UNIVERSIDAD DE COSTA RICA
SISTEMA DE ESTUDIOS DE POSGRADO

# VISUALIZACIONES LÚDICAS DE MÁQUINAS NOCIONALES PARA EL APRENDIZAJE CONSTRUCTIVISTA DE LA PROGRAMACIÓN DE COMPUTADORAS

Tesis sometida a la consideración de la Comisión del Programa de Estudios de Posgrado en Computación e Informática para optar al grado y título de Doctorado en Computación e Informática

JEISSON HIDALGO CÉSPEDES

Ciudad Universitaria Rodrigo Facio, Costa Rica

2018

# Dedicatoria

A Emilio, quien con su poca edad no podía entender por qué papá estuvo lejos, por esa palabra en la respuesta, pero que al menos le permitió a diario chequear con esperanza de que la respuesta a cuándo papá iba a terminar la *tesis* cambiara.

# Agradecimientos

Quiero agradecer sin detenerme a mi familia por la paciencia y el apoyo, en especial a mi esposa Ericka Méndez Chacón y a mi mamá María Hidalgo Céspedes. A Gabriela Marín Raventós, Vladimir Lara Villagrán, Carlos Vargas Castillo, y Amy Ogan, en mi comité de tesis, por su sabiduría y generosidad en formar investigadores.

A Wanda Dann, Donald Slater, Cleah Schlueter, Jacobo Carrasquel, y el equipo de Alice en *CARNEGIE MELLON UNIVERSITY* por recibirme como pasante. A Arturo Peña Hurtado por el diseño gráfico de botNeumann++. A Melissa González-Chaves por su apoyo en los análisis cualitativos. A Ken Resztak por hacer que el inglés de los artículos se pudiera entender. A mis compañeros del curso "Psicología cognitiva": Melissa Aguzzi Fallas, María Alejandra Esquivel, y Vanessa Vargas Vega por el diseño y prueba piloto del primer experimento. A los profesores de programación que incentivaron a sus estudiantes a participar en experimentos: Vladimir Lara Villagrán, Gustavo Esquivel Quirós, Edgar Casasola Murillo, Maureen Murillo Rivera. A los profesores de programación que participaron en grupos focales, como expertos del dominio en pruebas de usabilidad y evaluación de resultados experimentales: Braulio Solano Rojas, Edgar Casasola Murillo, Ricardo Gang Vincenzi, Alan Calderón Castro, Adolfo di Mare Hering. A los expertos en interacción humano computador: Marta Calderón Campos, Gustavo López Herrera, y Mariana López Quirós. A los estudiantes que participaron en la encuesta y las evaluaciones de las herramientas construidas en esta tesis para ellos. A Ricardo Villalón Fonseca por las discusiones de bajo nivel (sobre la máquina). A Susan Francis Salazar por acercarme al constructivismo y revisar que yo no me alejara de él en la propuesta de tesis. A mis compañeros de doctorado por poner el humor para sobrevivir a esta aventura.

Este trabajo de investigación fue realizado con apoyo del Centro de Investigaciones en Tecnologías de la Información y Comunicación (CITIC), la Escuela de Ciencias de la Computación e Informática (ECCI) de la Universidad de Costa Rica, y el Ministerio de Ciencia, Tecnología y Telecomunicaciones (MICITT) de Costa Rica.

Esta tesis fue aceptada por la Comisión del Programa de Estudios de Posgrado en Computación e Informática de la Universidad de Costa Rica, como requisito parcial para optar al grado y título de Doctorado en Computación e Informática.

Ileana Castillo Arias

**Representante del Decano del**

**Sistema de Estudios de Posgrado**

Dra. Gabriela Marín Raventós

**Directora de tesis**

Dr. Vladimir Lara Villagrán

**Asesor**

Dr. Carlos Vargas Castillo

**Asesor**

Ph.D Amy Ogan

**Asesora**

Ricardo Villalón Fonseca

**Representante de la Directora del**

**Programa de Posgrado en Computación e Informática**

Jeisson Hidalgo Céspedes

**Candidato**

# Tabla de contenido

**RESUMEN** ..... **VIII**

**LISTA DE CUADROS** ..... **X**

**LISTA DE FIGURAS** ..... **XII**

**LISTADOS DE CÓDIGO** ..... **XVI**

**1 INTRODUCCIÓN** ..... **1**

1.1 DEFINICIÓN DEL PROBLEMA ..... 2

1.2 OBJETIVOS ..... 7

1.3 PARADIGMA DE INVESTIGACIÓN Y METODOLOGÍA GENERAL ..... 7

*1.3.1 Requerimientos de software* ..... *14*
*1.3.2 Propuesta conceptual* ..... *15*
*1.3.3 Diseño de una solución* ..... *16*
*1.3.4 Evaluación de un prototipo* ..... *17*

1.4 ALCANCE Y DELIMITACIONES ..... 19

**2 MARCO TEÓRICO Y MODELO CONCEPTUAL** ..... **20**

2.1 MÁQUINAS NOCIONALES ..... 21

2.2 VISUALIZACIONES ..... 25

*2.2.1 Visualizaciones por computadora* ..... *26*
*2.2.2 Visualizaciones de software* ..... *27*
*2.2.3 Audiencia pretendida de las visualizaciones* ..... *29*

2.3 CONSTRUCTIVISMO SOCIOCULTURAL ..... 31

*2.3.1 Motivación para el aprendizaje* ..... *33*
*2.3.2 Contraposición conceptual* ..... *35*
*2.3.3 Asimilación del concepto* ..... *37*
*2.3.4 Aplicación del concepto* ..... *40*
*2.3.5 Formación de hábitos* ..... *41*
*2.3.6 Sistemas de conceptos* ..... *42*

2.4 METÁFORAS Y ALEGORÍAS ..... 45

*2.4.1 Metáforas en computación* ..... *49*
*2.4.2 Clasificación de metáforas por ámbito* ..... *50*
*2.4.3 Metáforas y alegorías visuales* ..... *56*
*2.4.4 Limitaciones de las metáforas* ..... *57*

2.5 LUDIFICACIÓN ..... 57

3 ANTECEDENTES Y ANÁLISIS DEL ESTADO DEL ARTE ..... 60
3.1 VISUALIZACIONES DE PROGRAMA EXISTENTES ..... 61
3.1.1 *Metodología de la revisión de literatura* ..... 62
3.1.2 *Resultados de la revisión de literatura* ..... 65
3.2 REQUERIMIENTOS DE SOFTWARE ..... 70
3.3 VERIFICACIÓN DE REQUERIMIENTOS ..... 72
3.4 ALEGORÍAS EN COMPUTACIÓN ..... 75
3.4.1 *Revisión de literatura* ..... 75
3.4.2 *Discusión de la revisión de literatura* ..... 89
3.5 EFECTO DE LA ENSEÑANZA ALEGÓRICA ..... 91
3.5.1 *Diseño experimental* ..... 91
3.5.2 *Participantes del experimento* ..... 96
3.5.3 *Sesiones experimentales* ..... 98
3.5.4 *Resultados del experimento* ..... 99
3.5.5 *Discusión de los resultados experimentales* ..... 103
4 DISEÑO Y VALIDACIÓN DE VISUALIZACIONES LÚDICAS ..... 106
4.1 PROPUESTA CONCEPTUAL ..... 107
4.2 ENCUESTA: SELECCIÓN DE LA MÁQUINA NOCIONAL ..... 111
4.3 DISEÑO 1: *PUPPETEER++* ..... 114
4.3.1 *Alegoría visual de PUPPETEER++* ..... 115
4.3.2 *Ludificación de PUPPETEER++* ..... 117
4.3.3 *Validación de PUPPETEER++* ..... 121
4.4 DISEÑO 2: BOTNEUMANN++ ..... 125
4.4.1 *Alegoría visual de botNeumann++* ..... 125
4.4.2 *Ludificación de botNeumann++* ..... 144
4.4.3 *Validación de botNeumann++* ..... 152
5 IMPLEMENTACIÓN Y EVALUACIÓN DE BOTNEUMANN++ ..... 155
5.1 PROTOTIPO 1: POWERPOINT ..... 156
5.1.1 *Implementación y lecciones aprendidas* ..... 156
5.1.2 *Evaluación de usabilidad* ..... 163
5.1.3 *Resultados de la evaluación de usabilidad* ..... 167
5.1.4 *Discusión de la evaluación de usabilidad* ..... 186
5.2 PROTOTIPO 2: C++ ..... 190
5.2.1 *Implementación y lecciones aprendidas* ..... 190
5.2.2 *Evaluación de usabilidad* ..... 197
5.2.3 *Resultados de la evaluación de usabilidad* ..... 199
5.2.4 *Discusión de la evaluación de usabilidad* ..... 203
5.3 EVALUACIÓN EXPERIMENTAL DEL PROTOTIPO C++ ..... 204

*5.3.1 Método experimental* ... *204*
*5.3.2 Participantes del experimento* ... *214*
*5.3.3 Resultados del experimento* ... *217*
*5.3.4 Discusión del experimento* ... *228*

**6 CONCLUSIONES ... 232**

6.1 DISCUSIÓN GENERAL ... 232

*6.1.1 Requerimientos de software* ... *233*
*6.1.2 Propuesta conceptual* ... *234*
*6.1.3 Diseño de una solución* ... *236*
*6.1.4 Evaluación de un prototipo* ... *238*

6.2 COMPENDIO DE LOS APORTES AL CONOCIMIENTO ... 240

6.3 TRABAJO FUTURO ... 242

6.4 PUBLICACIONES ... 244

**ANEXO A ELEMENTOS LÚDICOS ... 247**

**ANEXO B IMPLEMENTACIÓN DEL PROTOTIPO C++ ... 250**

B.1 ARQUITECTURA DEL PROTOTIPO ... 250

*B.1.1 El motor de visualización* ... *252*
*B.1.2 El motor de interacción* ... *254*
*B.1.3 El motor de ludificación* ... *255*

B.2 PANTALLA DE MENÚ ... 255

*B.2.1 Registro de eventos* ... *257*

B.3 PANTALLA DE SELECCIÓN DE NIVEL ... 259

B.4 ARCHIVO Y PANTALLA DE NIVEL ... 261

*B.4.1 Estados de la pantalla de nivel* ... *267*
*B.4.2 Estado de generación* ... *268*
*B.4.3 Estado de preparación* ... *272*
*B.4.4 El estado de animación* ... *273*

B.5 EL META-MODELO ... 274

B.6 LA ARQUITECTURA INTERNA ... 279

*B.6.1 El estado de pausa y velocidad de la animación* ... *283*

**REFERENCIAS ... 284**

**ÍNDICE DE MATERIAS ... 295**

# Resumen


En la enseñanza de la programación, los métodos estáticos con los que ampliamente se explica a los aprendices cómo los programas corren en una computadora han sido criticados y las visualizaciones de programa se han propuesto como solución. Pero, su eficacia ha sido también cuestionada revelando una necesidad de conocimiento que permita a los ingenieros construir visualizaciones de programa eficaces.

De una teoría de aprendizaje se infirieron 16 requerimientos de software que podrían influir en la eficacia de una herramienta educativa. Se encontró que las visualizaciones de programa existentes poco los satisfacen. Se conjeturó que con alegorías visuales concretas y ludificación se podrían satisfacer los requerimientos, lo que conformó una propuesta conceptual a la que se llamó *visualizaciones lúdicas de programa.*

No se encontraron visualizaciones de programa que implementaran alegorías o ludificación, pero sí un par de advertencias de efectos negativos de las alegorías en otros tipos de interfaces de usuario. Mediante una revisión de literatura y un experimento, se encontró que esas advertencias no son generalizables, sino que las alegorías tienen los mismos efectos que las metáforas inconexas usadas tradicionalmente en computación.

La carencia de alegorías visuales y ludificación en visualizaciones de programa despertó el interés científico de conocer si pueden tener un efecto positivo en la eficacia de estas herramientas para ayudar a comprender máquinas nocionales. Se diseñaron dos visualizaciones lúdicas de programa para C++ por ser el lenguaje más usado en la Escuela de Ciencias de la Computación e Informática. El primer diseño se llamó *PUPPETEER++* por usar un teatro de títeres como alegoría, pero una validación con expertos reveló que ofrecía más debilidades que fortalezas. El segundo diseño se llamó botNeumann++ por usar una fábrica robotizada futurista como alegoría y fue validado con éxito por dos investigadoras de *CARNEGIE MELLON UNIVERSITY.*

Se implementaron dos prototipos incrementales de botNeumann++ que permitieron recabar evidencia empírica sobre su eficacia. El primer prototipo fue construido con *MICROSOFT POWERPOINT* y tuvo una altísima eficacia, ya que cerca del 95% de las veces que los participantes advirtieron una fuga de memoria, fue gracias a una animación hecha por el prototipo.

El segundo prototipo fue construido con C++. Se comparó experimentalmente su eficacia contra las herramientas usadas tradicionalmente para detectar y corregir errores, dado que la comprensión de los programas en tiempo de ejecución es requisito para poder corregirlos. Se encontró un efecto estadísticamente significativo de la herramienta en la eficacia de corrección de errores de los participantes. Quienes usaron el prototipo de botNeumann++ lo consideraron más usable y corrigieron más errores en el mismo tiempo que quienes continuaron usando las herramientas a las que estaban habituados. Este efecto fue más drástico en los estudiantes que perdieron el curso. Por tanto se concluye que las visualizaciones lúdicas de programa fueron más eficaces, eficientes, y usables que las herramientas tradicionales, lo que sustenta la propuesta conceptual de esta tesis para mejorar estas herramientas.

# Abstract

In Computer Science Education field, the widely-used static methods for teaching how programs are run by computers have been criticized, and program visualizations have been proposed as a solution. However, their efficacy has been criticized as well, uncovering a need for knowledge that allows engineers to build effective program visualizations.

This study inferred 16 software requirements from a learning theory that may impact educative tools' efficacy. It was found that existing program visualizations do not satisfy these requirements. This research hypothesized that visual concrete allegories and gamification may be used for satisfying these requirements. This new theoretical proposal was called *gamified program visualization*.

This study did not find program visualizations that implement allegories or gamification. Instead, a couple of warnings were detected about allegories' negative effects on other types of user interfaces. We conducted a literature review and an experiment with students of our school, but the warnings were not confirmed. Most studies reported that allegories had the same effect than traditional unconnected metaphors used in our discipline.

The lack of visual allegories and gamification in program visualizations arose the scientific interest for knowing if implementing them may introduce a positive effect on the efficacy of these tools for helping students understand how notional machines work. Two gamified program visualizations were designed for C++ because it is the most used program language by our computer science students. The first design used a puppet theatre as its allegory and it was called Puppeteer++, but an experts' validation revealed that it had more weaknesses than strengths. The second design was named botNeumann++, because it used a futuristic robotized factory as its allegory. It was successfully validated by researchers of Carnegie Mellon University.

Two botNeumann++'s prototypes were implemented in order to collect empirical evidence about their efficacy. The first prototype was built using Microsoft PowerPoint. It showed high efficacy, because approximately 95% of the times that participants discovered a memory leak, they were watching a prototype's animation.

The second prototype was built using C++. Its efficacy was compared experimentally against the tools traditionally used for correcting errors in a programming course. This task was chosen because an understanding of the runtime behavior of a program is mandatory in order to correct its errors. A statistically significant effect of the tool on the participants' efficacy of error detection and correction was found. Participants who used the botNeumann++'s prototype considered it more usable and corrected more errors in the same timespan than participants who continued using the tools that they were used to. This effect was more dramatic for students that failed the course. Therefore, it was concluded that gamified program visualizations were more effective, efficient, and usable than traditional tools. This conclusion supports this thesis' conceptual proposal of using visual allegories and gamification to improve program visualizations.

# Lista de cuadros

Cuadro 2.1. Ejemplos de conceptos de la máquina nocional encontrados en la literatura científica ........ 23

Cuadro 3.1. Criterios de inclusión y exclusión de artículos ........ 63

Cuadro 3.2. Tipos de metáforas visuales usadas en visualizaciones existentes de programa ........ 64

Cuadro 3.3. Visualizaciones de programa activas (parte 1) ........ 65

Cuadro 3.4. Lista de visualizaciones de programa activas (parte 2) ........ 66

Cuadro 3.5. Requerimientos de software de alto nivel ........ 71

Cuadro 3.6. Verificación subjetiva de requerimientos en visualizaciones disponibles de programa ........ 72

Cuadro 3.7. Tareas evaluadas como indicador de rendimiento del participante ........ 97

Cuadro 3.8. Réplicas del experimento que compara metáforas contra alegorías ........ 98

Cuadro 3.9. Análisis de varianza del método de enseñanza ........ 100

Cuadro 3.10. Estadísticas descriptivas de la variable rendimiento ........ 100

Cuadro 4.1. Tipos de visualizaciones de programa de acuerdo al tipo de metáfora y ludificación ........ 111

Cuadro 4.2. Cursos relacionados con la programación que fueron entrevistados ........ 112

Cuadro 4.3. Conceptos de programación representados en la alegoría del teatro de títeres ........ 115

Cuadro 4.4. Valores y variables representados como cápsulas neumáticas ........ 133

Cuadro 5.1. Escala de codificación de severidad de los problemas de usabilidad ........ 171

Cuadro 5.2. Problemas de usabilidad en el prototipo PowerPoint considerados prioritarios de resolver por su número de reportes (R) y severidad (S) ........ 173

Cuadro 5.3. Porcentaje de acuerdo entre pares en la evaluación de usabilidad ........ 176

Cuadro 5.4. Tasas de acuerdo entre grupos de evaluadores ........ 177

Cuadro 5.5. Amenidades del prototipo C++ por número de reportes (R) ........ 201

Cuadro 5.6. Problemas de usabilidad del prototipo C++ reportados por estudiantes de "Introducción a la computación" ordenados por su número de reportes (R) y severidad (S) ........ 202

Cuadro 5.7. Visualizaciones de programa disponibles para C/C++ ........ 211

Cuadro 5.8. Archivos de un ejercicio de programación para el depurador visual ........ 213

Cuadro 5.9. Estructura de la hoja de cálculo usada por los expertos ........ 220

Cuadro 5.10. Escala Likert para calificar las justificaciones y correcciones de los participantes ........ 220

Cuadro 5.11. Análisis de varianza de las correcciones de errores en código fuente ........ 221

Cuadro 5.12. Estadísticas descriptivas de las correcciones de errores en código fuente ........ 221

Cuadro 5.13. Amenidades reportadas de botNeumann++ (bN) y depuradores visuales (DV) .................. 226

Cuadro 5.14. Problemas más severos de usabilidad de las tres herramientas en el experimento........... 227

Cuadro 6.1. Resumen de elementos lúdicos identificados por [Kapp 2012] ....................................................... 247

Cuadro 6.2 Sugerencia de campos para registro de eventos (adaptado de [Chung and Kerr 2012]) ..... 258

# Lista de figuras

Figura 1.1. Modelos mentales de la máquina nocional ........ 3

Figura 1.2. Captura de pantalla de Jeliot 3 ........ 5

Figura 1.3. Metodología adaptada de ciencia del diseño (métodos en itálica, productos en negrita) ........ 11

Figura 1.4. Proceso metodológico distribuido por objetivos (métodos en itálica, productos en negrita) 13

Figura 2.1. Cuerpos teóricos involucrados en la prueba de concepto ........ 20

Figura 2.2. Relación entre máquinas nocionales y la máquina física ........ 22

Figura 2.3. Una visualización es una representación de un fenómeno no visual (elaboración propia) .... 25

Figura 2.4. Esquema general de una visualización por computadora ........ 27

Figura 2.5. Esquema de visualizaciones de software (elaboración propia) ........ 28

Figura 2.6. Principios de aprendizaje del constructivismo sociocultural ........ 33

Figura 2.7. Ejemplo de un puntero cotidiano en una señal de tránsito ........ 38

Figura 2.8. Esquema teórico de la metáfora de acuerdo a Lakoff (elaboración propia) ........ 46

Figura 2.9. Esquema de la metáfora EL AMOR ES UN VIAJE con cinco correspondencias ........ 47

Figura 2.10. Dos ejemplos de metáforas clasificadas por modalidad ........ 50

Figura 2.11. Clasificación propuesta de metáforas por ámbito (elaboración propia) ........ 51

Figura 2.12. Alegoría al café y al banano en el billete de 5 colones (fuente: monedasybilletes.com.ar) .. 56

Figura 3.1. Metodología del objetivo específico 1 ........ 60

Figura 3.2. Captura de pantalla de *JELIOT CONAN*. Fuente: [Moreno et al. 2010, p.146] ........ 68

Figura 3.3. Captura de pantalla de SeeC visualizando el programa de la mediana ........ 69

Figura 3.4. Captura de pantalla de Virtual-C IDE visualizando el programa de la mediana ........ 70

Figura 3.5. Alegoría de un archivador para una máquina *SEQUEL*. Fuente: [Mayer 1981, p.131] ........ 76

Figura 3.6. Elementos seleccionados de la metáfora multimodal compuesta de [Waguespack 1989]: (a) tipos de datos, (b) registro, (c) punteros, (d) lista enlazada ........ 77

Figura 3.7. Las tres alegorías comparadas por [Lin 1989] ........ 78

Figura 3.8. Las tres interfaces comparadas por [Eberts and Bittianda 1993] ........ 79

Figura 3.9. Interfaces comparadas por [Smilowitz 1995] ........ 80

Figura 3.10. Algunas metáforas visuales usadas en los experimentos 1 a 6 por [Blackwell 1998] ........ 82

Figura 3.11. Uno de los cuatro ejemplos usados por [Blackwell 1998] en el experimento 7 ........ 83

Figura 3.12. Las tres interfaces comparadas por [Hsu 2000] ........ 84

Figura 3.13. Alegoría visual de *MAGIC CAP*. Fuente: [Kuniavsky 2010, pp.40-42] ... 87
Figura 3.14. Navegación textual y con alegoría visual usada por [Lee 2007, p.621] ... 89
Figura 3.15. Resumen de comparaciones experimentales entre alegorías y metáforas ... 90
Figura 3.16. Calculadora de Goldbach mostrando las sumas de primos para el número 100 ... 92
Figura 3.17. Una aplicación minimalista con interfaz gráfica congelada ... 93
Figura 3.18. Duración en minutos de cada escena en los dos videos ... 94
Figura 3.19. La aplicación minimalista Tarea Pesada reparada ... 96
Figura 3.20. Gráfico de cajas del rendimiento del grupo control y tratamiento ... 101
Figura 3.21. Distribución de participantes por nota del curso y rendimiento en el experimento ... 102
Figura 3.22. Percepción emotiva de los estudiantes por el método de enseñanza experimentado ... 103
Figura 4.1. Metodología de los objetivos 2 y 3 ... 106
Figura 4.2. Trazabilidad de los 16 requerimientos contra los 12 elementos lúdicos y alegorías ... 108
Figura 4.3. Uso de lenguajes de programación a través de los cursos ... 113
Figura 4.4. Familias de conceptos considerados difíciles pero útiles de aprender ... 114
Figura 4.5. Prototipo en papel de *PUPPETEER++* ... 116
Figura 4.6. Dos titiriteros controlando varias marionetas ... 117
Figura 4.7. Seleccionar una obra. (b) Seleccionar una escena ... 118
Figura 4.8. Familias de conceptos enumeradas en una captura de pantalla de *BOTNEUMANN++* ... 126
Figura 4.9. Representación del ambiente de ejecución de C/C++ en botNeumann++ ... 127
Figura 4.10. Segmento de código en el prototipo C++ de botNeumann++ ... 128
Figura 4.11. El segmento de datos representado como una estantería en botNeumann++ ... 129
Figura 4.12. La entrada estándar representada como un tubo neumático ... 130
Figura 4.13. El error estándar (izquierda) y la salida estándar (derecha) como tubos neumáticos ... 131
Figura 4.14. Un hilo de ejecución activo en botNeumann++ ... 134
Figura 4.15. Una pila de dos invocaciones a función señaladas con flechas ... 136
Figura 4.16. Representación del segmento de memoria dinámica ... 138
Figura 4.17. Valor temporal después de una evaluación de expresión ... 139
Figura 4.18. Metáforas visuales en botNeumann++ enumeradas para el discurso alegórico ... 140
Figura 4.19. Menú del juego de botNeumann++ y un trozo de narración ... 145
Figura 4.20. Mapa de niveles hipotético para una visualización lúdica de C ... 147
Figura 4.21. botNeumann++ como un juez automático ... 150

Figura 4.22. Pantalla de visualización de botNeumann++ usada durante la validación ..... 153

Figura 4.23. Pantalla de visualización después de la validación de botNeumann++ ..... 154

Figura 5.1. Metodología del objetivo 4 ..... 155

Figura 5.2. Menú del juego en el prototipo PowerPoint de botNeumann++ ..... 157

Figura 5.3. Mapa de niveles del prototipo ..... 157

Figura 5.4. Flujo de interacción del prototipo PowerPoint ..... 158

Figura 5.5. Pantalla de nivel 1-1 presentando parte del tutorial ..... 159

Figura 5.6. Tutorial introduciendo el segmento de memoria dinámica en el nivel 1-2 ..... 161

Figura 5.7. Desbordamiento del segmento de memoria dinámica ..... 162

Figura 5.8. Duración promedio de los seis estudiantes en las etapas de la evaluación de usabilidad. Las barras de error indican la desviación estándar. Las "X" indican el número promedio de intentos. ..... 168

Figura 5.9. Cantidad de errores cometidos por profesores y estudiantes en las tres líneas editables ... 170

Figura 5.10. Problemas de usabilidad clasificados por severidad y por la población que los identificó 172

Figura 5.11. Frecuencia de las tasas de acuerdo entre pares en la evaluación de usabilidad ..... 176

Figura 5.12. Correspondencia entre las categorías de problemas y tipos de participantes ..... 178

Figura 5.13. Cuestionario posterior a la evaluación de usabilidad de botNeumann++ ..... 181

Figura 5.14. Rendimiento de los estudiantes detectando fugas de memoria en bloques de código ..... 182

Figura 5.15. Historial de cambios en el desarrollo de prototipo C++ (adaptado de *GitHub*) ..... 190

Figura 5.16. Arquitectura del prototipo C++ de botNeumann++ ..... 191

Figura 5.17. Máquina de estados general del prototipo C++ ..... 192

Figura 5.18. Pantalla de menú en el prototipo C++ ..... 192

Figura 5.19. Pantalla de selección del nivel en el prototipo C++ ..... 193

Figura 5.20. Pantalla de nivel con el ejercicio de conversión de temperaturas ..... 194

Figura 5.21. Reubicación de las ventanas en el prototipo C++ ..... 195

Figura 5.22. Selector con trece casos de prueba ..... 196

Figura 5.23. Cantidad de líneas de código de los módulos en el prototipo C++ ..... 197

Figura 5.24. Cuestionario de evaluación de usabilidad del prototipo C++ ..... 199

Figura 5.25. Duración de la evaluación de usabilidad del prototipo C++ ..... 200

Figura 5.26. Distribución de la usabilidad percibida por los participantes ..... 200

Figura 5.27. Utilidad percibida del prototipo C++ por los participantes ..... 201

Figura 5.28. Diseño de grupo control con post-test únicamente ..... 205

Figura 5.29. Captura de pantalla de una sesión de depuración en *CODE::BLOCKS* 214
Figura 5.30. Captura de pantalla de una sesión de depuración en QtCreator 215
Figura 5.31. Extracto de un ejercicio para revisión de expertos 218
Figura 5.32. Rendimiento por correcciones de los errores en el código fuente 222
Figura 5.33. Distribución de participantes de acuerdo a su nota del curso (eje x) y su eficacia en detección y corrección de errores en el experimento (eje y) 223
Figura 5.34. Duración de los participantes en el experimento por grupo control y tratamiento 224
Figura 5.35. Duración de los participantes en el experimento de acuerdo al profesor 224
Figura 5.36. Comparación de la usabilidad entre el prototipo C++ y depuradores visuales 225
Figura 5.37. Utilidad percibida de la herramienta para comprender la máquina 227
Figura 6.1. Arquitectura de una visualización de software (adaptado de [Lanza 2003, p.3]) 250
Figura 6.2. Arquitectura del prototipo C++ de botNeumann++ 252
Figura 6.3. Historial de cambios en el desarrollo de prototipo C++ (adaptado de *GITHUB*) 253
Figura 6.4. Máquina de estados general del prototipo C++ 255
Figura 6.5. Pantalla de menú en el prototipo C++ 256
Figura 6.6. Administrador de jugadores en el prototipo C++ 256
Figura 6.7. Pantalla de selección del nivel en el prototipo C++ 259
Figura 6.8. Varios niveles agrupados por tubos neumáticos 261
Figura 6.9. Pantalla de nivel con el ejercicio de conversión de temperaturas 263
Figura 6.10. Modelo matemático de distribución de memoria en el prototipo C++ (imagen escaneada, rotar la página para leer) 264
Figura 6.11. Reubicación de las ventanas en el prototipo C++ 266
Figura 6.12. Máquina de estados de la pantalla de nivel en el prototipo C++ 267
Figura 6.13. Proceso de generación de archivos para la animación 269
Figura 6.14. Reporte de un error de compilación durante la generación de archivos 269
Figura 6.15. Ejecución de una instancia del programa del usuario contra un caso de prueba 273
Figura 6.16. Selector con trece casos de prueba 273
Figura 6.17. Programa de conversión de temperaturas en C visualizado por SeeC 275
Figura 6.18. Máquina de estados de la animación 280
Figura 6.19. Problema de conversión de temperaturas resuelto con dos hilos de ejecución 281
Figura 6.20. Controles de la animación y su velocidad 283

# Listados de código

Listado 3.1. Texto de búsqueda de artículos sobre visualizaciones de programa ........ 62

Listado 5.1. Código C++ inicial del ejercicio 1-1 Fahrenheit a Kelvin ........ 206

Listado 5.2. Código C++ inicial del ejercicio 1-2: Porcentaje de incremento ........ 207

Listado 5.3. Código C++ inicial del ejercicio 1-3: ¿Es binario? ........ 208

Listado 5.4. Código C++ inicial del ejercicio 1-4: Sonido de animales ........ 210

Listado 6.1. Ejemplo de una lista de niveles a cargar en el prototipo C++ ........ 260

Listado 6.2. Estructura de un archivo de nivel para el prototipo C++ ........ 262

Listado 6.3. Elemento raíz del ejercicio de conversión de temperaturas ........ 262

Listado 6.4. Contenido de un ejercicio de programación según el DTD ........ 263

Listado 6.5. Descripción de un ejercicio de programación (enunciado del problema) ........ 265

Listado 6.6. Código C++ inicial dado al usuario en el ejercicio de conversión de temperaturas ........ 267

Listado 6.7. Cuatro casos de prueba literales para el ejercicio de conversión de temperaturas ........ 270

Listado 6.8. Un programa generador de casos de prueba ........ 271

Listado 6.9. Un programa en C++ que resuelve el problema de conversión de temperaturas ........ 272

Listado 6.10. Fragmento de interacción entre botNeumann++ y *GDB/MI*. La línea 1 inicia una instancia de *GDB*. Línea 4 indica el ejecutable del estudiante a depurar. Línea 7 re-direcciona los flujos al caso de prueba. Línea 10 establece un *BREAKPOINT*. Línea 13 inicia la ejecución del programa del estudiante... 278

# Convenciones

El texto de esta tesis está estructurado en una jerarquía reflejada en los números de los títulos. Los nombres que se usarán para los cuatro niveles de la jerarquía son: capítulos, secciones, subsecciones y apartados. Este documento incluye vocablos en inglés con el fin de hacer referencia a los términos utilizados en la literatura científica. Con *VERSALES* se resaltan las palabras o siglas en ese idioma. Se utiliza el punto para separar decimales y la coma para separar miles por uniformidad con los lenguajes de programación.

Conceptos que son introducidos por primera vez en el documento se resaltan en **negrita**, con el fin de facilitar su ubicación visual. Los conceptos están acompañados de una definición, y se incluyen en el índice de materias al final del documento. Las referencias a la literatura científica se incluyen en la sección homóloga al final del documento, como es habitual. Las referencias a sitios web se incluyen como notas al pie de página. El estilo de referencia utilizado es el de *ACM JOURNALS*[1]. Si una referencia aparece al final de un párrafo después del punto y aparte, indica que todo el párrafo está sustentado por dicha referencia.

Algunas citas no literales contienen números de página con el fin de facilitar la búsqueda en la fuente de información. Cuando sea posible ubicar el texto referido en una o unas pocas páginas del documento fuente, se incluirá el número de página en la referencia. En caso de ser una idea que se establece en varias o la totalidad de las páginas de la fuente, se omitirá el número de página. Las figuras o cuadros para los que no se indique una fuente, son producto de elaboración propia.

Este documento de tesis se aparta de una estructura "tradicional" en los siguientes aspectos. En la introducción se describe el paradigma y la metodología general seguida en la investigación. En cada capítulo se detallan los aspectos metodológicos y se discuten los resultados. En el capítulo de conclusiones se hace una discusión general. En el capítulo de marco teórico, además de describir las teorías en las que se fundamenta la investigación, se elabora un modelo conceptual propio. En el capítulo sobre antecedentes, además de listar los trabajos previos, se analiza el estado del arte con el fin de pre-evaluar la pertinencia del modelo conceptual propio.

[1] https://www.acm.org/publications/submissions

# 1 INTRODUCCIÓN

La computación se originó por el trabajo de profesionales de otras disciplinas, principalmente de la matemática (algoritmos) y la ingeniería eléctrica (máquinas), que eventualmente producirían máquinas programables. El primer programa de educación universitaria de la nueva disciplina fue el "Diploma en análisis numérico y computación automática" de la Universidad de Cambridge en Reino Unido, abierto en 1953[2]. Era el equivalente a una maestría de un año. Por ser un programa de posgrado, sólo podía admitir profesionales formados en otras disciplinas, y se enfocaba en la programación de computadoras. Nueve años después, en 1962, la Universidad de Purdue abrió su programa de posgrado de ciencias de la computación en Estados Unidos con un proceso de admisión similar[3].

Los planes de pregrado[4] en computación aparecieron en los años setentas en ambas universidades, y se dispersaron como carreras universitarias en la década de los ochenta. Inicialmente eran planes de corta duración, y fundamentalmente orientados a la programación de computadoras. Los cursos introductorios "Programación I" (CS1) y "Programación II" (CS2) heredaron los métodos de enseñanza de los programas de posgrado con una seria consecuencia: a los estudiantes novatos de pregrado en computación se les enseñaba y evaluaba con los métodos pretendidos para estudiantes de posgrado [Sorva 2012]. Esta herencia inició un camino de cuestionamientos sobre la educación formal de la programación. Por ejemplo, tasas de reprobación concentradas entre 20% y 50% de los estudiantes [Caspersen and Bennedsen 2007, pp.33–34; Watson and Li 2014, p.41], y aprobación de dichos cursos por 85% de los estudiantes sin conocimientos básicos de programación a nivel mundial (inferido de las figuras 1 y 2 de [McCracken et al. 2001, p.130]). Estos son indicios de un problema serio, pues la programación se considera una habilidad fundamental en el ejercicio profesional de la disciplina [McCracken et al. 2001; Bennedsen and Caspersen 2007; Watson and Li 2014; Sorva 2012, p.18].

---

[2] "Computación automática" era el término para diferenciar la computación hecha por máquinas, de la "computación" hecha por personas manualmente. Luego el término migró a "ciencias de la computación". Véase http://www.cl.cam.ac.uk/events/EDSAC99/history.html

[3] https://www.cs.purdue.edu/history/history.html

[4] En este documento pregrado refiere a los primeros años para titularse en una carrera universitaria. Corresponde al *UNDERGRADUATE DEGREE* de Estados Unidos.

En más de 50 años de investigación relacionada con la educación de la computación, aún no se ha alcanzado un entendimiento comprensivo que explique por qué unos pocos estudiantes consiguen programar, mientras que la mayoría presenta serias dificultades para lograrlo [Watson et al. 2014; Vihavainen et al. 2014]. Varias líneas de investigación se han establecido para comprender los factores que influyen en el aprendizaje de la computación y la habilidad de programar, como las identificadas por [Fincher and Petre 2004]. Una de estas líneas plantea que la comprensión de cómo funcionan las computadoras es indispensable para poderlas programar, y herramientas de animación, visualización o simulación son necesarias para "hacer visibles" los procesos abstractos que en ellas ocurren. Esta línea de investigación es la que motiva este estudio. A continuación se delimita el problema a investigar.

## 1.1 Definición del problema

El propósito principal de la computación es la resolución de problemas con máquinas computacionales. Para alcanzar este propósito, el profesional en formación debe al menos: (1) aprender a resolver problemas, y (2) comprender cómo trabajan las máquinas computacionales. La resolución de problemas es un tema transversal y recurrente en los cursos de computación. Por su parte, con el fin de que los estudiantes alcancen una correcta comprensión de la máquina computacional, los currículos de pregrado de esta disciplina típicamente incluyen cursos como circuitos digitales, ensambladores, y arquitectura de computadoras. [ACM and IEEE Computer Society 2013]

El propósito de la programación es instruir máquinas computacionales para resolver problemas. Existe un difundido acuerdo en la literatura científica de que una correcta comprensión del funcionamiento de la máquina es requisito para poder programarla [Sorva 2012]. La afirmación anterior genera una paradoja de orden. Para aprender a programar es necesario comprender la máquina computacional, pero los cursos para comprender la máquina computacional son usualmente posteriores a los cursos de programación e incluso los tienen como requisito.

La paradoja de orden se atenúa porque los cursos de programación no utilizan el lenguaje máquina para la solución de problemas, sino lenguajes de un nivel mayor de abstracción. Los

constructos de estos lenguajes abstraen la máquina real y son los que el estudiante emplea directamente para la solución de problemas. Es decir, el programador no percibe la máquina real sino una abstracción de ella que en 1981 Boulay, O'Shea y Monk llamaron **máquina nocional** [Boulay et al. 1981].

Cada lenguaje de programación provee una abstracción de la máquina real. Es decir, existe una máquina nocional por cada lenguaje de programación de segunda generación o superior. Por ejemplo, la máquina nocional de C posee tipos de datos y ejecuta funciones, mientras que la máquina real utiliza datos sin tipo y registros de pila (llamados *EBP* y *ESP* en el ensamblador de Intel 32 bits). Un programador de *JAVA* podría concebir que la máquina computacional es capaz de ejecutar métodos polimórficamente y eliminar objetos en desuso automáticamente (con un recolector de basura) [Sorva 2013]. La parte derecha de la Figura 1.1 representa la relación entre la máquina nocional y la máquina real mediada por el lenguaje de programación.

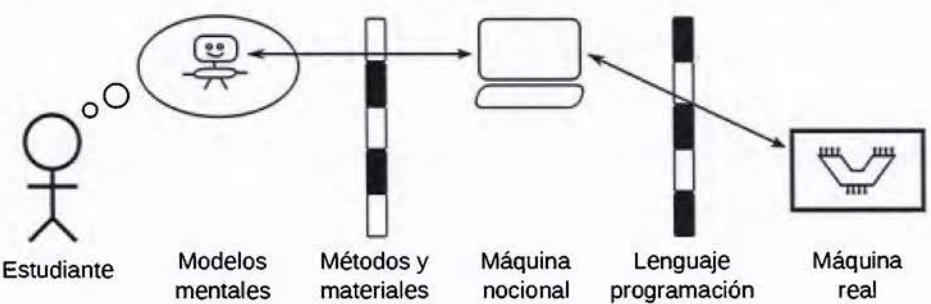



Figura 1.1. Modelos mentales de la máquina nocional

Cuando los estudiantes aprenden a programar en un lenguaje de programación, no se apropian directamente de su máquina nocional, sino que crean modelos mentales de ella [Sorva 2012], como se diagrama en el extremo izquierdo de la Figura 1.1. Los modelos mentales son inter-subjetivos y basados en creencias, por lo que son propensos a imprecisiones [Sorva 2012]. Existe un amplio acuerdo en la literatura científica de que modelos mentales correctos de la máquina nocional son imprescindibles para el aprendizaje de la programación [Sorva 2012; Sorva 2013].

Los estudiantes construyen sus modelos mentales de la máquina nocional a través de los métodos, técnicas y materiales empleados en los cursos de programación [Sorva et al. 2013]. La mitad izquierda de la Figura 1.1 refleja esta relación. A nivel mundial, el método de enseñanza de la máquina nocional reportado como prevalente en los cursos de programación

es la *clase magistral* [Naps et al. 2003; Isohanni and Järvinen 2014]. Una técnica empleada en las clases magistrales para enseñar la máquina nocional de un lenguaje de programación es la que Hertz y Jump llamaron **rastreos de memoria de programa** (en inglés, *PROGRAM MEMORY TRACING*); y consiste en ilustrar la distribución en la memoria de la computadora de los diferentes elementos de un programa y cómo estos cambian mientras el programa se ejecuta [Hertz and Jump 2013]. Para realizar rastreos de memoria de programa durante las clases magistrales, los profesores utilizan materiales como ilustraciones abstractas hechas a mano en la pizarra o en presentaciones proyectadas [Naps et al. 2003; Isohanni and Järvinen 2014; Berry and Kölling 2013, p.25].

La pertinencia de los materiales estáticos se ha cuestionado, en especial por su exigua eficiencia y eficacia para explicar un fenómeno inherentemente dinámico [Sorva et al. 2013, p.4; Hidalgo-Céspedes et al. 2016b]. Se ha propuesto que estas limitaciones pueden superarse con visualizaciones de programas [Sorva 2012]. El objetivo de una *visualización* es ayudar a las personas a construir una representación mental de un fenómeno complejo, abstracto, invisible o intangible. Una **visualización de programa** es un software que representa mediante imágenes el comportamiento en tiempo de ejecución de otros programas (adaptado de [Ebel and Ben-Ari 2006; Sorva et al. 2013; Price et al. 1993]). A modo de ejemplo, la Figura 1.2 muestra una captura de pantalla de Jeliot 3, una de las visualizaciones de programa más estudiadas en la literatura científica. Típicamente estas herramientas muestran el código fuente que el usuario quiere estudiar (rotulado ① en la Figura 1.2), realizan una animación que refleja el efecto de cada instrucción del programa en la máquina nocional (la pestaña de teatro ②), y proveen mecanismos para controlar la animación (③ en la Figura 1.2).

Desde los años 80 han surgido cerca de unas cincuenta visualizaciones de programa con la intención de ayudar a estudiantes a comprender máquinas nocionales [Sorva et al. 2013; Hidalgo-Céspedes et al. 2016a]. Menos de un tercio fueron evaluadas experimentalmente y muy pocas de ellas están diseñadas con fundamento en alguna teoría de aprendizaje [Sorva et al. 2013]. De los resultados reportados en la literatura científica se obtienen los dos siguientes hechos:

1. Las evaluaciones muestran una mezcla de resultados que no permiten concluir científicamente, si las visualizaciones de programa son realmente efectivas para ayudar a

los estudiantes a construir modelos mentales correctos de las máquinas nocionales o no [Sorva et al. 2013].

2. Las visualizaciones de programa son raramente utilizadas en los ambientes de enseñanza-aprendizaje [Naps et al. 2003; Isohanni and Järvinen 2014]. Aunque la mayoría de visualizaciones de programa están ideadas para aprendices, son más utilizadas por profesores en los procesos didácticos que por estudiantes en los procesos de aprendizaje [Isohanni and Järvinen 2014].

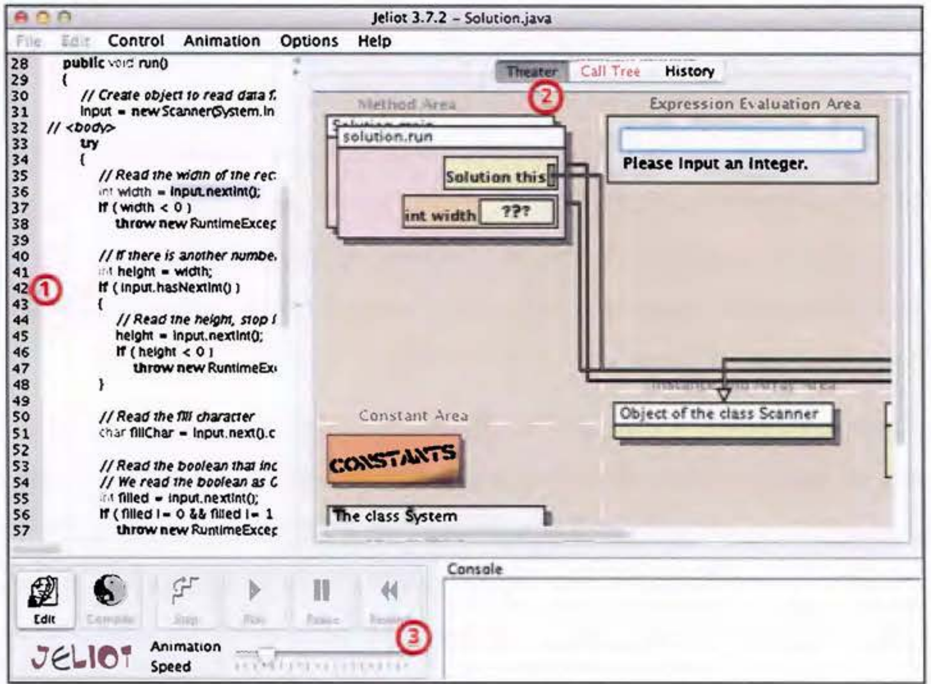



Figura 1.2. Captura de pantalla de Jeliot 3

En el ámbito costarricense, la problemática que motiva a este estudio no ha sido atendida directamente. Los trabajos científicos sobre el aprendizaje de la programación se enfocan en la educación primaria y secundaria, a través del Programa Nacional de Informática Educativa (PRONIE) del Ministerio de Educación Pública y la Fundación Omar Dengo [Badilla Saxe 1991; Alvarez et al. 1998; Coto Chotto and Dirckinck-Holmfeld 2007]. Pocos trabajos se orientan a la educación de la programación a nivel universitario, en temas variados como: enseñanza de la recursividad [Di Mare 1996]; definición curricular de los cursos de programación en cooperación entre la academia y la industria [Casasola 2004]; uso de una herramienta de documentación [Di Mare 2008]; enseñanza de pruebas unitarias en el primer curso de programación [Di Mare 2010a] y de la programación concurrente [Di Mare 2010b]; alternar

las lecciones magistrales con horas de laboratorio [Murillo Rivera 2011]; usar aulas virtuales compartidas en un curso coordinado de introducción a la programación [Coto Chotto and Mora Rivera 2012]; transición de un lenguaje de programación a otro entre cursos de programación [Di Mare 2013a]; diseño curricular de los cursos de programación en torno a aprendizaje basado en problemas y proyectos [Coto Chotto et al. 2012]; evaluar la anuencia de estudiantes a tener un mayor control en aprendizaje basado en proyectos [Coto Chotto et al. 2013]; comparar opiniones de estudiantes sobre: clases magistrales, programación basada en problemas (*PBL*, del inglés *PROBLEM BASED LEARNING*), y *PBL* junto con robots (*LEGO MINDSTORMS*) [Lykke et al. 2014; Lykke et al. 2015]; implementar aprendizaje basado en problemas a lo largo de los cursos de programación de la carrera [Mora Rivera et al. 2014]; entre otros. En el tema de visualizaciones de programa, el único trabajo encontrado es la propuesta de uso de la herramienta existente Jeliot 3 (Figura 1.2) durante las lecciones magistrales, con el fin de acelerar la enseñanza y el aprendizaje de la programación [Di Mare 2013b]. Dicho trabajo se enfoca en el uso de la herramienta y no en investigar cómo superar las limitaciones de las visualizaciones de programa, que es la motivación del presente estudio.

En síntesis, programar es una habilidad fundamental que debe desarrollar un profesional de la computación. Su aprendizaje es considerado complejo e históricamente los cursos de programación han presentado deficiencias en desarrollar esta habilidad [McCracken et al. 2001]. Este es un problema vigente. Una de las líneas de investigación sugeridas por [Fincher and Petre 2004] sostiene que (1) la comprensión de cómo la computadora ejecuta los programas es imprescindible para programar efectivamente [Sorva 2012], (2) objeta los materiales de instrucción tradicional por ser ineficientes [Sorva et al. 2013, p.4], y (3) propone el uso de visualizaciones de programa como una solución [Sorva 2012]. Sin embargo, la efectividad de las visualizaciones de programa es incierta y su uso es raro en los ambientes de enseñanza-aprendizaje [Sorva et al. 2013; Naps et al. 2003; Isohanni and Järvinen 2014].

Esta investigación aporta a la pregunta general sobre ¿cómo crear visualizaciones de programa efectivas para la comprensión de una máquina nocional? A diferencia de las otras investigaciones, esta es original en los requerimientos que atiende y en la propuesta conceptual para satisfacerlos. En este estudio las visualizaciones de programa son analizadas a la luz de una teoría de aprendizaje, la cual se usa para recabar una lista de requerimientos de software de alto nivel. Para satisfacer los requerimientos identificados, esta investigación

propone cambios computacionales conceptuales a las visualizaciones de programa y los evalúa. Estos objetivos se formalizan en la siguiente sección.

## 1.2 Objetivos

El objetivo general de esta investigación es enriquecer computacionalmente las visualizaciones de programa para satisfacer requerimientos de software derivados de una teoría de aprendizaje, y evaluar su efectividad en ayudar a estudiantes a comprender la máquina nocional de un lenguaje de programación. Se entiende por enriquecer el contemplar más elementos de los que las visualizaciones de programa existentes utilizan.

Los objetivos específicos son:

1. Identificar requerimientos de software de alto nivel, a partir de una teoría de aprendizaje, que no son atendidos por las visualizaciones existentes de programa.
2. Proponer cambios conceptuales a las visualizaciones de programa para satisfacer requerimientos de software seleccionados.
3. Diseñar una visualización de programa para un contexto específico que aplique los cambios computacionales propuestos en el objetivo 2.
4. Evaluar experimentalmente la efectividad, para ayudar a estudiantes a comprender una máquina nocional, de un prototipo de la visualización de programa diseñada en el objetivo 3.

La siguiente sección plantea los métodos generales para alcanzar los cuatro objetivos anteriores. Los métodos específicos serán detallados en los capítulos subsecuentes de este documento.

## 1.3 Paradigma de investigación y metodología general

Como método general para alcanzar los objetivos de investigación se siguió una adaptación de la metodología llamada *ciencia del diseño*, también conocida como *ciencia de lo artificial*, porque su objeto de estudio son los objetos y fenómenos creados por seres humanos llamados

**artefactos**, en contraposición a los fenómenos naturales estudiados por las *ciencias naturales*, y a los humanos estudiados por las *ciencias sociales* [Vaishnavi and Kuechler Jr. 2015, p.11]. La **ciencia del diseño** (en inglés, *DESIGN SCIENCE*), es el diseño e investigación de artefactos en un contexto con el fin de mejorarlos y producir conocimiento sobre ellos [Wieringa 2014, pp.3–4]. La ciencia del diseño pretende armonizar los dos intereses que conforman su nombre: (1) el de la ingeniería y otras disciplinas de crear artefactos a través de *diseños* para solucionar problemas; y (2) el de la *ciencia* de producir conocimiento sobre los fenómenos, que en este caso corresponden a los artefactos y su interacción con el contexto [Wieringa 2014, pp.4–6].

Conciliar los ideales de la ingeniería y de la ciencia en una misma investigación no es trivial [Wieringa 2014, p.6]. La ingeniería busca producir un cambio en el mundo al solucionar un problema. La solución resulta de un diseño y hay muchos posibles diseños para un mismo problema. Para evaluar un diseño se debe implementar, y medir su utilidad u otros indicadores, normalmente subjetivos, provenientes de las personas involucradas en el contexto de la solución [Wieringa 2014, p.6; Vaishnavi and Kuechler Jr. 2015, p.21]. De forma diferente, la ciencia busca producir conocimiento sobre el mundo sin cambiarlo. La ciencia positivista plantea preguntas de conocimiento que sólo pueden tener una posible respuesta, y trata de responderlas objetivamente en busca de la verdad, sin dejarse influir por los intereses de las personas involucradas [Wieringa 2014, p.6].

En la ciencia del diseño, los métodos de investigación son multi-paradigmáticos porque deben entrelazar los ideales de la ingeniería y de la ciencia [Vaishnavi and Kuechler Jr. 2015, p.10]. La ciencia del diseño sugiere plantear el objetivo ingenieril como un *problema de diseño* a resolver y el objetivo científico como *preguntas de conocimiento* a responder [Wieringa 2014, p.6].

En esta investigación, las visualizaciones de programa son los *artefactos* de estudio. Siguiendo la nomenclatura sugerida por [Wieringa 2014, p.16], el **objetivo ingenieril** (problema de diseño) de esta investigación es mejorar las visualizaciones de programa, mediante la introducción de otros constructos computacionales que satisfagan requerimientos de software de alto nivel derivados de una teoría de aprendizaje, con el fin de ayudar a estudiantes universitarios de programación a comprender la máquina nocional de un lenguaje de programación.

El **objetivo científico** (pregunta de conocimiento) de esta investigación, es determinar si las visualizaciones de programa que aplican los constructos computacionales propuestos en esta investigación (alegorías visuales y ludificación), son más eficaces que herramientas existentes para ayudar a estudiantes universitarios de programación a comprender la máquina nocional de un lenguaje de programación.

El tratamiento de los objetivos ingenieriles y científicos es distinto. El problema de diseño se resuelve con un ciclo de desarrollo ingenieril. Este ciclo genera como producto un artefacto que al interaccionar con el contexto del problema escogido produce una solución. Por tanto el objetivo ingenieril busca mejorar una situación mediante el diseño o rediseño de un artefacto [Wieringa 2014, p.15; Vaishnavi and Kuechler Jr. 2015, p.18].

Por su parte la pregunta científica se responde con métodos empíricos que requieren recolección de datos sobre el artefacto y su contexto, o métodos analíticos conceptuales tales como demostraciones matemáticas o lógicas [Wieringa 2014, p.17]. En esta investigación se usarán métodos empíricos para responder la pregunta de investigación. El objetivo científico al responder las preguntas de conocimiento es producir conocimiento científico sobre los artefactos y su interacción con el contexto. Las teorías científicas se crean de la generalización del conocimiento obtenido, que sirve para explicar o predecir los fenómenos, y deben sobrevivir en el tiempo a evaluaciones y críticas de pares [Wieringa 2014, p.19,43].

El conocimiento científico varía de acuerdo a la disciplina. Las ciencias básicas, como física y química, tratan de producir generalizaciones universales a partir de condiciones ideales, por ejemplo, despreciando fuerzas de rozamiento o deformaciones de la superficie. En el otro extremo, disciplinas como la medicina e ingeniería tienen que trabajar con condiciones reales, dado que es el contexto donde los artefactos deben interaccionar. Por tanto, se trabaja principalmente a nivel de casos y contextos. Las ciencias del diseño se ubican en un punto medio entre los dos escenarios anteriores, porque pretenden crear generalizaciones a contextos reales similares y no a situaciones universales ideales. [Wieringa 2014, p.9]

Por ejemplo, en caso promedio el algoritmo de búsqueda binaria es más eficiente que la búsqueda lineal, siempre y cuando, los datos estén ordenados y sean de acceso aleatorio. En este estudio, se trata de predecir si un software es usable y eficaz para estudiantes de programación en la Universidad de Costa Rica, sin generalizar a la población mundial por delimitaciones de la investigación.

El procedimiento para responder preguntas de conocimiento es el método científico, mientras que para resolver problemas se sigue un ciclo de desarrollo ingenieril. Varios autores de la ciencia del diseño han propuesto metodologías que entrelazan ambos procedimientos. En 2007, Peffers et al. realizaron una síntesis de las metodologías de ciencia del diseño existentes en la literatura científica [Peffers et al. 2007], que en 2015 fue extendida por [Vaishnavi and Kuechler Jr. 2015, p.19]. En esta tesis se seguirá la segunda con dos adaptaciones. La primera es incorporar la validación de los diseños indicada por Wieringa, además de usar términos más afines al ciclo de desarrollo ingenieril [Wieringa 2014, p.31]. La segunda adaptación es hacer explícito los roles del contexto (relevancia) y del conocimiento científico (rigor), como se propone en [Hevner et al. 2004], además de incorporar los pasos para responder la pregunta de conocimiento de [Wieringa 2014, p.111]. La metodología resultante se diagrama en la Figura 1.3 y se explica a continuación.

Las investigaciones en ciencias del diseño parten de un problema, usualmente en el contexto del usuario, rotulado con ① en la Figura 1.3, que en este caso es la necesidad de herramientas para visualizar programas en los cursos de programación. El investigador realiza varios análisis para poder comprender el problema, encerrados en un recuadro punteado en la Figura 1.3, como un análisis ingenieril que produce una lista de requerimientos para resolver el problema ②.

El ingeniero indaga si existen soluciones al problema que satisfagan los requerimientos. En caso negativo busca conocimiento sobre cómo crear soluciones exitosas para el problema que le atañe ③. Si se detectan vacíos de conocimiento para diseñar soluciones, el ingeniero puede plantear una pregunta de investigación ④.

Para responder la pregunta, el investigador realiza un razonamiento abductivo, que caracteriza la fase creativa del proceso ingenieril, y selecciona o propone conocimientos conceptuales que conjetura podrían servir para diseñar una solución que satisfaga los requerimientos ⑤. El resultado es una hipótesis que podría o no responder la pregunta de investigación [Vaishnavi and Kuechler Jr. 2015, p.18].

Para determinar si su hipótesis responde la pregunta de investigación, el investigador diseña un artefacto que aplica los conocimientos planteados en la hipótesis ⑥. El diseño produce un modelo. Es apremiante tener realimentación sobre el modelo antes de pasar a la fase de

implementación. El modelo se valida con expertos tratando de predecir su utilidad y sus limitaciones para ayudar solucionar el problema a través de métodos de investigación social ⑦. Los resultados de la validación pueden sugerir mejoras en el diseño, o incluso, cambios sustanciales. Estas mejoras se pueden incorporar en el modelo, el cual se puede revalidar, lo que forma el **ciclo de diseño**, rotulado con ⑧ en la Figura 1.3. [Wieringa 2014, p.27,30]

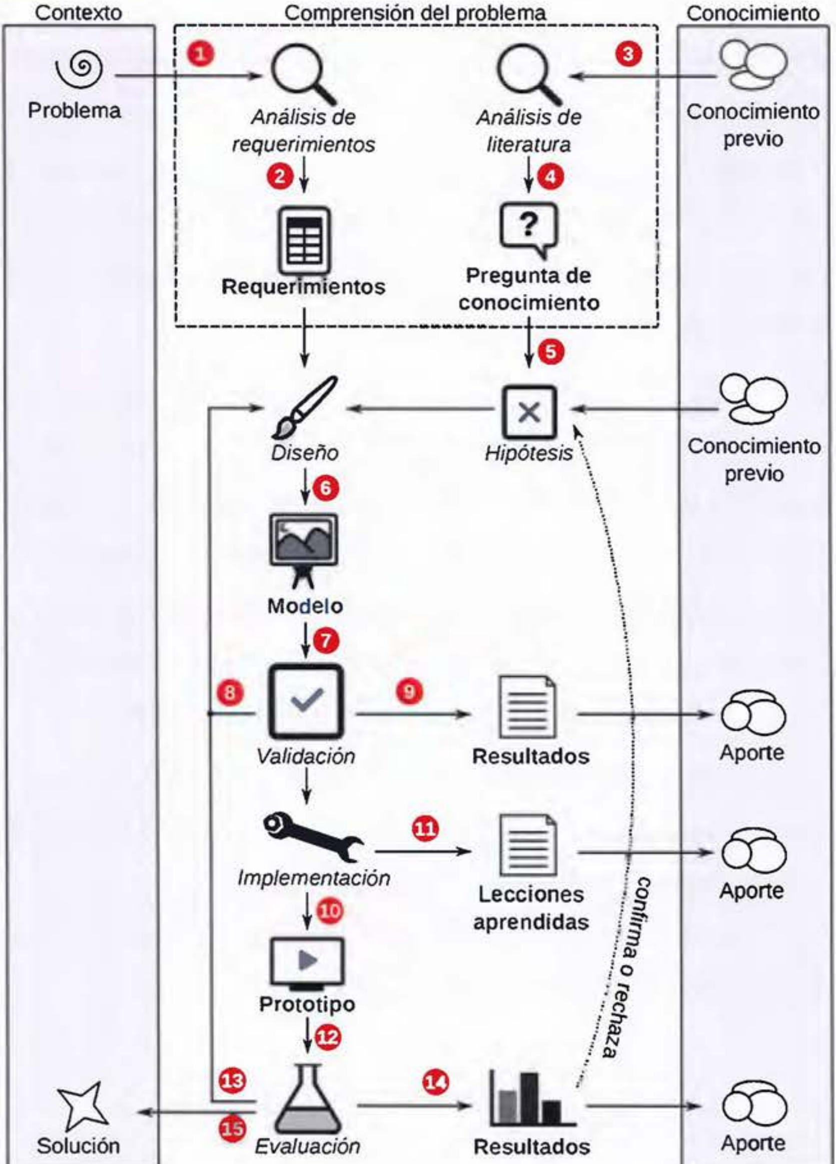



Figura 1.3. Metodología adaptada de ciencia del diseño (métodos en itálica, productos en negrita)

Los resultados de la validación ⑨ conforman el primer aporte a las **teorías de diseño**, las cuales consisten de conocimiento sobre la interacción de un artefacto y su contexto

pretendido [Wieringa 2014, p.62]. Estas teorías sirven para hacer predicciones entre los diseños y su interacción en el contexto. La validación trata de adelantar estas predicciones con opiniones de expertos (grupo focales), o investigación-acción, entre otros métodos. Los resultados de estos métodos aportan al criticar o confirmar las predicciones que pueden hacerse a través de la teoría [Wieringa 2014, p.62].

Una vez que el modelo ha sido validado como exitoso, el ingeniero procede a la fase de implementación, la cual produce un artefacto usable en el contexto ⑩. Del proceso de implementación surgen lecciones aprendidas que también son útiles para otros ingenieros ⑪, y en caso de poderse implementar un artefacto conforma en sí una prueba de factibilidad del modelo [Hevner et al. 2004, p.79; Vaishnavi and Kuechler Jr. 2015, p.23]. Para efectos de investigación, no es necesario tener un producto terminado, un prototipo del modelo es suficiente para poder evaluarlo en su contexto.

