UNIVERSIDAD CATÓLICA SANTO TORIBIO DE MOGROVEJO
FACULTAD DE INGENIERÍA
ESCUELA DE INGENIERÍA DE SISTEMAS Y COMPUTACIÓN

SISTEMA INTERACTIVO BASADO EN UN INTÉRPRETE DE ALGORITMOS PARA MEJORAR EL MÉTODO DE APRENDIZAJE DE LOS ALUMNOS DEL CURSO FUNDAMENTOS DE PROGRAMACIÓN

TESIS PARA OPTAR EL TÍTULO DE
INGENIERO DE SISTEMAS Y COMPUTACIÓN

EDDER ALAÍN SÁNCHEZ BACA

Chiclayo 21 de mayo de 2015

“SISTEMA INTERACTIVO BASADO EN UN INTÉRPRETE DE ALGORITMOS PARA MEJORAR EL MÉTODO DE APRENDIZAJE DE LOS ALUMNOS DEL CURSO FUNDAMENTOS DE PROGRAMACIÓN”

POR:

EDDER ALAÍN SÁNCHEZ BACA

Presentada a la Facultad de Ingeniería de la Universidad Católica Santo Toribio de Mogrovejo para optar el título de
INGENIERO DE SISTEMAS Y COMPUTACIÓN

APROBADA POR EL JURADO INTEGRADO POR

Ing. Héctor Miguel Zelada Valdivieso
PRESIDENTE

Ing. Carlos Rodas Díaz
SECRETARIO

Mgtr. Eduardo Alonso Pérez
ASESOR

DEDICATORIA

Este trabajo va dedicado a Camila, mi nueva razón de ser, quien me dará siempre las fuerzas para triunfar.

A mis padres, quienes me ayudan a tomar las mejores decisiones, y quienes siempre me apoyan cuando los necesito.

AGRADECIMIENTOS

Agradezco profundamente a mis padres porque es por ellos que he alcanzado mis logros, a ellos les debo todo lo que sé, lo que conozco, lo que soy.

A aquellos profesores que guiaron mi investigación, ya que aportaron mucha experiencia a este trabajo y depositaron su confianza en mí.

ÍNDICE

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

RESUMEN
Actualmente, en un mundo donde los dispositivos electrónicos tienen cada vez mayores capacidades (celulares, computadoras, tablets, etc.) se necesitan también profesionales que desarrollen aplicaciones y software de manera más eficiente. Cuando se analizó la problemática en nuestra sociedad actual, nos dimos cuenta de que el principal problema de los estudiantes de programación es la falta de práctica y análisis, ya que aproximadamente un 40% de los alumnos que fueron parte del estudio no intentaban realizar un algoritmo antes de programar en la computadora. Con la presente investigación se pretende contribuir motivando a los estudiantes para que incrementen sus horas de práctica y comprendan mejor los temas de programación mediante un sistema multimedia basado en un intérprete de algoritmos. Para el desarrollo del software se utilizó el marco de trabajo Proceso Unificado Ágil. Como resultado se obtuvo un producto que ayuda a incrementar el número de estudiantes aprobados en evaluaciones de estructuras condicionales y en estructuras repetitivas, una disminución del tiempo promedio de resolución de algoritmos y el incremento del porcentaje de satisfacción con el método de enseñanza-aprendizaje.

PALABRAS CLAVE*: intérprete, compilador, algoritmo, educación, enseñanza, aprendizaje, lógica, eficiencia.*

ABSTRACT
Today, in a world where electronic devices have increased capacities (cell phones, computers, tablets, etc.) are also needed professionals to develop applications and software more efficiently. When the problems in our society was analyzed, we realized that the main problem of programming students is the lack of practice and analysis, since about 40% of students who were part of the study did not attempt to perform an algorithm before programming the computer. This research contributes motivating students to increase their hours of practice and understanding the issues of programming through a multimedia system based on an algorithms' interpreter. The software was develop with the framework Agile Unified Process. The result is a product that helps to increase the number of students who passed assessments conditional structures and repetitive structures, a lower average resolution time algorithms and increasing the percentage of satisfaction with the teaching-learning method was obtained.

KEYWORDS*: interpreter, compiler, algorithm, education, teaching, learning, logic, efficiency.*

I. INTRODUCCIÓN

Actualmente el mundo está atravesando por una fase llamada "hiperconectividad", en la que podemos salir a las calles y ver, al menos a un 70% de la población con "celulares inteligentes", siempre conectados, siempre usando tecnología en la palma de sus manos. Pero, ¿Qué hay detrás de todas esas aplicaciones que mantienen tan ocupadas o entretenidas a las personas hoy en día?, pues no es nada más y nada menos que código, líneas y líneas de código, cada una de las cuales significa una acción o comportamiento específico que el dispositivo debe ejecutar, todo ello posible gracias a los lenguajes de programación y sus compiladores, que permiten plasmar las ideas de los programadores, en órdenes que cada computadora o dispositivo pueda realizar.

Sin embargo, incluso con la existencia de un lenguaje de programación, es difícil desarrollar software, ya que antes de comenzar la construcción o aprender a utilizar el lenguaje de programación, es necesaria la elaboración de un algoritmo, empleando la lógica. Joyanes (2004:27) define un algoritmo como un método con un conjunto de instrucciones utilizadas para resolver un problema.

Por lo tanto, todas aquellas personas que se interesan por la construcción de software (sistemas, aplicaciones, programas) deben dar un primer paso: aprender a construir algoritmos. Esto es muy importante porque una vez que tengamos de manera detallada todos los pasos a seguir podremos "transcribir" nuestra idea al lenguaje de programación, haciendo la tarea mucho más sencilla. Por otro lado, la enseñanza de algoritmos requiere, principalmente, el interés y la práctica del alumno.

En el mundo, es cada vez mayor la demanda de personas que requieren aprender a programar, incluso se ha instaurado como materia en muchos colegios de Estados Unidos y algunos países de Europa, y ha surgido ya muchas investigaciones y proyectos que analizan y pretenden ayudar a optimizar el proceso de enseñanza – aprendizaje de algoritmos. Anders Berglund y Raymond Lister, de Uppsala University (Suecia) y de University of Technology, Sydney (Australia), respectivamente, presentaron en 2010 su investigación "Introductory Programming and the Didactic Triangle", en la que reconocen que, como en todo proceso educativo, existe un triángulo didáctico: El profesor, el alumno y el conocimiento; y aportan en la problemática de cómo aprovechar la interrelación entre los tres elementos para optimizar los resultados del proceso enseñanza – aprendizaje.

Ya envueltos en este tema, evaluemos la situación de los alumnos de la carrera de Ingeniería de Sistemas y Computación en la Universidad Católica Santo Toribio de Mogrovejo.

Cada año se tiene un aproximado de 70 ingresantes a la Escuela Profesional de Ingeniería de Sistemas y Computación, de los cuales, cerca de un 30% no sabía que la programación es un tema básico en la carrera, esto hace que los alumnos sientan confusión al llevar los cursos relacionados a la programación, también tenemos que de los alumnos que cursaron la asignatura Fundamentos de Programación, el 66% (de 44) desaprobó el curso en el ciclo académico 2011-I, el 44% (de 32) en el ciclo académico 2011-II, y el 37% (de 57) en el ciclo académico 2012-I, esto implica que retrasen sus estudios al no poder llevar futuros cursos que como requisito base tienen la asignatura mencionada.

En el ciclo académico 2010-I, en el curso Estructuras de Datos, se demostró que de aproximadamente 30 alumnos, el 30% no lograba comprender la secuencia de unas líneas de código mostradas, esto es causado principalmente por la falta de práctica de algoritmos por parte de los alumnos, ya que la práctica es fundamental para desarrollar la lógica que permita interpretar líneas de código, y tiene como consecuencia un déficit de comprensión al momento de utilizar librerías, ya que si el alumno no puede interpretar código, confunde "reutilizar código" con "copiar código". Esta falta de práctica puede atribuirse a la falta de interés por el tema, es decir, que la programación no sea del agrado de los alumnos, podemos notarlo ya que de 57 estudiantes que cursaron la asignatura de Fundamentos de Programación en el ciclo 2012-I, solamente el 85% afirma que les gusta la programación, el resto de alumnos sintió frustración y aburrimiento en las clases de dicho curso y no prestaba suficiente atención al profesor. Además, aproximadamente un 8% de dichos estudiantes no cumplían con los trabajos designados para practicar en casa, y al profesor le tomaba más tiempo explicar conceptos nuevos al tener que repetir temas que los alumnos ya deberían saber.

De los 27 estudiantes que cursaron la asignatura de Metodologías de Programación en el ciclo 2012-I, un 40% intentaba desarrollar los ejercicios de programación directamente en la computadora, sin antes haber realizado el análisis del problema y el respectivo algoritmo en papel, siendo esto último una práctica que ayuda a desarrollar la lógica de programación en el alumno. Si bien es cierto, no es estrictamente necesario que se utilice papel para realizar el algoritmo previo a la programación, lo que se intenta explicar es que los alumnos no realizaban el análisis previo del problema.

Por lo tanto, podemos notar que existen problemas en el proceso de enseñanza – aprendizaje en los cursos de programación; en la presente investigación, haremos hincapié en los estilos y métodos de aprendizaje de los alumnos, de manera más concreta, al evaluar los problemas mencionados encontramos una falta de práctica e

interés por parte de los alumnos que desaprueban el curso, y teniendo esto en cuenta, las posibilidades de solución se redujeron a abordar el problema basándonos en la psicología educativa, y utilizando una herramienta de software para ponerla en práctica, de esta manera surge la idea de desarrollar un sistema interactivo basado en un intérprete de algoritmos.

Una vez obtenida la alternativa de solución, nos preguntamos: ¿De qué manera un sistema interactivo basado en un intérprete de algoritmos podrá mejorar el método de aprendizaje de los alumnos del curso Fundamentos de programación?

Ya que consideramos que el principal problema en el proceso de enseñanza - aprendizaje del curso Fundamentos de programación es la falta de práctica e interés por parte de los alumnos, se espera que con un pseudolenguaje en español y de estructura sencilla, enlazado a un entorno gráfico, simulando un videojuego del tipo plataforma se logre captar la suficiente atención de los mismos, logrando así un incremento de interés y práctica, y de esta manera mejorar el proceso de aprendizaje, tornándolo más eficiente.

Dicho esto, el objetivo general de esta investigación queda claro: mejorar el proceso de aprendizaje de los alumnos del curso Fundamentos de programación a través de la implementación de un sistema interactivo. Y los objetivos específicos a cumplir son:

- Incrementar el número de estudiantes aprobados en evaluación de estructuras condicionales.
- Incrementar el número de estudiantes aprobados en evaluación de estructuras repetitivas.
- Disminuir el tiempo promedio de resolución de algoritmos.
- Incrementar el porcentaje de satisfacción con el método de enseñanza-aprendizaje.

El desarrollo de este proyecto está justificado económicamente, tomando en cuenta que cuando los alumnos desaprueban un curso deben volver a llevarlo, no sólo pagando nuevamente el costo de la asignatura, sino que además pagan un cargo extra al llevarlo por segunda vez, monto que se incrementa si el curso es llevado por tercera vez. Además, cuando los alumnos no comprenden los temas tratados en una asignatura, se ven perjudicados cuando se asignan trabajos o cuando llega el momento de las evaluaciones, y algunos estudiantes prefieren pagar a otras personas para repasar los temas ya que consideran tedioso o vergonzoso preguntarle al profesor.

Mediante el desarrollo de este proyecto se logra también un aporte científico, ya que se estudia la construcción de un pseudolenguaje de programación y el comportamiento de su respectivo compilador – intérprete, además se pretende brindar a este sistema interactivo una licencia libre, contribuyendo así a la comunidad de software libre y teniendo el código fuente a disposición de quien más lo requiera para fines educativos.

Esta investigación, a la vez, ayudará a formar mejores profesionales de Ingeniería de Sistemas y Computación ya que tendrán un buen soporte en la línea de programación, brindando a la sociedad profesionales competentes, que tengan capacidad para resolver problemas en una organización. Por último, cabe resaltar que, programación, lenguajes y compiladores, y gráficos por computadora son las áreas de mayor interés del investigador de este proyecto, logrando así aportar a la comunidad científica con un proyecto elaborado con el esfuerzo y dedicación que merecen.

II. MARCO TEÓRICO

2.1 Antecedentes

En junio del 2011, el entonces estudiante Josué Elías Delgado Marrufo, realizó una investigación titulada "Sistema multimedia educativo para mejorar el nivel de los estilos individuales de aprendizaje en el curso de matemáticas mediante el uso del modelo VARK para los alumnos del 6° grado de la Institución Educativa Flemming College", en la cual plantea que cada estudiante aprende de manera diferente, y propone el uso de la tecnología para mejorar los estilos individuales de aprendizaje en el curso de matemáticas, mediante el uso del modelo VARK, un modelo para identificar estilos de aprendizaje (Visual, Aural, Read/Write, Kinesthetic). Para esta investigación se hizo uso de la metodología dinámica para el desarrollo de software educativo, y la metodología de diseño y desarrollo multimedia de Brian Blum.

En el año 2007, Layla Hirsh Martines, estudiante de la Pontificia Universidad Católica del Perú, realizó una investigación llamada "Intérprete y entorno de desarrollo para el aprendizaje de lenguajes de programación estructurada", la cual tiene como objetivo principal el diseño, desarrollo e implementación de un intérprete de un lenguaje de programación que pueda ser usado en los primeros cursos de introducción a la computación. En esta investigación la autora se centra principalmente en el lenguaje de programación que se escoge para enseñar a programar, es decir, analiza qué características debe tener el lenguaje de programación que se utiliza para programar por primera vez, entre las cuales se destaca que el lenguaje de programación debe permitir, con su estructura, una rápida comprensión por parte del estudiante, además, una de las principales desventajas con respecto a la educación en temas de programación en nuestro país, es que los lenguajes de programación más conocidos, con los que se desarrollan la mayoría de aplicaciones, están construidos en inglés, y esto es una desventaja porque en el Perú el aprendizaje del idioma inglés no es muy efectivo, y cuando los estudiantes desean aprender a programar, se les enseña con algún lenguaje de programación que existe en el mercado, como Java, Visual Basic, C, PHP, los cuales están en inglés, y esto causa una demora en el aprendizaje por parte de los alumnos. Como solución a los problemas expuestos, la autora plantea desarrollar un lenguaje de programación en español, que permita una rápida asimilación de los conceptos lógicos básicos que se necesitan para aprender a programar y desarrollar la lógica de construcción de algoritmos.

En febrero del año 2012, los alumnos de la Pontificia Universidad Católica del Perú, Renzo Gonzalo Gómez Díaz y Juan Jesús Salamanca Guillén, desarrollaron la investigación titulada "Intérprete para un lenguaje de programación orientado a objetos, con mecanismos de optimización y modificación dinámica de código" en la que se centran principalmente en las técnicas de construcción de compiladores e intérpretes,

para que el trabajo sirva de apoyo a personas que desean profundizar en el tema. Los autores de esta investigación explican que en el Perú no se incentiva el aprendizaje de construcción de compiladores, lo que causa que la gran mayoría de profesionales no comprendan el funcionamiento interno de los intérpretes y compiladores, brindando a la sociedad programas y sistemas que no utilizan eficientemente los recursos del hardware donde son ejecutados. Además, en esta investigación se explica claramente que todo compilador se divide en dos grandes partes, el "front-end" que no depende de la máquina donde se vaya a ejecutar el código pero sí depende del lenguaje en el que es escrito el código fuente a ser traducido, y el "back-end" que no depende del lenguaje del código fuente a traducir, pero sí depende de la máquina en la que se va a ejecutar el programa. A su vez, se explica que el "back-end" de todo compilador tiene procesos fundamentales como la optimización de código y la modificación dinámica de código, y son en estos procesos en los que se profundiza esta investigación.

En julio de 1999, Juan Pablo Casares Charles, estudiante del Instituto Tecnológico Autónomo de México desarrolló la herramienta AMIVA, producto de su investigación "AMIVA: Ambiente para la Instrucción Visual de Algoritmos", a diferencia de otras investigaciones en las que se establece desarrollar un intérprete o compilador para un lenguaje de programación fácil de aprender, con la finalidad de hacer más eficiente la enseñanza de algoritmos, el autor de esta investigación concluye de que la mejor forma para aprender a programar es aprender a desarrollar algoritmos correctamente. Primero hace un análisis de las formas y métodos de aprendizaje existentes, desde el punto de vista de la psicología educativa, y estipula que la mejor manera es desde el aprendizaje visual, por ello es que surge la idea de construir una herramienta que, mediante una adecuada interacción gráfica, permita que los estudiantes asimilen mejor los conceptos y técnicas para desarrollar algoritmos. Así nació AMIVA, un entorno en el que se puede desarrollar un algoritmo a manera de diagrama de flujo.

