Trabajo seleccionado del CoNaIISI 2016

Herramienta para el desarrollo de habilidades de programación en estudiantes no videntes

María Julia Blas[1]
Diego García Lozano[2]
Marta Castellaro[3]

[1] E-mail: mariajuliablas@santafe-conicet.gov.ar
INGAR Instituto de Desarrollo y Diseño
CONICET – UTN Facultad Regional Santa Fe

[2] E-mail: diegogarcialozano95@gmail.com
UTN Facultad Regional Santa Fe

[3]E-mail: mcastell@frsf.utn.edu.ar
UTN Facultad Regional Santa Fe

RESUMEN

En el ámbito universitario, la accesibilidad es un área de investigación y atención. La existencia de herramientas de software que contribuyen al aprendizaje de personas no videntes ha crecido, pero su aplicación y grado de apoyo varían según las disciplinas y habilidades requeridas. No siempre es posible encontrar aplicaciones que cumplan los objetivos específicos de una cátedra. En este trabajo, se presenta una herramienta de soporte a la enseñanza de programación basada en experiencias obtenidas a lo largo del cursado de un alumno no vidente. Las dificultades detectadas junto con las soluciones adoptadas, constituyen las bases de la herramienta propuesta.

ABSTRACT

Accessibility is an important research area at university level. The amount of software tools that helps blind people to learn has grown, but their application and support vary according to the disciplines and skills required. It is not always possible to find applications that meet all the objectives of a class. In this paper, we present a teaching support tool that helps blind students to program. The functions included in tool are based on the experiences of a blind student in the algorithms class, including the problems detected and the adopted solutions.

PALABRAS CLAVE:

Accesibilidad, programación no vidente, herramienta de soporte, aplicación de software.

INTRODUCCIÓN

El desarrollo de habilidades de programación es un tema prioritario en la enseñanza de ciencias de la computación. Son muchos los autores que a lo largo de los años han realizado numerosos esfuerzos para facilitar esta tarea. Desde la programación por demostración [1-2] hasta la programación visual [3] (pasando por programación mediante ejemplos [4], programación gráfica [5] y programación física [6]), el número de herramientas de programación preparadas para cubrir las necesidades específicas se ha incrementado.

En la actualidad, las herramientas de desarrollo facilitan el aprendizaje de técnicas de programación en base al análisis de diferentes aspectos y/o propiedades. Esto ha dado lugar a usuarios finales con mejores habilidades para programar, ya que cada usuario puede buscar el conjunto de herramientas que mejor se adapte a sus necesidades sin incurrir en un gran esfuerzo y/o costo. Sin embargo, la mayoría de estas herramientas utiliza interfaces de usuario centradas en la visualización (colores, secciones, indentación, resaltado, imágenes, íconos) para enfatizar aquellos aspectos relevantes que deben ser tenidos en cuenta durante el aprendizaje. Es evidente entonces, que estas herramientas no se encuentran orientadas al aprendizaje de programación para personas no videntes.

En este contexto, múltiples trabajos han detallado diferentes estrategias y soluciones aplicables a esta problemática [7-14]. Sin embargo, el proceso de enseñanza que da lugar al aprendizaje inicial, no es sencillo de llevar a cabo. Aunque existen muchas herramientas disponibles, la mayoría de ellas apunta a brindar soporte al proceso posterior al aprendizaje inicial (dando por sentado que el usuario -en este caso, estudiante- tiene conocimientos previos en el área). Estas soluciones (o combinaciones de ellas) son aplicables en un contexto controlado, pero la realidad cambia cuando uno de los alumnos del curso tiene dificultades para leer el material proporcionado y/o para resolver los ejercicios propuestos de la misma forma que sus compañeros. Esta situación se evidenció en 1º año de Ingeniería en Sistemas de Información a inicios del ciclo lectivo 2016, llevando a que los docentes de las diferentes cátedras busquen nuevas estrategias de enseñanza aplicables al caso.

Específicamente, dentro de la cátedra Algoritmos y Estructuras de Datos (AEDD), la particularidad de enseñar programación en C++ a un alumno no vidente no pareció presentar un obstáculo. Dada la cantidad de herramientas disponibles, parecía factible utilizar una combinación de las mismas a fin de llevar a cabo las clases teórico-prácticas sin afectar el desempeño del alumno. En este punto, es importante destacar que alumno no poseía conocimientos de programación previos al ingreso universitario, por lo que todas las herramientas a utilizar constituían nuevas experiencias dentro del proceso de aprendizaje.