El ingeniero evalúa el prototipo en el contexto pretendido con sus usuarios potenciales, a través de métodos empíricos ⑫. Los resultados de la evaluación podrían sugerir mejoras, tanto al prototipo como al modelo, lo que provoca un ciclo al que se llamará **ciclo de evaluación** en esta tesis ⑬. En estas evaluaciones, el ingeniero puede comparar la utilidad del artefacto creado contra los artefactos disponibles en el contexto pretendido. Los resultados de las comparaciones permitirán al ingeniero determinar si su hipótesis responde a la pregunta de investigación o no, y por tanto, si alcanzó su objetivo ingenieril.

La Figura 1.4 muestra cómo la metodología de ciencia del diseño se adaptó para responder cada objetivo específico de esta investigación. Los objetivos se representan como franjas horizontales separadas por líneas punteadas y se identifican por el producto principal de cada uno de ellos en la parte izquierda de la Figura 1.4. En las subsecciones siguientes se explican los métodos seguidos para realizar cada objetivo.

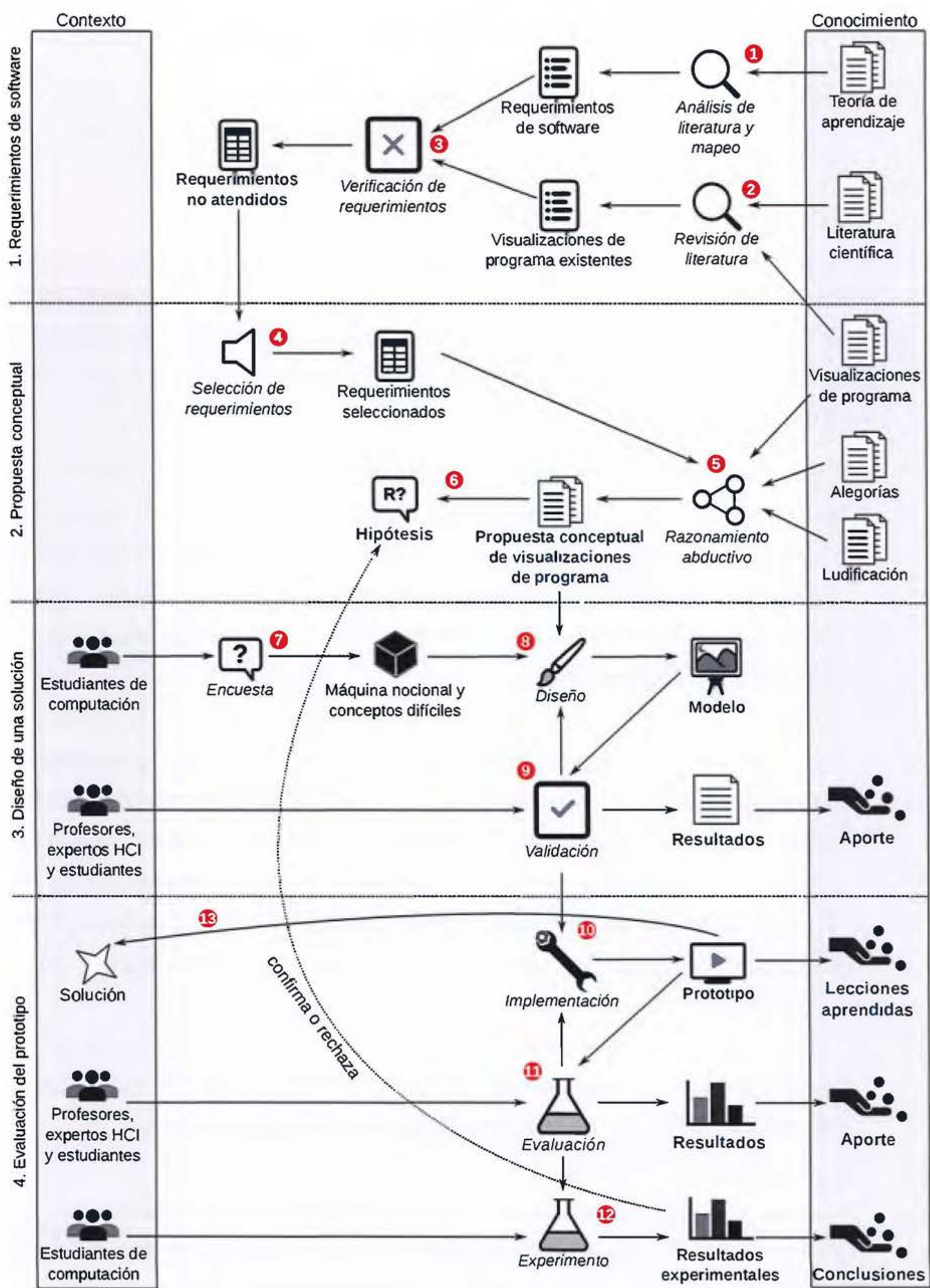



Figura 1.4. Proceso metodológico distribuido por objetivos (métodos en itálica, productos en negrita)

### 1.3.1 Requerimientos de software

Las investigaciones en ciencias del diseño parten de un problema ingenieril. El problema atendido en esta tesis es la necesidad de conocimiento que permita construir herramientas de visualización de programas *eficaces* para estudiantes de programación. Dado que las visualizaciones de programa son herramientas de aprendizaje, deberían aplicar las recomendaciones que una teoría de aprendizaje sugiera sobre la mediación de herramientas. Por lo tanto, el objetivo 1 pretende "Identificar requerimientos de software de alto nivel, a partir de una teoría de aprendizaje, que no son atendidos por las visualizaciones existentes de programa". Se realizó mediante los métodos y actividades resumidos en la siguiente lista. Los números en círculos corresponden a los rotulados en la Figura 1.4.

- *Análisis de literatura de la teoría de aprendizaje* ①. En la sección 2.3 del Marco teórico se discuten las cuatro teorías de aprendizaje existentes y se escogió una de ellas, el Constructivismo sociocultural, siguiendo los argumentos de [Greening 2000]. Mediante un mapeo de las sugerencias que esta teoría plantea sobre la mediación docente, se obtuvo una lista de 16 *requerimientos de software* de alto nivel que las visualizaciones de programa deberían satisfacer.
- *Revisión sistemática de literatura de visualizaciones de programa* ②. Para determinar en qué medida los requerimientos de software han sido satisfechos por las visualizaciones de programa existentes, se debe recabar la lista de estas herramientas. La única revisión de literatura exhaustiva afín es [Sorva et al. 2013], la cual encontró 46 herramientas en el período 1979 a 2012. En la sección 3.1 (Visualizaciones de programa existentes), el autor de esta tesis extendió dicha revisión al período entre 2013 y 2016. Se encontraron 7 nuevas herramientas. El resultado fue una lista de 53 visualizaciones de programa existentes.
- *Verificación de requerimientos en visualizaciones de programa existentes* ③. Se pretende conocer el nivel de cumplimiento de los requerimientos por parte de las visualizaciones de programa existentes. Para poder evaluar los requerimientos se filtraron las visualizaciones de programa disponibles, es decir, aquellas que pudieron ser descargadas y ejecutadas en una computadora. Se calificó subjetivamente en qué grado cada visualización de programa existente satisface cada uno de los 16 requerimientos. Se encontró que sólo dos requerimientos son atendidos medianamente por las

visualizaciones de programa existentes. Este hallazgo es el principal producto y conclusión del objetivo 1, que además justifica la creación de una propuesta conceptual para llenar el vacío de conocimiento.

### 1.3.2 Propuesta conceptual

Dado que la mayoría de requerimientos de software identificados no son satisfechos por las visualizaciones de programa disponibles, el objetivo 2 de esta investigación pretende "Proponer cambios conceptuales a las visualizaciones de programa para satisfacer requerimientos de software seleccionados". Este objetivo procura enriquecer conceptualmente las visualizaciones tradicionales de programa al aplicar otras teorías computacionales mediante un razonamiento abductivo. Los siguientes fueron los pasos realizados, cuya numeración en círculos proviene de la Figura 1.4.

- *Selección de los requerimientos* ④. Se seleccionaron los requerimientos de software a ser atendidos por la propuesta conceptual. Este es un paso heredado de la propuesta de tesis. Sin embargo, durante el desarrollo de la investigación se encontró que todos los 16 requerimientos pueden ser satisfechos con las dos teorías escogidas en el siguiente punto.
- *Creación de la propuesta conceptual* ⑤. Se realizó un razonamiento abductivo que caracteriza la fase creativa del proceso ingenieril [Vaishnavi and Kuechler Jr. 2015, p.18]. Se conjeturó que los requerimientos de software derivados de la teoría de aprendizaje pueden ser satisfechos si las visualizaciones de programa implementan alegorías visuales y ludificación. Las alegorías son una forma de combinar metáforas coherentemente y se presentan en la sección 2.4 (Metáforas y alegorías) del marco teórico. La ludificación consiste en aplicar elementos de juego a situaciones que no son juegos y se resumen en la sección 2.5 (Ludificación), también del marco teórico. La forma en que las visualizaciones de programa pueden usar alegorías y ludificación corresponde a la propuesta conceptual que se desarrolla en la sección 4.1 (Propuesta conceptual) y es el principal producto del objetivo 2. Durante el estudio de las teorías de metáforas se encontró en el campo de interacción humano-computador advertencias críticas de las alegorías, por lo que se hizo una revisión de literatura de las alegorías en computación (sección 3.4) y una comparación experimental entre metáforas y alegorías (sección 3.5).

- *Hipótesis de investigación* ⑥. Dado que ninguna visualización de programa existente implementa alegorías visuales ni ludificación, naturalmente surge la pregunta de conocimiento ¿serán las visualizaciones lúdico-alegóricas de programa más eficaces que herramientas tradicionales para comprender la máquina nocional de un lenguaje de programación? La hipótesis de acuerdo a la teoría de aprendizaje es que lo son. Sin embargo, se requieren datos empíricos para poder confirmar o rechazar esta hipótesis. Para obtener estos datos se diseñó e implementó una visualización lúdico-alegórica de programa.

### 1.3.3 Diseño de una solución

El objetivo 3 de esta investigación es "Diseñar una visualización de programa para un contexto específico que aplique los cambios computacionales propuestos en el objetivo 2". Se realizó mediante las siguientes actividades y métodos (los números en círculos señalan los de la Figura 1.4):

- *Selección de una máquina nocional* ⑦. El contexto de uso de la visualización de programa son los cursos de programación en la ECCI, y los estudiantes de estos cursos son sus usuarios finales. Mediante una encuesta se encontró que C++ (máquina nocional) es el lenguaje de programación más usado por los estudiantes durante la carrera, y que los conceptos de concurrencia y administración de memoria fueron considerados como los más difíciles y relevantes de comprender. Los detalles y resultados de la encuesta se presentan en la sección 4.2.
- *Diseño de una visualización lúdico-alegórica de programa para C++* ⑧. Este paso y el siguiente forman parte del ciclo de diseño. Se iteró este ciclo dos veces durante el desarrollo de la tesis. El primer diseño realizado utilizó una alegoría de un teatro de títeres para la máquina nocional de C++. Elementos de ludificación fueron aplicados, similares a los de un videojuego casual, como se documenta en la sección 4.3. El producto fue un modelo al que se le llamó *PUPPETEER++*.
- *Validación del modelo* ⑨. Se realizaron grupos focales con profesores de programación de la ECCI para obtener opinión experta. Además de algunas fortalezas, los expertos reportaron varias limitaciones en el modelo del teatro de títeres, para representar algunas

combinaciones de conceptos de C++. Los detalles del grupo focal y sus resultados se presentan en la subsección 4.3.3.

- *Segunda iteración del ciclo de diseño* ⑧⑨. Se diseñó una segunda visualización de programa usando robots en una fábrica como alegoría visual, y se aplicó ludificación de forma similar al modelo previo inspirada en un videojuego casual. Al modelo resultante se le llamó botNeumann++ y fue validado mediante entrevistas con expertos de *Carnegie Mellon University*, como se detalla en la sección 4.4. Los resultados fueron muy exitosos y provocaron mejoras en el diseño.

Los resultados de la validación de ambos diseños conforman un aporte a las teorías de diseño de visualizaciones de programa. Dado el éxito del segundo modelo se procedió a implementar un prototipo para evaluar su usabilidad y eficacia experimentalmente.

### 1.3.4 Evaluación de un prototipo

El objetivo 4 de esta investigación es "Evaluar experimentalmente la efectividad, para ayudar a estudiantes a comprender una máquina nocional, de un prototipo de la visualización de programa diseñada en el objetivo 3". Se realizó mediante las siguientes actividades (los números en círculos señalan los de la Figura 1.4):

- *Implementación de un prototipo* ⑩. Este paso y el siguiente forman parte del ciclo de evaluación. Se implementó un primer prototipo de un subconjunto del modelo diseñado en el objetivo anterior, usando animaciones de *Microsoft PowerPoint*. La creación del prototipo en sí es una prueba de factibilidad, y junto con las lecciones aprendidas en la creación del mismo conforman otro aporte a la teoría del diseño de visualizaciones de programa. Estos aportes se documentan en la sección 5.1 (Prototipo 1: PowerPoint).
- *Evaluación preliminar del prototipo* ⑪. El prototipo de *PowerPoint* fue evaluado por usabilidad con tres poblaciones relevantes en su contexto: profesores de programación, expertos en interacción humano-computador, y estudiantes de programación. Dado que las animaciones de *PowerPoint* no implementan un compilador o intérprete de C++, se usó un protocolo de mago de oz, donde el autor de esta tesis actuó como intérprete de código C++. Los resultados de la evaluación de usabilidad recabaron una lista considerable de mejoras que fueron incluidas en el diseño y en el segundo prototipo, implementado en

C++. Muchas lecciones aprendidas surgieron de esta etapa, en especial por la carencia de herramientas modernas para extraer información detallada de programas C++ en ejecución. Este segundo prototipo también fue evaluado por usabilidad con resultados satisfactorios. Las lecciones aprendidas y los resultados de la evaluación se incluyen en la sección 5.2 (Prototipo 2: C++).

- *Evaluación experimental del prototipo* ⑫. La pregunta de investigación es determinar si las visualizaciones lúdico-alegóricas de programa son más eficaces para ayudar a comprender una máquina nocional que herramientas tradicionales. Para responderla se comparó mediante un experimento la eficacia del prototipo (tratamiento) contra herramientas tradicionales para comprender el estado de un programa (control). Participaron estudiantes del curso "Programación II". La variable respuesta fue la eficacia de los participantes en detectar y corregir errores sutiles en programas de C++ utilizando las herramientas evaluadas, dado que la comprensión de los programas es indispensable para poder corregirlos. Los resultados encontraron un efecto significativo de la herramienta en la eficacia de detección y corrección de errores. Los estudiantes que usaron el prototipo corrigieron satisfactoriamente más errores que aquellos que usaron herramientas tradicionales. Este resultado confirma la hipótesis de investigación, lo que apoya la propuesta conceptual de que las teorías escogidas ayudan a la construcción de visualizaciones de programa más eficaces. La evaluación experimental fue complementada con una evaluación de usabilidad y métodos cualitativos. Los detalles del experimento y sus resultados se presentan en la sección 5.3.
- *Aporte ingenieril* ⑬. La ingeniería construye artefactos cuya interacción en un contexto puede ayudar en la solución de problemas. El prototipo de C++ corresponde a un aporte ingenieril, que puede ser extendido y usado en el contexto de la educación de la programación, y futuras investigaciones como se sugiere en la sección 6.3 (Trabajo futuro)[5].

---

[5] botNeumann++ es software libre y su código fuente se encuentra bajo control de versiones en https://github.com/citic/botNeumann

## 1.4 Alcance y delimitaciones

El alcance de esta investigación es *correlacional* de acuerdo a la clasificación de [Hernández Sampieri et al. 2010]. Se pretende comparar el efecto de visualizaciones de programa enriquecidas en la comprensión de la máquina nocional contra las herramientas tradicionalmente empleadas para este fin.

Douglas E. Comer[6] indica que la mayoría de evaluaciones experimentales en tesis doctorales de ciencias de la computación pueden clasificarse en dos categorías: de optimización y de prueba de concepto. Esta tesis cabe en la segunda categoría, puesto que efectúa una prueba de concepto que verifica científicamente si las visualizaciones de programa que atienden principios de aprendizaje constructivista son o no efectivas para la comprensión de la máquina nocional de un lenguaje de programación. Para la prueba de concepto no se pretende visualizar una máquina nocional completa, por ejemplo, conceptos como archivos o polimorfismo se tomaron como trabajo futuro.

El próximo capítulo (Marco teórico) presenta formalmente el conocimiento que esta investigación aplica para construir la propuesta conceptual con el fin de enriquecer las visualizaciones de programa.

---

[6] Douglas E. Comer, "*HOW TO WRITE A PH.D. DISSERTATION*", en https://www.cs.purdue.edu/homes/dec/essay.dissertation.html

# 2 MARCO TEÓRICO Y MODELO CONCEPTUAL

En este capítulo se presentan las teorías en que se fundamenta la investigación y se propone un modelo conceptual compuesto de requerimientos que una herramienta de aprendizaje debería satisfacer y una taxonomía de metáforas en computación. Para programar adecuadamente en un lenguaje particular, el estudiante debe comprender cómo trabaja su máquina nocional (sección 2.1). Para un aprendiz, la máquina nocional que debe programar es una "caja negra o mágica" [Boulay et al. 1981] como se esquematiza en la parte superior de la Figura 2.1. Métodos y materiales de instrucción estáticos se han utilizado tradicionalmente para explicar la máquina nocional [Naps et al. 2003; Isohanni and Järvinen 2014; Berry and Kölling 2013, p.25], con el fin de convertirla en una "caja de cristal", que permita al estudiante observar y comprender su funcionamiento interno [Boulay et al. 1981]. Los métodos y materiales estáticos son cuestionados por no ser eficientes ni eficaces para explicar un fenómeno dinámico [Sorva 2012; Hidalgo-Céspedes et al. 2016b], y las visualizaciones de programa se han propuesto como una solución [Sorva 2012; Sorva et al. 2013].

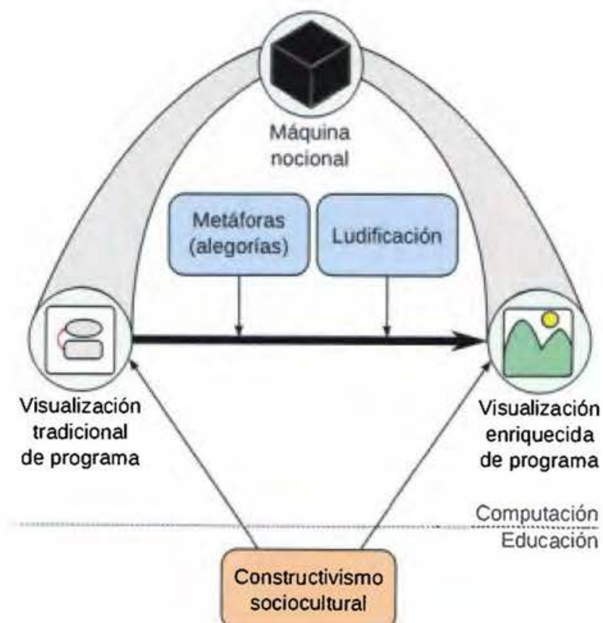



Figura 2.1. Cuerpos teóricos involucrados en la prueba de concepto

Sin embargo, las pocas evaluaciones experimentales de visualizaciones de programa existentes reportan resultados fluctuantes de efectividad [Sorva et al. 2013], y un uso esporádico en los ambientes de educación de la programación [Naps et al. 2003; Isohanni and

Järvinen 2014] que evidencia la necesidad de conocimiento de cómo crear visualizaciones de programa eficaces. En respuesta, esta investigación sugiere mejoras conceptuales a estas herramientas y evalúa su efectividad para la comprensión de la máquina nocional. Como se esquematiza en la Figura 2.1, los objetos de estudio de esta investigación son las visualizaciones de programa, presentadas en la sección 2.2. En la sección 2.3 se presenta el constructivismo sociocultural, por ser una teoría de aprendizaje recomendada para el campo de la computación [Greening 2000], y de ella se infieren, como parte del modelo conceptual, requerimientos que podrían influir en la eficacia de las herramientas de aprendizaje. En la sección 2.4 se analizan las teorías de metáforas, y como parte del modelo conceptual se propone una taxonomía de metáforas para computación, que incluye alegorías. El autor conjetura que las alegorías y la ludificación (sección 2.5) permiten satisfacer los requerimientos de software en la propuesta conceptual de la sección 4.1.

## 2.1 Máquinas nocionales

Las computadoras son máquinas programables. En las primeras computadoras, los programas se ingresaban directamente en el hardware, con dispositivos como interruptores o tarjetas perforadas. El hardware es programado a través del *lenguaje máquina*, compuesto de los símbolos cero y uno. El extremo derecho de la Figura 2.2 representa la programación directa en el lenguaje máquina. Se requiere una comprensión de cómo trabaja la máquina física para programarla con estos símbolos.

En la actualidad la programación de la máquina se hace a través de lenguajes intermedios, de bajo nivel (como ensamblador) y de alto nivel (como *JAVA* y C/C++)[7]. Programas escritos en estos lenguajes no son ejecutados directamente por el hardware, sino que deben ser traducidos por compiladores o intérpretes al lenguaje máquina. Los lenguajes de programación aíslan al programador de algunos conceptos del hardware (como los códigos de instrucción) y proveen constructos de programación que no existen en el hardware (como los objetos y funciones). Este aislamiento se representa con marcos sombreados en la Figura 2.2.

---

[7] Una lista de popularidad de lenguajes de programación puede consultarse en http://www.tiobe.com/index.php/tiobe_index

El programador soluciona un problema utilizando los constructos o nociones provistas por el lenguaje de programación, en lugar del hardware directamente. Al unir los constructos provistos por un lenguaje de programación se tiene una máquina que no existe físicamente, sino que es conceptual, a la que [Boulay et al. 1981] llamaron **máquina nocional** y [Blackwell 1996] llamó *máquina virtual.*

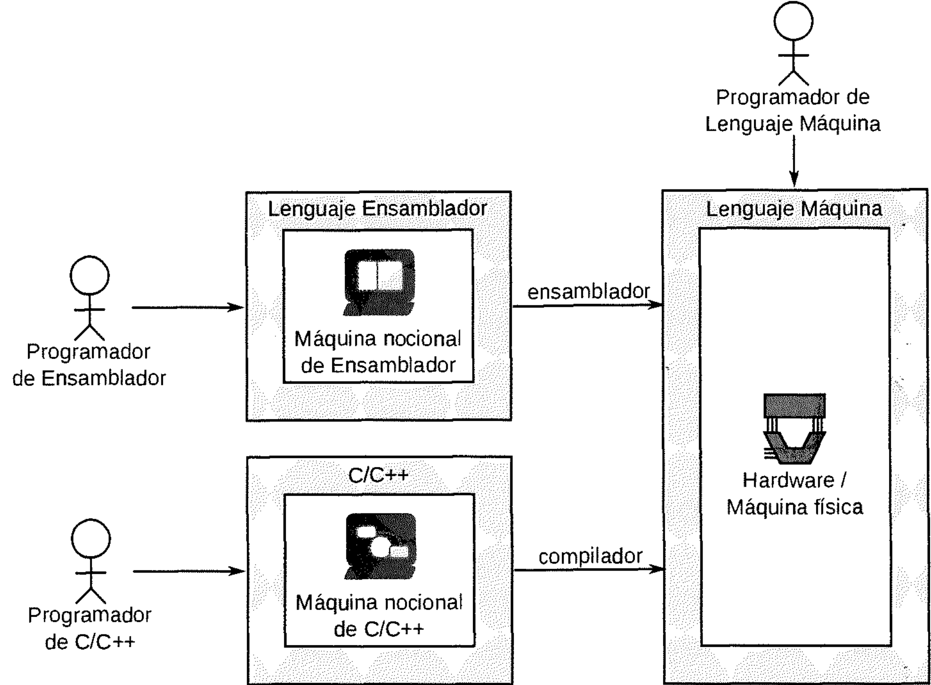



Figura 2.2. Relación entre máquinas nocionales y la máquina física

Cada lenguaje de programación está diseñado para programar una máquina computacional. El lenguaje máquina permite programar la máquina física, mientras que todos los demás lenguajes permiten programar una máquina nocional. La parte izquierda de la Figura 2.2 incluye dos ejemplos de máquinas nocionales: la de ensamblador y la de C/C++. En [Boulay et al. 1981] se utiliza el nombre del lenguaje de programación para identificar las máquinas nocionales, por ejemplo: "la máquina de C/C++" (en inglés, *THE C/C++ MACHINE*). En este documento se explicita la naturaleza nocional incluyendo dicho término, como en "la máquina *nocional* de C/C++".

Para [Sorva 2013, p.8:3], un lenguaje de programación puede tener varias máquinas nocionales, por ejemplo, una máquina nocional mono-hilada (en inglés, *SINGLE-THREADED*) de *JAVA* y una máquina nocional multi-hilada (en inglés, *MULTI-THREADED*) de *JAVA*. En este documento se prefiere mantener una relación uno-a-uno entre el lenguaje de programación y

su máquina nocional, la cual abarca todas las nociones presentes en el lenguaje. Dado que la máquina nocional es conceptual, no está limitada a detalles de implementación de compiladores o sistemas operativos. Si diferentes niveles de detalle de un lenguaje de programación definieran distintas máquinas nocionales, el número de ellas podría ser inmanejable para un lenguaje.

Pese a que el término de máquina nocional fue introducido en 1981, muy poco se ha utilizado y discutido en la literatura científica. Una duda conceptual que queda abierta es si todos los conceptos del lenguaje programación forman parte de una máquina nocional o sólo aquellos que intervienen en tiempo de ejecución del programa. Por ejemplo, ¿son las instrucciones para el preprocesador y las plantillas parte de la máquina nocional de C/C++? Las publicaciones sobre máquinas nocionales no atienden esta discusión directamente, pero incluyen ejemplos de conceptos de la máquina nocional, como los listados en el Cuadro 2.1.

Cuadro 2.1. Ejemplos de conceptos de la máquina nocional encontrados en la literatura científica

| Máquina nocional | Conceptos | Publicación |
|---|---|---|
| *Prolog* | Desbordamiento de pila, mecanismo de búsqueda de patrones (*pattern-matching mechanism*), control de flujo condicional | [Boulay et al. 1981] |
| *LOGO* | Comandos, argumentos, procedimientos almacenados, invocación de sub-procedimientos, recursión | [Boulay et al. 1981, p.247] |
| genérica | Constructos de ciclos, instrucciones condicionales, llamados a procedimientos, espacio de memoria, creación de variables, asignación de variables, apuntadores | [Ramadhan 1992, p.150; Ramadhan 2001, p.86] |
| *DISCOVER* | Declaración de variable, asignación de variable, impresión o salida (en pantalla), lectura de un valor (del teclado), instrucción condicional (*if-else*), iteración (*while*) | [Ramadhan 1992, pp.153–154; Ramadhan 2001, p.88] |
| *QBasic* | Variables, asignaciones de variables, expresiones aritméticas, condicionales, escritura en la salida, lectura de la entrada | [Dadic et al. 2008, p.486] |
| *Python* | Objeto, clase, referencias/apuntadores, invocación a funciones (no de biblioteca), paso de parámetros, evaluación de expresiones, estructuras de control, variables, recursión | [Sorva and Sirkiä 2010, pp.1–2] |
| *Java* | Objetos, estado de los objetos, clases, métodos, invocación de métodos, referencias, variables, tipo de valor, asignación, argumentos, recursión | [Sorva 2008, p.5; Berry and Kölling 2013, p.26] |

Todos los conceptos de la máquina nocional listados en el Cuadro 2.1 tienen efecto en tiempo de ejecución de los programas. Dado que tanto la máquina nocional como la máquina física son máquinas computacionales, ambas comparten la característica de ejecutar código. En publicaciones recientes, se ha tomado esta característica como el propósito exclusivo de las máquinas nocionales, es decir, explicar el comportamiento de los programas en tiempo de

ejecución [Dadic et al. 2008, p.486; Sorva et al. 2013]. Sin embargo, este documento se apega a la definición original de [Boulay et al. 1981] que no excluye constructos del lenguaje de programación que tienen efecto sólo en tiempo de compilación, como el caso del preprocesador de C y las plantillas de C++.

Para que una máquina nocional sea efectiva para el aprendizaje, debe cumplir dos principios: simplicidad y visibilidad [Boulay et al. 1981, p.237]. El **principio de simplicidad** establece que máquinas nocionales simples son más fáciles de comprender por los aprendices. El principio de simplicidad se subdivide en tres [Boulay et al. 1981, p.238]:

1. *Simplicidad funcional*: Cada instrucción del lenguaje de programación debe poderse explicar con pocas "transacciones" de la máquina nocional.
2. *Simplicidad lógica*: Los problemas de interés para el aprendiz deben poderse resolver con programas sencillos.
3. *Simplicidad sintáctica*: Las reglas para escribir instrucciones deben ser uniformes y utilizar nombres o símbolos intuitivos, de tal forma que el estudiante pocas veces tenga que recurrir a la memorización.

Las máquinas nocionales dependen más del lenguaje de programación que del hardware [Boulay et al. 1981, p.237]. El diseño de una máquina nocional para el aprendizaje de la programación afecta el diseño del lenguaje de programación. Sin embargo, es común que en los cursos introductorios de programación actuales se utilicen lenguajes de programación complejos, como *JAVA* y C/C++ [Schulte and Bennedsen 2006, p.20], diseñados con prioridades definidas por la industria y distintas a las didácticas. Estas elecciones impiden aplicar el principio de simplicidad de [Boulay et al. 1981]. Sin embargo, el principio de visibilidad aún puede atenderse.

El **principio de visibilidad** recomienda proveer métodos al estudiante para que pueda observar parte del funcionamiento de la máquina nocional en tiempo de ejecución [Boulay et al. 1981, p.237]. El objetivo del principio de visibilidad es procurar que la máquina nocional deje de ser una "caja negra o mágica" para el estudiante, y se convierta en una "caja de cristal", que permita observar y comprender su funcionamiento interno [Boulay et al. 1981]. Las visualizaciones de programa son un ejemplo de herramientas para ofrecer visibilidad de máquinas nocionales, y se analizan en la siguiente sección.

## 2.2 Visualizaciones

En la literatura científica, las visualizaciones de programa se tienden a presentar en relación con otros tipos de visualizaciones a través de taxonomías. Varias taxonomías han sido propuestas, como [Myers 1990], [Stasko and Patterson 1992], [Roman and Cox 1992], [Price et al. 1993], [Hundhausen et al. 2002], [Maletic et al. 2002], [Naps et al. 2003], [Kelleher and Pausch 2005], [Lahtinen et al. 2007], y [Sorva et al. 2013]. Esta sección adapta la taxonomía de [Sorva et al. 2013] y la extiende para incluir visualizaciones en general y visualizaciones por computadora.

Para propósitos de este documento, una **visualización** (en inglés, *VISUALIZATION*) es una representación mediante imágenes de fenómenos de otra naturaleza[8]. Un esquema general de una visualización se muestra en la Figura 2.3. Las visualizaciones se elaboran con el objetivo de ayudar a las personas a construir una representación mental de un fenómeno abstracto o complejo. Por ejemplo, mapas que ilustran la incidencia de una epidemia ayudan a las personas a comprender su expansión geográfica, el cual es un fenómeno no directamente visible. Otros ejemplos de visualizaciones son: la representación en pantalla de la información capturada por un radar, el ecualizador gráfico de una señal reproducida en un equipo de audio, y el análisis deportivo de un encuentro concluido utilizando guías visuales. Por otro lado, una fotografía o un video filmado no se consideran visualizaciones por ser el registro directo de un fenómeno que ya es visual.

Figura 2.3. Una visualización es una representación de un fenómeno no visual (elaboración propia)

Las visualizaciones no representan la totalidad del fenómeno en estudio, sino una parte de él. El diseñador de una visualización realiza una selección de características a representar del

[8] Adaptado de la 23ª edición del Diccionario de la lengua española de la Real Academia Española disponible en http://www.rae.es/.

fenómeno. Por ejemplo, una gráfica de un terremoto generada por un sismógrafo representa la intensidad de los movimientos terrestres cerca del epicentro, pero no representa una reestructuración de placas o los daños materiales de una comunidad. Una **visualización especializada** representa unas pocas características del fenómeno, con el fin de aislarlas y hacerlas claras al observador. Una **visualización genérica** representa muchas características con el fin de proveer una imagen comprensiva del fenómeno (adaptado de [Sorva et al. 2013, p.7]).

Las visualizaciones se pueden clasificar por la actualización en el tiempo. Una **visualización estática** (en inglés, *STATIC VISUALIZATION*) representa el fenómeno de estudio con imágenes que no cambian en el tiempo, por ejemplo, un dibujo hecho a mano en una pizarra del estado de un programa corriendo en una computadora, o un mapa geográfico generado por computadora que grafica la ubicación de los visitantes de un sitio web. Una **visualización dinámica** (en inglés, *DYNAMIC VISUALIZATION*) representa el fenómeno de estudio con imágenes que se actualizan en el tiempo.

Las visualizaciones se pueden también clasificar por el ente que realiza o produce la visualización. Una **visualización manual** es realizada por una persona y usualmente es una ilustración del fenómeno en estudio. Una **visualización automática** es realizada por equipo mecánico (por ejemplo, el sismógrafo en sus inicios), electrónico (por ejemplo, un osciloscopio), o informático (y por tanto, un software). Cuando una visualización automática es hecha por software se le llama *visualización por computadora* y se analiza a continuación.

### 2.2.1 Visualizaciones por computadora

Una **visualización por computadora** (en inglés, *COMPUTER VISUALIZATION*) es una visualización cuyas imágenes son generadas por un software en una computadora. A un nivel técnico, una visualización por computadora es un software que toma datos sobre un fenómeno, los procesa, y genera imágenes que despliega a un usuario a través de un dispositivo de salida. El usuario no está limitado a sólo observar, puede tener un nivel mayor de interacción, como controlar el origen de los datos y la forma en que estos son desplegados. Los componentes de una visualización por computadora se rotulan en la parte inferior de la Figura 2.4 y los resaltados en negrita se discuten en la siguiente lista.

- *Tipo de interacción*. Si el usuario puede interaccionar con la visualización para escoger qué visualizar u ocultar, cambiar los datos o los parámetros del modelo que producen las imágenes, se trata de una **visualización interactiva** (en inglés, *INTERACTIVE VISUALIZATION*). Si por el contrario, el usuario es sólo un espectador de las imágenes generadas por la computadora, se le llamará una **animación por computadora** (en inglés, *COMPUTER ANIMATION*) en este documento.
- *Origen de los datos*. La computadora genera imágenes a partir de un conjunto de datos sobre el fenómeno que se quiere representar. Si el origen de los datos para producir las imágenes es una base de datos, se trata de una **visualización de datos** (en inglés, *DATA VISUALIZATION*). Si los datos son generados por un modelo que imita el comportamiento de un sistema o proceso, se trata de una **simulación por computadora** (en inglés, *COMPUTER SIMULATION*).
- *Fenómeno en estudio*. Por definición, las visualizaciones representan mediante imágenes fenómenos de naturaleza no visible. Las visualizaciones se pueden utilizar para representar diversidad de fenómenos naturales, sociales, o artificiales. Cuando el fenómeno a explicar es precisamente software, se trata de una *visualización de software*, y se presenta en la subsección que continúa.

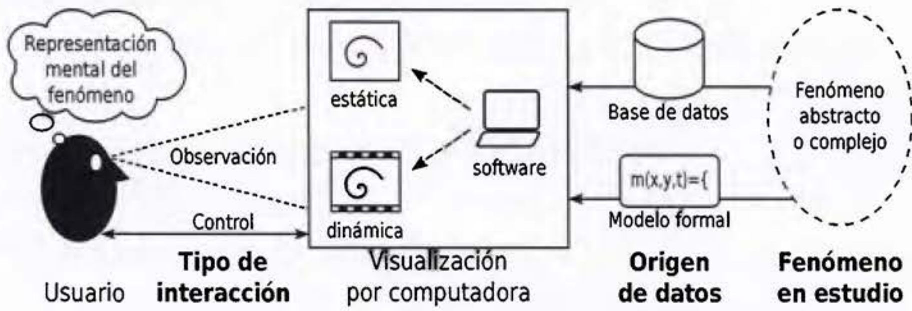



Figura 2.4. Esquema general de una visualización por computadora

## 2.2.2 Visualizaciones de software

El software es un ente abstracto e intangible. Es natural que visualizaciones se utilicen para ayudar a las personas a comprenderlo. Para efectos de este documento, una **visualización de software** (en inglés, *SOFTWARE VISUALIZATION*) es una *visualización por computadora* cuyo fenómeno en estudio es otro software. La Figura 2.5 esquematiza una visualización de software y, en su extremo derecho, el fenómeno a representar explícitamente se ha indicado

es el software en estudio. A un nivel técnico, una *visualización de software* es un software que genera imágenes, las cuales representan alguna o varias características de otro software, con el fin de ayudar a las personas a comprenderlo. Los componentes de una visualización de software se rotulan en negritas en la Figura 2.5.

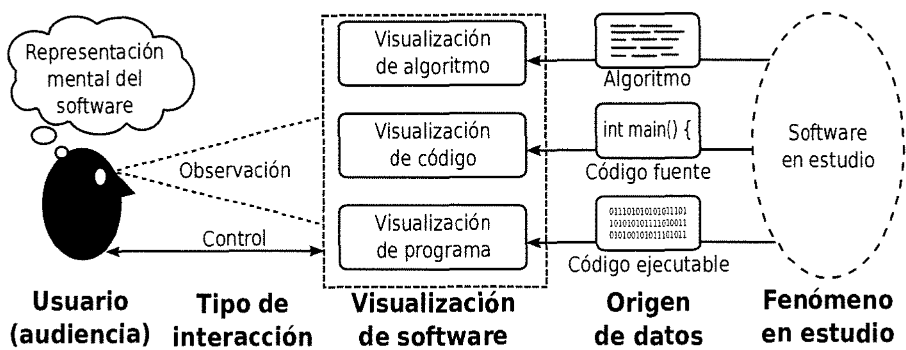


Figura 2.5. Esquema de visualizaciones de software (elaboración propia)

La mayoría de visualizaciones de software son *simulaciones por computadora* porque su origen de datos es un modelo formal: otro software. De acuerdo al nivel de abstracción con que se representa el software en estudio las visualizaciones de software se clasifican en:

1. **Visualización de algoritmo** (en inglés, *ALGORITHM VISUALIZATION*) representa el software en estudio en un alto nivel de abstracción, normalmente algoritmos generales como *QUICKSORT* o búsqueda binaria, independientemente del lenguaje de programación [Isohanni 2013; Sorva et al. 2013].
2. **Visualización de código** (en inglés, *CODE VISUALIZATION*) representa características estáticas del código fuente del software en estudio, por ejemplo, la dependencia entre archivos fuente o la evolución del código fuente en el tiempo [Sorva et al. 2013]. Por tanto, es dependiente del lenguaje de programación.
3. **Visualización de programa** (en inglés, *PROGRAM VISUALIZATION*) representa el comportamiento en tiempo de ejecución del software en estudio. Es dependiente tanto del lenguaje de programación como de su máquina nocional.

La nomenclatura anterior no es estándar. Por ejemplo, algunos autores utilizan el concepto *animación de programa* (en inglés, *PROGRAM ANIMATION*) como sinónimo de visualización de programa (por ejemplo [Ben-Ari et al. 2011; Sajaniemi and Kuittinen 2003; Bednarik et al. 2005]). Otros autores utilizan el término *visualización de programa* como un súper-conjunto

de la categoría 2 (visualización de código) y la categoría 3 (a la que llaman animación de programa) de la lista anterior, como es el caso de [Sorva et al. 2013].

El diseñador de una visualización de software debe escoger qué características del software en estudio se van a representar. Si la selección es de una o muy pocas características, se trata de una **visualización especializada de software.** Por ejemplo una *visualización especializada de programa* representa el comportamiento en tiempo de ejecución de uno o unos pocos conceptos de programación, como paso de parámetros o punteros [Sorva et al. 2013, p.7]. Si la visualización representa la mayor parte de las características del fenómeno, se trata de una **visualización genérica de software.** Por ejemplo una *visualización genérica de programa* representa la mayoría de conceptos de una máquina nocional (adaptado de [Sorva et al. 2013, p.7]).

Las visualizaciones especializadas de programa tienen la ventaja de concentrar la atención en un tema, utilizando metáforas y trucos visuales apropiados para explicarlo, abstrayendo otros conceptos de la máquina nocional que podrían generar distracción [Sorva et al. 2013, p.7]. Las visualizaciones genéricas de programa tienen la ventaja de proveer un panorama general de la máquina nocional que no se puede lograr con un cúmulo de visualizaciones específicas aisladas, cada cual con su curva de aprendizaje [Sorva et al. 2013, p.7]. Otra ventaja de las visualizaciones genéricas de programa es que tienden a admitir programas propios de los estudiantes, lo cual no es común en las visualizaciones específicas [Sorva et al. 2013, p.7].

### 2.2.3 Audiencia pretendida de las visualizaciones

Las visualizaciones se construyen para ayudar a las personas a comprender un fenómeno no directamente visible o complejo. Estas personas toman el rol de *observadores* si son pasivos o de *usuarios* si pueden interaccionar con la visualización. La **audiencia pretendida de una visualización** es una categorización de los observadores o usuarios a los que pretende ayudar. Por ejemplo, las imágenes por resonancia magnética están pretendidas para personal de salud.

Las características a explicar del fenómeno en estudio por una visualización pueden variar significativamente de acuerdo a la audiencia pretendida, como ocurre en las visualizaciones de software. Por ejemplo, una visualización de programa para un profesional en informática

podría resaltar detalles muy precisos o muy resumidos para ayudar a comprender la ejecución de un sistema complejo; mientras que una visualización de programa para un estudiante de informática podría resaltar conceptos básicos de programación, como ciclos o paso de parámetros (ampliado de [Sorva et al. 2013, p.7]). En el contexto de la educación de la computación, las visualizaciones de programa tienden a usarse en los cursos introductorios de programación, mientras que las visualizaciones de algoritmos en cursos posteriores sobre estructuras de datos y análisis de algoritmos [Isohanni 2013].

El objeto de estudio de esta investigación son las visualizaciones de programa pretendidas para aprendices de programación[9]. A menos de que se indique explícitamente, se asumirá esta audiencia en el resto de este escrito. Es decir, en adelante una **visualización de programa** es un software que genera imágenes para ayudar a estudiantes de programación a comprender el comportamiento en tiempo de ejecución de otro software. Para comprender la ejecución de un programa es necesario comprender cómo se distribuyen sus instrucciones y datos en la memoria de la computadora, y cómo las instrucciones son ejecutadas por el procesador. Por tanto, las visualizaciones de programa implementan la técnica de *rastreo de memoria de programa* descrita por [Hertz and Jump 2013].

La clasificación de visualizaciones presentada en esta sección, ayuda a categorizar los tres tipos de materiales más utilizados por los profesores para enseñar la máquina nocional de un lenguaje de programación:

1. *Visualizaciones estáticas.* Corresponden a ilustraciones o animaciones que no cambian en el tiempo, usualmente proyectadas en diapositivas.
2. *Visualizaciones manuales.* Tienden a ser simulaciones dinámicas en pizarra que el profesor actualiza conforme ejecuta las líneas de código del programa. También pueden ser realizadas por estudiantes como ocurre en [Hertz and Jump 2013].
3. *Visualizaciones de programa.* Refieren a simulaciones automáticas y dinámicas realizadas por un software. Son las que se estudian en esta investigación.

Los dos primeros tipos de visualizaciones son los más usados en los ambientes de aprendizaje para explicar la máquina nocional de un lenguaje de programación. Sin embargo han sido criticados como ineficientes para explicar un fenómeno inherentemente dinámico [Sorva

[9] Algunas visualizaciones de programa amplían su audiencia a profesores de programación, quienes pueden usarlas durante las lecciones para explicar los fenómenos de programación.

2012]. El tercer tipo, las visualizaciones de programa, se han propuesto como alternativa; pero su eficacia es cuestionada y su uso es raro en los ambientes educativos [Sorva et al. 2013; Naps et al. 2003; Isohanni and Järvinen 2014]. Dado que las visualizaciones de programa son herramientas de aprendizaje, la sección que continúa presenta una teoría de aprendizaje, y de ella se infieren requerimientos que podrían ayudar en su eficacia.

## 2.3 Constructivismo sociocultural

Hay cuatro orientaciones que agrupan a las teorías de aprendizaje: conductismo, constructivismo, humanismo y cognitivismo [Wu et al. 2012, p.2]. Se escogió el constructivismo porque se vislumbra como la teoría que permitirá hacer frente a los retos del aprendizaje de la computación, como la explosión de la información, los rápidos cambios tecnológicos, globalización y la necesidad de reconocer varios puntos de vista sobre el conocimiento [Greening 2000]. Hay dos tipos principales de constructivismo: el constructivismo cognitivo o individual basado en el trabajo de Jean Piaget, y el constructivismo sociocultural de Lev Vygotsky y su círculo [Powell and Kalina 2009]. Se escogió el segundo por no estar acotado al desarrollo ontogenético del estudiante, y por ser recomendado como el más apto para investigaciones con herramientas educativas [Bonk and Cunningham 1998, p.30].

Vygotsky y su círculo formularon a inicios de siglo XX una teoría de desarrollo cognitivo conocida como teoría histórico-cultural, teoría sociocultural, o teoría del desarrollo social. Un subconjunto de esta teoría del desarrollo atiende la temática del aprendizaje de forma constructivista. En este documento se le referirá como teoría del **constructivismo sociocultural** para distinguirla de otras formas de constructivismo, como el constructivismo radical y el construccionismo [Steffe and Gale 1995].

Se infirió el proceso de aprendizaje presentado en esta sección a partir del compendio de artículos en [Luria et al. 2011], publicados originalmente entre 1934 y 1961. De la mediación recomendada para el docente en el proceso inferido, se trató de identificar los requerimientos de software de alto nivel, ya que las visualizaciones de programa son herramientas socioculturales que median en el contexto del aprendiz.

El **aprendizaje** es un proceso de reestructuración neuronal cuya verificación conductual es estable en el tiempo (adaptado de [Francis 2012, p.55]). La capacidad de aprender es un mecanismo de adaptación compartido entre animales y personas. Sin embargo, las personas tienen otra capacidad biológica, no presente en los animales, la de crear asociaciones más complejas a partir de otras existentes, a las cuales Pavlov llamó *segundo sistema de señales* [Bogoyavlensky and Menchinskaya 2011a]. Esta capacidad permite al niño asimilar el lenguaje, y con éste la historia cultural de sus pares, generalizar las experiencias, pensar, y regular su comportamiento [Luria et al. 2011].

Conforme se apropia del lenguaje, el niño aprende las primeras nociones informales de historia, física, economía y de otras disciplinas, de acuerdo a su experiencia de vida junto a su grupo de personas cercanas. Durante la educación formal, que incluye la universidad, las nuevas nociones no se aprenden de forma aislada, sino que son asociadas a las ya construidas previamente gracias al segundo sistema de señales.

Para Vygotsky y su círculo el aprendizaje ocurre por la construcción de conceptos en un contexto sociocultural, en el cual intervienen otras personas o herramientas culturales (como visualizaciones de programa) que se les referirá como **mediadores**. En esta teoría, cada concepto tiene que ser asociado a otros ya presentes en la mente del estudiante. La construcción y asociación no es inmediata, sino que ocurre por un proceso. En los artículos consultados en [Luria et al. 2011] no se provee un modelo de este proceso, si no que se ofrecen varios principios que ayudan al aprendizaje. El autor de esta tesis sintetizó y adaptó de [Luria et al. 2011] los pasos diagramados en la Figura 2.6.

El proceso inicia con la selección del concepto que se pretende aprender, y continúa recorriendo los principios de la Figura 2.6 hasta lograr el que el concepto se integre a los sistemas de conceptos en la memoria de largo plazo del estudiante. A modo de ejemplo, supóngase que el concepto seleccionado es el puntero en C/C++. Cada uno de los seis principios resaltados en la Figura 2.6 son abordados en las siguientes subsecciones incluyendo el puntero como ejemplo.

Figura 2.6. Principios de aprendizaje del constructivismo sociocultural

### 2.3.1 Motivación para el aprendizaje

Un grupo de estudiantes con diversidad de historias de vida tendrá diversidad de motivaciones de aprendizaje. Como primera labor, conviene que el mediador indague qué motiva al estudiante a involucrarse en la tarea educativa. Preguntar al estudiante: ¿aprender para qué? Para ¿ser mejor que los demás? ¿hacerse más rico? ¿ser más "humano"? ¿complacer a los padres? [Cecchini 2011]. Toda acción que trate de emprender el mediador será amplificada o atenuada por esta intención en el estudiante. De este principio se obtiene el primer requerimiento de software:

| **Requerimiento 1: Motivación** |
| --- |
| Indagar qué motiva al estudiante a involucrarse en la tarea educativa, con el fin de personalizar la interacción con el usuario. |

Un estudiante motivado mantiene su mente en un estado activo. Una condición activa de la corteza es necesaria para la formación de conexiones temporales. Es decir, el cerebro en condición pasiva no forma ninguna conexión [Bogoyavlensky and Menchinskaya 2011a; Tam 2000]. Estas conexiones temporales son el paso inicial del aprendizaje. Cecchini indica que la

clase no es un grupo de oyentes pasivos, poco interesados, y dominados por un expositor [Cecchini 2011]. La clase es un colectivo de personas que interaccionan entre sí para conseguir un fin común. De este principio se infiere el segundo requerimiento de software de alto nivel:

| **Requerimiento 2: Estado activo** |
|---|
| Propiciar una interacción en la que el usuario mantenga su mente en estado activo. |

El mediador puede ayudar a estimular el interés de los estudiantes como condición preliminar de trabajo con un material nuevo. Algunas técnicas recopiladas por [Bogoyavlensky and Menchinskaya 2011a] son:

1. Ayudar a los alumnos a comprender la importancia y utilidad de dominar la temática (o los riesgos de no hacerlo).
2. Ayudar a los alumnos a comprender los objetivos de la sesión.
3. Establecer relaciones con las nociones precedentes.
4. Plantear situaciones problemáticas.
5. Introducir elementos que susciten en los alumnos actitudes emotivas ante la lección, por ejemplo, juegos.

A modo de ejemplo, se aplicará la técnica 1 de la lista anterior para introducir el concepto de puntero. El mediador podría ofrecer datos curiosos sobre este concepto o ayudar a los estudiantes a encontrarlos, como el siguiente. Se han propuesto en la literatura científica de computación muchos candidatos a ser conceptos umbral (en inglés, *THRESHOLD CONCEPTS*), pero sólo uno ha recibido acuerdo unánime: el puntero. Los conceptos umbrales transforman la mente del estudiante a una forma de pensar que es distinta de cualquier otra disciplina [Sorva 2010]. Así, para pensar como un informático, se debe comprender este concepto. Estos datos podrían despertar el interés por el concepto de puntero y propiciar, por tanto, una condición activa de la mente. De estas sugerencias se infiere el tercer requerimiento de software:

| **Requerimiento 3: Interés** |
|---|
| Estimular el interés del usuario por el concepto a aprender. |

Una vez iniciada la sesión, es ideal que los estudiantes conserven el interés, y por tanto, sus mentes continúen en condición activa. Este ideal se puede lograr si los aprendices se

mantienen realizando procesos analítico-sintéticos, comparando y contraponiendo el material nuevo con el viejo, entre otras técnicas, como se verá más adelante. [Bogoyavlensky and Menchinskaya 2011a].

El docente debe tener claro que las nociones que se pretenden asimilar en la educación formal se deben construir con el conocimiento previo que el estudiante trae de su experiencia de vida. Estas experiencias provocan revivir emociones que pueden afectar la motivación del estudiante hacia las nuevas nociones. Por ejemplo, el concepto de puntero requiere de variables enteras y éstas se asocian a incógnitas de álgebra. Si en la educación secundaria el estudiante tuvo experiencias negativas con álgebra, estas emociones pueden revivir y afectar el aprendizaje de los punteros.

De acuerdo constructivismo social, el contexto se asocia también a las nuevas nociones. Dado que la interacción del mediador y las emociones son parte del contexto, el estudiante las asociará con el nuevo concepto. Cuando este concepto se use para asociar a otros nuevos, las emociones pueden resurgir y tener un efecto estimulante o inhibidor para el aprendizaje de esos nuevos conceptos. Se recomienda entonces al mediador proveer realimentación en caso de éxito, y ofrecer explicaciones o ayudar a los estudiantes a encontrarlas en caso de errores. [Bogoyavlensky and Menchinskaya 2011a]. Por ejemplo, si un estudiante trata de sumar un puntero con un entero, el profesor podría preguntar al estudiante si lo que quiere sumar es la dirección de memoria del puntero o el valor apuntado, en caso de que responda lo segundo, volvería a preguntarle cómo se puede obtener el valor apuntado por un puntero, y así sucesivamente hasta que logre resolver el problema. De este principio se infiere el cuarto requerimiento de software de alto nivel:

| **Requerimiento 4: Realimentación** |
| --- |
| Realimentar al usuario sobre sus éxitos y ofrecer explicaciones en caso de error. |

### 2.3.2 Contraposición conceptual

El trabajo del mediador es ayudar al estudiante a alcanzar un mayor nivel de desarrollo a través del aprendizaje, influyendo en su zona de desarrollo próximo [Vygotsky 2011]. Después de crear un ambiente de motivación para el aprendizaje, el mediador puede confrontar al estudiante con las nociones que necesita o quiere aprender para alcanzar el

desarrollo planteado. Sin embargo, la mente del alumno no es un repositorio que absorbe nociones como una esponja absorbe agua. El aprendizaje se da por asociación de las nuevas nociones con nociones previas, ya adquiridas en la etapa informal o formal. Por ejemplo, L.I. Kaplan solicitó a sus alumnos universitarios leer un texto y luego explicar lo que comprendieron y estos lo hicieron con otras palabras a las del texto, aunque con un significado similar. Las personas no reproducen el texto sino que lo reconstruyen utilizando su conocimiento previo [Bogoyavlensky and Menchinskaya 2011a].

Con los conceptos que los estudiantes aprendieron en la educación informal o previa, se construyen los conceptos de la educación formal [Vygotsky 2011; Bogoyavlensky and Menchinskaya 2011a]. Por muy abstracto que pueda ser el concepto, su asociación es siempre con la experiencia directa de la persona. No se puede asociar un concepto a la nada y decir que se haya aprendido o que se pueda aplicar [Bogoyavlensky and Menchinskaya 2011a]. Idealmente el mediador podría evaluar el conocimiento previo de sus estudiantes y las asociaciones emotivas que de este conocimiento tengan, ya que será el terreno donde se construirán las nuevas nociones. De este principio se obtiene el quinto requerimiento de software de alto nivel:

**Requerimiento 5: Nociones previas**

Evaluar las nociones previas del estudiante y las asociaciones emotivas que de ellas se tengan, para adaptar la forma de introducir un concepto.

Por ejemplo, si el estudiante tiene nociones equivocadas o incompletas sobre el concepto de incógnita, se tornará difícil la construcción del concepto variable entera, y por consecuencia, de puntero. En estos casos se recomienda emplear la técnica de la contraposición conceptual [Bogoyavlensky and Menchinskaya 2011a].

La **contraposición conceptual** (conocida en Occidente como *disonancia cognitiva* [Harmon-Jones and Mills 1999]) consiste en hacer ver al estudiante que sus viejas nociones son insuficientes o contradictorias para alcanzar los objetivos a los que está motivado a alcanzar. Estas contradicciones internas entre sus aspiraciones y el nivel de desarrollo por él alcanzado hacen surgir la fuerza motriz del desarrollo [Kostiuk 2011].

Por ejemplo, antes de introducir el concepto de puntero, el mediador podría retar a los estudiantes a resolver un problema que requiere este concepto (pero sin hacer explícita esta necesidad), como una función de intercambio (en inglés, *SWAP*) [Hidalgo-Céspedes et al.

2016b]. Es probable que los estudiantes traten de implementar la función con los conceptos que conocen, como variables (parámetros por valor). Cuando los estudiantes ejecuten su solución, encontrarán que su código no funciona como se espera, y por tanto deducir que las nociones que conocen no son suficientes. De esta técnica se infiere el sexto requerimiento de software:

**Requerimiento 6: Contraposición conceptual**

Contraponer conceptualmente los nuevos conceptos a los ya existentes en la mente el estudiante cuando las viejas nociones no permitan la construcción de las nuevas nociones.

La contraposición conceptual crea un grado de incertidumbre cognoscitiva en el estudiante al ver que sus nociones y habilidades son insuficientes. Dicha incertidumbre sólo puede ser superada mediante la reorganización de viejos conceptos y la construcción de nuevos. Este es un estado de apertura mental en el estudiante para la construcción del nuevo concepto, con el fin de reducir la incertidumbre, superar las contradicciones que lo intrigan, y alcanzar un estado de satisfacción. Este estado de gratificación constituye la motivación intrínseca del proceso mismo, por tanto, se trata de un proceso auto-motivado que no requiere de un refuerzo externo [Cecchini 2011]. Una vez que la mente del estudiante se encuentra en estado activo, de interés y de incertidumbre, está listo para la asimilación del nuevo concepto.

### 2.3.3 Asimilación del concepto

Si se ha utilizado la contraposición conceptual, la mente del alumno probablemente estará necesitando el concepto que produzca la satisfacción de su incertidumbre. Por ejemplo, para hacer trabajar correctamente la función de intercambio de variables. El mediador puede entonces presentar el nuevo concepto explicado en función de otros conceptos que ya forman parte del conocimiento del aprendiz.

Por ejemplo, para introducir el concepto de puntero, se puede recurrir primero a punteros usados en la vida cotidiana, como los tres presentes en la señal de tránsito de la Figura 2.7. Supóngase que un conductor ve que en la señal se dice su destino, el aeropuerto, sin embargo, el número 4 y la flecha hacia la derecha le indican que el aeropuerto no está ahí, sino que debe virar hacia la derecha y recorrer cuatro kilómetros más para llegar a su destino. Usando esta

metáfora se puede introducir el concepto de puntero en la memoria de la computadora, y asociarlo con otros conceptos previos que los lenguajes de programación usan para implementarlo, como variable entera, dirección de memoria, y tipo de datos.

Figura 2.7. Ejemplo de un puntero cotidiano en una señal de tránsito

En el caso de una visualización de programa, el puntero podría representarse como una variable entera con capacidades adicionales para apuntar a algún otro lugar de la memoria. Del paso de asimilación se infiere el séptimo requerimiento de software:

> **Requerimiento 7: Asimilación**
>
> Visualizar el nuevo concepto asociándolo con nociones que el estudiante ya tiene en su mente, traídas de su experiencia de vida.

En condiciones normales, el estudiante no se apropiará del concepto de inmediato, sino que sólo habrá establecido un sistema de conexiones temporales del concepto nuevo con los existentes en la memoria de corto plazo. Seguidamente, se recomienda que el estudiante ejercite un proceso de análisis-síntesis-aplicación que irá gradualmente estabilizando las asociaciones en la memoria a largo plazo que se explica a continuación.

Al enfrentarse a un objeto de estudio, sea un concepto o un proceso nuevo, la mente del estudiante entrará en una fase de *análisis* o *abstracción*. El estudiante descompone el objeto (concreto o verbal) en sus partes para abstraer sus propiedades esenciales e identificar los componentes que ya conoce [Bogoyavlensky and Menchinskaya 2011a]. Por ejemplo, el estudiante puede descomponer el concepto de puntero en sus partes, como su identificador, su valor (la dirección de memoria), el tipo de datos apuntado, el tamaño del puntero, y su lugar en la memoria.

El alumno reconoce las propiedades abstraídas en algunos géneros de objetos y no en otros. Por ejemplo, al ver varios ejemplos de programas que involucren punteros y variables enteras, el estudiante identifica que los punteros se distinguen de las variables enteras principalmente por la propiedad del dato apuntado. Las propiedades identificadas pasan a

considerarse propiedades generales del concepto expresadas con ayuda de la palabra. Es decir, el estudiante realiza una fase de *síntesis* o *generalización* verbal del concepto en función de otros que ya conocía. Por esta razón, el alumno no utiliza exactamente las mismas palabras con que el mediador introdujo el concepto [Bogoyavlensky and Menchinskaya 2011a].

Dado que el estudiante analiza y generaliza el nuevo concepto en función de conceptos que ha aprendido en su vida cotidiana, se abre una ventana de posibilidades para que el nuevo concepto sea mal construido [Bogoyavlensky and Menchinskaya 2011a]. Por ejemplo, dado que todos los punteros son variables enteras, un estudiante podría generalizar que ambos son intercambiables. Si los estudiantes cometen errores, prácticas como las dos siguientes pueden ayudar a aprovecharlos para el aprendizaje.

La primera práctica es que el estudiante realice una cuidadosa comparación del concepto nuevo con los viejos para identificar sus propiedades elementales comunes, y distinguir sus diferencias fundamentales. Por ejemplo, el estudiante podría experimentar intercambiando variables enteras con punteros en el lenguaje de programación, asignarle valores negativos a ambos, o indagar el significado del valor cero para cada uno de ellos. De esta práctica se infiere el octavo requerimiento de software de alto nivel:

**Requerimiento 8: Comparación de conceptos**

Permitir al usuario realizar una cuidadosa comparación del concepto nuevo con los viejos, para identificar sus propiedades elementales comunes y distinguir sus diferencias fundamentales.

La segunda práctica es fortalecer la asociación del nuevo concepto con cierto número de ejercicios, los cuales varían entre ellos en aspectos no esenciales, mientras conservan el aspecto esencial que es el concepto o la regla en estudio. Al ser confrontado con un nuevo concepto, el estudiante construye un sistema de conexiones temporales. Al realizar múltiples ejercicios que mantienen el principio fundamental del concepto recién introducido, el aprendiz estabiliza el subconjunto de conexiones del principio fundamental; mientras que los aspectos no esenciales se mantienen como conexiones inestables, lo que reduce el riesgo de tomar precedencia sobre el principio fundamental. Por ejemplo, el estudiante podría responder una serie de preguntas conceptuales sobre los punteros y los conceptos relacionados, como ¿qué ocurre si se "des-referencia" un puntero sin inicializar?, ¿se puede asignar una variable entera a un puntero?, ¿cuál es la diferencia en el significado del valor cero

en una variable entera y en un puntero? De esta práctica se infiere el noveno requerimiento de software de alto nivel:

> **Requerimiento 9: Ejercicios conceptuales**
>
> Fortalecer la asociación del nuevo concepto con cierto número de ejercicios, que varían entre ellos en aspectos no esenciales, mientras conservan el aspecto esencial del concepto.