Por otro lado, en la Universidad de Carnegie Mellon, un grupo de jóvenes estudiantes crearon la herramienta ALICE, un entorno de programación en 3D que hace fácil la tarea de crear una animación para una historia, jugar un videojuego interactivo, o un vídeo para compartir en la web. ALICE es una herramienta de enseñanza disponible de manera gratuita, diseñada para ser el primer contacto de los estudiantes con la programación orientada a objetos. Permite a los estudiantes aprender conceptos de programación fundamentales en el contexto de creación de películas animadas y videojuegos simples. En ALICE, los objetos 3D (personas, animales, vehículos) forman parte de un mundo virtual, y los estudiantes crean un programa para animar los objetos. ALICE permite a los estudiantes ver inmediatamente cómo se ejecutan sus programas de animación, permitiéndoles entender fácilmente la relación entre las instrucciones de programación y el comportamiento de los objetos en su animación.

Basándose en la herramienta ALICE, Irene Montano Rodríguez, de la Universidad Rey Juan Carlos de España, realizó entre el 2009 y el 2010 una investigación denominada "REBECA: Software Educativo de Introducción a la programación para hispano hablantes" en la que pretende adaptar el software ALICE al idioma español y también a la realidad de la enseñanza de informática en España.

En la Universidad Católica Santo Toribio de Mogrovejo, en el año 2003, Eduardo Alonso Pérez y José Alvites Rodas desarrollaron "ALGOR: Un intérprete para la enseñanza de algoritmos", un pseudolenguaje en español y su intérprete desarrollado en lenguaje C, que surgió con la finalidad de encontrar nuevas estrategias que ayuden a la enseñanza de algoritmos en carreras de computación e informática.

2.2 Bases Teórico Científicas

2.2.1 Aprendizaje

El aprendizaje es un proceso en el cual, debido a la experiencia, se produce un cambio relativamente permanente en nuestra actividad; existen cuatro tipos de aprendizaje:

- Aprendizaje motor: Consiste en la adquisición de secuencias de movimientos coordinados. Es decir, adquirir la destreza de determinado movimiento corporal.
- Aprendizaje cognoscitivo: Es el proceso a través del cual se adquieren conocimientos y estrategias de resolución de problemas que involucran procesos de discernimientos. (...)También se incluyen en esta variedad la meta cognición que consiste en el aprendizaje de conocimientos que permitan la obtención de conocimientos.
- Aprendizaje afectivo: Es el proceso por el cual se adquieren o modifican nuestros afectos y sus formas de expresión hacia determinadas personas y objetos.
- Aprendizaje social: Es el proceso en el cual se incorporan las normas sociales, costumbres e ideología de una determinada sociedad.

A todo esto podemos determinar que el tipo principal de aprendizaje requerido en el ámbito educativo es el aprendizaje cognoscitivo, resaltando que para el proceso de enseñanza se necesita de la meta cognición, ya que quien enseña, debe conocer de qué manera llegar al alumno o estudiante, y debe conocer los métodos que le permitan obtener mayores conocimientos (ADUNI, 2011).

2.2.2 Estilos de aprendizaje

No existe una única definición de estilos de aprendizaje, sino que son muchos los autores que dan su propia definición del término (Navarro, 2008).

"Los estilos de aprendizaje son los rasgos cognitivos, afectivos y fisiológicos que sirven como indicadores relativamente estables, de cómo los alumnos perciben interacciones y responden a sus ambientes de aprendizaje" (Keefe, 1988).

"El estilo de aprendizaje es la manera en la que un aprendiz comienza a concentrarse sobre una información nueva y difícil, la trata y la retiene" (Dunn et Dunn, 1985).

El término "estilo de aprendizaje" se refiere al hecho de que cada persona utiliza su propio método o estrategias a la hora de aprender. Aunque las estrategias varían según lo que se quiera aprender, cada uno tiende a desarrollar ciertas preferencias o tendencias globales, tendencias que definen un estilo de aprendizaje. Se habla de una tendencia general, puesto que, por ejemplo, alguien que casi siempre es auditivo puede en ciertos casos utilizar estrategias visuales.

Cada persona aprende de manera distinta a las demás: utiliza diferentes estrategias, aprende con diferentes velocidades e incluso con mayor o menor eficacia incluso aunque tengan las mismas motivaciones, el mismo nivel de instrucción, la misma edad o estén estudiando el mismo tema. Sin embargo más allá de esto, es importante no utilizar los estilos de aprendizaje como una herramienta para clasificar a los alumnos en categorías cerradas, ya que la manera de aprender evoluciona y cambia constantemente.

Revilla(1998) Nos indica que los estilos de aprendizaje son relativamente estables, aunque pueden cambiar; pueden ser diferentes en situaciones diferentes; son susceptibles de mejorarse; y cuando a los alumnos se les enseña según su propio estilo de aprendizaje, aprenden con más efectividad.

Las características de los distintos grupos de estilos de aprendizaje, como son el aprendizaje visual, el auditivo y el aprendizaje kinestésico. Cada alumno puede presentar uno de estos estilos o una combinación de ellos. El aprendizaje visual es aquel que permite que el alumno aprenda de manera más eficiente cuando utiliza el contacto visual, de tal manera que percibe mejor la información observando imágenes o leyendo, ya que sus ojos se convierten en la principal herramienta de aprendizaje (Navarro, 2008).

También nos explica sobre el estilo de aprendizaje auditivo, con el cual, los estudiantes retienen la información con mayor eficiencia, si se les presenta a manera de audio o sonidos. Tienen habilidad para recordar la información de manera secuencial. Debido a esto, podemos deducir que los estudiantes que presentan este estilo de aprendizaje son los que mejor rendimiento tienen en la etapa escolar, ya que el esquema social educativo en la actualidad trata sobre el docente, explicando un tema, y los alumnos anotan lo que escuchan del profesor. Por tanto, no podemos decir que los estudiantes con mejor rendimiento académico son necesariamente mejores alumnos o estudiantes, sino que son los que mejor responden al estímulo con el cual se les está enseñando, y si se desea brindar una enseñanza que llegue a todos los estudiantes, debe adaptarse a todos los estilos de aprendizaje que presenten los alumnos.

Navarro también nos menciona un tercer estilo de aprendizaje, que es el kinestésico, que son aquellos que pueden almacenar y comprender información de mejor manera al asociar el conocimiento con movimientos o sensaciones corporales, puesto que aprenden preferentemente interactuando con objetos. Menciona que el aprendizaje de estos estudiantes es un poco más lento que el resto, puesto que en nuestra sociedad, el estudiante ha sido acostumbrado a percibir el conocimiento de manera auditiva o visual.

2.2.3 Psicología Educativa

Bravo Luis (2009), nos habla de la psicología educacional, y nos dice que emergió con el siglo XX de manera paralela al desarrollo de la psicología, como ciencia autónoma, derivada de la psicología general, y su objetivo fue aportar a la educación los avances que aparecían en el campo de la experimentación de los procesos del aprendizaje, los conocimientos sobre el desarrollo infantil y los estudios diferenciales y posteriormente sobre los procesos cognitivos involucrados en el aprendizaje escolar.

Coll (1991) aportó un modelo científico de aproximación a los procesos educacionales, fue origen de que cincuenta años más tarde llegara “a ocuparse progresivamente de todos los problemas y aspectos relevantes el fenómeno educativo”, lo cual hizo que sus contenidos se extendieran enormemente y sus límites se desdibujaran, perdiendo parcialmente su identidad inicial.

Burrhus F. Skinner determinó que el cerebro debe ser estimulado (entradas) para obtener respuestas (salidas), cuando las respuestas son correctas se debe otorgar un refuerzo positivo, un premio. Por ello, Skinner diseñó una máquina de enseñar, la cual se encargaba de plantear preguntas al alumno, el cual debía presionar un botón que

correspondía a la respuesta correcta, si acertaba, se mostraba la siguiente pregunta, si se equivocaba, el ejercicio volvía a comenzar. De esta manera, "Cada información nueva suministrada por la máquina da lugar, así, a elecciones que testimonian la comprensión obtenida, con tantas repeticiones como sean necesarias y con un progreso ininterrumpido en caso de éxitos constantes. Cualquier disciplina puede, pues, programarse de acuerdo con ese principio, ya se trate de razonamiento puro o de simple memoria" (Piaget, 1967)

Por otro lado, Jean Piaget considera cuatro métodos básicos de enseñanza:

- Receptivo: En el que el profesor se encarga de brindar el conocimiento.
- Activo: Se adquiere el conocimiento a través de la acción, el estudiante obtiene el conocimiento y lo transforma en una reflexión y abstracción.
- Intuitivo: Se suministra a los alumnos representaciones audiovisuales de actividades, sin conducir a una realización efectiva de éstas.
- Programado: Es un aprendizaje basado en acción y reacción, estímulo y respuesta. Puede enseñar eficientemente un saber, pero no un razonamiento.

El aprendizaje no es la recepción pasiva de la enseñanza, sino un trabajo activo de parte del estudiante, en el que el maestro juega un papel importante apoyando, cuestionando y actuando como modelo o entrenador (Croty, 1997).

2.2.4 Algoritmo

Joyanes (2004) define un algoritmo como un método con un conjunto de instrucciones utilizadas para resolver un problema específico.

Hernández, Lázaro y Dormido, Ros (2001) definen 3 propiedades básicas que debe satisfacer todo algoritmo:

- En primer lugar el algoritmo debe ser correcto, es decir debe solucionar el problema en todos los casos posibles.
- Además, debe estar compuesto por una serie dada de pasos y para cada uno de ellos no debe existir ambigüedad en la definición de cuál es el siguiente.
- El número de pasos debe ser finito, y por supuesto, el algoritmo debe terminar.

Sin embargo, Luis Joyanes (2004) menciona una perspectiva más amplia y detallada, definiendo las siguientes propiedades:

- Especificación precisa de la entrada: La forma más común del algoritmo es una transformación que toma un conjunto de valores de entrada y ejecuta algunas manipulaciones para producir un conjunto de valores de salida. Un algoritmo debe dejar claros el número y tipo de valores de entrada y las condiciones iniciales que deben cumplir esos valores de entrada para conseguir que las operaciones tengan éxito.
- Especificación precisa de cada instrucción: Cada etapa de un algoritmo debe ser definida con precisión. Esto significa que no puede haber ambigüedad sobre las acciones que deban ejecutarse en cada momento. Las ambigüedades de los idiomas propios de los algoritmos (pseudocódigo en inglés, español, etc.) no pueden afectar al algoritmo en sí.
- Exactitud, corrección: Un algoritmo debe ser exacto, correcto. Se debe poder demostrar que el algoritmo resuelve el problema. Con frecuencia, esto se plantea en el formato de un argumento, lógico o matemático, al efecto de que si las condiciones de entrada cumplen y se ejecutan los pasos del algoritmo, entonces se producirá la salida deseada. En otras palabras, se debe calcular la función deseada, convirtiendo cada entrada a la salida correcta. Un algoritmo se espera resuelva un problema.
- Etapas bien definidas y concretas: Un algoritmo se compone de una serie de etapas concretas, concreta significa que la acción descrita por esta etapa está totalmente comprendida por la persona o máquina que debe ejecutar el algoritmo. Cada etapa debe ser ejecutable en una cantidad finita de tiempo. Por consiguiente, el algoritmo nos proporciona una "receta" para resolver el problema en etapas y tiempos concretos.
- Número finito de pasos: Un algoritmo se debe componer de un número finito de pasos. Si la descripción del algoritmo se compone de un número infinito de etapas, nunca se podrá implementar como un programa de computador. La mayoría de los lenguajes que describen algoritmos (español, inglés o pseudocódigo) proporcionan un método para ejecutar acciones repetidas, conocidas como iteraciones que controlan las salidas de bucles o secuencias repetitivas.
- Un algoritmo debe terminar: En otras palabras, no puede entrar en un bucle infinito.
- Descripción del resultado o efecto: Por último, debe estar claro cuál es la tarea que el algoritmo debe ejecutar. La mayoría de las veces, esta condición se expresa por la producción de un valor como resultado que tenga ciertas propiedades. Con menor frecuencia, los algoritmos se ejecutan para un efecto lateral, tal como imprimir un valor en un dispositivo de salida. En cualquier caso la salida esperada debe estar especificada completamente.

2.2.5 Lenguajes de Programación

Rodríguez (2003, 65) nos dice que un lenguaje de programación es un lenguaje artificial que se utiliza para expresar programas de ordenador.

Cada ordenador, según su diseño, "entiende" un cierto conjunto de instrucciones elementales (lenguaje máquina). Nos obstante, para facilitar la tarea del programador, se dispone también de lenguajes de alto nivel más fáciles de manejar y que no dependen del diseño específico de cada ordenador. Los programas escritos en un lenguaje de alto nivel no podrán ser ejecutados por un ordenador mientras no sean traducidos al lenguaje propio de éste.

2.2.5.1 Paradigmas de Programación

Un paradigma de programación es una colección de patrones conceptuales que moldean la forma de razonar sobre los problemas, de formular soluciones y de estructurar programas. Los paradigmas de programación son:

2.2.5.1.1 Programación imperativa

En este paradigma, un programa es una secuencia finita de instrucciones, que se ejecutan una tras otra. Los datos utilizados se almacenan en memoria principal y se referencian utilizando variables. Ejemplo de lenguajes que utilizan este paradigma: Pascal, Ada, Cobol, C, Modula-2 y Fortran.

2.2.5.1.2 Programación funcional

Paradigma en el que todas las sentencias son funciones en el sentido matemático del término. Un programa es una función que se define por composición de funciones más simples. La misión del ordenador será evaluar funciones. Ejemplo de lenguaje: LISP.

2.2.5.1.3 Programación lógica

En este paradigma un programa consiste en declarar una serie de hechos (elementos conocidos, relación de objetos concretos) y reglas (relación general entre objetos que cumplen unas propiedades) y luego preguntar por un resultado. Ejemplo: Prolog.

2.2.5.1.4 Programación orientada a objetos

El paradigma orientado a objetos (OO) se refiere a un estilo de programación. Un lenguaje de programación orientado a objetos (LOO) puede ser tanto imperativo como funcional o lógico. Lo que caracteriza un LOO es la forma de manejar la información que está basada en tres conceptos:

- Clase: Tipo de dato con unas determinadas propiedades y una determinada funcionalidad (ejemplo: clase "persona").
- Objeto: Entidad de una determinada clase con un determinado estado (valores del conjunto de sus propiedades) capaz de interactuar con otros objetos (ejemplos: "Pedro", "Sonia",...).
- Herencia: Propiedad por la que es posible construir nuevas clases a partir de las clases ya existentes (ejemplo: la clase "persona" podría construirse a partir de la case "ser vivo").

Ejemplos de LOO: Smalltalk, C++, Java.

2.2.5 Compiladores

Aho y Ullman (1986) en el muy conocido Libro del Dragón, por su particular portada, nos dicen que, a grandes rasgos, un compilador es un programa que lee un programa escrito en un lenguaje, el lenguaje fuente, y lo traduce a un programa equivalente en otro lenguaje, el lenguaje objeto. Como parte importante de este proceso de traducción, el compilador informa a su usuario de la presencia de errores en el programa fuente.

Muchas herramientas de software que manipulan programas fuente realizan primero algún tipo de análisis. Algunos ejemplos de tales herramientas son:

- Editores de estructuras: Un editor de estructuras toma como entrada una secuencia de órdenes para construir un programa fuente. El editor de estructuras no sólo realiza las funciones de creación y modificación de textos de un editor de textos ordinario, sino que también analiza el texto del programa, imponiendo al programa fuente una estructura jerárquica apropiada. De esa manera, el editor de estructuras puede realizar tareas adicionales útiles para la preparación de programas. Por ejemplo, puede comprobar si la entrada está formada correctamente, puede proporcionar palabras clave de manera automática (por ejemplo, cuando el usuario escribe "while", el editor proporciona el correspondiente "do" y le recuerda al usuario que entre las dos palabras debe ir una condicional) y puede saltar desde un "begin" o un paréntesis izquierdo hasta

su correspondiente “end” o paréntesis derecho. Además, la salida de tal editor suele ser similar a la salida de la fase de análisis de un compilador.

- Impresoras estéticas: Una impresora estética analiza un programa y lo imprime de forma que la estructura del programa resulte claramente visible. Por ejemplo, los comentarios pueden aparecer con un tipo de letra especial, y las proposiciones pueden aparecer con un sangrado proporcional a la profundidad de su anidamiento en la organización jerárquica de las propiedades.
- Verificadores estáticos: Un verificador estático lee un programa, lo analiza e intenta descubrir errores potenciales sin ejecutar el programa. La parte del análisis a menudo es similar a la que se encuentra en los compiladores de optimización (...). Así, un verificador estático puede detectar si hay partes de un programa que nunca se podrán ejecutar o si cierta variable se usa antes de ser definidas. Además, puede detectar errores de lógica, como intentar utilizar una variable real como apuntador, empleando las técnicas de verificación (...).
- Intérpretes: En lugar de producir un programa objeto como resultado de una traducción, un intérprete realiza las operaciones que implica el programa fuente. Para una proposición de asignación, por ejemplo, un intérprete podría construir un árbol (...), y después efectuar las operaciones de los nodos conforme recorre el árbol. (...). Muchas veces los intérpretes se usan para ejecutar lenguajes de órdenes, pues cada operador que se ejecuta en un lenguaje de órdenes suele ser una invocación de una rutina compleja, como un editor o un compilador. Del mismo modo, algunos lenguajes de “muy alto nivel”, como APL, normalmente son interpretados, porque hay muchas cosas sobre los datos, como el tamaño y la forma de las matrices, que no se pueden deducir en el momento de la compilación.