Aunque la utilización de las herramientas disponibles sentó una base sólida para el trabajo cotidiano, muchos aspectos importantes quedaban relegados al entendimiento y accionar del alumno. Dentro del aula, estos aspectos podían ser solucionados con una explicación del docente, pero el problema se acrecentaba cuando estas dificultades se presentaban por fuera de la clase. Además, al profundizar los temas curriculares y plantear el desarrollo de programas más extensos de forma colaborativa, la lectura de códigos desarrollados por otros compañeros constituía para el alumno no vidente una tarea altamente difícil y tediosa. Por estos motivos, desde la cátedra se impulsó el diseño e implementación de una herramienta de soporte a la tarea de programación que facilite el tratamiento de los problemas ya detectados y, al mismo tiempo, posibilite la incorporación de nuevas características a partir de problemas futuros. La herramienta propuesta posibilita al usuario incorporar marcas y sugerencias dentro de un código ya desarrollado; permitiéndole al alumno leer e interpretar desarrollos realizados por otras personas, como así también depurar sus propios códigos. Este trabajo se presenta a fin de difundir la propuesta en la comunidad educativa no sólo para divulgar

la herramienta sino también para enriquecer los aspectos trabajados con nuevas perspectivas.

El resto del artículo se encuentra estructurado de la siguiente manera. La sección 2 presenta el conjunto de herramientas que comúnmente utilizan los programadores no videntes. Un alumno con discapacidad visual que quiere aprender a programar tiene a su disposición múltiples herramientas, por lo que esta sección resume y analiza las aplicaciones básicas requeridas [15]. La sección 3 detalla el contexto de la propuesta, poniendo énfasis en los problemas detectados en el alumno no vidente durante el cursado del primer cuatrimestre. La sección 4 conceptualiza el diseño de la herramienta a fin de sentar las bases necesarias para solucionar de forma automática los problemas previamente identificados. La sección 5 introduce los resultados obtenidos hasta el momento. Finalmente las secciones 6 y 7 sintetizan, respectivamente, los trabajos futuros y las conclusiones del trabajo desarrollado.

HERRAMIENTAS DE UTILIDAD PARA DESARROLLADORES NO VIDENTES

SISTEMA OPERATIVO (SO)

En la actualidad, existe una innumerable cantidad de SO disponibles. Sin embargo, su nivel de adecuación para personas no videntes es bajo, ya que no existen suficientes usuarios como para garantizar una mejor accesibilidad.

Sólo el 1% de las personas ciegas utiliza Linux. Esto se debe a que, entre otras cosas, los lectores de pantalla disponibles no son lo suficientemente avanzados como en otros SO y el manejo del escritorio no es simple e intuitivo. Aunque en un principio puede parecer el SO más apropiado (debido a que es sólo texto), una de sus principales desventajas es la dificultad de trabajo con navegadores ya que el cambio de perspectiva (de escritorio a Internet) complejiza el entendimiento del contenido.

Por su parte, el mayor inconveniente que posee Mac OS X para con los usuarios ciegos es que es cerrado. Mientras que en Linux y Windows existen múltiples herramientas de accesibilidad para ser instaladas (por ejemplo, lectores de pantalla y sintetizadores de voz), en este SO sólo pueden instalarse herramientas propietarias. De esta manera, la falta de libertad en los usuarios para elegir las herramientas de accesibilidad más convenientes de acuerdo a sus necesidades se transforma en su principal desventaja.

Desde este punto de vista, Windows es uno de los SO más simples de manipular por personas ciegas ya que flexibiliza la mayoría de los aspectos mencionados. Sin embargo, es importante destacar que cualquier SO puede ser utilizado por usuarios no videntes. Todo depende del objetivo que se tenga en mente al momento de instalarlo en el equipo.

LECTOR DE PANTALLA (LP)

Múltiples estudios demuestran que al utilizar aplicaciones basadas en audio las personas ciegas desarrollan y estimulan su cognición [16–18]. Los LP utilizan esta estrategia de forma tal de reproducir la información visualizada en un monitor como una lectura textual de contenido. En términos generales, estas herramientas constituyen un programa de software que provee una interfaz entre el SO, las aplicaciones y el usuario [19].

Comúnmente, el usuario indica comandos con diferentes combinaciones de teclas a fin de solicitar al lector que informe los cambios de texto que tienen lugar dentro de la pantalla. Cada comando refiere a diferentes acciones, a saber: deletrear una palabra, leer una línea completa, leer un texto completo, encontrar una cadena de caracteres dentro del texto, localizar el cursor o seleccionar un ítem. Lectores más avanzados poseen mayor cantidad de funciones, como ser: localizar texto de un color específico, leer partes preseleccionadas, leer únicamente texto resaltado, identificar la opción activa dentro de un menú, entre otras.

Entre los LP más populares se destacan JAWS (Job Access WithSpeech) [20], NVDA (NonVisual Desktop Access) [21], Win-

dow-Eyes [22], VoiceOver [23], Orca [24], Speakup Project [25], Google Talkback y ChromeVox [26].

ENTORNO DE DESARROLLO INTEGRADO (INTEGRATED DEVELOPMENT ENVIRONMENT – IDE)

Un IDE es una herramienta informática que proporciona servicios integrales para el desarrollo de software [27]. Generalmente, incluye un editor de código fuente, herramientas de construcción automática y un depurador. Además, suele incluir compilador y/o intérprete.

La mayoría de las características incluidas en los IDE actuales son de utilidad para los desarrolladores ciegos. Esto les facilita la tarea de programación, ya sea por el auto-completado inteligente de código y/o por la navegación por bloques. Sin embargo, una de las características más requeridas (y que no todos los IDE presentan), es el uso de comandos de teclado para acceder a las operaciones. Esta particularidad es altamente deseable para los usuarios no videntes ya que les permite prescindir del LP.