### 2.3.4 Aplicación del concepto

La asimilación de los conceptos abstractos a través de procesos de análisis y síntesis no es el punto final de la adquisición de conocimiento. La segunda parte, del pensamiento abstracto a la práctica es tan importante como la primera, y normalmente la de mayor dificultad. Esta dificultad se ha encontrado, por ejemplo, en estudiantes que aprenden trigonometría del triángulo rectángulo utilizando dibujos abstractos, y luego se les cambia por dibujos de tejados u otros escenarios (incluso manteniendo las mismas medidas), varios estudiantes se sienten confundidos y no logran resolver los ejercicios. Según [Fleshner 2011] la causa de estas confusiones se debe a que los estudiantes están en capacidad de resolver problemas que tengan la misma naturaleza que el material estudiado (la clase, el libro de texto). Cuando se les presentan problemas aplicados, los estudiantes requieren de un doble análisis para comprender el problema y luego aplicar las abstracciones ya conocidas. Es decir, estos ejercicios tienen el doble de dificultad [Bogoyavlensky and Menchinskaya 2011a], y son de importancia para ayudar al estudiante a aplicar los conocimientos aprendidos en la vida cotidiana o profesional, porque "la instrucción pedagógica verbal, que el niño no pone en práctica, no aporta ningún cambio real a su vida, a su posición en el colectivo" [Kostiuk 2011, p.54]. De este principio se infiere el décimo requerimiento:

> **Requerimiento 10: Aplicación del concepto**
>
> Permitir al usuario aplicar el nuevo concepto a situaciones cotidianas, reales, o profesionales, útiles en su experiencia de vida. Cada situación plantea cambios en el contexto de aplicación del concepto, pero mantiene el principio fundamental del concepto.

Por ejemplo, se puede retar al estudiante a resolver problemas que requieren la aplicación de punteros, como implementar una función cuadrática que retorne el valor de las raíces en parámetros por referencia, o corregir una función que retorna un puntero a una variable local.

Dos medios efectivos para que los alumnos apliquen las nociones a situaciones reales son la solución de problemas [Bogoyavlensky and Menchinskaya 2011a] y la educación artística [Teplov 2011]. Ambos requieren de un método general, un proceso para llegar a una solución o a un producto. Se recomienda al mediador centrarse más el proceso que en el producto final. Al evaluar lo aprendido, es valioso que el mediador preste atención al proceso seguido por el estudiante, el cual revela una riqueza de detalles sobre el aprendizaje de las nociones y, en especial, de su aplicación.

Se ha encontrado que para resolver un problema exitosamente, se requiere un análisis inicial completo de las condiciones del problema, preparar un plan de acción, y sólo después, ejecutarlo en la práctica. Esto pone de manifiesto la importancia del análisis de las instrucciones verbales (la palabra) antes de pasar a las acciones, y no lo opuesto. [Bogoyavlensky and Menchinskaya 2011a]

Por ejemplo, al brindar ejemplos en clase, como la función de intercambio (en inglés, *SWAP*), el profesor puede hacer explícito el proceso que sigue para resolver los problemas. Al ayudar a los estudiantes a resolver problemas aplicados, el docente puede solicitarles verbalizar su proceso de resolución de problemas, e influir para que este proceso se convierta en un hábito adecuado. De este principio se infiere el undécimo requerimiento de software de alto nivel:

**Requerimiento 11: Enfoque en el proceso**

Enseñar y evaluar un proceso más que un producto final en la etapa consciente antes de que las acciones se conviertan en hábitos.

### 2.3.5 Formación de hábitos

Los métodos para la solución de problemas pueden convertirse en hábitos, de tal forma que la persona resuelve los problemas que le son más familiares de forma inmediata, durante la fase de percepción de las condiciones de la tarea. La persona sigue resolviendo los problemas menos familiares mediante procesos de análisis y síntesis tradicionales. La tarea de la enseñanza de resolución de problemas es proporcionar un método en la etapa consciente

antes de que las acciones se conviertan en operaciones mecánicas (hábitos). Los hábitos se forman a consecuencia del refinamiento de un método a través de su repetición, ya que cada acción repetida no debe ser una copia de la acción anterior, sino que involucra cambios menores, pero manteniendo el principio fundamental. [Bogoyavlensky and Menchinskaya 2011a]

Las reacciones a las primeras repeticiones de la tarea son ampliamente difusas para el estudiante, quien debe recurrir a procesos de análisis y síntesis para resolver el problema partiendo de lo general hacia lo específico. Cada repetición suscita una nueva reacción de los componentes no ejercitados por la iteración anterior. Al iniciar una iteración, el estudiante establece una conexión natural con la anterior, ya que mantienen el mismo principio teórico. Esta conexión se establece entre el juicio deductivo (el razonamiento) de la segunda tarea con el juicio operativo (la acción) recién realizado de la tarea anterior sin interrupciones de razonamiento. Por esto, el juicio deductivo será menor en la segunda tarea, y cada vez se irá reduciendo más conforme el estudiante realiza más tareas afines hasta un punto en que es casi inexistente, al que se le reconoce como la formación del hábito. [Bogoyavlensky and Menchinskaya 2011a]. De este principio se infiere el próximo requerimiento:

| **Requerimiento 12: Formación de hábitos** |
|---|
| Incentivar al usuario a aplicar el nuevo concepto hasta que desarrolle un hábito. |

Por ejemplo, el mediador puede retar al estudiante con un conjunto de ejercicios que requieren la aplicación de punteros. Cada ejercicio varía el contexto en el que se aplica el concepto, de tal forma que el estudiante al resolver la secuencia de ejercicios formará hábitos de aplicación de los punteros. Durante la resolución de los mismos, el profesor puede prestar atención al proceso seguido por el aprendiz.

### 2.3.6 Sistemas de conceptos

Para Vygotsky, el conocimiento no se forma con la asimilación de conceptos aislados, sino en sistemas de conceptos conectados o relacionados. Estos sistemas son redes de conexiones que reflejan las relaciones existentes entre los objetos y los fenómenos del mundo real. Por ejemplo, se ha encontrado que los alumnos asimilan mejor los conceptos de ángulos adyacentes y opuestos por el vértice, cuando están incluidos en el concepto más amplio de

"ángulos con un vértice común" [Bogoyavlensky and Menchinskaya 2011a]. El mismo principio puede aplicarse al concepto de puntero, al incluirlo en un concepto más general que puede llamarse "indirección de memoria", que abarca otros conceptos como referencias en C++.

En el caso de la programación, al menos, cada nuevo concepto puede asociarse a una diversidad de conceptos previos. Por ejemplo, es apremiante que el estudiante relacione el puntero con conceptos como los de entrada y salida (leer a través de punteros, imprimir direcciones de memoria, imprimir valores apuntados con otros tipos de datos), expresiones (combinar valores apuntados con operadores), condicionales (punteros en un contexto booleano), ciclos (repetir hasta que un puntero que se haga nulo), subrutinas (punteros a funciones), arreglos (aritmética de punteros, acceso por índice o por puntero, y arreglos de punteros a funciones). De estos sistemas de conceptos se infiere el siguiente requerimiento de software:

| **Requerimiento 13: Sistemas de conceptos** |
|---|
| Ayudar al estudiante asociar el nuevo concepto con otros ya presentados para formar sistemas de conceptos. |

Se recomienda organizar el material de aprendizaje de tal forma que refleje una jerarquía natural de conceptos. Después de que el estudiante desarrolle hábito con un concepto, se le puede retar con ejercicios que requieren la aplicación de ese concepto junto con otros conceptos previos. Estos ejercicios incentivan la formación de sistemas de conexiones y aplicarlos a nuevas situaciones. [Bogoyavlensky and Menchinskaya 2011b]. Por ejemplo, en un curso de programación, el docente puede ordenar los conceptos jerárquicamente por relaciones de dependencia. En el caso del puntero, este concepto depende de variables enteras, direcciones de memoria, y expresiones aritméticas.

Una vez asimilado el concepto de puntero, puede establecer sistemas de conceptos con subrutinas, paso de parámetros, y segmentos de memoria. Conviene ordenar en el tiempo estos conceptos y sus ejercicios antes que el puntero, de tal forma que se encuentren construidos en la memoria a largo plazo del estudiante en el momento de presentar el puntero. Si este orden no se cumple, el puntero no estaría en la zona de desarrollo próximo del estudiante al presentarlo, lo que impediría su aprendizaje. De este principio se infiere el siguiente requerimiento:

**Requerimiento 14: Organización de conceptos**

Reflejar una organización natural de los sistemas de conceptos.

Es habitual realizar evaluaciones indirectas sobre los conocimientos inmediatamente después de adquiridos y llamarlos "nivel de asimilación" de nociones[10]. Pero, de acuerdo a los autores, estas mediciones son incompletas y no fiables. Se recomienda confirmar que estos resultados sean estables con el paso del tiempo y sean aplicados a situaciones nuevas. [Bogoyavlensky and Menchinskaya 2011b, p.70]

Por ejemplo, mientras el nuevo concepto (puntero) es asociado a otros previos (entrada, salida, expresiones, condicionales, ciclos, arreglos, subrutinas, mencionados anteriormente), conviene que el mediador preste atención al dominio de esos conceptos previos por parte del estudiante. De este principio se infiere el siguiente requerimiento de software:

**Requerimiento 15: Evaluación**

Evaluar que los sistemas de conceptos asimilados por el estudiante sean estables en el tiempo.

[Mileryan 2011] encontró experimentalmente que seguir pasivamente un método no es suficiente para que los estudiantes transfieran habilidades de un contexto a otro, o de una disciplina a otra. Los aprendices requieren conocer los principios teóricos de los que se deriva el método y las herramientas para lograr la transferencia. De esta forma, el mediador puede procurar que el estudiante no sólo aprenda un método, sino varios. Además de saber cómo escoger un método adecuado a una situación particular, o cómo generar uno nuevo a partir de las bases teóricas.

Tras la construcción de cada nuevo sistema de conceptos, el estudiante podrá resolver problemas de mayor complejidad. Por ejemplo, al asociar conceptos como arreglo, puntero a función, y valores pseudo-aleatorios, el estudiante puede resolver problemas donde la máquina escoge al azar subrutinas a partir de un arreglo de punteros a funciones, como podría ser el caso de un juego de azar que va incrementando su dificultad en el tiempo. Se recomienda retar al aprendiz con problemas contextualizados en otras disciplinas o áreas del

[10] El conocimiento no se puede medir directamente, sino a través de evidencia que el estudiante genera. Por ejemplo, a través de un proceso realizado o el producto obtenido al resolver un problema.

conocimiento, de tal forma que ayude en la eventual transferencia que realizará en su quehacer profesional. De esta transferencia se infiere el último requerimiento de software:

| **Requerimiento 16: Transferencia** |
| --- |
| Ayudar al estudiante a aplicar los sistemas de conceptos a situaciones nuevas y complejas. |

Tras haber aprendido el nuevo concepto, el estudiante habrá alcanzado un mayor nivel de desarrollo, y con él la sociedad [Bogoyavlensky and Menchinskaya 2011a]. El desarrollo mental no se detiene ahí, sino que continúa con la selección y el aprendizaje de nuevos conceptos (Figura 2.6, p33).

El constructivismo confiere mucha importancia a la asociación de conceptos para que el aprendizaje ocurra. Dado que los conceptos en computación son abstractos, se recurre a su asociación con conceptos de la experiencia de vida a través de metáforas.

## 2.4 Metáforas y alegorías

Históricamente, desde Aristóteles, las metáforas han sido consideradas un fenómeno lingüístico usado para describir algo como si fuese otra cosa, en virtud de características similares [Lakoff 1993; Blackwell 2006; Colburn and Shute 2008]. En el siglo XX, la **teoría de comparación y sustitución** estableció que la metáfora es convertida a un símil con el fin de ser entendida, esto es, una comparación, y luego sustituida por un significado común [Black 1977; Cameron 2003]. Por ejemplo, la frase “nuestro hijo es un cerdo” es literalmente ilógica; por tanto, el lector debe moverse a la interpretación figurativa, cambiando la frase por un símil “nuestro hijo es como un cerdo”; luego encontrar características del dominio de los cerdos que puedan ser aplicadas a las personas, tales como "descuidado", “sucio” o “apestoso”; y finalmente sustituir la metáfora por la característica seleccionada, como “nuestro hijo es descuidado” [Sease 2008].

Max Black se opuso a la teoría de comparación y sustitución. Él propuso la **teoría de interacción** de la metáfora. Esta teoría estipula que el significado de la metáfora es establecida en la interacción o relación entre los dos términos, en lugar de características comunes que ambos tenían previamente [Black 1955; Black 1977]. Por ejemplo, la metáfora

"los cuatro tigres asiáticos" refiriéndose a Taiwán, Singapur, Hong Kong y Corea del Sur; no puede ser convertida a un símil con el fin de encontrar significado. El significado es establecido en la interacción socialmente construida entre esos países y el tigre en la cultura china.

Lakoff y Johnson propusieron luego una tercera teoría llamada teoría de la **metáfora conceptual** [Lakoff and Johnson 2003; Lakoff 1993]. Esta teoría establece que la metáfora es realmente un fenómeno cognitivo, una forma de pensar de nuestras mentes, en lugar de un fenómeno puramente lingüístico. Esta afirmación está basada en el uso ubicuo del lenguaje metafórico, en especial para referirse a conceptos abstractos; tales como tiempo, estado, cambio, causa, propósito y medios; los cuales son conceptualizados a través de metáforas [Lakoff 1993]. Nuevos estudios confirman la naturaleza neuropsicológica de las metáforas, porque activan dos regiones (dominios) conectados del cerebro simultáneamente [Lakoff and Johnson 2003].

Lakoff definió **metáfora** como un mapeo estructurado (en el sentido matemático) entre un dominio origen y un dominio destino de experiencia [Lakoff 1993]. El mapeo es estructurado porque está compuesto de correspondencias ontológicas entre entidades en el dominio origen y entidades en el dominio destino [Lakoff 1993]. En la Figura 2.8 se intenta esquematizar los componentes teóricos de la metáfora conceptual de acuerdo a la definición de Lakoff. El dominio origen es usualmente ordinario, conocido, o familiar, porque es usado para representar un dominio destino abstracto o menos conocido. Las correspondencias entre entidades permiten proyectar inferencias del dominio origen al dominio destino [Lakoff 1993].

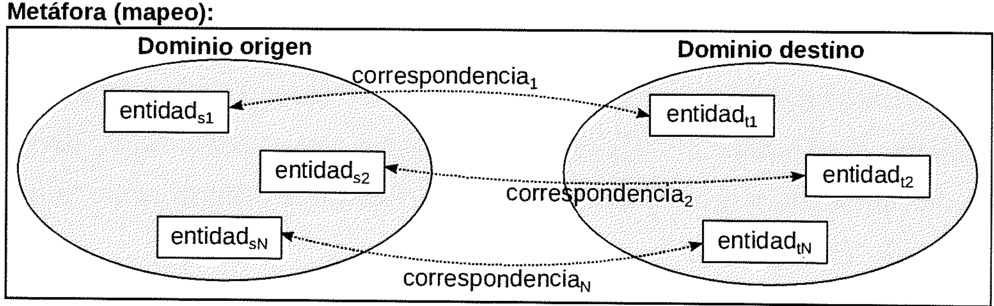



Figura 2.8. Esquema teórico de la metáfora de acuerdo a Lakoff (elaboración propia)

Una de las metáforas más analizadas por Lakoff en [Lakoff 1993] es EL AMOR ES UN VIAJE. Las mayúsculas en versalitas del nombre de la metáfora indican los dominios, de acuerdo a la

convención de Lakoff. Esta metáfora se esquematiza en la Figura 2.9 con cinco correspondencias a modo de ejemplo. Esas correspondencias permiten a las personas entender *expresiones metafóricas*, tales como: "Esta relación no está *yendo a ningún lugar*. Nuestras *ruedas están patinando*. Estamos en una *intersección*, y pueda que tengamos que *seguir por caminos separados*." (adaptado del inglés de [Lakoff 1993, p.206]. Las palabras en itálicas requieren interpretación metafórica. De acuerdo a Lakoff, la frases previas no son metáforas, sino **expresiones metafóricas**, y un número arbitrario de ellas pueden generarse a partir de una metáfora.

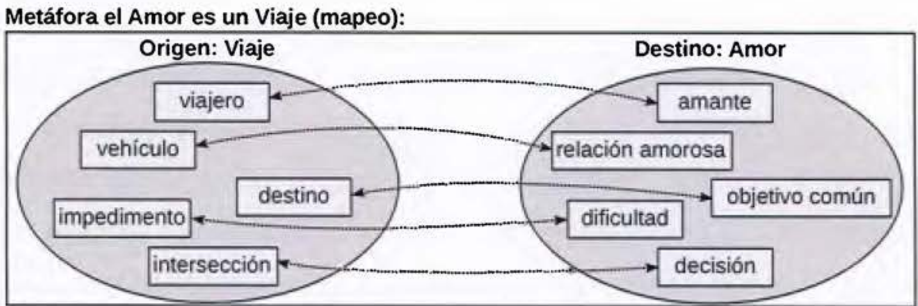



Figura 2.9. Esquema de la metáfora EL AMOR ES UN VIAJE con cinco correspondencias

La metáfora es el mapeo conceptual, tal como EL AMOR ES UN VIAJE encerrado en un rectángulo global en la Figura 2.9. Las metáforas pueden explicarse a través de sus correspondencias. Por ejemplo, EL AMOR ES UN VIAJE es explicado por Lakoff como sigue (se enfatizan las entidades con itálicas): "Los *amantes* son *viajeros* en un viaje juntos, con sus *objetivos comunes de vida* vistos como *destinos* a alcanzar. La *relación* es su *vehículo*, que les permite perseguir juntos esos *objetivos comunes*. Se dice que la *relación* consigue su propósito mientras les permita progresar hacia sus *objetivos comunes*. El viaje no es fácil, hay *impedimentos* y lugares (*intersecciones*) donde una *decisión* debe tomarse sobre cuál dirección tomar y si continuar viajando juntos" (adaptado del inglés de [Lakoff 1993, p.206]).

Lakoff clasifica las metáforas en dos tipos: metáforas conceptuales y metáforas de imagen [Lakoff 1993]. Hasta el momento se ha presentado la **metáfora conceptual** (en inglés, *CONCEPTUAL METAPHOR*), la cual se compone de correspondencias entre entidades de dos dominios conceptuales distintos. Una **metáfora de imagen** (en inglés, *IMAGE METAPHOR*) es un caso especial de una metáfora conceptual que tiene una única correspondencia entre una imagen mental convencional y una imagen mental destino [Lakoff 1993]. Por ejemplo, para entender la metáfora de imagen "mi esposa... cuya cintura es un reloj de arena", es requerido

tener una imagen mental de la forma de un reloj de arena, y proyectar la parte central del reloj en la cintura de ella, ya que la frase no provee esta información [Lakoff 1993, p.230].

Lakoff identificó que las metáforas en inglés, y en varios idiomas más, siguen dos principios: generalización e invariancia. El **principio de generalización** establece que hay una relación de herencia entre las entidades metafóricas. De esta forma, la metáfora se debe especificar al nivel supra-ordinado de las entidades. Por ejemplo, la metáfora EL AMOR ES UN VIAJE tiene una entidad origen *vehículo* (Figura 2.9) porque está al nivel supra-ordinado, y corresponde con la entidad destino *relación amorosa*. Entidades derivadas o subordinadas de vehículo; tales como, automóvil, tren, barco, o avión; heredan tal correspondencia. Expresiones metafóricas como "nuestra relación se va a hundir", no crean una nueva metáfora, sino que es considerada como parte de la metáfora EL AMOR ES UN VIAJE por el principio de generalización. Si las entidades de una metáfora no están al nivel supra-ordinado, la metáfora es considerada como derivada de una más general en lugar de una nueva metáfora. Por ejemplo, EL AMOR ES UN VIAJE EN CARRO sería considerada como un caso especial de la metáfora EL AMOR ES UN VIAJE. Herencia múltiple también es posible entre metáforas, llamada *dualismo* por Lakoff. Por ejemplo, EL AMOR ES UN VIAJE hereda tanto de UNA VIDA CON PROPÓSITO ES UN VIAJE y de UNA VIDA CON PROPÓSITO ES UN NEGOCIO, porque los amantes son compañeros en un negocio de dos personas. [Lakoff 1993]

El **principio de invariancia** establece que correspondencias en metáforas preservan la estructura de la experiencia de vida del dominio origen en el dominio destino [Lakoff 1993]. Por ejemplo, lo que es un interior en el dominio origen debe ser mapeado a un interior en el dominio destino, lo que es un exterior es mapeado a un exterior, y lo que es una trayectoria es mapeado a una trayectoria. Por ejemplo, la metáfora MÁS ES ARRIBA está basada en la experiencia de verter un líquido en un contenedor y ver su nivel subir, o agregar más elementos a una pila y ver que incrementan su altura. Esta metáfora es aplicada en objetos como termómetros y gráficos de la bolsa de valores, donde incrementos en la temperatura o de los precios son representados verticalmente ascendentes, y decrementos son representados verticalmente descendentes, nunca lo opuesto. El principio de invariancia no debe ser violado, porque limitaría automáticamente las posibilidades de mapeo [Lakoff 1993].

La teoría de la metáfora conceptual está altamente alineada con la teoría del constructivismo sociocultural, porque ambas declaran la importancia de asociaciones (mapeos) y

conocimiento previo para comprender las metáforas. Ya que las metáforas son asociaciones neurales de un dominio origen (una región del cerebro) a un dominio destino (a otra región) [Lakoff 1993; Lakoff and Johnson 2003], se consideran "poderosos dispositivos de mediación (herramientas psicológicas) para la internalización de los conceptos" (traducido del inglés de [Hung 2002, p.197]). Las metáforas unifican diferentes aspectos de la comunicación humana en una forma condesada y fácil de entender que asocia el discurso interno del individuo (intrasubjetivo) con el discurso externo (intersubjetivo) [Ghassemzadeh 2005]. Las metáforas son especialmente útiles en computación, ya que permiten a las personas asociar los conceptos abstractos e invisibles de la disciplina con conceptos más ordinarios de la experiencia de vida de las personas [Sanford et al. 2014].

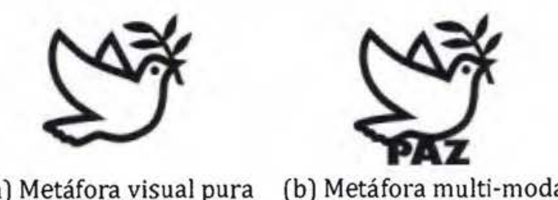
### 2.4.1 Metáforas en computación

El uso de metáforas es ubicuo en el lenguaje natural [Lakoff 1993], y también lo es en la computación como disciplina formal [Colburn and Shute 2008]. Las metáforas en computación han sido estudiadas principalmente en dos campos: educación de la computación e interacción humano-computador. Estudios en ambos campos están fundamentados principalmente en la teoría de metáfora conceptual. En educación de la computación, las metáforas se usan como herramientas de aprendizaje para mapear los conceptos abstractos e invisibles de la disciplina (el dominio destino) con conceptos más concretos y ordinarios (el dominio origen) [Colburn and Shute 2008; Blackwell 2006]. En interacción humano-computador, las metáforas se usan principalmente como herramientas de diseño de interfaces, con el fin de facilitar la comunicación entre el software y sus usuarios finales [Blackwell 2006; Sease 2008], y para incrementar la usabilidad del software [Neale and Carroll 1997].

Varias taxonomías se han propuesto para clasificar metáforas en computación, de acuerdo a su función en la interfaz, la familiaridad con el dominio origen, el tipo de imagen usada como origen, la modalidad y el ámbito [Neale and Carroll 1997; Sanford et al. 2014]. Esta tesis usa la clasificación de metáforas por modalidad y extiende la clasificación por ámbito.

Las metáforas se pueden clasificar por **modalidad** porque no son sólo tropos (figuras literarias), sino una forma de pensar [Lakoff 1993], por tanto el habla no es el único modo en

que se pueden expresar. Las metáforas pueden ser visuales, auditivas, hápticas (táctiles), o verbales [Carroll 1994; Forceville and Urios-Aparisi 2009]. Estas últimas, las metáforas verbales, pueden subdividirse en orales y textuales. Por ejemplo, el sonido de una caja registradora es una metáfora auditiva frecuentemente usada para representar interés económico en un personaje animado. Las metáforas pueden ser uni-modales; por ejemplo la metáfora visual pura de la Figura 2.10(a); o multi-modales, es decir, una combinación de varios modos de comunicación, como la metáfora visual-textual de la Figura 2.10(b) [Heath et al. 2014].




(a) Metáfora visual pura (b) Metáfora multi-modal

Figura 2.10. Dos ejemplos de metáforas clasificadas por modalidad

Sanford et al. clasifican las metáforas por un segundo criterio, el **ámbito** (en inglés, *SCOPE*) definido como el número de características que la metáfora mapea entre el dominio origen y el destino [Sanford et al. 2014]. El ámbito clasifica en *metáforas atómicas* que mapean una única característica entre el dominio origen y el dominio destino, y *metáforas complejas* que mapean varias características [Sanford et al. 2014]. Sin embargo, Sanford et al. no definen precisamente qué es una característica. Por su parte, en la teoría de la metáfora conceptual, una metáfora per se mapea una entidad (metáfora de imagen) o varias entidades (metáfora conceptual) entre ambos dominios. Como se discute adelante, las metáforas en computación cumplen parcialmente la definición de metáfora de Lakoff. Por consiguiente, esta investigación propone extender el criterio de ámbito como se hace en la siguiente sección.

### 2.4.2 Clasificación de metáforas por ámbito

Esta tesis propone extender el criterio de ámbito para distinguir entre metáforas individuales y metáforas múltiples, como se hace en la parte izquierda de la Figura 2.11. Esta propuesta se explica en los siguientes párrafos.

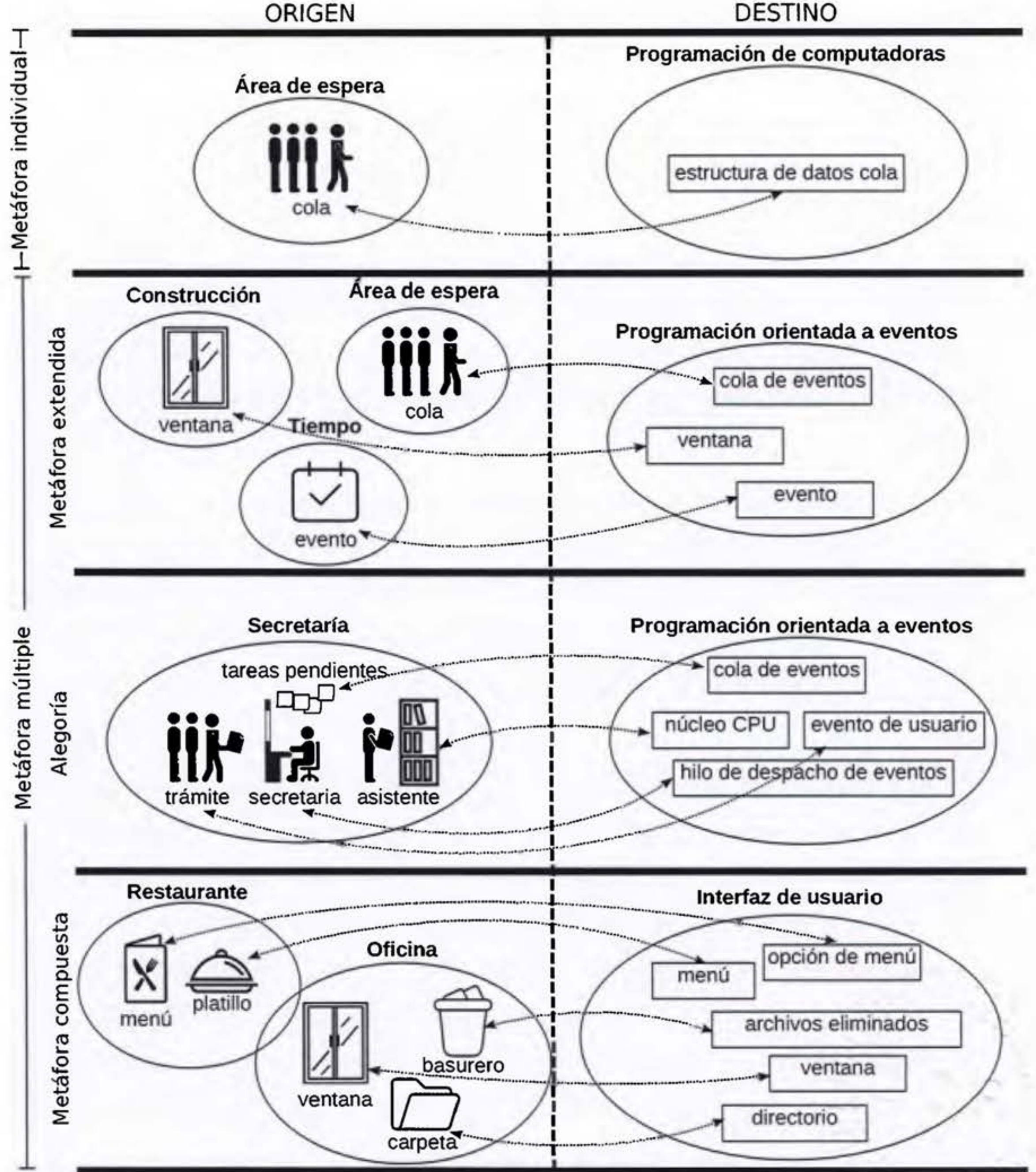



Figura 2.11. Clasificación propuesta de metáforas por ámbito (elaboración propia)

Casi todos los conceptos computacionales son considerados metáforas de conceptos ordinarios; por ejemplo: carpetas, archivos, servidores, clientes, hilos, fugas de memoria, recolectores de basura, pilas, colas, árboles y flujos [Colburn and Shute 2008]. La primera fila de Figura 2.11 muestra la metáfora de cola. En el dominio destino de la programación de computadoras, la cola es un concepto tomado de las áreas de espera para representar una

estructura de datos compuesta de elementos, donde el primer elemento agregado a la cola será el primero en ser removido.

De acuerdo a la clasificación de Lakoff, las metáforas en computación podrían ser consideradas *metáforas de imagen*. Sin embargo, las metáforas computacionales podrían no cumplir con los principios de generalización e invariancia. Por ejemplo, un flujo (en inglés, *STREAM*) en programación de computadoras no puede ser generalizado a una masa de agua o a un fluido físico. Un flujo en programación rompe el principio de invariancia porque no fluye involuntariamente por una fuerza gravitacional, sino que un cursor se mueve a lo largo del flujo, y podría hacerlo hacia atrás (en inglés, *SEEKING*) o incluso empezar de nuevo (en inglés, *REWIND*).

La mayoría de artículos científicos en computación no usan los términos *metáfora de imagen* de Lakoff o *metáfora atómica* de Sanford. La mayoría de autores no consideran los conceptos computacionales como metáforas, ya que son los nombres oficiales para dichos conceptos. Por tanto, en esta investigación se usará el término **metáfora individual** (en inglés, *SINGLE METAPHOR*) para distinguirlas de las metáforas múltiples (véase la parte izquierda de la Figura 2.11).

Debido a la cantidad de metáforas individuales en computación y su diversidad de orígenes, el discurso es típicamente desconectado y sin sentido en el dominio origen. Por ejemplo, la frase "la ventana se congela porque su hilo está ocupado atendiendo el primer evento en la cola" tiene sentido para un programador experimentado de interfaces gráficas, pero es ininteligible para personas que, aunque conozcan el significado de esas palabras en español, no conozcan el dominio destino. Los estudiantes de computación son una población crítica, que debe desarrollar la habilidad de pensar fluidamente en el dominio destino usando las metáforas desconectadas del dominio origen, con el fin de poder formarse en dicha disciplina.

La segunda fila de la Figura 2.11 esquematiza algunos de los componentes de nuestra frase de ejemplo "la ventana se congela porque su hilo está ocupado atendiendo el primer evento en la cola". De acuerdo a la teoría de la metáfora conceptual, esta frase no es estrictamente una expresión metafórica, porque no se deriva de ninguna metáfora. Las entidades orígenes involucradas en esta frase; dígase, ventana, hilo, evento y cola; no comparten el mismo dominio origen. Cada entidad es una *metáfora individual* que proviene de un dominio distinto. Las interrelaciones entre esas metáforas individuales rompen el *principio de invariancia*.

Dichas metáforas individuales fueron forzadas a relacionarse entre sí porque las interrelaciones entre entidades en el discurso destino las requieren de esta forma.

Nuestra frase de ejemplo se compone de varias metáforas individuales. Más aún, de la discusión en el párrafo anterior, la frase no es metafórica porque sus entidades no cumplen la definición de metáfora. Sin embargo, tampoco es literal, porque el lector debe recurrir al pensamiento figurativo para entenderla. Tales frases podrían ser representadas por un dispositivo de pensamiento distinto. Esta investigación propone denominarlas *metáforas extendidas* y *alegorías*.

Una **metáfora extendida** tiene varias cláusulas metafóricas (expresiones) cuyo lenguaje relaciona directamente ambos dominios, el origen y el destino (adaptado de [Crisp 2008]). En esta tesis se propone usar el término *metáfora extendida* para referir el discurso computacional habitual, el cual se compone de varias metáforas individuales conectadas por interrelaciones impuestas por el dominio destino. Esas interrelaciones tienen que hacerse explícitas, por tanto, las interrelaciones destino se mezclan las entidades origen y rompen la coherencia en el dominio origen.

Sin embargo, es posible tener en computación un discurso coherente en el dominio origen, gracias a la versatilidad de las metáforas. Una metáfora conceptual válida, de acuerdo a la teoría de Lakoff, puede mapear un dominio origen arbitrario a metáforas individuales del discurso computacional tradicional. Un ejemplo es parcialmente esquematizado en la tercera columna de la Figura 2.11, donde un dominio origen de una secretaría es usado para explicar el dominio destino de programación orientada a eventos en interfaces gráficas de usuario. Las entidades origen están mapeadas a metáforas destino individuales, por ejemplo, el personal corresponde a núcleos de procesador, una solicitud de un trámite corresponde a un evento de usuario, una persona ocupada atendiendo trámites corresponde a un hilo de ejecución, la secretaria en la ventanilla corresponde al hilo de despacho de eventos (término en inglés *EVENT-DISPATCH THREAD* proveniente del lenguaje de programación *JAVA*), y una lista de solicitudes pendientes anotadas en papeles adhesivos (conocidos comercialmente como *POST-ITS*) corresponde a una cola de eventos.

Las interrelaciones entre las entidades origen (la secretaría en nuestro ejemplo) deben reflejar las interrelaciones entre las entidades computacionales destino (programación orientada a eventos) acorde al principio de invariancia. El siguiente es un ejemplo del discurso

origen para la metáfora en la tercera fila de la Figura 2.11: "La secretaria en la ventanilla recibe solicitudes de los usuarios, y ella las anota inmediatamente en papeles adhesivos. Cuando la secretaria está ociosa, ella realiza el primer trámite pendiente anotado en sus papeles adhesivos. Mientras la secretaria está ocupada haciendo un trámite, ella desatiende la ventanilla..." Nótese que esta vez el discurso origen es comprensible para hispanohablantes sin conocimiento del dominio destino, siempre y cuando tengan nociones sobre una secretaría. El discurso anterior es coherente con el siguiente discurso destino "El hilo de despacho de eventos recibe eventos del usuario, y los agrega inmediatamente a la cola de eventos. Cuando el hilo de despacho de eventos queda ocioso, atiende el primer evento pendiente en la cola de eventos. Mientras el hilo de despacho de eventos está ejecutando una tarea, no atiende la interfaz de usuario".

Dado que se tienen dos discursos paralelos compuestos de metáforas individuales, esta tesis propone usar el término alegoría para denominarlas. Una **alegoría** es una metáfora súper-extendida, cuyo significado en el dominio origen es independiente, autónomo, y no obviamente metafórico [Crisp 2008; Crisp 2001]. Tanto las metáforas extendidas como las alegorías son dispositivos cognitivos desarrollados a lo largo de un pasaje o narrativa (discurso), y ambas tienen un significado literal superficial y un significado figurativo profundo [Rubio-Fernández et al. 2016; Carston and Wearing 2011]. Sin embargo, las metáforas extendidas pueden hacer explícito tanto el discurso origen como el destino, mientras que las alegorías sólo hacen explícito el dominio origen [Crisp 2008]. Por tanto, las alegorías requieren tener un discurso origen que mapea coherentemente el discurso destino elidido para que ambos puedan tener significado [Hidalgo-Céspedes et al. 2016a].

El término alegoría es raramente usado en la literatura computacional (ej.: [Cafaro et al. 2014; Hidalgo-Céspedes et al. 2016a]). Los autores han usado términos similares para referirse a su dispositivo cognitivo subyacente, tales como:

- *Metáfora explicativa* (en inglés, *EXPLANATORY METAPHOR*). Por ejemplo [Carroll and Mack 1985; Blackwell and Green 1999; Blackwell 2006; Hsu 2006]).
- *Metáfora integral* (en inglés, *INTEGRAL METAPHOR*). Por ejemplo [Smilowitz 1995; Hsu and Schwen 2003]).
- *Metáfora simple* (en inglés, *SIMPLE METAPHOR*). Por ejemplo [Hsu and Schwen 2003].

La mayoría de autores simplemente usan el término *metáfora* para referirse a las alegorías, y el término no-metáfora para referirse al discurso computacional tradicional, constituido de metáforas individuales. Por ejemplo [Hsu 2006; Hsu 2007].

La última fila de la Figura 2.11 incluye en la clasificación por ámbito a la metáfora compuesta. Una **metáfora compuesta** (en inglés, *COMPOSITE METAPHOR*) es la unión de dos o más alegorías cuyos dominios origen son distintos [Smilowitz 1995; Hsu 2005]. Usualmente una alegoría es la primaria o fundamental (en inglés, *UNDERLYING*), y las otras son metáforas secundarias o auxiliares [Cates 2002]. Debido a que el discurso origen se rompe a causa de distintos dominios, la metáfora compuesta es un caso especial de una metáfora extendida. En la clasificación propuesta se incluyó la metáfora compuesta porque ha sido la figura teórica que explica la metáfora de interfaz de usuario comercial más popular: la metáfora de escritorio. El dominio origen de la metáfora de escritorio es una oficina, la cual ha sido mezclada con otros dominios, tales como restaurantes (menús) o navíos (navegar).

El término *metáfora múltiple* ha sido usado como sinónimo de metáfora compuesta (ej.: [Hsu and Schwen 2003]). Sin embargo, en esta tesis se prefiere usar el término **metáfora múltiple** (en inglés, *MULTIPLE METAPHOR*) como un reemplazo de metáfora compleja en la clasificación por ámbito de Sanford. Esta decisión está basada en nuevos estudios que distinguen *metáforas complejas* de *metáforas primarias*. Una metáfora primaria (en inglés, *PRIMARY METAPHOR*) está "directamente basada en la experiencia cotidiana que conecta nuestra experiencia sensorial-motora al dominio de los juicios subjetivos", por ejemplo, AFECTO ES CALIDEZ (adaptado del inglés de [Lakoff and Johnson 2003, p.255]). Una **metáfora compleja** (en inglés, *COMPLEX METAPHOR*) está "compuesta de metáforas primarias que hacen uso de marcos conceptuales socialmente construidos" (adaptado del ingles de [Lakoff and Johnson 2003, p.257]). Por tanto, las metáforas complejas "pueden variar significativamente de una cultura a otra" (adaptado del inglés de [Lakoff and Johnson 2003, p.257]).

Si se combina el criterio de modalidad (verbal (textual/oral), visual, auditiva, háptica, multi-modal) con el criterio de ámbito (metáfora individual, metáfora extendida, alegoría, metáfora compuesta), varias combinaciones emergen, tales como, metáfora visual extendida, alegoría visual, y metáfora verbal. Para esta investigación, las metáforas y alegorías visuales son de especial interés, dado que las visualizaciones de programa las emplean para representar los conceptos abstractos de programación.

### 2.4.3 Metáforas y alegorías visuales

Una **metáfora visual** es una metáfora cuyas sus entidades destino son representadas mediante entidades origen visuales (adaptado de [Heath et al. 2014]). Por ejemplo, la imagen de una paloma blanca con una rama de olivo en el pico (Figura 2.10(a), p. 50) es una metáfora visual del concepto destino "paz" [Heath et al. 2014]. El uso de metáforas visuales es muy difundido en nuestra cultura, en señalización vial, logotipos de productos o de empresas, y servicios públicos (baños, teléfonos), entre otros.

Una **metáfora visual individual** puede considerarse en la teoría de Lakoff como un caso de una metáfora de imagen. Si la metáfora consta de varias entidades origen visuales que corresponden a entidades destino que pueden estar o no presentes (usualmente usando otros modos), se trata de una **metáfora visual extendida**. Por ejemplo, una mezcla de gráficos con textos para explicar un programa en ejecución. Si sólo el discurso visual origen está presente explícitamente y las interrelaciones entre las entidades visuales reflejan el discurso destino elidido, se trata de una **alegoría visual**. Un ejemplo es la "Alegoría del café y el banano", pintada en 1897 por el italiano Aleardo Villa en el techo del Museo Nacional de Costa Rica, y que en 1968 fue reproducida en el reverso del billete de cinco colones (Figura 2.12)[11]. La alegoría refleja la época agroexportadora de Costa Rica (1821-1963)[12], donde la economía del país creció gracias a la exportación de café y banano. La pintura muestra de derecha a izquierda a través de metáforas un proceso desde, el cultivo del café (plantas de café) y del banano (hojas de banano), su recolección (mujeres con canastos), transporte (una yunta de bueyes), empaque (sacos y cajas), hasta su exportación (barcos en la costa y el mar).

Figura 2.12. Alegoría al café y al banano en el billete de 5 colones (fuente: monedasybilletes.com.ar)

[11] https://es.wikipedia.org/wiki/Alegoría_del_café_y_el_banano

[12] Carvajal-Alvarado, Guillermo. *El pintoresco billete de 5 colones de 1968.* http://www.ticovision.com/cgi-bin/index.cgi?action=viewnews&id=14838

### 2.4.4 Limitaciones de las metáforas

Dado que los conceptos de programación son abstractos y no sensoriales, las visualizaciones de programa tienen ineludiblemente que recurrir a metáforas visuales para representarlos. Hay, sin embargo, riesgos del uso de metáforas en el proceso de aprendizaje que deben tenerse presentes, como los dos siguientes.

1. El origen no puede representar todas las características del concepto computacional destino [Sanford et al. 2014]. Por tanto, el estudiante podría escoger las características incorrectas para establecer el mapeo. Se ha recomendado hacer explícito no sólo las entidades origen, sino también las correspondencias que conforman el mapeo.
2. El origen de la metáfora debe formar parte del sistema de conceptos en la mente del estudiante, y tener significado para él.

La investigación sobre metáforas en computación se ha hecho casi de forma exclusiva sobre metáforas extendidas. Los trabajos en alegorías son escasos y esta tesis hace un aporte al conocimiento sobre este concepto. En la siguiente sección se presentará resumidamente el concepto de ludificación, el cual es parte estructural, junto con las alegorías visuales, de la propuesta conceptual del capítulo 4.

## 2.5 Ludificación

De acuerdo a Kapp, las personas tienden a involucrarse en actividades lúdicas con tasas mayores de motivación intrínseca y compromiso que en actividades "serias" como la educación y el ejercicio laboral. La **ludificación** (en inglés, *GAMIFICATION*) es la aplicación cuidadosa y considerada de los *elementos del juego* o *elementos lúdicos* a actividades que no se consideran juegos, con el fin de incrementar la motivación, el compromiso y la calidad de los resultados de las personas en esas actividades "serias". Algunos ejemplos de ludificación son videojuegos para motivar a las personas a mantener una rutina de ejercicios físicos, videojuegos para aprender a maniobrar equipo especializado (como un montacargas) sin lastimar a alguien, juegos como complemento a una clase magistral, y aprender sobre liderazgo en un juego de roles masivo en línea. [Kapp 2012]

Un **juego** es una actividad realizada por jugadores involucrados en un reto abstracto; definido por un sistema de reglas de interacción y realimentación; con el fin de producir resultados físicos, cognitivos y emocionales verificables (adaptado de [Kapp 2012]). Los investigadores estudian qué características tienen los juegos que motivan y comprometen a las personas. Kapp llama a estas características **elementos del juego** o **elementos lúdicos** y son de interés porque pueden aplicarse a las actividades serias. [Kapp 2012] distingue 12 elementos de juego que se resumen en la siguiente lista. Una descripción más detallada se puede encontrar en el anexo 1.

1. *Abstracción de conceptos y realidad.* Los juegos no son la realidad, sino que abstrae o elimina detalles de la realidad para permitir al jugador concentrarse en los principios que se pretenden aprender o desarrollar en el juego.
2. *Objetivos.* Los juegos tienen objetivos específicos no ambiguos que dan propósito al juego, y permiten fácilmente a los jugadores saber si este propósito se ha alcanzado o si han progresado hacia él.
3. *Reglas.* Las reglas son restricciones para limitar las acciones de los jugadores con el fin de mantener al juego manejable, aunque deben otorgar alguna libertad a los jugadores de alcanzar los objetivos a través de métodos diversos.
4. *Conflicto, competición o cooperación.* Un juego plantea una situación de reto que impide el progreso de los jugadores llamada conflicto. Para superar el conflicto, los jugadores deben competir, cooperar o una mezcla de ambas.
5. *Tiempo.* Los juegos utilizan el tiempo para incrementar el nivel de estrés en los jugadores y la necesidad de optimizar acciones para alcanzar el objetivo del juego. Es un valioso recurso en una ludificación si también lo es en la actividad seria.
6. *Reconocimiento.* Los juegos recompensan a los jugadores por su progreso en alcanzar los objetivos con puntos, insignias, o reconocimiento social (tablas de marcadores), entre otros. Se puede permitir al jugador canjear estos premios por beneficios que le ayudan en su progreso o aprendizaje.
7. *Realimentación.* Los juegos ofrecen al jugador realimentación en tiempo real de su progreso hacia el objetivo. Informa de acciones correctas o erróneas al aprendiz, pero no cómo corregir una acción errónea.
8. *Niveles.* Los juegos utilizan niveles para dar estructura y evitar que el jugador vague sin rumbo por el juego. En cada nivel el jugador cumple un pequeño conjunto de objetivos,

antes de pasar al siguiente. Los niveles se ordenan de tal forma que cada nivel agrega un trozo a un relato general, introduce conocimientos y habilidades o fortalece los desarrollados en niveles previos, e incrementa ligeramente el nivel de dificultad. Se puede proveer niveles fáciles, intermedios y difíciles, e ir revelándolos al jugador de acuerdo a su progreso para evitar aburrimiento o frustración.

9. *Narración*. La narración provee contexto con el fin de darle relevancia y significado al juego, y puede involucrar al jugador. Es muy útil en actividades educativas, en especial para ayudar a la memoria, y a comprender los objetivos del juego.
10. *Curva de interés*. Es el curso de eventos que mantiene al jugador interesado en el juego a través del tiempo. Es útil medirla en las evaluaciones del juego.
11. *Estética*. La estética abarca el arte, los detalles y la belleza de los componentes del juego para comunicar claramente un mensaje. Sin belleza estética, un juego tiene alta probabilidad de ser considerado como aburrido.
12. *Reintento*. A diferencia de una actividad seria, en los juegos el fallo es aceptado e incentiva la exploración, la curiosidad y el aprendizaje por descubrimiento.

La ludificación de una actividad seria no es una tarea trivial [Kapp 2012]. El éxito de la ludificación no radica en la implementación de algunos elementos lúdicos separados, sino en la interrelación de dichos elementos [Kapp 2012].

Esta tesis conjetura que el uso de ludificación junto con alegorías visuales pueden contribuir en la eficacia de las visualizaciones de programa. En el siguiente capítulo se recabará la lista de visualizaciones de programa existentes y en ellas se analizará el nivel de soporte de alegorías y ludificación.

# 3 ANTECEDENTES Y ANÁLISIS DEL ESTADO DEL ARTE

En este capítulo, además de incluir los antecedentes, se documenta el proceso para alcanzar el objetivo específico 1 de esta investigación: "Identificar requerimientos de software de alto nivel, a partir de una teoría de aprendizaje, que no son atendidos por las visualizaciones existentes de programa". Los métodos para alcanzar este objetivo se encuentran numerados en la Figura 3.1. En la sección 3.1 se actualiza una revisión de literatura que recaba la lista de visualizaciones de programa activas y que puedan ser comparadas ①. Los requerimientos de software que se extrajeron de la teoría de constructivismo sociocultural en el marco teórico ② , se listan en la sección 3.2. En la sección 3.2 se reporta una verificación que determina el grado en que los requerimientos de software son satisfechos por las visualizaciones de programa disponibles ③. De esta verificación se obtiene la lista de requerimientos no atendidos que son el producto de este objetivo.

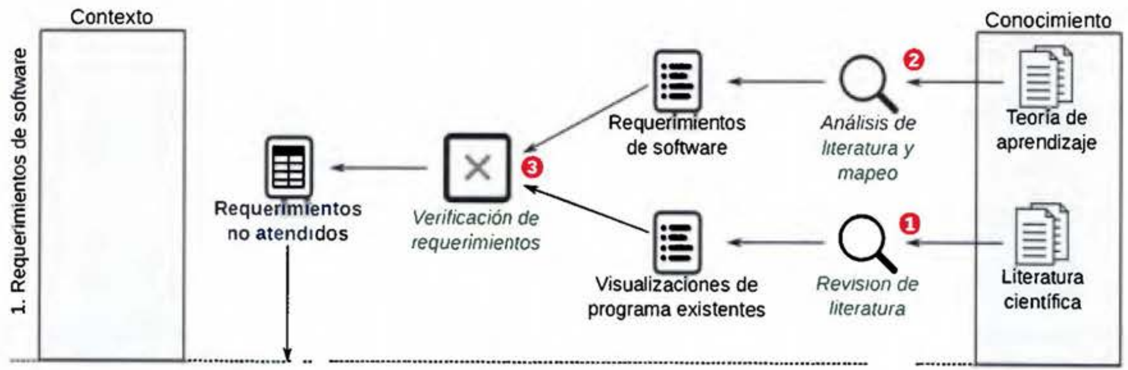



Figura 3.1. Metodología del objetivo específico 1

Como un resultado interesante de esta primer revisión de literatura, no se encontró alguna visualización de programa que implemente alegorías visuales. Sin embargo, se encontraron advertencias de efectos negativos de alegorías en interfaces de usuario [Hsu 2006; Blackwell 2006]. Antes de continuar con la investigación inicialmente planteada, se determinó vital confirmar estas advertencias en otros estudios y en los estudiantes de la ECCI, ya que las alegorías forman parte de la propuesta conceptual de esta tesis. Para confirmar las advertencias mencionadas, se realizó una segunda revisión de literatura sobre alegorías en el campo de la computación, no limitada a visualizaciones de programa (sección 3.4), y se condujo un experimento que comparó el efecto de las alegorías y las metáforas tradicionales

en la resolución de problemas de estudiantes de la ECCI (sección 3.5). Los resultados de la revisión y los del experimento no confirmaron las advertencias negativas sobre las alegorías.

## 3.1 Visualizaciones de programa existentes

El trabajo previo a esta investigación corresponde a visualizaciones de programa utilizadas en el ambiente de la enseñanza o aprendizaje de la máquina nocional de un lenguaje de programación. Existen algunas revisiones de literatura no exhaustivas sobre visualizaciones de algoritmo, visualizaciones de código, o visualizaciones de software en general, tales como [Myers 1990], [Price et al. 1993], [Maletic et al. 2002], [Urquiza-Fuentes and Velázquez-Iturbide 2009], y [Xinogalos 2013]. Sorva, Karavirta y Malmi aseguran que su estudio es la primera revisión de literatura exhaustiva sobre visualizaciones de programa [Sorva et al. 2013]. Tal estudio encontró 46 sistemas entre 1979 y 2012, inclusive. De 2013 al 2016 no se encontró ninguna otra revisión de literatura. Por tanto, durante el trabajo de tesis se extendió la revisión de literatura de [Sorva et al. 2013] en el período comprendido entre 2013 al primer cuatrimestre de 2016. El resultado fue presentado a la conferencia *IEEE FRONTIERS IN EDUCATION 2016* (*FIE*) [Hidalgo-Céspedes et al. 2016a].

[Sorva et al. 2013] utilizó una definición más general de visualización de programa, que abarca visualización de código, animación de programa, programación visual, y simulación visual de programa. Pese a que su definición es más general su revisión incluye, casi en su totalidad, herramientas educativas que representan visualmente el comportamiento en tiempo de ejecución de programas escritos en un lenguaje de propósito general [Sorva et al. 2013, p.8], lo cual coincide con la definición más delimitada de visualización de programa utilizada en esta tesis.

El recabado de la lista de visualizaciones de programa activas se hizo en dos fases. En la primera fase se recorrió cada uno de los 46 sistemas que fueron reportados activos por [Sorva et al. 2013] y se trató de corroborar si se mantenían en este estado, mediante la publicación de nuevos artículos, el mantenimiento de algún sitio web, o algún repositorio de software. En la segunda fase se realizó una revisión sistemática de literatura para obtener la lista de

visualizaciones de programa aparecidas en el período 2013 a 2016 cuya metodología se explica a continuación.

### 3.1.1 Metodología de la revisión de literatura

Con el fin de encontrar nuevas visualizaciones de programa en el período 2013 a 2016, se condujo una revisión sistemática de literatura siguiendo la metodología recomendada en [Wohlin et al. 2012]. Los artículos se buscaron en cuatro bases de datos: *ACM DIGITAL LIBRARY*, *IEEE XPLORE*, *WEB OF SCIENCE*, y *SCOPUS*. Las dos primeras son relevantes debido a que en ellas se encuentran publicados los artículos de control sobre visualizaciones de programa ya conocidas. Las últimas dos bases de datos fueron incluidas porque indexan las revistas más prestigiosas en computación.

Dado que la terminología para referirse a las visualizaciones de programa varía de acuerdo a los autores, se incluyeron varios términos afines en el texto de búsqueda (en inglés, *SEARCH STRING*), y se limitaron al contexto educativo (enseñanza o aprendizaje). El Listado 3.1 muestra el texto de búsqueda empleado:

```
("program visualization" OR
 "program visualisation" OR
 "program animation" OR
 "software visualization" OR
 "software animation" OR
 "visual debugger") AND
(learn* OR teach* OR educat*) AND
publication year on or after 2013
```

Listado 3.1. Texto de búsqueda de artículos sobre visualizaciones de programa

Todos los resultados obtenidos por el texto de búsqueda fueron revisados por el autor de esta tesis. Para decidir si un artículo debía ser incluido o excluido, los criterios del Cuadro 3.1 fueron cotejados en el título, palabras claves y resumen de los artículos. Si esos campos fueron insuficientes, se cotejó en el texto completo. Las secciones de referencias en los artículos seleccionados fueron también analizadas para detectar potenciales artículos faltantes, lo que se conoce como proceso de bola de nieve (en inglés, *SNOWBALL PROCESS*). Cada artículo que resultó incluido, fue leído con el fin de detectar nuevas visualizaciones de programa entre 2013 y 2016.

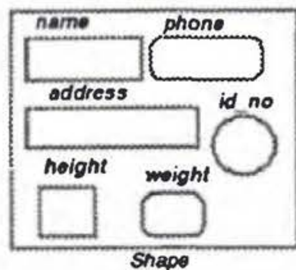
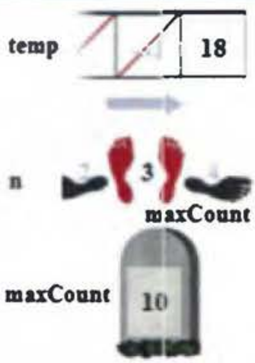
Cuadro 3.1. Criterios de inclusión y exclusión de artículos

| | |
|---|---|
| Inclusión | 1. El artículo introduce una nueva visualización de programa (entre 2013 y 2016)<br>2. El artículo usa una visualización de programa introducida en otros artículos<br>3. El artículo relaciona varias visualizaciones de programa (ej.: una revisión de literatura)<br>4. El artículo trata sobre visualizaciones de programa en general, no refiere a una en particular<br>5. El artículo trata sobre un visualización de programa especializada<br>6. El artículo no trata de una visualización de programa, pero usa alegorías visuales o ludificación |
| Exclusión | 1. El documento no es un artículo completo (es un resumen, póster, o taller)<br>2. El artículo no está escrito en inglés<br>3. El texto completo del artículo no estaba disponible (en el proceso de bola de nieve)<br>4. El artículo está duplicado (por tanto, presente en más de una base de datos)<br>5. El artículo ya estaba incluido en la revisión de literatura previa ([Sorva et al. 2013])<br>6. La herramienta es una visualización de algoritmo<br>7. La herramienta es una visualización de código<br>8. La herramienta no es una "visualización de programa" como está definida en esta tesis<br>9. El artículo no está relacionado con la temática (visualización, educación de la programación) |

En cada visualización de programa encontrada se analizaron tres variables afines a la hipótesis de investigación de esta tesis: ámbito de metáforas visuales (alegorías o metáforas extendidas), tipo de origen visual (concreto o abstracto), y soporte de ludificación. El Cuadro 3.2 explica las cuatro combinaciones posibles de las dos primeras variables con ejemplos encontrados en visualizaciones existentes de programa. Este análisis es análogo al hecho en [Sorva et al. 2013] con las variables de compromiso e interacción.

Las tres variables de interés (ámbito, origen y ludificación) se recabaron tanto para las *nuevas* visualizaciones encontradas entre 2013 y 2016, como para las visualizaciones de programa *activas* listadas en [Sorva et al. 2013]. Un sistema se consideró **activo** si tiene una actividad reciente, como nuevas publicaciones, nuevas versiones del software, o cambios en repositorios de control de versiones. Un sistema está **disponible** si el software puede ser descargado y probado, lo cual fue necesario para la verificación de requerimientos presentada más adelante en este capítulo.

Cuadro 3.2. Tipos de metáforas visuales usadas en visualizaciones existentes de programa

| | **Metáfora visual extendida** | **Alegoría visual** |
|---|---|---|
| **Origen abstracto** | **Metáfora visual extendida abstracta.** Usa elementos visuales abstractos para representar conceptos, también abstractos, de programación. Por ejemplo, Jeliot 3 en la figura de arriba, representa las variables y las invocaciones de funciones con rectángulos, y las referencias con flechas [Moreno and Myller 2003]. No existe un discurso coherente entre los elementos visuales (figuras geométricas) que reflejen las interrelaciones entre los conceptos de programación. Es el tipo de metáfora visual más empleado por las visualizaciones de programa. |  **Alegoría visual abstracta.** Usa elementos visuales abstractos para representar los conceptos abstractos de programación. Existe un discurso coherente entre los elementos visuales que explica la interrelación entre los conceptos de programación. Por ejemplo, [Waguespack 1989] representó variables enteras con rectángulos redondeados y reales con rectángulos de esquinas cuadradas, con el fin de resaltar que un valor entero cabe en uno real de la misma forma que un rectángulo redondeado cabe en uno con esquinas cuadradas, mientras que el recíproco produce pérdida información (decimales). |
| **Origen concreto** |  **Metáfora visual extendida concreta.** Usa elementos visuales concretos (objetos físicos) para representar los conceptos abstractos de programación. Por ejemplo, *PlanAni* en la figura de arriba, utiliza huellas (pasos) para representar contadores y lápidas para valores constantes [Sajaniemi and Kuittinen 2003][Kuittinen and Sajaniemi 2004]. No existe un discurso coherente entre los elementos visuales que refleje interrelaciones entre los conceptos de programación. | **Alegoría visual concreta.** Usa elementos visuales concretos para representar los conceptos abstractos de programación. Existe un discurso coherente entre los elementos visuales que refleja las interrelaciones entre los conceptos de programación. No se encontraron visualizaciones de programa que implementen este tipo de metáfora visual. |

### 3.1.2 Resultados de la revisión de literatura

El texto de búsqueda recuperó 169 resultados entre las cuatro bases de datos seleccionadas. Se descartaron 44 resultados duplicados. Después de aplicar los criterios de selección y exclusión (Cuadro 3.1) a los restantes 125 resultados, 36 artículos entre 2013 y el primer cuatrimestre de 2016 pasaron los criterios de inclusión, y por tanto, a la fase de revisión de texto completo. En el período 2013-2016, siete nuevas visualizaciones de programa fueron encontradas y se resaltan en fondo gris en el Cuadro 3.3. Este cuadro incluye también las visualizaciones de programa aún activas o disponibles, inicialmente recabadas en [Sorva et al. 2013]. En las filas se encuentran las visualizaciones de programa ordenadas por la fecha de aparición (columna "Desde").

Cuadro 3.3. Visualizaciones de programa activas (parte 1)

| # | Nombre | Desde | Referencias/URL |
|---|---|---|---|
| 1 | *GRASP / JGRASP* | 1996 | http://www.jgrasp.org/ |
| 2 | *THE TEACHING MACHINE* | 2000 | http://www.theteachingmachine.org/ |
| 3 | *PLANANI* | 2002 | http://www.cs.uef.fi/~saja/var_roles/planani/index.html |
| 4 | *JIVE* | 2002 | http://www.cse.buffalo.edu/jive/ |
| 5 | *JELIOT* | 2003 | http://cs.joensuu.fi/jeliot/ |
| 6 | *VIP* | 2005 | http://www.cs.tut.fi/~vip/en/ |
| 7 | *VILLE* | 2005 | https://ville.cs.utu.fi/old/ |
| 8 | *METAPHOR-BASED OO VISUALIZER* | 2007 | http://www.cs.uef.fi/~saja/oo_metaphors/index.html |
| 9 | *UUHISTLE* | 2009 | http://www.uuhistle.org/ |
| 10 | *ONLINE PYTHON TUTOR* | 2010 | http://www.pythontutor.com/ |
| 11 | *JELIOT CONAN* | 2013 | [Moreno et al. 2013; Moreno et al. 2014] https://cs.joensuu.fi/jeliot/ |
| 12 | *BLUEJ NOVIS* | 2013 | [Berry and Kölling 2013; Berry and Kölling 2014; Berry and Kölling 2016] http://bluej.org/novis.zip |
| 13 | *SEEC* | 2013 | [Egan and McDonald 2013; Egan and McDonald 2014] http://seec-team.github.io/seec/ |
| 14 | *VIRTUAL-C IDE* | 2014 | [Pawelczak and Baumann 2014] https://sites.google.com/site/virtualcide/ |
| 15 | *PROVIT* | 2014 | [Yan et al. 2014] http://provit.u-aizu.ac.jp/ |
| 16 | *FM VISUALIZATION* | 2014 | [Al-Fedaghi and Alrashed 2014] |
| 17 | *JAVELINACODE* | 2015 | [Yang et al. 2015] http://javelina-cc.sourceforge.net/ |

usadas del todo. De acuerdo a la columna "Origen metáfora", todas las visualizaciones de programa usan imágenes abstractas para representar conceptos de programación, excepto dos sistemas: *PLANANI* [Sajaniemi and Kuittinen 2003] y *METAPHOR-BASED OO VISUALIZER* [Sajaniemi et al. 2007]. Ambas visualizaciones fueron creadas por el mismo grupo de investigación. Se distinguen por usar imágenes concretas, la primera para representar roles de variables, y la segunda para conceptos de programación orientada a objetos.

