Uso de llaves en bloques de instrucciones:** todo bloque de instrucciones debe ir entre llaves, aun en los casos que no sean necesario (ver Figura 18).

```
if (keywordsTable.TryGetValue(lexeme, out kind))
     {
    return kind;
     }
```

Figura 18: Uso de llaves en bloques de instrucciones

- **Posición de las llaves en bloques de instrucciones:** las llaves de apertura se colocarán debajo de la sentencia que delimita el bloque de instrucciones, las de cierre se alinean con las de apertura (ver Figura 19).

```
private void FillKeywordsTable()
{
    keywordsTable.Add("float", TokenKind.Float);
    keywordsTable.Add("int", TokenKind.Int);
    keywordsTable.Add("write", TokenKind.Write);
}
```

Figura 19: Posición de las llaves en bloques de instrucciones

- **Indentación:** la indentación debe ser a cuatro espacios sin caracteres de tabulación, se debe realizar con tabulaciones, por tanto, se debe fijar éste en cuatro caracteres que es la opción por defecto del entorno Visual Studio Community 2013 (ver Figura 20).

```
private bool IsSeparator()
{
    switch (currentChar)
    {
        case ' ':
        case '\n':
        case '\r':
        case '\t':
            return true;
        default:
            return false;
    }
}
```

Figura 20: Identación

2.7.2 Tareas de Ingeniería

Las Tareas de Ingeniería (TI) se utilizan para describir las tareas que se deben realizar para implementar el sistema. Estas tareas se relacionan con las HU. Al terminar la asignación de tareas por HU se obtuvieron un total de 14 TI. A continuación se detallan las tareas que se deben realizar para la HU 1 (ver Tablas de la 21 a la 23), las restantes se encuentran disponibles en los anexos (ver Anexos del 3 al 13):

Tabla 21: TI Construir la interfaz principal del IDE.

Tarea de Ingeniería	
Número: 1	**Nombre Historia de Usuario:** Abrir Programa
Nombre de la Tarea: Construir la interfaz principal del IDE.	
Tipo de tarea: desarrollo **(desarrollo/corrección/mejora)**	**Puntos Estimados(días):** 1
Fecha inicio: 28/04/2015	**Fecha fin:** 28/04/2015
Programador responsable: Anabel González Valiente	
Descripción: en esta tarea se construirá la interfaz principal del sistema.	

Tabla 22: TI Comunicar la interfaz principal del IDE con la clase controladora del sistema.

Tarea de Ingeniería	
Número: 2	**Nombre Historia de Usuario:** Abrir Programa
Nombre de la Tarea: Comunicar la interfaz principal del IDE con la clase controladora del sistema.	
Tipo de tarea: desarrollo **(desarrollo/corrección/mejora)**	**Puntos Estimados(días):** 1
Fecha inicio: 29/04/2015	**Fecha fin:** 29/04/2015
Programador responsable: Anabel González Valiente	
Descripción: en esta tarea se implementará la comunicación entre la interfaz principal del IDE y la clase controladora del sistema.	

Tabla 23: TI Implementar las funciones principales de la interfaz principal del IDE

Tarea de Ingeniería	
Número: 3	**Nombre Historia de Usuario:** Abrir Programa
Nombre de la Tarea: Implementar las funciones principales de la interfaz principal del IDE	
Tipo de tarea: desarrollo **(desarrollo/corrección/mejora)**	**Puntos Estimados(días):** 1
Fecha inicio: 30/04/2015	**Fecha fin:** 30/04/2015
Programador responsable: Anabel González Valiente	
Descripción: en esta tarea se implementarán todas las funciones de la interfaz principal, principalmente abrir programa.	

2.7.3 Tokens, diagramas de transición y gramática de la solución

Para la implementación del IDE fue necesario identificar los tokens del lenguaje ensamblador soportados por la herramienta, y el establecimiento del mecanismo para su reconocimiento. Por último se procede a definir la gramática.

Identificación de los tokens: los tokens identificados son los que aparecen a continuación:

Identificador	*Peso*	*Cadena*
Separador	*PesoDoble*	*Coma*
Comentario	*Type*	*Mas*
Retorno	*Size*	*Menos*
Desconocido	*Register*	*Por*
Directive	*Instruction*	*Division*
Asignacion	*Numero*	*ParentAbre*

ParentCierra	*LlaveAbre*	*Macro*
DosPuntos	*LlaveCierra*	*Constante*
CorAbre	*Etiqueta*	*Logica*
CorCierra	*Variable*	
Eof	*Jump*	

Mecanismo para el reconocimiento de los tokens: el mecanismo que se utiliza para el reconocimiento de los tokens es el diagrama de transición. Los diagramas de transición describen las acciones que sucederán cuando el analizador léxico, ante un pedido del analizador sintáctico, va en busca del próximo token. En el diagrama de transición se realiza un traslado de una posición a otra a medida que se van leyendo los caracteres (Acevedo Martínez, 2011). A continuación se muestran los diagramas de transición correspondientes al reconocimiento de los tokens *Identificador, Cadena, Separador, Mas, Menos, Por* y *Division* (ver Figuras de la 21 a la 23).

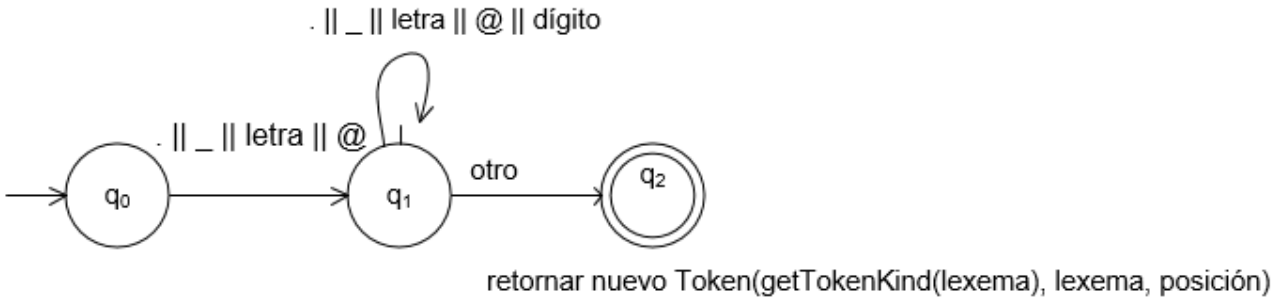

Figura 21: Diagrama de transición para reconocer los identificadores.

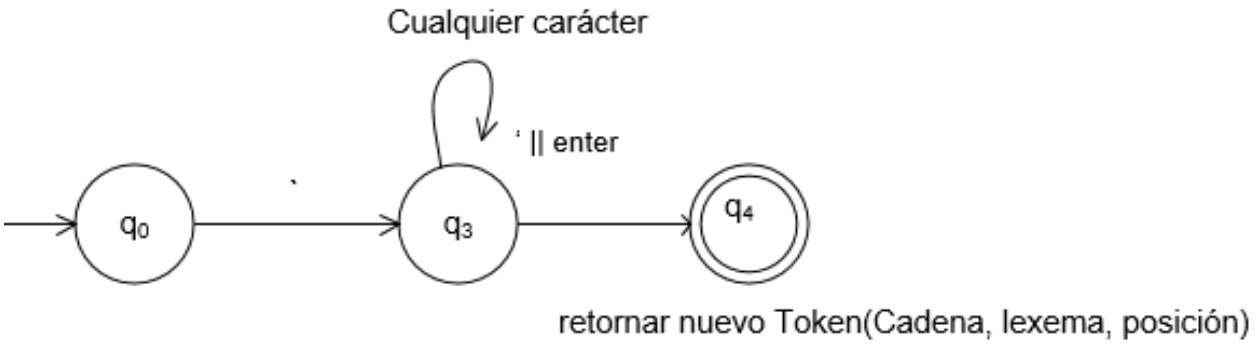

Figura 22: Diagrama de transición para reconocer las cadenas.