Según los datos obtenidos del famoso “Libro del Dragón”, podemos analizar y deducir por ejemplo, que el lenguaje Java, es un lenguaje interpretado, en el que el lenguaje fuente con extensión *.java, es “compilado” (traducido) a un archivo con extensión *.jar, de tal manera que la JVM (Java Virtual Machine) se encarga, en cada computadora y plataforma diferente, de interpretar este código fuente.

III. MATERIALES Y MÉTODOS

3.1 Diseño de investigación

3.1.1 Tipo de Investigación

La presente investigación es de tipo tecnológica aplicada y cuasi-experimental.

Tecnológica aplicada porque se sustenta a partir de los resultados de un estudio y análisis exhaustivos, que permitirán evaluar los procesos internos, comprenderlos y crear una estructura para luego elaborar un software que se pueda probar, estimar y evaluar.

En cuanto a la técnica de contrastación, ésta es cuasi-experimental, se tomará un solo grupo al que se evaluará antes y después de la aplicación del software.

3.1.2 Hipótesis

Mediante un sistema interactivo basado en un intérprete de algoritmos, se podrá mejorar el método de aprendizaje de los alumnos del curso Fundamentos de Programación.

3.1.3 Variables

En el desarrollo de esta investigación se ha detectado dos variables, la variable independiente es el Sistema interactivo basado en un intérprete de algoritmos, y la variable dependiente es el Método de aprendizaje de los alumnos del curso Fundamentos de Programación.

3.1.4 Indicadores

Los indicadores los podemos visualizar en la Tabla 1.

Tabla 1: Indicadores

Objetivo Específico	Indicador	Definición Conceptual	Unidad de Medida	Instrumento	Definición Operacional
Incrementar el número de estudiantes aprobados en evaluación de estructuras condicionales.	Número de aprobados en estructuras condicionales.	Número de estudiantes del curso Fundamentos de programación que logran una calificación mayor o igual a catorce en la evaluación de estructuras condicionales.	Puntaje (0-20)	Evaluación del docente.	*CantidadAlumnosAprobados*
Incrementar el número de	Número de aprobados en	Número de estudiantes del curso Fundamentos de	Puntaje	Evaluación	*CantidadAlumnosAprobados*

estudiantes aprobados en evaluación de estructuras repetitivas.	estructuras repetitivas.	programación que logran una calificación mayor o igual a catorce en la evaluación de estructuras repetitivas.	(0-20)	del docente.	
Disminuir el tiempo promedio de resolución de algoritmos.	Tiempo de resolución de algoritmos.	Tiempo promedio de resolución de algoritmos. Se mide desde el momento en que el estudiante recibe el ejercicio hasta que lo entrega para evaluación.	Minutos.	Registro de tiempos por estudiante.	$\frac{\sum(TiempoPorAlumno)}{CantidadAlumnos}$
Incrementar el porcentaje de satisfacción con el método de enseñanza-aprendizaje.	Porcentaje de satisfacción.	Porcentaje de satisfacción con el método de enseñanza - aprendizaje. Se mide según el resultado de una encuesta.	Porcentaje.	Encuesta.	$\frac{\sum(PorcentajePorAlumno)}{CantidadAlumnos}$

3.1.5 Población y muestra

La población, para esta investigación, es el conjunto de estudiantes del curso Fundamentos de Programación y el docente. Para poder comprobar los resultados según la hipótesis planteada, y según el tipo de investigación, se evaluarán los indicadores antes y después de aplicar el software.

3.1.6 Métodos y técnicas de recolección de datos

Las técnicas a utilizar las podemos observar en la Tabla 2.

Tabla 2: Métodos y técnicas de recolección de datos

Técnica	**Instrumento**	Descripción
Observación	Consolidados.	Se presenció las clases del curso Fundamentos de programación, donde se pudo apreciar y comprobar algunos de los problemas descritos en esta investigación.
Documental	Libros, artículos científicos, tesis, sitios web.	Se visitó diferentes bibliotecas virtuales así como también bases de datos, libros, tesis, artículos científicos y trabajos de investigación; de las cuales se obtuvo información del tema de estudio.
De campo	Encuestas.	Se realizaron encuestas para determinar cuáles son los problemas o qué está faltando

		en el proceso de enseñanza-aprendizaje de algoritmos.
De campo	Entrevistas.	Se realizó una entrevista con la cual se pudo tener conocimiento de las opiniones de los docentes del curso Fundamentos de Programación y cursos dependientes del mismo (Estructuras de Datos, Metodologías de Programación), en cuanto al proceso de enseñanza-aprendizaje de algoritmos.

3.2 Metodología

Para el desarrollo de este proyecto, se decidió trabajar bajo el marco de trabajo Proceso Unificado Ágil (AUP, por sus siglas en inglés Agile Unified Process), el cual es una versión simplificada del Proceso Unificado Rational (RUP, por sus siglas en inglés Rational Unified Process), desarrollado por Scott Ambler. Describe un enfoque simple y fácil de entender, para desarrollar software usando técnicas y conceptos ágiles pero sin perder la estructura de RUP. Algunas de las técnicas ágiles aplicadas en el Proceso Unificado Ágil para mejorar la productividad son: Desarrollo basado en pruebas (TDD, por sus siglas en inglés Test-Driven Development), modelado ágil (AM, por sus siglas en inglés Agile Modeling), gestión de cambios ágil y reingeniería de base de datos.

Muchos autores coinciden con que AUP es serial en lo grande, e iterativo en lo pequeño, debido a la estructura general de este marco de trabajo, ya que en la línea del tiempo, el proyecto debe ser dividido en cuatro fases, cada fase tiene iteraciones del proyecto, las cuales están elaboradas en base a las disciplinas descritas por el marco de trabajo.

3.2.1 Fases

Se dice que el Proceso Unificado Ágil es serial en lo largo, puesto que todo el proyecto se divide en cuatro grandes fases: Iniciación, elaboración, construcción y transición; cada una de las cuales tiene una serie de actividades principales e hitos que marcan el avance al que se debe llegar en cada iteración (Ver Figura 1).

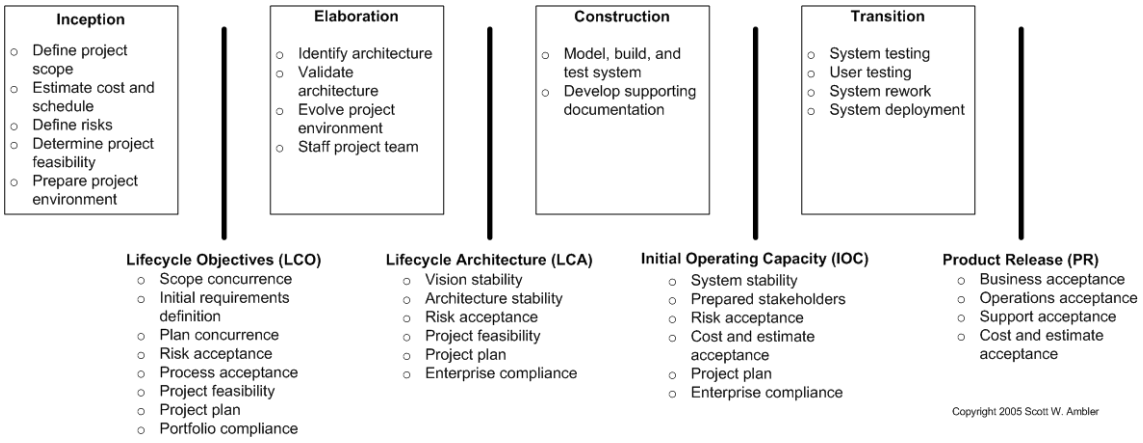

Figura 1: Fases de AUP

3.2.1.1 Iniciación

Esta fase tiene como meta principal la definición de objetivos y el alcance del proyecto así como la estimación general de costos y cronograma. De manera general se definen las siguientes actividades:

3.2.1.1.1 Definir el alcance del proyecto

Incluye la definición, a un alto nivel, de lo que el sistema hará. Igualmente importante es definir lo que el sistema no hará, de tal manera que se establezcan los límites con los que el equipo trabajará. Esto usualmente se trabaja como una lista de características de alto nivel y/o de casos de uso.

3.2.1.1.2 Estimar costos y cronograma

El cronograma y los costos del proyecto deben ser estimados a un alto nivel, es decir, de manera muy general. Debido a que los proyectos no pueden ser estimados de manera completa en un solo paso, es necesario planificar a detalle el futuro cercano, y de manera más general el futuro lejano.

3.2.1.1.3 Definir riesgos

Los riesgos del proyecto son definidos por primera vez aquí. La gestión de riesgos es importante en un proyecto de Proceso Unificado Ágil, la lista de riesgos es una compilación que cambiará con el tiempo conforme los riesgos van siendo identificados, mitigados, evitados y/o materializado y tratado. Los riesgos conducen la gestión del proyecto, como los riesgos de alta prioridad conducen la programación de iteraciones. Por ejemplo: Los riesgos con mayor prioridad se abordan en las iteraciones anteriores de los riesgos de menor prioridad.

3.2.1.1.4 Determinar la viabilidad del proyecto

El proyecto debe tener sentido en las perspectivas: técnica, operacional y de negocio. En otras palabras, hay que ser capaz de construirlo, y una vez construido, hay que ser capaz de ejecutarlo, y para hacer esto debe tener un sentido económico. Si el proyecto no es viable, debe ser cancelado.

3.2.1.1.5 Preparar el entorno del proyecto

Esto quiere decir, reservar los espacios de trabajo para el equipo, requerir a las personas cuando van a ser necesitadas, obtener hardware y software que se necesiten inmediatamente, y compilar una lista de hardware y software anticipado que serán necesitados después. Además se debe ajustar el Proceso Unificado Ágil para adaptarlo a las necesidades del equipo y del proyecto.

3.2.1.2 Elaboración

La primera meta de la fase de Elaboración es probar la arquitectura del sistema a ser desarrollado. El punto es asegurar que el equipo pueda realmente desarrollar un sistema que satisfaga los requerimientos, y la mejor forma de hacer eso es construir un esqueleto del sistema de extremo a extremo.

Es importante notar que los requerimientos no son especificados completamente en este punto. Estos son detallados lo suficiente para entender los riesgos de la arquitectura para asegurar que hay un entendimiento del alcance de cada requerimiento. Los riesgos de arquitectura son identificados y priorizados; los más significantes son abordados durante la elaboración.

Durante la etapa de la Elaboración, el equipo también debe prepararse para la fase de Construcción. A medida que el equipo va avanzando con la arquitectura del sistema, debe ir configurando el entorno para la Construcción, comprando hardware, software y herramientas. Para un punto de vista de gestión de proyectos, se aborda la contratación de personal, los recursos son requeridos y/o contratados.

3.2.1.3 Construcción

El objetivo de la fase de construcción es desarrollar el sistema hasta el punto donde esté listo para las pruebas de pre - producción. En las fases anteriores, la mayoría de los requisitos tienen que ser identificados y la arquitectura para el sistema ha sido establecida.

3.2.1.4 Transición

La fase de Transición se enfoca en liberar el sistema a producción. Deben hacerse pruebas extensivas a lo largo de esta fase, incluyendo las pruebas beta. El tiempo y esfuerzos necesarios en la Transición varían de un proyecto a otro.

Por lo general, los sistemas internos son más simples de desplegar que los sistemas externos; la liberación de un nuevo sistema de marca puede traer consigo la compra y configuración de hardware mientras se actualiza un sistema existente que también puede traer una conversión de datos y una coordinación exhaustiva con la comunidad de usuarios. Cada proyecto es diferente, es único.

3.2.2 Disciplinas

A diferencia de RUP, el Proceso Unificado Ágil tiene sólo siete disciplinas:

- **Modelado:** Entender el negocio de la organización, el dominio del problema que será abordado, y encontrar una solución viable para abordar el problema.
- **Implementación:** Transformar el/los modelo(s) en código ejecutable y realizar un nivel básico de pruebas.
- **Pruebas:** Realizar una evaluación objetiva para asegurar la calidad. Esto incluye encontrar defectos, validar que el sistema trabaje como fue diseñado, y verificar que los requerimientos se hayan cumplido.
- **Despliegue:** Planificar la entrega del sistema y ejecutar el plan para hacer al sistema disponible para el usuario final.
- **Gestión de la configuración:** Gestionar el acceso a los artefactos del proyecto. Esto incluye no sólo el seguimiento de las versiones de los artefactos en el tiempo, sino también controlar y gestionar sus cambios.
- **Gestión del proyecto:** Dirigir las actividades que se desarrollarán dentro del proyecto. Esto incluye gestión de riesgos, dirección de personas (asignación de actividades, seguimiento de los progresos, etc.) y coordinar con las personas y sistemas fuera del proyecto para asegurarse de que están entregando dentro del plazo y presupuesto establecido.
- **Entorno:** Garantizar que el proceso, la orientación (normas y directrices) y las herramientas (hardware, software, etc.) estén disponibles para el equipo según sea necesario.

3.2.3 Hitos

Existen cuatro hitos en el Proceso Unificado Ágil, en cada uno de éstos hitos, los cuales señalan el final de una fase, se debería considerar tener una "revisión de hitos" para verificar que el equipo de trabajo está cumpliendo satisfactoriamente con los criterios de hitos, los cuatro hitos son: Objetivos del ciclo de vida, arquitectura del ciclo de vida, capacidad operacional inicial, liberación del producto.

3.2.3.1 Fase de inicio: Objetivos del ciclo de vida

- **Acuerdo del Alcance:** Los interesados llegan a un acuerdo sobre el alcance del proyecto.
- **Definición Inicial de Requerimientos:** Existe un acuerdo en que el conjunto correcto de requisitos han sido capturados, en un nivel alto, y hay un entendimiento común de estos requisitos.
- **Acuerdo del Plan:** Los involucrados llegan a un acuerdo con el costo inicial y la estimación del cronograma.
- **Aceptación del Riesgo:** El riesgo ha sido identificado, evaluado y se han abordado estrategias aceptables para controlarlos.
- **Aceptación del Proceso:** La metodología del Proceso Unificado Ágil ha sido inicialmente adoptado y aceptado por todas las partes.
- **Viabilidad:** El proyecto tiene sentido desde la perspectiva técnica, operacional y del negocio.
- **Plan del proyecto:** Existen adecuados planes para la siguiente fase.
- **Cumplimiento del Portafolio:** ¿El alcance del proyecto encaja bien en su organización general del portafolio de proyectos?

3.2.3.2 Fase de elaboración: Arquitectura del ciclo de vida

- **Estabilidad de la visión:** La visión del proyecto ha sido estabilizada y es realista.
- **Estabilidad de la arquitectura:** Está de acuerdo en que la arquitectura está estable y es suficiente para satisfacer los requerimientos. La arquitectura ha sido prototipada apropiadamente para ser direccionada con los riesgos principales de la arquitectura.
- **Aceptación del riesgo:** El riesgo ha sido evaluado para asegurar que ha sido apropiadamente entendido, documentado y que se han desarrollado estrategias para manejarlo como aceptable.

- **Viabilidad:** El proyecto aún tiene sentido desde la perspectiva técnica operacional y del negocio.
- **Plan del proyecto:** Plan de iteración detallado para las próximas iteraciones de la etapa de Construcción, así como un plan de proyecto de alto nivel ya elaborado.
- **Cumplimiento de la organización:** ¿La arquitectura del sistema refleja las realidades de la arquitectura de la empresa?

3.2.3.3 Fase de construcción: Capacidad Operacional Inicial

- **Estabilidad del Sistema:** El software y la documentación de soporte son aceptables (estable y madura) para implementar el sistema a los usuarios.
- **Involucrados preparados:** Los involucrados (y el negocio) están listos para que el sistema sea implementado (aunque aún necesiten entrenamiento).
- **Aceptación del riesgo:** El riesgo ha sido evaluado para asegurar que ha sido apropiadamente entendido, documentado y que se han desarrollado estrategias para manejarlo como aceptable.
- **Aceptación y estimación del costo:** Los gastos son aceptables y las estimaciones razonables han sido calculadas y programadas para los costos futuros.
- **Plan del proyecto:** Plan de iteración detallado para las próximas iteraciones de la etapa de Transición, así como un plan de proyecto de alto nivel ya elaborado.
- **Cumplimiento de la organización:** ¿El producto elaborado por el equipo cumple con los estándares apropiados de la organización?