La columna "Lúdica" muestra que sólo una visualización de programa activa o disponible usa elementos de ludificación. Esta herramienta se discute en la próxima sección. Dado que sólo visualizaciones activas o disponibles se listan en el Cuadro 3.3 y el Cuadro 3.4, al menos una de las columnas "Activa" o "Disponible" debe tener un valor "Sí". Las herramientas nuevas, en fondo oscuro, se consideraron activas dado sus publicaciones recientes. Herramientas inactivas reportadas en [Sorva et al. 2013] no se replicaron en el Cuadro 3.4.

En [Sorva et al. 2013] se incluye una pequeña reseña de cada visualización de programa. Esta reseña no se repite en esta tesis, sino que se realiza sólo para *JELIOT CONAN* por el aspecto de ludificación, y para las dos nuevas visualizaciones disponibles encontradas en el lapso 2012 a 2016.

#### 3.1.2.1 JELIOT CONAN

*JELIOT CONAN* es una variante de *JELIOT 3* que crea *animaciones conflictivas* (Figura 3.2). Cuando un programa corre en *JELIOT CONAN*, algunos pasos serán visualizados incorrectamente de forma automática. El objetivo de las animaciones conflictivas es estimular la atención e intriga de los estudiantes [Moreno et al. 2013]. *JELIOT CONAN* es usado en ambientes lúdicos como competencias. Para ganar los estudiantes en equipos deben detectar tantos fallos como puedan, en programas defectuosos creados por otros equipos [Moreno et al. 2013]. Cuando un fallo es detectado, el botón "*FAULT*" debe presionarse en la interfaz de *JELIOT CONAN* (Figura 3.2).

El Cuadro 3.4 continúa al Cuadro 3.3 para agregar otros datos de interés en esta investigación. La columna "Máquina nocional" indica la o las máquinas nocionales visualizadas. De ella se puede inferir que *JAVA* y C son los dos lenguajes de programación más apoyados por nuevos desarrollos.

Cuadro 3.4. Lista de visualizaciones de programa activas (parte 2)

| # | Nombre | Máquina nocional | Plata-forma | Ámbito | Origen metáfora | Lú-dica | Act iva | Dispo nible |
|---|---|---|---|---|---|---|---|---|
| 1 | *GRASP / JGRASP* | *JAVA* | *JAVA* | Extendida | Abstracta | No | Sí | Sí |
| 2 | *THE TEACHING MACHINE* | C++, *JAVA* | Web | Extendida | Abstracta | No | Sí | Sí |
| 3 | *PLANANI* | C, *JAVA*, Pascal, *PYTHON* | *WIN* | Extendida | Concreta | No | Sí | Sí |
| 4 | *JIVE* | *JAVA* | Eclipse | Extendida | Abstracta | No | Sí | Sí |
| 5 | *JELIOT* | *JAVA* | *JAVA* | Extendida | Abstracta | No | Sí | Sí |
| 6 | *VIP* | C++ | *JAVA* | Extendida | Abstracta | No | Sí | Sí |
| 7 | *VILLE* | *JAVA*/C++/ *PYTHON*/*JS*/PHP | *JAVA*, Web | Extendida | Abstracta | No | Sí | Sí |
| 8 | *METAPHOR-BASED OO VISUALIZER* | Pascal | Web (Flash) | Extendida | Concreta | No | No | Sí |
| 9 | *UUHISTLE* | *PYTHON* | *JAVA* | Extendida | Abstracta | No | Sí | Sí |
| 10 | *ONLINE PYTHON TUTOR* | *PYTHON*/*JAVA* C/C++/*JS*/... | Web | Extendida | Abstracta | No | Sí | Sí |
| 11 | *JELIOT CONAN* | *JAVA* | *JAVA* | Extendida | Abstracta | Sí | Sí | No[13] |
| 12 | *BLUEJ NOVIS* | *JAVA* | *WIN*, *MAC*, *ANDROID* | Extendida | Abstracta | No | Sí | No[14] |
| 13 | *SEEC* | C | *WIN*, *LIN*, Mac | Extendida | Abstracta | No | Sí | Sí |
| 14 | *VIRTUAL-C IDE* | C | *WIN*, *MAC*, *ANDROID* | Extendida | Abstracta | No | Sí | Sí |
| 15 | *PROVIT* | C | Web | Extendida | Abstracta | No | Sí | No |
| 16 | *FM VISUALIZATION* | C | *WIN*, *LIN*, *MAC* | Extendida | Abstracta | No | Sí | No |
| 17 | *JAVELINACODE* | *JAVA* | Web | Extendida | Abstracta | No | Sí | No |

De la columna "Ámbito" en el Cuadro 3.4, es claro que todas las visualizaciones de programa activas o disponibles usan metáforas visuales inconexas (metáforas extendidas) para representar los conceptos de programación. En otras palabras, las alegorías visuales no son

[13] Se dispone de Jeliot3, pero no incluye el agregado de animaciones conflictivas.
[14] El prototipo disponible para descarga no se encuentra en un estado funcional para visualizar.

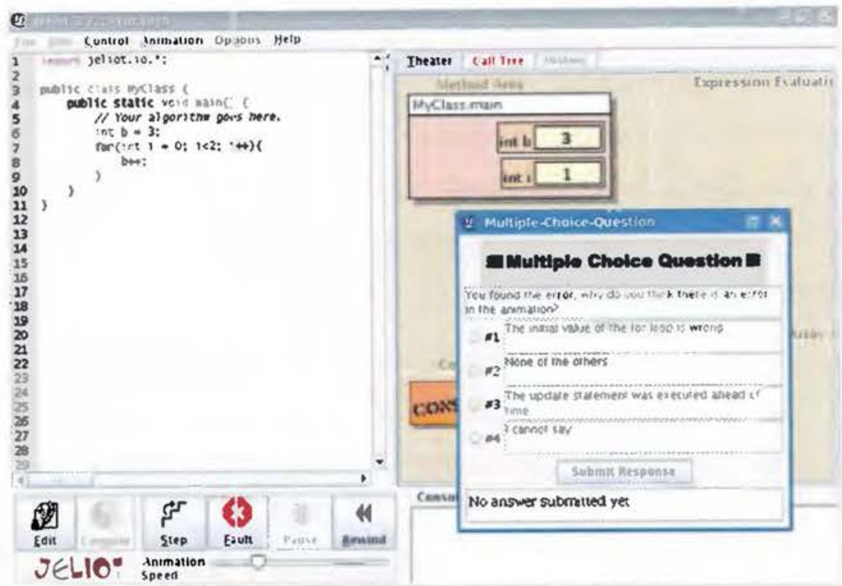


Figura 3.2. Captura de pantalla de *JELIOT CONAN*. Fuente: [Moreno et al. 2010, p.146]

### *3.1.2.2 SeeC*

*SEEC* (pronunciado '*SEEK*' en inglés) es un depurador visual ideado para aprendices de programación que implementa depuración reversible y detección automática de errores [Egan and McDonald 2013, p.4]. La depuración reversible (en inglés *REVERSIBLE DEBUGGING*, ó *TRACE-BASED DEBUGGING*) es la capacidad de "regresar en el tiempo". *SEEC* es realmente una adaptación del compilador de C de *CLANG*[15]. Cuando un programa en C se compila con *SEEC*, genera un archivo ejecutable distinto al usual.

El ejecutable contiene un generador de animaciones. Cuando el ejecutable es lanzado, actúa con normalidad, por ejemplo, pregunta por valores de variables en la entrada estándar e imprime resultados en la salida estándar. Una vez que el ejecutable termina de correr genera un archivo con una animación, similar a un video. Cuando este archivo se abre en el visor de *SEEC*, visualizará la ejecución del programa del estudiante con que se generó, de forma similar en que se visualiza un video (Figura 3.3). El estudiante puede controlar la ejecución de la animación, como retroceder o pasar a la próxima instrucción. Pero no puede interaccionar con los elementos de la animación, o modificar el código fuente. Si quiere hacerlo, deberá volver a correr el programa generador de animaciones. [Egan and McDonald 2013].

[15] https://clang.llvm.org/

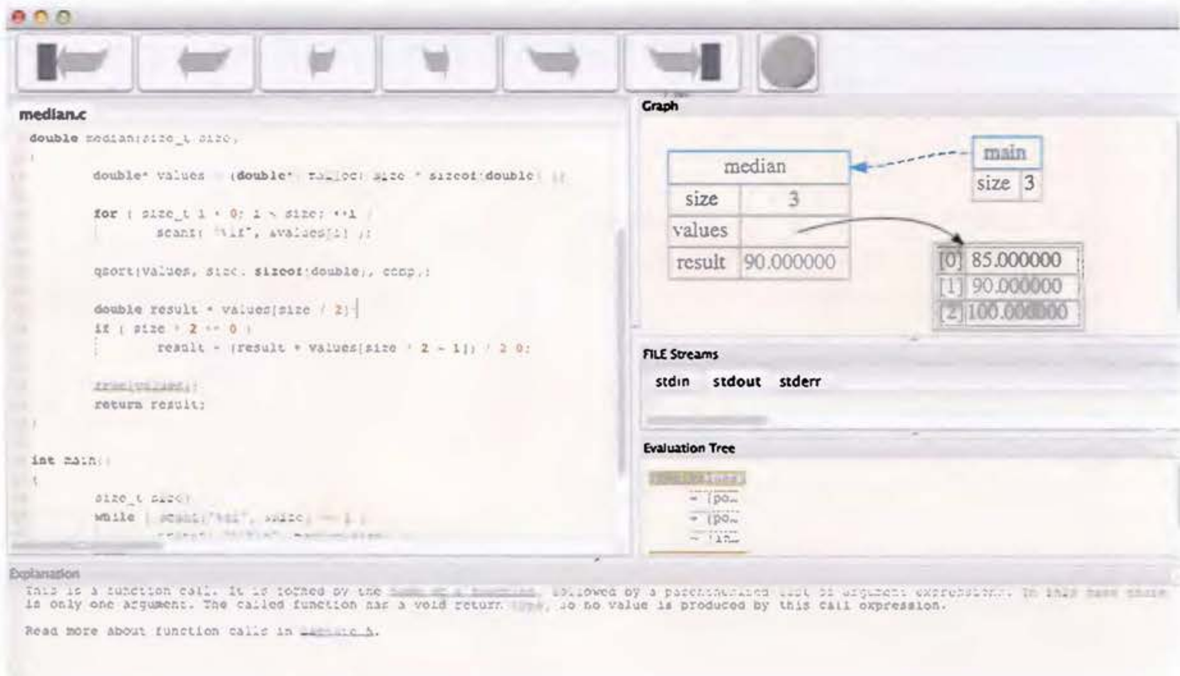



Figura 3.3. Captura de pantalla de SeeC visualizando el programa de la mediana

### *3.1.2.3 Virtual-C IDE*

Virtual-C IDE es un ambiente de desarrollo integrado elaborado a partir del año 2012 por la Universidad Bundeswehr de Múnich, Alemania, con varios objetivos en mente: visualizar el control de flujo y datos de los programas; facilitar la programación gráfica, de videojuegos y web; evaluar y detectar plagio automáticamente; e incrementar la tasa de aprobación en su curso de programación en C. [Pawelczak and Baumann 2014]

Virtual-C IDE está diseñado para apoyar la mayoría de tareas del proceso de enseñanza-aprendizaje. Por ejemplo, utiliza fuentes grandes por defecto para realizar demostraciones durante las lecciones. Los autores han reportado también su uso en el auto-aprendizaje, y como acompañamiento de tareas o evaluaciones.

Virtual-C IDE usa una metáfora multi-modal abstracta, mayoritariamente textual como se aprecia en la Figura 3.4 para un programa en C que calcula la mediana de un conjunto de valores. Virtual-C IDE no reporta algún tipo de ludificación.

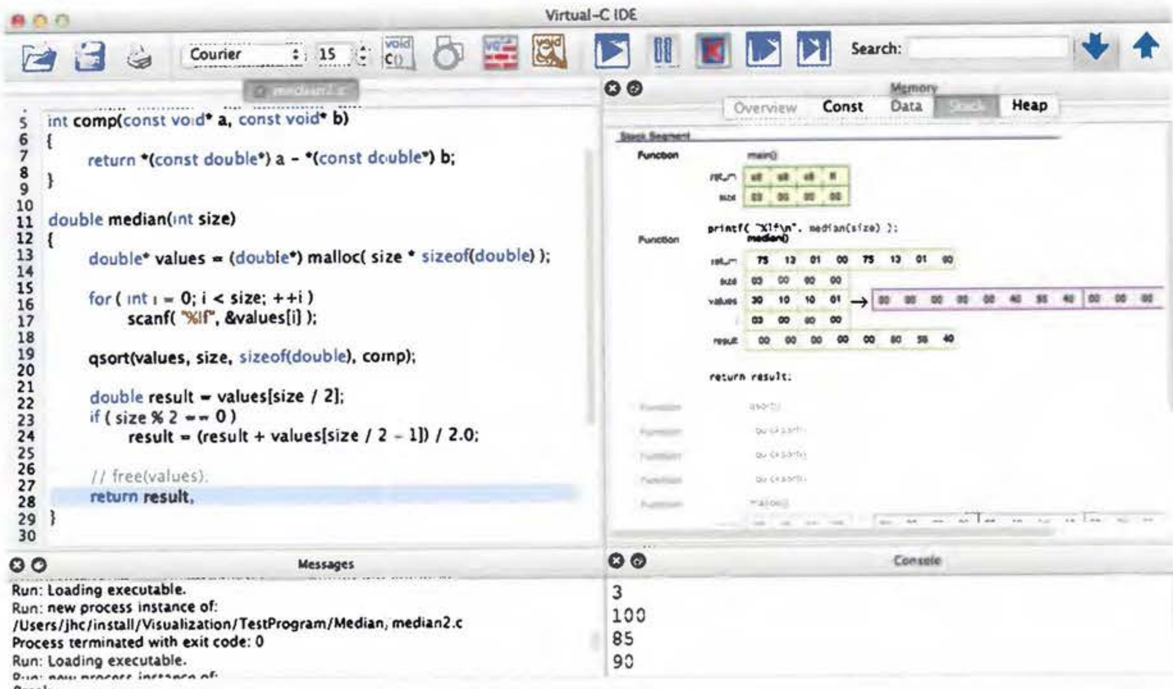



Figura 3.4. Captura de pantalla de Virtual-C IDE visualizando el programa de la mediana

En las visualizaciones de programa disponibles se validará el nivel de satisfacción de los requerimientos de software. Estos requerimientos se extrajeron de una teoría de aprendizaje en la sección 2.3 del marco teórico, y se recapitulan a continuación.

## 3.2 Requerimientos de software

El constructivismo sociocultural es una teoría de desarrollo, la cual incluye el tema de aprendizaje. De esta teoría se derivan recomendaciones didácticas que los mediadores pueden aplicar para ayudar en el proceso de aprendizaje. La mediación no está limitada exclusivamente a otras personas. Herramientas socioculturales, tales como computadoras y software, pueden también mediar en el proceso de aprendizaje [Rodríguez-Arocho 2002].

Dado que las herramientas de aprendizaje son mediadores en el proceso de aprendizaje, se adaptaron a través un razonamiento deductivo las recomendaciones para el mediador en requerimientos de software de alto nivel. Este proceso deductivo se realizó por facilidad de presentación en la sección 2.3 (Constructivismo sociocultural) del marco teórico. Se

obtuvieron 16 requerimientos de software de alto nivel. En esta sección simplemente se recapitulan los requerimientos en el Cuadro 3.5.

Cuadro 3.5. Requerimientos de software de alto nivel

| # | Nombre | Requerimiento de alto nivel |
|---|---|---|
| 1 | Motivación | Indagar qué motiva al estudiante a involucrarse en la tarea educativa, con el fin de personalizar la interacción con el usuario. |
| 2 | Estado activo | Propiciar una interacción en la que el usuario mantenga su mente en estado activo. |
| 3 | Interés | Estimular el interés del usuario por el concepto a aprender. |
| 4 | Realimentación | Realimentar al usuario sobre sus éxitos y ofrecer explicaciones en caso de error. |
| 5 | Nociones previas | Evaluar las nociones previas del estudiante y las asociaciones emotivas que de ellas se tengan, para adaptar la forma de introducir un concepto. |
| 6 | Contraposición conceptual | Contraponer conceptualmente los nuevos conceptos a los ya existentes en la mente el estudiante cuando las viejas nociones no permitan la construcción de las nuevas nociones. |
| 7 | Asimilación | Visualizar el nuevo concepto asociándolo con nociones que el estudiante ya tiene en su mente, traídas de su experiencia de vida. |
| 8 | Comparación de conceptos | Permitir al usuario realizar una cuidadosa comparación del concepto nuevo con los viejos, para identificar sus propiedades elementales comunes y distinguir sus diferencias fundamentales. |
| 9 | Ejercicios conceptuales | Fortalecer la asociación del nuevo concepto con cierto número de ejercicios, que varían entre ellos en aspectos no esenciales, mientras conservan el aspecto esencial del concepto. |
| 10 | Aplicación del concepto | Permitir al usuario aplicar el nuevo concepto a situaciones cotidianas, reales, o profesionales, útiles en su experiencia de vida. Cada situación plantea cambios en el contexto de aplicación del concepto, pero mantiene el principio fundamental del concepto. |
| 11 | Enfoque en el proceso | Enseñar y evaluar un proceso más que un producto final en la etapa consciente antes de que las acciones se conviertan en hábitos. |
| 12 | Formación de hábitos | Incentivar al usuario a aplicar el nuevo concepto hasta que desarrolle un hábito. |
| 13 | Sistemas de conceptos | Ayudar al estudiante asociar el nuevo concepto con otros ya presentados para formar sistemas de conceptos. |
| 14 | Organización de conceptos | Reflejar una organización natural de los sistemas de conceptos. |
| 15 | Evaluación | Evaluar que los sistemas de conceptos asimilados por el estudiante sean estables en el tiempo. |
| 16 | Transferencia | Ayudar al estudiante a aplicar los sistemas de conceptos a situaciones nuevas y complejas. |

Los requerimientos del Cuadro 3.5 podrían ya estar satisfechos por las visualizaciones de programa existentes. En la próxima sección se determinará en qué medida estas herramientas los satisfacen a través de una verificación de requerimientos.

## 3.3 Verificación de requerimientos

Con el fin de tener una aproximación de la medida en que los requerimientos de software son apoyados por las visualizaciones de programa existentes, se realizó una verificación de software cuyo resultado se encuentra en el Cuadro 3.6. Dado que se requiere una copia del software para realizar la verificación, sólo las visualizaciones de programa disponibles fueron incluidas en el Cuadro 3.6.

Para cada visualización de programa disponible, los 16 requerimientos de software de alto nivel (Cuadro 3.5, página 71) fueron subjetivamente evaluados con un porcentaje, donde 0% significa ningún soporte, y 100% que el requerimiento es soportado por completo. Los colores de fondo de las celdas del Cuadro 3.6 reflejan el valor del porcentaje, donde 0% es blanco y 100% es gris oscuro.

Cuadro 3.6. Verificación subjetiva de requerimientos en visualizaciones disponibles de programa

| | | Requerimientos de software | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| # | Visualización | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | Promedio |
| 1 | GRASP / jGRASP | 0 | 50 | 0 | 30 | 0 | 0 | 10 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 5.63 |
| 2 | The Teaching Machine | 0 | 40 | 0 | 40 | 0 | 0 | 10 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 5.63 |
| 3 | PlanAni | 0 | 60 | 0 | 70 | 0 | 0 | 60 | 0 | 0 | 0 | 0 | 0 | 0 | 10 | 0 | 0 | 12.50 |
| 4 | JIVE | 0 | 80 | 0 | 60 | 0 | 0 | 10 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 9.38 |
| 5 | Jeliot 2000 / Jeliot 3 | 0 | 70 | 0 | 50 | 0 | 0 | 20 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 8.75 |
| 6 | VIP | 0 | 40 | 0 | 60 | 0 | 0 | 10 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 6.88 |
| 7 | ViLLE | 0 | 100 | 30 | 70 | 0 | 25 | 20 | 20 | 0 | 0 | 0 | 0 | 75 | 50 | 10 | 0 | 25.00 |
| 8 | Metaphor-based OO visualizer | 0 | 20 | 10 | 20 | 0 | 0 | 60 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 6.88 |
| 9 | UUhistle | 0 | 100 | 30 | 80 | 0 | 60 | 30 | 20 | 0 | 0 | 80 | 20 | 30 | 50 | 0 | 0 | 31.25 |
| 10 | Online Python Tutor | 0 | 80 | 0 | 40 | 0 | 0 | 10 | 0 | 0 | 0 | 0 | 0 | 0 | 10 | 0 | 0 | 8.75 |
| 11 | Jeliot ConAn | 0 | 100 | 75 | 50 | 0 | 50 | 20 | 10 | 10 | 10 | 10 | 10 | 0 | 10 | 0 | 0 | 22.19 |
| 12 | BlueJ Novis | 0 | 60 | 0 | 50 | 0 | 0 | 20 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 8.13 |
| 13 | SeeC | 0 | 25 | 0 | 70 | 0 | 0 | 10 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 6.56 |
| 14 | Virtual-C IDE | 0 | 50 | 0 | 50 | 0 | 0 | 10 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 6.88 |
| | Promedio | 0.0 | 61.0 | 9.7 | 52.0 | 0.0 | 9.0 | 20.7 | 3.3 | 0.7 | 0.7 | 6.0 | 2.0 | 7.0 | 8.7 | 0.7 | 0.0 | 11.33 |

La fila "Promedio" informa el nivel promedio de satisfacción de cada requerimiento por las visualizaciones de programa disponibles, y constituye el principal producto del objetivo 1 de esta tesis. En general, el soporte de todos los requerimientos es muy bajo, excepto para los requerimientos 2 (estado activo) y 4 (realimentación).

El requerimiento de software 2 (estado activo) es satisfecho porque la mayoría de las visualizaciones de programa generales son interactivas. En ellas el usuario escribe código, provee parámetros de ejecución, o controla la ejecución de la visualización. *UUHISTLE*, *VILLE*, y *JELIOT CONAN* tienen los mayores niveles de soporte de este requerimiento. *UUHISTLE* fue diseñado con el principio constructivista de estado activo en mente: el usuario es invitado a realizar las animaciones, en lugar de hacerlas la computadora [Sorva and Sirkiä 2010]. El

estado activo es implementado en *VILLE* a través de su sistema de preguntas, el cual alienta a los estudiantes a estudiar activamente el código y su visualización con el fin de responderlas correctamente.

El requerimiento de software 4 (realimentación) sugiere proveer realimentación inmediata durante el proceso de aprendizaje. Todas las visualizaciones de programa proveen información visual inmediata sobre el estado del programa mientras está corriendo en la máquina nocional. Sin embargo, unas pocas herramientas proveen explicaciones verbales o de alto nivel cuando un programa falla o realiza una operación compleja, lo cual es una característica deseable.

Hay seis requerimientos que son ligeramente satisfechos por unas pocas visualizaciones, con promedios entre 6% y 20.7%. Se describen a continuación.

El requerimiento 3, sobre estimular el interés del estudiante en la tarea educativa, es principalmente satisfecho por *JELIOT CONAN*, ya que fue diseñado para actividades lúdicas. *VILLE* y *UUHISTLE* permiten al instructor escribir un texto introductorio en cada ejercicio de programación. Esos textos pueden usarse para proveer a los estudiantes contexto sobre la importancia de los conceptos de programación involucrados en el ejercicio.

El requerimiento 6, sobre intrigar a los estudiantes usando la técnica de la contraposición conceptual, es parcialmente apoyado por tres visualizaciones de programa. Los estudiantes podrían experimentar intriga cuando observan una animación anómala en *JELIOT CONAN*, cuando no saben cómo animar un paso en *UUHISTLE*, o cuando no pueden predecir el resultado de una pregunta en *VILLE*.

El requerimiento 7, sobre asociar nuevos conceptos con nociones previas en la mente del estudiante, es satisfecho mayoritariamente por las dos visualizaciones que usan metáforas visuales concretas: *PLANANI* y *METAPHOR-BASED OO VISUALIZER*. Sin embargo, esas metáforas son inconexas y explicaciones verbales son requeridas como complemento.

El requerimiento 11, sobre enfocar más el proceso que el producto final, es apoyado principalmente por *UUHISTLE*. Mientras los estudiantes producen la animación, *UUHISTLE* provee realimentación inmediata sobre acciones esperadas o equivocadas. Sin embargo, la herramienta no guarda el rastro o informa del progreso general del estudiante. Este progreso podría aprovecharse, por ejemplo, para evitar que los estudiantes sobre-animen conceptos que ya comprenden correctamente.

El requerimiento 13, sobre construir sistemas de conceptos, es apoyado especialmente por algunas preguntas en *VILLE* que requieren combinar varios conceptos de programación. Por ejemplo, los estudiantes deben aplicar reglas de precedencia de operadores con el fin de evaluar correctamente expresiones aritméticas que combinan: sumas de números, concatenaciones de cadenas de caracteres, e invocaciones al método `String.length()` en *JAVA*.

El requerimiento 14, sobre organizar conceptos para que reflejen relaciones naturales, es principalmente satisfecho por las visualizaciones que permiten a los instructores organizar ejercicios jerárquicamente: *VILLE* y *UUHISTLE*. Idealmente, la organización de conceptos debería reflejar una estructura de red.

Los restantes ocho requerimientos de software no son satisfechos por las visualizaciones existentes de programa. En otras palabras, ninguna visualización investiga qué motiva a los estudiantes (requerimiento 1), evalúa nociones o emociones previas (requerimiento 5), compara conceptos nuevos contra conocidos (requerimiento 8), ejercita un nuevo concepto aplicándolo a varios contextos (requerimiento 9), resuelve problemas reales que requieren la aplicación del nuevo concepto (requerimiento 10) hasta que los estudiantes desarrollen un hábito (requerimiento 12), evalúa que las nociones y habilidades de los estudiantes sean estables en el tiempo (requerimiento 15), y fomenta a que los estudiantes transfieran este conocimiento a nuevas situaciones (requerimiento 16).

En general, los requerimientos de software inferidos de una teoría de aprendizaje poco son satisfechos por las visualizaciones de programa disponibles. Esta tesis plantea que dichos requerimientos pueden satisfacerse con alegorías y ludificación. Ninguna visualización disponible usa alegorías y sólo una se diseñó para usarse en contextos lúdicos. Sin embargo, se encontraron dos trabajos que advierten de efectos negativos del uso de alegorías visuales en interfaces de usuario [Hsu 2006; Blackwell 2006]. Por este motivo, antes de aplicar alegorías a las visualizaciones de programa se realizó una revisión de literatura sobre alegorías en computación y un experimento que las compara contra metáforas tradicionales.

## 3.4 Alegorías en computación

En esta sección se hace una revisión cronológica de los trabajos relacionados con alegorías en los campos de educación de la computación (*CSE*, del inglés *COMPUTER SCIENCE EDUCATION*) e interacción humano-computador (*HCI*, *HUMAN-COMPUTER INTERACTION*). En *HCI* existe un interés especial en *metáforas visuales* para diseñar interfaces gráficas de usuario (*GUI*, del inglés *GRAPHICAL USER INTERFACE*). Sin embargo, se incluyen en esta revisión sólo aquellas metáforas que son *alegorías* visuales, y por tanto sus imágenes son coherentes en el dominio origen.

El método para la siguiente revisión no exhaustiva de literatura, fue un proceso de bola de nieve (en inglés, *SNOWBALL PROCESS*), el cual partió de los reportes experimentales hechos por Blackwell y Hsu [Blackwell 2001; Hsu 2006]. Este método se prefirió sobre una búsqueda sistemática debido a la incongruente diversidad de nomenclaturas para referirse a las metáforas y alegorías, como se estudió en la sección 2.4.2 (Clasificación de metáforas por ámbito) del marco teórico. No se encontró ninguna revisión de literatura sobre alegorías en computación, por tanto, la siguiente subsección provee además un aporte al conocimiento.

### 3.4.1 Revisión de literatura

En 1981, Richard Mayer realizó una serie de experimentos, dos relacionados a alegorías. Mayer usó una alegoría de un archivero para representar la máquina nocional de un lenguaje de consultas basado en *SEQUEL* (Figura 3.5) [Mayer 1981]. La alegoría fue usada como parte de un *organizador avanzado*, el cual es una corta introducción no técnica que se hace antes de presentar un contenido técnico [Mayer 1981, p.124]. En el primer experimento, el grupo tratamiento fue expuesto a la alegoría en forma de un organizador avanzado, y el grupo control no. Sus resultados mostraron que ambos grupos tuvieron un buen rendimiento en los problemas simples, pero el grupo control sobresalió en problemas complejos que requerían integración creativa de los constructos del lenguaje de consulta evaluados. Como un segundo resultado, la alegoría pareció incrementar el rendimiento de los participantes menos expertos, sin embargo, los aprendices con más habilidad que experimentaron la alegoría, tuvieron un rendimiento inferior que sus pares del grupo control. Mayer conjeturó que los aprendices más hábiles podrían ya poseer sus propios "modelos" de la máquina nocional, por tanto, podrían influirse menos por la alegoría usada como organizador avanzado. Para el

segundo experimento los participantes en el grupo control respondieron, en sus propias palabras, preguntas usando el contexto de la alegoría (el archivador). Los resultados encontraron un incremento en el rendimiento del tratamiento en comparación con el primer experimento [Mayer 1981].

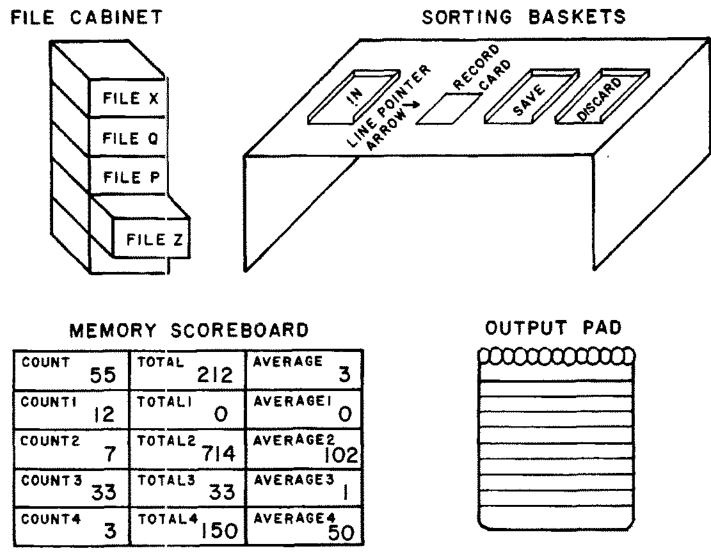



Figura 3.5. Alegoría de un archivador para una máquina *SEQUEL*. Fuente: [Mayer 1981, p.131]

En 1989, Waguespack propuso una metáfora compuesta multimodal para representar algunos conceptos del lenguaje de programación Pascal con una combinación de metáforas abstractas y concretas [Waguespack 1989]. Por ejemplo, los tipos de datos y registros se representaron con figuras geométricas, los punteros por carruajes de hilo, y una lista enlazada con una mezcla de ambos (Figura 3.6). Waguespack escogió rectángulos con las esquinas redondeadas para representar números enteros y rectángulos con las esquinas cuadradas para números reales, dado que si los reales se almacenan en enteros se pierden decimales, así como los rectángulos perderían las esquinas cuadradas. La metáfora de [Waguespack 1989] fue una propuesta, de la cual no se realizaron evaluaciones empíricas.

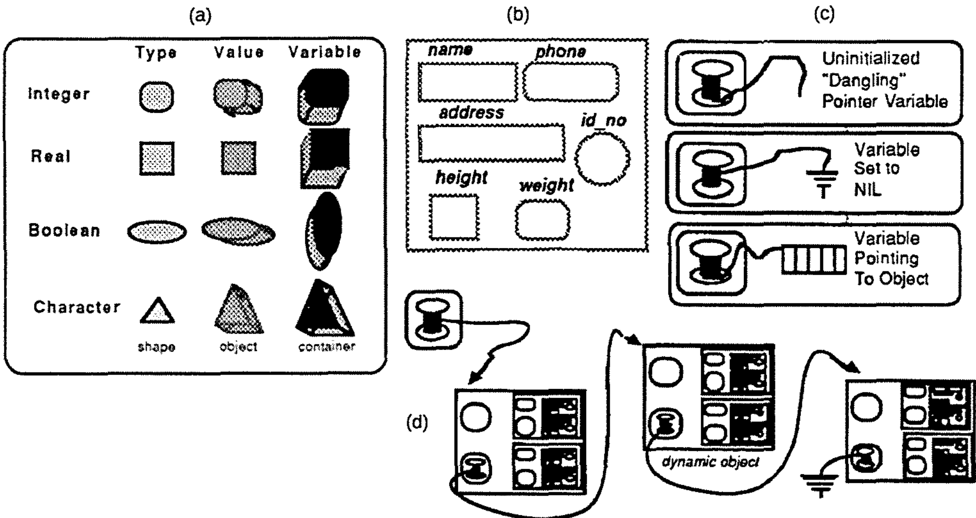



Figura 3.6. Elementos seleccionados de la metáfora multimodal compuesta de [Waguespack 1989]: (a) tipos de datos, (b) registro, (c) punteros, (d) lista enlazada

También en 1989, Lin realizó tres experimentos con el fin de comparar tres interfaces alegóricas para un sistema de información turística [Lin 1989]. El experimento 1 comparó tres alegorías: libro, ficha de notas, y mapa (Figura 3.7). El experimento 2 comparó las mismas tres interfaces alegóricas y una metáfora compuesta resultado de combinar las tres. El experimento 3 usó las mismas interfaces que el experimento 2, pero cambiando otros factores, como el tiempo de entrenamiento [Lin 1989]. A los participantes en los tres experimentos se les plantearon preguntas sobre información turística, y tres variables se midieron en sus respuestas: corrección, tiempo de respuesta y bloqueo mental. Los experimentos 2 y 3 son los relevantes a esta tesis, ya que las alegorías (el tratamiento) fueron comparadas contra metáforas inconexas (el control). El experimento 2 no encontró diferencias significativas en las tres variables. El experimento 3 encontró una diferencia significativa sólo en el tiempo de respuesta, que fue menor para las alegorías [Lin 1989].

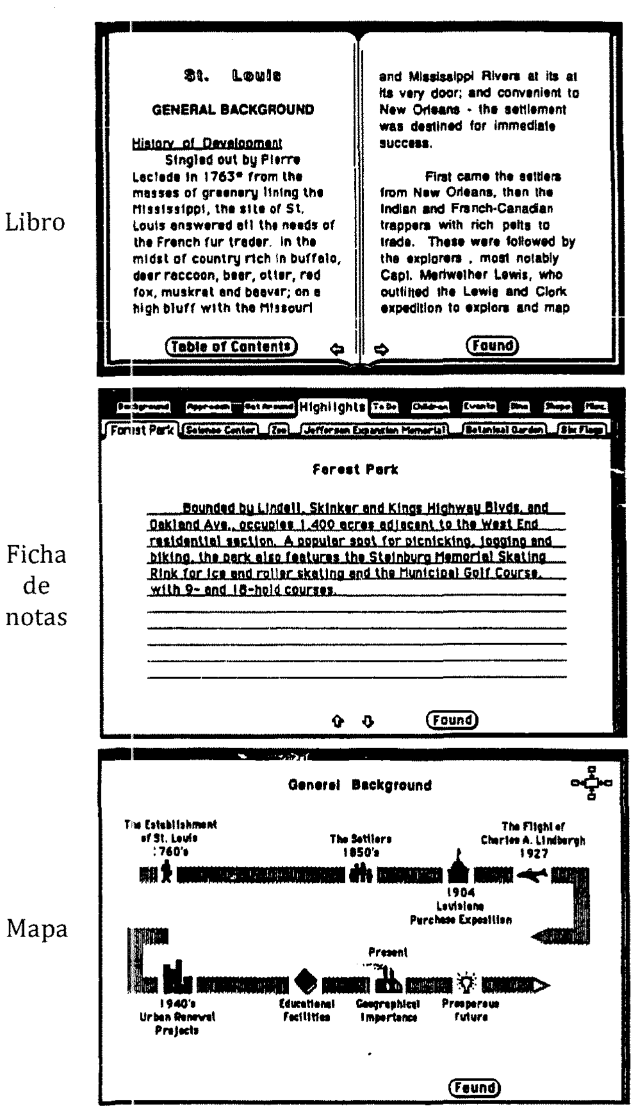
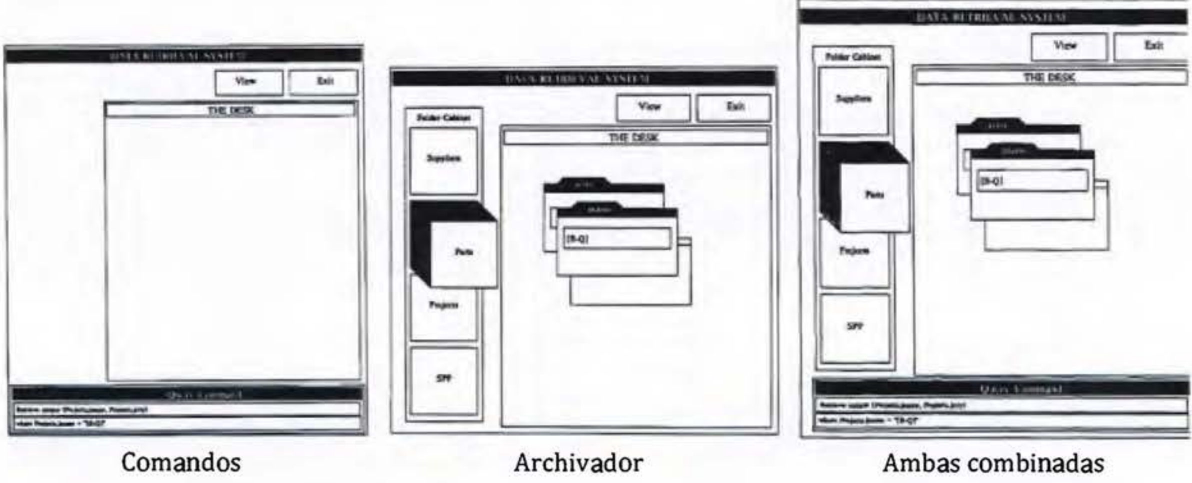



Figura 3.7. Las tres alegorías comparadas por [Lin 1989]

En 1993 Eberts y Bittianda compararon experimentalmente tres interfaces para hacer consultas a una base de datos: basada en comandos *SQL*, manipulación directa, y una combinación de ambas (Figura 3.8) [Eberts and Bittianda 1993]. Una *interfaz de manipulación directa* (en inglés, *DIRECT-MANIPULATION*) es una interfaz gráfica de usuario que permite a los usuarios controlar el objeto de estudio a través de un dispositivo de apuntar. Eberts y Bittianda usaron un archivador como alegoría visual para representar una base de datos, cuyas gavetas eran las tablas, las carpetas eran campos, y las hojas de papel eran los registros [Eberts and Bittianda 1993]. Veintiún estudiantes de pregrado en computación fueron asignados aleatoriamente a las tres interfaces. Los participantes entrenaron usando un folleto y un video. En el primer experimento, los participantes respondieron 25 preguntas que

requerían consultas a la base de datos. No se encontraron diferencias en el porcentaje de respuestas correctas, pero los participantes en el grupo con la interfaz de manipulación directa (alegoría) lo hizo en un tiempo significativamente menor que los otros grupos, principalmente en las tareas más complejas [Eberts and Bittianda 1993]. Un segundo experimento, en el que los mismos participantes hicieron tres tareas más complejas que las anteriores, encontró diferencias significativas únicamente en la tercera tarea, alterar la base de datos, de nuevo a favor de la alegoría [Eberts and Bittianda 1993].

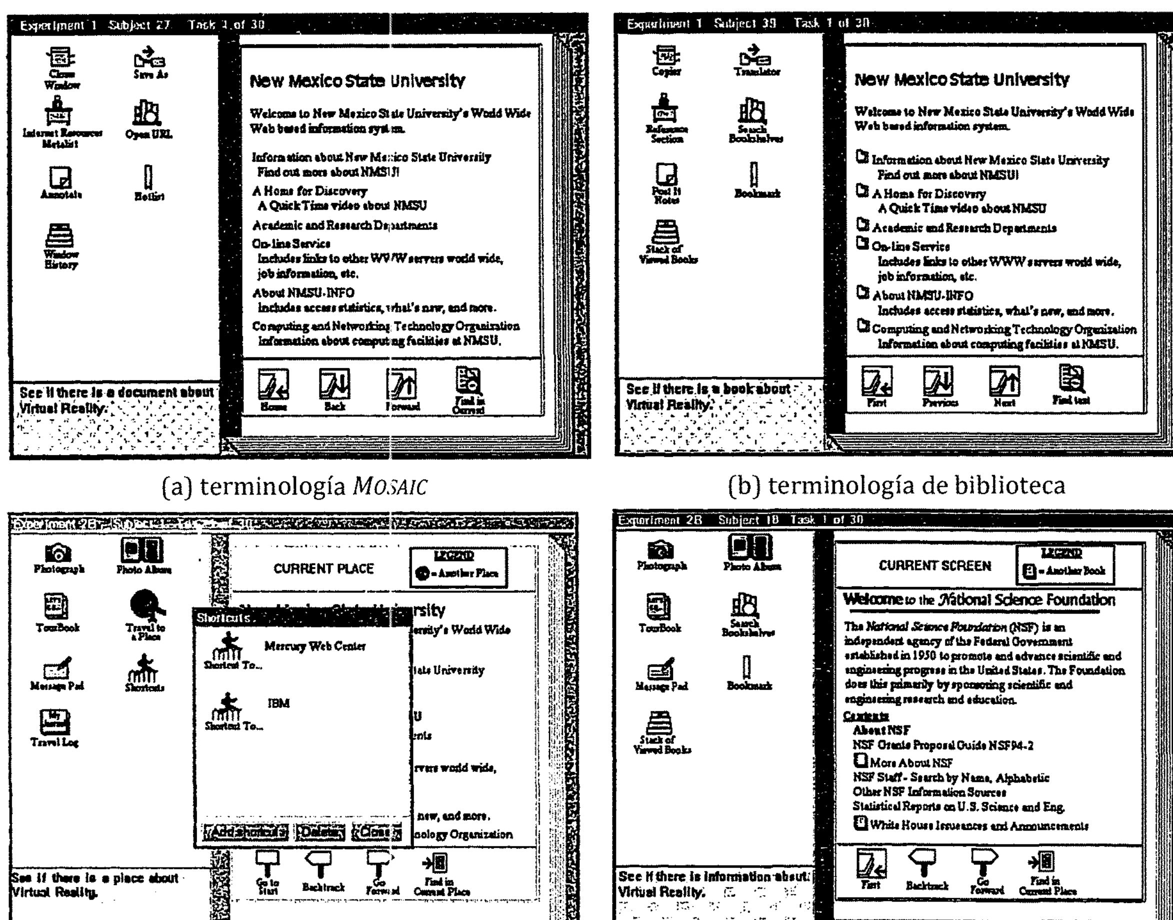



Comandos Archivador Ambas combinadas

Figura 3.8. Las tres interfaces comparadas por [Eberts and Bittianda 1993]

In 1995, Elissa Smilowitz condujo cinco experimentos para comparar alegorías con metáforas compuestas [Smilowitz 1995]. A los participantes, sin experiencia previa en navegación web, se les pidió completar algunas tareas típicas de navegación.

1. En el experimento 1, el grupo control usó un navegador especialmente diseñado con la misma terminología que el popular navegador *NCSA MOSAIC* (metáforas tradicionales) (Figura 3.9(a)). El grupo tratamiento usó el mismo navegador web adaptado con terminología alegórica a una biblioteca (Figura 3.9(b)). El grupo tratamiento (biblioteca) hizo significativamente menos errores, en menos tiempo, y completó más tareas que el grupo control (*MOSAIC*).
2. El experimento 2 fue idéntico al anterior, pero agregó un tratamiento: una alegoría con términos de viaje (Figura 3.9(c)). La alegoría de biblioteca superó de nuevo al control (*MOSAIC*), pero la alegoría de viaje no tuvo diferencia sobre el control.




(a) terminología *MOSAIC* (b) terminología de biblioteca

(c) terminología de viaje (d) metáfora compuesta de biblioteca y viaje

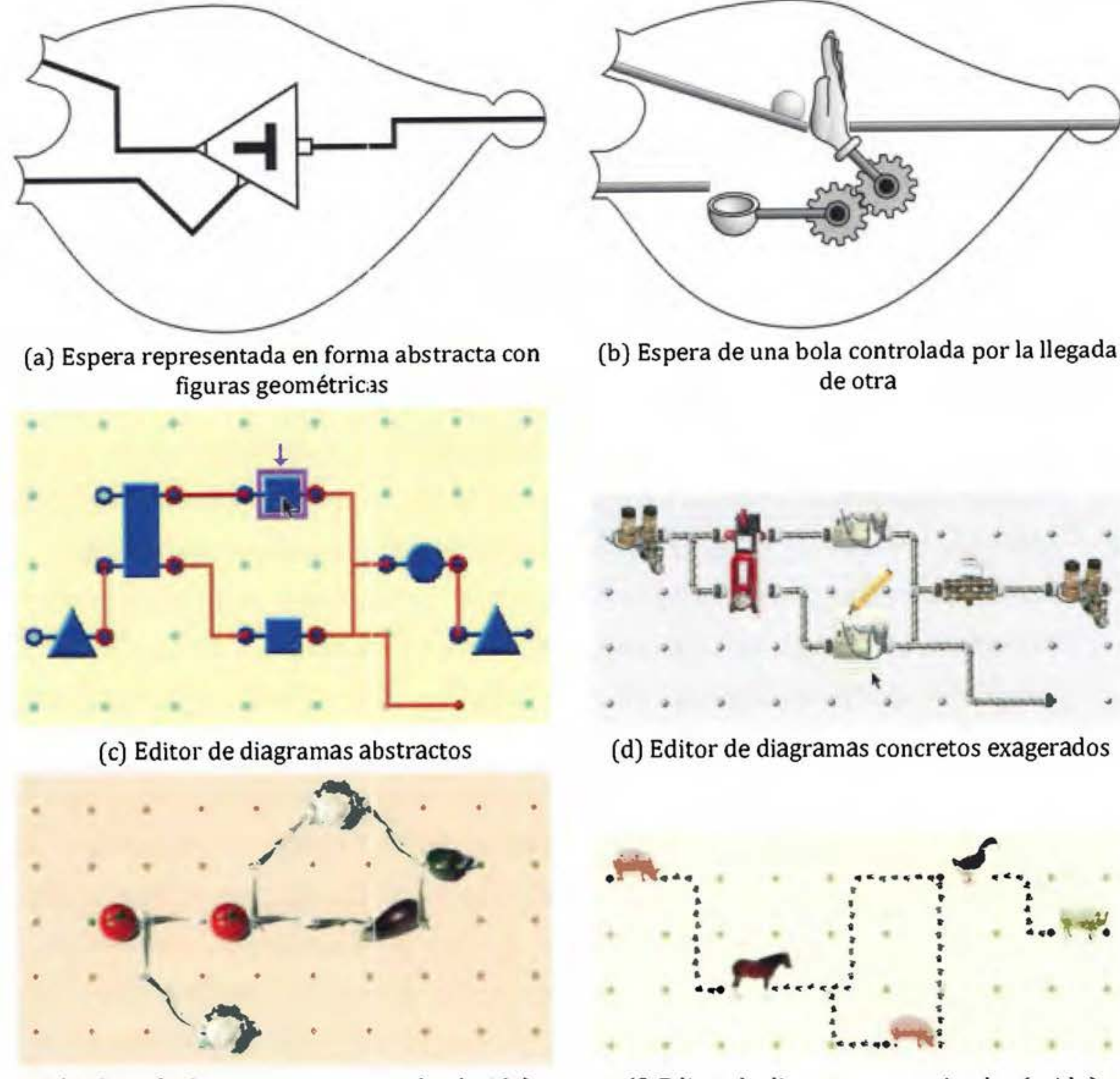
Figura 3.9. Interfaces comparadas por [Smilowitz 1995]

3. El experimento 3 comparó las dos alegorías (biblioteca y viaje) contra una metáfora compuesta resultado de una selección de metáforas de biblioteca y de viaje (control) (Figura 3.9(d)). De nuevo, la alegoría de biblioteca superó a la metáfora compuesta, pero la alegoría de viaje no tuvo diferencia con el control.
4. En el experimento 4, tanto la alegoría de biblioteca como la alegoría de viaje fueron comparadas en una tarea de búsqueda de información y en una tarea de localización de lugares. Como era de esperarse, la alegoría de viaje fue efectiva para la tarea de localización pero no para la búsqueda de información, mientras que la alegoría de biblioteca fue efectiva para ambas tareas.
5. El experimento 5 replicó el experimento 3, es decir, las dos alegorías (biblioteca y viaje) fueron comparadas contra una metáfora compuesta. Pero esta vez, la metáfora compuesta fue actualizada para combinar las características de ambas alegorías que fueron más

efectivas en el experimento 4. No se encontraron diferencias significativas en las tres variables dependientes [Smilowitz 1995].

Smilowitz concluyó que la efectividad de las alegorías y de las metáforas compuestas es dependiente de la relación entre los dominios origen y destino. Si las correspondencias entre esos dominios están bien diseñadas, las metáforas compuestas y las alegorías tendrán la misma efectividad [Smilowitz 1995].

Alan Blackwell reportó en 1999 y 2001 siete experimentos. Él comparó alegorías visuales (llamadas por él "representaciones pictóricas") contra metáforas visuales extendidas tradicionales (llamadas por él representaciones no-pictóricas, representaciones abstractas, o metáforas explícitas), ambas ideadas para procesos de aprendizaje [Blackwell and Green 1999; Blackwell 2001]. En los primeros seis experimentos se midieron dos variables dependientes: duración y rendimiento de novatos. El rendimiento se definió como la medida en la cual los novatos se aproximaron al rendimiento de los expertos. El experimento 1 comparó diagramas de flujo que usan bolas rodantes como tratamiento (Figura 3.10(b)), contra los mismos diagramas con figuras geométricas abstractas en el grupo control (Figura 3.10(a)). El experimento 2 comparó en el tratamiento, un editor de diagramas concretos exagerados con máquinas y tubos por conectores (Figura 3.10(d)), contra el mismo editor usando figuras geométricas y líneas como conectores en el grupo control (Figura 3.10(c)). El experimento 3 agregó al experimento 2, dos editores de ruido que usan vegetales (Figura 3.10(e)) y animales (Figura 3.10(f)). Únicamente el experimento 2 encontró diferencias significativas de rendimiento a favor del grupo control. Los restantes cuatro experimentos variaron el nivel de explicaciones dadas sobre las metáforas y alegorías a los participantes (explicaciones sistemáticas, explicaciones sin sentido, y sin explicaciones del todo). Los experimentos 4 a 6 no encontraron diferencias significativas, sin embargo resultaron estadísticamente no concluyentes debido a sus pequeñas poblaciones [Blackwell 2001].



(a) Espera representada en forma abstracta con figuras geométricas

(b) Espera de una bola controlada por la llegada de otra

(c) Editor de diagramas abstractos

(d) Editor de diagramas concretos exagerados

(e) Editor de diagramas con vegetales (ruido)

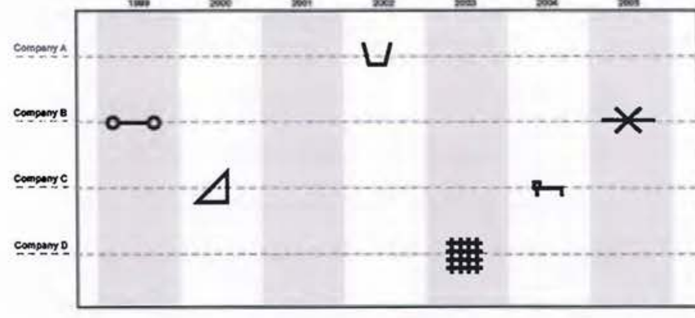
(f) Editor de diagrama con animales (ruido)

Figura 3.10. Algunas metáforas visuales usadas en los experimentos 1 a 6 por [Blackwell 1998]

De los experimentos de Blackwell, el séptimo es el más relevante para esta tesis. Los participantes fueron expuestos a cuatro ejemplos no relacionados con la programación: una mina de oro, la bolsa de valores (Figura 3.11), el diseño gráfico de un periódico, y rutas de vuelo [Blackwell 2001]. Cada participante experimentó dos ejemplos representados por alegorías visuales concretas (tratamientos) y dos ejemplos representados por metáforas visuales abstractas inconexas (controles). Tanto en los tratamientos como los controles, una metáfora en un ejemplo se explicó al participante y la otra no. Estas asignaciones aleatorias fueron distribuidas usando un diseño de cuadrado latino. Los participantes realizaron tareas

de comprensión de resolución de problemas mientras experimentaban cada uno de los cuatro ejemplos asignados al azar, pero no se encontraron diferencias significativas entre alegorías y metáforas inconexas. Después del cuarto ejemplo, los participantes definieron de memoria varios conceptos que habían sido presentados en los cuatro ejemplos. Los participantes tuvieron mejor precisión de recuerdo de los ejemplos que fueron presentados con alegorías no explicadas (un tratamiento), en comparación a las tres combinaciones restantes. Blackwell infirió que cuando no se proveen explicaciones verbales de las metáforas, los participantes crean sus propias explicaciones. Más aún, las alegorías visuales ayudaron en el proceso de creación de significado e influenciaron el rendimiento de los participantes al recordarlas. Para Blackwell, este resultado apoya la teoría de interacción de la metáfora de Max Black, ya que los aprendices construyeron significado de la interacción de dos dominios previamente desconectados. [Blackwell and Green 1999; Blackwell 2001]

(a) Cambios en la bolsa de valores de 4 compañías en siete años. Los cambios son representados usando figuras abstractas que se asemejan a las figuras concretas de la figura de abajo.




(b) Igual a la imagen anterior con metáforas concretas. Un subibaja indica que las acciones oscilan sin subir. Un tobogán es un descenso en la bolsa. Un columpio indica que las acciones oscilan pero incrementan su valor cada año. Una malla para escalar es un ascenso. Un caballo mecedor es un comportamiento inestable. Un carrusel es un ciclo en la bolsa de valores.

Figura 3.11. Uno de los cuatro ejemplos usados por [Blackwell 1998] en el experimento 7

En 1999, Yu-chen Hsu condujo un experimento en los Estados Unidos, aunque fue reportado hasta el 2003 [Hsu and Schwen 2003]. Hsu comparó tres interfaces para encontrar información en un sitio web: una estructura textual jerárquica, una alegoría de un libro, y una metáfora compuesta que mezcla la alegoría de libro con una metáfora secundaria de un sistema de carpetas (Figura 3.12) [Hsu and Schwen 2003]. A los 54 participantes no expertos

se les pidió que respondieran 15 preguntas cuyas respuestas debían encontrarse en el sitio web. Las variables dependientes fueron: corrección de las respuestas, duración en responder, el número de artículos leídos en el sitio web, y satisfacción de los participantes. La metáfora compuesta (libro con carpetas) sobresalió sobre la interfaz jerárquica en corrección, pero no se reportaron diferencias por la alegoría. Para las demás tres variables, no se encontraron diferencias significativas entre los tres tipos de interfaces [Hsu and Schwen 2003].

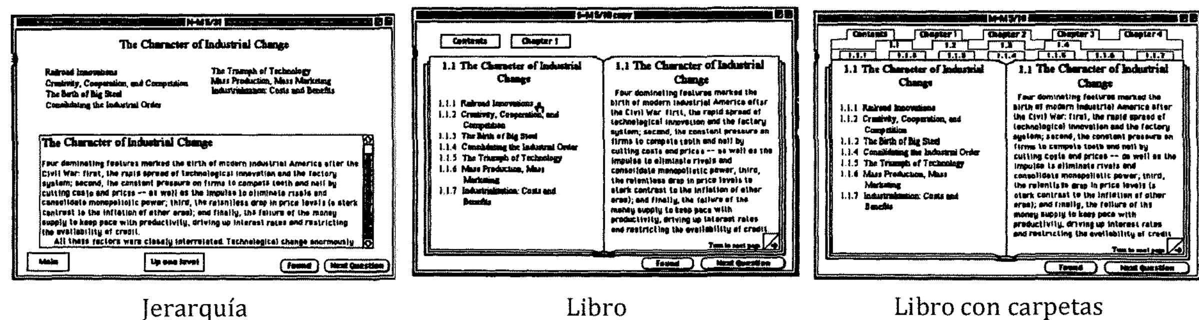



Figura 3.12. Las tres interfaces comparadas por [Hsu 2000]

En el 2003, Padovani y Lansdale condujeron un experimento para saber si algunas características de las alegorías visuales podrían tener efecto en ayudar a las personas a encontrar información que ellos habían visto antes en un sitio web (revisita) [Padovani and Lansdale 2003]. Se compararon dos alegorías; la primera fue concreta, intensamente familiar, y espacial; mientras la segunda carecía de esas características. La primera alegoría fue una casa virtual tridimensional, donde cada página web era un cuarto que contenía información, y los hiperenlaces eran puertas o pasillos que conectaban a los cuartos. Se le pidió a los participantes encontrar cinco elementos representados por flores (fase de búsqueda). Más tarde, se le pidió a los participantes retornar a regar esas cinco flores (fase de revisita). La segunda alegoría fue una red social donde una página web era un estudiante y un hiperenlace representaba que dos personas se conocían (amistad o vecindad). Los participantes en esta segunda alegoría eran detectives resolviendo un crimen donde las personas podrían conocer hechos importantes para resolver el caso. Se les pidió encontrar cinco testigos (fase de búsqueda), acordar citas, y hacer las interrogaciones luego (fase de revisita). La mitad de los participantes en cada grupo hizo su tarea bajo presión, la otra mitad no. La alegoría espacial-familiar (la casa) superó significativamente a la alegoría no espacial (la red social) en las variables medidas tanto en la fase de búsqueda como la fase de revisita. Es decir, los participantes con la alegoría de la casa fueron más rápidos en completar las tareas, visitaron

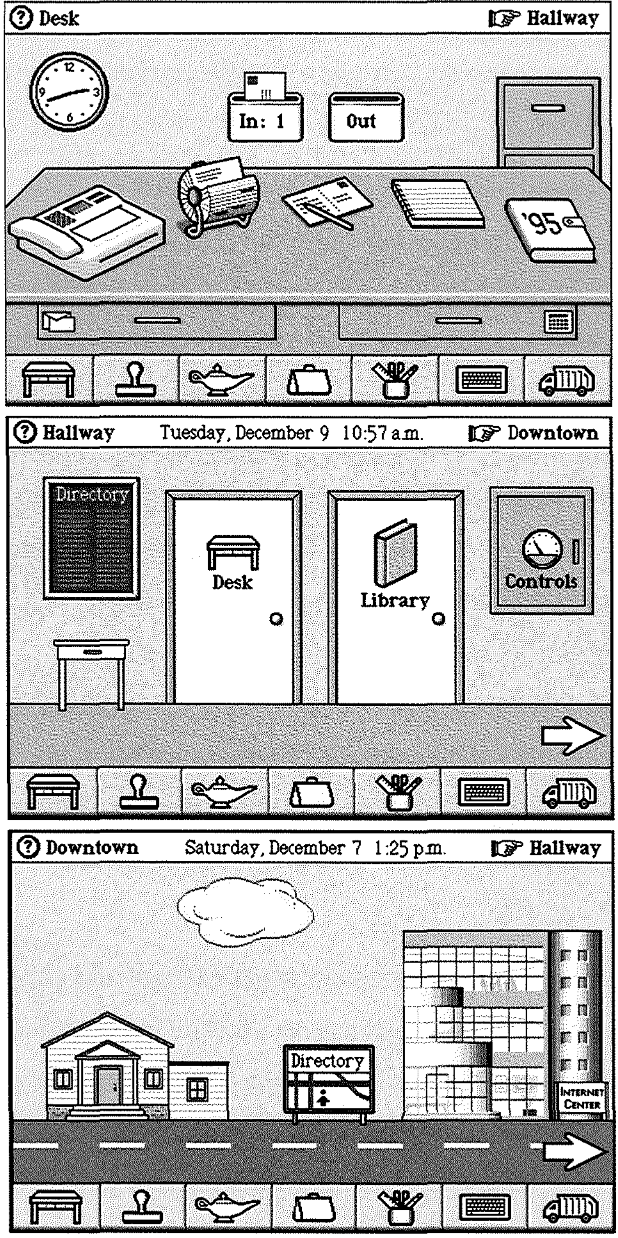
menos nodos para lograrlo, hicieron menos pasos redundantes, encontraron más objetivos, presionaron menos veces el botón para regresar, usaron estrategias de búsqueda selectiva en lugar de búsqueda exhaustiva, y recordaron con más precisión la estructura del sitio web [Padovani and Lansdale 2003].