ENSEÑANZA DE PROGRAMACIÓN A ALUMNOS NO VIDENTES: DIFICULTADES

A lo largo del 1º cuatrimestre, el programa de la cátedra AEDD propone desarrollar un conjunto de contenidos que buscan introducir al alumno en las nociones básicas de programación. En una primera instancia, se utiliza un pseudo-interprete en lenguaje natural (que posibilita la adaptación del alumno al pensamiento algorítmico), para luego pasar al paradigma de programación estructurada en C++. De esta manera, los alumnos realizan una transición paulatina entre el lenguaje natural y los lenguajes de programación.

En las clases de teoría los docentes utilizan ejemplos motivadores como parte del proceso de enseñanza. Así, mientras el docente desarrolla los conceptos teóricos, el alumno puede asimilar tales conceptos dentro de un caso práctico específico. Por su parte, las clases de práctica se dividen en dos tipos de actividades basadas en guías de trabajo: actividades en aula y actividades en laboratorio. En el ámbito del aula, las prácticas se llevan a cabo en papel a fin de que el alumno plasme sus ideas de forma individual. De esta manera, se fomenta el desarrollo de estrategias de solución y no la construcción de soluciones por prueba y error. En el ámbito del laboratorio, los alumnos trabajan de forma cooperativa y colaborativa haciendo uso de herramientas de software que les facilitan la codificación de las soluciones. Además del IDE propuesto por la cátedra [28] utilizan URI Online Judge [29] como mecanismo de auto-evaluación para verificar la correctitud de las soluciones propuestas (para un conjunto de ejercicios previamente identificados por los docentes).

DIFICULTADES Y PROBLEMAS IDENTIFICADOS

Durante el desarrollo de las clases teóricas, la utilización del lector JAWS [20] permitió al alumno no vidente acceder al material bibliográfico (tanto libros como transparencias) proporcionado por la cátedra. La lectura de los códigos de ejemplo se desarrolló combinando el lector con la visualización del código fuente en el IDE, lo que facilitó el seguimiento de las estrategias de solución aplicadas.

En contraposición, durante las clases prácticas se evidenció una mayor cantidad de inconvenientes para comprender las soluciones de los ejercicios. Estas dificultades se relacionaron con aspectos propios de ubicación, localización y especificación de los códigos desarrollados y no con la elaboración de los algoritmos. Al iniciar el cursado, la habilidad del alumno para codificar ejercicios simples en C++ (20-30 líneas) era similar a la de sus compañeros. Sin embargo, al avanzar en los contenidos y dirigir la programación hacía ejercicios más complejos y extensos, se evidenció la falta de mecanismos de soporte para ubicar al usuario dentro de un archivo fuente y poder ayudarlo a comprender la estructura del programa. Se detallan a continuación las dificultades detectadas hasta el momento.

1. Ubicación en el código fuente. La identificación por número de línea es una de las principales formas de ubicación dentro del có-

digo fuente de un programa. Esta localización es de utilidad al momento de corregir errores de sintaxis y/o depurar el código en busca de otro tipo de errores. Sin embargo, en el caso de los programadores ciegos, sirve además como soporte para entender el programa, permitiéndoles analizar las expresiones utilizadas línea a línea y contextualizarlas dentro del desarrollo remanente. Esta característica es de utilidad en códigos reducidos, pero en códigos extensos la simple numeración de las líneas puede no aportar información acerca del contenido del programa.

Tómese como ejemplo la expresión (1), la cual muestra textualmente el contenido que obtendrá un programador ciego del LP al posicionarse con el cursor del IDE en la línea 10 de un código fuente cualquiera. Como puede observase, es difícil determinar la correctitud de la línea sin conocer el contexto dentro del cual se encuentra inserta. Claro está que este problema no se limita a programadores no videntes, ya que para comprender cualquier expresión es necesario analizar su contexto. Sin embargo, en el caso de personas no videntes, esta lectura es una tarea altamente costosa ya que implica desplazar el cursor hacia las líneas precedentes y consecuentes en espera de que el LP las traduzca. En estos casos, proveer (al menos) la información asociada al método, función o estructura dentro del cual se encuentra una línea de código es información de valor para los usuarios ciegos.

10 if(cordenadax>0 &&cordenaday>0){ (1)

2. Delimitación de bloques. Muchos lenguajes de programación utilizan llaves para delimitar bloques de código. Cuando los estudiantes aprenden a programar, frecuentemente confunden la delimitación de bloques por medio de llaves con otro tipo de estrategias (por ejemplo, el uso de sangrías). Aún más, es común que los alumnos de los niveles iniciales no tengan en claro los objetivos de los bloques que definen, por lo que suelen trabajar cerrando llaves en lugares incorrectos. Esta situación da lugar a bloques de códigos mal definidos.