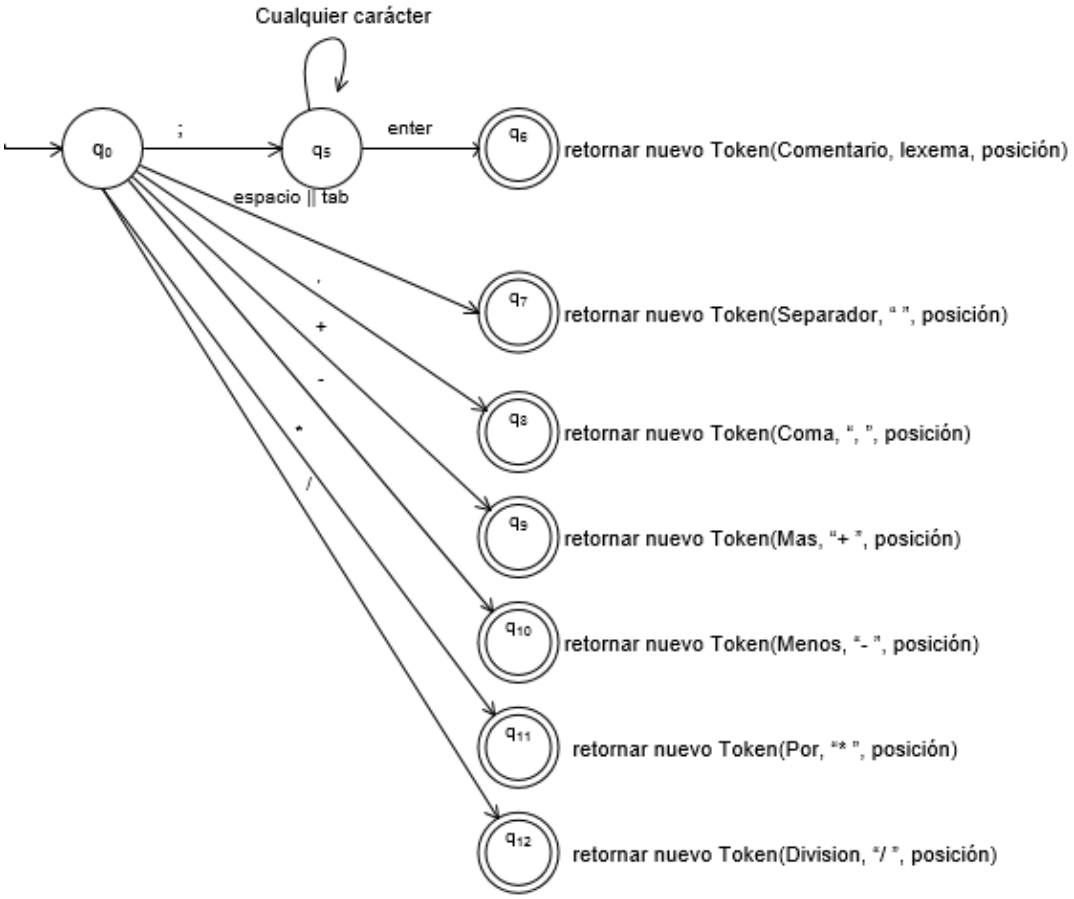

Figura 23: Diagrama de transición para reconocer los tokens Comentario, Separador, Coma, Mas, Menos, Por y Division.

Gramática: es el conjunto de reglas que definen la sintaxis. A continuación se muestran algunas de estas reglas (ver Tabla 23):

Tabla 23: Fragmento de gramática

Fragmento de gramática:
<programa> -> <body>
<body> -> <lista_sentencias>
< lista_sentencias > -> <sentencia> < lista_sentencias> | <sentencia>
<sentencia> -> <sentencia_declaracion_var> | <sentencia_declaracion_const> | <sentencia_instruccion> | <sentencia_ directiva>
<sentencia_declaracion_var> -> Identificador DosPuntos Type <list_valores> | Identificador Type <list_valores>
<valor> -> Cadena | Numero
<list_valores> -> <valor> | <valor> Coma <list_valores>
<sentencia_declaracion_const> -> Identificador Asignacion <valor>

<sentencia_instruccion> -> Instruction | Instruction1 <operando>| Instruction2 <operando> Coma <operando>
<operando> -> Register | Variable | CorAbre Register CorCierra | CorAbre Variable CorCierra

2.8 Conclusiones del capítulo

Con el presente, quedaron definidos los elementos fundamentales para la construcción del IDE. Como parte de la fase de planificación se elaboró el plan de iteraciones y el plan de entregas, los cuales facilitaron la planificación y organización en el proceso de desarrollo.

Se definieron en total nueve requisitos funcionales y tres no funcionales para describir las características y requerimientos del sistema, lo que contribuyó a la comprensión y visualización de los requisitos del cliente. A partir de la obtención de los requisitos funcionales se modelaron, a través de nueve HU, las funcionalidades del sistema.

Se definió como patrón arquitectónico “Tuberías y Filtros”, pues es el que mejor se ajusta a las características del sistema. Por otro lado, se describieron las clases presentes en la solución con el objetivo de facilitar su comprensión.

Como parte de la fase de codificación se desglosaron las HU en TI, para lograr una mayor especificidad de las tareas a realizar para obtener el sistema. Se definió un estándar de codificación para promover una mayor organización y mantenibilidad. Además, se describieron los tokens implementados, se realizaron algunos diagramas de transición y se plasmó un fragmento de la gramática utilizada para permitir una mayor comprensión de la solución.

CAPÍTULO 3: VALIDACIÓN DE LA PROPUESTA DE SOLUCIÓN

En el presente capítulo se realiza la verificación y validación de la propuesta de solución. Se validan los requisitos funcionales y se aplican métricas para garantizar la calidad del producto desarrollado. Como parte de la estrategia de pruebas se verifica el sistema mediante pruebas de unidad y se valida el sistema mediante pruebas de aceptación. Por último se comprueba el cumplimiento del objetivo general mediante el juicio de expertos.

3.1 Validación de requisitos funcionales

La validación de requisitos examina la especificación para asegurar que todos los requisitos de software se han establecido de manera precisa. Certifica la correspondencia de los requisitos con las necesidades del cliente y los usuarios. Este proceso cobra vital importancia y se debe hacer antes de comenzar a desarrollar el software para no correr riesgos en la implementación. A continuación se describen las técnicas y métricas utilizadas para validar los requisitos:

Construcción de prototipos: en esta técnica se hace una representación aproximada de la interfaz de usuario. Esto facilita que los clientes y usuarios entiendan la propuesta de los desarrolladores, y de esta forma aprueben la propuesta.

Esta técnica se aplicó diseñando el prototipo de la ventana principal del IDE para que el cliente entendiera de qué forma iban a quedar las interfaces y la visualización de las funcionalidades. Para ello se realizaron varios encuentros con el cliente donde se evaluaron las propuestas de prototipo hasta que quedó finalmente ajustado a los requisitos deseados. El prototipo aprobado es el siguiente (ver Figura 24):

Figura 24: Prototipo de la interfaz de usuario

Métricas aplicadas para validar los requisitos:

Para validar los requisitos de la solución propuesta se decide usar las siguientes métricas (Pressman, 2005):

- Especificidad de los requisitos
- Grado de validación de los requisitos

Aplicación de la métrica especificidad (falta de ambigüedad) de los requisitos:

Esta métrica evalúa la consistencia en la interpretación de los revisores de cada requisito. La especificidad se calcula como:

$$Q_1=\frac{N_{ui}}{N_r}$$

Donde:

Q_1= Especificidad de los requisitos
N_{ui}= Número de requisitos que todos los revisores interpretaron de la misma manera
N_r= Total de requisitos definidos
Mientras más cercano esté a 1 mayor será la consistencia de la interpretación de los revisores, y menor será la ambigüedad en los requisitos.

Aplicando la métrica se obtuvo como resultado, Q_1=9/9=1. Como el valor es 1 se puede interpretar que los requisitos no son ambiguos.

Aplicación de la métrica grado de validación de los requisitos:

Esta métrica mide la corrección de la definición de los requisitos. La validación se calcula como:

$Q_3=N_c/(N_c+N_{nv})$

Donde:

Q_3= Grado de validación de los requisitos.
N_c= Total de requisitos validados correctamente.
N_{nv}= Total de requisitos no validados
Mientras más cercano esté el resultado a 1, mayor será la corrección de los requisitos.