En el 2005, Hsu replicó el experimento que hizo en Estados Unidos en 1999, pero ahora en Taiwán con 98 estudiantes [Hsu 2005]. Ella encontró que los participantes que usaron la jerarquía tradicional (grupo control) tuvo la mayor eficiencia. Esto es, los participantes leyeron menos artículos y respondieron las mismas preguntas que aquellos usando la alegoría (libro) y la metáfora compuesta (libro con carpetas) (Figura 3.12). No se encontraron diferencias significativas para las restantes tres variables. Sin embargo, algunas diferencias fueron detectadas en la interacción de los tres tipos de metáforas y la experiencia previa que los participantes tenían sobre búsqueda en la web. La alegoría y la metáfora compuesta fueron más útiles para los expertos que para los novatos. Hsu atribuyó la discrepancia entre estos resultados y los de su experimento previo al contexto, ya que los participantes estaban en diferentes países, la ejecución de los experimentos estaba separada por seis años, y el uso de la web se había extendido durante este lapso. Adicionalmente, Hsu dividió la tarea en tres sub-tareas de dificultad incremental, con el objetivo de medir efectos a largo plazo de las metáforas en la curva de aprendizaje. Encontró que los tres tipos de metáforas tuvieron el mismo efecto en las dos sub-tareas, pero el grupo control sobresalió en la tercera sub-tarea [Hsu 2005]. Sus resultados apoyan la afirmación de que las metáforas son apropiadas en las fases iniciales del aprendizaje, pero deben ser descartadas cuando los usuarios han desarrollado sistemas de conceptos estables (modelos mentales) sobre el software [Neale and Carroll 1997; Hsu 2005].

Cambiando de alegorías visuales, en el 2006, Hsu comparó experimentalmente *alegorías textuales* contra *metáforas textuales extendidas* tradicionales [Hsu 2006]. Los 77 participantes fueron asignados aleatoriamente a dos grupos. Los participantes en el grupo tratamiento leyeron un hipertexto sobre el protocolo de internet (*IP*, del inglés *INTERNET PROTOCOL*) que usó el sistema de correo tradicional como alegoría. Los participantes en el grupo control leyeron un hipertexto sobre el mismo protocolo usando el discurso tradicional compuesto de metáforas individuales inconexas. En ambos grupos, expertos y principiantes en computación participaron. Después de la fase de aprendizaje, los participantes respondieron tres pruebas. Hsu encontró que el grupo control (metáfora extendida tradicional) superó al grupo

tratamiento (alegoría) en la prueba de preguntas básicas, aunque no hubo diferencias en las otras dos pruebas. El principal resultado reportado por Hsu es que las alegorías deterioraron el rendimiento de los principiantes en responder preguntas básicas, porque ellos aún no tienen sistemas de conceptos bien organizados (modelos mentales, en la teoría Piagetiana) sobre el protocolo de internet para mapear la alegoría dada [Hsu 2006].

También en el 2006, Blackwell documentó la historia del uso de metáforas en el diseño de interfaces de usuario [Blackwell 2006]. Específicamente sobre *alegorías visuales*, indicó que éstas se usaron en productos comerciales como *MAGIC CAP* de *GENERAL MAGIC* (1994) (Figura 3.13) y *MICROSOFT BOB* (1995) [Blackwell 2006]. Esos productos se influenciaron por directrices corporativas que recomendaban diseñar interfaces de usuario basadas en metáforas familiares. Dichos productos recibieron costosos esfuerzos de desarrollo, pero resultaron en fracasos comerciales, y también fueron severamente criticados por la literatura científica. Blackwell manifiesta que la aplicación de metáforas en esos productos fue hecha sin un fundamento cuidadoso en las teorías de metáforas o evidencia científica [Blackwell 2006]. La influencia de metáforas visuales extra-realistas (alegorías visuales) en interfaces de usuario parece provenir de la aplicación de las teorías de desarrollo de Piaget y Ausubel al software educativo por parte de Seymour Papert y Alan Kay [Blackwell 2006]. Basado en un comentario de un taller de investigación, Blackwell metafóricamente criticó las alegorías visuales como una capa de azúcar que cubre el contenido real que se supone los estudiantes deben aprender, pero "los niños subvierten la intención del diseñador "jugando" con la interfaz de usuario como si fuera un videojuego, logrando el efecto de chupar la capa de azúcar y escupir el contenido de la píldora" (traducido de [Blackwell 2006, p.508]). Blackwell argumenta que la razón previa podría ser la causa que explique el fallo en el diseño de interfaces de productos como *MAGIC CAP* y *MICROSOFT BOB*. Otros productos de software también compartieron el mismo destino, tales como *MICROSOFT TASK GALLERY* (2000) [Robertson et al. 2000] y *SUN LOOKING GLASS* (2003).




(a) **Pantalla de inicio**. *Magic Cap* fue el sistema operativo creado por *General Magic* entre 1990 y 1994, para dispositivos móviles que dos décadas después llegarían a tomar el nombre de "celulares inteligentes" (*smartphones*). La interfaz de usuario de *Magic Cap* extiende la metáfora de escritorio, popularizada masivamente desde una década atrás, al pasar de una vista plana de un escritorio a una vista espacial de una oficina. Los objetos en la oficina permiten acceder a aplicaciones de información personal como agenda, teléfono, correo, notas, y contactos.

(b) **Pasillo**. La extensión espacial de la metáfora de escritorio se restringió a sí misma. Por ejemplo, el espacio en la oficina fue insuficiente y para mantener la coherencia con la alegoría, los autores crearon un pasillo el cual permite acceder a otros cuartos dentro del edificio, con más aplicaciones. Las flechas en el pasillo indican que aún hay más cuartos en el edificio, incluso se puede salir de él como se ve en la tercera imagen al centro de la ciudad. La interacción con *Magic Cap* semeja más a un juego de aventura que a un sistema operativo para usuarios de negocios.

(c) **Centro de la ciudad**. La ciudad permite al usuario comunicarse con el mundo externo, a través de aplicaciones que usan internet, como un navegador. La alegoría es criticada al romper la uniformidad, por ejemplo, los objetos de oficina que se mantienen en la parte inferior durante el recorrido por la ciudad. No se sabe si la sobre-interpretación literal de la metáfora de escritorio en *Magic Cap* podría ser la causa del fracaso comercial. Otras razones se han conjeturado como el alto costo y bajo rendimiento del hardware y de las comunicaciones a mediados de los noventa.

Figura 3.13. Alegoría visual de *Magic Cap*. Fuente: [Kuniavsky 2010, pp.40–42]

En el 2007, Hsu comparó experimentalmente el efecto de una alegoría multimodal contra una alegoría textual con 68 sujetos [Hsu 2007]. Los participantes en el grupo control leyeron un texto sobre el protocolo de internet (*IP*) usando el sistema de correo tradicional como metáfora. Los participantes en el grupo tratamiento leyeron un texto con imágenes sobre el protocolo de internet. Las imágenes eran metáforas visuales del sistema de correo tradicional. Sin embargo, no es claro si el texto para el grupo tratamiento usó el sistema de correo tradicional o fue un texto tradicional. No se encontraron diferencias significativas entre la

alegoría multimodal y la alegoría textual [Hsu 2007]. Hsu detectó una preferencia notoria de los participantes por las alegorías visuales sobre las textuales, a partir de datos cualitativos recolectados mediante entrevistas [Hsu 2007].

También en el 2007, Jiunde Lee comparó el efecto en conocimiento estructural y desorientación de la navegación web con hiperenlaces y la navegación basada en una alegoría visual. Participaron 116 personas interesadas en aprender a construir un sitio web [Lee 2007]. Los participantes leyeron un sitio web que contiene información sobre cómo crear sitios web. El grupo control experimentó un sitio web tradicional donde las páginas estaban conectadas por hiperenlaces de texto (Figura 3.14(a)). El grupo tratamiento experimentó el mismo sitio web, pero reemplazando su navegación por una alegoría visual de "dormitorios universitarios" (Figura 3.14(b)). La alegoría fue propuesta previamente por estudiantes durante una sesión de lluvia de ideas. Finalmente, los participantes respondieron cuestionarios sobre interrelaciones de conceptos sobre la creación de sitios web (conocimiento estructural), y su percepción de sentirse perdidos en lugar de entender la información dada (desorientación). Lee encontró que la navegación alegórica tuvo un efecto significativamente superior en el conocimiento estructural de los estudiantes en comparación a la navegación con hiperenlaces. En la percepción de desorientación auto-reportada, no se encontraron diferencias significativas [Lee 2007].

En el 2011, [Dørum and Garland 2011] replicaron el experimento de [Padovani and Lansdale 2003]. Compararon tres alegorías para navegación web variando el nivel de familiaridad y espacialidad: una casa, un pueblo, y una red social [Dørum and Garland 2011]. Encontraron que la alegoría de la casa sobresalió en tres variables dependientes: tiempo de la tarea, número correcto de nodos colocados, y número correcto de conexiones hechas. Para la cuarta variable, desorientación, no se detectaron diferencias significativas.

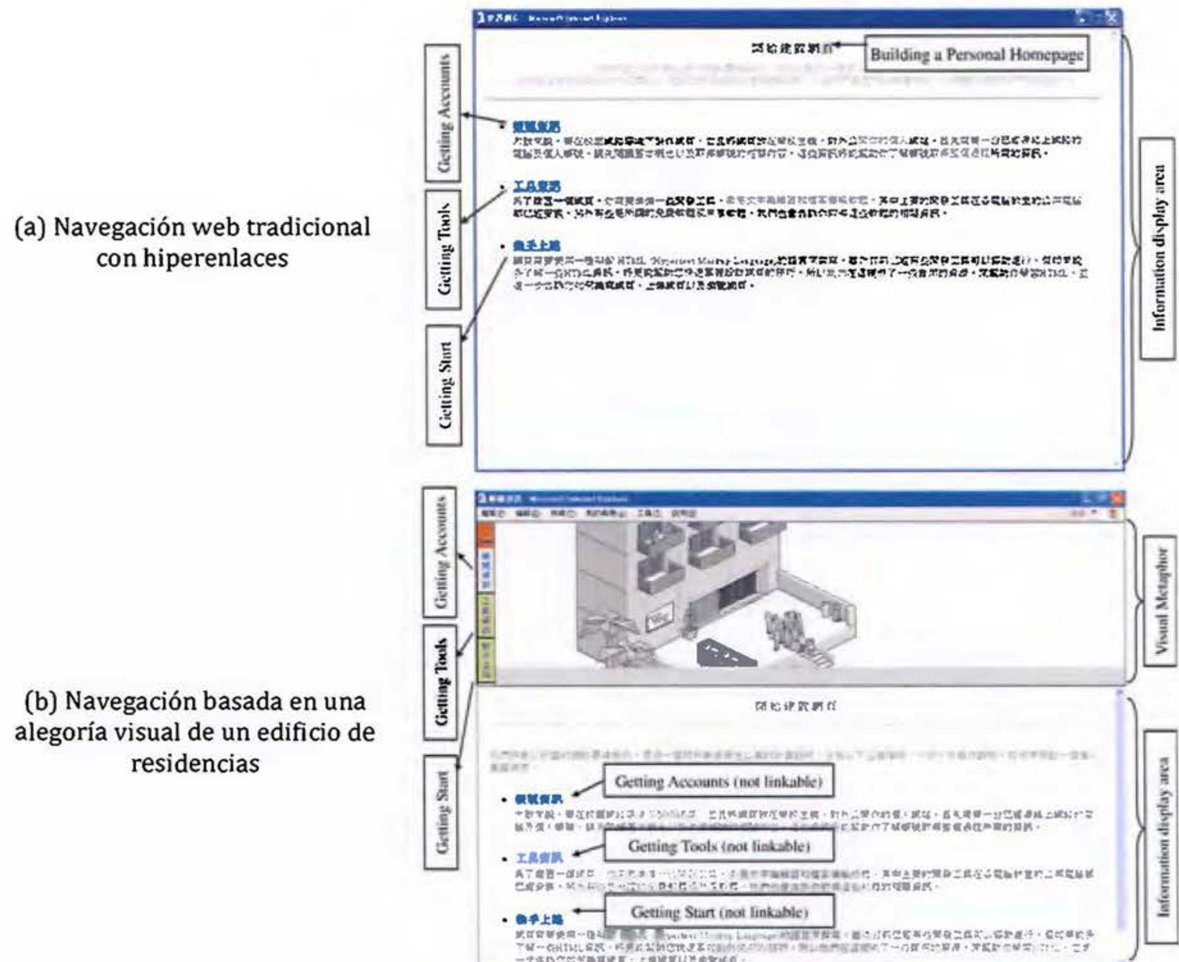



Figura 3.14. Navegación textual y con alegoría visual usada por [Lee 2007, p.621]

## 3.4.2 Discusión de la revisión de literatura

Casi la totalidad de los estudios consultados han comparado alegorías visuales y textuales contra sus correspondientes metáforas tradicionales a través de uno o más experimentos. Estos trabajos se muestran en el eje-y de la Figura 3.15. En cada estudio se midieron una o más variables dependientes. En el eje-x se muestra la cantidad de variables respuesta que encontraron efectos a favor del tratamiento (alegorías), del control (metáforas tradicionales inconexas), o ausencia de diferencias significativas.

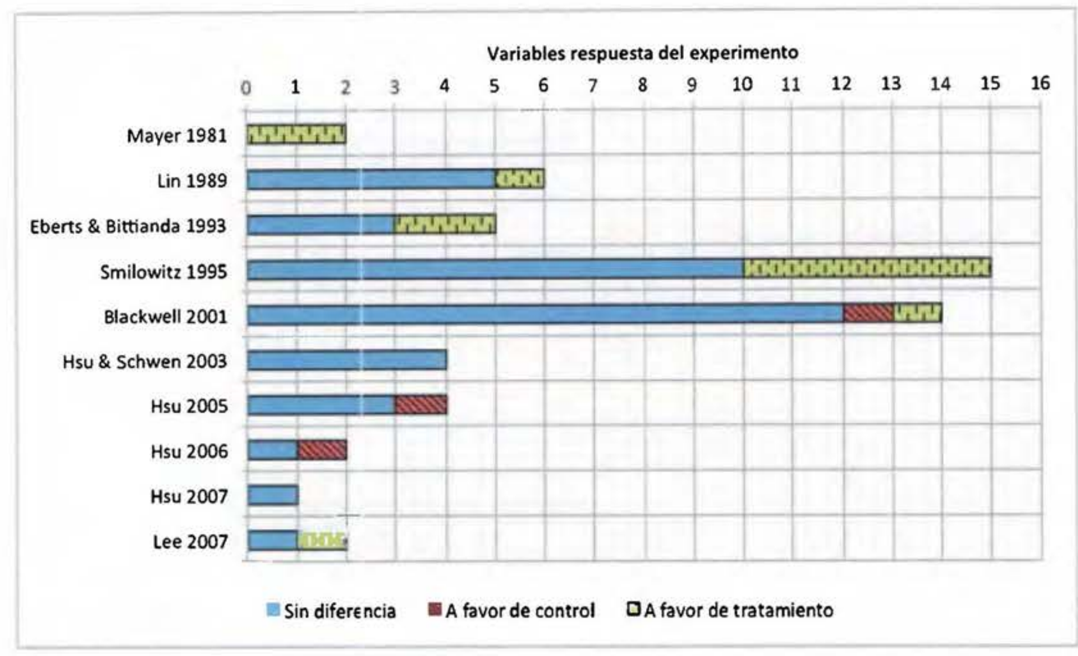



Figura 3.15. Resumen de comparaciones experimentales entre alegorías y metáforas

De las 55 variables medidas en los experimentos revisados, 40 de ellas (73%) no encontraron diferencias significativas entre alegorías y metáforas tradicionales; mientras que 12 variables (22%) encontraron que las alegorías son más efectivas que las metáforas tradicionales; y 3 variables (el restante 5%) detectaron que las metáforas tradicionales son más efectivas que las alegorías. Estos datos no confirman las advertencias de Blackwell y Hsu sobre efectos negativos de las alegorías visuales [Hsu 2006; Blackwell 2006] que inspiraron esta revisión. Por el contario, las alegorías y las metáforas tienen prácticamente el mismo efecto en la disciplina de la computación, con unas pocas excepciones donde las alegorías o las metáforas sobresalen. Esos resultados podrían explicarse por la influencia de nociones previas o el contexto, pero más investigación detallada es requerida para verificarlo.

La revisión de literatura realizada en esta sección descubrió que las alegorías visuales y textuales estaban dirigidas a facilitar procesos de aprendizaje de interfaces de usuario o de temas computacionales, y que la mayoría de las comparaciones experimentales encuentran que el efecto de las alegorías no difiere de las metáforas inconexas tradicionales. Sin embargo, ningún trabajo comparó metáforas orales contra alegorías orales, que son las más usadas para explicar máquinas nocionales durante las lecciones magistrales. La siguiente sección compara su efecto en una de las habilidades más importantes de la disciplina: la resolución de problemas.

## 3.5 Efecto de la enseñanza alegórica

El método más usado en la educación de la computación es la clase magistral [Naps et al. 2003; Isohanni and Järvinen 2014]. La enseñanza tradicional usa un discurso compuesto de metáforas orales inconexas, al que se le llama en este documento **enseñanza metafórica**. Ninguno de los estudios revisados en la sección anterior evaluó alegorías orales durante las lecciones magistrales, a lo que se llama en este documento **enseñanza alegórica**.

Como parte del curso de especialidad SP3024 "Psicología cognitiva", el autor de esta tesis junto con tres estudiantes de psicología y una de lingüística, diseñaron un experimento controlado para comparar el efecto de la enseñanza alegórica contra la enseñanza metafórica en una tarea de resolución de problemas de programación. Se realizó una primera ejecución del experimento al finalizar el primer semestre de 2014 con 8 participantes, y una segunda ejecución al finalizar el segundo semestre de 2014 con 7 participantes. Ambas ejecuciones sirvieron como pruebas pre-experimentales debido al escaso tamaño de muestra. Una tercera ejecución del experimento fue realizada por el autor de esta tesis al finalizar el primer semestre de 2017 con 28 participantes. Esta última iteración es la que se reporta en esta sección.

El objetivo del experimento fue responder dos preguntas de investigación:

1. ¿Afecta la enseñanza alegórica el rendimiento de resolución de problemas?
2. ¿Afecta la enseñanza alegórica aspectos motivacionales del proceso de aprendizaje?

### 3.5.1 Diseño experimental

Se siguió un enfoque constructivista para diseñar la tarea experimental. Con el fin de incrementar la *motivación* de los estudiantes (primer paso en la teoría de aprendizaje), se retó a los estudiantes a resolver un problema intrigante para desarrolladores de software, el cual no se cubre como un tema regular del curso. El tema seleccionado fue "¿por qué una interfaz gráfica de usuario (GUI) se congela cuando se inicia un proceso pesado?" Este es un problema interesante para estudiantes de computación porque es muy probable que enfrenten situaciones similares en su carrera profesional, y tendrán que entregar soluciones funcionales a sus clientes reales.

Durante el experimento, se presentó la *Calculadora de Goldbach* a los estudiantes (Figura 3.16). Se trata de una aplicación gráfica cuyo código en *JAVA* parece perfecto, pero se congela mientras está calculando. La conjetura de Goldbach establece que cada entero par mayor a 2 puede escribirse como la suma de dos primos, y cada entero impar mayor a 5 puede escribirse como la suma de tres primos. Por ejemplo:

```
 6 = 3 + 3              7 = 2 + 2 + 3
 8 = 3 + 5              9 = 2 + 2 + 5 = 3 + 3 + 3
10 = 3 + 7 = 5 + 5     11 = 2 + 2 + 7 = 3 + 3 + 5
```

La Calculadora de Goldbach (Figura 3.16) espera que el usuario ingrese un número positivo en el campo de texto rotulado con "Número" en la parte superior. Cuando se presiona el botón "Calcular", la aplicación calcula por fuerza bruta las sumas de dos o tres primos que equivalen al número ingresado de acuerdo a si es par o impar. Cada vez que encuentra una suma, se agrega al área de texto en la parte de abajo.

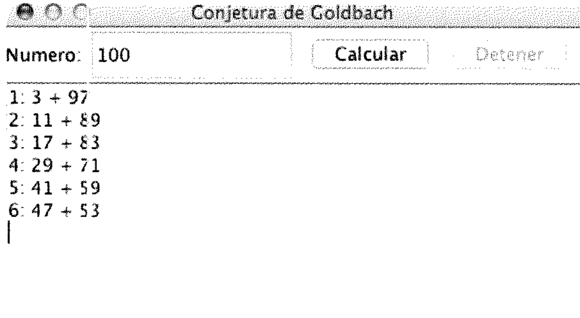



Figura 3.16. Calculadora de Goldbach mostrando las sumas de primos para el número 100

La aplicación incluye un botón "Detener", ya que los cálculos para números grandes pueden tomar tiempo considerable. Sin embargo, la Calculadora de Goldbach provista a los participantes no implementa concurrencia. Por tanto, para números grandes la aplicación no muestra las sumas parciales en el área de texto, la interfaz gráfica se congela, y el botón "Detener" se mantiene inactivo, hasta que el cálculo termine.

Se le solicitó a los participantes reparar el código suministrado de la Calculadora de Goldbach. Se escogió esta tarea compleja de resolución de problemas porque las alegorías se han considerado más útiles para estas situaciones [Hsu 2006]. De acuerdo a la teoría de aprendizaje, los estudiantes en este momento podrían experimentar contraposición

conceptual, ya que su conocimiento es insuficiente para resolver un problema que les interesa. Por tanto, ellos necesitan andamiaje para superar este estado de incertidumbre.

Se prepararon dos videos, uno metafórico y otro alegórico, que explican la teoría requerida para resolver el problema de la interfaz gráfica congelada. Ambos videos están estructurados en tres escenas: el problema, la teoría, y una solución. En la primera escena, el problema, se crea una aplicación minimalista llamada Tarea Pesada (Figura 3.17). Cuando se presiona el botón "Iniciar" de la Tarea Pesada, la aplicación ejecuta un ciclo *FOR* incrementando una variable hasta un entero muy grande. Aunque cada componente gráfico fue implementado en el código fuente durante la escena, la interfaz se congela de la misma forma que lo hace la Calculadora de Goldbach.

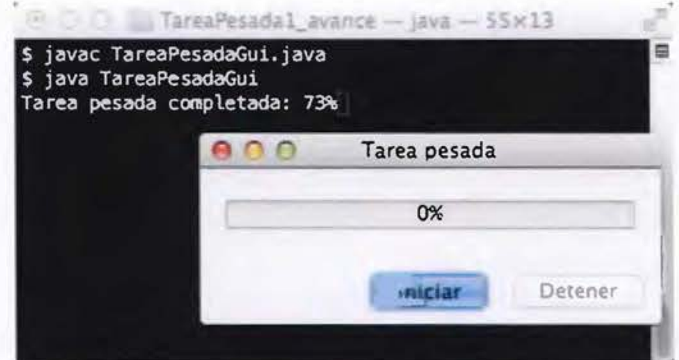



Figura 3.17. Una aplicación minimalista con interfaz gráfica congelada

La Tarea Pesada también imprime el porcentaje completado en el error estándar. Como se puede ver en la instantánea de la Figura 3.17, la aplicación reporta en la terminal que ha completado un 73% del ciclo *FOR*, pero la barra de progreso se mantiene en 0% y el botón "Detener" se mantiene deshabilitado. Durante la primera escena, se crea, se explica y se corre el código *JAVA* de la aplicación Tarea Pesada. La escena se creó mediante grabación de pantalla (en inglés, *SCREENCAST*) con una duración de 5 minutos y 33 segundos, y fue idéntica para el video del grupo control y del grupo tratamiento (Figura 3.18).

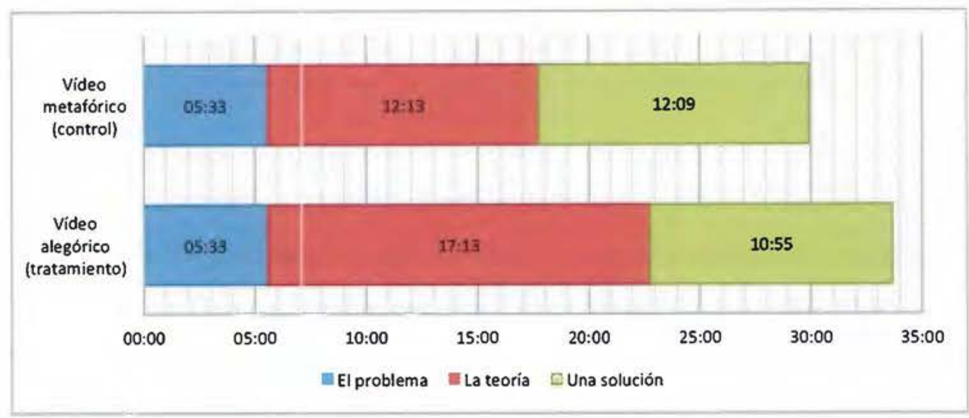



Figura 3.18. Duración en minutos de cada escena en los dos videos

En la segunda escena, el autor de esta tesis en rol de profesor fue filmado explicando la teoría requerida para comprender y resolver el problema. El enfoque fue distinto en cada video. En el primer video, llamado el *video metafórico*, el profesor explicó los conceptos abstractos sobre programación orientada a eventos mediante enseñanza metafórica. Es decir, el profesor usó las convencionales metáforas individuales inconexas, en la misma forma en que son empleadas en una clase magistral tradicional o por un libro sobre el tema. Un extracto resumido del guión es el siguiente:

> La interfaz gráfica no procesa los eventos del usuario de inmediato, sino que los agrega a una "cola de eventos" para procesarlos luego, cuando esté ociosa. Esta cola es requerida porque la interfaz sólo procesa un evento a la vez, y debe recordar los eventos pendientes para procesarlos luego. De esta forma, si la interfaz está realizando una tarea pesada, no responderá a nuevos eventos del usuario hasta que haya terminado esa tarea [...] Este problema se soluciona separando la interfaz gráfica de la tarea pesada en dos hilos de ejecución.

En el segundo video, llamado *video alegórico*, el profesor explicó la teoría requerida sobre programación orientada a eventos usando una alegoría. Un extracto del guión es el siguiente:

> La interfaz gráfica es como la ventanilla de una secretaría. Cuando un usuario realiza una solicitud (un evento), la recepcionista (un hilo de ejecución) no la atiende de inmediato, sino que la anota en una libreta de pendientes llamada "cola de eventos". Nuestra recepcionista es muy estructurada. Ella usa la libreta para evitar olvidar alguna tarea, y ella sólo procesa una tarea a la vez. Si ella está ocupada haciendo algún trámite (una tarea pesada) en el archivo de la secretaría, descuidará la ventanilla (la

> interfaz) y no responderá a nuevas solicitudes de los usuarios [...] Este problema se soluciona haciendo que la recepcionista se encargue únicamente de la ventanilla. Cada vez que un trámite es solicitado, ella delega la tarea a uno o más asistentes (hilos de ejecución).

La duración de la segunda escena fue de 12:13 minutos para el video metafórico, y de 17:13 minutos para el video alegórico (Figura 3.18). El video alegórico fue 5 minutos mayor debido a dos razones principales. Primero, la narrativa del guión alegórico es más extensa, como se puede percibir en los extractos anteriores. Un estudio previo reportó también este hecho [Hsu 2007]. Segundo, es más difícil presentar temas en una forma no familiar [Blackwell 2001]. Se conjeturó que esta diferencia podía afectar la percepción motivacional de los estudiantes, al considerar el video alegórico más lento o más aburrido que el tradicional. Por tanto, se aplicó un cuestionario al finalizar la sesión experimental con preguntas motivacionales que incluyeron variables sobre la velocidad y el interés percibido por los participantes sobre los videos.

La tercera escena sobre una solución, mostró cómo aplicar la teoría de la escena anterior para reparar el código de la Tarea Pesada. Se creó mediante grabación de la pantalla (*SCREENCAST*) de la misma forma que la primera escena. Se hicieron las mismas modificaciones al código en ambos videos para resolver el problema, pero las explicaciones orales fueron diferentes para cada video, usando metáforas inconexas o alegorías de acuerdo al tipo de video. Por ejemplo, cuando se movió el ciclo *FOR* de la interfaz gráfica a una clase nueva heredada de `javax.swing.SwingWorker`, el profesor dijo "ahora movemos la tarea pesada de la interfaz gráfica a un nuevo hilo de ejecución" en el video metafórico, y "ahora liberamos a nuestra recepcionista del papeleo y lo delegamos a nuestro nuevo asistente" en el video alegórico. Al final de ambos videos, la aplicación minimalista Tarea Pesada corrió como se esperaba: incrementa la barra de progreso, habilita el botón "Detener", y detiene el proceso pesado si este botón se presiona (Figura 3.19).

La duración de la tercera escena fue similar para ambos videos: 12:09 minutos para el video metafórico y 10:55 para el video alegórico, como se aprecia en la Figura 3.18. La duración total del video metafórico fue de 29 minutos y 55 segundos, mientras que el alegórico fue de 33 minutos y 41 segundos. Es decir, el video alegórico fue aproximadamente 11% más extenso que el metafórico.

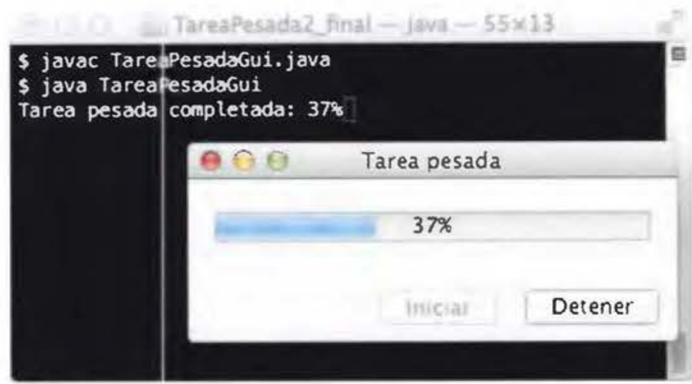



Figura 3.19. La aplicación minimalista Tarea Pesada reparada

## 3.5.2 Participantes del experimento

Los participantes fueron estudiantes del curso "Programación I". Esta población homogénea es una de las poblaciones interesantes identificadas por Blackwell, ya que son potenciales beneficiarios de las alegorías [Blackwell 2001]. Este curso se enseña con *JAVA*, y como tema opcional, algunos estudiantes pueden aprender a programar interfaces gráficas de usuario (*GUI*).

Se siguió un diseño entre-sujetos para investigar el efecto de las metáforas y alegorías orales. Se dividieron al azar los estudiantes en dos grupos: control (metáforas tradicionales) y tratamiento (alegoría). Cada estudiante recibió un folleto impreso con las instrucciones para la actividad de acuerdo al grupo asignado. Los estudiantes primero respondieron dos preguntas en el folleto sobre su estilo de aprendizaje, y su experiencia previa con la programación concurrente. Luego, leyeron en el folleto el problema de la Calculadora de Goldbach donde se les retó a resolverlo.

El folleto contenía un hiperenlace (*URL*, del inglés *UNIFORM RESOURCE LOCATOR*) hacia el video a ser usado como referencia para resolver el problema. El *URL* era distinto entre los dos folletos. De esta forma, los participantes en el grupo control vieron el video metafórico tradicional, y los sujetos en el grupo tratamiento vieron el video alegórico. Los estudiantes descargaron una copia completa del video a sus máquinas locales, y podían controlar a gusto su reproducción. Se les pidió usar el video como su única referencia para resolver el problema de la Calculadora de Goldbach.

De acuerdo a la teoría del constructivismo sociocultural, los estudiantes realizaron una asimilación de conceptos mientras veían el video asignado, y aplicación de conceptos al escribir código para la Calculadora de Goldbach, con el fin de resolver el problema en la sesión de dos horas. La corta duración de la sesión y la complejidad del tema, no permitió a los estudiantes construir hábitos o sistemas de conceptos. Por tanto, de acuerdo a la teoría, la estabilidad de las asociaciones sería débil, es decir, estas conexiones serían establecidas en la memoria de corto plazo.

Al terminar la tarea o al vencerse el tiempo de la sesión, los estudiantes llenaron un cuestionario corto al final del folleto. El cuestionario preguntaba a los estudiantes por sus percepciones motivacionales sobre el método de enseñanza usado en el video, su nivel percibido de aprendizaje, y sugerencias para mejorar el material de instrucción.

Se le solicitó a los estudiantes subir una copia de su solución de la Calculadora de Goldbach a una asignación publicada en la plataforma educativa oficial del curso (*LMS*, del inglés *LEARNING MANAGEMENT SYSTEM*). Los estudiantes recibieron crédito extra en el curso por su rendimiento. El experimentador, que no era profesor de "Programación I" en el semestre en que se ejecutó el experimento, evaluó las ocho tareas listadas en el Cuadro 3.7 a todas las soluciones subidas en la plataforma educativa. Estas tareas son requeridas para resolver el problema planteado.

Cuadro 3.7. Tareas evaluadas como indicador de rendimiento del participante

| # | Descripción corta |
|---|---|
| 1 | Crea una clase que hereda de `SwingWorker` |
| 2 | Mueve la lógica de calcular sumas de la interfaz a la clase nueva |
| 3 | Implementa el método `doInBackground()` de la clase nueva |
| 4 | Hace que la interfaz gráfica provea el número a calcular a la clase nueva |
| 5 | Actualiza la interfaz mientras las sumas se están calculando |
| 6 | Instancia la nueva clase cuando se presiona el botón Calcular y le invoca `execute()` |
| 7 | Cancela el cálculo de las sumas cuando se presiona el botón Cancelar |
| 8 | Actualiza la interfaz cuando el cálculo de las sumas ha terminado |

El experimentador calificó cada tarea a cada estudiante usando un porcentaje que indica el nivel de completitud de implementación de la tarea. El promedio de las notas de las tareas obtenidas por un estudiante fue considerado como su *rendimiento*, y es la variable de respuesta para la primera pregunta de investigación del experimento.

Dos profesores voluntariamente aceptaron correr el experimento en una de sus lecciones regulares del curso y ofrecer crédito extra a los estudiantes por su rendimiento. Dado que un profesor tuvo dos grupos del curso "Programación I", tres grupos en total participaron. Por consiguiente, el experimento se replicó tres veces para ajustarse a los horarios de los grupos, como se muestra en el Cuadro 3.8. Esos grupos se consideran *réplicas* del experimento en el resto de esta sección.

Cuadro 3.8. Réplicas del experimento que compara metáforas contra alegorías

| | **Réplica 1** | **Réplica 2** | **Réplica 3** | **Total** |
|---|---|---|---|---|
| Profesor: | A | A | B | |
| Fecha: | 13-jun-2017 | 13-jun-2017 | 15-jun-2017 | |
| Hora: | 11:00-13:00 | 15:00-17:00 | 9:00-11:00 | |
| Participantes: | 16 | 8 | 10 | 34 |
| Entregas válidas: | 11 | 8 | 9 | 28 |
| Entregas de control: | 5 | 4 | 4 | 13 |
| Entregas de tratamiento: | 6 | 4 | 5 | 15 |

### 3.5.3 Sesiones experimentales

El experimento se condujo cerca del final del ciclo lectivo, dado que el tema de programación de interfaces gráficas es uno de los últimos en cubrirse en el curso. Las tres réplicas se realizaron entre el 13 y 15 de junio de 2017 con 34 participantes (Cuadro 3.8). Los participantes fueron distribuidos aleatoriamente entre los grupos control y tratamiento con el siguiente algoritmo. Se intercalaron los folletos de control y tratamiento uno tras otro. Se colocó la pila de folletos, iniciando con uno de control en la parte superior, en la entrada del laboratorio de computadoras. Cada estudiante tomó el folleto en la parte superior de la pila cuando llegó al laboratorio. Seguidamente, el estudiante escogió una computadora y siguió las instrucciones en el folleto. La asignación de participantes a los grupos control y tratamiento es aleatoria dado que no se controla el orden en que los estudiantes llegan al laboratorio. Este simple algoritmo tiene la ventaja de que automáticamente balancea la cantidad de participantes en ambos grupos.

Durante la sesión experimental los profesores de los cursos no estuvieron presentes. El experimentador, quien grabó los videos, medió la sesión y luego calificó las soluciones entregadas. El experimentador no conocía a los estudiantes (ni el recíproco), y no proveyó

ayuda de programación durante el experimento, a excepción de clarificar dudas de compresión sobre la sesión.

Los estudiantes tuvieron un lapso de dos horas para reparar el problema en la Calculadora de Goldbach. En promedio tardaron 1 hora y 47 minutos. Cuando terminaron, los participantes subieron su código a la plataforma educativa del curso y respondieron el cuestionario al final del folleto. En la primera réplica, cuatro estudiantes no resolvieron el problema y rehusaron subir su código *JAVA* a la plataforma educativa. En la misma réplica, un estudiante subió archivos equivocados pero modificó los originales tras el experimento, por tanto, su solución fue excluida de los análisis. Un estudiante en la réplica 3 subió un archivo vacío, y también se descartó para los análisis estadísticos. A pesar de esto, los seis estudiantes respondieron el cuestionario motivacional al final del folleto. En síntesis, el número de soluciones válidas presentadas para análisis fue de 28 observaciones como se muestra en el Cuadro 3.8, con 13 observaciones en el grupo control y 15 en el grupo tratamiento. En la siguiente sub-sección se usan estas observaciones para responder las dos preguntas de investigación del experimento.

### 3.5.4 Resultados del experimento

Para saber si la instrucción alegórica afecta el rendimiento de resolución de problemas (primera pregunta de investigación del experimento), el experimentador calificó cada una de las ocho tareas del Cuadro 3.7 a cada una de las 28 soluciones presentadas. Cada tarea fue calificada usando una rúbrica compuesta de cinco categorías valoradas entre 0.0 y 1.0 en pasos de 0.25, donde 0.0 indica que la tarea no se realizó del todo, y 1.0 significa que la tarea fue completamente implementada. La variable dependiente fue el *rendimiento*, calculado como el promedio de las ocho tareas hechas por un participante. La relación entre el tipo de enseñanza (metafórica o alegórica) y el rendimiento produce tres hipótesis que podrían responder la pregunta:

H0: La enseñanza alegórica tiene *el mismo* rendimiento que la metafórica.

H1: La enseñanza alegórica tiene *mayor* rendimiento que la metafórica.

H2: La enseñanza alegórica tiene *menor* rendimiento que la metafórica.

Se realizó un análisis de varianza de un factor (*ANOVA*, del inglés *ANALYSIS OF VARIANCE*) para comparar el efecto del método de enseñanza en el rendimiento de los estudiantes en resolver el problema. Se usó R como herramienta estadística y el resultado del análisis de varianza se lista en el Cuadro 3.9. No hubo un efecto significativo del método de enseñanza en el rendimiento de resolución de problemas para rechazar la hipótesis nula H0 (valor p=0.43576). Por tanto, los estudiantes tuvieron el mismo rendimiento independientemente de si se les enseñó con metáforas tradicionales inconexas o con una alegoría.

Cuadro 3.9. Análisis de varianza del método de enseñanza

| **Fuente** | **GL** | **SC** | **CM** | **Valor f** | **Valor p** |
|---|---|---|---|---|---|
| Método de enseñanza | 1 | 252.9 | 252.86 | 0.6283 | 0.43576 |
| Bloqueo | 2 | 2879.7 | 1439.86 | 3.5775 | 0.04367* |
| Residuos | 24 | 9659.4 | 402.47 | | |

(GL=grados de libertad, SC=suma de cuadrados, CM=cuadrados medios, *p<0.05)

Como se muestra en las estadísticas descriptivas del Cuadro 3.10, el rendimiento esperado de los estudiantes en resolver la tarea usando el video con metáforas tradicionales fue de un 57.3%. El video que usó una alegoría incrementó el rendimiento esperado a 63.3%. La Figura 3.20 muestra en un gráfico de cajas cómo los dos factores tuvieron un efecto similar.

Cuadro 3.10. Estadísticas descriptivas de la variable rendimiento

| | **n** | **mediana** | **promedio** | **desviación estándar** | **Intervalo de confianza** | |
|---|---|---|---|---|---|---|
| | | | | | **inferior** | **superior** |
| Control (metáforas) | 13 | 60.00 | 57.32 | 20.98 | 44.63 | 69.98 |
| Tratamiento (alegoría) | 15 | 62.50 | 63.33 | 22.77 | 50.72 | 75.94 |

En el análisis de datos se bloqueó por réplica dado que cada grupo del curso es un importante factor de variabilidad debido a razones ambientales, tales como cultura, profesor, estudiantes, horario y otras fuentes de ruido. El análisis de varianza en el Cuadro 3.9 estadísticamente confirmó esta sospecha al obtener una varianza significativa producida por la réplica (valor p=0.04367, $\alpha$=0.05). Este es un resultado interesante que podría enfatizar la importancia del contexto para la efectividad de las metáforas, pero más investigación es requerida, dado que la réplica fue un bloque en lugar de un factor en el diseño del experimento.

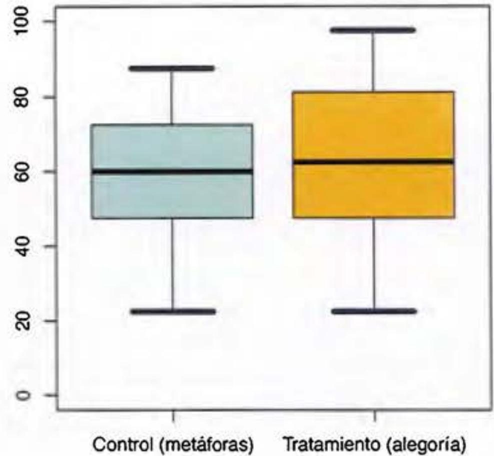



Figura 3.20. Gráfico de cajas del rendimiento del grupo control y tratamiento

El análisis de varianza es sensible a la normalidad de los residuos y la igualdad de varianzas. Para confirmar que los datos cumplen el supuesto de normalidad, se realizó una prueba de Shapiro-Wilk y no se encontró sustento estadístico para rechazar la hipótesis de que los residuos están normalmente distribuidos (valor p=0.1534). De la misma forma, una prueba de Bartlett no encontró evidencia suficiente para rechazar la homogeneidad de varianzas entre el grupo control y el grupo tratamiento (valor p=0.7731).

Varios trabajos previos han comparado el rendimiento de principiantes contra expertos. En este experimento se tuvo una población de estudiantes novatos en su primer curso de programación de la carrera. Sin embargo, en la Figura 3.21 se comparó el rendimiento que los participantes tuvieron en el experimento (eje y) contra la nota final que tuvieron en el curso de "Programación I" (eje x). Mediante un análisis de conglomerados se identificaron dos poblaciones en la Figura 3.21: estudiantes de bajo rendimiento y estudiantes de alto rendimiento, tanto en el grupo control como el grupo tratamiento. La distribución en la Figura 3.21 muestra que las alegorías influenciaron un rendimiento en el experimento ligeramente menor en estudiantes de bajo rendimiento en el curso, e influenciaron un rendimiento en el experimento mayor para estudiantes con alto rendimiento en el curso. Este resultado puede ser mapeado a la distribución de rendimiento entre principiantes y expertos reportada por Hsu, y por tanto, sustentando sus conclusiones [Hsu 2006].

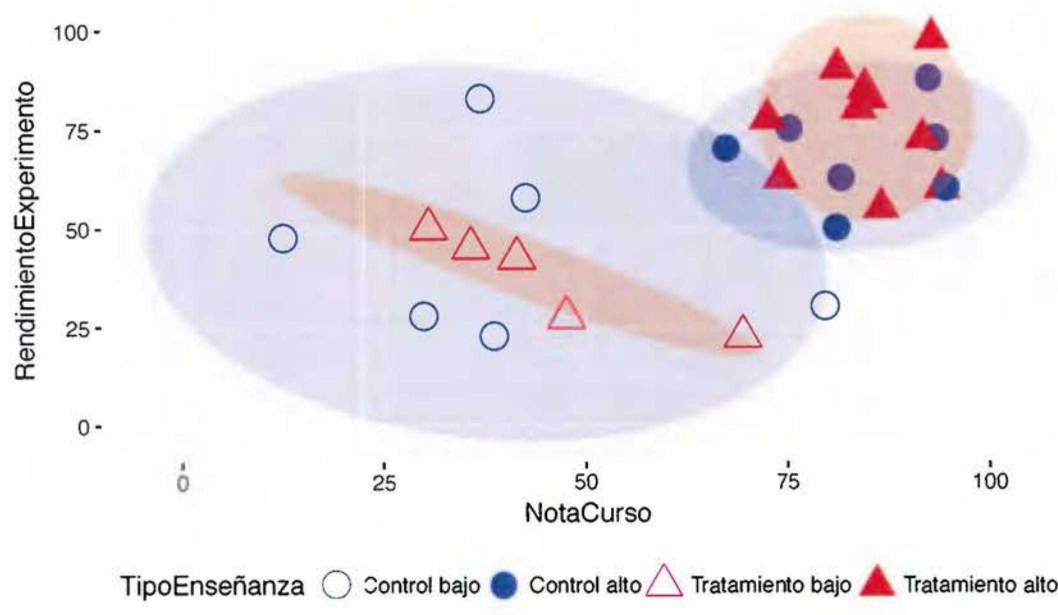



Figura 3.21. Distribución de participantes por nota del curso y rendimiento en el experimento

Para saber si la instrucción alegórica afecta aspectos motivacionales del proceso de aprendizaje (segunda pregunta de investigación del experimento), se solicitó a los estudiantes reportar sus percepciones sobre el material de enseñanza que experimentaron. Tras finalizar la sesión experimental, los participantes respondieron un instrumento llamado diferencial semántico (en inglés, *SEMANTIC DIFFERENTIAL*) [Osgood 1964]. Se evaluaron seis aspectos motivacionales del método de enseñanza usado en el video. Para cada aspecto, los estudiantes escogieron una posición entre dos adjetivos bipolares, por ejemplo, entre aburrido e interesante. Cada aspecto tuvo una escala de cinco pasos. Los datos se codificaron en un rango de -2 (percepción muy negativa) a 2 (percepción muy positiva). Como puede verse en la Figura 3.22, los estudiantes consideraron que ambos videos fueron muy comprensibles, muy útiles, muy didácticos, algo agradables, y ligeramente interesantes. Sin embargo, el video alegórico fue considerado ligeramente más interesante que el tradicional. Los estudiantes encontraron que ambos videos tienen una velocidad normal.

Se realizaron seis análisis de varianza (*ANOVA*) de un factor para comparar el efecto del método de enseñanza (video metafórico o alegórico) en cada aspecto motivacional, pero no se encontraron diferencias significativas entre los dos tipos de enseñanza. Se repitió este análisis para el nivel de aprendizaje auto-reportado por los estudiantes en una escala Likert entre 1

(nada) y 5 (mucho), pero tampoco hubo diferencias significativas entre los dos tipos de enseñanza.

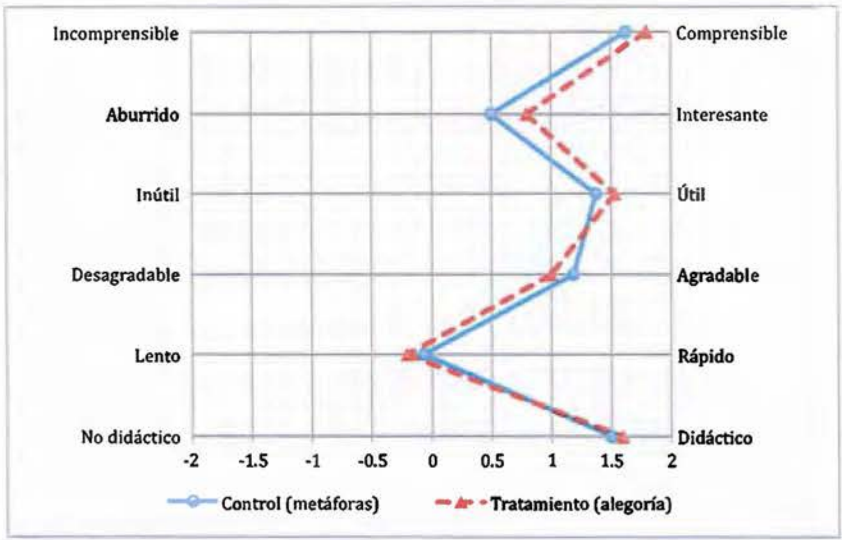



Figura 3.22. Percepción emotiva de los estudiantes por el método de enseñanza experimentado

## 3.5.5 Discusión de los resultados experimentales

Los resultados estadísticos obtenidos mostraron que los estudiantes tuvieron el mismo rendimiento resolviendo una tarea de programación, indiferentemente de si el conocimiento requerido fue explicado usando alegorías o metáforas tradicionales inconexas. Por una parte, estos resultados no confirman algunos potenciales beneficios de las metáforas tradicionales disímiles sobre las alegorías reportados por Blackwell, tales como crear significados, mejorar el aprendizaje, o evitar abrumarse [Blackwell 2006]. Por otra parte, los resultados indican que beneficios potenciales de las alegorías podrían no ser inmediatos y se requieren análisis adicionales a partir de las teorías.

Las principales explicaciones teóricas para la igualdad de rendimiento entre metáforas y alegorías son el conocimiento previo y el contexto. Tanto la teoría de la metáfora conceptual como el constructivismo sociocultural enfatizan la importancia del conocimiento previo en la mente de los estudiantes para asociar las nuevas nociones [Bogoiavlenski et al. 1963; Lakoff 1993; Lakoff and Johnson 2003]. En este experimento, las nuevas nociones sobre programación concurrente son todas metáforas. Independientemente de si estas nuevas metáforas están coherentemente interrelacionadas (alegoría) o no (metáforas tradicionales)

en el dominio origen, no se puede controlar el conjunto de conceptos que los estudiantes utilizaron para asociarlas. Investigación adicional sobre nociones previas relacionadas con las metáforas en computación podría dar luz en este aspecto.

Aunque el constructivismo sociocultural y la teoría de metáfora conceptual no desestiman la importancia del contexto, algunas teorías recientes las enriquecen al agregar el contexto como un factor o como el objeto de estudio. Dos ejemplos son la teoría de aprendizaje situado, y el comportamiento de la organización de la información humana (*HIOB*, del inglés *HUMAN INFORMATION ORGANIZING BEHAVIOR*). El segundo es un campo de la teoría de comportamiento de la información. Desde la perspectiva de *HIOB*, Sease afirma que "el contexto en el que se hace la declaración, el lugar en el cual es interpretada, y la motivación del usuario para comprenderla se combinan para afectar el significado con que es interpretada" (traducido de [Sease 2008, p.11]). El contexto podría ser la principal explicación de la igualdad de efecto entre las metáforas y las alegorías. De hecho, la influencia estadísticamente significativa del bloqueo por replicación en el rendimiento de los estudiantes, podría explicarse por estas nuevas teorías. Cada replicación del experimento fue un contexto completamente distinto, influenciado por un profesor, un grupo de estudiantes, sus establecidas relaciones interpersonales, y los eventos durante la sesión experimental.

La revisión de literatura en la sección anterior encontró que la carencia de diferencias significativas entre alegorías y metáforas es el resultado más reportado en la literatura científica en computación (en un 73% de las variables medidas). Los resultados del experimento confirman este hecho y agregan que es indiferente de la modalidad de la metáfora (visual, textual, oral, o multimodal).

Hsu concluyó que las alegorías parecen deteriorar el rendimiento de los novatos, ya que ellos no tienen sistemas de conceptos establecidos para asociar las nuevas nociones, mientras que las alegorías mejoran el rendimiento de los expertos porque les ayuda a recordar sus nociones ya establecidas [Hsu 2006]. El experimento realizado encontró resultados similares al comparar su rendimiento en el experimento contra las notas finales que obtuvieron en el curso. Si embargo, este podría ser un resultado anecdótico que podría ser evaluado con otros métodos de investigación.

Estudios previos han reportado las alegorías textuales como verbosas [Hsu 2007] y más difícil de presentar debido a que son menos familiares que las metáforas tradicionales en

computación [Blackwell 2001]. El experimento confirmó ambas situaciones al obtener un video alegórico más extenso y pausado que el tradicional. Se esperó que este hecho afectara la percepción motivacional de los estudiantes. Sin embargo, los resultados mostraron que los estudiantes consideraron ambos videos motivacionalmente casi idénticos. Análisis de varianza adicionales confirmaron este resultado.

En resumen, la revisión de literatura de la sección anterior encontró que las alegorías tienen en general el mismo efecto que las metáforas tradicionales. Este resultado fue también comprobado experimentalmente en estudiantes de la Escuela de Ciencias de la Computación e Informática. Por tanto, no se confirmaron las advertencias de Blackwell sobre efectos negativos de las alegorías en computación. Con este resultado se procede en el próximo capítulo a enriquecer visualizaciones de programa con alegorías y ludificación para satisfacer los requerimientos de software inferidos de la teoría de aprendizaje.

# 4 DISEÑO Y VALIDACIÓN DE VISUALIZACIONES LÚDICAS

Este capítulo desarrolla el segundo y tercer objetivo de esta investigación. El segundo objetivo consiste en "Proponer cambios conceptuales a las visualizaciones de programa para satisfacer requerimientos de software seleccionados". El procedimiento para obtener la propuesta conceptual se encuentra en la parte superior de la Figura 4.1 (pasos ① a ③) y se desarrolla en la sección 4.1. Por claridad se acompaña la propuesta conceptual con dos ejemplos producto del objetivo especifico 3: "Diseñar una visualización de programa para un contexto específico que aplique los cambios computacionales propuestos en el objetivo 2", cuya metodología se encuentra en la parte inferior de la Figura 4.1. Mediante una encuesta ④ se encontró que la máquina nocicnal de C++ es la más usada por los estudiantes en los cursos de la ECCI (sección 4.2), por tanto se diseñó ⑤ una visualización de programa para C++ basada en la alegoría de un teatro de títeres (sección 4.3), que luego fue sustituida por robots en una fábrica (sección 4.4), en una segunda iteración del ciclo de validación ⑥.

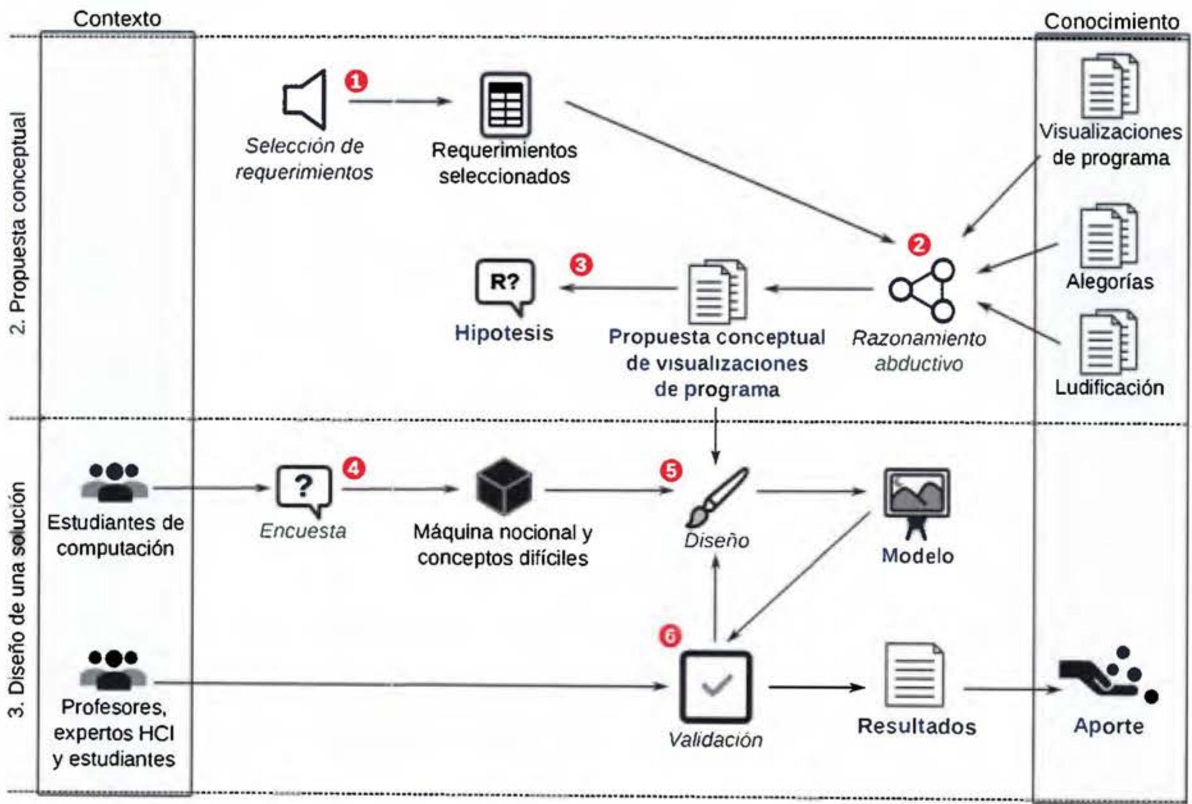



Figura 4.1. Metodología de los objetivos 2 y 3

## 4.1 Propuesta conceptual

De acuerdo a la metodología de la Figura 4.1, el primer paso para obtener la propuesta conceptual es la selección de los requerimientos de software de alto nivel a ser atendidos ①. Este es un paso metodológico heredado de la propuesta de tesis. Durante el desarrollo de la investigación se encontró que la totalidad, es decir, los 16 requerimientos de software (resumidos en el Cuadro 3.5, p.71), se pueden satisfacer conceptualmente con las teorías escogidas: alegorías y ludificación. La trazabilidad entre estas dos teorías y los requerimientos de software que satisfacen se expone a lo largo de este capítulo.

El segundo paso en la metodología (rotulado ② en la Figura 4.1), proveniente de la ciencia del diseño, es la fase creativa del proceso ingenieril, que mediante un razonamiento abductivo genera sugerencias para resolver un problema [Vaishnavi and Kuechler Jr. 2015, p.18]. Mediante abducción se conjeturó que los requerimientos inferidos de la teoría de aprendizaje pueden ser satisfechos enriqueciendo las visualizaciones de programa con alegorías y ludificación. Dado que estas elecciones no están justificadas en otros trabajos de la literatura científica, conforman entonces una hipótesis de investigación (Figura 4.1). Es decir, si estas justificaciones existieran en trabajos previos, no sería aporte de esta investigación. Por tanto, la curiosidad científica de que las alegorías y la ludificación puedan ayudar a crear visualizaciones de programa eficaces, es la tesis o hipótesis principal de esta investigación.

La Figura 4.2 mapea los 16 requerimientos de software (en la parte izquierda), con alegorías y los 12 elementos del juego (en la parte derecha) que se conjeturan pueden usarse para satisfacerlos. La discusión de los mapeos y ejemplos de los mismos inician en esta sección pero se extienden a lo largo del capítulo, por tanto múltiples referencias se hacen a esta figura de trazabilidad de requerimientos.

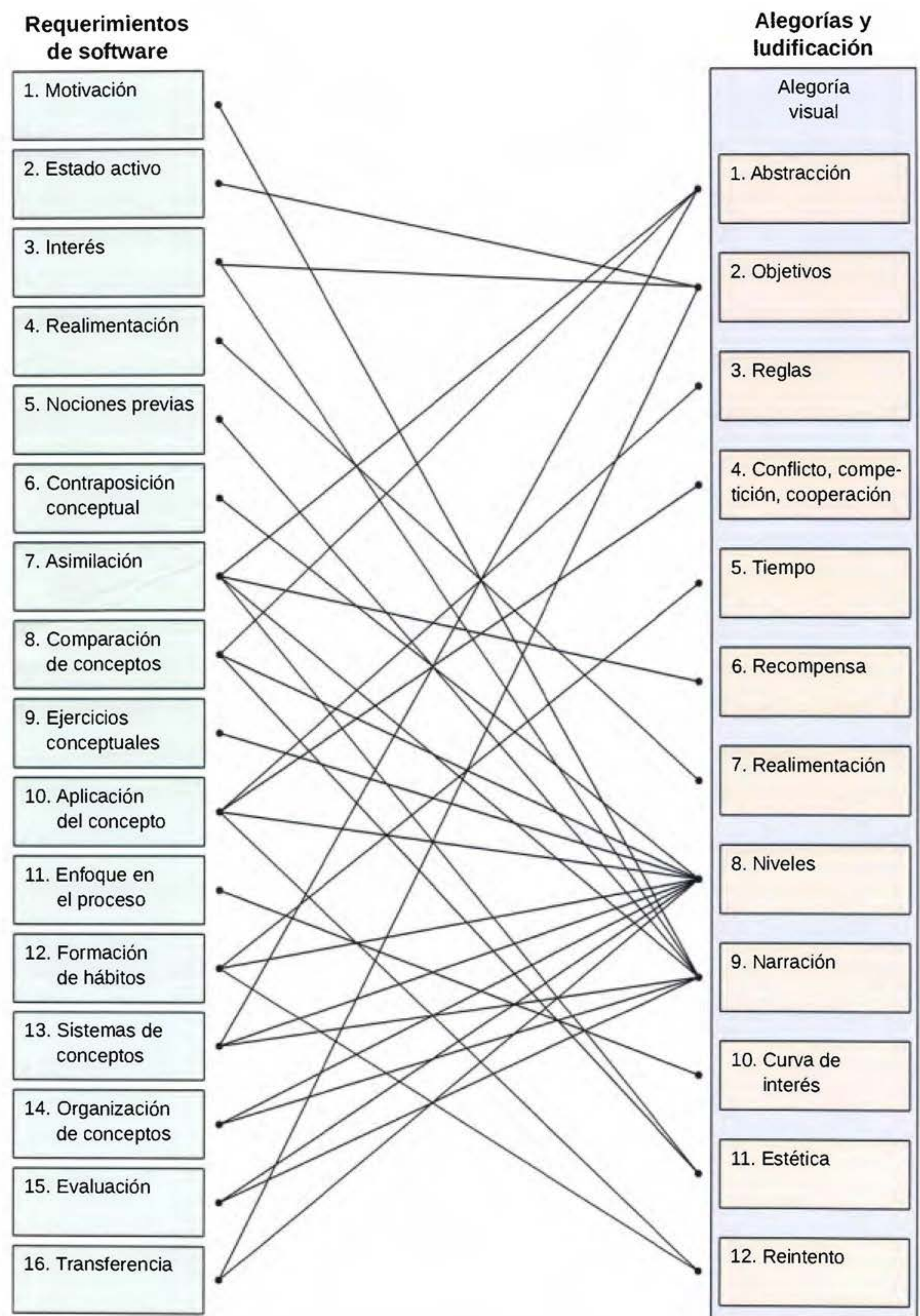



Figura 4.2. Trazabilidad de los 16 requerimientos contra los 12 elementos lúdicos y alegorías

De acuerdo a Kapp, el primer elemento lúdico es la abstracción (anexo 1). Este elemento establece que los juegos no son el mundo real sino que usan generalizaciones o *modelos* para representar la realidad [Kapp 2012, p.26]. Por otra parte, la realidad de una visualización de programa es la máquina nocional de un lenguaje de programación. Dado que esta realidad es intangible, las visualizaciones de programa tienen que emplear imágenes ópticas para representarla. El diseñador de la visualización debe escoger estas imágenes, las cuales por no ser la realidad son metáforas visuales de la realidad.