3.2.3.4 Fase de transición: Liberación del Producto

- **Aceptación de los involucrados del negocio:** Los involucrados del negocio están satisfechos con el sistema y lo aceptan.
- **Operación de aceptación:** Las personas se responsabilizan de operar el sistema una vez que éste está en producción y están satisfechos con los procedimientos y documentación relevantes.
- **Aceptación del soporte:** Las personas se responsabilizan del soporte del sistema una vez que éste está en producción y están satisfechos con los procedimientos y documentación relevantes.
- **Aceptación del costo estimado:** Los gastos actuales son aceptados, y las estimaciones razonables han sido hechas para los costos futuros de producción.

3.2.4 Roles

Antes de definir la tabla de roles (

Tabla 3), hay que entender unos asuntos importantes:

1. Los roles pueden ser asumidos por varias personas.
2. Una persona puede tomar varios roles.
3. Un rol no es un puesto.
4. Hay que tratar de convertirse en un **especialista general** que domine una o más especialidades (por ejemplo, administración de base de datos, administración de proyectos, entre otras), un conocimiento general de todo el proceso del software y una gran comprensión del dominio de sus labores.

Tabla 3: Tabla de roles del proyecto

Rol	**Descripción**	**Disciplina(s)**
DBA Ágil	Un administrador de bases de datos (DBA) que trabaja de manera colaborativa con los integrantes del equipo del proyecto para diseñar, probar y brindar soporte a los diferentes esquemas de datos.	Implementación
Modelador Ágil	Alguien que cree y desarrolle modelos, ya sean dibujos, tarjetas, o archivos complejos realizados con herramientas CASE, de manera colaborativa y evolutiva. Los modelos ágiles son apenas lo suficientemente buenos.	Modelado Implementación
Cualquiera	Cualquier otra persona en otro rol distinto.	Administración de la configuración Administración del Proyecto
Administrador de la configuración	Un administrador de configuración se encarga de proporcionar la infraestructura y crear el medio ambiente para el equipo de desarrollo.	Administración de la configuración
Implementador	Un implementador es responsable de disponer el sistema en los ambientes de pre-producción y producción.	Desarrollo
Desarrollador	Es quien escribe código, realiza pruebas y construye el software.	Modelado Implementación Desarrollo

Especialista del proceso	Desarrolla, adapta y apoya el material de los procesos de la organización (descripción de procesos, plantillas, guías, ejemplos, entre otros).	Entorno
Administrador del proyecto	Administra los miembros de los equipos de trabajo, crea relaciones con los involucrados, coordina las interacciones con los involucrados, planea, administra y dispone recursos, enmarca prioridades y mantiene el equipo enfocado.	Modelado Pruebas Desarrollo Administración del proyecto
Examinador	Evalúa los productos del proyecto, inclusive "el trabajo en progreso", suministrando retroalimentación al equipo de trabajo.	Pruebas
Involucrado	Un involucrado es cualquiera que sea usuario directo, usuario indirecto, administrador de usuarios, administrador, miembro de equipo de operación o soporte, desarrolladores que trabajan en otros sistemas que se integran o interactúan con el sistema implementado, en fin, todo aquel que se vea afectado de una u otra forma con el proyecto.	Modelado Implementación Pruebas Desarrollo Administración del proyecto
Documentador técnico	Es responsable de producir documentación para los involucrados, tal como: materiales de capacitación, documentación de operaciones, documentación de mantenimiento, y documentación de usuario.	Desarrollo
Administrador de pruebas	El administrador de pruebas es el responsable del éxito de las pruebas, incluye planificar la administración y promover las pruebas y las actividades de calidad.	Pruebas
Equipo de pruebas	El equipo de pruebas es responsable de ejecutar las pruebas y documentar los resultados que proyecten.	Pruebas
Especialista en herramientas	Es responsable de seleccionar, adquirir, configurar y brindar mantenimiento al equipo requerido.	Entorno

IV. RESULTADOS

4.1 Inicio

4.1.1 Definición del alcance

En esta etapa se elaboró los requerimientos funcionales del software. Según los objetivos que se desean cumplir, y la propuesta mencionada en anteriores capítulos, se definió los siguientes requisitos funcionales:

1. Debe presentar varios niveles de dificultad, según la estructura del sílabo del curso Fundamentos de Programación, para que los alumnos puedan practicar según lo que el profesor enseña en clase.
2. Debe tener varios escenarios por nivel, de manera que puedan practicar con diversos ejercicios cada tema.
3. El usuario debe poder elegir el nivel en el que desea practicar, de ninguna manera debe ser obligado a seguir una secuencia. Los niveles presentados al usuario sólo servirán como una guía, siendo el alumno quien elige qué practicar.
4. El compilador debe mostrar al usuario información de errores de programación como el tipo de error, la ubicación y el mensaje de error, para que el usuario pueda resolver el problema sin perder tiempo en ubicar el error.
5. El pseudolenguaje utilizará en su totalidad palabras reservadas en español, para reducir el tiempo de aprendizaje del lenguaje.
6. El sistema mostrará información teórica en cada nivel, para recordar al alumno el tema visto en clase.
7. El sistema tendrá una función de ayuda en la que se mostrará al usuario algunas funciones o estructuras que se podrán usar en cada nivel.
8. El editor de texto permitirá abrir y guardar los archivos con el código (algoritmo) de los escenarios desarrollados por el usuario, para facilitar la revisión de código a posteriori.
9. Con cada ejecución correcta (logro del objetivo en un escenario) se suman puntos de experiencia al usuario.
10. El compilador no calificará el algoritmo del alumno, sólo verificará que sea eficaz, es decir, que cumpla el objetivo.

Adicionalmente, se especificó los siguientes requerimientos no funcionales:

1. El software debe ser portable, no requerirá instalación.
2. El software debe ser multiplataforma, deberá ejecutarse principalmente en GNU/Linux, Windows y Mac OS.

4.1.2 Elaboración del cronograma

En este punto, se calculan los tiempos de manera general, para ello tomo en cuenta las fases de intérpretes según Alfred V. Aho: Análisis léxico, análisis sintáctico, análisis semántico, generación de código intermedio, ejecución. Además se consideró dos etapas para el desarrollo del "videojuego" y una etapa de unión de ambas partes (compilador y videojuego). Las actividades fueron programadas a 3 (tres) meses. El cronograma se puede consultar en el Anexo 5.

4.1.2 Elaboración del presupuesto

Para el desarrollo del proyecto, se utilizará una computadora portátil con las siguientes características:

- Marca: Lenovo.
- Modelo: V470.
- Procesador: Intel Core i3 2.8GHz.
- Memoria RAM: 4GB.
- Disco Duro: 500GB.

La elaboración de este proyecto tiene un costo calculado en S/. 4339.90 (Ver Anexo 4). Para la elaboración del presupuesto de manera general, tomamos en cuenta los gastos de la elaboración del proyecto (computador y mano de obra), así como gastos en útiles de oficina y gastos personales como pasajes y alimentación.

4.1.3 Definición de riesgos

En la etapa de inicio nos limitamos a la definición de los riesgos del proyecto, abordando únicamente los riesgos más importantes que deben ser mitigados en este punto.

4.1.3.1 Identificación

En la presente investigación, se tomó en cuenta la posibilidad de ocurrencia de riesgos técnicos (que tienen que ver con las herramientas físicas de trabajo), así como los riesgos no técnicos, como los que dependen de las personas, con mayor detalle se listan a continuación:

1. Al trabajar con una computadora de uso personal, es posible que se deteriore y que surjan problemas técnicos como la avería de componentes como el disco duro, lo que causaría la pérdida de la información, más específicamente, de los archivos del proyecto de software.

2. Debido a que el autor de esta investigación no desarrolló con anterioridad proyectos de compiladores o de videojuegos, es posible que se elija librerías de terceros y que en la práctica se descubra que las mismas no se puedan adaptar completamente al proyecto, ocasionando un grave retraso en el desarrollo del software.

3. Dado que la Universidad Católica Santo Toribio de Mogrovejo tiene dentro de sus máximas la mejora continua, puede ocurrir que exista una mejora autónoma del proceso de enseñanza – aprendizaje, es decir, que la Escuela de Ingeniería de Sistemas y Computación encuentre otra solución a los problemas planteados en esta investigación y la ponga en práctica, haciendo de alguna manera que el software desarrollado sea innecesario.

4.1.3.2 Matriz de probabilidad

La matriz de probabilidad desarrollada la podemos apreciar en la Tabla 4.

Tabla 4: Matriz de probabilidad de ocurrencia de riesgos

Código	Riesgo	Probabilidad
1	Avería de componentes de la computadora portátil (disco duro), causando pérdida de información.	0.8
2	Incompatibilidad de las librerías usadas con el proyecto final, causando retraso en el desarrollo.	0.5
3	Mejora autónoma del proceso enseñanza – aprendizaje, causando que el software desarrollado sea obsoleto.	0.5

4.1.3.3 Matriz de impacto

La matriz de impacto desarrollada la podemos apreciar en la Tabla 5.

Tabla 5: Matriz de impacto de riesgos

Código	Riesgo	Impacto
1	Avería de componentes de la computadora portátil (disco duro), causando pérdida de información.	1
2	Incompatibilidad de las librerías usadas con el proyecto final, causando retraso en el desarrollo.	0.5
3	Mejora autónoma del proceso enseñanza – aprendizaje, causando que el software desarrollado sea obsoleto.	0.8

4.1.3.4 Matriz de importancia

La matriz de importancia resultante la podemos apreciar en la Tabla 6.

Tabla 6: Matriz de importancia de riesgos

Código	Riesgo	Importancia
1	Avería de componentes de la computadora portátil (disco duro), causando pérdida de información.	0.8
2	Incompatibilidad de las librerías usadas con el proyecto final, causando retraso en el desarrollo.	0.25
3	Mejora autónoma del proceso enseñanza – aprendizaje, causando que el software desarrollado sea obsoleto.	0.4

Calificamos como riesgo alto, si la importancia resultante es igual o mayor a 0.7.

Calificamos como riesgo medio, si la importancia resultante es mayor a 0.3 y menor que 0.7.

Calificamos como riesgo bajo, si la importancia resultante es igual o menor a 0.3.

4.1.4 Viabilidad del proyecto

Para evaluar la viabilidad del proyecto evaluamos los planes de contingencia para los riesgos medios y altos. Los planes de contingencia para los riesgos bajos serán evaluados en la etapa de elaboración.

Como vimos anteriormente, contamos con un riesgo alto y un riesgo medio. Por su naturaleza ninguno de ellos puede prevenirse, por lo tanto se ha optado como mejor opción la mitigación de ambos riesgos.

En el caso del riesgo 1: Avería de componentes de la computadora portátil (disco duro), causando pérdida de la información:

- La computadora hará uso del sistema operativo GNU/Linux, reduciendo drásticamente la posibilidad de infección por virus informático que pueda dañar los componentes de la computadora (por sobrecalentamiento).
- Con cada actualización del software y de los archivos de la documentación técnica así como del presente informe, se subirá a una cuenta en Google Drive y a una cuenta en el servicio de alojamiento de archivos Mega.

De esta manera, reducimos la probabilidad de ocurrencia y el impacto del riesgo 1.

En el caso del riesgo 3: Mejora autónoma del proceso enseñanza – aprendizaje, causando que el software desarrollado sea obsoleto:

- Debido a que la mejora del proceso es algo positivo para el caso de aplicación de la presente investigación, de ninguna manera podemos tratar de evitar o mitigar este "riesgo", por lo tanto, se consignó agregar un requisito: El software debe permitir añadir escenarios, de tal manera que siempre se ajuste a lo especificado por el docente, y a las necesidades de práctica del estudiante.

De esta manera reducimos el impacto del riesgo 3.

4.1.5 Preparación del entorno del proyecto

Antes de comenzar con el desarrollo del software, haremos un breve análisis de las herramientas que necesitaremos, de tal manera que estén disponibles cuando se requiera utilizarlas, para ello hemos dividido el proyecto en tres grandes etapas: La construcción del intérprete, la construcción del sistema interactivo, la unión del intérprete y el sistema interactivo. Estas etapas no están necesariamente ordenadas cronológicamente, por ejemplo, las actividades de la etapa de unión del intérprete y el sistema interactivo se van ejecutando a medida que se desarrollan la primera y segunda etapa, esto lo veremos con detenimiento más adelante. Ahora veamos la lista de "fases" de cada etapa.

En primer lugar se presenta una lista de las fases de la construcción del intérprete:

- Analizador Léxico.
- Analizador Sintáctico.
- Analizador Semántico.
- Generación de código.
- Ejecución.

En segundo lugar se presenta una lista con las fases la de construcción del sistema interactivo:

- Definición de clases y elementos.
- Representación gráfica y animación.

- Constructor de escenarios.

Por último, listamos las fases en la unión del compilador con el sistema interactivo:

- Construcción del editor.
- Construcción del ejecutor (Ejecuta el intérprete y envía las órdenes al sistema interactivo)

Basados en la experiencia en construcción de software diverso, pero específicamente en aplicaciones referentes a la computación gráfica y a compiladores, se cree indispensable utilizar librerías para la primera y segunda etapa.

Como podemos observar en los requerimientos, se desea que el programa sea portable hacia cualquier plataforma de escritorio, o ante las más usadas en el mercado actual, como Windows, GNU/Linux y Mac, por lo tanto, se cree conveniente utilizar el lenguaje de programación JAVA, por sus características de ejecución multiplataforma, además para demostrar a la comunidad de desarrolladores de software la potencia que tiene la máquina virtual de Java (JVM), cuyos tiempos de ejecución promedio de aplicaciones es bastante rápido (subjetivamente) siendo ésta un intérprete. Dicho esto, tenemos claro que para algunas de estas fases, (llamadas en adelante "iteraciones") necesitaremos librerías o software adicional de/para el lenguaje de programación Java.

- Analizador Léxico: Luego de revisar las alternativas que ofrece la web, se decidió trabajar con la herramienta JFlex, por la facilidad para encontrar documentación en línea, por la gran similitud en la sintaxis comparada con la herramienta flex (herramienta que el autor de esta investigación utilizó en un curso de compiladores) y por el prestigio ganado entre los desarrolladores. Entre sus características tenemos que el resultado (un analizador léxico) es compatible con la herramienta CUP.
- Analizador Sintáctico: Debido a que la herramienta que utilizaremos en la construcción del analizador léxico se complementa perfectamente con la herramienta de generación de analizadores sintácticos CUP, se decidió aprovechar esta característica y utilizar la herramienta CUP.
- Representación Gráfica y animación: Se buscó en la web entre varias herramientas como motores gráficos y generadores de videojuegos, pero se decidió trabajar con la herramienta JSlick 2D, una librería en lenguaje Java que permite crear gráficos y animaciones de manera sencilla en un proyecto de escritorio en Java, sin quitar el trabajo del pensamiento lógico al desarrollador.

4.2 Elaboración

Como se mencionó anteriormente, el desarrollo del proyecto está dividido en tres grandes etapas, el desarrollo del compilador, el desarrollo del sistema multimedia, y la unión del intérprete al sistema multimedia. Para ello, mostramos la arquitectura del proyecto en la Figura 2, en la que se aprecian claramente las dos primeras etapas, y de manera abstracta la tercera etapa.

En esta etapa de la gestión del proyecto, ya se ha analizado con detalle la librería Slick 2D, y no se ha encontrado funcionalidades que puedan en un futuro ser incompatibles, con lo que reducimos la probabilidad de ocurrencia del riesgo 2: Incompatibilidad de las librerías usadas con el proyecto final, causando un retraso en el desarrollo del software.