Para facilitar la delimitación de bloques, los IDE suelen utilizar colores. La Figura 1 presenta un código C++ desarrollado en Zinjal [28], dentro del cual se visualiza esta particularidad. Como puede observarse, el cursor se encuentra posicionado en una de las llaves de cierre ubicada en la línea 23. Este posicionamiento da como consecuencia que el par de llaves involucradas en el bloque definido sea resaltado en color rojo (llaves de las líneas 12 y 23).Aunque esta información es útil para la mayoría de los programadores, las personas ciegas no poseen ningún mecanismo de soporte que les permita evidenciar este comportamiento.

3. Separación de líneas. En un programa C++ todas las sentencias que no se corresponden con estructuras de control deben finalizar con punto y coma. En un caso ideal, cada línea de un archivo debería contener una (y solo una) sentencia. Sin embargo, los lenguajes de programación no suelen restringir la estructuración de sus códigos a un formato específico, ya que esto depende del editor utilizado para dar soporte al lenguaje y no del lenguaje en sí. Por este motivo, es común encontrar códigos C++ con líneas que siguen el esquema presentado en la expresión (2), donde se visualizan dos sentencias diferentes (una de salida estándar y otra de entrada estándar) dentro de una misma línea (la línea 5).

5 cout<<”Ingrese un dato”; cin>>x; (2)

Para las personas ciegas, la lectura organizada del código es de mucha utilidad. Este tipo de lectura les posibilita analizar el contenido de un programa de forma sectorizada, permitiéndoles comprender con mayor nivel de certeza las instrucciones, ya que focalizan su atención en la lectura de una sentencia por vez (es decir, una traducción del LP por línea). Además, les facilita la lectura carácter a carácter debido a que pueden reconocen cada punto y coma como la marca de fin de una sentencia. En este contexto, la individualización de sentencias en una única línea constituye un problema a trabajar.

```
#include <iostream>
using namespace std;

int main(int argc, char *argv[]) {
float cordenadax, cordenaday;
cout<<"ingrese el punto en el plano carteciano"<<endl;
cin>>cordenadax;
cin>>cordenaday;

    if (cordenadax>0 && cordenaday>0) {
        cout<<"el punto está en el primer cuadrante"<<endl;}
    else {
    if (cordenadax<0 && cordenaday>0) {
    cout<<"el punto está en el segundo cuadrante"<<endl;}
    else {
    if (cordenadax<0 && cordenaday<0) {
        cout<<"el punto está en el tercer cuadrante"<<endl;}
    else {
if (cordenadax>0 && cordenaday<0) {
        cout<<"El punto está en el cuarto cuadrante"<<endl;}
else {
    cout<<"es el origen"<<endl;
}}}}
    return 0;
}
```

Figura 1: Ejemplo C++ del mecanismo de delimitación de bloques que utiliza el IDE Zinjal.

4. Uso de acentos y distinción entre letras mayúsculas y minúsculas. Habitualmente los lenguajes de programación estructurados son case sensitive. Esto quiere decir que, dado un texto, se da importancia tanto a las mayúsculas como a las minúsculas para expresar unicidad. Lo mismo ocurre, en algunos lenguajes que utilizan codificaciones modernas, con el uso de caracteres acentuados.

En general, los alumnos que inician cátedras de programación utilizan únicamente letras minúsculas (en la mayoría de los casos, sin incluir acentos) para realizar sus definiciones. Como resultado, no evidencian las características del lenguaje hasta encontrarse en etapas avanzadas del aprendizaje. Sin embargo, en el caso de programadores no videntes, su propia formación los lleva a programar códigos de acuerdo a las normas de escritura tradicionales (como ser las reglas de ortografía). Esta situación se pone en evidencia al dar los primeros pasos en programación, dando como resultado la aparición de múltiples errores de sintaxis al compilar los códigos desarrollados. Ante la ocurrencia de estos errores, el LP no brinda una ayuda simple y clara. Mientras que la lectura palabra por palabra omite la distinción de caracteres especiales, la lectura carácter a carácter (en su forma estándar) no suele estar configurada para detectar esta diferencia. Entonces, la búsqueda de estrategias que ayuden a evitar la ocurrencia de estos errores constituye un problema de interés.

DIFICULTADES Y PROBLEMAS A SER DETECTADOS EN EL FUTURO

Es probable que, al avanzar con los contenidos, se evidencien nuevas dificultades en relación a la codificación de soluciones. Tales dificultades deberán ser afrontadas tanto por el alumno no vidente como por los docentes a cargo de su comisión. Esta situación implica que, al momento de plantear una solución automatizada para los problemas ya identificados, es importante conceptualizar una estrategia de solución que permita incorporar nuevas características a fin de dar solución a eventuales problemas futuros.

AUTOMATIZACIÓN: HERRAMIENTA PARA "HACER EXPLÍCITO LO IMPLÍCITO"

Dadas las dificultades identificadas y considerando la posibilidad de identificar dificultades a futuro, desde la cátedra de AEDD se impulsó la idea colaborativa de desarrollar una herramienta de software que permita re-

solver estos problemas de forma automática.