Aplicando la métrica se obtuvo como resultado, Q_3=9/(9+0)= 1. Como el resultado es 1 se puede interpretar que la definición de los requisitos es correcta.

3.2 Métricas de calidad del software

Los objetivos principales de las métricas orientadas a objetos son (García Sánchez, 2010):

- Comprender mejor la calidad del producto.
- Estimar la efectividad del proceso.
- Mejorar la calidad del trabajo en el nivel del proyecto.

Se sabe que la clase es la unidad principal de todo sistema orientado a objetos. Por consiguiente, las medidas y métricas para una clase individual, la jerarquía de clases, y las colaboraciones de clases, resultarán sumamente valiosas para un ingeniero de software que tenga que estimar la calidad de un diseño (García Sánchez, 2010). A continuación se describen las métricas que se utilizan:

Métricas CK (Chidamber y Kemerer):

Son métricas orientadas a clases: clases individuales, herencia y colaboraciones. Es uno de los conjuntos de métricas más referenciado (García Sánchez, 2010). Dentro de este tipo de métricas se usan:

- **Profundidad en el árbol de herencia (DIT: Depth Inheritance Tree):** es la distancia desde una clase a la raíz del árbol de herencia. Cuanto más alto es el mayor valor de DIT, mayor complejidad hay en el diseño, y cuanto más alto sea el valor de DIT de una clase más posibilidades existen de que reutilice/refine métodos heredados. Se prefiere que el valor de esta métrica esté entre 0 y 4 (Sharma, 2014).

  Las principales interpretaciones de esta métrica son las siguientes (García Sánchez, 2010):

  - A mayor profundidad de la clase, más métodos puede heredar y es más difícil de explicar su comportamiento.
  - A mayor profundidad de una clase, mayor posibilidad de reutilización de métodos heredados.
- **Acoplamiento entre clases:** es el número de clases acopladas a una clase. Dos clases están acopladas cuando los métodos de una de ellas usan variables o métodos de una instancia de la otra clase. Si existen varias dependencias sobre una misma clase es computada como una sola. No es deseable que el acoplamiento sea mayor que 14. Cuanto más alto es el acoplamiento más difícil será el mantenimiento y la reutilización, y en general el código será más propenso a fallos.

  Las interpretaciones de esta métrica son las siguientes (García Sánchez, 2010):

  - Cuanto mayor es el acoplamiento, peor es la modularidad y la reutilización.

- Cuanto mayor es el acoplamiento, peor es el encapsulamiento y más cuesta mantenerlo.

Complejidad Ciclomática: proporciona una medición cuantitativa de la complejidad lógica de un programa como el número de caminos independientes dentro de un fragmento de código. En general los resultados se interpretan dentro de estos rangos: una complejidad ciclomática de 1 a 10 es un programa simple sin mucho riesgo, de 10 a 20 es un riesgo más complejo, de 21 a 50, muy complejo, un programa de alto riesgo y más de 50, programa no evaluable (ya que la métrica indica el número mínimo de caminos independientes que han de ejecutarse (pruebas) para estar seguros de haber ejecutado al menos una vez todas las sentencias de un programa y todas las condiciones lógicas en sus vertientes verdaderas y falsas) (García Sánchez, 2010).

Para la aplicación de las métricas anteriores se hizo uso de la herramienta Code Metrics que viene integrada en el IDE Visual Studio Community 2013. El uso de esta herramienta es una buena práctica que permite hacer aplicaciones más mantenibles y menos complejas (Sharma, 2014). Además de las métricas descritas anteriormente, la herramienta calcula la siguiente métrica:

Índice de Mantenimiento: esta métrica indica cuán mantenible es el código. Un valor alto de esta métrica, indica mayor mantenibilidad del código. Los valores de esta métrica van de 0 a 100. De acuerdo con Microsoft un valor mayor que 20 se puede considerar bueno, un valor entre 10 y 19 indica una mantenibilidad moderada y un valor menor o igual que 9 indica una baja mantenibilidad (Sharma, 2014).

A continuación se muestra una imagen con los resultados de las métricas anteriormente descritas utilizando Code Metrics (ver Figura 25):

Code Metrics Results

Filter: None Min: Max:

Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling
SymbolsTable	56	69	1	17
Symbol	82	18	1	2
{ } Compiler.Sintactico	80	175	1	59
ParserExt	75	29	1	15
Parser	64	146	1	57
Esperado	100	0	1	0
{ } Compiler.Semantico	58	80	1	48
Checker	58	80	1	48
{ } Compiler.Reserved	69	65	1	14
ReservedTable	50	39	1	14
ReservedInfo	88	26	1	1
{ } Compiler.Lexico	78	171	1	24
Token	80	18	1	5
State	100	0	1	0
Reader	66	29	1	17
Location	89	11	1	1
LexerExt	54	40	1	4
Lexer	55	73	1	11
Kind	100	0	1	0
{ } Compiler.Errors	82	20	3	14
Msg	52	1	1	1
Message	90	6	1	2
HandledException	92	1	3	2
ErrorManager	88	8	1	9
Error	90	4	2	7

Figura 25: Resultados obtenidos en la herramienta Code Metrics

Escalas aplicadas a los valores de las métricas:

A continuación se muestran las escalas aplicadas al valor de cada métrica (ver Tabla 24):

Tabla 24: Escala aplicada a cada métrica

Índice de Mantenimiento	Alta	>= 20
	Media	Entre 10 y 19
	Baja	<= 9
Complejidad Ciclomática	Alta	Entre 1 y 10
	Media	Entre 11 y 20
	Baja	Mayor que 20
Profundidad de Herencia	Alta	>=9
	Media	Entre 5 y 8
	Baja	<= 4
Acoplamiento	Alta	>10
	Media	Entre 6 y 10
	Baja	<=6

Resultados de las métricas e interpretación de los resultados:

Luego de obtener el valor de las métricas y llevarlos a escala se obtuvieron los siguientes resultados (ver Figura 26):

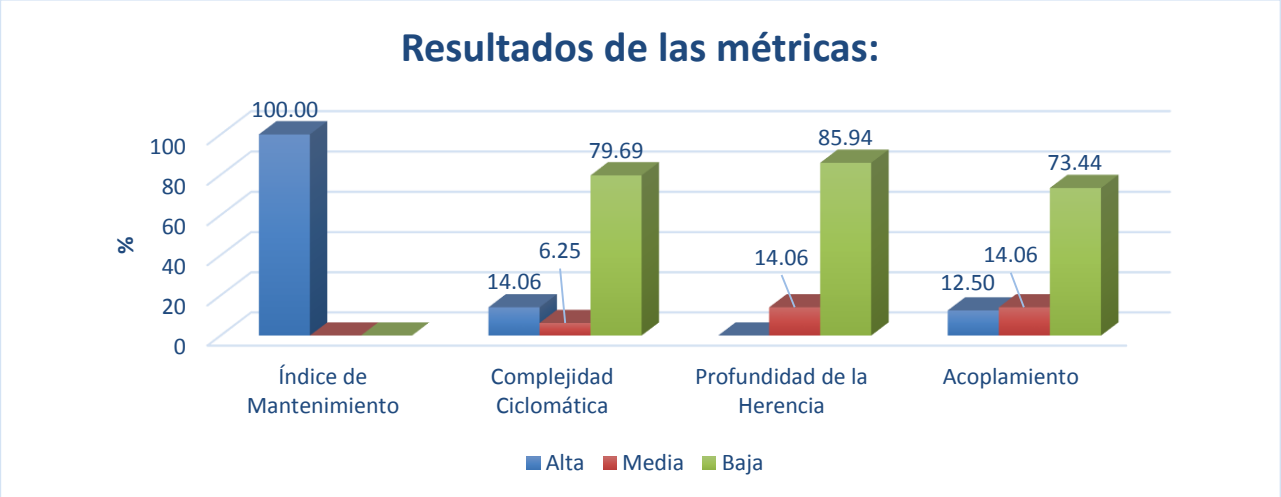

Figura 26: Resultados de las métricas

Como se puede observar en la gráfica anterior el 100 % de las clases del sistema tienen un alto grado de mantenimiento, el 79.69 % de las clases poseen una complejidad ciclomática baja, el 85.94 % de las clases tienen una profundidad de herencia baja y el 73.44 % de las clases poseen bajo

acoplamiento. A partir de los resultados anteriores, se puede concluir que el sistema es fácil de mantener y aplicarle pruebas.