Al ludificar una visualización de programa, se debe abstraer la realidad (la máquina nocional) creando un *modelo* de ella. Un **modelo** es "un conjunto de proposiciones o declaraciones que expresan relaciones entre constructos" (traducido de [Vaishnavi and Kuechler Jr. 2015, p.21]) y los **constructos** son *conceptos* en el vocabulario de un dominio o problema [Vaishnavi and Kuechler Jr. 2015, p.20]. Por definición, una metáfora es un mapeo estructurado entre un dominio origen y uno destino que permite expresar el destino en términos del origen [Lakoff 1993]. Si las interrelaciones entre los *conceptos* del dominio origen son coherentes con las interrelaciones de los *conceptos* del dominio destino, es decir, una alegoría, se cumple la definición anterior de *modelo* y por tanto, una alegoría visual es una abstracción de la realidad. Por este razonamiento, la propuesta conceptual de esta tesis sugiere que para ludificar una visualización de programa, se abstraiga su máquina nocional con una alegoría visual.

El éxito de la ludificación radica en la interrelación de los elementos lúdicos [Kapp 2012]. La abstracción propuesta (alegoría visual) sustenta a los demás once elementos lúdicos, porque provee los conceptos con que los demás elementos serán diseñados, y facilita este diseño gracias a su discurso origen coherente. Por ejemplo, la creación de una narración (historia) coherente que pueda sustentar una curva de interés en el tiempo. Por tanto, los elementos lúdicos se ilustran encima de la alegoría en la Figura 4.2. Resulta apremiante, entonces, que la elección de la alegoría visual de la máquina nocional sea la primera tarea de la fase de diseño de una visualización lúdica de programa. Esta recomendación se sigue al presentar los dos diseños de visualizaciones lúdicas, más adelante en este capítulo.

Como se aprecia en la Figura 4.2, el elemento lúdico 1 que abstrae la realidad con una alegoría ayuda a satisfacer tres requerimientos. El requerimiento 7 (asimilación) solicita "Visualizar el nuevo concepto asociándolo con nociones que el estudiante ya tiene en su mente, traídas de

su experiencia de vida." Es decir, la alegoría ayuda a asociar los conceptos de programación con nociones previas. El diseñador debe escoger elementos visuales origen que tengan características comunes con los conceptos de programación destino para incrementar el éxito de la metáfora [Smilowitz 1995, p.114]. El diseñador puede escoger entre los cuatro tipos de metáforas visuales presentados en el Cuadro 3.2 (p. 63). Es decir, la elección debe efectuarse entre imágenes abstractas o concretas, y entre metáforas inconexas o alegorías.

Un análisis de metáforas en un período de 300 años de literatura inglesa, encontró una tendencia de asociar conceptos abstractos con conceptos concretos, como un intento de hacer el mundo abstracto comprensible a las personas [Smith et al. 1981]. Este hallazgo es apoyado por la teoría de la metáfora conceptual de Lakoff, que establece que las metáforas son el dispositivo cognitivo principal para referirse a lo abstracto [Lakoff 1993]. La tendencia de asociar lo abstracto con lo concreto también es apoyada por la teoría del constructivismo sociocultural, debido a que las nociones que los aprendices construyen se deben asociar a lo que hayan construido en su experiencia de vida previa [Luria et al. 2011]. Por estas razones se escogieron metáforas concretas para la propuesta conceptual, es decir, la segunda fila del Cuadro 3.2 (p. 63).

De acuerdo a los resultados de la revisión de literatura sobre alegorías en computación (sección 3.4), y los resultados del experimento que las compara contra metáforas tradicionales en estudiantes de la ECCI (sección 3.5), el efecto de usar metáforas inconexas o alegorías podría predecirse como similar. Sin embargo, dado que la metáfora escogida va a ser ludificada en la propuesta conceptual, una alegoría provee un discurso origen coherente que facilita la interacción de elementos del juego, como la creación de una narración (historia) coherente que pueda sustentar una curva de interés en el tiempo.

Al unir las elecciones de los dos párrafos anteriores, se obtiene que la propuesta conceptual de esta tesis usa *alegorías visuales concretas*. Los resultados de la revisión de literatura (sección 3.1.2) encontraron que las visualizaciones de programa existentes no utilizan alegorías concretas, sino los otros tres tipos de metáforas visuales. Una potencial explicación de esta carencia podría estar dada por [Gračanin et al. 2005], quienes afirman que el reto principal de las visualizaciones de software, es encontrar mapeos efectivos de los diferentes aspectos del software a representaciones gráficas usando metáforas visuales. Este reto es probablemente aún mayor al agregar las restricciones de los discursos coherentes de las

alegorías. Por tanto, al construir visualizaciones de programa con alegorías visuales concretas, esta tesis está haciendo un aporte innovador al conocimiento.

Al agregar ludificación a los cuatro tipos de metáforas visuales, se obtienen las ocho combinaciones del Cuadro 4.1. Dado que la ludificación se usa en combinación con alegorías visuales concretas para satisfacer los requerimientos de software, la propuesta conceptual de esta tesis es una **visualización lúdica-alegórica concreta de programa**. En este documento se usará el término simplificado *visualización lúdica de programa* como sinónimo por razones de legibilidad, resaltado en gris en el Cuadro 4.1. Una **visualización lúdica de programa** se define como una herramienta interactiva de software diseñada aplicando elementos lúdicos para motivar a los estudiantes a comprender cómo los programas corren en una máquina nocional a través de las imágenes que genera.

La propuesta conceptual de esta tesis es usar las alegorías en combinación con elementos de juego para satisfacer requerimientos de software de alto nivel. Por tanto, ambos se incluyen en la parte derecha de la Figura 4.2. Dado que es apremiante explicar los mapeos de esta figura con ejemplos, las explicaciones se delegan para las secciones que presentan las dos visualizaciones lúdicas de programa diseñadas más adelante en este capítulo. Antes se realizó una encuesta para escoger la máquina nocional a visualizar cuyos resultados se presentan en la siguiente sección.

Cuadro 4.1. Tipos de visualizaciones de programa de acuerdo al tipo de metáfora y ludificación

| # | Lúdica | Ámbito | Origen | Nombre de visualización de programa |
|---|---|---|---|---|
| 1 | No | Metáfora extendida | Abstracto | Visualización metafórica abstracta de programa |
| 2 | No | Metáfora extendida | Concreto | Visualización metafórica concreta de programa |
| 3 | No | Alegoría | Abstracto | Visualización alegórica abstracta de programa |
| 4 | No | Alegoría | Concreto | Visualización alegórica concreta de programa |
| 5 | Sí | Metáfora extendida | Abstracto | Visualización lúdica-metafórica abstracta de programa |
| 6 | Sí | Metáfora extendida | Concreto | Visualización lúdica-metafórica concreta de programa |
| 7 | Sí | Alegoría | Abstracto | Visualización lúdica-alegórica abstracta de programa |
| 8 | Sí | Alegoría | Concreto | Visualización lúdica-alegórica concreta de programa |

## 4.2 Encuesta: selección de la máquina nocional

Con el fin de escoger la máquina nocional a ser visualizada, se encuestó a la población meta de este estudio: los estudiantes de la Escuela de Ciencias de la Computación e Informática. Se les

preguntó por el lenguaje de programación que usan más en la carrera, y los temas de programación que ellos consideran más útiles y a la vez más difíciles de aprender.

La encuesta se hizo a todos los estudiantes que completaron los cinco cursos listados en el Cuadro 4.2. Estos cursos están fuertemente relacionados con programación, y se ubican desde el segundo hasta el cuarto año de la carrera. Por la dinámica de requisitos (última columna del Cuadro 4.2), un estudiante no puede estar matriculado en dos de estos cursos al mismo tiempo, por tanto no podría ser encuestado dos veces.

Cuadro 4.2. Cursos relacionados con la programación que fueron entrevistados

| **Acrónimo** | **Nombre** | **Ciclo** | **Requisito** |
|---|---|---|---|
| Progra2 | Programación II | 2.I | Progra1 |
| EDAA | Estructuras de datos y análisis de algoritmos | 2.II | Progra2 |
| BD1 | Bases de datos I | 3.I | EDAA |
| IS1 | Ingeniería de software I | 3.II | BD1 |
| IS2 | Ingeniería de software II | 4.I | IS1 |

El cuestionario auto-administrado y anónimo fue respondido por 144 estudiantes, 114 hombres (81%), 27 mujeres (19%), un estudiante que no reportó la variable sexo, y dos cuestionarios que fueron entregados incompletos. Estas tasas en la muestra reflejan la distribución por género de la población estudiantil.

El curso de "Programación I" (Progra1) se enseña con *JAVA* y el curso de "Programación II" se enseña con C++. Se les pidió a los estudiantes el porcentaje aproximado de uso de éstos y otros lenguajes. En promedio, C++ fue reportado como el lenguaje de programación más usado (51%), seguido por *JAVA* (33%). El uso de los lenguajes varía a través de los cursos debido a las preferencias de los cursos o de los profesores como se aprecia en la Figura 4.3. En el curso de "Estructura de datos y análisis de algoritmos" (EDAA) se prefiere C++, en los cursos de ingeniería de software (IS1, IS2) se prefiere *JAVA*. Como es de esperarse, el uso de otros lenguajes de programación, tales como SQL, *JAVASCRIPT*, *LISP* y *C#*, incrementa conforme los estudiantes avanzan por la carrera.

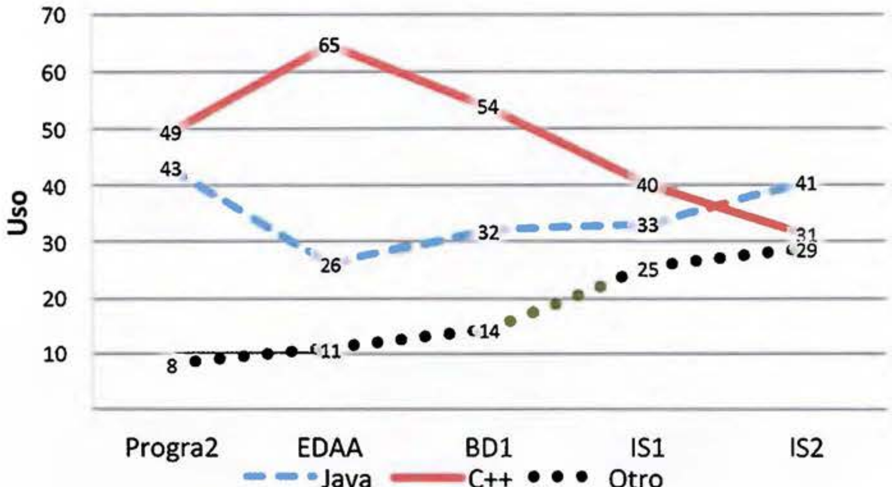



Figura 4.3. Uso de lenguajes de programación a través de los cursos

Se les solicitó a los estudiantes calificar el grado de dificultad de aprendizaje que ellos habían experimentado con ciertas familias de conceptos, en una escala de 1 a 10, donde 1 significa ninguna dificultad y 10 la mayor dificultad. También se les solicitó reportar la utilidad que ellos han encontrado de esas familias de conceptos. Los resultados se resumen en la Figura 4.4. Esas variables sirven para priorizar aquellos conceptos de programación a ser visualizados, dado que no se quiere priorizar conceptos difíciles que no son considerados útiles o viceversa. Por tanto se multiplicó la dificultad y la utilidad de cada familia de conceptos y al resultado se le llamó *relevancia de aprendizaje*. La Figura 4.4 está ordenada en dirección de las manecillas del reloj por esta relevancia.

De la Figura 4.4 puede inferirse que los conceptos de programación concurrente o paralela fueron considerados como los más difíciles (7.15 de 10) y útiles (7.8 de 10) de aprender. Los conceptos de administración de memoria fueron los segundos más útiles (8.9 de 10), a la vez que difíciles (6.0 de 10) de aprender. Este resultado es congruente con otros trabajos en la literatura científica que reportan los punteros y administración de la memoria en C o C++ como difíciles de aprender (por ejemplo [Craig and Petersen 2016; Milne and Rowe 2002; Lahtinen et al. 2005; Egan and McDonald 2013; Allevato et al. 2009]).

Figura 4.4. Familias de conceptos considerados difíciles pero útiles de aprender

En síntesis, los resultados de la encuesta reportaron que C++ es el lenguaje de programación más usado en los cursos de carrera por los estudiantes de la ECCI, y los temas de programación concurrente y administración de memoria son considerados como los más relevantes de aprender. En la siguiente sección se diseña una visualización lúdica de programa para la máquina nocional de C++ orientada a estos conceptos considerados relevantes.

## 4.3 Diseño 1: *PUPPETEER++*

En la primera iteración del ciclo de diseño (entre pasos ⑤ y ⑥ de la Figura 4.1, p.106) se recurrió a una alegoría de teatro de títeres para representar concurrencia y administración de memoria en C++. En la subsección 4.3.1 se explica el diseño de la alegoría visual y en la subsección 4.3.2 su ludificación. Finalmente en la subsección 4.3.3 su validación y resultados.

### 4.3.1 Alegoría visual de *PUPPETEER++*

Dado que la encuesta preguntó por familias de conceptos, en lugar de conceptos particulares, el Cuadro 4.3 lista y describe los conceptos de concurrencia y administración de memoria que fueron seleccionados para representar en la visualización lúdica de programa. De acuerdo a la propuesta conceptual (Figura 4.2, p.108), una alegoría se usa para presentar los conceptos abstractos de programación asociándolos con conceptos ordinarios (requerimiento 7). La elección de la alegoría se hizo por similitud, tratando de encontrar características comunes entre el dominio destino (los conceptos del Cuadro 4.3) y el dominio origen a escoger, como ha sido recomendado por [Smilowitz 1995, p.114]. A continuación se explican las similitudes entre ambos dominios.

Cuadro 4.3. Conceptos de programación representados en la alegoría del teatro de títeres

| Concepto | Descripción |
|---|---|
| Hilo de ejecución | Es un conjunto de valores que permiten a un núcleo de un procesador ejecutar instrucciones de código independientemente de otras instrucciones en ejecución. |
| Memoria compartida | Memoria que puede ser accedida simultáneamente por dos o más hilos de ejecución. |
| Segmento de memoria | Secciones de memoria asignadas con un propósito particular. Típicamente un programa dispone de segmentos de código, datos, pila y memoria dinámica (*HEAP*). Los sistemas operativos imponen límites en sus tamaños, excepto el último de ellos. En C/C++ los punteros son requeridos para acceder a la memoria dinámica. Los hilos de ejecución de un mismo programa comparten todos los segmentos excepto sus pilas. |
| Puntero | Una variable entera que almacena la dirección de algún dato ubicado en la memoria. El puntero tiene un tipo de datos que le permite interpretar la memoria apuntada por su valor. |
| Invocación de función | Acción de ejecutar el código de una subrutina. Una invocación de función puede recibir parámetros y generar un resultado. |

El razonamiento inició con los segmentos de memoria, los cuales se asemejan a áreas de la memoria para almacenar objetos. Los segmentos tienen tamaños limitados, excepto la memoria dinámica cuyo tamaño se puede pensar como un área mucho mayor pero no infinita. Un hilo de ejecución se asemeja a un trabajador. Los trabajadores llevan a cabo tareas, pero su área de trabajo es limitada (el segmento de pila). Para tener acceso al área grande de almacenamiento (memoria dinámica) requieren un mecanismo de acceso especial (puntero). Dos o más trabajadores pueden acceder simultáneamente a los objetos en el área grande, pero cada trabajador no comparte su área de trabajo directo (segmento de pila).

Se escogió el teatro de títeres (Figura 4.5) como alegoría por el siguiente razonamiento. Los títeres son caracteres inanimados que actúan en un escenario (segmento de memoria dinámica) controlados por titiriteros (hilos de ejecución). Los titiriteros no se supone que actúen, por tanto, no aparecen en el escenario, sino que trabajan sobre una plataforma (segmento de pila) en la parte superior del teatro, ocultos a la audiencia. Un titiritero controla su títere en el escenario a través de cuerdas (punteros). Los titiriteros animan sus marionetas siguiendo un libreto (segmento de código) al pie de la letra. Dado que los titiriteros (en inglés, *PUPPETEERS*) hablan en un idioma llamado C++, los libretos deben estar escritos en este lenguaje. Por esta razón se le dio el nombre *PUPPETEER++* a la visualización lúdica de programa.

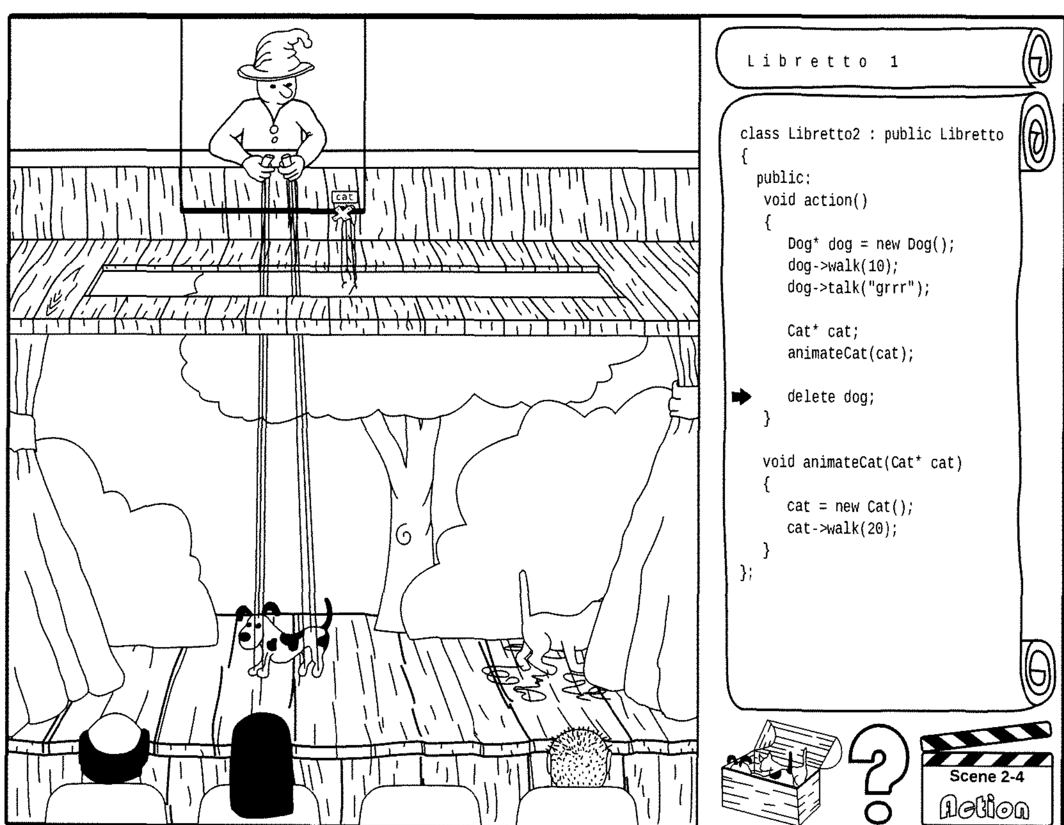



Figura 4.5. Prototipo en papel de *PUPPETEER++*

Un titiritero puede controlar varias marionetas, pero no al mismo tiempo. Para poder cambiar de un títere a otro, el titiritero dispone de un perchero (invocación a función) que sirve para sujetar los títeres inactivos (Figura 4.6). Un perchero sostiene los títeres involucrados en una acción que puede repetirse en varios lugares del libreto (funciones). Estas acciones pueden

iniciar otras, por tanto los percheros pueden apilarse (segmento de pila) y el titiritero trabaja únicamente con el perchero que tenga en la parte superior.

Para un titiritero, el cambio de una marioneta a otra introduce una espera visible a la audiencia. Si varios títeres deben actuar simultáneamente en la misma escena, varios titiriteros podrían trabajar juntos (concurrencia), como se ilustra en la Figura 4.6. Una marioneta larga, por ejemplo, un dragón chino, requiere la acción coordinada de varios titiriteros (memoria compartida).

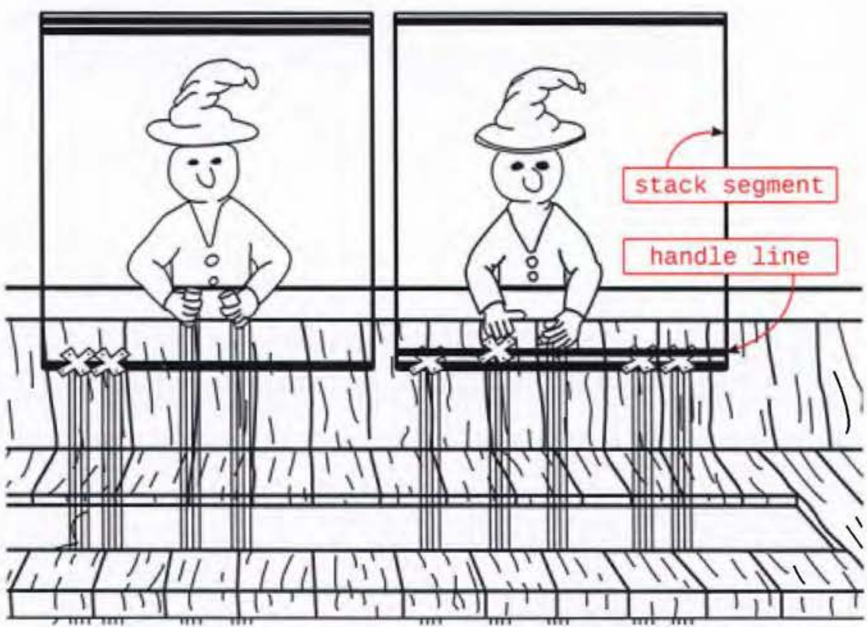



Figura 4.6. Dos titiriteros controlando varias marionetas

Una vez obtenida una alegoría para representar conceptos de programación, el siguiente paso, de acuerdo a la propuesta conceptual, es la ludificación de la alegoría para satisfacer los requerimientos de software.

### 4.3.2 Ludificación de *Puppeteer++*

En esta subsección, se proveen ejemplos de cómo los elementos lúdicos pueden usarse para satisfacer los requerimientos. Con estos ejemplos (y los del diseño en la segunda iteración del ciclo de diseño) se trazaron las asociaciones en la Figura 4.2 (p.108) de la propuesta conceptual al inicio de este capítulo.

Estimular el interés del estudiante (requerimiento 3) puede apoyarse con la narración (elemento lúdico 9), la cual puede mostrar obras de teatro incompletas y retar al estudiante a completarlas o producir las propias (elemento lúdico 2: objetivos). De esta forma, el estudiante adopta el rol de dramaturgo (requerimiento 2: estado activo) en lugar de sólo un espectador pasivo de la visualización.

La pantalla de bienvenida en la Figura 4.7(a) muestra las obras de teatro disponibles como ventanillas. Los estudiantes pueden crear nuevas obras presionando el botón con el signo de suma (+). Inicialmente no sabrán cómo escribir obras de teatro (programas), lo que apoya la contraposición conceptual (requerimiento 6) y necesitarán andamiaje para superarla. Se incluye una obra con este fin, titulada "Training" en la Figura 4.7(a), la cual provee una historia en lenguaje natural que los estudiantes traducen al lenguaje de los titiriteros (C++).

Después de seleccionar una obra en la pantalla de bienvenida (Figura 4.7(a)), la visualización lúdica muestra sus escenas (elemento lúdico 8: niveles) representadas como claquetas en la Figura 4.7(b). Inicialmente cada escena está incompleta, sin traducir, lo que se representa con una claqueta abierta en la Figura 4.7(b). Las escenas completadas pueden reproducirse en secuencia con el convencional botón en forma de triángulo en la parte superior derecha de la Figura 4.7(b).

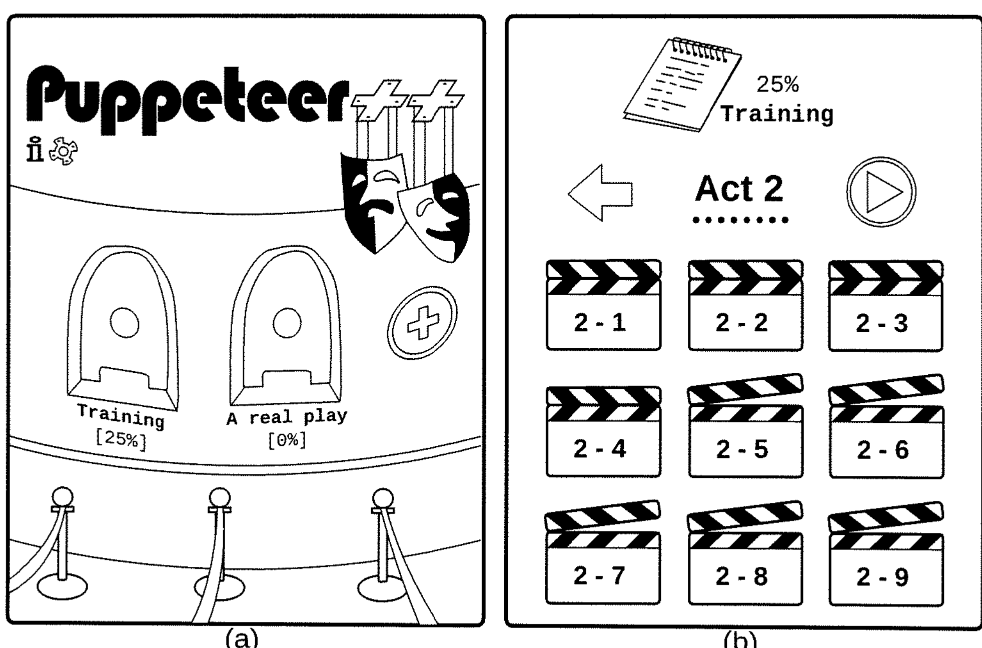



Figura 4.7. Seleccionar una obra. (b) Seleccionar una escena

Como puede verse en la Figura 4.2 (p.108) de la propuesta conceptual, el elemento lúdico de niveles puede usarse para satisfacer parcialmente la mayoría de los requerimientos de software. Los niveles satisfacen el requerimiento 14 de organizar los conceptos de programación. En *PUPPETEER++* las escenas (niveles) se agrupan en actos (mundos), como el segundo acto en la Figura 4.7(b). En la obra "Training", cada acto introduce un nuevo concepto de programación. Por ejemplo, el acto 1 introduce invocaciones a funciones. La escena 1-1 tiene un títere en la escena y reta al estudiante a instruir al titiritero para que el títere salude a la audiencia (decir "hola mundo" invocando un método). La escena 1-2 solicita mover el títere en el escenario, y así sucesivamente.

El acto 2 introduce el concepto de creación y eliminación de títeres del escenario (memoria dinámica). En cada escena (nivel) se satisfacen varios requisitos al unirse con otros elementos lúdicos. Por ejemplo, al entrar en la escena 2-1 se reta al estudiante a crear un títere, decir algunas palabras a la audiencia, moverlo detrás de las cortinas, y eliminarlo del escenario (elemento lúdico 4: conflicto). Esta solicitud puede generar contraposición conceptual (requerimiento 6) si el estudiante desconoce cómo realizar las nuevas operaciones (crear y eliminar objetos). El botón de ayuda con forma de signo de pregunta en la parte inferior derecha de la Figura 4.5 satisface el requerimiento 7 (asimilación), al presentar información sobre la creación de objetos en C++ y su efecto en el escenario, es decir, en la máquina nocional a través de la alegoría. Esta información podría costar algún dinero ficticio que el estudiante ha ganado de la audiencia (elemento lúdico 6: recompensa). La información sobre los nuevos conceptos es asociada en memoria de corto plazo en la mente del estudiante. Al retornar a la escena, el estudiante aplicará los conceptos (requerimiento 10) para resolver el reto planteado en el nivel.

Si por alguna razón el estudiante crea un objeto local en la escena 2-1, por ejemplo la declaración `Dog dog` en C++, el títere aparecerá en el perchero (segmento de pila), la audiencia no lo verá, y la escena no estará completa. Los estudiantes tendrán una pista visual del problema. Tratar de corregir el problema en la alegoría (el títere no apareció en el escenario), implica corregir el problema en el código fuente, lo cual fortalece la asociación. El estudiante efectuará cambios en el código y la visualización lúdica le permite reintentar (elemento lúdico 12: reintento). Sin embargo, si el estudiante realiza un número considerable de reintentos con errores, la visualización lúdica podría sugerir al estudiante comparar los conceptos

involucrados (requerimiento 8), en este caso los segmentos de pila y memoria dinámica, y cómo estos se acceden desde el lenguaje de programación.

Cuando el estudiante use memoria dinámica en la escena 2-1, por ejemplo `Dog* dog = new Dog()`, el títere aparecerá en el escenario, detrás de las cortinas, como se esperaba. El titiritero sostendrá en sus manos un manejador rotulado "`dog`" conectado al títere a través de cuerdas. La escena 2-1 aún no está completa. Cuando la cortina se cierra, es obligatorio que el escenario esté limpio con el fin de que esté listo para la próxima escena (elemento lúdico 3: reglas). Cuando el estudiante realmente haya eliminado todos los títeres (con el operador `delete` de C++), la escena estará completa y recibirá el aplauso de la audiencia (requerimiento 4 y elemento lúdico 7: realimentación).

La escena 2-2 reta a los estudiantes a crear diferentes tipos de títeres. La escena 2-3 reta a crear un títere y animarlo mediante la invocación de métodos usados en el acto 1. La escena 2-4 reta a crear dos títeres y animarlos uno después del otro. Cada una de estas escenas conserva el principio fundamental del concepto introducido en el segundo acto: creación y eliminación de objetos, y lo aplica a distintas situaciones (requerimiento 10: aplicación del concepto implementado con el elemento lúdico 8: niveles en la Figura 4.2, p.108). Al crear y destruir objetos una y otra vez a lo largo de los niveles, las primeras reacciones difusas del estudiante se convertirán en acciones casi mecánicas al final del acto (requerimiento 12: formación de hábitos).

Los conceptos introducidos en actos previos son reaplicados en actos posteriores. Por ejemplo, el acto 3 introduce el concepto de concurrencia: dos o más titiriteros animando marionetas. Los conceptos del acto 1 (invocación de métodos para crear animaciones) y del acto 2 (creación y destrucción de objetos) deben re-aplicarse en el acto 3. De esta forma, los nuevos conceptos son asociados a los existentes lo que forma sistemas de conceptos (requerimiento 13). El acto 4 reta a los estudiantes a crear sus propios títeres, y el acto 5 ayuda a los estudiantes a crear sus propios escenarios. Tras completar la obra "*TRAINING*", los estudiantes pueden aplicar sus sistemas de conceptos para construir una obra "real" de teatro, bajo el título "*A REAL PLAY*" en la pantalla de bienvenida de la Figura 4.7(a) o para construir sus propias obras (requerimiento 16: transferencia, implementado con niveles y objetivos). Crear una obra de teatro es una tarea laboriosa. Obras elaboradas pueden construirse en colaboración, lo cual apoya el proceso de aprendizaje socialmente mediado del

constructivismo sociocultural. Por ejemplo, los estudiantes comparten el código de sus títeres y unen escenas para conformar la obra.

La visualización lúdica de programa *PUPPETEER++* no está limitada a concurrencia y administración de memoria. Otros conceptos de la máquina nocional pueden ser visualizados como depuración, y herencia y polimorfismo, que son la cuarta y quinta familia de conceptos más relevantes a visualizar de acuerdo a los resultados de la encuesta (Figura 4.4, p.114).

La depuración es el proceso metódico de detectar y corregir errores. Cada vez que una escena está en acción, la línea siendo ejecutada por un titiritero es resaltada en el libreto (segmento de código). Los estudiantes pueden pausar la obra, y correrla línea por línea. Cuando algo anómalo se encuentra en el código, la realimentación se refleja en la alegoría visual. Por ejemplo, en la Figura 4.5 el puntero en el segmento de pila hacia el títere de gato se perdió cuando un método terminó su ejecución, pero el objeto apuntado no fue eliminado (una fuga de memoria). Las cuerdas que ataban el manejador con el gato son visualmente cortadas.

Conceptos de herencia y el polimorfismo son requeridos para que los estudiantes puedan crear sus propias marionetas y escenarios. La visualización de programa provee clases especiales como `Puppet` y `Scenery` que los estudiantes pueden reutilizar. La alegoría permite otros retos avanzados de programación. Por ejemplo, los titiriteros siguen los libretos al pie de la letra, pero si el código toma decisiones basadas en variables aleatorias, la obra puede tomar diferentes cursos de eventos en cada ejecución. En algunas obras reales de teatro, los actores invitan a la audiencia a participar de la obra. En el caso de *PUPPETEER++* la audiencia es el usuario real frente a la computadora, quien puede ser invitado a proveer valores de entrada.

La visualización de programa *PUPPETEER++* hasta aquí discutida es un modelo. De acuerdo a la metodología de la ciencia del diseño (Figura 4.1, p.106), es apremiante validar el modelo antes de pasar a la fase de implementación, lo cual se hace en la siguiente subsección.

### 4.3.3 Validación de *PUPPETEER++*

Las teorías de diseño consisten de conocimiento sobre la interacción de un artefacto con su contexto pretendido, el cual permite hacer predicciones. La validación trata de adelantar estas predicciones con opiniones de expertos u otros métodos [Wieringa 2014, p.62]. Para validar el

diseño de *PUPPETEER++* se realizó un grupo focal con todos los profesores de "Programación II" de la ECCI (excepto el autor de esta tesis).

El grupo consistió de 7 profesores, con experiencias en la enseñanza universitaria entre los 9 y 34 años. El número de veces que habían impartido "Programación II" variaba entre 1 semestre a 28 semestres. Por restricciones de horario, el grupo focal se realizó en dos sesiones: el 9 de abril de 2014 con cuatro profesores, y al día siguiente con tres profesores.

Las sesiones se organizaron de acuerdo a siete preguntas, las cuales fueron proyectadas usando diapositivas. Las preguntas se muestran adelante junto con un resumen de los principales resultados. Previo a algunas preguntas, el moderador hizo una corta introducción o presentación la cual se indica entre paréntesis antes de la pregunta. En las preguntas más críticas para la validación del diseño *PUPPETEER++*, los participantes escribieron sus respuestas en forma individual antes de discutirlas grupalmente, con el fin de facilitar su recuerdo durante la discusión, y evitar contaminación por las opiniones de otros profesores.

El audio de las discusiones fue grabado con consentimiento de los participantes. Ambas sesiones del grupo focal duraron una hora y 15 minutos cada una. Se siguió el mismo proceso de análisis indicado por [Aberg 2010]. Las grabaciones se escucharon varias veces mientras el investigador tomó notas. Las notas fueron analizadas y agrupadas en temas. Se escribió un párrafo resumen por cada tema. A continuación se resumen los resultados principales por cada una de las siete preguntas realizadas.

*Pregunta 1. (Introducción) ¿Qué hace difícil el aprendizaje en "Programación II"?*

> Las razones mencionadas fueron: deficiencias de aprendizaje en los cursos previos ("Programación I"), la complejidad del lenguaje de programación (C++), inmadurez de los estudiantes, su carencia de disciplina, y sus estrategias de estudio.

*Pregunta 2. (Teorías de aprendizaje y de metáforas) ¿Cuáles metáforas utiliza usted en sus clases? ¿Siente que le han sido de utilidad?*

> La mayoría de participantes expresaron que las metáforas son importantes para el aprendizaje. Las que más utilizan son metáforas visuales abstractas como rectángulos para variables y flechas para punteros. Los participantes expresaron conocimiento de otros profesores que usan metáforas dramatizadas. Dos analogías concretas o cotidianas fueron citadas: las plantillas de C++ son sellos de goma o acciones de

copiar-pegar-buscar-reemplazar, y la máquina computacional es como la mente humana.

*Pregunta 3. ¿Qué características cree usted hace a una metáfora buena o mala para el proceso de enseñanza-aprendizaje de la programación?*

Los participantes citaron las siguientes características de una buena metáfora: (1) Familiar, relevante para los estudiantes. (2) Gráfica, visual o dramatizada. (3) Abstracta, simple, como la visualización de programa Jeliot 3 que oculta detalles innecesarios como la implementación de la biblioteca estándar. (4) Didáctica, que dibuje las estructuras de datos como lo hacen los libros. (5) Fácil de medir su impacto en experimentos controlados.

*Pregunta 4. (Presentación del diseño* PUPPETEER++*) ¿Qué fortalezas y debilidades le encuentra usted a la metáfora del titiritero para el proceso de enseñanza-aprendizaje de los conceptos de manejo de memoria y concurrencia? [Responder individualmente antes de la discusión grupal]*

Los participantes citaron algunas fortalezas y no se generó debate sobre ellas: (1) La metáfora del teatro de títeres es clara, principalmente para ilustrar la creación de instancias e invocación de métodos. (2) Es gráfica, visual. (3) Es entretenida, por tanto, despertará el interés de los estudiantes. (4) Es fácil de usar. (5) Provee realimentación inmediata. (6) Incrementa la motivación de los estudiantes.

Extensa discusión se dio sobre las debilidades de *PUPPETEER++*. Las siguientes fueron la debilidades identificadas en orden de la más a la menos discutida. (1) No ilustra colecciones, índices o iteradores. (2) No visualiza declaraciones complicadas como punteros a punteros, o punteros a vectores de instancias. (3) La metáfora no es auto-explicativa, por tanto los estudiantes deberán invertir tiempo entendiendo la metáfora, además del que tienen que dedicar a entender la realidad (la máquina). (4) Cobertura limitada de los temas oficiales del curso "Programación II". (5) La capacidad del escenario es muy limitada para muchos actores, por ejemplo "101 dálmatas". (6) ¿Es lúdico? (7) Dado que sólo un subconjunto de C++ es implementado, los estudiantes podrán construir sólo programas limitados, y no todos los programas de C++ requieren memoria dinámica. (8) La alegoría no es clara en el envío y recepción de mensajes entre objetos (títeres) y compartir información.

*Pregunta 5. ¿Qué le mejoraría?*

Las siguientes ideas fueron sugeridas: (1) Permitir a los estudiantes colaborar, por ejemplo, que ellos puedan crear diferentes títeres por separado y luego importarlos en la obra completa, de forma similar en que lo hace un director. (2) Ofrecer un repositorio oficial de títeres que permitan a los estudiantes compartir o "contratar" actores, y así acelerar el proceso de dramatización. (3) Simplificar la metáfora, por ejemplo, remover algunos distractores en el escenario como arbustos y cortinas. (4) Permitir a los estudiantes correr instrucciones sin tener que hacer un proceso completo de compilación (interpretación de código).

*Pregunta 6. ¿Plantearía una metáfora alternativa para el manejo de memoria y concurrencia?*

Se sugirieron dos metáforas alternativas. La primera metáfora podría ser cualquier escenario donde la administración de recursos (administración de memoria) y completado rápido de tareas (concurrencia) sea requerido. Por ejemplo, construcción de edificios como en *MINECRAFT* (2011). En esta metáfora los estudiantes administran simultáneamente varios trabajadores (hilos de ejecución) como albañiles y carpinteros, y recursos como materiales de construcción y dinero.

La segunda metáfora, sugerida en el primer grupo focal por los profesores con mayor experiencia de impartir "Programación II", fue más conservadora. Se sugirió un depurador simbólico visual que muestre una abstracción de la máquina computacional. La visualización de programa Jeliot 3 tiene este ideal, pero es muy limitado. Por ejemplo, Jeliot no ilustra didácticamente estructuras de datos como pilas o árboles. La nueva metáfora podría superar las limitaciones de Jeliot 3.

*Pregunta 7. ¿Cree que jugar con una herramienta como la propuesta, realmente impactará positiva o negativamente el aprendizaje y/o la motivación de los estudiantes? ¿Por qué razones? [Responder individualmente antes de la discusión grupal]*

Tres de los siete profesores expresaron que *PUPPETEER++* impactaría positivamente el aprendizaje y la motivación de los estudiantes. Un profesor expresó que tendría tanto efectos positivos como negativos. Los restantes tres profesores se mostraron inseguros de la efectividad.

A modo de discusión, los resultados del grupo focal mostraron posiciones divididas entre los profesores, lo que no permite predecir una interacción efectiva de la herramienta con su contexto. Aunque el diseño propuesto, *PUPPETEER++*, tiene claras fortalezas, fueron más numerosas y enfáticas sus debilidades. Las sugerencias y advertencias planteadas por el grupo de expertos fueron tomadas en cuenta en el diseño de un segundo modelo, presentado en la siguiente sección.

## 4.4 Diseño 2: botNeumann++

En la segunda iteración del ciclo de diseño se trató de superar las limitaciones de *PUPPETEER++*. En lugar de diseñar una visualización de programa enfocada en familias de conceptos considerados difíciles y útiles por estudiantes, se diseñó una alegoría basada en robots para representar la mayoría de los conceptos de programación de la máquina nocional de C/C++. Estos lenguajes, en especial C, son considerados una delgada capa de abstracción sobre la arquitectura subyacente. Esta arquitectura es en principio la que John von Neumann documentó en un reporte técnico que influenció la mayoría de computadoras modernas [von Neumann 1945]. Por estas razones, se le dio el nombre de *BOTNEUMANN++* al segundo diseño[16], el cual se presenta en la subsección 4.4.1 y su correspondiente validación en la subsección 4.4.3.

### 4.4.1 Alegoría visual de botNeumann++

botNeumann++ usa robots en una fábrica futurista como alegoría para explicar cómo la máquina nocional de C/C++ corre los programas. La Figura 4.8 es una captura de pantalla de un prototipo hecho con *MICROSOFT POWERPOINT* corriendo un programa en C++ que determina si vectores de enteros son simétricos o no. Las metáforas visuales individuales fueron agrupadas por familias de conceptos, y los grupos se indican con números en la Figura 4.8. En

[16] Nombre sugerido por Arturo Peña Hurtado, diseñador gráfico, quien ilustró, en coordinación con el autor de esta tesis, los componentes gráficos para botNeumann++. Las figuras mostradas en esta sección se construyeron a partir de estos componentes y de capturas de pantalla de los prototipos.

los siguientes apartados se presenta cada familia de conceptos en el mismo orden que la figura. En cada apartado se explican primero los conceptos destino en C/C++ y sus interrelaciones, luego la metáfora visual origen para cada concepto. Para hacer más claras las asociaciones, texto encerrado entre llaves se usa para marcar el {dominio destino} dentro de un discurso origen. Al final de esta subsección se dilucida cómo las metáforas visuales seleccionadas pueden formar un discurso origen que mapea la ejecución de un programa C/C++, el cual puede usarse para crear la narrativa de una visualización lúdica.

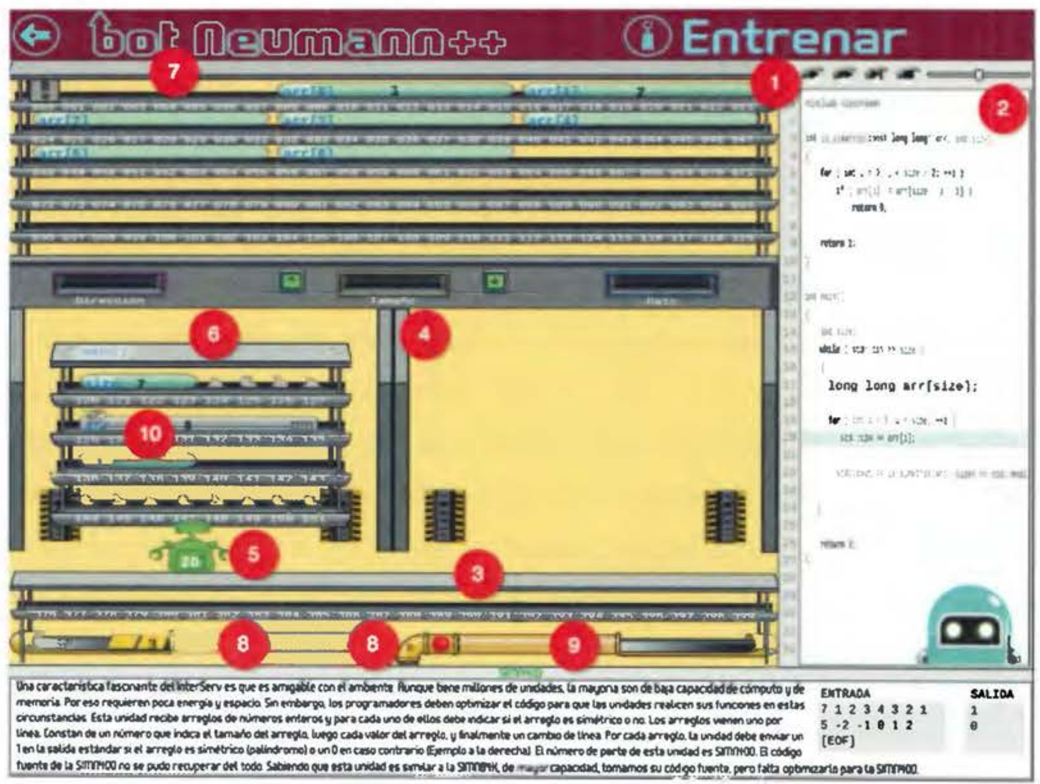


Figura 4.8. Familias de conceptos enumeradas en una captura de pantalla de *BOTNEUMANN++*

#### *4.4.1.1 Ambiente de ejecución*

El **ambiente de ejecución** de C/C++ (en inglés, *RUNTIME ENVIRONMENT*) es una delgada capa de abstracción que permite a los programas construidos en este lenguaje correr sobre múltiples combinaciones de hardware y sistemas operativos. Normalmente es implementado por el fabricante del compilador en forma de una biblioteca estática que se agrega a los ejecutables en C/C++ o una biblioteca dinámica accesible en tiempo de ejecución. El ambiente de

ejecución de C/C++ divide la memoria del programa en varias secciones, a las cuales se les ha llamado por razones históricas **segmentos de memoria**. El ambiente de ejecución de un programa típico en C/C++ contiene un segmento de código, un segmento de datos, una pila para cada hilo de ejecución, un segmento de memoria dinámica (en inglés, *HEAP SEGMENT*), archivos de entrada y salida estándar, entre otros conceptos.

botNeumann++ representa el ambiente de ejecución de C/C++ como una futurista fábrica automatizada (Figura 4.9). La fábrica está dividida en aposentos que representan los segmentos de memoria del programa, etiquetados en la Figura 4.9. Los hilos de ejecución son representados por robots que realizan el trabajo de la fábrica, quienes tienen acceso a los aposentos {segmentos de memoria}. Los cuatro segmentos típicos se explican en los siguientes apartados.

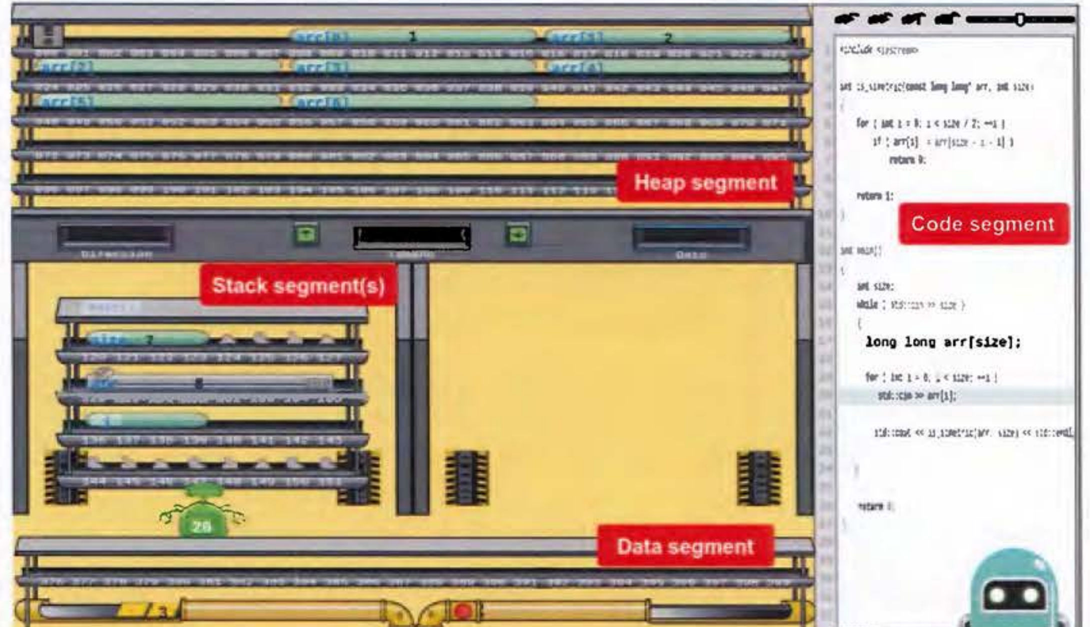



Figura 4.9. Representación del ambiente de ejecución de C/C++ en botNeumann++

#### 4.4.1.2 *Segmento de código*

El **segmento de código** (en inglés, *CODE SEGMENT*) o **segmento de texto** (en inglés, *TEXT SEGMENT*) contiene las instrucciones ejecutables en lenguaje máquina que fueron traducidas del código fuente de C/C++. Usualmente el segmento de código es de sólo lectura. Cuando un programa en C/C++ inicia su ejecución, el ambiente de ejecución hace algunas preparaciones como abrir los archivos de entrada, salida y error estándar; crear el segmento de memoria

dinámica, si lo hay; e iniciar la ejecución del programa desde su punto de entrada (en inglés, *ENTRY POINT*); entre otras tareas. En C el punto de entrada es la función con nombre `main`, pero en C++ puede variar.

Para apoyar el requerimiento de software 2 (estado activo), botNeumann++ es una visualización interactiva que implementa el segmento de código como un editor de texto (Figura 4.10). De esta forma los estudiantes pueden introducir, modificar y experimentar con código C/C++. En la barra de herramientas del editor se proveen botones para realizar operaciones básicas de edición (como cortar, copiar, deshacer), botones para controlar la ejecución del código (como ejecutar/pausar/reanudar, pasar a la siguiente instrucción, y detener), y la velocidad de la animación con la barra deslizable (en inglés, *SLIDER*) a la derecha de los botones como se ve en la Figura 4.10.

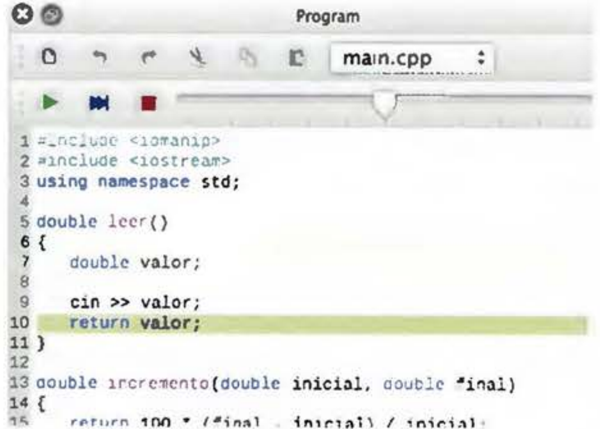



Figura 4.10. Segmento de código en el prototipo C++ de botNeumann++

Cuando el programa es ejecutado y animado, el editor de texto podría tornarse de sólo lectura para mimetizar el comportamiento del segmento de código real. Sin embargo, botNeumann++ permite la edición de código durante la animación en pro de la interactividad, con el fin de permitir a los estudiantes corregir errores apenas los detecten. Mientras la animación están en progreso, la línea de código siendo ejecutada por cada robot {hilo de ejecución} es resaltada usando el color respectivo del robot, como es el caso de la línea 10 en la Figura 4.10.

#### *4.4.1.3 Segmento de datos*

El segmento de datos aloja las variables globales y estáticas de C/C++. Estas variables son cargadas directamente del código objeto, y asignadas a cero si el programador no las inicializa. El tamaño del segmento de datos es conocido en tiempo de compilación, y no crece

o se contrae mientras el programa está en ejecución. Las variables en el segmento de datos se crean antes de que se ejecute el punto de entrada del programa (normalmente la función `main`), pueden cambiar sus valores durante la ejecución del programa, y son destruidas después de que el programa ha terminado su ejecución.

botNeumann++ representa el segmento de datos como una estantería para colocar tubos neumáticos {variables}. La Figura 4.11 muestra un segmento de datos con dos variables globales: una flotante (`double`) y un puntero. Los estantes del segmento de datos se mantienen limpios porque sus variables siempre son inicializadas. No sólo el segmento de datos es representado con estantes, también el segmento de pila y de memoria dinámica. Los lugares donde las variables pueden colocarse en cualquiera de los estantes están numerados {direcciones de memoria}. El segmento de datos está colocado en la parte inferior de la fábrica, e incluye dos tubos neumáticos debajo de los estantes. Por el tubo de la izquierda, la fábrica recibe cápsulas con datos que provienen del mundo externo, como otras fábricas {entrada estándar}. Por el tubo neumático de la derecha, la fábrica puede enviar cápsulas con datos a otras fábricas {salida estándar}. Ambos tubos se incluyen en el segmento de datos porque en C/C++ la entrada y salida estándar están provistas al programador como variables globales.

Figura 4.11. El segmento de datos representado como una estantería en botNeumann++

### *4.4.1.4 Entrada estándar*

La **entrada estándar** es un mecanismo de comunicación que permite al programa recibir datos del ambiente. Se implementa como un archivo en C (`stdin`) o un objeto de flujo en C++ (`std::cin` ó `std::wcin`). Estos símbolos se declaran como variables globales al incluir los encabezados para entrada y salida de las bibliotecas estándar. El ambiente de ejecución los conecta automáticamente al dispositivo de entrada por defecto de la terminal (usualmente el teclado), pero podría ser redirigido a alguna otra fuente de datos. Por defecto la entrada estándar es un archivo o flujo de texto con caché para líneas, aunque puede permitir datos

binarios. Un marcador de fin de archivo (*EOF*, del inglés *END-OF-FILE*) es provisto por el sistema operativo para indicar cuando no hay más datos en el archivo o flujo de entrada.

botNeumann++ representa la entrada estándar como un tubo neumático (Figura 4.12), ubicado en el segmento de datos por ser una variable global. Los datos que llegan por la entrada estándar vienen en cápsulas neumáticas. Cada cápsula contiene un carácter de 1 byte, aún para una entrada binaria. Las cápsulas se animan una tras otra viniendo de afuera (el ambiente) por el extremo derecho de la Figura 4.12 hasta alcanzar el extremo abierto a la izquierda, donde se consideran listas para ser leídas. Cuando el programa corre, el tubo no está completamente lleno de cápsulas, sino que se llena una línea a la vez para reflejar su caché de líneas (en inglés, *BUFFERING*). El marcador de fin de archivo es visualizado como un carácter especial con el valor `EOF`, siempre al final del flujo de datos. Como se explicará más adelante, botNeumann++ es también un juez automático, y sus datos en la entrada estándar provienen de casos de prueba (redirección), aunque los usuarios pueden ingresarlos al accionar el tubo.

Figura 4.12. La entrada estándar representada como un tubo neumático

#### 4.4.1.5 *Salida estándar y error estándar*

La **salida estándar** y el **error estándar** son mecanismos de comunicación que permiten al programa enviar datos al ambiente. La salida estándar está prevista para comunicar resultados o eventos del flujo de ejecución normal del programa, mientras que el error estándar está planeado para reportar condiciones anómalas o excepcionales. La salida estándar y el error estándar se implementan como archivos en C (`stdout` y `stderr`) u objetos de flujo en C++ (`std::cout`/`std::wcout` y `std::cerr`/`std::wcerr`). Estos símbolos son declarados como variables globales en los encabezados correspondientes de la biblioteca estándar. El ambiente de ejecución los conecta al dispositivo de salida por defecto en la terminal, usualmente la pantalla, pero pueden redirigirse a otro destino. La salida estándar tiene un caché de líneas (*BUFFERED*) mientras que el error estándar no, por tanto los datos en el error estándar son enviados al dispositivo de inmediato, mientras que en la salida estándar puede haber una espera hasta que el caché se llene, se envíe un cambio de línea, o el programador solicite enviarlos explícitamente (*FLUSH*).

botNeumann++ representa la salida estándar y el error estándar como tubos neumáticos (Figura 4.13), ambos ubicados en el segmento de datos por ser variables globales en C/C++. Los datos enviados por esos tubos son convertidos de sus representaciones binarias a cápsulas neumáticas que simbolizan caracteres de 1 byte por el robot {hilo de ejecución} que ejecuta la instrucción de impresión. Luego son animados viajando desde el robot hacia el tubo correspondiente. En el caso de la salida estándar los caracteres entran por el extremo abierto y esperan ahí hasta que una de las condiciones de envío ocurran: se llene el caché, llegue un carácter de cambio de línea, o se solicite el envío (*FLUSH*). El error estándar no tiene una sección de espera dado que no tiene un caché para líneas (*BUFFER*), sino que los caracteres son despachados apenas llegan al tubo.

Como se explicará en el apartado 4.4.2.4, botNeumann++ es también un juez automático, y si se tienen casos de prueba, los datos producidos por el programa del estudiante en la salida y error estándar, son comparados contra las salidas y errores esperadas en el caso de prueba. Si estas salidas coinciden, un dispositivo lumínico se torna verde, de lo contrario se mantiene en rojo.

Figura 4.13. El error estándar (izquierda) y la salida estándar (derecha) como tubos neumáticos

#### 4.4.1.6 *Valores y variables*

Un **valor** es un elemento del dominio de un tipo de datos. Una **variable** es un espacio en memoria que puede almacenar un valor y es identificado en el código fuente por un nombre. Dado que C/C++ sigue el paradigma imperativo, las variables pueden cambiar de valor durante la ejecución del programa. El tipo de datos debe proveerse al declarar una variable en C/C++ para establecer su tamaño (en bytes) y restricciones sobre el dominio de valores que puede almacenar. Los tipos de datos de C/C++ pueden clasificarse en tres categorías: escalares, in-dirección, y compuestos. Los tipos de datos escalares almacenan valores "atómicos" numéricos, tales como: `bool`, `char`, `int`, y `float`. Los tipos de datos de in-dirección hacen referencia indirecta a través de direcciones de memoria a otros valores en la memoria principal, tales como punteros y referencias. Los tipos de datos compuestos almacenan múltiples valores del mismo tipo de datos (arreglos) o de diversos tipos de datos (registros: estructuras, uniones, o clases) en regiones continuas de memoria. Los tipos de datos escalares

y de in-dirección son considerados tipos de datos primitivos, mientras que los tipos de datos de in-dirección y los compuestos son considerados tipos de datos recursivos.

botNeumann++ usa cápsulas neumáticas como metáfora visual para los valores (Cuadro 4.4). Las variables son representadas como valores con su identificador anotado en una adhesiva, por ejemplo, el entero year en la línea 3 del Cuadro 4.4. Si no hay espacio en la capsula para el identificador, botNeumann++ podría ocultarlo, pero se puede obtener por métodos alternativos como el menú contextual o haciendo clic en su dirección de memoria. Las variables se crean en los estantes de algún segmento de memoria y permanecen inmóviles. De forma diferente, los valores pueden desplazarse por los tubos neumáticos {entrada y salida estándar} o ser desplazados por acción de los robots {hilos de ejecución} de un lugar a otro dentro de la fábrica. Por ejemplo, en una lectura de un número real, el robot toma cápsulas que contienen caracteres del tubo de entrada, las transporta a su área de trabajo, las convierte en una cápsula rectangular con el valor real, y finalmente transporta el valor real resultante dentro de la cápsula de la variable, lo que reemplaza su viejo valor.

Las cápsulas para tipos de datos integrales tienen los extremos redondeados, mientras que las cápsulas para los tipos de punto flotante tienen esquinas cuadradas, para indicar que si los flotantes son asignados a enteros, habrá una pérdida de precisión, de acuerdo a la sugerencia de [Waguespack 1989]. La longitud de una cápsula es proporcional al número de bytes que ocupa su tipo de datos, y botNeumann++ hace esta proporción consistentemente visible en las variables, porque son almacenadas sobre las direcciones de memoria de algún estante.

Los punteros son variables enteras sin signo, cuyo valor es la dirección de algún dato en la memoria. Los punteros en C/C++ tienen dos datos: la dirección almacenada como valor y el tipo de datos usado para interpretar la memoria referida cuando el puntero es "des-referenciado". botNeumann++ representa los punteros como variables enteras (Cuadro 4.4), agregándoles una antena parabólica. La antena puede hacer referencia a otro lugar de la fábrica. Cuando el usuario interacciona con un puntero, como hacer clic sobre él, botNeumann++ anima ondas electromagnéticas que viajan desde la antena hasta la memoria apuntada. La antena se visualiza inactiva y gris si el puntero es nulo (su valor entero es 0), verde si apunta a un lugar válido y accesible para el programa, o roja en otro caso, como un puntero no inicializado.

Dado que C/C++ almacena los arreglos como elementos continuos en una región de memoria, botNeumann++ dibuja los arreglos como una sucesión de elementos (por ejemplo el arreglo ep en la línea 8 del Cuadro 4.4). botNeumann++ envuelve los múltiples valores de un tipo de datos compuesto usando un tubo neumático semitransparente (por ejemplo la línea 9 en el Cuadro 4.4). Si el usuario crea un arreglo de estructuras, botNeumann++ dibuja un pequeño tubo conector entre cada par de elementos para hacer visible dónde una estructura termina y la siguiente inicia (por ejemplo la línea 10 en el Cuadro 4.4).