Conceptualmente, la herramienta se encuentra diseñada como software de soporte al programador no vidente a fin de simplificar las tareas de lectura y análisis de código fuente. Para esto, el programador deberá elegir un código C++ implementado en un IDE cualquiera y, luego, haciendo uso de la herramienta propuesta generará un código equivalente que mantendrá el contenido original incorporando un conjunto de marcas a fin de agilizar su interpretación. Cada tipo de marca actuará como mecanismo de resolución de una o más de las dificultades identificadas. El código resultante de este proceso de marcado no será más que un nuevo código fuente C++ que podrá abrirse y editarse con cualquier IDE. De esta manera, no se pierden las bondades de los IDE ni se imposibilita el uso de herramientas complementarias por parte de los usuarios no videntes (como por ejemplo el LP).

DESARROLLO BASADO EN PROTOTIPOS EVOLUTIVOS

El desarrollo basado en prototipos posibilita la construcción de modelos de aplicaciones de software sobre los cuales es posible analizar las funcionalidades básicas requeridas sin necesidad de incluir toda la lógica y/o características del modelo final. Así, permite al cliente evaluar en forma temprana el producto e interactuar con diseñadores y desarrolladores para determinar si el desarrollo actual cumple con las expectativas [30].

El modelo basado en prototipos pertenece al conjunto de modelos de desarrollo evolutivos, ya que cada prototipo es evaluado por el cliente a fin de lograr una retroalimentación con la que se refinan los requisitos del software y se ajusta el prototipo actual. Esto permite que, al mismo tiempo, el desarrollador entienda que es lo que se debe construir y el cliente tenga resultados a corto plazo. Si sobre el modelo evolutivo se incorpora una estrategia de desarrollo incremental, se da lugar a un modelo de desarrollo evolutivo e incremental. Este tipo de modelos brinda la posibilidad de controlar la complejidad y los riesgos, desarrollando una parte del producto software en una etapa y reservando el resto de los aspectos requeridos para el desarrollo futuro. De esta manera, los prototipos desarrollados son mecanismos de prueba de un conjunto de funcionalidades que condensan la idea principal del sistema en desarrollo, aumentando su funcionalidad con el paso del tiempo.

Para la herramienta propuesta, las ventajas de utilizar un desarrollo basado en prototipos son evidentes. No solo permitirá evaluar la adecuación de las soluciones propuestas para cada una de las dificultades identificadas, sino que además posibilitará la incorporación de nuevas funcionalidades a medida que avance el desarrollo. Teniendo en cuenta que existe un único usuario con la capacidad de evaluar las soluciones propuestas, es altamente provechoso lograr una retroalimentación inmediata de forma tal que se agilice el proceso de desarrollo en base a un esquema de mejora continua del producto de software resultante.

ESTILO ARQUITECTÓNICO: PIPELINE

Los sistemas de flujo de datos se caracterizan por la forma en la cual se mueve la información a través del sistema. En general, su arquitectura tiene dos o más componentes de procesamiento (CP) que mapean de diferente forma los datos de entrada en datos de salida. La naturaleza de estos datos no queda restringida por el estilo arquitectónico [31]. Si los vínculos entre componentes se dan de forma secuencial, la arquitectura corresponde a un modelo pipeline. En este modelo, el flujo de datos de salida de un componente se transforma en el flujo de datos de entrada del siguiente.

Si se piensa que existe un CP por cada dificultad identificada y que cada CP trabaja en la automatización de la solución propuesta para tal dificultad, es posible plantear el diseño de la herramienta como una arquitectura pipeline (Figura 2). De acuerdo con este esquema, cada CP actúa sobre una versión del código fuente a fin de modificarla (por medio de incorporación de marcas) y retransmitirla