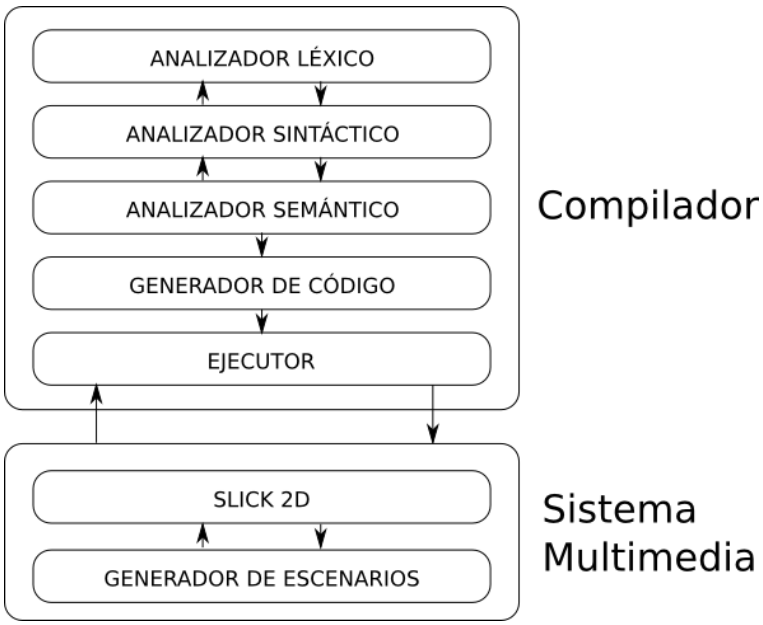

Figura 2: Arquitectura general del proyecto

A continuación, se muestra el prototipo inicial y a manera de bosquejo de las interfaces de usuario, se muestran en orden cronológico tal como se fueron elaborando en el desarrollo del proyecto (Figura 3 y Figura 4):

1. Editor Pizarra.

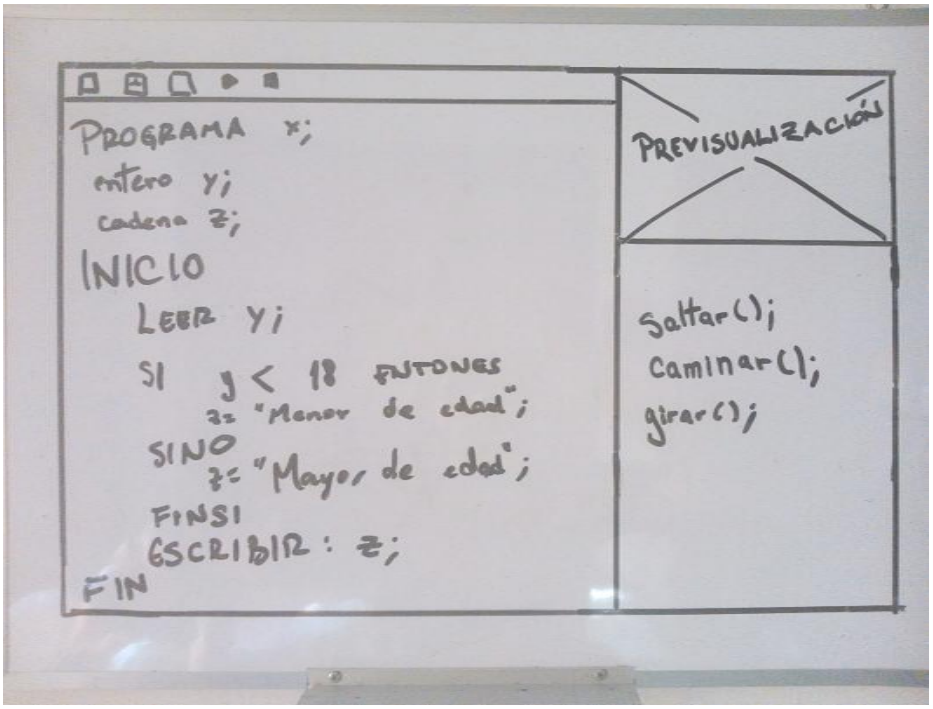

Figura 3: Bosquejo del editor

2. Escenario Pizarra.

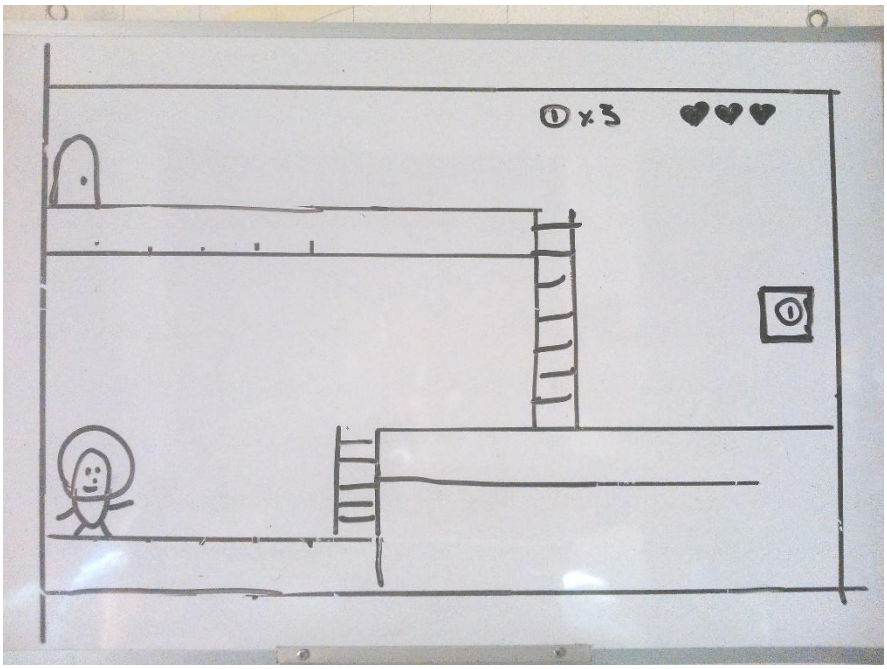

Figura 4: Bosquejo de escenario

Por último, en la etapa de elaboración, se buscó en la web recursos libres y gratuitos como sprites (imágenes utilizadas en el videojuego) y sonidos. Las imágenes principales se muestran a continuación (Figura 5, Figura 6, Figura 7, Figura 8):

1. Personaje

Figura 5: Personaje

2. Bloques

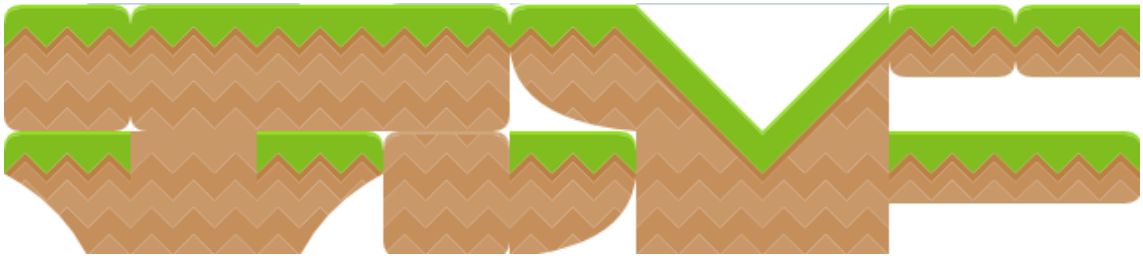

Figura 6: Bloques de piso

3. Fondo

Figura 7: Fondo de escenario

4. Puertas, palancas, escaleras, etc.

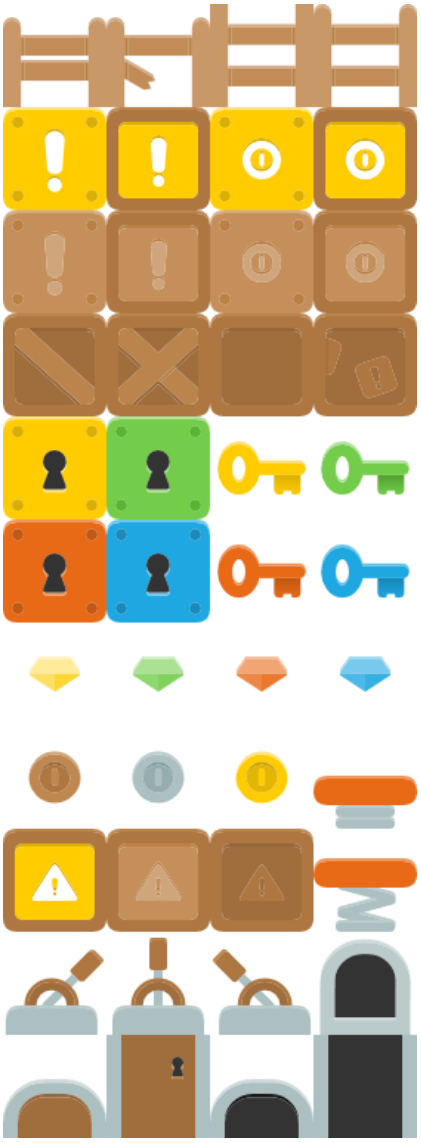

Figura 8: Elementos de interacción

4.3 Construcción

La construcción del proyecto está dividida en diez fases, se decidió que cada fase formará una iteración dentro del desarrollo del proyecto.

4.3.1 Iteración 1: Analizador Léxico

4.3.1.1 Análisis y diseño

En primer lugar revisemos los requerimientos funcionales que se interrelacionan con esta iteración:

1. El compilador debe mostrar al usuario información de errores de programación como el tipo de error, la ubicación y el mensaje de error, para que el usuario pueda resolver el problema sin perder tiempo en ubicar el error.
2. El pseudolenguaje utilizará en su totalidad palabras reservadas en español, para reducir el tiempo de aprendizaje el lenguaje.

Ahora listamos los requisitos del lenguaje:

- Entrada de datos
- Asignación
- Control del flujo
    - SI
    - PARA
    - MIENTRAS
    - HACER
- Salida de datos
- Tipos de datos
    - CARÁCTER
    - CADENA
    - ENTERO
    - REAL
    - LÓGICO
- Operaciones
    - CONCATENACIÓN: +
    - ARITMÉTICAS: +, -, *, /, ^
    - RELACIONALES: >, <, >=, <=, ==, !=
    - LÓGICAS: &&, ||, !
- Declaración de variables

Reglas del lenguaje:

- Toda variable debe ser previamente declarada.
- La variable y la expresión asignada deben ser del mismo tipo.
- Cada sentencia debe terminar en “;”,

Ejemplo de código

```
PROGRAMA programa_1;
entero i;
INICIO
      i = 0;
      MIENTRAS ( i <= 5 ) HACER
            caminar();
            i = i + 1;
      FINMIENTRAS

      saltar_obstaculo();
FIN
```

Una vez que hemos analizado los requisitos del lenguaje, y el ejemplo, pasamos a crear una tabla de lexemas y tokens, como podemos ver en la Tabla 7.

Tabla 7: Tabla de lexemas y tokens

LEXEMA	**TOKEN**
PROGRAMA	PROGRAMA
nombre ejemplo3	IDENTIFICADOR
;	;
:	:
cadena	CADENA
entero	ENTERO
Real	REAL
carácter	CARÁCTER
Leer	LEER
escribir	ESCRIBIR
INICIO	INICIO
FIN	FIN
SI	SI
ENTONCES	ENTONCES
SINO	SINO
FINSI	FINSI
PARA	PARA
FINPARA	FINPARA

MIENTRAS	MIENTRAS
HACER	HACER
FINMIENTRAS	FINMIENTRAS
-	-
+	+
*	*
/	/
^	^
>	>
<	<
>=	MAYIG
<=	MENIG
==	IGUAL
!=	DIFERENTE
&&	Y
||	O
!	!
“ingresa tu nombre:” “Bienvenido a MonoCode”	CTE_CADENA
18	CTE_ENTERO
2.4 3.2	CTE_REAL
'a' '4' 'x'	CTE_CARACTER
verdadero falso	CTE_LOGICA
(	(
)	)
=	=
,	,

Ahora que ya tenemos identificados los tokens, procedemos a definir los patrones.

Las palabras reservadas no necesitan un patrón, puesto que se utiliza la misma palabra para el reconocimiento, tampoco los tokens cuyo lexema es de un sólo carácter, puesto que utilizamos el mismo carácter, esto lo podemos apreciar en la Tabla 8.

Tabla 8: Tokens y patrones

TOKEN	PATRÓN
LETRA	[a-zA-Z]
DIGITO	[0-9]
IDENTIFICADOR	letra (letra | digito | “_”)*
CTE_ENTERO	digito+
CTE_REAL	digito+ “.” digito+
CTE_CADENA	“ [^” \n]* “
CTE_CARACTER	' ([^] | \n | \t) '
CTE_LOGICA	“verdadero”|”falso”
Y	“&&”
O	“||”
MAYIG	“>=”
MENIG	“<=”
IGUAL	“==”
DIFERENTE	“!=”

4.3.1.2 Desarrollo

Cabe señalar que para el desarrollo se utilizará la herramienta Sublime Text como editor de texto.

Antes de crear el archivo de entrada para la herramienta JFlex, es conveniente crear la tabla de codificación de tokens, ya que nos servirá tanto en el analizador léxico como en el sintáctico. Podemos apreciar la codificación en la Tabla 9.

Tabla 9: Codificación de tokens

TOKEN	CODIFICACIÓN
PROGRAMA	T_PROGRAMA
IDENTIFICADOR	T_IDENTIFICADOR
;	;
:	:
CADENA	T_CADENA
ENTERO	T_ENTERO
REAL	T_REAL
CARACTER	T_CARACTER

LEER	T_LEER
ESCRIBIR	T_ESCRIBIR
INICIO	T_INICIO
FIN	T_FIN
SI	T_SI
ENTONCES	T_ENTONCES
SINO	T_SINO
FINSI	T_FINSI
PARA	T_PARA
FINPARA	T_FINPARA
MIENTRAS	T_MIENTRAS
HACER	T_HACER
FINMIENTRAS	T_FINMIENTRAS
-	-
+	+
*	*
/	/
^	^
>	>
<	<
MAYIG	T_MAYIG
MENIG	T_MENIG
IGUAL	T_IGUAL
DIFERENTE	T_DIFERENTE
Y	T_Y
O	T_O
!	!
CTE_CADENA	T_CTE_CADENA
CTE_ENTERO	T_CTE_ENTERO
CTE_REAL	T_CTE_REAL
CTE_CARACTER	T_CTE_CARACTER
CTE_LOGICA	T_CTE_LOGICA
(	(
)	)
=	=
,	,

Esta codificación se encuentra en un archivo que llamaremos Parser.java, que se verá con mayor detenimiento en la iteración del Analizador Sintáctico. Ahora veamos cómo generamos el archivo de entrada para la herramienta JFlex, archivo que llamaremos analizadorlex.flex:

- La estructura de un archivo flex se divide en tres partes separadas por dos símbolos de porcentaje "%%", la primera sección (llamada sección de declaraciones) se utiliza para importar librerías del proyecto, la segunda sección es llamada sección de reglas de usuario, que es donde ponemos todas las reglas vistas anteriormente, y la tercera sección es la sección de rutinas de usuario, en la que especificamos las acciones que se desencadenarán cuando el analizador reconozca un componente léxico.
- Explicado esto, en la primera sección importamos dos paquetes, el paquete de Símbolos (archivos que manejan la tabla de símbolos del intérprete) y el paquete sintáctico (archivos que manejan en analizador sintáctico). Podemos apreciar un poco del código en la Figura 9.

```
package Lexico;
import Simbolos.*;
import Sintactico.*;
```

Figura 9: Sección 1, analizador léxico

- En la segunda sección especificamos los patrones para los componentes léxicos. Podemos apreciar un poco del código en la Figura 10 y en la Figura 11.

```
%class AnalizadorLex
%unicode
%byaccj
%line
%column
%public

%{
    StringBuffer string = new StringBuffer();
    private Parser yyparser;
    private Tabla tabla;
    public AnalizadorLex(java.io.Reader r, Parser yyparser, Tabla tabla){
        this(r);
        this.yyparser = yyparser;
        this.tabla = tabla;
    }
%}
```

Figura 10: Sección 2, analizador léxico, configuración

```
// eol = end of line
eol = \r | \n | \r\n
caracter = [^\r\n]
espacio = {eol} | [ \t\f]

cte_entero = "0" | [1-9][0-9]*
cte_logica = "verdadero" | "falso"

identificador = [:jletter:] ([:jletterdigit:] | '_')*

op_y = "&&"
op_o = "||"
op_rel = "<" | ">" | ">=" | "<=" | "==" | "!="

%state STRING
```

Figura 11: Sección 2, analizador léxico, reglas léxicas

- En la tercera sección especificamos las acciones que se desencadenan cuando se encuentra un componente léxico, en este caso, se retorna el token correspondiente. Además se cargan las constantes e identificadores en la tabla de símbolos. Podemos apreciar un poco del código en la Figura 12, y en la Figura 13.

```
<YYINITIAL> "PROGRAMA"      { return Parser.T_PROGRAMA; }
<YYINITIAL> "INICIO"        { return Parser.T_INICIO; }
<YYINITIAL> "FIN"           { return Parser.T_FIN; }
<YYINITIAL> "SI"            { return Parser.T_SI; }
<YYINITIAL> "ENTONCES"      { return Parser.T_ENTONCES; }
<YYINITIAL> "SINO"          { return Parser.T_SINO; }
<YYINITIAL> "FINSI"         { return Parser.T_FINSI; }
<YYINITIAL> "MIENTRAS"      { return Parser.T_MIENTRAS; }
<YYINITIAL> "HACER"         { return Parser.T_HACER; }
<YYINITIAL> "FINMIENTRAS"   { return Parser.T_FINMIENTRAS; }
<YYINITIAL> "PARA"          { return Parser.T_PARA; }
<YYINITIAL> "HASTA"         { return Parser.T_HASTA; }
<YYINITIAL> "FINPARA"       { return Parser.T_FINPARA; }
<YYINITIAL> "ESCRIBIR"      { return Parser.T_ESCRIBIR; }
<YYINITIAL> "LEER"          { return Parser.T_LEER; }
<YYINITIAL> "cadena"        { return Parser.T_CADENA; }
<YYINITIAL> "entero"        { return Parser.T_ENTERO; }
<YYINITIAL> "logico"        { return Parser.T_LOGICO; }
```

Figura 12: Sección 3, analizador léxico, palabras reservadas

```
<YYINITIAL> {
    {cte_entero}            { yyparser.yylval.info.pos = this.tabla.adicionarConstanteEntera(yytext()); return Parser.T_CTE_ENTERO; }
    {cte_logica}            { yyparser.yylval.info.pos = this.tabla.adicionarConstanteLogica(yytext()); return Parser.T_CTE_LOGICO; }

    {identificador}         { yyparser.yylval.info.pos = this.tabla.adicionarIdentificador(yytext()); return Parser.T_IDENTIFICADOR; }

    {espacio}               { }

    {op_y}                  { return Parser.T_Y; }
    {op_o}                  { return Parser.T_O; }
    {op_rel}                { yyparser.yylval = new ParserVal(yytext()); return Parser.T_OP_REL; }
    ","                     { return Parser.T_COMA; }
    ";"                     { return Parser.T_PTO_COMA; }
    ":"                     { return Parser.T_DOS_PUNTOS; }
    "("                     { return Parser.T_PAR_ABRE; }
    ")"                     { return Parser.T_PAR_CIERRA; }
    "+"                     { return Parser.T_MAS; }
    "-"                     { return Parser.T_MENOS; }
    "*"                     { return Parser.T_POR; }
    "/"                     { return Parser.T_ENTRE; }
    "!"                     { return Parser.T_NO; }
    "="                     { return Parser.T_IGUAL; }
    \"                      { string.setLength(0); yybegin(STRING); }
    {caracter}              { return yytext().charAt(0); }
}

<STRING> {
    \"                      { yybegin(YYINITIAL);
                              yyparser.yylval.info.pos = this.tabla.adicionarConstanteCadena(string.toString()); return Parser.T_CTE_CADENA; }

    [^\n\r\"\\]+            { string.append( yytext() ); }
    \\t                     { string.append( '\t'); }
    \\n                     { string.append( '\n' );}
    \\r                     { string.append( '\r' );}
    \\\"                    { string.append( '\"' );}
    \\                      { string.append( '\\' );}
}
```

Figura 13: Sección 3, analizador léxico, acciones

Una vez hecho esto, se compila el archivo utilizando la herramienta JFlex, lo que genera un archivo escrito en java AnalizadorLex.java, que será luego importado al proyecto. En la Figura 14 podemos ver cómo queda el paquete Lexico.