Cuadro 4.4. Valores y variables representados como cápsulas neumáticas

| Categoría | # | Ejemplo de declaración | Ejemplo de metáfora visual |
|---|---|---|---|
| Escalar | 1 | `bool` | |
| | 2 | `char` | a |
| | 3 | `int` | year 2017 |
| | 4 | `double` | 6.67e-11 |
| In-dirección | 5 | puntero nulo | values 0 |
| | 6 | puntero válido | score 136 |
| | 7 | puntero inválido | enemy 78961040 |
| Compuesto | 8 | arreglo de primitivos | ep 11 121 1331 |
| | 9 | estructura o clase | A N A score 385 |
| | 10 | arreglo de estructuras | A N A score 385 D D D score 713 |

#### 4.4.1.7 Hilos de ejecución

Un **hilo de ejecución** (en inglés, *EXECUTION THREAD*) es un conjunto de valores (registros del procesador) que una unidad central de procesamiento (*CPU*, del inglés *CENTRAL PROCESSING UNIT*) requiere como contexto para ejecutar una serie de instrucciones. El conjunto de valores incluye al menos, un identificador (del hilo de ejecución), un contador de programa (en inglés, *PROGRAM COUNTER*, también llamado *INSTRUCTION POINTER*) que indica la instrucción de código siendo ejecutada, y un puntero a su pila exclusiva para invocar funciones. Los hilos de ejecución que pertenecen al mismo proceso comparten los demás recursos asignados por el ambiente de ejecución al proceso, como el segmento de código, el segmento de datos, y los archivos abiertos. Un hilo de ejecución requiere un núcleo de *CPU* disponible para correr. Si no hay núcleos de CPU disponibles en un determinado momento, el hilo de ejecución tendrá que esperar hasta que alguno se libere. Cuando un programa es ejecutado (proceso), siempre

inicia un hilo de ejecución principal, y finaliza cuando este hilo termina su ejecución. Los hilos de ejecución pueden crear más hilos, y hasta cierto punto, controlarlos. Los hilos de ejecución están disponibles a través de bibliotecas, como *PTHREADS* en C y `std::thread` en la biblioteca estándar de C++.

botNeumann++ representa los hilos de ejecución como robots que realizan las operaciones de la fábrica (Figura 4.14). Los robots ejecutan las instrucciones en C/C++ provistas por el usuario al pie de la letra. Cada robot tiene un pequeño monitor donde muestra el número de línea {contador de programa} del código C/C++ que está ejecutando. Cada robot tiene una pila exclusiva de invocaciones a funciones. La pila está representada por estanterías, una por cada invocación de función, colocadas en frente del robot. Si un hilo de ejecución está activo, estará trabajando en una de las estaciones de trabajo de la fábrica {núcleo de *CPU*}. Si un hilo de ejecución pasa a estado inactivo, botNeumann++ lo anima moviéndose a la línea de espera y su segmento de pila será almacenado temporalmente bajo el piso de la fábrica (con el fin de ahorrar espacio en la pantalla). Los robots {hilos de ejecución} trabajan exclusivamente en sus estaciones de trabajo, ya que no les está permitido invadir el espacio de otros robots. Un robot podría comunicarse con otro sólo si tiene un puntero o estructura que se lo permita. Aunque los robots {hilos de ejecución} no comparten sus estaciones de trabajo {segmentos de pila}, comparten el resto de la fábrica {proceso}, como el protocolo de producción {segmento de código}, la estantería detrás de ellos {segmento de datos} con sus tubos neumáticos {entrada y salida estándar}, y la bodega {segmento de memoria dinámica}. La distribución espacial de los aposentos de la fábrica muestra coherentemente que los robots pueden acceder a estos aposentos compartidos.

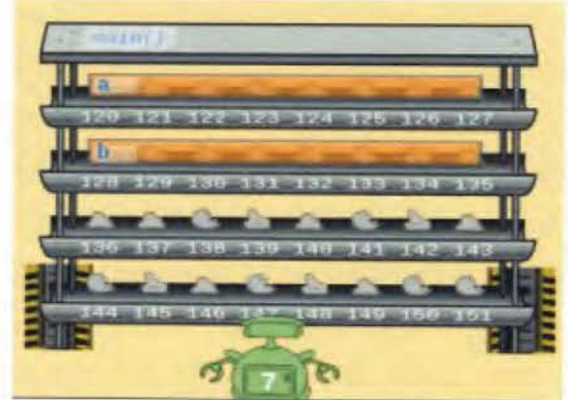



Figura 4.14. Un hilo de ejecución activo en botNeumann++

#### *4.4.1.8 Segmento de pila*

El segmento de pila es la sección de la memoria del programa dedicada a almacenar la información requerida por las subrutinas activas (invocaciones de funciones). En programas multi-hilados cada hilo de ejecución tiene su propio segmento de pila. En este documento, el término **segmento de pila** (en inglés, *STACK SEGMENT*) se usa para referir a la región de memoria asignada a un hilo de ejecución para almacenar los datos de sus invocaciones de funciones, mientras que el término **pila de invocaciones** (en inglés, *CALL STACK*) se refiere a la estructura de datos que crece y contrae dentro de este segmento. Cada vez que una función es invocada, algunos datos (como la dirección de retorno, los argumentos, y las variables locales) son apiladas. Un hilo sólo ejecuta la invocación en la parte superior de la pila, mientras las demás están en espera. Los sistemas operativos limitan el tamaño de los segmentos de pila. Si una pila de invocaciones agota su segmento de pila (por ejemplo, debido a una recursión infinita o a grandes cantidades de variables locales), un error de desbordamiento de pila (en inglés, *STACK OVERFLOW*) es emitido y típicamente el proceso completo es terminado por el sistema operativo.

Cuando la función en la parte superior de la pila termina su ejecución, sus datos son retirados de la pila, y el control retorna a la invocación de la función previa (conocida como la función invocadora o llamadora). Si la última función en una pila termina su invocación, el hilo de ejecución termina su ejecución, y sus recursos se liberan. El mecanismo de invocar y retornar de funciones es automáticamente implementado por C/C++, por tanto, es oculto para el programador. Sin embargo, los programadores deben entender este mecanismo para poder implementar subrutinas efectivamente.

botNeumann++ representa una pila de invocaciones con varias estanterías (Figura 4.15), las cuales son usadas directa y exclusivamente por un robot {hilo de ejecución}. La estantería más hacia el frente es la función siendo ejecutada por el robot. El robot no puede acceder a las estanterías {invocaciones de función} que están detrás, ni a las que pertenecen a otros robots. Sin embargo, el robot podría tener valores de in-dirección como punteros o referencias hacia datos que estén en esas otras estanterías.

Cuando un robot tiene que hacer una tarea {se invoca una función}, botNeumann++ realiza una compleja animación. Si hay estanterías en la estación de trabajo, el robot las corre hacia atrás sobre un riel {pila de invocaciones} para despejar una escotilla en el piso de la estación

de trabajo. El robot abre la escotilla {registro de pila} en el piso y hace subir una estantería nueva desde la escotilla, con restos rotos de viejas cápsulas que estuvieron en los estantes. El robot sube la estantería conforme necesite estantes para ensamblar las cápsulas {variables locales} que necesita para realizar la tarea {función} y las va rotulando con nombres en adhesivas {identificadores de variables}. El robot cambia el contenido de algunas cápsulas. El robot puede replicar el valor de otras cápsulas presentes en la estantería previa {paso de parámetros}, y en otras cápsulas insertar valores {inicialización de variables} indicados en el protocolo de trabajo {segmento de código}. Las cápsulas que se crearon y no se guardaron valores en ella, mantienen los restos de las viejas cápsulas que estuvieron en el estante {valores indeterminados de variables no inicializadas}. Finalmente, el robot cierra la escotilla y continúa trabajando en la nueva tarea. Sin embargo, el robot puede abrir esta escotilla de nuevo para obtener más estantes {creación de más variables locales} o liberar estantes desocupados {eliminación de variables locales, por ejemplo, al cerrar un bloque}. Si las invocaciones agotan el espacio del segmento de pila, por ejemplo en una recursión infinita, botNeumann++ anima las estanterías descarrilándose {desbordamiento de pila} y detiene la animación.

Cuando el robot termina la tarea actual {retorno de función}, abre la escotilla y la estantería entera retorna al sótano {memoria libre del segmento de pila}. Conforme la estantería va atravesando la escotilla, las cápsulas se rompen, lo que deja escombros en los estantes. Finalmente, las estanterías que estaban detrás, se mueven hacia delante sobre el riel {pila de invocaciones} y el robot continúa trabajando en la tarea pendiente previa {función invocadora}.

Figura 4.15. Una pila de dos invocaciones a función señaladas con flechas

#### *4.4.1.9 Segmento de memoria dinámica*

Los programas de computadora procesan datos. La cantidad de datos en el segmento de código (de sólo lectura) y del segmento de datos, es conocida en tiempo de compilación y no puede variar en tiempo de ejecución. Aunque el tamaño del segmento de pila es restringido por el sistema operativo, los datos en él puede crecer sin sobrepasarlo o decrecer en tiempo de ejecución, pero son gestionados automáticamente y no por el programador. Cuando los programadores requieren alojar cantidades arbitrarias de memoria principal en tiempo de ejecución, se debe emplear memoria dinámica. El **segmento de memoria dinámica** (adaptación del inglés de *HEAP SEGMENT*) es una región virtual de la memoria que contiene las asignaciones dinámicas de memoria hechas por un proceso. Esta región puede crecer o contraerse mientras el programa está en ejecución. El programador puede gestionar memoria dinámica a través de funciones de biblioteca en C (como `malloc()`, `calloc()`, `realloc()` and `free()`) y operadores en C++ (`new`, `delete`, `new[]`, `delete[]`). Tanto las funciones como los operadores trabajan exclusivamente con punteros, por tanto, la memoria dinámica en C/C++ puede ser accedida únicamente de forma indirecta. C/C++ ha sido criticado porque los programadores son los responsables de liberar las asignaciones hechas en memoria dinámica, y esta práctica es especialmente difícil para aprendices. Dos errores comunes en este lenguaje son:

[1] Fuga de memoria (en inglés, *MEMORY LEAK*). Ocurre cuando se olvida liberar datos en memoria dinámica antes de que todos sus punteros sean destruidos.

[2] Puntero colgado (en inglés, *DANGLING POINTER*). Ocurre al tratar de acceder a memoria dinámica después de que ésta ha sido liberada.

botNeumann++ representa el segmento de memoria dinámica como una bodega, el más extenso aposento de la fábrica, porque su tamaño es el menos restringido por el sistema operativo (Figura 4.16). Los robots {hilos de ejecución} pueden almacenar cantidades mayores de cápsulas en esta bodega que en sus estaciones de trabajo, pero no pueden ingresar a ella directamente, sino que deben usar las ventanillas {administrador de memoria dinámica} que median entre la bodega y las estaciones de trabajo. Todos los estantes de la bodega están rotulados con números {direcciones de memoria} y estos números deben indicarse en cualquier intercambio de cápsulas a través de las ventanillas, junto con la acción que se quiere realizar: reservar estantes, recuperar una cápsula de un estante, alojar una

cápsula en un estante, o liberar estantes. Con el fin de hacer las fugas de memoria visibles, botNeumann++ tiñe de rojo las cápsulas y sus direcciones si se pierde el último puntero hacia ellas. Si un puntero apunta a memoria que fue liberada, botNeumann++ pinta su antena roja (como ocurre con enemy en el Cuadro 4.4, p.133).

Figura 4.16. Representación del segmento de memoria dinámica

#### *4.4.1.10 Expresiones*

Las expresiones son combinaciones de valores o variables con operadores que son evaluadas recursivamente y generan un único valor. El orden de evaluación está gobernado por las reglas de "aridad" (número de argumentos), precedencia, y asociación (en inglés, *ASSOCIATIVITY*), impuestas a los operadores por el lenguaje de programación. Los operadores son funciones que reciben argumentos y producen valores temporales. Los valores temporales son gestionados automáticamente, aunque C++11 y versiones posteriores, permiten realizar algunas manipulaciones con valores temporales. Sin embargo, el autor de esta tesis ha notado anecdóticamente que los estudiantes tienen grandes dificultades para comprender este tema.

botNeumann++ muestra la evaluación de expresiones como el trabajo inmediato hecho por los robots {hilos de ejecución}. Cuando un operador es evaluado, el robot consigue cápsulas (valores o copias) de los "operandos", y los mantiene en sus manos (Figura 4.17). El valor temporal producido por el operador queda en las manos del robot. La visualización de la evaluación de expresiones puede deshabilitarse en el botón de configuración del segmento de código, en caso de que se deseen menos detalles o acelerar la animación.

Figura 4.17. Valor temporal después de una evaluación de expresión

#### *4.4.1.11 Otros conceptos*

C/C++ es uno de los lenguajes de programación más complejos, que abarca varios paradigmas y considerable cantidad de conceptos de programación. Crear una visualización de programa genérica que abarque la máquina nocional completa de C/C++, excede los límites de esta investigación. Por tanto, conceptos como archivos/flujos, excepciones, herencia, polimorfismo, y plantillas, se delegan para trabajo futuro.

#### *4.4.1.12 Discurso alegórico*

A diferencia de las metáforas, las alegorías permiten crear discursos en el dominio origen que son coherentes con el discurso elidido destino. De la misma forma que Lakoff provee expresiones metafóricas para ejemplificar una metáfora, a continuación se provee un discurso alegórico al combinar las metáforas discutidas en las secciones previas. Dada la naturaleza visual de la alegoría, los números encerrados en círculos dentro del texto corresponden a las metáforas visuales etiquetadas en la Figura 4.18.

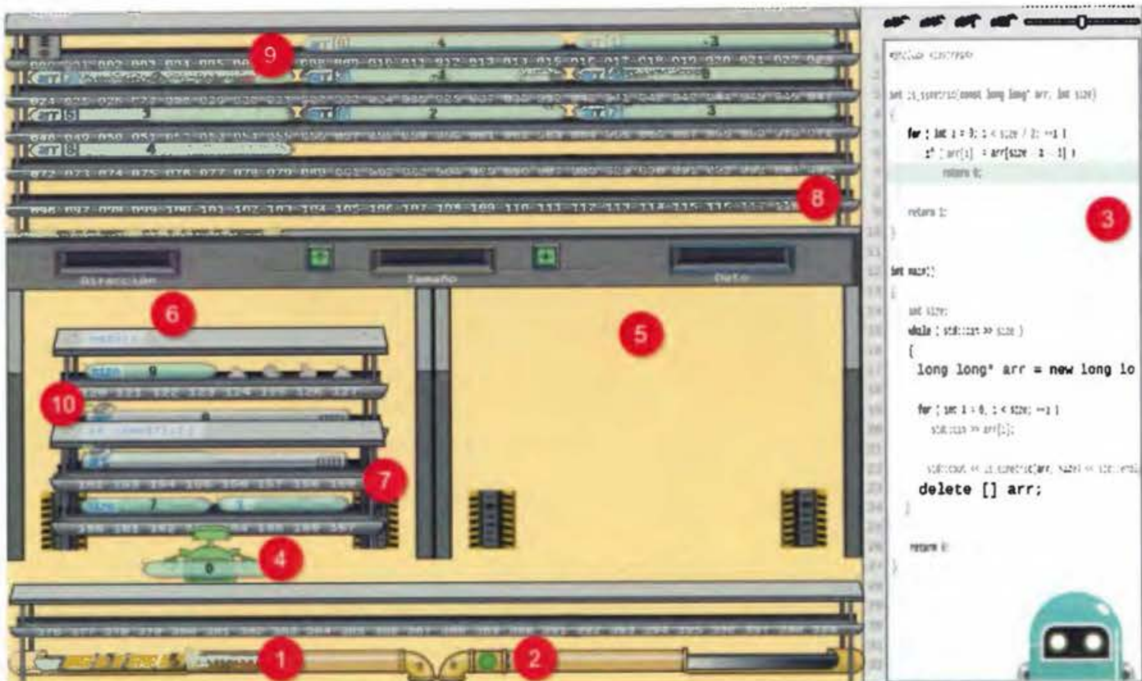



Figura 4.18. Metáforas visuales en botNeumann++ enumeradas para el discurso alegórico

Los programas en C/C++ son fábricas {ambientes de ejecución} (Figura 4.18) donde los trabajadores convierten un producto, como materias primas {datos} en productos intermedios o acabados {información}. Las fábricas en nuestro complejo están intercomunicadas por eficientes tubos neumáticos que transportan las materias primas o productos (① y ② en la Figura 4.18). Cada fábrica recibe la materia prima en cápsulas {valores} por un tubo neumático {entrada estándar} ①, la procesa, y sus productos se envían a otras fábricas {procesos} o clientes {usuarios} por otro tubo neumático {salida estándar} ②. Las fábricas son muy versátiles y los trabajadores en ellas pueden convertir materias primas en productos de formas inimaginables. Para entregar un producto a un cliente {problema}, varias fábricas pueden intervenir en el proceso de producción. A cada fábrica, entonces, se le asigna un protocolo del proceso de producción {código fuente del programa} ③ para que los trabajadores sepan las funciones que deben realizar. El protocolo del proceso de producción puede cambiar, por ejemplo, cuando se cambia el producto, el cliente, o se identifican mejoras en el proceso de producción. Por tanto, el protocolo siempre está pegado en una zona visible de la fábrica {segmento de código} ③ para que todos los trabajadores en la fábrica puedan verlo. Los ingenieros {estudiantes} crean y dan mantenimiento a los protocolos de producción de las fábricas, y sus trabajadores no están

autorizados a modificarlos {sólo lectura}. Nuestras fábricas están completamente automatizadas, por lo que nuestros trabajadores son robots {hilos de ejecución} ④. Los robots siguen los protocolos al pie de la letra. Los robots son muy versátiles pero no entienden lenguaje natural, sino que están diseñados por el fabricante para seguir protocolos escritos en un extraño lenguaje llamado C/C++.

Las fábricas de la empresa podrían variar en sus capacidades. Algunas fueron construidas con aposentos {segmentos} más grandes que otras, y que permiten tener más cápsulas con materias primas o productos. Algunas pueden tener más espacios para trabajadores (estaciones de trabajo) {núcleos de procesador} ⑤ que otras. Pero en general, todas las fábricas tienen la misma distribución espacial para que los robots puedan trabajar en ellas, como se ve en la fábrica de ejemplo de la Figura 4.18.

Dado que los robots tienen que manufacturar los productos, ellos trabajan en un espacio {núcleo de procesador} ⑤ donde pueden colocar las materias primas y transformarlas {expresiones} ④ paulatinamente en los productos. La cantidad de trabajadores en una fábrica podría variar de acuerdo a su carga de trabajo. Nuestra compañía tiene muchos robots, la mayoría de ellos pasan ociosos y se mantienen apagados para ahorrar energía. Cuando se sabe que la carga de trabajo de una fábrica puede crecer en el tiempo, los ingenieros pueden, en el protocolo de producción, instruir a los robots a pedir ayuda de otros. Cuando un robot A pide ayuda con una tarea {una función}, la fábrica {ambiente de ejecución} enciende a otro robot B {creación de hilo de ejecución} y le entrega la tarea que debe realizar. El robot B necesita una estación de trabajo {núcleo de procesador} ⑤ para hacer la tarea solicitada y se mueve a la línea de espera {cola de espera}. Si existe alguna estación de trabajo disponible, el robot se moverá a ella y hará la tarea. Si todas las estaciones están ocupadas, el robot B deberá esperar a que una se desocupe. Mientras esté en la línea de espera, duerme para ahorrar energía. Generalmente los robots no esperan por mucho tiempo; ellos cortésmente toman turnos por las estaciones de trabajo. La fábrica le da al robot A un comunicador portátil {un puntero u objeto}, que A coloca en su estación de trabajo y que puede usar para comunicarse con el robot B. Por ejemplo, para saber cuándo B ha terminado su tarea o si A tuviera que esperar por B. Cuando B finalmente ha terminado la tarea, se auto-apaga {terminación normal de hilo de ejecución} y el robot A puede enterarse a través del comunicador. Cuando el último robot en una fábrica termina su tarea, el

protocolo está completo y no hay trabajo pendiente, por tanto, apaga la fábrica para ahorrar energía {terminación normal de proceso}.

El flujo de trabajo habitual de una fábrica es como sigue. Un robot enciende la fábrica {hilo principal de ejecución} y sigue el protocolo de producción que asignaron los ingenieros. Usualmente este protocolo solicita al robot recibir las materias primas {valores} que vienen en cápsulas por el tubo neumático de entrada. Si el protocolo indica que debe trabajar solo, el robot hará las transformaciones de las materias primas en productos, de lo contrario podría ceder, total o parcialmente esta transformación a otros robots y recibir de ellos los productos. Finalmente este robot envía los productos por el tubo neumático de salida. Los tubos de entrada y salida de cápsulas están colocados en una estantería compartida por todos los robots {segmento de datos}. Los robots no pueden agregar o quitar cápsulas de esta estantería, pero sí pueden alterar los contenidos de las cápsulas en ella o enviar y recibir cápsulas por los tubos. Los protocolos deben estar bien diseñados por los ingenieros para evitar que dos o más robots realicen operaciones simultáneamente en un mismo recurso de esta estantería compartida {condición de carrera, en inglés *RACE CONDITION*}. Por ejemplo, si dos robots envían cápsulas al mismo tiempo por el tubo de salida, las cápsulas se mezclarán y pueden generar un producto que no es el esperado por el cliente.

Algunos procesos de producción son muy complejos y los robots podrían tener que manipular considerable cantidad de cápsulas {valores}. Con el fin de evitar un desorden, los robots requieren que los protocolos estén estructurados en tareas de manufactura {funciones}. Muchas tareas son repetitivas y los robots saben realizarlas {funciones de biblioteca}, como calcular un logaritmo. Esto ahorra mucho trabajo a los ingenieros, que pueden pedir a los robots hacer estas tareas sin tener que indicar en los protocolos de producción cómo realizarlas {reutilización de código}. Cuando un robot está haciendo una tarea particular, agrupa todas las cápsulas que necesita para esa tarea en una estantería {invocación de función} ⑥, y etiqueta la estantería con el nombre de la tarea {puntero a función}. Si mientras un robot está haciendo una tarea y el protocolo solicita iniciar otra, el robot anotará el número de línea del protocolo donde quedó sobre la estantería {contador de programa, en inglés *PROGRAM COUNTER*}, y empujará las estanterías de las tareas incompletas hacia atrás, las cuales se deslizan sobre un riel {pila de invocaciones} ⑦. Eso da espacio para obtener una estantería para la nueva tarea, la cual surgirá del sótano {memoria libre del segmento de pila} a través de una escotilla {registro de pila}. El robot tomará las

cápsulas que necesite de la tarea anterior {paso de parámetros} y ubicará otras que necesite {variables locales}. El proceso de iniciar una nueva tarea se detalla en el apartado 4.4.1.8 (Segmento de pila). Dado que los recursos de la fábrica son finitos, a cada robot se le da un número limitado de estantes para trabajar. Estos estantes se almacenan en el sótano y el robot los puede usar para armar estanterías para las tareas que debe realizar. Si un robot agota sus estantes asignados {desbordamiento de pila}, se considera un mal diseño del protocolo de producción, y la fábrica cesa de trabajar hasta que los ingenieros corrijan el protocolo o lo asignen a una fábrica con más recursos. Cuando un robot termina su tarea actual, regresa la estantería al sótano. Los estantes son desensamblados para luego poder armar nuevas estanterías. En el proceso de desensamblado, las cápsulas que aún quedan en los estantes se rompen, lo que deja escombros en ellos.

Si los robots requieren más espacio del que les permite sus estaciones de trabajo, las fábricas disponen de una amplia bodega con estantes {segmento de memoria dinámica} ⑧. La bodega es compartida por todos los robots en la fábrica. Para aprovechar el espacio, la bodega no tiene pasillos entre los estantes, por tanto los robots no pueden ingresar en esta área. Sin embargo, esta área se controla por un mecanismo automático {administrador de memoria dinámica}. Si un robot necesita almacenar cápsulas en la bodega, solicita reservar algunos estantes. Si el mecanismo de la bodega encuentra estantes libres, los reserva y le retornará al robot una cápsula especial que contiene un número que identifica el estante reservado {dirección de memoria} ⑨. Más aún, todos los estantes de la fábrica están enumerados de forma única. La cápsula retornada es especial porque incorpora un conveniente dispositivo transceptor de radio {puntero} ⑩ que permite al robot acceder directamente a las cápsulas en la bodega. Los robots pueden almacenar cápsulas en la bodega por el tiempo que lo necesiten. Sin embargo, los ingenieros deben diseñar los protocolos de producción para liberar los estantes tan pronto como no los necesiten, con el fin de que puedan ser aprovechados por otros o el mismo robot luego en el tiempo. Los robots deben ser muy cuidadosos con los transceptores de radio entregados por la bodega. Si ellos destruyen un transceptor antes de liberar los estantes reservados, el mecanismo de la bodega no tiene forma de saber si están siendo usados o no {fuga de memoria}, por tanto, los mantiene intactos consumiendo espacio hasta que la fábrica se apague. Si un robot solicita reservar estantes, pero la bodega no tiene suficientes libres, el mecanismo responderá con una transceptor hacia un número de estante no válido (cero) {puntero nulo}

> que los robots no deben tratar de usar. Todo transceptor tiene un botón para liberar los estantes reservados, que cuando se presiona rompe las capsulas a distancia en la bodega {liberación de memoria}. Si el robot trata de usar las cápsulas luego de que sus estantes fueron liberados, habrá un error de diseño de producción y pueda que la fábrica se detenga hasta que los ingenieros reparen el protocolo.

De acuerdo a la propuesta conceptual, una vez diseñada la alegoría visual, se incorporan a la visualización de programa los elementos lúdicos para satisfacer los requerimientos de software, lo cual se hace en la siguiente subsección para botNeumann++.

## 4.4.2 Ludificación de botNeumann++

En esta subsección, se proveen ejemplos de cómo los elementos lúdicos pueden usarse en botNeumann++ para satisfacer los requerimientos de software. Con estos ejemplos (y los del diseño de *PUPPETEER++*) se establecieron las asociaciones en la Figura 4.2 (p.108) de la propuesta conceptual al inicio de este capítulo. Dado que la narración y los niveles pueden ayudar a satisfacer la mayoría de requerimientos, se incluye un apartado para cada uno de ellos, luego un apartado para los restantes elementos del juego, y finalmente se discuten algunas decisiones de diseño influenciadas por los jueces automáticos de la programación competitiva.

### *4.4.2.1 Narración y objetivos*

El discurso alegórico permite la creación de una narración (elemento lúdico 9) y los objetivos del juego (elemento lúdico 2) basados en las interrelaciones de las entidades origen. Las personas recuerdan las historias más efectivamente que listas inconexas de conceptos [Kapp 2012]. Por tanto, es más probable que los estudiantes recuerden los conceptos de programación si están interrelacionados en una narración que en una presentación tradicional. La siguiente podría ser una narración potencial para botNeumann++, a presentar con el tutorial del juego (Figura 4.19).

> Como ya debes saber por las noticias, un grupo de hackers esperaron estratégicamente esta fecha, el 22 de febrero de 2222, para explotar vulnerabilidades de las fábricas

automatizadas de manufactura del mundo. Los procesos de producción de estas fábricas robotizadas fueron construidos desde sus orígenes en formas primitivas de C/C++.

El ataque aprovechó las vulnerabilidades en los protocolos para poner las fábricas en estado de espera por correcciones, lo que permitió a los hackers inyectar protocolos que sobrecalentaron los robots y dañaron algunas fábricas. El equipo de emergencia ha logrado recuperar fragmentos de los protocolos de producción de la mayoría de fábricas, pero actualmente la mayoría están colapsadas y la humanidad urge de la restitución de los servicios.

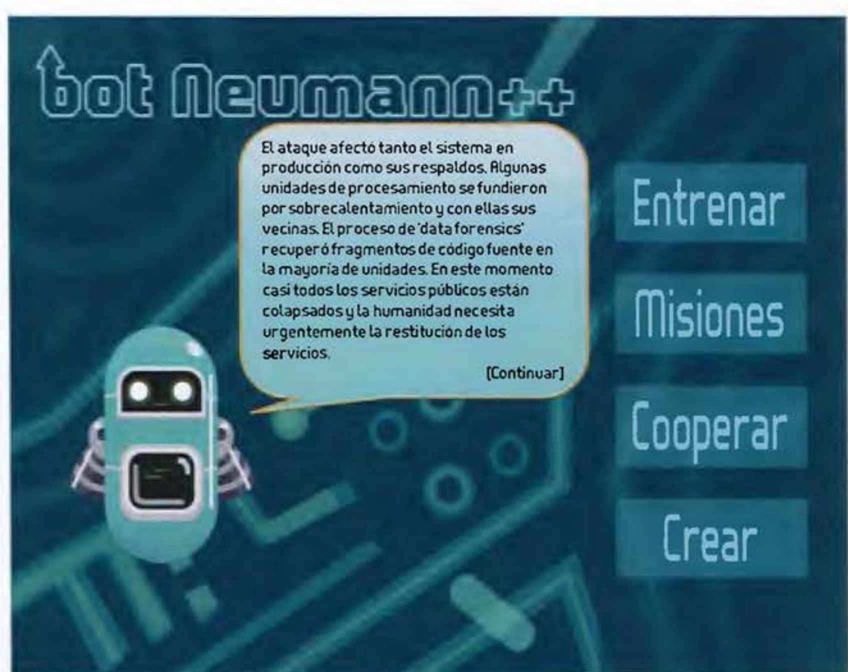



Figura 4.19. Menú del juego de botNeumann++ y un trozo de narración

Mientras la narración está en progreso, un diálogo puede establecerse con el estudiante. Este diálogo puede ayudar a conocer la motivación inicial del estudiante para aprender programación de computadoras (requerimiento 1 en la Figura 4.2 p.108) al inicio del juego. La narración puede acompañar su progreso por los niveles, evaluar nociones previas y emociones sobre cada concepto de programación (requerimiento 5). La narración puede estructurarse en episodios de acuerdo a una organización natural de conceptos (requerimiento 14). La narración puede incentivar el interés de los estudiantes en la tarea de aprendizaje (requerimiento 3), ya que es probable que quieran averiguar qué ocurre en el

siguiente episodio (nivel). El siguiente es un ejemplo de cómo botNeumann++ puede dialogar con el estudiante y plantear el objetivo del juego.

> Hola. Mi nombre es John ¿Cuál es el tuyo? María
>
> ¿Sabes programar en C/C++? ( x ) sí (   ) no
>
> ¡Oh genial! Es que necesitamos la ayuda de todas las personas que conozcan C/C++ para restaurar las fábricas de manufactura del mundo. Los nuevos procesos de producción no deben tener vulnerabilidades, con el fin de evitar nuevos ataques. Yo te enseñaré cómo estas fábricas trabajan. Presiona el botón "Entrenar" para iniciar.

Cada nuevo concepto de programación puede introducirse como parte de la narración, la cual puede involucrar a los conceptos previos más relacionados, con el fin de ayudar a los estudiantes a asociar conocimiento nuevo con existente (requerimiento 7). Por ejemplo, al introducir el concepto de memoria dinámica, puede hacerse en "una fábrica que usaba la bodega pero no chequeaba la cantidad de cápsulas {arreglos} que guardaba en ella, por lo que un atacante aprovechó para inyectar una cápsula con un número grande en el tubo de entrada y poner la fábrica en estado de recuperación". La narración puede ayudar a conectar conceptos con otros previos, lo que ayuda a construir sistemas de conceptos (requerimiento 13).

La narración puede comunicarse de forma uni-modal (con metáforas visuales únicamente), o multimodal (por ejemplo, imágenes más textos o sonidos). Si el texto es incluido, una pregunta a responder es si debería la narración usar los nombres de las entidades origen (la fábrica), de las entidades destino (conceptos de programación), o una combinación de ambos. Por ejemplo, botNeumann++ usa robots para representar hilos de ejecución. ¿Debería la narración de botNeumann++ usar sólo el término "robot", o sólo el término "hilo de ejecución", o debería alternarlos? Para responder esta pregunta se requiere investigación futura.

#### *4.4.2.2 Niveles*

Las visualizaciones lúdicas de programa pueden usar niveles (elemento lúdico 8) para reflejar una organización natural de los conceptos de programación (requerimiento 14 en la Figura 4.2 p.108). La Figura 4.20 diagrama un mapa de niveles hipotético para una visualización lúdica de programa para la máquina nocional de C. Cada nivel (representado como

rectángulos redondeados en la Figura 4.20) introduce un nuevo concepto de programación. Varios ejercicios teóricos, como comparar un nuevo concepto con conceptos relacionados (requerimiento 9), o ejercicios que aplican el nuevo concepto a varios contextos distintos (requerimiento 10), podrían ser representado como sub-niveles (círculos numerados en la Figura 4.20, pero otros recursos como barras de progreso podrían ser más convenientes). Grupos de conceptos de programación relacionados temáticamente, pueden representarse como "mundos" (islas en la Figura 4.20). El terreno cubierto por nubes oscuras en la Figura 4.20 representa los conceptos de programación aún no explorados en el progreso del estudiante a través de la visualización lúdica. La organización de conceptos de la Figura 4.20 está basada en la estructura del libro de C de Kernighan y Ritchie [Kernighan and Ritchie 1988]. Se incluyeron pocos conceptos por razones de espacio.

Figura 4.20. Mapa de niveles hipotético para una visualización lúdica de C

Los niveles en combinación con estética (elemento lúdico 11) proveen realimentación inmediata (requerimiento 4) a los estudiantes sobre su progreso a través del proceso de aprendizaje (requerimiento 11). Por ejemplo, la Figura 4.20 usa colores inspirados en las luces de semáforos para denotar si las soluciones del estudiante (círculos numerados) son válidas (verde), erróneas (rojo), o algo intermedio (amarillo). Los mismos colores se usan en los niveles (rectángulos redondeados) para indicar un estimado del nivel de dominio sobre cada concepto de programación.

Existen varias formas de organizar los conceptos de programación reflejando relaciones naturales entre ellos (requerimiento 14). Por ejemplo, los conceptos de programación en la Figura 4.20 están conectados por caminos que representan relaciones de dependencia.

Cuando el estudiante resuelve un mínimo de ejercicios sobre un concepto de programación, su rectángulo se torna verde (como un semáforo) y los caminos que salen del nivel se habilitan, indicando que el estudiante puede avanzar. Dado que los caminos representan dependencia en la Figura 4.20, todas las dependencias deben satisfacerse para que un nivel se habilite. Por ejemplo, en el mundo de las "Variables" de la Figura 4.20, el estudiante debe resolver suficientes problemas sobre "Inicialización" y "Tipos de datos" para poder activar el nivel sobre "Entrada estándar". Los estudiantes pueden seguir distintos recorridos de aprendizaje. Por ejemplo, después de aprender sobre declaración de variables, un estudiante podría escoger entre identificadores, inicialización, o tipos de datos, de acuerdo a la Figura 4.20.

Los niveles pueden usarse para fomentar el desarrollo de hábitos (requerimiento 12). Un hábito se desarrolla al aplicar un concepto a distintos contextos. Idealmente una visualización lúdica de programa podría tener una base de datos de retos (ejercicios), y dinámicamente retar al estudiante seleccionando retos de acuerdo al rendimiento que haya mostrado en cada concepto de programación. Algunos indicadores podrían usarse para estimar el grado de desarrollo de hábitos del estudiante, como el tiempo transcurrido resolviendo un ejercicio (elemento lúdico 5), eficiencia de las soluciones, o conteo de reintentos (elemento lúdico 12).

El sistema de niveles puede ayudar a los estudiantes a desarrollar sistemas de conceptos (requerimiento 16). Por ejemplo, al finalizar cada "mundo" en la Figura 4.20, se provee un nivel (en forma de castillo) con retos que requieren la aplicación de los conceptos aprendidos en los niveles previos de ese mundo.

#### *4.4.2.3 Modos de juego*

botNeumann++ está diseñado con cuatro modos de juego inspirados en videojuegos casuales. Existe un botón para cada modo en la pantalla de menú (Figura 4.19 p.145).

1. El modo de juego "Entrenar" provee un tutorial y niveles que ayudan a un estudiante en el proceso de familiarización con la herramienta y a aprender el lenguaje de programación.
2. El modo de juego "Misiones" permite al jugador resolver ejercicios de una base de datos accesible a través de internet, de forma similar a las colecciones de retos disponibles en jueces automáticos usados en programación competitiva. Está ideado para ayudar al

estudiante a adquirir un nivel de conocimiento avanzado sobre la programación de computadoras.

3. El modo de juego "Cooperar" plantea retos que requieren la cooperación de dos o más estudiantes para poderlos resolver. Por ejemplo, ejercicios de programación distribuida, donde los estudiantes deben dividir la manufactura en dos o más fábricas (computadoras conectadas por internet).
4. El modo de juego "Crear" permite al estudiante diseñar sus propios ejercicios de programación, y compartirlos con otros jugadores a través de la base de datos.

Cada uno de estos modos de juego provee reconocimiento social al jugador, ya que requieren niveles de conocimiento incrementales. Esta tesis se concentra en el modo de "Entrenamiento" por razones de delimitación. Los tres restantes modos son una oportunidad para investigación futura.

#### *4.4.2.4 Juez automático*

A inicios de la década del 70, varias universidades en Estados Unidos organizaron concursos de programación. En 1977 la organización de los concursos se realizó en conjunto con la Conferencia de Ciencias de la Computación de la *ACM* (del inglés, *Association for Computing Machinery*) y recibieron el nombre de *ACM-ICPC* (del inglés, *ACM International Collegiate Programming Contest*) [Antczak et al. 2018]. El *ACM-ICPC* influenció otros concursos e incluso, la forma de reclutar informáticos en varias compañías tecnológicas en el mundo. En el área de la educación de la computación, surge especial interés por los retos algorítmicos planteados en los concursos de programación, y por la herramienta que evalúa a los equipos concursantes [Queirós and Leal 2013, p.38,39]. Esta herramienta se llama **juez automático** porque provee realimentación a los participantes y asigna puntajes que designan a los ganadores. El proveer retos algorítmicos, realimentación inmediata, y calificaciones oportunas a los estudiantes de programación, hace llamativo el aprovechar las enormes bases de ejercicios y los jueces automáticos construidos por los concursos de programación [Antczak et al. 2018]. Sin embargo, la elevada dificultad de los ejercicios y la poca realimentación de los jueces automáticos no son aptos para el nivel introductorio de estos cursos. Por tanto, investigaciones se han efectuado con el fin de adaptar estas herramientas a las necesidades de los aprendices de la programación [Petit et al. 2012]. El diseño de

botNeumann++ colabora en esta adaptación de los jueces automáticos para la educación de la programación.

Un juez automático de un concurso de programación plantea retos, los cuales tienen un enunciado textual del problema a resolver, que puede incluir imágenes, ejemplos de datos que los programas recibirán de entrada, y ejemplos de las salidas correspondientes que el programa debe generar. Los retos en botNeumann++ se representan con niveles. Cuando un estudiante selecciona un nivel, botNeumann++ presenta el enunciado del reto en la pestaña "Descripción" (① en Figura 4.21).

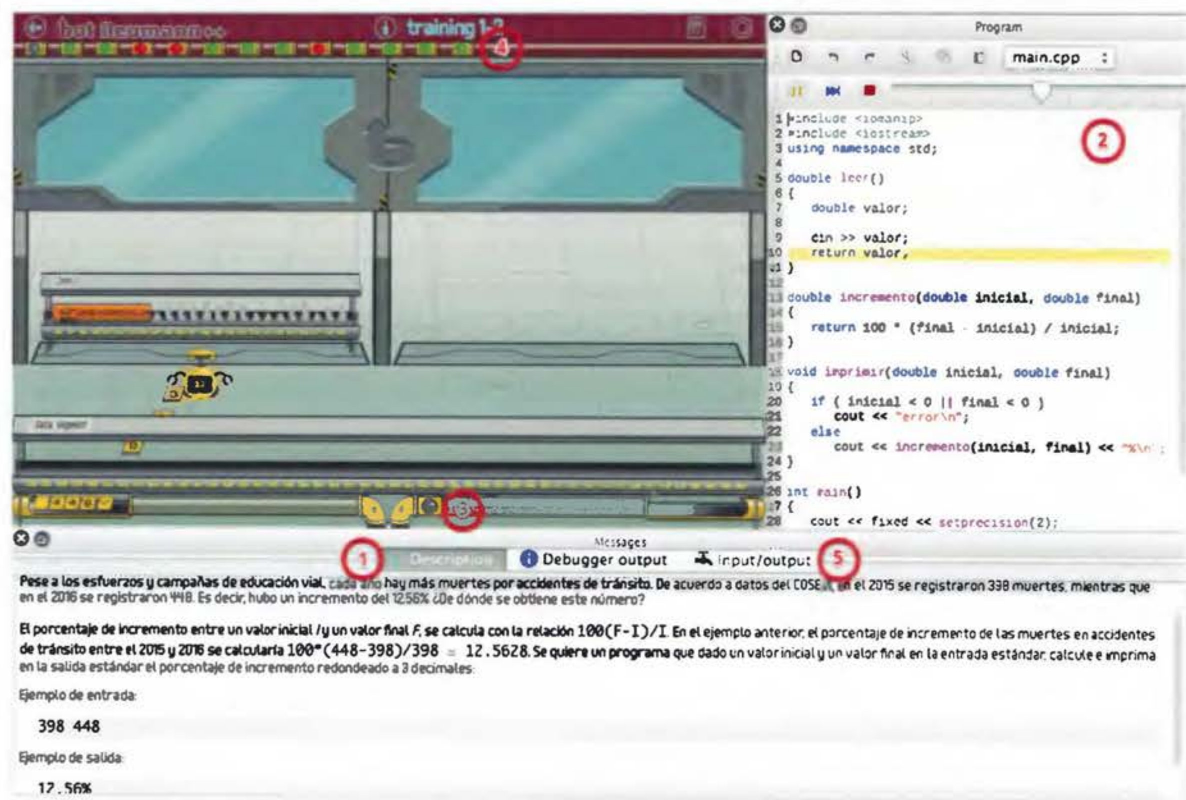



Figura 4.21. botNeumann++ como un juez automático

Los ejercicios pueden incluir código fuente inicial el cual se provee a los equipos que concursan. Esta capacidad es especialmente útil en ambientes de aprendizaje, para proveer código con errores o con características ricas para ejemplificar un concepto de programación. botNeumann++ carga el código fuente inicial en el segmento de código (② en Figura 4.21), y los estudiantes lo modifican para resolver el problema.

Cada ejercicio en un juez automático tiene uno o más casos de prueba. Un **caso de prueba** es una pareja de un texto de entrada y un texto de salida esperada. Cuando el programa de un competidor se somete al juez automático, el juez corre el programa del competidor tantas veces como casos de prueba tenga el ejercicio. En cada ejecución, el programa recibirá por entrada estándar el texto de entrada del caso de prueba. El juez automático captura la salida del programa del competidor y la compara contra la salida esperada del caso de prueba. Si ambas salidas coinciden exactamente, el competidor obtiene los puntos que el caso de prueba estipule, de lo contrario, falla el caso de prueba.

botNeumann++ permite al diseñador de los niveles incluir una cantidad arbitraria de casos de prueba. Cuando el estudiante corre su solución, botNeumann++ anima la ejecución del mismo como de costumbre, pero además agrega un dispositivo luminoso en el tubo neumático de la salida estándar (③ en Figura 4.21). Cada cápsula que pasa por este dispositivo es comparada contra la salida esperada del caso de prueba. Mientras las salidas coinciden, el dispositivo emite una luz verde. En el momento que se detecta una discrepancia, el dispositivo se torna rojo y se mantiene en este color. Por tanto, el estudiante recibe realimentación de si se ha pasado o no el caso de prueba a través de este código de colores.

Si el ejercicio contiene varios casos de prueba, botNeumann++ corre la solución del estudiante contra todos los casos de prueba, y un tubo con dispositivos luminosos (④ en la Figura 4.21) indica el resultado de cada caso de prueba. El estudiante puede accionar cualquiera de estos dispositivos, por ejemplo, aquellos en rojo. botNeumann++ reinicia la animación con los datos de entrada del caso de prueba seleccionado para permitir al aprendiz estudiar las razones del fallo.

Un juez automático tradicional sólo reporta los puntos ganados o la cantidad de casos de prueba aprobados a los equipos concursantes, ninguna información adicional. En cambio, los estudiantes de programación necesitan abundante realimentación. botNeumann++, además de la animación de la máquina nocional, permite a los estudiantes comparar las salidas de sus programas contra las salidas esperadas por los casos de prueba, y resalta el texto donde difieren en la pestaña rotulada "Input/output" (⑤ en la Figura 4.21). Esta comparación es sumamente útil para ayudar a los aprendices a detectar errores en el código.

El formato en que botNeumann++ representa los ejercicios, es una especificación *XML* (del inglés, *EXTENSIBLE MARKUP LANGUAGE*), inspirada en otras notaciones para representar ejercicios

de jueces automáticos existentes, más las necesidades particulares de visualización de botNeumann++[17]. La especificación resultante incluye características avanzadas, derivadas de las necesidades del autor de esta tesis para la creación de ejercicios, como la capacidad de agregar generadores automáticos y validadores automáticos de casos de prueba.

### 4.4.3 Validación de botNeumann++

El diseño de botNeumann++ fue validado mediante entrevistas con dos investigadoras de *CARNEGIE MELLON UNIVERSITY*, durante la pasantía del autor de esta tesis en Pittsburgh, Pennsylvania. Durante las sesiones, el autor de esta tesis presentó imágenes del modelo botNeumann++ en una computadora portátil, como la pantalla de visualización de la Figura 4.22. Las imágenes estáticas fueron controladas por el autor mediante un programa de diseño gráfico, mientras explicaba el funcionamiento de la visualización lúdica de programa. Mediante un diálogo, se le solicitó a las investigadoras observaciones y sugerencias durante la sesión.

La primera entrevista fue realizada con la doctora Wanda Dann, directora del proyecto Alice e investigadora en visualización y lenguajes de programación. La doctora Dann resaltó el valor de ser la primera visualización concreta sobre concurrencia que ella haya visto, que en su opinión es el principal aporte al conocimiento.

La segunda entrevista fue realizada a la doctora Amy Ogan, investigadora del *HUMAN-COMPUTER INTERACTION INSTITUTE* (*HCII*), en el campo de la tecnología educativa. El autor de esta tesis fue oyente de su curso *DESIGN OF EDUCATIONAL GAMES*. La doctora Ogan estuvo muy preocupada de la enorme complejidad de la visualización e hizo varias recomendaciones orientadas a simplificarla. Entre ellas, sugirió en lugar de visualizar y evaluar la mayoría de conceptos de la máquina nocional, visualizar uno o a lo sumo dos conceptos. Aunque esta sugerencia reduciría los plazos temporales de la investigación, no se realizó debido a que la separaba del objetivo inicial sobre visualizaciones generales de programa, y porque reducía el sentido de usar alegorías visuales en lugar de metáforas.

---

[17] La especificación puede consultarse en https://raw.githubusercontent.com/citic/botNeumann/master/units/botnu-1.0.dtd

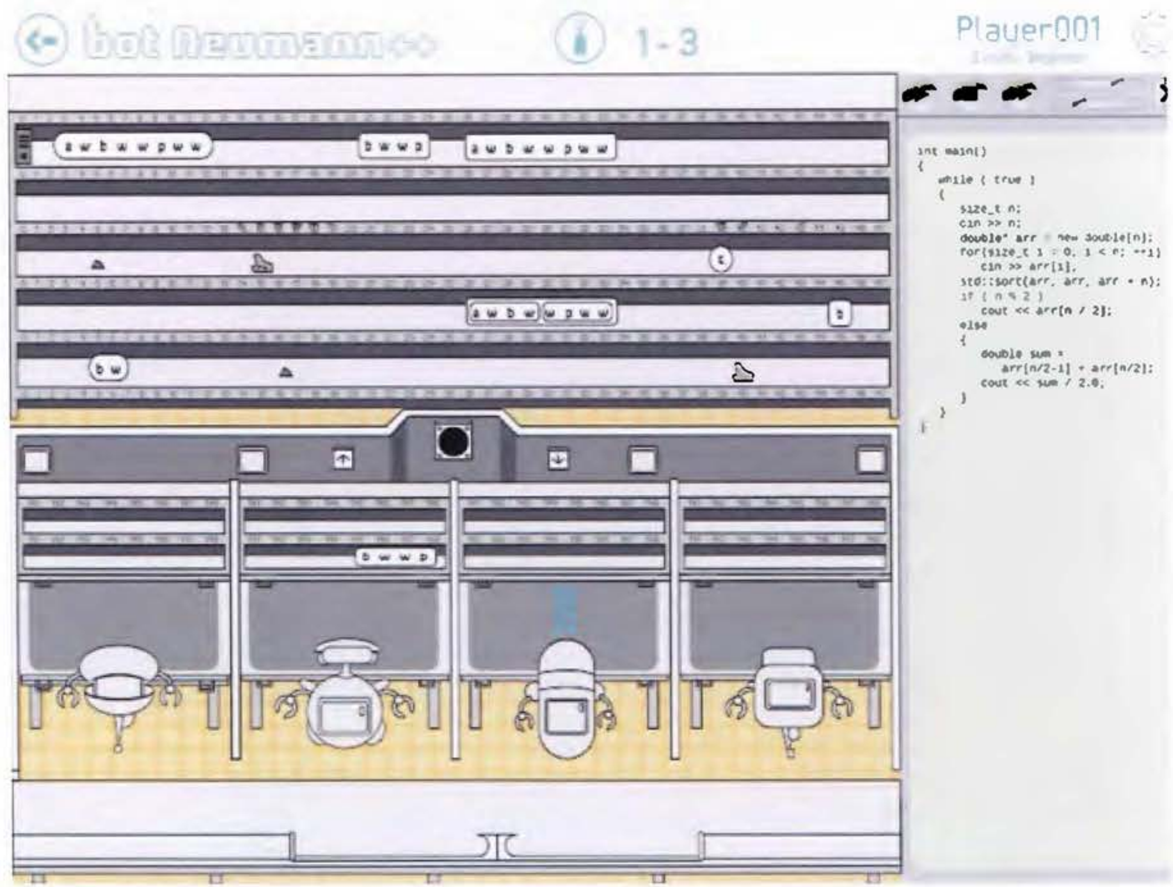



Figura 4.22. Pantalla de visualización de botNeumann++ usada durante la validación

La segunda recomendación importante de la Dra. Ogan fue cambiar la metodología de desarrollo por una escalonada, la cual incluye varias iteraciones de evaluación de los prototipos. Esta sugerencia se tomó en consideración y el diseño fue implementado y evaluado en diferentes artefactos: prototipo en papel, prototipo en PowerPoint y prototipo en C++, que son documentados en el próximo capítulo.

Durante las sesiones con ambas investigadoras, surgieron varias sugerencias sobre la interfaz y la interacción que fueron incluidas en el diseño de botNeumann++, como se aprecia en la Figura 4.23. Por ejemplo, se intercambiaron los tubos de entrada y de salida para hacer natural la animación del flujo de datos, la metáfora de oficina cambió a una fábrica, las mesas cambiaron a estantes curvos para cápsulas neumáticas, y las funciones se apilaron hacia el fondo por claridad, entre otras mejoras.

Figura 4.23. Pantalla de visualización después de la validación de botNeumann++

En general, el diseño tuvo una muy buena aceptación y comentarios positivos sobre su interacción con estudiantes. Ambas investigadoras lo consideraron válido para ser implementado y evaluado, lo cual es el tema del siguiente capítulo.

# 5 IMPLEMENTACIÓN Y EVALUACIÓN DE BOTNEUMANN++

Este capítulo documenta el desarrollo del cuarto y último objetivo de la tesis: "Evaluar experimentalmente la efectividad, para ayudar a estudiantes a comprender una máquina nocional, de un prototipo de la visualización de programa diseñada en el objetivo 3". Aunque el objetivo refiere explícitamente a la evaluación experimental, otros métodos de evaluación previos al experimento fueron realizados siguiendo el ciclo de evaluación de la metodología de la ciencia del diseño (① y ② en Figura 5.1). En este ciclo se aplicaron las sugerencias de la doctora Amy Ogan de crear prototipos incrementales. Se implementaron dos prototipos de botNeumann++, que conforman en sí dos pruebas de factibilidad del modelo y se reportan las principales lecciones aprendidas. El primer prototipo se hizo en *MICROSOFT POWERPOINT* y fue evaluado por usabilidad ② (sección 5.1). El segundo prototipo fue implementado en C++ y se evaluó por usabilidad ② (sección 5.2) y se comparó su eficacia con herramientas tradicionales a través de un experimento ③ (sección 5.3), el cual confirmó la hipótesis de investigación. El prototipo en C++ es el principal producto ingenieril que puede usarse en contextos de aprendizaje de la programación en C/C++ ④ y se encuentra disponible al mundo por ser un proyecto de software libre.

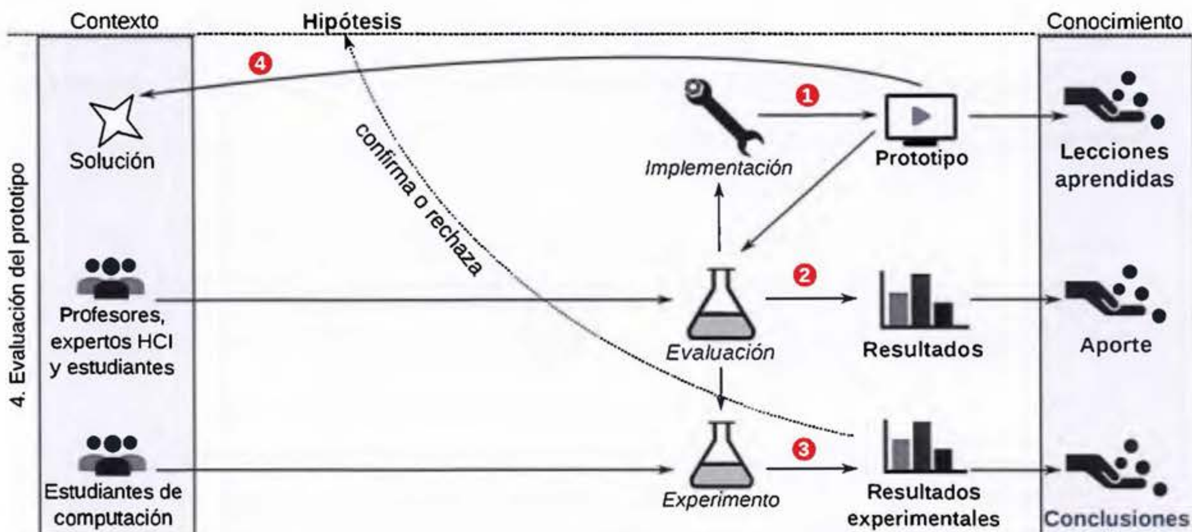



Figura 5.1. Metodología del objetivo 4

## 5.1 Prototipo 1: PowerPoint

Después de ser validado exitosamente el diseño botNeumann++ en *CARNEGIE MELLON UNIVERSITY*, se ejecutó el ciclo de evaluación según la metodología de la ciencia del diseño (Figura 5.1). Antes del desarrollo de los dos prototipos en software, se construyó una "maqueta" en papel de botNeumann++, la cual pudo ser manipulada por un experimentador mediante un protocolo de mago de oz. La maqueta se diseñó para estudiantes del curso de "Programación II", dado que es el curso donde aprenden C++. Como tarea se incluyeron dos problemas de programación estructurados como ejercicios de un juez automático. Se escogió un primer problema trivial, de imprimir la suma de dos números, con el propósito de introducir la herramienta y los conceptos de visualización. Para el segundo problema se escogió corregir una fuga de memoria, dado que la administración de memoria dinámica es un tema prioritario de acuerdo a los resultados de la encuesta en la sección 4.2. Además, el autor ha identificado que la fuga de memoria es un concepto conflictivo para los estudiantes de la ECCI, potencialmente debido a la influencia de la administración de memoria dinámica en *JAVA* que ellos aprenden en el curso previo [Hidalgo-Céspedes et al. 2016b]. Se desea determinar si la visualización de programa es útil para ayudar a los estudiantes a detectar la fuga de memoria y corregirla.

Se hizo una evaluación de la maqueta con un estudiante de Programación II. Sin embargo, la manipulación de una cantidad considerable de elementos gráficos en papel (por ejemplo, variables), saturó al experimentador y la sesión requirió un tiempo mayor al esperado. Por tanto, la maqueta fue descontinuada para evaluaciones adicionales. Por recomendación de Gustavo López-Herrera, se elaboró un prototipo usando animaciones en *MICROSOFT POWERPOINT* con algunas rutinas de interacción implementadas en *VISUAL BASIC FOR APPLICATIONS*. Este prototipo pudo ser evaluado por usabilidad y es el que se reporta en esta sección.

### 5.1.1 Implementación y lecciones aprendidas

El primer prototipo se construyó como una presentación de 37 diapositivas en *MICROSOFT POWERPOINT*, al cual se le llamará de forma corta **prototipo PowerPoint**, para distinguirlo del segundo prototipo implementado en C++. Los eventos del ratón y del teclado para pasar entre

diapositivas fueron deshabilitados y reemplazados por imágenes. Estas imágenes al ser accionadas con el ratón: (1) cambian entre diapositivas previas o posteriores (lo que permitió ciclos), (2) inician animaciones, o (3) ejecutan código de *VISUAL BASIC FOR APPLICATIONS*.

Dado que botNeumann++ es una visualización lúdica de programa, se incorporaron elementos de un videojuego casual, como un tutorial, una pantalla de menú (Figura 5.2), un mapa de niveles (Figura 5.3), y una pantalla de nivel. En la Figura 5.4 se muestran los posibles flujos de interacción que permite el prototipo.

Figura 5.2. Menú del juego en el prototipo PowerPoint de botNeumann++

Figura 5.3. Mapa de niveles del prototipo

Al iniciar el prototipo (rotulado ① en la Figura 5.4) se presenta el *menú del juego* (Figura 5.2), e inicia el tutorial que se expande a las otras pantallas. El tutorial lo realiza un robot que narra una historia del juego similar a la presentada en el apartado 4.4.2.1, pregunta por el nombre del usuario y su experiencia con C++, explica cómo usar la visualización y el significado de las metáforas visuales de los conceptos de programación. De esta forma, no se requirió intervención del investigador durante la fase de familiarización con la herramienta.

Al presionar el botón de "Entrenar" en la pantalla de menú (marcado ② en el flujo de la Figura 5.4), el prototipo muestra el mapa de niveles (Figura 5.3). Se implementaron dos niveles, correspondientes a los utilizados en la maqueta en papel. El nivel rotulado "1-1" siempre está habilitado, y el nivel "1-2" sólo se habilita cuando el usuario ha resuelto el nivel anterior. Al accionar un nivel que esté habilitado, se carga en la pantalla de nivel correspondiente (rutas ③ y ⑥ en la Figura 5.4).

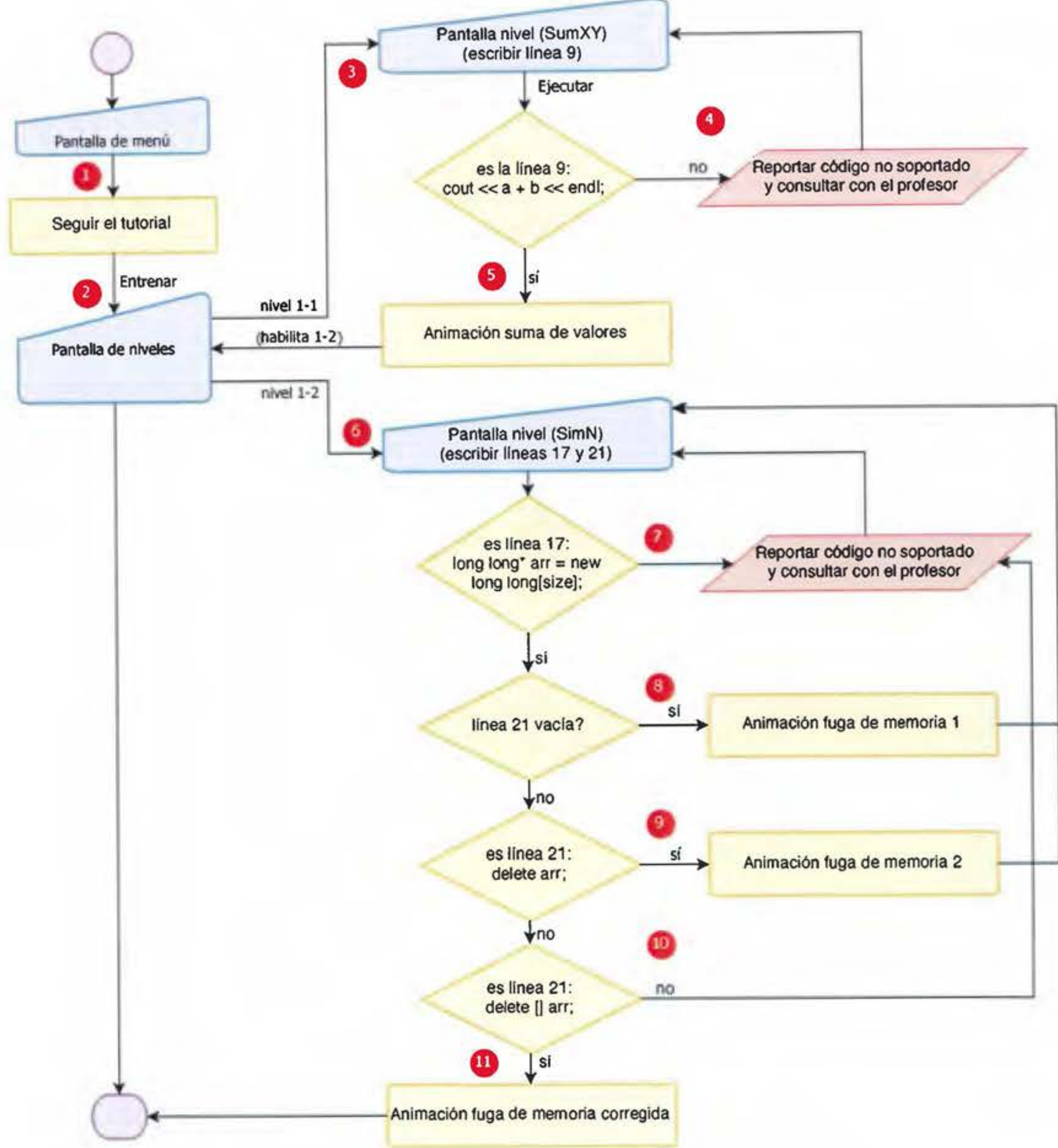



Figura 5.4. Flujo de interacción del prototipo PowerPoint

#### *5.1.1.1 Nivel 1-1*

Al iniciar cualquiera de los dos niveles se presenta parte del tutorial. Para el nivel 1-1, el tutorial explica cómo usar la interfaz de botNeumann++ (Figura 5.5). Esta interfaz está influenciada por el funcionamiento de los jueces automáticos. El tutorial invita al usuario a leer la descripción del problema en la parte inferior de la pantalla (rotulada ① en la Figura 5.5). Para el nivel "1-1" se pide completar un programa que lee dos números reales de la entrada estándar y debe imprimir la suma de ellos en la salida estándar. Seguidamente el

tutorial explica de dónde se obtiene el código inicial cargado en el segmento de código (② en la Figura 5.5), usando la narración del juego como contexto. El tutorial solicita visualizar ese código inicial sin modificaciones. Cuando el usuario presiona el botón de reproducir ③ en el segmento de código, botNeumann++ inicia la animación del código fuente y el tutorial interviene para explicarla. El tutorial explica el efecto del código fuente en: los aposentos {segmentos de memoria} en que se divide la fábrica, la animación al invocar la función punto de entrada (`main`), la asociación entre los números en el robot y los números de la línea en el segmento de código, la entrada y salida estándar, y el significado del dispositivo luminoso en la salida estándar. Finalmente solicita al usuario corregir el programa.