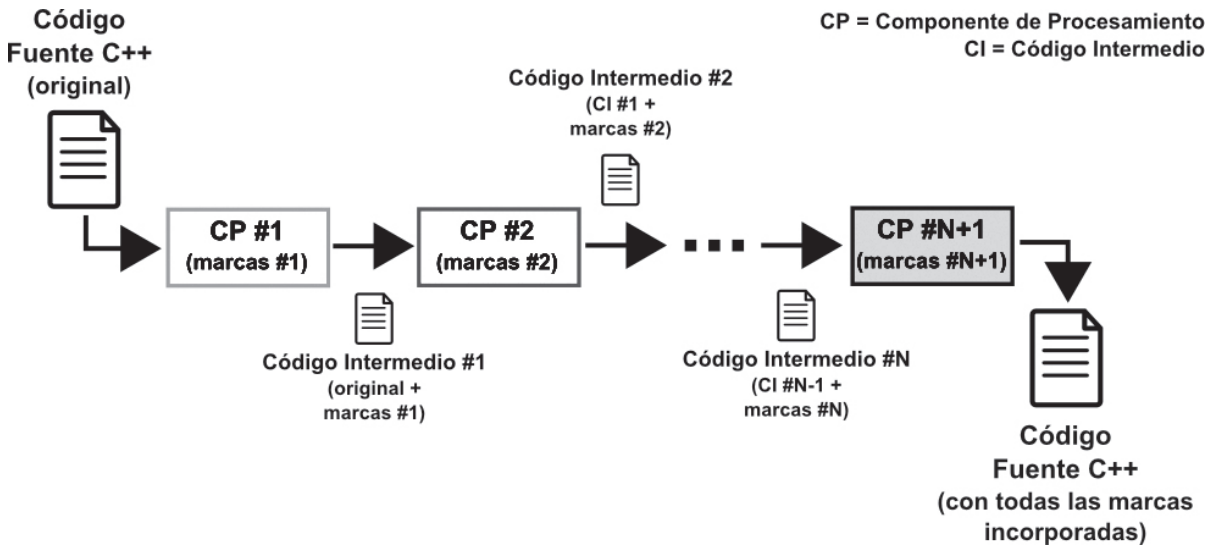

Figura 2: Arquitectura pipeline de la herramienta propuesta.

como flujo de salida hacia otro componente.

Tomando como base a las dificultades previamente enunciadas, se planteó un conjunto de soluciones automatizadas a ser implementadas en los primeros cuatro CP. Estas soluciones se detallan en los siguientes apartados. Por su parte, las responsabilidades de los CP restantes deberán detallarse a medida que se detecten nuevas dificultades y/o problemas.

CP #1 - Formato legible del código fuente: Aunque la legibilidad es una propiedad del código fuente a nivel visual, la organización de la información resultante es de mucha ayuda para los programadores no videntes. Por este motivo, se propone como objetivo del primer CP dar un formato comprensible a un código fuente cualquiera a fin de garantizar la correcta legibilidad de su contenido. Teniendo en cuenta que existen múltiples herramientas desarrolladas específicamente para estos fines (embellecedores, beautifiers, formateadores, entre otros), se propone realizar un análisis de las aplicaciones de código abierto existentes a fin de seleccionar una de ellas y utilizarla como primer módulo del software bajo desarrollo. Es importante destacar que el formateador de código debe ser el primer módulo de la herramienta ya que la información resultante posee un patrón de formato, simplificando el trabajo de procesamiento a realizar en los componentes restantes.

CP #2 - Comentarios descriptivos en el cierre de bloques: A fin de delimitar explícitamente los bloques de código, se propone la inserción de comentarios descriptivos luego de la llave de cierre asociada al bloque. Específicamente, el comentario a insertar deberá indicar el contexto en el cual está actuando el contenido del bloque. Esta responsabilidad estará asociada al objetivo del segundo CP de la herramienta.

CP #3 –Eliminación de caracteres alfabéticos con acentos y letras mayúsculas: A fin de solucionar los problemas asociados con los caracteres acentuados y letras mayúsculas, se propone modificar el código entrante reemplazando tales caracteres por los caracteres en minúsculas no acentuados equivalentes. Esta transformación será incorporada a la herramienta como objetivo del tercer CP.

CP #4 - Descriptor de cantidad de líneas por función: A fin de especificar un tamaño para cada función, se propone incorporar como parte del encabezado un comentario descriptivo que indique la cantidad de líneas involucradas y los números de línea de inicio y fin asociados. De esta manera, el lector puede (sin necesidad de conocer el desarrollo de la función) dimensionar su extensión. La im-

plementación de esta responsabilidad queda como objetivo asociado al cuarto CP de la herramienta.

RESULTADOS PRELIMINARES

Tal como se ha mencionado con anterioridad, la herramienta propuesta en este trabajo se encuentra actualmente en proceso de desarrollo. Un becario de grado ha sido asignado a esta tarea, trabajado en estrecha relación con los docentes y auxiliares de la cátedra. El lenguaje de programación Java fue elegido para la implementación debido a sus ventajas en cuanto a portabilidad y accesibilidad.