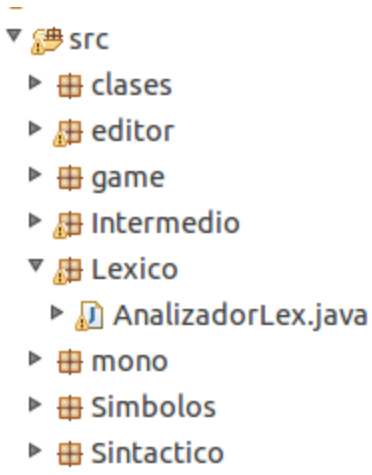

Figura 14: Paquetes del proyecto, analizador léxico

4.3.1.3 Pruebas

Para efectos de pruebas, se crea dentro del paquete Sintactico, un archivo llamado Parser.java en el que se declaran los tokens de cada componente léxico. Se crea también dentro del archivo principal del proyecto una instancia de la clase AnalizadorLex, y enviamos una cadena con lexemas propios de nuestro lenguaje, así como también lexemas “incorrectos” para verificar el correcto funcionamiento del analizador.

4.3.2 Iteración 2: Analizador Sintáctico

4.3.2.1 Análisis y diseño

Una vez que tenemos el analizador léxico, es hora de crear la gramática del lenguaje. Como vimos anteriormente, las reglas del lenguaje son:Que cada variable sea declarada antes de su uso, que cada sentencia termine en punto y coma, y que en una asignación, la expresión sea del mismo tipo que el identificador.

En este caso, para construir la gramática del lenguaje, sólo nos importan las dos primeras reglas. Tomando en cuenta el ejemplo que preparamos en el analizador sintáctico, comenzamos a armar las reglas de la gramática.

Un programa tiene la siguiente estructura: <encabezado><variables><cuerpo>.

Cada una de estas partes está definido según las siguientes reglas de gramática analizadas por el autor de esta investigación:

- <encab> ::= T_PROGRAMA T_IDENTIFICADOR ;
- <variables> ::= <variables><declaracion> | <declaracion>
- <declaracion> ::= <tipo><lista_identificadores> ;
- <tipo> ::= T_CADENA | T_CARACTER | T_ENTERO | T_REAL | T_LOGICA
- <lista_identificadores> ::= <lista_identificadores> , T_IDENTIFICADOR | T_IDENTIFICADOR
- <cuerpo> ::= T_INICIO <lista_sentencias> T_FIN
- <lista_sentencias> ::= <lista_sentencias><sentencia> | <sentencia>
- <sentencia> ::= <sentencia_leer> | <sentencia_escribir>| <sentencia_asignacion>| <sentencia_condicional> | <sentencia_para>| <sentencia_repetitiva>| <sentencia_hacer>| <sentencia_funcion>
- <sentencia_leer> ::= T_LEER : T_IDENTIFICADOR ;
- <sentencia_escribir> ::= T_ESCRIBIR : <expresion> ;
- <sentencia_asignacion> ::= T_IDENTIFICADOR = <expresion> ;
- <sentencia_condicional>::= T_SI <expresion> T_ENTONCES <lista_sentencias><parte_sino> T_FINSI

- <parte_sino> ::= T_SINO <lista_sentencias> | ε
- <sentencia_para> ::= T_PARA T_IDENTIFICADOR = T_CTE_ENTERO T_HASTA T_CTE_ENTERO <lista_sentencias> T_FINPARA
- <sentencia_repetitiva> ::= T_MIENTRAS <expresion> T_HACER <lista_sentencias> T_FINMIENTRAS
- <sentencia_hacer> ::= T_HACER <lista_sentencias> T_MIENTRAS <expresion> ;
- <sentencia_funcion> ::= <funcion> ;
- <funcion> ::= T_IDENTIFICADOR ( )
- <expresion> ::= <expresion> T_O <termino_log> | <termino_log>
- <termino_log> ::= <termino_log> T_Y <factor_log> | <factor_log>
- <factor_log> ::= ! <factor_log> | <expresion_rel>
- <expresion_rel> ::= <exprecion_rel><op_rel><expresion_arit> | <expresion_arit>
- <expresion_arit> ::= <expresion_arit> + <termino_arit> | <expresion_arit> - <termino_arit> | <termino_arit>
- <termino_arit> ::= <termino_arit> * <factor_arit>| <termino_arit> / <factor_arit>| <factor_arit>
- <factor_arit> ::= (<expresion>) | - <factor_arit> | ^ <expresion> | T_IDENTIFICADOR | T_CTE_CADENA | T_CTE_CARACTER | T_CTE_ENTERO | T_CTE_REAL | T_CTE_LOGICA | <funcion>
- <op_rel> ::= >| < | T_MAYIG | T_MENIG |T_IGUAL | T_DIF

4.3.2.2 Desarrollo

Para el desarrollo del analizador sintáctico también hacemos uso de la herramienta Sublime Text como editor de texto, y hacemos uso del generador de analizadores sintácticos CUP.

Nombramos al archivo de entrada de CUP como analizadorsin.cup, este archivo tiene tres secciones divididas por “%%”.

- La primera sección, llamada sección de declaraciones, sirve para importar librerías a utilizar y también para escribir algunos comandos de configuración para CUP. En la segunda sección, llamada sección de reglas, escribimos las

reglas que hemos diseñado, de la forma: <no_terminal> : <terminales> | <no_terminales>;. En la tercera sección, llamada reglas de usuario, se escribe código en java que será copiado con exactitud al archivo generado por CUP.

- Explicado esto, en la primera sección escribimos las declaraciones necesarias para la correcta configuración de CUP para nuestra gramática. Podemos ver un poco del código en la Figura 15.

```
%{
package Sintactico;
import java.io.*;
import Lexico.*;
import Simbolos.*;
import Intermedio.*;
%}

%token <info> T_PROGRAMA, T_IDENTIFICADOR, T_INICIO, T_FIN;
%token <info> T_SI, T_ENTONCES, T_SINO, T_FINSI;
%token <info> T_PARA, T_FINPARA, T_MIENTRAS, T_FINMIENTRAS, T_HACER, T_HASTA;
%token <info> T_CADENA, T_ENTERO, T_LOGICO, T_LEER, T_ESCRIBIR;
%token <info> T_PTO_COMA, T_DOS_PUNTOS;
%token <info> T_OP_REL, T_Y, T_O, T_NO, T_PAR_ABRE, T_PAR_CIERRA, T_IGUAL, T_COMA;
%token <info> T_CTE_CADENA;
%token <info> T_CTE_LOGICO;
%token <info> T_CTE_ENTERO;

%left T_MENOS, T_MAS;
%left T_POR, T_ENTRE;
%left T_NEGACION; /* menos unario */

%type <info> programa, encab, variables, cuerpo;
%type <info> declaracion, tipo, lista_identificadores;
%type <info> lista_sentencias, sentencia, identificador;
%type <info> sentencia_leer, sentencia_escribir, sentencia_asignacion, sentencia_condicional,
             entonces, parte_sino, sino, sentencia_para, sentencia_mientras, hacer, sentencia_hacer, sentencia_funcion;
%type <info> expresion, funcion, termino_log, factor_log, expresion_rel, expresion_arit, termino_arit;
%type <info> factor_arit;
%type <info> pto_coma, para_asignacion, para_expresion, para_incremento, para_ssf, constante_entero;

%start programa;
```

Figura 15: Sección 1, analizador sintáctico

- En la segunda sección escribimos nuestras reglas, hay que recalcar que se debió hacer algunas modificaciones por problemas técnicos al momento de guardar información de errores y demás información necesaria. Podemos ver un poco del código en la Figura 16.

```
programa : encab variables cuerpo {}
    ;

encab : T_PROGRAMA T_IDENTIFICADOR T_PTO_COMA {}
    ;

variables : variables declaracion | declaracion
    ;
```

Figura 16: Una parte de la sección 2, analizador léxico

- En la tercera sección escribimos el código necesario para el tratamiento de errores y función para cargar las palabras reservadas (las funciones del personaje y del escenario). Podemos ver un poco del código en la Figura 17.

```
private AnalizadorLex lexer;
private Tabla tabla;
private int tipo;
private Codigo codigoIntermedio;
private int ip;

private int yylex() { ...
}

private void yyerror(String error){
    System.err.println("Error: " + error);
}

private void errorSemantico(String error, int pos){
    System.out.println(error + ": " + tabla.getSimbolo(pos).getNombre());
}

public Parser(Reader r){ ...
}

private void cargarPalabrasReservadas(){ ...
}

public Codigo getCodigoIntermedio(){
    return this.codigoIntermedio;
}
```

Figura 17: Sección 3, analizador sintáctico

Una vez que hemos terminado de editar nuestro archivo CUP, pasamos a compilarlo. La herramienta CUP generará dos archivos java. AnalizadorSin.java y Parser.java. El archivo Parser.java contiene las especificaciones de los tokens que configuramos en el analizador léxico. Ambos archivos debemos copiarlos al paquete Sintactico de nuestro proyecto como apreciamos en la Figura 18.

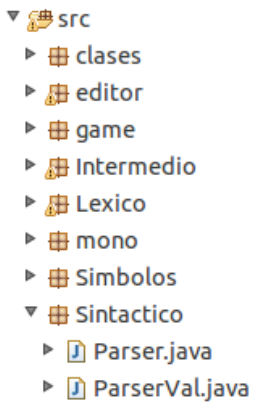

Figura 18: Paquetes del proyecto, analizador sintáctico

4.3.2.3 Pruebas

Para probar que el analizador sintáctico funciona correctamente, abrimos nuestro proyecto en eclipse y en el archivo principal creamos una instancia del analizador

léxico, y una instancia del analizador sintáctico, pasando como parámetro en el constructor al objeto de analizadorLex.

Creamos una cadena con un ejemplo de código, incluimos también "errores" en el código, y ejecutamos la función yyparse() del analizador sintáctico.

4.3.3 Iteración 3: Analizador Semántico

4.3.3.1 Análisis y diseño

En esta etapa, nos encargamos de verificar principalmente los tipos de datos, por ejemplo en la sentencia de asignación, ambas partes deben ser del mismo tipo, así como también especificamos las reglas para cuando encontramos operadores con diferentes tipos de datos, como cuando sumamos un entero y un real. También verificamos lo que establecimos en las reglas de la gramática con respecto a las variables: Toda variable debe ser previamente declarada.

Especificamos entonces las siguientes reglas semánticas:

- Al declarar la lista de identificadores, si el identificador ya se encuentra en la tabla de símbolos: llamamos a la función errorSemantico, indicando que la variable ya ha sido declarada.
- En la sentencia LEER, si el identificador no se encuentra en la tabla de símbolos: llamamos a la función errorSemantico, indicando que la variable no ha sido previamente declarada.
- En la sentencia asignación, si el identificador no se encuentra en la tabla de símbolos: llamamos a la función errorSemantico, indicando que la variable no ha sido previamente declarada.
- En la sentencia PARA, si el identificador no se encuentra en la tabla de símbolos: llamamos a la función errorSemantico, indicando que la variable no ha sido previamente declarada.

4.3.3.2 Desarrollo

La implementación de esta iteración la realizamos en el mismo archivo del analizador sintáctico analizadorsin.cup, debido a que estas comprobaciones se deben ejecutar al momento en que es reconocida una regla de la gramática.

Por lo tanto, en la tercera sección del archivo CUP, creamos una función llamada errorSemantico, que se encargará de mostrar el error y la variable a la que hace referencia el error.

Luego hacemos las validaciones correspondientes en cada una de las reglas de la gramática especificadas líneas arriba. Además, se asignan los tipos a los elementos de la tabla de símbolos. Podemos ver un ejemplo en la Figura 19.

```
if(tabla.getSimbolo($1.pos).getTipo() == Simbolo.TIPO_NINGUNO)
    errorSemantico("Variable sin declarar", $1.pos);
```

Figura 19: Ejemplo de verificación de declaración de variables

Una vez terminado, debemos volver a compilar el archivo y copiarlo al proyecto principal en eclipse.

4.3.3.3 Pruebas

Para probar que el analizador semántico funciona correctamente, modificamos la cadena de prueba establecida para probar el analizador sintáctico, esta vez probamos los tipos de datos dentro del código, por tanto debemos establecer casos de prueba como múltiple declaración de una variable, uso de variables sin declarar, asignación con diferentes tipos de datos.

4.3.4 Iteración 4: Generador de código

4.3.4.1 Análisis y diseño

Luego de pasar por las fases de análisis, el siguiente paso para un compilador/intérprete es generar código intermedio, este código no llega a ser el lenguaje de máquina ni tampoco es el lenguaje fuente en el que está escrito el programa. La característica principal de este código intermedio es que sus instrucciones son muy simples, generalmente realizan una o dos operaciones como máximo. En nuestro caso, generaremos un lenguaje intermedio orientado a pilas.

Cada operación del lenguaje debe tener un operador en el lenguaje intermedio que indique en la ejecución lo que se debe realizar, así identificamos los operadores del lenguaje intermedio:

- Asignación: asigna un valor a un identificador en la tabla de símbolos.
- Valor: obtiene un valor de la tabla de símbolos.
- Suma: Suma dos valores de la pila de ejecución.

- Resta: Resta dos valores de la pila de ejecución.
- Multiplicación: Multiplica dos valores de la pila de ejecución.
- División: Divide dos valores de la pila de ejecución.
- Negativo: Multiplica el elemento de la pila por -1.
- Menor que: Compara si un elemento de la pila es menor que el siguiente elemento de la pila.
- Mayor que: Compara si un elemento de la pila es mayor que el siguiente elemento de la pila.
- Igual: Compara si un elemento de la pila es igual al siguiente elemento de la pila.
- Menor o igual que: Compara si un elemento de la pila es menor o igual que el siguiente elemento de la pila.
- Mayor o igual que: Compara si un elemento de la pila es mayor o igual que el siguiente elemento de la pila.
- Diferente: Compara si un elemento de la pila es diferente al siguiente elemento de la pila.
- O: Devuelve verdadero si por lo menos uno de dos valores es verdadero. De lo contrario devuelve falso.
- Y: Devuelve verdadero si ambos elementos de la pila son verdaderos, de lo contrario devuelve falso.
- No: Convierte un valor verdadero en falso, y un valor falso en verdadero.
- Leer: Obtiene un valor del flujo de entrada.
- Imprimir valor: Muestra un valor en el flujo de salida.
- Saltar si falso: Salta la ejecución hasta la línea indicada si el elemento en la pila es falso.
- Si: Continúa la ejecución si el elemento en la pila es verdadero.
- Función: Ejecuta una función.

Una vez hecho el análisis y luego de haber definido la lista de operaciones pasamos a codificarlas, y asignarles un valor, que será usado posteriormente para generar el código intermedio y en la ejecución:

- OP_ASIG = 5101.

- OP_VALOR = 5102.
- OP_SUM = 5103.
- OP_RES = 5104.
- OP_MUL = 5105.
- OP_DIV = 5106.
- OP_NEG = 5107.
- OP_MEN = 5108.
- OP_MAY = 5109.
- OP_IG = 5110.
- OP_MENIG = 5111.
- OP_MAYIG = 5112.
- OP_DIF = 5113.
- OP_OR = 5114.
- OP_AND = 5115.
- OP_NOT = 5116.
- OP_LEER = 5117.
- OP_IMPRVALOR = 5118.
- OP_SSF = 5119.
- OP_SI = 5120.
- OP_FUNC = 5121.