3.3 Pruebas realizadas al sistema

Una prueba de software es la ejecución de programas de software con el objetivo de detectar defectos y fallas. Permite comprobar y mostrar la calidad de un producto de software (Pressman, 2005).

Las pruebas tienen los siguientes objetivos (Pressman, 2005):

1. La prueba es el proceso de ejecución de un programa con la intención de descubrir un error.
2. Un buen caso de prueba es aquel que tiene una alta probabilidad de mostrar un error no descubierto hasta entonces.
3. Una prueba tiene éxito si descubre un error no detectado hasta entonces.

La metodología XP divide las pruebas en dos grupos: pruebas unitarias, desarrolladas por los programadores, encargadas de verificar el código de forma automática, y las pruebas de aceptación, destinadas a evaluar si al final de una iteración se obtuvo la funcionalidad requerida, además de comprobar que dicha funcionalidad sea la esperada por el cliente (Beck, y otros, 2000).

Las pruebas constituyen un elemento primordial en la metodología XP y deben aplicarse de forma temprana y frecuente.

Para la realización de las pruebas se establece una estrategia de pruebas. Para la verificación del sistema se usan las pruebas unitarias para probar el código de la aplicación y para la validación del sistema se usan las pruebas de aceptación para demostrar la conformidad de los requisitos.

Además de las pruebas que se describen a continuación, el sistema fue probado por el grupo de calidad del Centro de Gobierno Electrónico (CEGEL). Para ello se realizó una iteración de pruebas donde fueron encontradas diez no conformidades que fueron resueltas en la misma iteración, quedando liberado el sistema. En los anexos se encuentra el acta de liberación (ver Anexo 14).

3.3.1 Pruebas unitarias

Para realizar las pruebas de unidad se hace uso la herramienta NUnit que permite automatizarlas; con el objetivo de facilitar la creación y ejecución de pruebas unitarias.

Las pruebas unitarias se diseñaron de la siguiente forma: se elaboró un ejemplo de código en ensamblador, se realizó el análisis correspondiente al fichero, y se verificó que la salida fuera

correcta. Luego se comparó que para ese mismo ejemplo siempre arrojara la misma salida. En los anexos se encuentra el ejemplo de código y la salida de cada análisis (ver Anexos del 15 al 17).

A continuación se muestra una imagen de las pruebas unitarias realizadas al sistema (ver Figura 27):

```
12  namespace Compiler
13  {
14      [TestFixture]
        0 references | Rydem Storm +1, 6 minutes ago | 6 changes
15      public class TestSource
16      {
17          [Test]
            0 references | Rydem Storm +1, 6 minutes ago | 2 changes
18          public void PruebaLexer1()
19          {
20              string file = @"C:\Tmp\Docencia\code(2).asm";
21              var val = ObtenerTokens(file);
22              var esp = EsperadoL(file);
23              Assert.AreEqual(val, esp);
24          }
25          [Test]
            0 references | Rydem Storm +1, 6 minutes ago | 2 changes
26          public void PruebaLexer2()...
33          [Test]
            0 references | Rydem Storm +1, 6 minutes ago | 2 changes
34          public void PruebaParser1()
35          {
36              string file = @"C:\Tmp\Docencia\code(2).asm";
37              var val = ObtenerAsts(file);
38              var esp = EsperadoS(file);
39              Assert.AreEqual(val, esp);
40          }
41          [Test]
            0 references | Rydem Storm +1, 6 minutes ago | 2 changes
42          public void PruebaParser2()...
49
50
51
```

Figura 27: Pruebas unitarias

En la imagen que se muestra a continuación, se pueden observar los resultados obtenidos luego de ejecutar las pruebas de unidad (ver Figura 28):

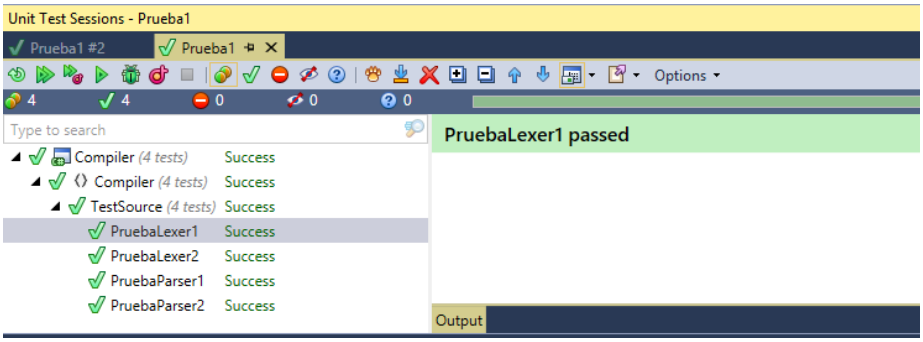

Figura 28: Resultados de las pruebas de unidad

Las pruebas de unidad se realizaron regularmente y cada vez que se hacían cambios significativos en el sistema. Los errores que fueron encontrados, se solucionaron inmediatamente.

3.3.2 Pruebas de aceptación

Las pruebas de aceptación permiten que el cliente valide todos los requisitos (Pressman, 2005).

Para la realización de las pruebas de aceptación se diseñaron los siguientes casos de prueba, con el objetivo de comprobar los requisitos funcionales del sistema (ver Tablas de la 25 a la 31):

Tabla 25: Prueba de aceptación Abrir Programa

Caso de Prueba de Aceptación	
Código: HU1_P1	**Historias de Usuario:** 1
Nombre: Abrir Programa.	
Descripción: permite comprobar que los programas se abran correctamente.	
Condiciones de ejecución: debe existir algún programa guardado con anterioridad.	
Entradas/Pasos de Ejecución: se selecciona la opción Abrir y se escoge el fichero deseado.	
Resultados esperados: el IDE carga en el editor de código el fichero seleccionado.	

Tabla 26: Prueba de aceptación Guardar Programa

Caso de Prueba de Aceptación	
Código: HU2_P1	**Historias de Usuario:** 2
Nombre: Guardar Programa.	
Descripción: permite comprobar que los programas se guarden correctamente.	
Condiciones de ejecución: no aplica.	
Entradas/Pasos de Ejecución: se selecciona la opción Guardar o la opción Guardar como y se escoge la ruta deseada para guardar el programa.	
Resultados esperados: prueba satisfactoria.	

Tabla 27: Prueba de aceptación Compilar programa

Caso de Prueba de Aceptación	
Código: HU3_P1, HU4_P1, HU5_P1	**Historias de Usuario:** 3, 4 y 5
Nombre: Compilar programa.	
Descripción: verifica que el programa compile correctamente y en caso de existir errores los señale correctamente.	
Condiciones de ejecución: debe existir un programa en el Editor de código	
Entradas/Pasos de Ejecución: se selecciona la opción compilar.	
Resultados esperados: prueba satisfactoria.	

Tabla 28: Prueba de aceptación Integración de la compilación y ejecución del código

Caso de Prueba de Aceptación	
Código: HU6_P1	**Historias de Usuario:** 6
Nombre: Integración de la compilación y ejecución del código.	
Descripción: permite comprobar que el programa se ejecute correctamente.	
Condiciones de ejecución: no aplica.	
Entradas/Pasos de Ejecución: se selecciona la opción Ejecutar.	
Resultados esperados: el IDE ejecuta correctamente el código.	

Tabla 29: Prueba de aceptación Detectar errores

Caso de Prueba de Aceptación	
Código: HU7_P1	**Historias de Usuario:** 7
Nombre: Detectar errores.	
Descripción: permite comprobar que el IDE detecta y muestra los errores en tiempo de codificación.	
Condiciones de ejecución: no aplica.	
Entradas/Pasos de Ejecución: se escribe código incorrecto.	
Resultados esperados: el IDE muestra los errores al cambiar de línea.	

Tabla 30: Prueba de aceptación Colorear sintaxis

Caso de Prueba de Aceptación	
Código: HU8_P1	**Historias de Usuario:** 8
Nombre: Colorear sintaxis.	
Descripción: permite comprobar que el IDE colorea la sintaxis del lenguaje.	
Condiciones de ejecución: el editor de código debe contener código.	
Entradas/Pasos de Ejecución: no aplica	
Resultados esperados: el IDE resaltará la sintaxis del lenguaje.	