Teniendo en cuenta la finalidad del primer CP, se realizó un estudio de los embellecedores existentes. Se decidió trabajar con Uncrustify [32] debido a que, además de ser una herramienta de código abierto, es lo suficientemente completa como para abarcar todos los posibles problemas de formato asociados a un código C++. Esta herramienta se ejecuta por consola y modifica el archivo fuente original, por lo que provee además las interfaces necesarias para trabajar en conjunto con

```
#include <iostream>
using namespace std;

int main(int argc, char *argv[]) {
 float cordenadax, cordenaday;
 cout<<"ingrese el punto en el plano carteciano"<<endl;
 cin>>cordenadax;
 cin>>cordenaday;

     if (cordenadax>0 && cordenaday>0) {
          cout<<"el punto está en el primer cuadrante"<<endl;}
     else {
     if (cordenadax<0 && cordenaday>0) {
     cout<<"el punto está en el segundo cuadrante"<<endl;}
     else {
     if (cordenadax<0 && cordenaday<0) {
         cout<<"el punto está en el tercer cuadrante"<<endl;}
     else {
if (cordenadax>0 && cordenaday<0) {
         cout<<"El punto está en el cuarto cuadrante"<<endl;}
else {
     cout<<"es el origen"<<endl;
}}}}
     return 0;
 }
```

```
#include <iostream>
using namespace std;
int main(int argc, char *argv[]) {
 float cordenadax, cordenaday;
 cout<<"ingrese el punto en el plano carteciano"<<endl;
 cin>>cordenadax;
 cin>>cordenaday;
if (cordenadax>0 && cordenaday>0) {
 cout<<"el punto está en el primer cuadrante"<<endl;
}//CIERRA EL BLOQUE DE if (cordenadax>0 && cordenaday>0)
else {
if (cordenadax<0 && cordenaday>0) {
 cout<<"el punto está en el segundo cuadrante"<<endl;
}//CIERRA EL BLOQUE DE if (cordenadax<0 && cordenaday>0)
else {
if (cordenadax<0 && cordenaday<0) {
 cout<<"el punto está en el tercer cuadrante"<<endl;
}//CIERRA EL BLOQUE DE if (cordenadax<0 && cordenaday<0)
else {
if (cordenadax>0 && cordenaday<0) {
 cout<<"El punto está en el cuarto cuadrante"<<endl;
}//CIERRA EL BLOQUE DE if (cordenadax>0 && cordenaday<0)
else {
 cout<<"es el origen"<<endl;
}//CIERRA EL BLOQUE DE else DE if (cordenadax>0 && cordenaday<0)
}//CIERRA EL BLOQUE DE else DE if (cordenadax<0 && cordenaday<0)
}//CIERRA EL BLOQUE DE else DE if (cordenadax<0 && cordenaday>0)
}//CIERRA EL BLOQUE DE else DE if (cordenadax>0 && cordenaday>0)
 return 0;
}//CIERRA EL BLOQUE DE int main(int argc, char *argv[])
```

Figura 3: Prueba del prototipo: código original (izquierda) vs. código equivalente (derecha).

el resto de los componentes propuestos. De forma adicional se implementó una primera aproximación del segundo CP propuesto.

Con ambos CP implementados, se procedió a la prueba del primer prototipo. Para esto, se tomó un conjunto de códigos de ejemplo desarrollados por el alumno no vidente a lo largo del cursado. La Figura 3 visualiza el mismo código de ejemplo (implementado en Zinjal), en dos instancias diferentes: antes de la ejecución del prototipo (izquierda) y después de la ejecución del prototipo (derecha). Como puede observarse, además de tener mayor legibilidad, el código resultante incorpora un conjunto de comentarios descriptivos. Estos comentarios permiten al programador no vidente interpretar su significado una vez que han sido traducidos por el LP. Existen dos aspectos importantes del código resultante a destacar. Primero: no posee indentaciones. Esto se debe a que los LP tienen únicamente la capacidad de leer contenido (es decir, no leen espacios en blanco y/o tabulaciones bajo el modo de lectura tradicional). En este contexto, la incorporación de indentaciones como parte del código sólo dificultaría su lectura. Segundo: la cantidad de líneas de código se ha incrementado (de 25 a 30) como resultado del proceso de transformación. Sin embargo, aunque el código es más extenso, su interpretación es mucho más sencilla.

TRABAJOS FUTUROS

En base a la estrategia propuesta, se espera completar el diseño y desarrollo de la herramienta incorporando los CP restantes junto con nuevos módulos que respondan a dificultades futuras. La arquitectura elegida facilita la adición de estos componentes en interacción con los ya implementados por medio del intercambio de información.

La construcción basada en prototipos evolutivos permitirá realizar una prueba en las diferentes etapas de desarrollo, lo que posibilitará mejorar cada uno de los módulos implementados de forma independiente. En consecuencia, al finalizar la construcción de la herramienta el proceso de pruebas estará concluido.

Dentro de la facultad, la herramienta quedará a disposición del alumno no vidente para que la utilice a lo largo del aprendizaje. Podrá ser mejorada por las cátedras que así lo requieran a medida que el alumno avance en la carrera. Se dejará disponible para su descarga una versión empaquetada que podrá ser utilizada por cualquier usuario que así lo desee.

Como extensión de este trabajo, existe la posibilidad de desarrollar un complemento para el IDE Zinjal [28] de forma tal que las marcas y sugerencias se incorporen al código fuente a medida que el programador no vidente se encuentra codificando sus archivos.

CONCLUSIONES

Se ha presentado una herramienta de soporte a la enseñanza de programación pensada para ser utilizada por alumnos no videntes. Tanto la incorporación de nuevas características como la mejora de las existentes, forman parte del proceso continuo de adaptación de la herramienta, no restringiendo su ámbito a la cátedra AEDD. Tal como se ha mencionado con anterioridad, la arquitectura definida posibilita la incorporación de nuevas características. Aunque los principales aspectos de la herramienta se encuentran aún bajo desarrollo, los resultados preliminares han demostrado ser de utilidad para el alumno.

En este contexto, es importante destacar que la accesibilidad es un tema que aún debe ser atendido y explorado a nivel universitario. A diferencia de la educación primaria y secundaria, donde se tienen asistentes especiales y talleres extracurriculares de apoyo, a nivel universitario existe una ausencia de mecanismos de contención que ayuden tanto al docente como al alumno a lograr un verdadero aprendizaje.

REFERENCIAS

[1] Cypher, A.; Halbert, D.C. (1993). Watch what I do: programming by demonstration. MIT Press.