4.3.4.2 Desarrollo

La generación de código se especifica dentro del mismo archivo del analizador sintáctico y semántico.

Primero creamos el archivo Operaciones, dentro del paquete Intermedio, en el que declaramos constantes para los operadores definidos anteriormente.

Ahora, en el archivo analizadorsin.CUP, especificamos las acciones que hay que tomar una vez que reconocemos cada regla de la gramática. Hay que tener en cuenta que

nuestro pseudo lenguaje es de un nivel más alto que nuestro lenguaje intermedio, por lo que una instrucción puede convertirse en dos, tres, cuatro, o más operaciones en el lenguaje intermedio.

Por ejemplo: en una instrucción A = B + C; el código intermedio generado sería:

1. OP_VALOR
2. POS B
3. OP_VALOR
4. POS C
5. OP_SUM
6. POS A
7. OP_ASIG

Como podemos observar, una sola línea de código se convirtió en siete instrucciones del lenguaje intermedio.

Por lo general, es relativamente fácil de generar código intermedio similar al ejemplo en las reglas de la gramática, pero hay que tener especial cuidado en las estructuras de control de flujo como las sentencias condicionales y las estructuras repetitivas. Finalmente, nuestro paquete de la tabla de símbolos podemos apreciarlo en la Figura 21 y el paquete del código intermedio en la Figura 20.

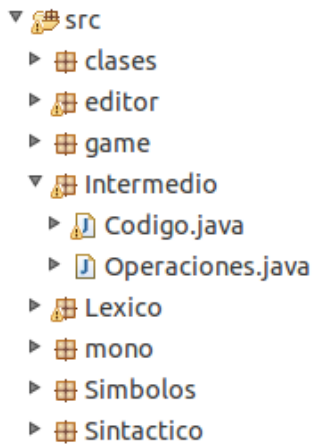

Figura 20: Paquetes del proyecto, código intermedio

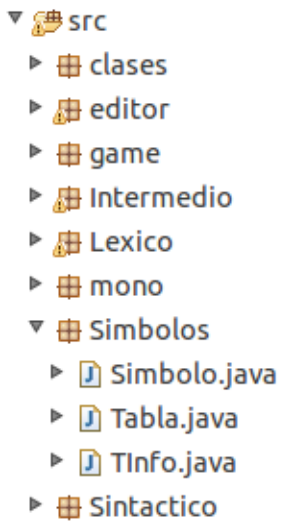

Figura 21: Paquetes del proyecto, tabla de símbolos

4.3.4.3 Pruebas

Para probar el código intermedio generado, una vez compilado el archivo y copiado al proyecto, hacemos la misma prueba que en el analizador sintáctico y semántico, pero esta vez, imprimimos el arreglo generado por el generador de código.

4.3.5 Iteración 5: Ejecutor

El desarrollo del ejecutor es relativamente sencillo, lo único que hará es recorrer el arreglo del código intermedio, si es una posición en memoria, la deja pasar, de lo contrario, si es una operación definida en la clase OPERADORES, ejecuta las acciones respectivas.

Esto ha sido programado en la clase codigo.java, dentro del paquete Intermedio. Veamos la estructura del archivo en laFigura 22.

```java
public class Codigo {
    public ArrayList<Integer> codigo;
    private BufferedReader objEnt;
    private ArrayList<ParserVal> ejecutor;
    private Principal juego;
    private Tabla tabla;
    private int index, top;
    private Escenario esc;

    public Codigo(Tabla tabla){
        this.codigo = new ArrayList<>();
        this.objEnt = new BufferedReader(new InputStreamReader(System.in));
        this.ejecutor = new ArrayList<>();
        this.tabla = tabla;
        this.index = 0;
        this.top = -1;
    }

    public void setJuego(Principal juego){
        this.juego = juego;
    }

    public Principal getJuego(){
        return this.juego;
    }

    public void setEscenario(){
        this.esc = (Escenario)juego.getActualEscenario();
    }

    public void ejecutarSiguiente(){ ...
    }

    public void ejecutar(){
        while(index < codigo.size()){
            ejecutarSiguiente();
        }
    }
}
```

Figura 22: Estructura del archivo codigo.java

4.3.6 Iteración 6: Definición de Clases y Elementos

4.3.6.1 Análisis y diseño

Al analizar la librería JSlick2D, se pudo comprobar que ofrece facilidad para el trabajo con gráficos, y animaciones, pero para trabajar mejor, se decidió crear clases para los objetos que interactúan en el escenario, así por ejemplo tenemos:

- Caja
- Escalera
- Moneda
- Palanca
- Personaje
- Piso
- Puerta

Ahora, una vez definidos, tendremos que crear un gestor de colisiones que evalúe constantemente cuando dos o más elementos "se choquen" en el escenario, y ejecutar las acciones respectivas. Por ello creamos las siguientes clases:

- ControladorCaja
- ControladorEscalera
- ControladorMoneda
- ControladorPiso
- GestorColision

De esta manera, cada vez que se cree un objeto de alguno de los elementos antes mencionados, deben también ser agregados en sus respectivos controladores, que tendrán una lista de los elementos, y que, gracias al gestor de colisión, evaluarán cuando los elementos colisionen y enviarán la señal a cada elemento para que se ejecuten las acciones necesarias.

4.3.6.2 Desarrollo

Para implementar el trabajo hecho en el punto de diseño, debemos primero crear algunas clases de ayuda:

- Animacion
- AnimacionMovil
- ElementoEscenario
- IColisionable
- IControlador
- Punto
- Sprite
- Vector

Estas clases contienen código de ayuda, por ejemplo una animación contiene una secuencia de imágenes que cambiarán dando el efecto de una animación, esta posibilidad nos brinda la librería JSlick, así que simplemente extendemos de una clase Animation, y configuramos para que se adecúe más a nuestro proyecto. Una animación es por ejemplo: la rueda de un coche girando. Una animación móvil, además de pasar en

secuencia la lista de imágenes, también desplaza la imagen por el escenario (modificando las coordenadas). Por ejemplo: el personaje al caminar.

Así, cada una de estas clases contiene código que nos facilitará el movimiento, la animación y la interacción del personaje con los elementos del escenario.

Al final del desarrollo de esta interacción, nuestro paquete de clases queda como se muestra en la Figura 23.

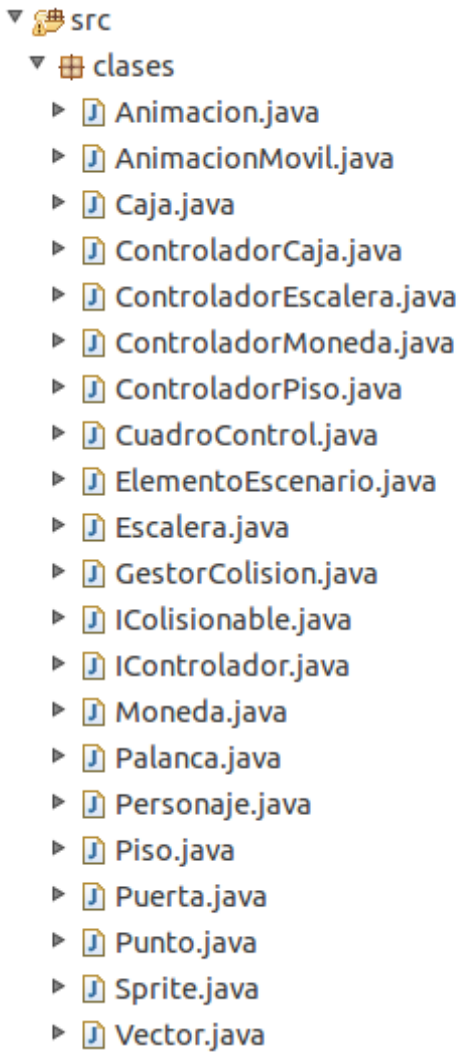

Figura 23: Proyecto principal, paquete clases

4.3.7 Iteración 7: Representación Gráfica y Animación

4.3.7.1 Análisis y diseño

Para poder graficar y utilizar las clases para representar los gráficos y la animación, necesitamos conocer cómo funciona la librería JSLick 2D.

En primer lugar, hay que saber que esta librería ejecuta las animaciones de manera asíncrona, mediante hilos de ejecución. En esta aplicación se crean escenarios, y estos son añadidos a un contenedor, de tal manera que "lanzamos" el contenedor y dentro de éste podemos cambiar entre escenarios. Un escenario tiene funciones de renderizado y actualización, como un JPane. En algunos tutoriales que podemos encontrar en la web, observamos que para cada escenario los programadores crean una clase, ya que todos los elementos y toda la lógica está programada dentro de estas clases, y cada escenario se comporta de manera diferente.

En esta etapa, nos concentraremos en crear tan sólo un escenario, comprenderemos bien el funcionamiento de la librería y luego veremos si se puede optimizar este proceso.

4.3.7.2 Desarrollo

Para esta iteración hemos creado el paquete "game" en donde guardamos las clases propias del juego. Tan sólo hemos definido tres clases:

- Escenario.
- Fin.
- Principal.

La clase principal, nos sirve para cargar toda la configuración que necesita ingresarse en la librería para que pueda cargar las gráficas. En la clase escenario hemos programado la lógica de los elementos, el renderizado y actualización. Todavía no unimos el juego a la parte del compilador, por lo que, para poder probar que funciona correctamente hemos creado un método "updateTeclado" que mueve al personaje presionando algunas teclas.

La clase Fin se crea principalmente para solucionar un problema técnico. Al programar el escenario y el contenedor, surgió un problema, resulta que al lanzar una vez la aplicación, ésta carga correctamente, pero al lanzarla por segunda vez (tercera, cuarta, etc.) ya no cargan las gráficas y sólo se muestra una pantalla en blanco. Investigando

cómo funciona la librería, nos dimos con la sorpresa de que al cargar el contenedor, utiliza algunos componentes hardware y algunos puertos de sonido y video, y al cerrar la ventana a través del botón cerrar (el que aparece en cualquier ventana) éste no libera los recursos, por lo que al cargar nuevamente el escenario, los recursos se encuentran ocupados. Por ello se creó la clase Fin, lo que hace es liberar los recursos antes de cerrar el contenedor, para ello, estando en el escenario, se presiona la tecla Esc.

Una vez desarrollada esta iteración, nuestro paquete queda como se muestra en laFigura 24.

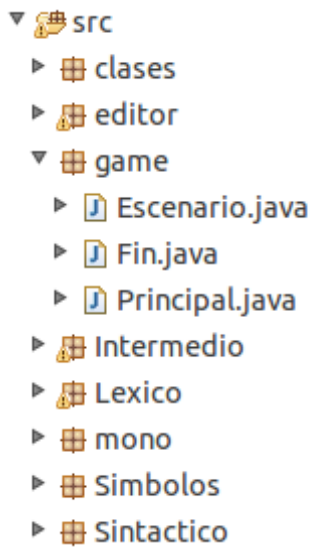

Figura 24: Proyecto principal, paquete game

4.3.7.3 Pruebas

Como se especificó líneas arriba, en esta iteración aún no hemos unido el compilador a la parte del juego, por lo tanto, se hizo pruebas a la animación utilizando el teclado.

4.3.8 Iteración 8: Constructor de Escenarios

4.3.8.1 Análisis y diseño

Como especificamos en la fase anterior, en los tutoriales y guías sobre la librería JSlick2D, que pudimos encontrar en la web, muchos programadores crean una clase para cada escenario del juego. Sin embargo, se pudo analizar con detenimiento el funcionamiento de esta clase que hereda de la clase State. Se consideró que por lo general los programadores creaban una clase por cada escenario porque cada escenario contenía una lógica distinta, sin embargo en nuestro proyecto, todos los escenarios tienen la misma lógica, los mismos elementos. Por lo tanto, se debería crear una única

clase general llamada Escenario. Se trabajó esto modificando la clase que ya tenemos en el paquete “game”.

4.3.8.2 Desarrollo

La clase general “Escenario” lee un archivo de entrada en donde se especifica las coordenadas de los sprites y las coordenadas donde deben ser dibujados. De esta manera, nuestra clase escenario sirve como un constructor de escenarios, ya que cada escenario se crea de manera automática gracias a un archivo de especificación. En este archivo sólo debemos especificar coordenadas.

Podemos apreciar en la Figura 26un ejemplo de un archivo de especificación de escenario. Y en la Figura 25podemos apreciar la estructura principal de la clase Escenario.

```
43    private GestorColision gestorColision;
44
45+   public void setEscenario(File escenario){
48
49+   public void init(GameContainer container, StateBasedGame game)
54
56+   public void enter(GameContainer container, StateBasedGame game) throws SlickException{
73
74+   private void cargarElementos(){
127
128+  private void addPuerta(int x, int y, int estado) throws SlickException{
146
147+  private void addPalanca(int x, int y) throws SlickException{
151
152+  private void addPersonaje(int x, int y) throws SlickException{
159
160+  public void render(GameContainer container, StateBasedGame game, Graphics g)
194
195+  public void update(GameContainer container, StateBasedGame game, int delta)
206
207+  private void updateTeclado(GameContainer container, StateBasedGame game, Graphics g){
216
217+  public void dibujarFondo(){
226
227+  public void dibujarRejilla(Graphics g){
236
237+  public void añadirCajas() throws SlickException{
245
246+  public void añadirMonedas() throws SlickException {
251
252+  public int getID() {
256
257+  public void setConfig(int puntaje, int numVidas, Codigo codigo) {
```

Figura 25: Estructura archivo Escenario.java

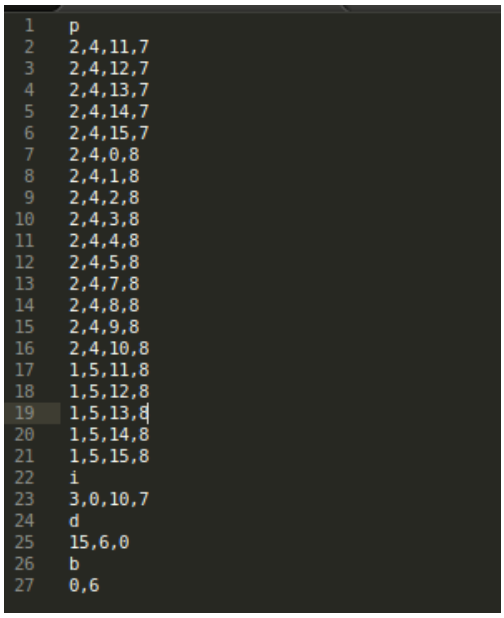

Figura 26: Especificación de un escenario

4.3.8.3 Pruebas

Para probar si nuestro constructor funciona correctamente, sólo creamos más archivos de escenario, y modificamos la clase Escenario para leer cada archivo.

4.3.9 Iteración 9; Construcción del Editor

4.3.9.1 Análisis y diseño

Como pudimos apreciar en el prototipo del editor, Figura 3 (página 41), nuestro editor se divide principalmente en cuatro secciones:

- Menú.
- Pre visualización.
- Sección de ayuda.
- Editor.

En el menú incluimos botones como “Ejecutar”, “Guardar archivo”, “Abrir archivo”, etc. En la pre visualización, el editor grafica el escenario, sin animaciones, como una única imagen del escenario. En la sección de ayuda se cargan algunas funciones como ayuda para el usuario, son funciones que puede usar en el código en dicho escenario específicamente. Y el editor es donde el usuario tipea todo el código.

4.3.9.2 Desarrollo

Para el desarrollo se utilizó las clases de la interfaz de usuario SWIMG, que podemos encontrar dentro de la API de Java. Con algunos botones en la sección menú, y un área de texto para la edición.

Aquí programamos las funciones para guardar archivos, abrir, y ejecutar. Sin embargo, la función ejecutar es parte de la siguiente iteración. En laFigura 27 podemos apreciar cómo quedó nuestro editor una vez lanzado.