Tabla 31: Prueba de aceptación Autocompletar código

Caso de Prueba de Aceptación	
Código: HU9_P1	**Historias de Usuario:** 9
Nombre: Autocompletar código.	
Descripción: permite comprobar que el IDE autocomplete código de manera inteligente.	
Condiciones de ejecución: no aplica	
Entradas/Pasos de Ejecución: el programador escribe una instrucción en lenguaje ensamblador y pulsa Ctrl+ Espacio.	
Resultados esperados: el IDE mostrará el autocompletamiento que corresponda.	

En la siguiente gráfica se presentan los resultados de las pruebas de aceptación aplicadas al finalizar cada iteración en que estuvo dividido el desarrollo del IDE (ver Figura 36):

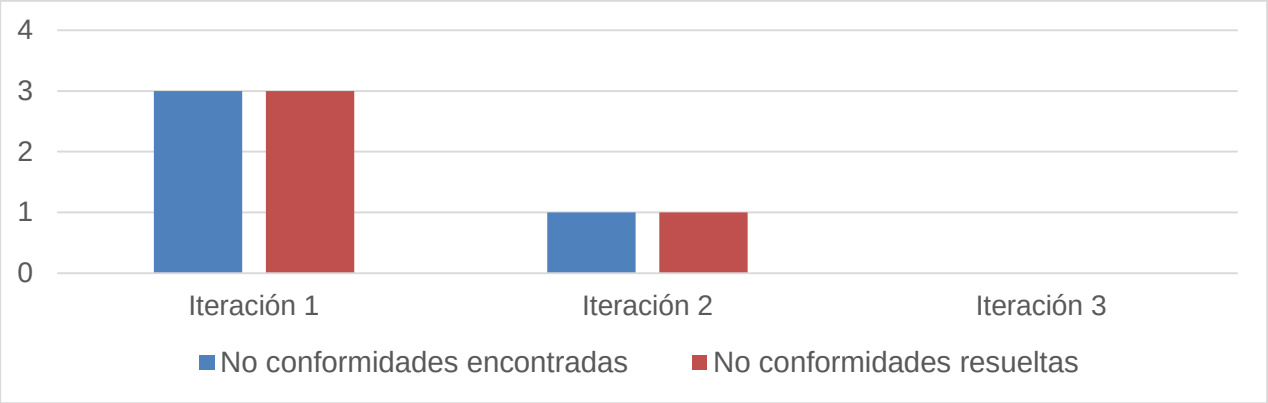

Figura 29: Resultados de las pruebas de aceptación

Como se puede observar, en la primera iteración se encontraron un total de tres no conformidades, las cuales fueron corregidas. En la segunda iteración se detectó una no conformidad que fue corregida. Por último se realizó una última iteración para validar el correcto funcionamiento del sistema, que tuvo como resultado la no existencia de nuevas no conformidades.

3.4 Validación por criterio de expertos

La herramienta desarrollada facilita en gran medida los procesos de codificación, compilación y ejecución del código en lenguaje ensamblador. A continuación se muestra una tabla donde se evidencia dicha afirmación, a partir de una comparación entre las herramientas utilizadas actualmente y la propuesta (ver Tabla 33).

Tabla 32: Comparación entre las herramientas utilizadas y la propuesta

Antes	**Después**
No se puede visualizar los errores en tiempo de codificación.	Se puede visualizar los errores en tiempo de codificación, lo cual es muy beneficioso para los estudiantes, ya que no hay que llegar a compilar el código para ver los errores.
Sólo se puede visualizar un error durante la compilación.	Se pueden visualizar todos los errores que presenta el código en tiempo de codificación y compilación. Esto ahorra tiempo y esfuerzo ya que en una sola ejecución del programa se pueden visualizar todos los errores.

La compilación del código se hace a nivel de comando en consola.	La compilación se realiza en la interfaz visual de la herramienta, mediante el uso de un botón.
La ejecución del código se hace a nivel de comandos en consola.	La ejecución se integra con la compilación y se realiza en la interfaz visual de la herramienta, mediante el uso de un botón.
No se cuenta con un mecanismo para el autocompletamiento de código.	La herramienta brinda la facilidad de autocompletamiento de código, lo cual facilita la codificación.

De forma general estas ventajas de la propuesta representan un gran apoyo para la enseñanza del lenguaje ensamblador en la asignatura de Arquitectura de Computadoras. Su uso permitirá a los profesores dedicar más tiempo al desarrollo de las habilidades de implementación y menos al adiestramiento en el uso de las herramientas.

Para evaluar el impacto de la herramienta en la enseñanza y la programación del lenguaje ensamblador, se aplicó una encuesta a un total de trece profesionales, que han impartido estas asignaturas y tienen elevada experiencia en este tema. El promedio de años de experiencia en la programación en ensamblador de los encuestados es de aproximadamente 9 años y el nivel de conocimiento de 8. Los indicadores a los que se les evaluó el impacto sobre la asimilación del lenguaje ensamblador fueron:

1. Menor tiempo de programación y compilación
2. Mayor calidad de los programas
3. Menor esfuerzo y dificultad de los programadores
4. Mejoría en los procesos de codificación, compilación y ejecución del código
5. Mayor grado de asimilación del contenido asociado
6. Mayor satisfacción de los estudiantes y profesores

Los resultados arrojados se muestran en la siguiente figura (ver Figura 30):

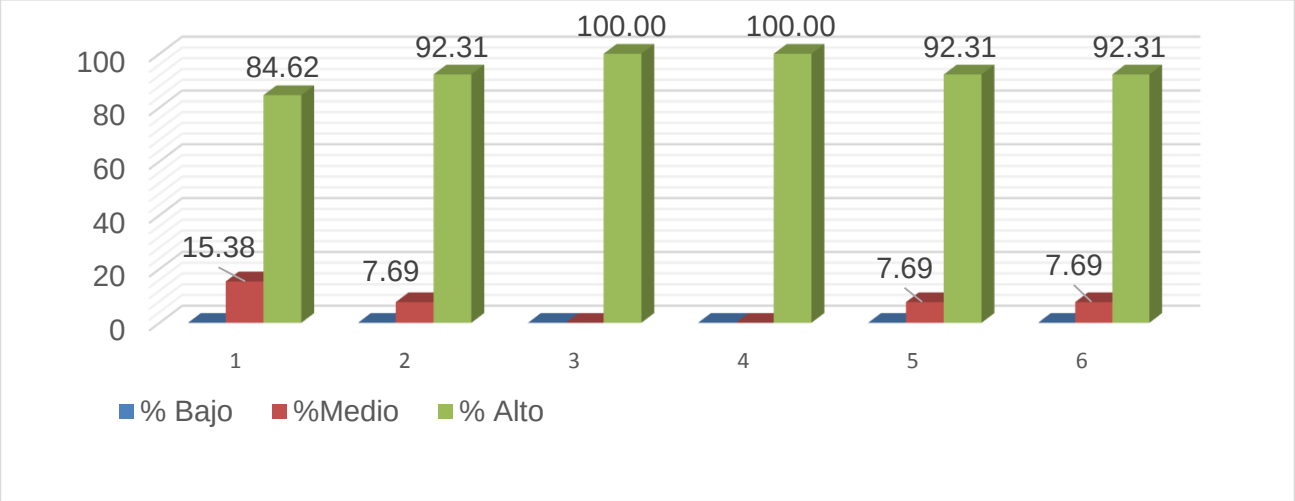

Figura 30: Resultados de la encuesta aplicada para valorar el impacto de la herramienta

Como se pudo constatar se prevé que la explotación del IDE desarrollado favorecerá con un impacto alto la habilidad de implementar en lenguaje ensamblador; haciendo el proceso más eficiente y amigable para estudiantes y profesores. De esta forma queda evidenciado el cumplimiento del objetivo general propuesto, de desarrollar un IDE que facilite los procesos de codificación, compilación y ejecución del código en lenguaje ensamblador, para los estudiantes de segundo año de la carrera Ingeniería en Ciencias Informáticas.