[2] Mc Daniel, R.; Myers, B. (1999). Getting more out of programming-by-demonstration. Human factors in computing systems, ACM CHI'99 Proceeding, 1(1), 442-449.

[3] Rosson, M.; Seals, Ch. (2001). Teachers as simulation programmers: minimalist learning and reuse. Human factors in computing systems, ACM CHI'01 Proceedings, 1 (2), 237-244.

[4] Liberman, H. (2001). Your wish is my command: Programming by example. Morgan Kaufmann.

[5] Travers, M. (1994). Recursive interfaces for reactive objects. Human factors computing systems, ACM CHI'94 Proceedings, 1(1), 379-385.

[6] Montemayor, J. (2001). Physical programming: Software you can touch. Human factors in computing systems, ACM CHI'01 Extended abstracts, 1(1), 81-82.

[7] Sánchez, J.; Aguayo, F. (2005). APL: Un Lenguaje de Programación basado en Audio para Aprendices Ciegos. IE Comunicaciones: Revista Iberoamericana de Informática Educativa, 1(2005), 31-38.

[8] Kopecek, I.; Jergová, A. (1997). Programming and visually impaired people. Proceedings of the XV. World Computer Congress ICCHP 1997, 365-372.

[9] Frauenberger, Ch.; Noistering, M. (2003). 3D audio interfaces forblind. Proceedings of international conference on auditory display, Boston, USA, 280-283.

[10] Smith, A.C.; Francioni, J.M.; Matzek, S.D. (2000). A Java programming to for

students with visual disabilities. 4th ACM conference on assistive technologies proceedings, 1(1), 142-148.
[11] Siegfried, R.M. (2006). Visual programming and the blind: the challenge and the opportunity. ACM SIGCSE Bulletin, 38 (1), 275-278.
[12] Kirchner, C.; Schmeidler, E. (2001). Adding audio description: Does´t make a difference?. Journal of Visual Impairment & Blindness, 95 (4), 197-212.
[13] Alistair, E. (1989). Soundtrack: An auditory interface for blind users. Human-computer interaction, 4(1), 45-66.
[14] Sánchez, J.; Aguayo, F. (2005). Blind learners programming through audio. Human factors in computing systems, ACM CHI'05 Extended abstracts, 1, 1769-1772.
[15] The Tools of a Blind Programmer. Disponible en https://www.parhamdoustdar.com/2016/04/03/tools-of-blind-programmer/. [Último acceso: 21-Jul-2016].
[16] Mereu, S.W.; Kazman, R. (1996). Audio enhanced 3D interfaces for visually impaired users. Human factors in computing systems. ACM CHI '96 Proceedings, 1(1), 72-78.
[17] Sánchez, J.; Lumbreras, M.; Cernuzzi, L. (2001). Interactive virtual acoustic environments for blind children: computing, usability, and cognition. Human factors in computing systems, ACM CHI'01 Extended abstracts, 1(1), 65-66.
[18] Sánchez, J.; Baloian, N.; Hassler, T.; Hoppe U. (2003). Audio battleship: blind learners collaboration through sound. Human factors in computing systems, ACM CHI'03 Extended abstracts, 1(1), 798-799.
[19] American Foundation for the Blind. Disponible en http://www.afb.org/default.aspx. [Último acceso: 21-Jul-2016].
[20] Jaws Screen Reader - Best in Class. Disponible en http://www.freedomscientific.com/Products/Blindness/JAWS. [Último acceso: 21-Jul-2016].
[21] NV Access. Disponible en http://www.nvaccess.org/. [Último acceso: 21-Jul-2016].
[22] GW Micro - Window-Eyes. Disponible en http://www.gwmicro.com/window-eyes/. [Último acceso: 21-Jul-2016].
[23] Accessibility - OS X – Voice Over - Apple. Disponible en http://www.apple.com/ accessibility/osx/voiceover/. [Último acceso: 21-Jul-2016].
[24] Orca. Disponible en https://help.gnome.org/users/orca/stable/. [Último acceso: 21-Jul-2016].
[25] The Speakup Project. Disponible en http://www.linux-speakup.org/. [Último acceso: 21-Jul-2016].
[26] Chrome Vox. Disponible en http://www.chromevox.com/. [Último acceso: 21-Jul-2016].
[27] SALAVERT, Isidro Ramos y PÉREZ, María Dolores Lozano. (2000). Ingeniería del software y bases de datos: tendencias actuales. Universidad de Castilla La Mancha.
[28] Zinjal. Disponible en http://zinjai.sourceforge.net/.[Último acceso: 21-Jul-2016].
[29] URI Online Judge. Disponible en https://www.urionlinejudge.com.br. [Último acceso: 21-Jul-2016]
[30] Pressman, R.S. (2010). Software Engineering: A Practitioner´s Approach, 7th ed. McGraw-Hill.
[31] Albin, S.T. (2003). The art of software architecture: design methods and techniques. John Wiley & Sons.
[32] Uncrustify. Disponible en http://uncrustify.sourceforge.net/. [Último acceso: 21-Jul-2016].