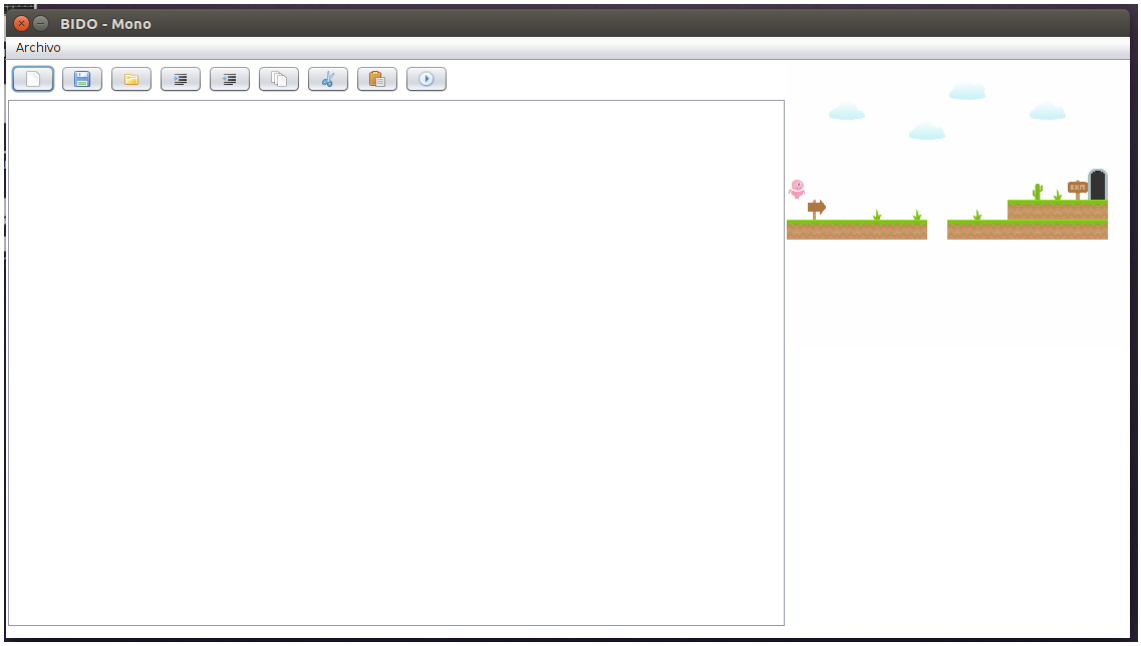

Figura 27: Editor

4.3.9.3 Pruebas

Para realizar las pruebas en esta iteración, cargamos diferentes escenarios, revisamos en la ventana de pre visualización, guardamos y abrimos archivos, y por último, probamos la función ejecutar, que por el momento, abre el contenedor y carga el escenario, pero aún no ejecuta el código programado en el editor.

4.3.10 Iteración 10; Construcción del Ejecutor

4.3.10.1 Análisis y diseño

Para unir las fases de compilación (el intérprete) y de animación (el juego), debemos crear dentro de la clase Codigo.java, del paquete Intermedio, los métodos ejecutar y ejecutarSiguiente.

La idea es que primero se ejecute el compilador, y una vez generado el código intermedio, comience a ejecutar las órdenes. Sin embargo, como se mencionó en iteraciones anteriores, la librería JSlick2D trabaja el contenedor de escenarios de manera asíncrona, esto dificulta el trabajo, porque cuando se está ejecutando el contenedor de escenarios, no puede recibir órdenes del ejecutor.

Como solución a este problema, se creó un objeto ejecutor dentro del contenedor, de la misma manera un objeto contenedor dentro del ejecutor. Con esto logramos lo siguiente: Cuando se ejecuta el contenedor, inhabilitamos el renderizado y actualización de los gráficos, y llamamos al método ejecutarSiguiente(), esto ejecuta la siguiente instrucción en el código intermedio, como sabemos el código intermedio contiene instrucciones de asignación, sumas, restas, etc. Y además contiene funciones propias del juego, por lo tanto, cuando toque el turno de ejecutar las funciones propias del juego, desde el ejecutor volvemos a habilitar el renderizado y actualización de gráficos del contenedor. Así logramos que el ejecutor de código y el juego simulen una sincronización.

4.3.10.2 Desarrollo

Como ya se explicó, se tuvo que hacer una simulación de sincronización, podemos ver un poco del código en la Figura 28 y en la Figura 29.

```
public void update(GameContainer container, StateBasedGame game, int delta)
        throws SlickException {
    if(!over){
        this.personaje.update(delta, this.entrada);
        //this.monedas.update(delta, this.gestorColision);
        this.gestorColision.comprobarColisiones();
        this.ejecutarSiguiente();
    }else{
    }

}
```

Figura 28: Método update en el contenedor

```
public void setCodigo(Codigo codigo){
    this.codigo = codigo;
}

public void ejecutarSiguiente(){
    if(this.codigo != null)
        this.codigo.ejecutarSiguiente();
}
```

Figura 29: Método ejecutarSiguiente en el contenedor

También tuvimos que agregar código en el ejecutor del intérprete, haciendo referencia a las acciones del personaje en el contenedor, como podemos apreciar en la Figura 30.

```
case Operaciones.OP_FUNC:
    Simbolo s = this.tabla.getSimbolo(this.codigo.get(this.index + 1));
    switch(s.getNombre()){
        //Funciones del personaje
        case "saltar":
            this.esc.getPersonaje().saltar();
            break;
        case "caminar":
            this.esc.getPersonaje().caminar();
            break;
        case "saltar_obstaculo":
            this.esc.getPersonaje().saltar_obstaculo();
            break;
        case "girar":
            this.esc.getPersonaje().girar();
            break;
        case "accionar":
            if(this.esc.getPersonaje().isPuedeAccionar()){
                this.esc.getPalanca().accionar();
            }
            break;
        case "subir":
            this.esc.getPersonaje().subir();
            break;
        case "bajar":
            this.esc.getPersonaje().bajar();
            break;
```

Figura 30: Funciones propias del escenario y del personaje

4.3.10.3 Pruebas

En este momento, podemos probar en su totalidad la unión de todo el proyecto, cargando un escenario y programando el algoritmo de solución.

4.4 Transición

En esta etapa, hacemos pruebas exhaustivas del producto. Para comenzar, se crean los archivos de escenarios para diez niveles, tomando en cuenta la estructura del sílabo del curso Fundamentos de Programación. Los elementos que se crean dentro de cada escenario obligan a que se utilicen las estructuras necesarias en cada nivel, como las estructuras condicionales y repetitivas.

V. DISCUSIÓN

Para la contrastación de nuestra hipótesis se ha realizado una evaluación en el curso Fundamentos de programación durante aproximadamente un mes y medio sin implementar el sistema multimedia, periodo en el cual se ha registrado algunos criterios mostrados en el Anexo 1, para su correspondiente análisis en comparación con los resultados obtenidos luego de haber implementado la herramienta tecnológica durante el siguiente mes y medio.

5.1 Indicador 1: Número de estudiantes aprobados en evaluación de estructuras condicionales.

Durante el primer mes y medio se evaluó a los estudiantes sobre estructuras condicionales, Tenemos como resultado que el 27% de los alumnos aprobaron la evaluación (8 estudiantes).

En el mes y medio siguiente, luego de usar la herramienta tecnológica para practicar, se evaluó a los estudiantes con una prueba similar, obteniendo el 67% de aprobados (20 estudiantes).

Como podemos observar, se tuvo un incremento del 40% del total de estudiantes (12 alumnos),

5.2 Indicador 2: Número de estudiantes aprobados en evaluación de estructuras repetitivas.

Durante el primer mes y medio se evaluó a los estudiantes sobre estructuras repetitivas, Tenemos como resultado que el 17% de los alumnos aprobaron la evaluación (5 estudiantes).

En el mes y medio siguiente, luego de usar la herramienta tecnológica para practicar, se evaluó a los estudiantes con una prueba similar, obteniendo el 47% de aprobados (14 estudiantes).

Como podemos observar, se tuvo un incremento del 30% del total de estudiantes (9 alumnos),

5.3 Indicador 3: Tiempo promedio de resolución de algoritmos.

Para la evaluación de este indicador se ha tomado los datos de la tabla en Anexo 1, para luego realizar el cálculo de las fórmulas y comparar si la hipótesis es aceptada o rechazada.

- **M1:** Promedio de duración de resolución de algoritmos sin la implementación de la herramienta tecnológica.
- **M2:** Promedio de duración de resolución de algoritmos con la implementación de la herramienta tecnológica.

Hipótesis H0: M1 < M2

Hipótesis H1: M2 < M1

Para $\alpha = 0.05$ (nivel de significancia)

Valor crítico de "t" de Student: 1.6973

Valor experimental: 3.3505

Siento 3.3505 > 1.6973, por lo tanto se rechaza la hipótesis H0.

Podemos entonces decir que existe evidencia estadística que rechaza H0 y se acepta H1, demostrando que se ha disminuido el promedio de duración de resolución de algoritmos en los estudiantes del curso Fundamentos de Programación con la implementación de la herramienta tecnológica.

5.4 Indicador 4: Porcentaje de satisfacción con el método de enseñanza-aprendizaje.

Para la evaluación de este indicador se ha tomado los datos de la tabla en Anexo 3, para luego realizar el cálculo de las fórmulas y comparar si la hipótesis es aceptada o rechazada.

- **M1:** Promedio de satisfacción con el método de enseñanza - aprendizaje sin la implementación de la herramienta tecnológica.
- **M2:** Promedio de satisfacción con el método de enseñanza - aprendizaje con la implementación de la herramienta tecnológica.

Hipótesis H0: M1 > M2

Hipótesis H1: M2 > M1

Para $\alpha = 0.05$ (nivel de significancia)

Valor crítico de "t" de Student: 1.6973

Valor experimental: 2.6816

Siento 2.6816 > 1.6973, por lo tanto se rechaza la hipótesis H0.

Podemos entonces decir que existe evidencia estadística que rechaza H0 y se acepta H1, demostrando que se ha incrementado el promedio de satisfacción de los estudiantes con el método de enseñanza – aprendizaje en el curso Fundamentos de programación con la implementación de la herramienta tecnológica.

VI. CONCLUSIONES

- Se incrementó el número de estudiantes aprobados en evaluación de estructuras condicionales en un 40%, logrando estudiantes mejor preparados en este tema e incrementando la lógica de programación de los mismos.
- Se incrementó el número de estudiantes aprobados en evaluación de estructuras repetitivas en un 30%, logrando estudiantes que interpretan y entienden mejor las estructuras de control del flujo.
- Se disminuyó el tiempo promedio de resolución de algoritmos en un 60%, logrando que los estudiantes sean más eficientes al programar.
- Se incrementó el porcentaje de satisfacción con el método de enseñanza – aprendizaje en un 19%, demostrando que existen factores externos a las aulas de clase que impiden que la tasa de aprobados sea aún mayor.

Recomendaciones

Se recomienda utilizar la herramienta tecnológica a partir de las primeras semanas de clase, para que los estudiantes puedan practicar desde el inicio y para que las prácticas cubran todos los temas abordados en clase.

Para futuras investigaciones queda por mejorar la gestión de errores, para brindar información más detallada al estudiante, y que pueda comprender aún mejor cómo solucionar un error.

VII.REFERENCIAS BIBLIOGRÁFICAS

Aho Raviseth, Alfred, y Ullman Jefrey D. *Compiladores: Principios, técnicas y herramientas.* New Jersey: Pearson Education, 1998.

Arjona Fernández, María Luisa. *Importancia y elementos de la programación dedáctica.* España: AFOE, 2010.

Bravo V, Luis. *Psicología educacional, psicopedagogía y educación especial.* Santiago de Chile: Universidad Nacional Católica de Chile, 2009.

Dunn, Dunn et. *Learning Style Inventory.* Lawrens, 1985.

Henson, Kennet, y Ben Eller. *Psicología educativa para la enseñanza eficaz.* México: Cengage Learning Editores, 2000.

Hernández, Roberto. *Estructuras de datos y algoritmos.* Madrid: Pearson Education, 2001.

Joyanes Aguilar, Luis. *Fundamentos de programación.* Madrid: McGraw-Hill, 2003.

Keefe, J. *Aprendiendo Perfiles de Aprendizaje: manual del examinador.* España: NASSP, 1988.

Navarro Jiménez, Manuel Jesús. *Cómo diagnosticar y mejorar los estilos de aprendizaje.* Asociación Procompal, 2008.

Revilla, Diana. *Estilos de aprendizaje.* Lima: Pontificia Universidad Católica del Perú, 1998.

Rodríguez Sala, Jesús Javier. *Introducción a la programación, teoría y práctica.* Alicante: Club Universitario, 2003.

Soloway, E. «Learning to program = Learning to construct mechanisms and explanations.» *Comunications of the ACM*, 1986.

Sommerville, Ian. *Ingeniería del Software.* Madrid: Pearson Education, 2005.

VIII. ANEXOS

8.1 Tiempo promedio de resolución de algoritmos por estudiante.

	Antes	Después
Estudiante 1	31	17
Estudiante 2	32	18
Estudiante 3	31	17
Estudiante 4	30	16
Estudiante 5	29	15
Estudiante 6	34	20
Estudiante 7	38	24
Estudiante 8	40	26
Estudiante 9	36	22
Estudiante 10	35	21
Estudiante 11	35	21
Estudiante 12	37	23
Estudiante 13	36	22
Estudiante 14	38	24
Estudiante 15	41	27
Estudiante 16	32	18
Estudiante 17	33	19
Estudiante 18	39	25
Estudiante 19	39	25
Estudiante 20	42	28
Estudiante 21	29	15
Estudiante 22	31	17
Estudiante 23	37	23
Estudiante 24	35	21
Estudiante 25	39	25
Estudiante 26	38	24
Estudiante 27	40	26
Estudiante 28	33	19
Estudiante 29	36	22
Estudiante 30	31	17
Promedio	**35.2333333**	**21.2333333**

Anexo 1: Tiempo de resolución de algoritmos

8.2 Encuesta sobre la satisfacción con respecto al método de enseñanza – aprendizaje.

Encuesta sobre la satisfacción del alumno respecto al proceso de enseñanza – aprendizaje en el curso Fundamentos de programación.

Marca con un aspa (X) el puntaje de cada pregunta, siendo cero la calificación más baja, y cinco la calificación más alta.

Pregunta	**0**	**1**	**2**	**3**	**4**	**5**
¿Cómo calificarías e l método de enseñanza del profesor?						
¿Consideras que el profesor te brinda la ayuda necesaria para desarrollar los ejercicios planteados?						
¿Crees que el profesor te brinda los recursos necesarios en cada tema desarrollado?						
¿Consideras suficiente el tiempo que dedica el profesor para el asesoramiento a los alumnos?						

Anexo 2: Encuesta sobre la satisfacción delos alumnos con el curso Fundamentos de programación

8.3 Resultados de la encuesta sobre la satisfacción con respecto al método de enseñanza - aprendizaje

	Antes	**Después**
Estudiante 1	15	18
Estudiante 2	18	19
Estudiante 3	13	20
Estudiante 4	17	18
Estudiante 5	12	19
Estudiante 6	18	19
Estudiante 7	14	20
Estudiante 8	14	17
Estudiante 9	16	17
Estudiante 10	10	15
Estudiante 11	17	19
Estudiante 12	14	19
Estudiante 13	12	19
Estudiante 14	16	14
Estudiante 15	16	19
Estudiante 16	11	19
Estudiante 17	15	18
Estudiante 18	14	17
Estudiante 19	14	17
Estudiante 20	13	15
Estudiante 21	12	16
Estudiante 22	17	18
Estudiante 23	12	18
Estudiante 24	11	19
Estudiante 25	18	14
Estudiante 26	10	15
Estudiante 27	11	17
Estudiante 28	12	18
Estudiante 29	14	19
Estudiante 30	14	19
Promedio	**14**	**17.7**
Porcentaje	**70%**	**89%**

Anexo 3: Resultados de la encuesta sobre la satisfacción de los alumnos en el curso Fundamentos de programación

8.4 Presupuesto para la elaboración del proyecto.

Para el siguiente presupuesto, se tuvo en cuenta la devaluación de los equipos bajo el método lineal, con los siguientes datos:

1. Laptop: Valor inicial: 2500; valor final: 500; vida útil 5 de años (60 meses), tiempo utilizado comúnmente para calcular la devaluación de componentes electrónicos.
2. Impresora: Valor inicial: 700, valor final: 200, vida útil de 5 años (60 meses), tiempo utilizado comúnmente para calcular la devaluación de componentes electrónicos.

Materiales			
Detalles	**Cantidad**	**Valor (S/.)**	**Total (S/.)**
Millar de Papel A4	3	30	90
Tinta Impresora	4	40	160
Memoria USB	1	20	20
Útiles de oficina	1	50	50
Laptop (devaluación por meses de uso)	3	33.3	99.9
Impresora (devaluación por meses de uso)	3	8.33	24.99
		Sub Total	444.89

Servicios			
Detalles	**Cantidad**	**Valor (S/.)**	**Total (S/.)**
Meses de trabajo	3	1000	3000
Energía eléctrica	3	10	30
Servicio de internet	3	5	15
		Sub Total	3045

Otros Varios			
Detalles	**Cantidad**	**Valor (S/.)**	**Total (S/.)**
Transporte	300	1.5	450
Refrigerios	100	3.5	350
Imprevistos	1	50	50
		Sub Total	850
		Total	4339.89

Anexo 4: Presupuesto para la elaboración del proyecto

8.5 Cronograma general de actividades para la elaboración del proyecto.

N°	Actividad	Semanas												
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**
1	Gestión del proyecto													
2	Analizador Léxico													
3	Analizador Sintáctico													
4	Analizador Semántico													
5	Generación de código													
6	Ejecución													
7	Definición de clases y elementos													
8	Representación gráfica													
9	Constructor de escenarios													
10	Construcción del editor													
11	Construcción del ejecutor													

Anexo 5: Cronograma general para la elaboración del proyecto