3.5 Conclusiones del capítulo

A lo largo del presente capítulo se realizó la verificación y validación del sistema, analizando los resultados obtenidos en cada métrica o prueba realizada al sistema, obteniendo los siguientes resultados:

- Los requisitos son estables y no presentan ambigüedad en su definición.
- El sistema es fácil de mantener y realizarle pruebas.
- A partir de las pruebas unitarias realizadas de forma automática y periódica, se verificó que el sistema mantuviera su correcto funcionamiento.
- Las pruebas de aceptación validaron que el sistema desarrollado responde y satisface las demandas del cliente.
- A partir de la valoración de trece expertos con 9 años de experiencia promedio y 8 de nivel de experticia media, se pudo constatar que el impacto de la solución en mayor eficiencia, menor esfuerzo, mejor grado de asimilación del contenido, y mayor satisfacción de estudiantes y profesores, es alto. Cumpliéndose de esta forma el objetivo general propuesto.

De esta forma se da por cumplido satisfactoriamente el objetivo.

CONCLUSIONES GENERALES

Con la realización del presente trabajo de diploma se arriban a las siguientes conclusiones:

- A partir de la encuesta realizada a trece profesionales de alta experiencia en la programación en ensamblador, se pudieron constatar los impactos negativos del uso de las herramientas definidas por la universidad, para el desarrollo de las habilidades de implementación en ensamblador de los estudiantes. Por estas razones, se hace necesario el desarrollo de un IDE que facilite la impartición de la asignatura, y la asimilación por parte de los estudiantes de los contenidos de la misma.
- Para la obtención del sistema se siguieron las pautas de la metodología XP, logrando la calidad del producto debido a la comunicación continua con el cliente y a las pruebas de unidad ejecutadas periódicamente.
- Las métricas aplicadas, las pruebas unitarias y de aceptación, verificaron el correcto funcionamiento del sistema, la calidad de la solución y el cumplimiento de las expectativas del cliente.
- A partir de la valoración de trece expertos con 9 años de experiencia promedio y 8 de nivel de experticia media, se pudo constatar que el impacto de la solución en mayor eficiencia, menor esfuerzo, mejor grado de asimilación del contenido, y mayor satisfacción de estudiantes y profesores, es alto. Cumpliéndose de esta forma el objetivo general propuesto.

RECOMENDACIONES

Se recomienda:

- Aumentar el reportorio de instrucciones soportado por la herramienta.
- Implementar el procesamiento de macros.
- Implementar el procesamiento de ficheros externos.

BIBLIOGRAFÍA

1. **Acevedo Martínez, Liesner, Osorio Ramírez, Karel. 2011.** Manual de apoyo a la docencia: Técnicas de Compilación Manual Práctico para estudiantes de Informática. La Habana : Universidad de las Ciencias Informáticas, 2011.

2. **Aho, Alfred V., y otros. 2010.** Compilers Principles, Techniques and Tools. 2nd. 2010.

3. **Alitimón, Arletis Francis y Serrano, Nicole Almenares. 2014.** Portal Web del Ministerio de. La Habana : s.n., 2014.

4. **Balderas Ortigosa, Yesenia G. 2011.** Entorno de programación EASY CODE. San Luis de Potosí, México : Instituto Tecnológico de Matehuala, 2011.

5. **Beck, Kent y Flower, Martin. 2000.** Planning Extreme Programming. 2000. 0201710919.

6. **Bell, D y Parr, M. 2003.** Java para estudiantes. s.l. : Pearson Editorial, 2003. ISBN 9702601444.

7. **Brena, Ramón. 2003.** Autómatas y Lenguajes Un enfoque se diseño. 2003. pág. 214.

8. **Calabria, Luis y Píriz, Pablo. 2003.** Metodología XP. Uruguay : s.n., 2003.

9. **Chacabana Yato, Nathali, Gutiérrez Cárdenas, Juan Manuel y Orihuela Ordóñez, Lenin. 2012.** Aplicación del Estándar ISO 9001:2000 a la Metodología de. 2012.

10. **Cnarte Ojeda, Francisco. 2009.** Ensamblador. Juan Ignacio Luca de Tena, 15. 28027 Madrid : Ediciones Anaya Multimedia, 2009. ISBN978- 84- 415- 2511- 5.

11. **Dart, S y Ellison, R. 2008.** Software Development Environments. IEEE Computer. 2008. págs. 18-28. Vol. XX.

12. **Ferrer Muñoz, Yeni y Acevedo Lara, Daniel Rafael. 2013.** Agente Inteligente en un Entorno Virtual Inmersivo 3D en función del trabajo con los estilos de aprendizajes de los usuarios en los Centros de Autoaprendizaje y Servicios de Idiomas Extranjeros. La Habana : s.n., 2013.

13. **Found, John. 2015.** Introduction. Fresh IDE. [En línea] 2015. [Citado el: 12 de Febrero de 2015.] http://fresh.flatassembler.net/index.cgi?page=content/0index.txt.

14. **Gálvez Rojas, Sergio y Mora Mata, Miguel Ángel. 2005.** Java a Tope: Traductores Y Ccompiladores con LEX/YACC, JFLEX/CUP Y JAVACC. Málaga : s.n., 2005. ISBN: 84-689-103 7-6.

15. **Gamma, Erich, y otros. 1994.** Design Patterns: Elements of Reusable Object-Oriented Software. 1994.

16. **Garbusi, Pablo. 2015.** Diseño de compiladores. 2015.

17. **García Sánchez, Ana María. 2010.** Evaluación de métricas de calidad del software sobre un programa Java. Facultad de Informática, Universidad Complutense de Madrid. 2010. pág. 89, Tesis de Maestría.

18. **Git. 2014.** Git. Git. [En línea] 2014. [Citado el: 6 de Febrero de 2015.] http://www.git-scm.com/.

19. **Gómez Díaz, Renzo Gonzalo y Salamanca Guillén, Juan Jesús. 2012.** Intérprete para un lenguaje de programación orientado a objetos, con mecanismos de optimización y modificación dinámica de código. Lima : s.n., 2012.

20. **Grysztar, Tomasz. 2015.** flat assembler. flat assembler. [En línea] 2015. [Citado el: 4 de Febrero de 2015.] http://flatassembler.net/download.php.

21. **Grystar, Tomasz. 2010.** flat assembler Programmer´s Manual. 2010.

22. **Guerber, Carlos. 2007.** Compiladores e interpretes. Universidad de Contestado. 2007.

23. **IEEE. 2004.** Guide to the Software Engineering Body of Knowledge (SWEBOK). 2004.

24. **INTECO. 2009.** Metodologías ágiles de desarrollo de software. s.l. : Laboratorio Nacional de Calidad del Software, 2009.

25. **Joskowicz, José. 2008.** Reglas y Prácticas en eXtreme Programming. 2008.

26. **Laborda, Javier, y otros. 1985.** Biblioteca práctica de la computación. Barcelona : Ediciones Óceano-Éxito, 1985.

27. **Larman, Craig. 2003.** UML y Patrones. s.l. : Prentice Hall, 2003.

28. **Letelier, Patricio y Penadés, M.Carmen. 2006.** Métodologías ágiles para el desarrollo de software. Valencia : Universidad Politécnica de Valencia, 2006.

29. **Manushin, Dimitry. 2015.** Learn and talk about SAMS. SAMS, Simple crossplatform IDE for NASM, MASM, GAS, FASM assembley languages. [En línea] GitHub Pages, 14 de Junio de 2015. dman95.github.io/SAMS/ english.html.

30. **Matellan Olivera, V. 2004.** Compilación de ensayos sobre software libre. Madrid : Dykinson, 2004. Matellan Olivera, V.

31. **Microsoft. 2014.** Características. Visual Studio. [En línea] 2014. [Citado el: 3 de Febrero de 2015.] http://www.visualstudio.com/es-es/explore/features-overview-vs.aspx.

32. **Microsoft. 2010.** Introducción al lenguaje C# y .NET Framework. Microsoft. [En línea] 2010. [Citado el: 2 de Febrero de 2015.] https://msdn.microsoft.com/es-es/library/z1zx9t92.aspx.

33. **Microsoft. 2010.** Resumen de las características de C#. Microsoft. [En línea] 2010. [Citado el: 8 de Febrero de 2015.] https://msdn.microsoft.com/es-es/library/aa287483%28v=vs.71%29.aspx.

34. **Microsoft. 2015.** Visual Studio Community 2013. Visual Studio. [En línea] 2015. [Citado el: 3 de Febrero de 2015.] http://www.visualstudio.com/products/visual-studio-community-vs.

35. **MonoDevelop. 2015.** Cross platform IDE for C#, F# and more. MonoDevelop. [En línea] 2015. [Citado el: 16 de Enero de 2015.] http://www.monodevelop.com/.

36. **Moreno, Alan. 2013.** Clases de programación con MASM+RADASM. s.l. : RVLCM, 2013.

37. **Navarrete, Christian. 2012.** Pwneando Ensamblador con ejemplos. s.l. : Creative Commons Reconocimiento-No Comercial-SinObraDerivada 3.0 Unported Licence, 2012.

38. **Pérez, María José Pérez. 2010.** Guía Comparativa de Metodología Ágiles. 2010.

39. **Pressman, Roger S. 2005.** Ingeniería del Software un enfoque práctico. 6th. s.l. : Mc Graw Hill, 2005. ISBN 9781430271987.

40. **Rouse, Margaret. 2007.** What is integrate development environment (IDE). In Software Quality information, news and tips. [En línea] 2007. [Citado el: 4 de Marzo de 2015.] searchsoftwarequality.techtarget.com/definition/integrated-development-environment.

41. **Rumbaugh, James, Jacobson, Ivar y Booch, Grady. 2004.** Unified Modeling Language. s.l. : The. Pearson Higher Education, 2004.

42. **Sacristán Donoso, Juan Marcos. 2011.** Desarrollo de compiladores. 2011. ISBN 84-689-4299-5.

43. **Sharma, Ritesh. 2014.** Code Metrics in Visual Studio. C# Corner. [En línea] 13 de Octubre de 2014. [Citado el: 18 de Mayo de 2015.] www.c-sharpcorner.com/UploadFile/78607b/code-metrics-in-visual-studio/.

44. **SharpDevelop. 2014.** Herramienta GNU para desarrolladores similar a Visual Studio. SharpDevelop. [En línea] 26 de Enero de 2014. [Citado el: 22 de Enero de 2015.] http://sharpdevelop.uptodown.com/.

45. **Sierra, Alejandro Agular. 2006.** Introducción a la Programación Extrema. México : Universidad Autónoma de México, 2006.

46. **Sommerville, Ian. 2005.** Ingeniería del software. Séptima. Madrid : Pearson Educación, S.A, 2005. ISBN: 84-7829-074-5.

47. **Terence Parr. 2009.** Lenguage Implementation Patterns. [ed.] Susannah Davidson Pialzer. 2009. ISBN-13: 978-1-934356-45-6.

ANEXOS

Anexo 1: Encuesta aplicada

Estimado(a) compañero(a), la presente encuesta forma parte de la aplicación de la Consulta a Expertos con el objetivo de obtener sus consideraciones sobre un Entorno de Desarrollo Integrado para el FASM de LINUX en la Facultad 3.

Se agradece su valiosa y calificada colaboración. Se le asegura que sus criterios contribuirán al perfeccionamiento de la propuesta y a la validación teórica de la presente investigación.

1. Información sobre los expertos

Nombre y Apellidos: _____

Graduado de: _____

Fecha de graduación: _____

Categoría Docente: Instructor__ Asistente__ Prof. Auxiliar__ Prof. Titular__

Título académico y/o grado científico: Máster___ Doctor___

Puesto de trabajo actual: _____

Años de experiencia en la programación en ensamblador: _____

2. ¿Qué experiencia usted posee acerca de la programación en ensamblador en Linux? _____

Indicación: Indique con un valor en la siguiente escala creciente del 1 (ninguno) al 10 (máximo) el valor que corresponda al grado de experiencia que usted tiene sobre el tema.

3. Le pedimos su criterio sobre los siguientes elementos que se corresponden con las características de la programación en ensamblador sobre Linux. Esto permitirá determinar la necesidad de un Entorno de Desarrollo Integrado para FASM en Linux.

Indicación: Marque con una equis (x) el criterio que más se ajusta al suyo para cada elemento de acuerdo a la propuesta.

Leyenda para los criterios: I – Inadecuado; PA – Poco Adecuado; A – Adecuado; BA – Bastante Adecuado; MA – Muy Adecuado.

#	Elementos a valorar de la programación en ensamblador sobre Linux	Criterios				
		I	PA	A	BA	MA
1	Se realiza sobre la consola de Linux					
2	Es preciso dominar los comandos para la compilación y ejecución					
3	El compilador sólo detecta un error por compilación					

4	Sólo se da información de la línea del error					
5	No hay distinción de sintaxis					
6	No existe autocompletamiento de código					
7	Codificación, compilación y ejecución del código engorrosos					

Otras valoraciones que usted desee aportar sobre la presente investigación serán de gran valor para su enriquecimiento y se le invita a realizarlas a continuación:_____
_____

4. Le pedimos su criterio sobre el impacto del uso de un Entorno de Desarrollo Integrado que resuelva estas limitaciones.

Indicación: Marque con una equis (x) el criterio que más se ajusta al suyo para cada elemento de acuerdo a la propuesta.

Leyenda para los criterios: B- Bajo; M- Medio; A – Alto.

#	Elementos a valorar el impacto	B	M	A
1	Menor tiempo de programación y compilación			
2	Mayor calidad de los programas			
3	Menor esfuerzo y dificultad de los programadores			
4	Mejoría en los procesos de codificación, compilación y ejecución del código			
5	Mayor grado de asimilación del contenido asociado			
6	Mayor satisfacción de estudiantes y profesores			

Otras valoraciones que usted desee aportar sobre el impacto de la propuesta serán de gran valor para su enriquecimiento y se le invita a realizarlas a continuación:_____
_____

Anexo 2: Repertorio de instrucciones soportado por la herramienta

Repertorio de instrucciones básico del lenguaje ensamblador:

Instrucciones de movimiento:

- MOV
- XCHG
- PUSH
- PUSHW

- PUSHA
- PUSHAW
- PUSHAD
- POP
- POPW
- POPA
- POPA
- POPAW
- POPAD

Instrucciones de conversión de tipos:

- MOVSX
- MOVZX

Instrucciones aritméticas:

- ADD
- ADC
- INC
- SUB
- SBB
- DEC
- CMP
- NEG
- XADD
- MUL
- DIV

Instrucciones lógicas:

- NOT
- AND
- OR
- XOR
- SHL
- SHR
- SHLD
- SHRD
- ROL
- RCL
- ROR
- RCR
- TEST

Instrucciones de transferencia de control:

- JMP
- CALL
- RET
- IRET
- LOOP
- LOOPW
- LOOPD
- LOOPE
- LOOPZ
- JO
- JNO
- JC
- JB
- JNAE
- JNC
- JAE
- JNB
- JE
- JZ
- JNE

- JNZ
- JBE
- JNA
- JA
- JNBE
- JS
- JNS
- JP
- JPE
- JNP
- JPO
- JL
- JNGE
- JGE
- JNL
- JLE
- JNG
- JG
- JNLE
- INT

Instrucciones de entrada/salida:

- IN
- OUT

Instrucciones de cadena:

- MOVS
- MOVS

Anexo 3: TI Guardar programa

Tarea de Ingeniería	
Número: 4	**Nombre Historia de Usuario:** Guardar programa
Nombre de la Tarea: Implementar la funcionalidad guardar programa	
Tipo de tarea: desarrollo **(desarrollo/corrección/mejora)**	**Puntos Estimados(días):** 1
Fecha inicio: 01/05/2015	**Fecha fin:** 01/05/2015
Programador responsable: Anabel González Valiente	
Descripción: en esta tarea se implementará la funcionalidad guardar y guardar como.	

Anexo 4: TI Implementar las funcionalidades para permitir reconocer los caracteres que conforman el programa fuente.

Tarea de Ingeniería	
Número: 5	**Nombre Historia de Usuario:** Análisis Léxico
Nombre de la Tarea: Implementar las funcionalidades para permitir reconocer los carácter que conforman el programa fuente.	

Tipo de tarea: desarrollo **(desarrollo/corrección/mejora)**	**Puntos Estimados(días):** 2
Fecha inicio: 15/03/2015	**Fecha fin:** 16/03/2015
Programador responsable: Anabel González Valiente	
Descripción: en esta tarea se implementará las funcionalidades necesarias para leer los caracteres que conforman el programa fuente.	

Anexo 5: TI Implementar las funcionalidades que permiten obtener los tokens que conforman el programa fuente.

Tarea de Ingeniería	
Número: 6	**Nombre Historia de Usuario:** Análisis Léxico
Nombre de la Tarea: Implementar las funcionalidades que permiten obtener los tokens que conforman el programa fuente.	
Tipo de tarea: desarrollo **(desarrollo/corrección/mejora)**	**Puntos Estimados(días):** 1
Fecha inicio: 17/03/2015	**Fecha fin:** 17/03/2015
Programador responsable: Anabel González Valiente	
Descripción: en esta tarea se implementarán las funcionalidades para conformar los tokens.	

Anexo 6: TI Implementar la clase Parser

Tarea de Ingeniería	
Número: 7	**Nombre Historia de Usuario:** Análisis Sintáctico
Nombre de la Tarea: Implementar la clase Parser.	
Tipo de tarea: desarrollo **(desarrollo/corrección/mejora)**	**Puntos Estimados(días):** 10
Fecha inicio: 18/03/2015	**Fecha fin:** 28/03/2015
Programador responsable: Anabel González Valiente	
Descripción: en esta tarea se implementará la clase Parser, encargada de realizar el análisis sintáctico.	

Anexo 7: TI Implementar los AST.

Tarea de Ingeniería	
Número: 8	**Nombre Historia de Usuario:** Análisis Sintáctico
Nombre de la Tarea: Implementar los AST.	
Tipo de tarea: desarrollo **(desarrollo/corrección/mejora)**	**Puntos Estimados(días):** 20
Fecha inicio: 29/03/2015	**Fecha fin:** 18/04/2015
Programador responsable: Anabel González Valiente	
Descripción: en esta tarea se implementará las clases relacionadas con el árbol de sintaxis abstracta.	

Anexo 8: TI Implementar la clase Checker.

Tarea de Ingeniería	
Número: 9	**Nombre Historia de Usuario:** Análisis Semántico.
Nombre de la Tarea: Implementar la clase Checker.	
Tipo de tarea: desarrollo **(desarrollo/corrección/mejora)**	**Puntos Estimados(días):** 6
Fecha inicio: 19/04/2015	**Fecha fin:** 25/04/2015
Programador responsable: Anabel González Valiente	
Descripción: en esta tarea se implementará la clase Chequer, encargada de verificar la semántica de las instrucciones que componen el programa fuente.	

Anexo 9: TI Implementar las validaciones semánticas en cada AST.

Tarea de Ingeniería	
Número: 10	**Nombre Historia de Usuario:** Análisis semántico
Nombre de la Tarea: Implementar las validaciones semánticas en cada clase AST	
Tipo de tarea: desarrollo **(desarrollo/corrección/mejora)**	**Puntos Estimados (días):** 2
Fecha inicio: 26/04/2015	**Fecha fin:** 28/04/2015

Programador responsable: Anabel González Valiente
Descripción: en esta tarea se implementarán las validaciones semánticas en las clases AST.

Anexo 10: TI Implementar la ejecución del código.

Tarea de Ingeniería	
Número: 11	**Nombre Historia de Usuario:** Integrar la compilación y ejecución del código.
Nombre de la Tarea: Implementar la ejecución del código.	
Tipo de tarea: desarrollo **(desarrollo/corrección/mejora)**	**Puntos Estimados (días):** 2
Fecha inicio: 02/05/2015	**Fecha fin:** 03/05/2015
Programador responsable: Anabel González Valiente	
Descripción: En esta tarea se implementará la integración y ejecución del código, esto se realizará de la siguiente forma: - Se manda a compilar con el compilador FASM, en caso de no existir errores se manda a ejecutar con la máquina virtual Qemu.	

Anexo 11: TI Implementar la funcionalidad mostrar errores.

Tarea de Ingeniería	
Número: 12	**Nombre Historia de Usuario:** Mostrar errores
Nombre de la Tarea: Implementar la funcionalidad mostrar errores.	
Tipo de tarea: desarrollo **(desarrollo/corrección/mejora)**	**Puntos Estimados (días):** 1
Fecha inicio: 03/05/2015	**Fecha fin:** 04/05/2015
Programador responsable: Anabel González Valiente	
Descripción: se implementará la funcionalidad de mostrar los errores en tiempo de codificación.	

Anexo 12: TI Colorear la sintaxis del lenguaje.

Tarea de Ingeniería

Número: 13	**Nombre Historia de Usuario:** Colorear sintaxis.
Nombre de la Tarea: Colorear la sintaxis del lenguaje	
Tipo de tarea: desarrollo **(desarrollo/corrección/mejora)**	**Puntos Estimados (días):** 3
Fecha inicio: 04/05/2015	**Fecha fin:** 06/05/2015
Programador responsable: Anabel González Valiente	
Descripción: se implementarán las funcionalidades necesarias para colorear la sintaxis del lenguaje ensamblador.	

Anexo 13: TI Implementar el autocompletamiento de código.

Tarea de Ingeniería	
Número: 14	**Nombre Historia de Usuario:** Autocompletar código.
Nombre de la Tarea: Implementar el autocompletamiento de código.	
Tipo de tarea: desarrollo **(desarrollo/corrección/mejora)**	**Puntos Estimados (días):** 7
Fecha inicio: 06/05/2015	**Fecha fin:** 13/05/2015
Programador responsable: Anabel González Valiente	
Descripción: se implementarán las funcionalidades necesarias para realizar el autocompletamiento del código.	

Anexo 14: Acta de Liberación

UCI Universidad de las Ciencias Informáticas | FACULTAD # 3
CENTRO DE GOBIERNO ELECTRÓNICO

CEGEL

Acta de Liberación Interna de Productos Software

Fecha de emisión del acta: 03/06/2015

Emitida a favor de: Tesis Entorno de Desarrollo Integrado para el lenguaje ensamblador

Datos del producto

Artefacto	Versión	Estado final	Cantidad Iteraciones	Tipos de pruebas realizadas	Fecha de liberación
App: ASM#	*1.0*	*0*	*1*	*Evaluación dinámica* *Pruebas de Funcionalidad*	*03/06/2015*

Ing. Doris Maza Oval
Asesor de Calidad del Centro

Anabel González Valiente
Autor

UCI Universidad de las Ciencias Informáticas
Centro de Gobierno Electrónico

1

Anexo 15: Ejemplo de código pruebas de unidad

```
1        mov ebx,0xb8000
2        mov edi,160*12+48
3        mov ah,4eh
4        mov esi,msg
5        @@: mov al,[esi]
6        cmp al,0
7        je  @f
8        mov [ebx+edi],ax
9        add edi,2
10       inc esi
11       jmp @b
12       @@: jmp $
13
14       msg db 'Texto en el espacio exterior  ;-)',0
```

Anexo 16: Salida del Lexer

```
TabTab,Separador
mov,Instruction
Tab,Separador
ebx,Register
,,Coma
0xb8000,Numero
Retorno,Retorno
TabTab,Separador
mov,Instruction
Tab,Separador
edi,Register
,,Coma
160,Numero
*,Por
12,Numero
+,Mas
48,Numero
Retorno,Retorno
TabTab,Separador
mov,Instruction
Tab,Separador
ah,Register
,,Coma
4eh,Numero
Retorno,Retorno
TabTab,Separador
mov,Instruction
Tab,Separador
esi,Register
,,Coma
msg,Identificador
Retorno,Retorno
Tab    ,Separador
```

Anexo 17: Salida del parser

```
mov ebx,0xb8000
mov edi,160 * 12 + 48
mov ah,4eh
mov esi,msg
@@:
mov al,[esi]
cmp al,0
je @f
mov [ebx + edi],ax
add edi,2
inc esi
jmp @b
@@:
jmp $
msg db 'Texto en el espacio exterior  ;-)',0
```