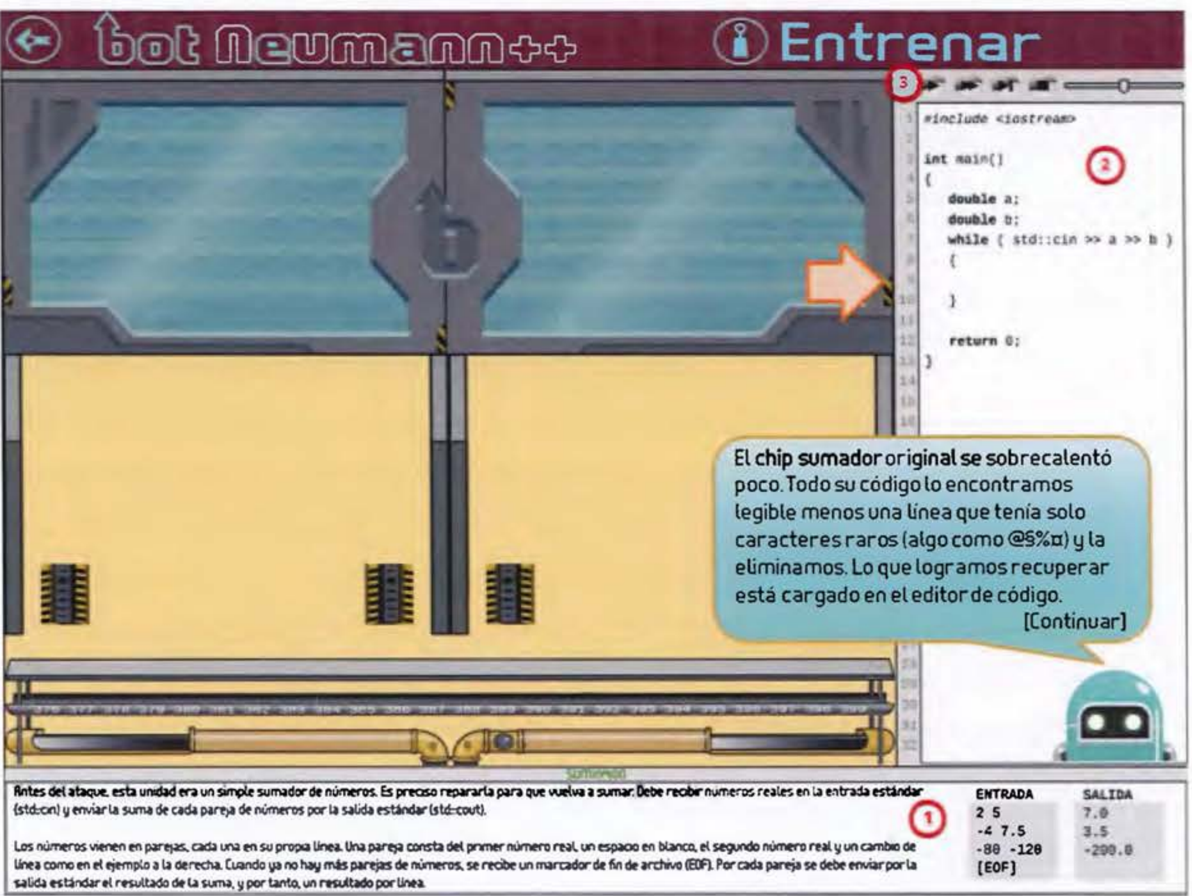



Figura 5.5. Pantalla de nivel 1-1 presentando parte del tutorial

Una limitación de la plataforma utilizada (*MICROSOFT POWERPOINT*) es que no se puede invocar un programa ejecutable externo y controlar su entrada y salida interactivamente. Esta capacidad es necesaria para permitir que el usuario pueda escribir programas arbitrarios en el editor de texto, dado que se necesitaría invocar un compilador de C/C++, un depurador, y el

ejecutable del usuario. Por este motivo, se delimitó a que el usuario pudiera editar pocas líneas del programa. En el caso del nivel "1-1", sólo la línea 9 podía modificarse.

Cuando el usuario presiona el botón "Reproducir" para iniciar la animación, código en *VISUAL BASIC FOR APPLICATIONS* compara el texto que el usuario haya ingresado en la línea 9 contra algunos textos esperables. Como se ve en el ciclo rotulado con ④ en la Figura 5.4, si el usuario escribe un código distinto a los esperados, se presenta un mensaje de error indicando que el texto no es soportado por el intérprete y que consulte con el profesor. El profesor, mediante un protocolo de mago de oz durante las evaluaciones de usabilidad, proveyó una respuesta ubicándola en la pantalla, si era posible, o una explicación oral en casos complejos.

Si el usuario ingresa uno de los textos esperados, es decir, código C++ que imprime la suma de las dos variables, el prototipo automáticamente corre una animación que resuelve el problema del nivel "1-1" (flujo ⑤ en la Figura 5.4). Cuando la animación concluye, felicita al usuario y regresa al mapa de niveles, donde el nivel "1-2" estará habilitado (Figura 5.3).

#### *5.1.1.2 Nivel 1-2*

Al iniciar el nivel "1-2" (flujo ⑥ en la Figura 5.4), el tutorial sugiere al usuario leer el enunciado del problema, sobre corregir un programa que lee arreglos de la entrada estándar y debe indicar en la salida estándar si esos arreglos son simétricos o no. Seguidamente invita al usuario a correr el programa sin modificaciones. El código inicial lee los arreglos en variables locales, que desbordan la pequeña pila disponible en botNeumann++. En este punto el tutorial pregunta al estudiante la razón del desbordamiento, le solicita detener la animación, explica el segmento de memoria dinámica (parte superior de la Figura 5.6) que estaba deshabilitado en el nivel 1-1, y reta al estudiante a corregir el programa usando memoria dinámica.

El nivel "1-2" permite modificar únicamente dos líneas en el segmento de código, una para crear el arreglo en memoria dinámica (línea 17) y otra para eliminarlo (línea 21). A diferencia del nivel "1-1", el tutorial no indica cuáles líneas se pueden modificar, con el fin de no sesgar los estudiantes hacia el uso del operador `delete[]` y evaluar qué tan útil es la animación para ayudarles a detectar y comprender la necesidad de eliminar la memoria dinámica.

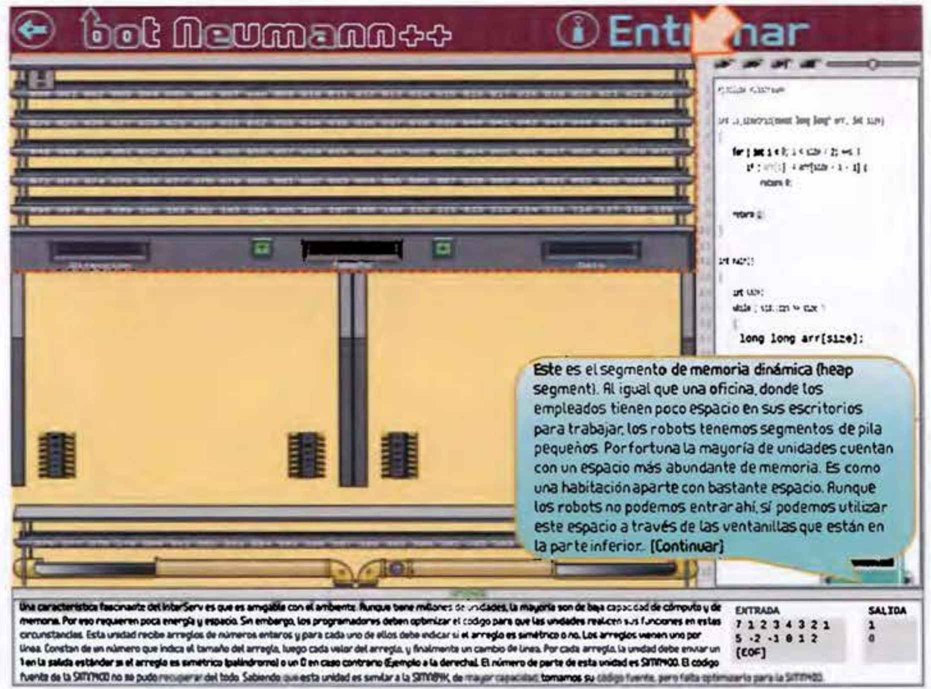



Figura 5.6. Tutorial introduciendo el segmento de memoria dinámica en el nivel 1-2

Como se indica en los flujos ⑦ a ⑪ de la Figura 5.4, el prototipo reacciona de acuerdo a los textos que el usuario ingrese en las líneas 17 y 21. Si escribe textos no esperados (rutas ⑦ o ⑩), se presenta el mensaje que activa el protocolo de mago de oz con el profesor. Si se crea el arreglo correctamente en la línea 17, pero se deja la línea 21 vacía (flujo ⑧), se realiza una animación que lee tres arreglos de la entrada estándar y los aloja en el segmento de memoria dinámica. Sin embargo, el tercero de los arreglos desborda este segmento por falta de capacidad, dado que no se eliminaron los arreglos previos (Figura 5.7). Si el usuario trata de eliminar el arreglo con el operador `delete` (flujo ⑨), se hará una animación que crea los dos primeros arreglos y sólo les elimina el primer elemento de cada uno de ellos. Por lo tanto, el segmento de memoria dinámica se desborda al crear el tercer arreglo, de la misma forma que la animación anterior. Se decidió eliminar sólo el primer elemento en la animación porque, de acuerdo al estándar C++, usar el operador `delete` para eliminar un arreglo produce un comportamiento indeterminado. En cualquiera de estas dos animaciones, el prototipo se mantiene en el nivel 1-2 donde el estudiante puede corregir el código y reintentar.

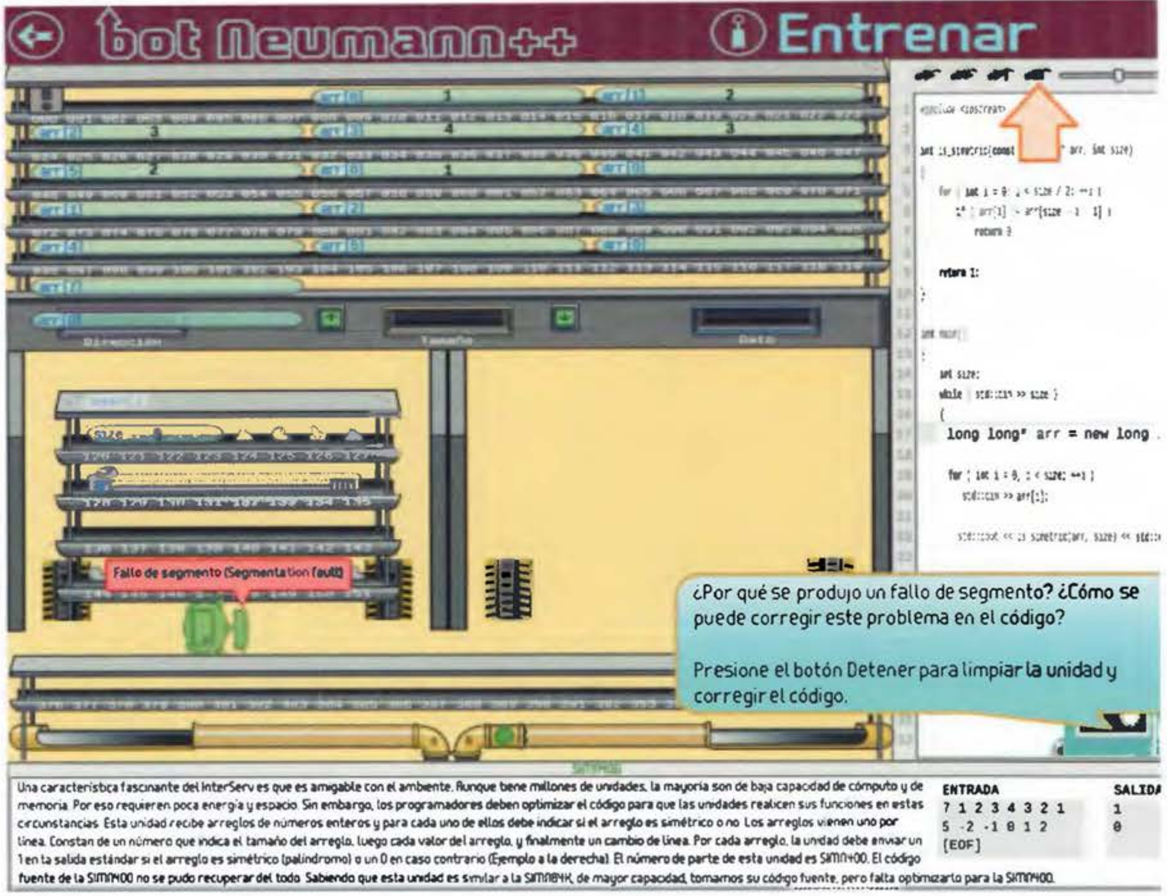



Figura 5.7. Desbordamiento del segmento de memoria dinámica

Siguiendo el flujo ⑪ de la Figura 5.4, si el estudiante crea correctamente el arreglo en la línea 17 y lo elimina en la línea 21 con el operador `delete[]`, el prototipo realiza una animación que crea y elimina arreglos sin desbordar el segmento de memoria dinámica, es decir, resuelve el problema adecuadamente. El prototipo felicita al usuario y regresa al mapa de niveles donde puede ver que ambos están resueltos, se le indica que la sesión ha terminado y que se tendrán más niveles en versiones futuras.

#### *5.1.1.3 Lecciones aprendidas*

La implementación del prototipo en *MICROSOFT POWERPOINT* generó varias lecciones aprendidas. Primero se discuten las ventajas. Esta herramienta se escogió por la familiaridad previa del autor y porque facilita la creación de animaciones usando una interfaz amigable. La implementación del prototipo tardó aproximadamente tres semanas, lo cual fue considerablemente más rápido que el tiempo requerido por el segundo prototipo en C++ que

se presentará en la próxima sección. Al crear las animaciones se tuvo que escoger los elementos gráficos involucrados y sus posiciones en los tres ejes, además de los tipos de animaciones, sus efectos, orden y duración en la línea de tiempo. Estas decisiones no fueron triviales y la información generada fue sumamente útil para implementar el segundo prototipo en C++.

Como desventajas se incluyen las siguientes. El lenguaje de programación disponible en la herramienta (*VISUAL BASIC FOR APPLICATIONS*) no permite invocar programas externos y controlar su ejecución, como compiladores o depuradores, lo cual impidió implementar funcionalidad que permitiera al estudiante experimentar programas arbitrarios y poder visualizar su ejecución. Por esta razón, se tuvo que recurrir al protocolo de mago de oz. Este mecanismo permite obtener realimentación valiosa y oportuna de la visualización de programa, pero no es apto para la eventual evaluación experimental propuesta en el cuarto objetivo de esta tesis. Otra lección surgida es que *MICROSOFT POWERPOINT* resultó adecuado para animaciones sencillas en diapositivas tradicionales, como las animaciones iniciales del tutorial de botNeumann++, pero muy limitado para las animaciones finales que requerían el manejo de cientos de elementos gráficos. Pese a estas limitaciones, este prototipo permitió aprender de las primeras interacciones de botNeumann++ con usuarios reales, como se expone en la siguiente subsección.

### 5.1.2 Evaluación de usabilidad

Dado que las visualizaciones de programa son sistemas interactivos, la usabilidad es una de sus propiedades clave y debe ser evaluada, además de su efectos en los procesos de aprendizaje [Urquiza-Fuentes and Velázquez-Iturbide 2009, p.9:2-3]. La **usabilidad** es la facilidad con que las personas aprenden y usan un artefacto para realizar una tarea (adaptado de [Tullis and Albert 2013, p.5; Dumas and Redish 1999, p.4]). Si un artefacto tiene problemas de usabilidad, los usuarios no podrán o tendrán dificultades para conseguir sus objetivos con él. En el caso de una visualización de programa, problemas críticos de usabilidad impedirán a los estudiantes comprender la máquina nocional de un lenguaje de programación a través de la visualización. Por lo tanto, problemas severos de usabilidad en una herramienta pueden sesgar los resultados de una comparación experimental contra otra herramienta. Por este

motivo, es importante detectar y corregir problemas críticos de usabilidad en el prototipo de botNeumann++ antes de compararlo experimentalmente.

De acuerdo a la nomenclatura de [Dumas and Fox 2012], se condujo una prueba diagnóstica de usabilidad del prototipo PowerPoint realizada localmente, con cuatro propósitos: (1) explorar la usabilidad de conceptos iniciales de diseño, (2) diagnosticar problemas de usabilidad, (3) corregir problemas de usabilidad, y (4) validar la usabilidad. El prototipo tuvo un nivel de funcionalidad interactivo, excepto en las situaciones donde se requirió intervención del mago de oz. El alcance de la prueba de usabilidad fue del prototipo completo, y por tanto, de los posibles flujos en el diagrama de la Figura 5.4 (p. 158). Las sesiones fueron moderadas por el autor de esta tesis, quien además realizó el protocolo de mago de oz. Por tanto, las sesiones fueron locales, realizadas en las instalaciones de la Escuela de Ciencias de la Computación e Informática.

#### *5.1.2.1 Sesiones y tareas*

La evaluación de usabilidad estuvo planeada para una sesión de 45 minutos a una hora por participante, divida en cuatro etapas:

1. *Presentación.* El moderador proveyó instrucciones sobre la tarea a realizar, y enfatizó la evaluación de la herramienta y no del participante. El moderador realizó un ejemplo del protocolo "pensar en voz alta" y aclaró potenciales dudas del participante. El participante firmó el consentimiento informado.
2. *Tarea 1.* El participante realizó la tarea de entrenamiento (nivel "1-1") sobre imprimir la suma de dos números, detallada en el apartado 5.1.1.1 (p. 158).
3. *Tarea 2.* El participante realizó la tarea de evaluación de la visualización (nivel "1-2"), sobre determinar arreglos simétricos usando memoria dinámica sin producir fugas de memoria. Esta tarea se detalló en el apartado 5.1.1.2 (p. 160).
4. *Entrevista.* El estudiante respondió un cuestionario mediante una entrevista, para conocer su opinión sobre la herramienta y si ésta le había ayudado a comprender el concepto de fuga de memoria. Se detalla más adelante en el apartado 5.1.3.6.

El prototipo PowerPoint se instaló en una computadora portátil. La interacción del participante con la herramienta se grabó mediante un software de captura de pantalla entrelazado al sonido ambiental registrado con el micrófono de la computadora. Se les pidió a

los participantes "pensar en voz alta" (en inglés, *CONCURRENT THINK-ALOUD PROTOCOL*) mientras realizaban las etapas correspondientes a la tarea 1 y tarea 2.

#### *5.1.2.2 Participantes*

Aunque no existe acuerdo en la literatura, se ha recomendado que las pruebas de usabilidad diagnóstica se realicen con poblaciones de entre cinco y ocho usuarios finales de diferente nivel de pericia [Dumas and Fox 2012] o entre tres a cinco expertos en el dominio de forma longitudinal [Sensalire et al. 2009, p.19]. Los usuarios finales de botNeumann++ son estudiantes de la Escuela de Ciencias de la Computación e Informática que deben programar en C++, y los expertos en el dominio son profesores de programación en este lenguaje. También se consideraron como expertos en el dominio a especialistas en interacción humano-computador (*HCI*), dado que la interfaz gráfica de botNeumann++ es novedosa en cuanto a la introducción de una alegoría visual lúdica. Por esta razón, la prueba de usabilidad diagnóstica del prototipo PowerPoint se realizó con seis estudiantes del curso "Programación II" como usuarios finales, tres profesores del mismo curso como expertos en el dominio, y tres profesionales en *HCI* como expertos en usabilidad. De acuerdo a los resultados de la revisión de literatura hecha por [Urquiza-Fuentes and Velázquez-Iturbide 2009] y una revisión informal de herramientas posteriores hecha por el autor de esta tesis, botNeumann++ es muy probablemente la primera visualización de programa que se evalúa por usabilidad con estas tres poblaciones.

La evaluación de usabilidad se realizó al finalizar el primer ciclo lectivo de 2015. Los estudiantes fueron seleccionados por los tres profesores que impartieron "Programación II" en dicho ciclo, entre aquellos que habían aprobado el curso, con diferentes niveles de rendimiento académico comprendidos entre 7.0 y 10.0 (diferentes niveles de pericia). Trece estudiantes fueron recomendados por los profesores, los cuales fueron invitados por correo electrónico a participar. No se ofreció recompensa monetaria o académica. Ocho estudiantes (una mujer y siete varones) respondieron la invitación y participaron en tres versiones del prototipo, como se explica a continuación.

El primer estudiante interaccionó con la maqueta en papel el 22 de julio de 2015. Tras esta prueba se decidió reemplazar el prototipo en papel. Una vez construido el prototipo en *MICROSOFT POWERPOINT*, se realizó una prueba piloto con el segundo estudiante el 7 de agosto

de 2015. Tras esta prueba se incorporaron varios cambios en el prototipo. Por ejemplo, se ajustó la velocidad de las animaciones, se resaltaron las líneas que estaban siendo ejecutadas en el segmento de código, y se crearon animaciones para los dos tipos de fuga de memoria en el nivel 1-2 (rutas ⑧ y ⑨ en la Figura 5.4, p.158).

Las evaluaciones de usabilidad se realizaron con los restantes seis estudiantes de "Programación II", tres profesores de este curso, y dos expertos en *HCI*, entre 17 a 29 de agosto de 2015. El tercer experto en *HCI* realizó la prueba el 17 de noviembre de 2015. El tipo de evaluación de usabilidad varió de acuerdo al grupo de participantes.

Los expertos en *HCI* realizaron evaluaciones heurísticas. Una **evaluación heurística** es un método informal de análisis de usabilidad donde los expertos comentan sobre el diseño de una interfaz [Nielsen and Molich 1990]. La evaluación realizada por el primer experto en *HCI* no fue interactiva. El experto evaluó la herramienta y proveyó un reporte con los problemas de usabilidad identificados. Las restantes dos evaluaciones heurísticas fueron interactivas con el moderador. A través de un diálogo, el moderador proveyó detalles técnicos sobre la máquina nocional de C++ y sobre el contexto educativo para facilitar la labor del experto en usabilidad.

A los profesores de programación se les solicitó identificar problemas en la herramienta recurriendo a su experiencia en la enseñanza de la programación. Aunque hubo un diálogo entre el moderador y el profesor, el moderador no proveyó apoyo o soluciones de los niveles, con el fin de observar si la herramienta ayudaba al profesor a descubrir fugas de memoria. Por tanto, las pruebas de usabilidad realizadas por los profesores fueron evaluaciones heurísticas con estudios observacionales. En un **estudio observacional** el moderador observa cómo el participante usa el sistema y toma notas de lo que considera importante sobre la evaluación [Urquiza-Fuentes and Velázquez-Iturbide 2009, p.9:3]. Durante la evaluación de botNeumann++ el moderador tomó notas en papel y además se grabó la interacción del profesor con la visualización a través de captura de pantalla y micrófono.

Los estudiantes participaron como usuarios mientras el moderador realizó estudios observacionales, con entrevistas posteriores. Mientras el participante realizaba las dos tareas (nivel "1-1" y "1-2"), el moderador mantuvo al mínimo las interacciones, excepto para los protocolos de mago de oz y para recordar al participante "pensar en voz alta" en caso de

silencio. El moderador anotó preguntas conceptuales formuladas por el participante durante las tareas, para responderlas después de la entrevista.

### 5.1.3 Resultados de la evaluación de usabilidad

Los datos generados durante las sesiones de evaluación fueron las notas del moderador y los videos capturados de la actividad en la pantalla y micrófono de la computadora portátil. El moderador tomó notas sobre un impreso en papel del diagrama de la Figura 5.4 (p. 158), que facilitó rastrear la navegación del participante con el prototipo y ubicar las observaciones en la etapa en que ocurrieron.

Durante el análisis de datos, el autor de esta tesis transcribió las observaciones de las hojas de notas y las interacciones de los videos a una hoja de cálculo. En las columnas de la hoja de cálculo se ubicaron los 12 participantes y en las filas se registraron las acciones relevantes a la prueba de usabilidad. Por ejemplo, si un participante emitió un comentario sobre una acción, se transcribió a la celda correspondiente. El nivel de detalle de la transcripción fue de paráfrasis, con el fin de hacer manejable la longitud los textos en las celdas, obtenidos de las 14 horas y 39 minutos de grabaciones. Las acciones en las filas se agruparon por la etapa donde ocurrieron de acuerdo al diagrama de flujo en la Figura 5.4 (p. 158) y los participantes en las columnas se agruparon por el tipo de usuario.

Con los datos transcritos se realizaron varios análisis como la duración de los estudiantes en las tareas, tasas de completitud de las tareas, problemas de usabilidad detectados, y respuestas al cuestionario final. Los resultados de estos análisis se reportan en los siguientes apartados.

#### *5.1.3.1 Mediciones de rendimiento*

El tiempo en la tarea es típicamente un indicador de eficiencia en herramientas que deben usarse repetitivamente [Tullis and Albert 2013, p.74], como en transferencias bancarias o trámites gubernamentales. Sin embargo, no siempre "más rápido es mejor", como ocurre en videojuegos o herramientas educativas donde mayor exposición a la herramienta puede provocar un mejor resultado educativo [Tullis and Albert 2013, p.74], como es el caso de botNeumann++.

Las duraciones de los seis estudiantes en cada una de las etapas de la sesión fueron obtenidas a partir de los videos y los promedios se grafican en la Figura 5.8. Los estudiantes tardaron más tiempo en la segunda tarea sobre la fuga de memoria (nivel "1-2") con un promedio cercano a los 47 minutos y una desviación estándar aproximada a los 15 minutos. El análisis cualitativo de los datos reveló que las duraciones se debieron principalmente a errores de sintaxis de C++ para crear los arreglos en memoria dinámica. Los estudiantes tuvieron claro que debían usar el operador new[ ], pero no recordaban su sintaxis, por lo tanto, realizaron varios intentos de prueba y error en la visualización.

Se consideró como un intento cada vez que un participante presionó el botón de reproducir. Esta acción provoca que el prototipo PowerPoint simule compilar, y en caso de que los textos ingresados en las líneas editables coincidan con algunos de los esperados, realizará la animación de programa correspondiente. Los estudiantes hicieron en promedio 3.2 intentos ($\sigma$=1.17) para resolver el problema de la suma de dos números en el nivel "1-1". Por su parte, realizaron en promedio 11.3 intentos ($\sigma$=5.92) para resolver el problema de simetría de vectores en el nivel "1-2". El número promedio de intentos se denota con el signo "X" sobre los respectivos niveles en la Figura 5.8.

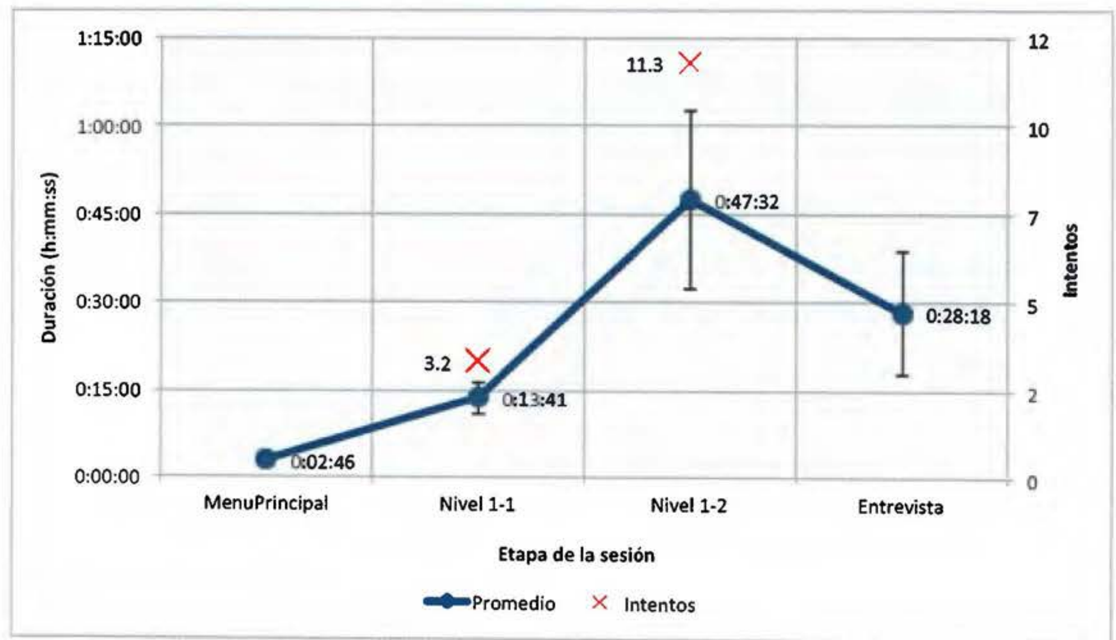



Figura 5.8. Duración promedio de los seis estudiantes en las etapas de la evaluación de usabilidad. Las barras de error indican la desviación estándar. Las "X" indican el número promedio de intentos.

Durante la sesión se les permitió a los estudiantes realizar búsquedas en la web a través de un navegador que les permitió superar los problemas de sintaxis. Todos los estudiantes completaron ambas tareas dado que no se impusieron límites de tiempo, a excepción de un estudiante quien requirió ayuda del moderador sobre la sintaxis para crear arreglos en su vigésimo intento del nivel "1-2". Después de esta ayuda, el estudiante completó el nivel sin intervención adicional del moderador. Por tanto, la *tasa de completitud* de la tarea de entrenamiento "1-1" fue del 100%, mientras que de la tarea de evaluación "1-2" fue de $5/6 \approx 83\%$.

La **eficiencia de la tarea** es la razón entre la tasa de completitud y el tiempo promedio [Tullis and Albert 2013, p.91]. Básicamente expresa la completitud de la tarea por unidad de tiempo. En la tarea de entrenamiento (nivel "1-1") los estudiantes avanzaron 7.3% de la tarea en cada minuto, mientras que esta eficiencia decayó a 1.8% de avance por minuto en la tarea de evaluación (nivel "1-2").

El número de intentos por errores de sintaxis tuvo también un efecto dilatador en la tercera etapa, la entrevista. Los estudiantes, lejos de expresar signos de aburrimiento, se mostraron sumamente intrigados por los errores sintácticos y sus dificultades en superar el reto planteado. Por tanto, en la tercera etapa se dedicó parte de la entrevista a aclarar dudas sobre la tarea, además de las preguntas inicialmente planeadas en el cuestionario.

##### *5.1.3.2 Errores de usabilidad*

Un **error de usabilidad** es un tipo de acción incorrecta por parte del usuario que lo desvía del camino para completar exitosamente una tarea con el sistema [Tullis and Albert 2013, p.83]. Por ejemplo, son errores de usabilidad ingresar un dato incorrecto, escoger una opción incorrecta de un menú, realizar una secuencia de acciones necesarias para una tarea en el orden incorrecto o saltarse algunas de ellas. Aunque no existe una definición aceptada, un **problema de usabilidad** (en inglés, *USABILITY ISSUE*) puede considerarse una característica conductual de un sistema que con probabilidad induce a un usuario a cometer un error de usabilidad (adaptado de [Tullis and Albert 2013, p.82,99,100]). Es decir, los problemas de usabilidad se consideran causas y los errores de usabilidad se consideran efectos [Tullis and Albert 2013, p.82], pero no todo error de usabilidad es causado por un problema de usabilidad.

De acuerdo a [Tullis and Albert 2013, p.83] es apremiante medir *errores de usabilidad* cuando un error implica una pérdida significativa de eficiencia, un incremento de los costos, o en el fallo de la tarea. En el caso del prototipo PowerPoint, los errores de usabilidad pueden implicar pérdida de eficiencia o el fallo de la tarea, por tanto es apremiante identificarlos.

Es esperable que hayan muchas oportunidades de error en un sistema, por lo que se debe restringir a aquellas de interés para la investigación [Tullis and Albert 2013, p.84]. En el caso del prototipo PowerPoint, las principales oportunidades de error que afectaron la eficiencia o el éxito en completar las tareas, ocurrieron en las líneas de texto donde los usuarios ingresaban código C++. Las tres líneas editables en el prototipo fueron, la línea 9 para imprimir la suma de dos números en el nivel "1-1", y las líneas 17 y 21 para crear y eliminar respectivamente el arreglo en el nivel "1-2". El gráfico de cajas de la Figura 5.9 muestra la cantidad de errores introducidos por estudiantes y profesores en las tres líneas mencionadas.

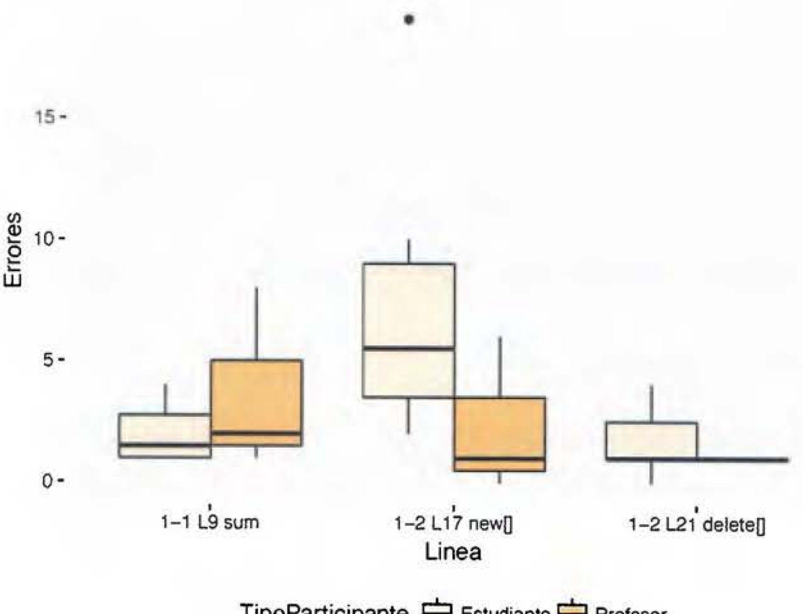



Figura 5.9. Cantidad de errores cometidos por profesores y estudiantes en las tres líneas editables

Las medianas en el gráfico de la Figura 5.9 indican que en general se introdujeron errores en las tres líneas sin importar el tipo de participante. Curiosamente los profesores introdujeron más errores en el nivel de entrenamiento ("1-1") que los estudiantes. Esto se debió a que los profesores conocen formas más complejas de lograr el mismo efecto en C++ pero introdujeron errores de digitación, o intentaron probar cómo reaccionaba el prototipo ante formas

elocuentes de código. En el nivel "1-2", los profesores se enfocaron en completar la tarea e introdujeron menos errores en ambas líneas que los estudiantes, como es de esperarse al comparar poblaciones de expertos contra principiantes. El gráfico revela que la mayor cantidad de errores se introdujeron al crear el arreglo en memoria dinámica (línea 17 del nivel "1-2"), que como se indicó anteriormente, se debieron a errores sintácticos del operador `new[]` que los estudiantes superaron con la ayuda de un navegador, excepto el estudiante representado por el valor extremo en la parte superior de la Figura 5.9.

#### *5.1.3.3 Problemas de usabilidad*

Identificar problemas de usabilidad y proveer recomendaciones de diseño es quizás el aporte más importante de una prueba de usabilidad [Tullis and Albert 2013, p.99]. De las transcripciones de los videos y notas del moderador se consideraron como problemas de usabilidad las acciones incorrectas y expresiones verbales de confusión, frustración, insatisfacción, e indecisión para lograr la tarea [Tullis and Albert 2013, p.103]. En total se tabularon 169 reportes hechos por los doce participantes. La severidad de cada problema de usabilidad fue codificada por el desarrollador del prototipo (el autor de esta tesis) con la escala de -1 a 2 explicada en el Cuadro 5.1. Esta escala fue adaptada de [Tullis and Albert 2013, p.101,104,106].

Cuadro 5.1. Escala de codificación de severidad de los problemas de usabilidad

| Código | Severidad | Descripción |
|---|---|---|
| -1 | Falso positivo | Problema reportado por un participante debido a una interpretación equivocada y que es poco probable que se reporte en evaluaciones posteriores. |
| 0 | Baja | Problema que molesta o frustra a los participantes que lo experimentan pero que no influye en completar o no la tarea. El problema afecta ligeramente su eficiencia y satisfacción. |
| 1 | Media | Problema que incrementa significativamente la dificultad de la tarea, pero no provoca que quienes la experimentan fallen la tarea, porque encuentran métodos alternativos. Sin embargo, el problema afecta significativamente su eficiencia y satisfacción. |
| 2 | Alta | Problema que provoca que los participantes que lo experimentan fallen la tarea, con efectos negativos en su eficacia, eficiencia y satisfacción. |

La Figura 5.10 grafica la cantidad de problemas de usabilidad (eje y) reportados por tipo de participante (eje x). Puede verse que los expertos en *HCI* y los estudiantes reportaron cantidades similares de problemas de usabilidad, mientras que los profesores hicieron cerca de la mitad de los reportes que los otros tipos de participantes. La gráfica incluye la cantidad

de reportes clasificada por severidad en cada grupo de participantes, que ayuda a visualizar las siguientes inferencias. La mayoría de reportes de falsos positivos fueron realizadas por expertos en *HCI*, en especial debidas a una prueba heurística no interactiva. Al recorrer el eje-x hacia la derecha, puede observarse que entre mayor conocimiento sobre el dominio de la programación, mayor es la proporción de problemas de severidad media reportados.

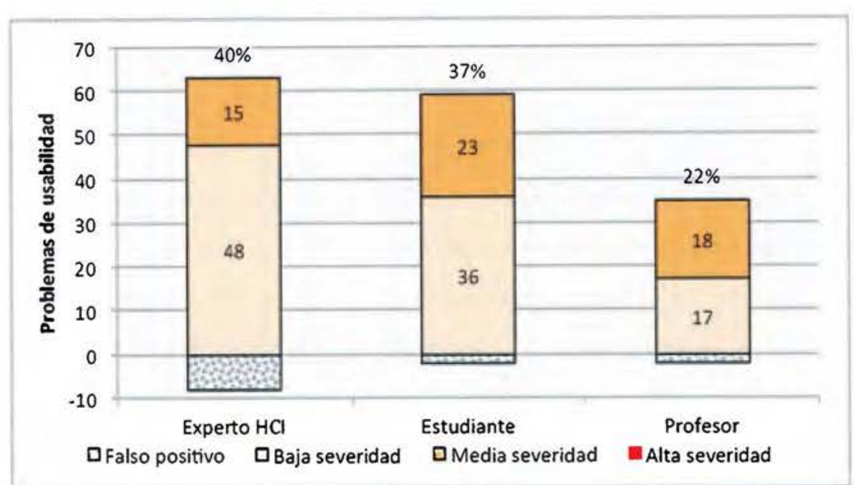



Figura 5.10. Problemas de usabilidad clasificados por severidad y por la población que los identificó

Ninguno de los 169 reportes fue considerado de severidad alta. Un problema de esta severidad impide que el participante logre realizar la tarea [Tullis and Albert 2013, p.104]. El único participante que requirió intervención del moderador para completar la tarea, había olvidado nociones sobre programación de memoria dinámica en C++, como el participante mismo expresó.

El prototipo se diseñó para una tarea de comprensión y resolución de problemas (detectar y corregir fugas de memoria), en lugar de una tarea de enseñanza y comprensión de un concepto (alojamiento de memoria dinámica). La ausencia de reportes de problemas de alta severidad se consideró el principal indicador de que el software es usable, y pudo ser una consecuencia positiva de la evaluación piloto.

Como se indicó al inicio de este apartado, identificar problemas de usabilidad y proveer recomendaciones de diseño es quizás el aporte más importante de una prueba de usabilidad. Es normal que de un problema de usabilidad se hagan varios reportes provenientes de distintos participantes, pero se trata del mismo problema, al que se le llama **problema único**

**de usabilidad** [Tullis and Albert 2013, p.108]. De los 169 reportes, 12 fueron falsos positivos. De los restantes 157 reportes, 75 fueron problemas únicos de usabilidad.

Se consideraron como más prioritarios de atender los problemas de usabilidad que tienen mayor severidad ($S$) y que fueron reportados por mayor número participantes ($R$). Para poder identificarlos, se calculó la prioridad como el producto de ambos valores de la forma $R(S+1)$. A modo de ejemplo, el Cuadro 5.2 lista los 14 problemas de usabilidad más prioritarios de resolver (su prioridad fue 5 o más). Mientras el prototipo PowerPoint fue evaluado, el prototipo independiente del protocolo de mago de oz estaba siendo implementado en C++. Por tanto, las recomendaciones para superar los problemas de usabilidad en la última columna del Cuadro 5.2, fueron consideradas para el prototipo C++ que se describe más adelante en este capítulo. En el siguiente apartado se determina la confiabilidad de estos resultados.

Cuadro 5.2. Problemas de usabilidad en el prototipo PowerPoint considerados prioritarios de resolver por su número de reportes (R) y severidad (S)

| # | R | S | Problema prototipo PowerPoint | Solución prototipo C++ |
|---|---|---|---|---|
| 1 | 6 | 1 | El tamaño de la tipografía en el enunciado del problema es muy pequeño (véase la parte inferior de la Figura 5.6, p.161), debido al poco espacio disponible y a que no se puede ajustar las dimensiones de las partes de la ventana. | Se usó por defecto tamaño de fuente de 11 ó 12 puntos (dependiendo del sistema operativo) y puede ser ajustado en tiempo de ejecución. El usuario puede cambiar el tamaño y la ubicación de las partes de la ventana. |
| 2 | 6 | 1 | El usuario está forzado a ver la animación completa. No puede controlarla ni detenerla. Aunque existen botones diseñados con este fin y un control de velocidad (sobre el editor de código en la Figura 5.6, p.161), no están implementados. | Se implementaron los botones para iniciar, pausar, adelantar y detener la animación del programa, además de la barra deslizable para controlar la velocidad. |
| 3 | 8 | 0 | Varios controles que no están implementados, como los botones "Misiones", "Cooperar", "Crear" en la Figura 5.2 (p.157) deben aparecer como desactivados o no aparecer del todo. | Todo control que está deshabilitado se muestra como tal y no reacciona a la interacción. Por ejemplo, a los botones desactivados del menú del juego se les aplica una opacidad del 20%. |
| 4 | 4 | 1 | En el tutorial hay un enlace rotulado "[Continuar]" para avanzar en los diálogos con el robot (véase Figura 5.5, p.159), pero no hay uno para regresar. Si el usuario lo presiona accidentalmente dos o más veces, no podrá conocer lo que pasó en medio. | Se sugiere proveer botones de flecha (o enlaces) para adelantar o retroceder en el tutorial. En el prototipo C++ no se implementó un tutorial por restricciones de tiempo. |
| 5 | 4 | 1 | Durante el tutorial, cuando se trata de llamar la atención del usuario hacia un lugar de la pantalla, se le apunta con una flecha animada (véase la flecha apuntado al editor de código en la Figura 5.5, p.159). La flecha | En lugar de una flecha, que el robot del tutorial apunte con su mano. Se puede oscurecer las áreas de la pantalla que no son relevantes. En el prototipo C++ no se implementó un tutorial. |

| # | R | S | Problema prototipo PowerPoint | Solución prototipo C++ |
|---|---|---|---|---|
| | | | compite por la atención con el texto del tutorial y a veces no están sincronizados. | |
| 6 | 4 | 1 | No es claro cuál o cuáles líneas de código se debían modificar. Algunos participantes las descubrieron intuitivamente mientras otros hicieron clics aleatorios sobre el editor de código. | El mecanismo de las líneas editables se debe a una limitación de *MICROSOFT POWERPOINT*. El prototipo C++ usa un editor de código real, por tanto, los usuarios pueden editar cualquier línea de forma natural. |
| 7 | 4 | 1 | No se puede estudiar algo que pasó en una animación previa, porque el prototipo no permite volver a verla sin hacer cambios en el código. | Cuando se presiona el botón de reproducir sin hacer cambios en el código fuente, el prototipo C++ vuelve a realizar la misma animación. |
| 8 | 4 | 1 | Las tres líneas editables del prototipo tienen anchos fijos. Si se escriben cantidades de texto que sobrepasan estos anchos, una parte del texto se hace no visible. | El ancho fijo de las ventanas fue una simplificación para poder hacer prototipado rápido. En el prototipo C++ el tamaño de las ventanas se puede ajustar y el editor permite escribir código fuente arbitrario. |
| 9 | 6 | 0 | La tipografía usada en el tutorial es difícil de leer (véase el diálogo del robot en la Figura 5.5, p.159). Algunos participantes leyeron palabras incorrectamente. La poca separación entre líneas dificulta la lectura. | La tipografía fue escogida por congruencia con la alegoría de los robots. Se debe reemplazar por otra que mantenga la congruencia pero que sea más legible. En el prototipo de C++ no se implementó el tutorial. |
| 10 | 3 | 1 | Debido al poco espacio para el enunciado del problema, el ejemplo de entrada y salida se posicionó a la derecha del enunciado (véase la esquina inferior derecha de la Figura 5.5, p.159). Los participantes tuvieron dificultad para localizar e identificar este ejemplo. | El prototipo C++ permite tener enunciados de longitud arbitraria, ya que los usuarios pueden desplazarse por el texto. Por tanto, los ejemplos de entrada se pueden escribir de forma más natural. |
| 11 | 3 | 1 | El ejemplo de entrada del nivel "1-2" contiene dos arreglos, cada uno precedido por su tamaño, los cuales parecen matrices (véase la esquina inferior derecha de la Figura 5.6, p.161). Hubo dificultad para distinguir el tamaño del resto de datos. | Dado que el prototipo C++ permite tener enunciados de longitud arbitraria, se puede formatear y explicar con más detalle cada parte de la entrada y salida. |
| 12 | 3 | 1 | Cuando el usuario hace un cambio en el código fuente, no necesita volver a ver la animación desde el inicio, como obliga el prototipo. Probablemente querrá poder adelantar la animación a la línea que modificó o le interesa. | Si el usuario establece puntos de parada (*BREAKPOINT*) en el código fuente del prototipo C++, la animación se hará a máxima velocidad y se detendrá cuando alcance un *BREAKPOINT*. Luego el usuario puede solicitar animar una línea a la vez o reanudar la animación. |
| 13 | 3 | 1 | En el nivel "1-2" el segundo vector en el ejemplo de tamaño 5 (véase la esquina inferior derecha de la Figura 5.6, p.161) no coincide con el vector animado de tamaño 9 y causa confusión al usuario. | Dado que en el prototipo de C++ los enunciados no están limitados de longitud, se pueden hacer coincidir los ejemplos con los casos de prueba. |
| 14 | 5 | 0 | Una vez completados los dos niveles, no hay | No se proveyó un mecanismo de salida |

| # | R | S | Problema prototipo PowerPoint | Solución prototipo C++ |
|---|---|---|---|---|
| | | | un botón o mecanismo para salir del prototipo. Los usuarios buscan intuitivamente formas de poder salir. | para evitar que los participantes descubrieran que estaba implementado en *MICROSOFT POWERPOINT* y desviar su atención de la sesión por la curiosidad que podría generar. El prototipo C++ implementa los mecanismos de salida de una aplicación tradicional. |

#### *5.1.3.4 Confiabilidad de la prueba (acuerdo)*

De acuerdo a [Hertzum and Jacobsen 2001], las pruebas de usabilidad presentan un fenómeno al que llamaron **efecto del evaluador**, e indica que la probabilidad de que dos participantes detecten los mismos problemas de usabilidad es muy baja. El nivel de acuerdo entre dos evaluadores puede medirse como la cantidad de problemas de usabilidad encontrados en común por ambos entre la cantidad de problemas totales detectada por ambos. Matemáticamente, el nivel de acuerdo $A_{ij}$ entre un participante $i$ que detectó un conjunto $P_i$ de problemas y un participante $j$ que detectó un conjunto $P_j$ de problemas puede calcularse por la relación:

$$A_{ij} = \frac{|P_i \cap P_j|}{|P_i \cup P_j|}$$

Si en una prueba de usabilidad participan $N$ evaluadores, se puede calcular la tasa de acuerdo entre cada par de evaluadores. En total habrá $N(N-1)/2$ pares de comparaciones. El Cuadro 5.3 muestra el porcentaje de acuerdo de los 66 pares, obtenidos de los 12 evaluadores del prototipo PowerPoint. Las tasas de acuerdo variaron entre 0% a 28%, con más frecuencia de acuerdos bajos como se aprecia en el histograma de la Figura 5.11. Anecdóticamente, las tasas de acuerdo más altas las obtuvieron los primeros tres estudiantes y las más bajas los últimos tres estudiantes (Cuadro 5.3).

El promedio de las tasas de acuerdo de todos los pares de evaluadores se conoce en inglés como *ANY-TWO AGREEMENT*, y ha sido aceptada como la métrica de confiabilidad de una evaluación de usabilidad [Dumas and Fox 2012, p.1235]. En la revisión de literatura de [Hertzum and Jacobsen 2001], los promedios de acuerdo entre todos los pares variaron entre 5% y 65% con una mediana de 13%. En el caso del prototipo PowerPoint de botNeumann++,

el acuerdo general de los 66 pares que evaluaron el prototipo fue del 10.44% ($\sigma$=6.86), el cual es un valor cercano a la mediana reportada en los estudios de referencia.

Cuadro 5.3. Porcentaje de acuerdo entre pares en la evaluación de usabilidad

| | H2 | H3 | P1 | P2 | P3 | E1 | E2 | E3 | E4 | E5 | E6 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| H1 | 11 | 13 | 9 | 13 | 9 | 13 | 18 | 19 | 4 | 6 | 3 |
| H2 | | 9 | 0 | 8 | 5 | 7 | 15 | 13 | 8 | 5 | 6 |
| H3 | | | 10 | 19 | 6 | 14 | 13 | 23 | 4 | 3 | 12 |
| P1 | | | | 19 | 5 | 13 | 17 | 9 | 0 | 12 | 7 |
| P2 | | | | | 19 | 28 | 18 | 3 | 0 | 20 | 5 |
| P3 | | | | | | 17 | 17 | 14 | 10 | 12 | 0 |
| E1 | | | | | | | 27 | 19 | 6 | 13 | 15 |
| E2 | | | | | | | | 24 | 9 | 0 | 6 |
| E3 | | | | | | | | | 7 | 4 | 11 |
| E4 | | | | | | | | | | 0 | 0 |
| E5 | | | | | | | | | | | 7 |

(H=experto en HCI, P=profesor de Programación II, E=estudiante de Programación II)

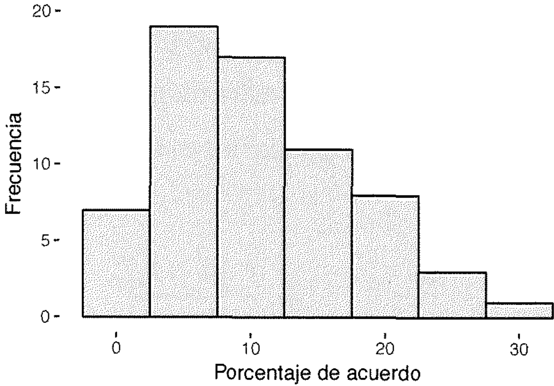



Figura 5.11. Frecuencia de las tasas de acuerdo entre pares en la evaluación de usabilidad

Como se indicó al inicio de esta sección, botNeumann++ es probablemente la primera visualización de programa cuya usabilidad se evalúa con tres poblaciones: estudiantes (usuarios finales), profesores de programación (expertos en el dominio), y profesionales en *HCI* (expertos en usabilidad). Los datos mostraron un marcado efecto evaluador entre todos los participantes. Sin embargo, sería esperable que exista mayor acuerdo entre los participantes de una misma población que entre las distintas poblaciones. El Cuadro 5.4 con las tasas de acuerdo entre grupos de evaluadores, parece reflejar esta hipótesis para los expertos en *HCI* y los profesores de programación cuyas tasas son mayores que el promedio general, pero no para los estudiantes cuyo acuerdo fue mayor con los expertos en *HCI* que entre los estudiantes mismos. Este es un resultado anecdótico que puede inspirar investigación futura.

Cuadro 5.4. Tasas de acuerdo entre grupos de evaluadores

| | H | P | E |
|---|---|---|---|
| H | 11.3 | 8.7 | 10.3 |
| P | | 14.5 | 11.1 |
| E | | | 9.8 |

(H=expertos en HCI, P=profesores de Programación II, E=estudiantes de Programación II)

Para conocer los intereses de cada grupo de evaluadores, se categorizaron los 75 problemas únicos de usabilidad por afinidad temática. Se identificaron 20 categorías temáticas de problemas y se realizó un análisis de correspondencia entre el tipo de participante y los temas que más reportaron. La Figura 5.12 muestra que temáticamente los tres grupos de participantes están evidentemente distanciados. El eje horizontal es el más importante, con 71.4% de la inercia. Por la distribución de temas se interpretó que este eje separa problemas por contenido. Hacia la izquierda se ubican los problemas relacionados con la forma de comunicar los contenidos, como la redacción o cómo hacerla más clara. Notoriamente los expertos en *HCI* son los más interesados en esta dirección. Hacia la derecha de la Figura 5.12 se ubican los problemas relacionados con la forma de presentar información sobre programación en C++. Los profesores y estudiantes se interesan más en esta dirección.

El eje vertical de la Figura 5.12 con 28.6% de la inercia separa problemas por funcionalidad. Hacia arriba se ubican los problemas que limitan el proceso tradicional de programación, como poder usar un editor de código y escribir código C++ arbitrario. Los profesores de programación son los más interesados en esta dirección. Hacia abajo se ubican los problemas que limitan una interacción natural con la interfaz, como los botones que deberían estar habilitados y poder controlar el flujo de diálogo del tutorial. Son los estudiantes los más interesados en esta dirección. Los expertos en *HCI* se ubican en un punto medio en el eje vertical de funcionalidad.

Si se traza en la Figura 5.12 una línea que una a los estudiantes con los expertos *HCI* y otra hacia los profesores, se puede ver que hay varios temas en la vecindad de ambos trayectos, mientras que pocos alrededor del punto de los estudiantes. Este comportamiento refleja las tasas del Cuadro 5.4, con mayor acuerdo entre los estudiantes y otros tipos de participantes que entre los estudiantes mismos. En los alrededores de los expertos en *HCI* y de los profesores se ubican problemas que fueron reportados casi exclusivamente por estas poblaciones. Por tanto, este resultado confirma que para tener una mayor cobertura temática de los problemas de usabilidad, conviene contar con grupos de usuarios de diversa naturaleza.

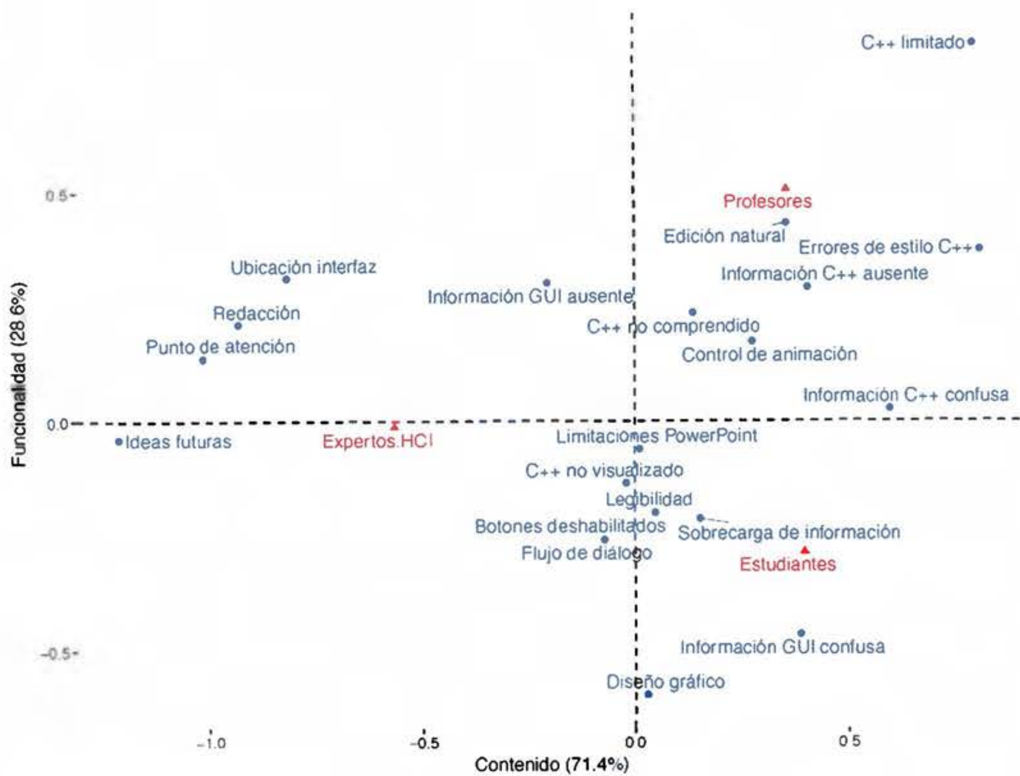



Figura 5.12. Correspondencia entre las categorías de problemas y tipos de participantes

#### *5.1.3.5 Detección de la fuga de memoria*

La fuga de memoria resulta de memoria dinámica reservada por un programa, que deja de utilizarla pero nunca la libera mientras el programa está en ejecución. Escribir programas sin esta mala práctica ha resultado ser especialmente difícil para estudiantes del curso "Programación II" [Hidalgo-Céspedes et al. 2016b, p.14]. Un factor que podría explicar esta dificultad es la influencia del lenguaje de programación *JAVA* en el primer curso de programación, porque tiene la misma sintaxis que C++ para asignar memoria dinámica pero el recolector de basura de *JAVA* exime al programador de liberarla [Hidalgo-Céspedes et al. 2016b, p.14]. Un segundo factor podría deberse a que los programas con fugas de memoria en C++ no generan evidencia del problema, a menos de que el programa corra por un tiempo prolongado y sea detenido al solicitar más memoria dinámica de la permitida por el sistema operativo. Dadas estas condiciones, resulta especialmente interesante determinar si una visualización de programa ayuda o no a los participantes a detectar y corregir fugas de memoria dinámica. Este resultado podría considerarse como un indicio de efectividad de la visualización de programa.

Durante la evaluación de usabilidad se trató cualitativamente, mediante observación, determinar si el prototipo PowerPoint de botNeumann++ ayuda o no al participante a detectar y corregir fugas de memoria. La segunda tarea en el prototipo se diseñó para visualizar dos tipos de fugas de memoria: cuando los arreglos no son eliminados del todo (ruta 8 en el flujo de la Figura 5.4, p.158), y cuando los arreglos se eliminan con el operador `delete` en lugar de `delete[]` (ruta 9 en la Figura 5.4, p.158). Mientras la visualización de una fuga de memoria estaba en proceso, el moderador prestó atención a la reacción del participante. Si el participante expresó que olvidó eliminar la memoria durante la animación, o si al final de la animación el participante buscó la manera de eliminar la memoria, se considera que la animación fue útil para detectar el problema. En caso contrario, si tras la animación el participante expresaba desconocimiento sobre lo ocurrido en la animación, la visualización se consideraba insuficiente para detectar la fuga de memoria.

La observación se realizó con profesores y estudiantes, a quienes no se les explicitó sobre las líneas editables en el prototipo. Los tres profesores y cuatro de los seis estudiantes crearon el arreglo en memoria dinámica (línea 17) y presionaron el botón para reproducir. El prototipo reprodujo la primera animación de fuga de memoria (ruta 8 en la Figura 5.4, p.158). Al terminar la animación buscaron una línea donde pudieran eliminar la memoria. De esta forma, la animación se consideró eficaz para hacer visible la fuga de memoria. Al descubrir la línea 21, eliminaron la memoria con el operador `delete` e iniciaron la segunda animación de fuga de memoria (ruta 9 en la Figura 5.4, p.158). Inmediatamente esta animación llegó a la línea 21, los tres profesores se percataron de la falta de los corchetes (operador `delete[]`). Los cuatro estudiantes detectaron que habían eliminado parcialmente los arreglos gracias a la animación. De esta forma se consideró que la visualización fue eficaz nuevamente. A diferencia de los profesores, los cuatro estudiantes no sabían cómo corregir la segunda fuga de memoria. Tres intentaron usar un ciclo para eliminar cada uno de los elementos del arreglo, pero se detuvieron por contar con sólo una línea editable en lugar de dos para eliminar la memoria. Los cuatro estudiantes finalmente llegaron al operador `delete[]` a través de búsquedas en la web con apoyo de un navegador. Durante la entrevista posterior a la sesión, algunos estudiantes expresaron que durante el curso de "Programación II" habían trabajado con arreglos de la biblioteca estándar (`std::vector`) y no con los del lenguaje de programación (punteros).

Los dos restantes estudiantes, E1 y E5, fueron casos especiales. Ambos descubrieron la línea 21 antes de visualizar cualquiera de las fugas de memoria. El estudiante E1 descubrió la línea 21 porque necesitaba eliminar la memoria dinámica, y lo hizo con el operador `delete[]`. Sin embargo, realizó 19 intentos de crear el arreglo en memoria dinámica con errores sintácticos, por lo que requirió ayuda del moderador. En el intento 9 borró el texto que había escrito en ambas líneas, por lo que eventualmente activó la primera animación de fuga de memoria, pero no la comprendió. Al finalizar la animación, escribió de nuevo en la línea 21 el operador `delete[]` correctamente con lo que finalizó la tarea. Posterior a la entrevista, este estudiante expresó que siempre que solicitaba memoria dinámica, la liberaba con el operador `delete[]` porque "si le ponía los corchetes le daban puntos en los exámenes y si los quitaba perdía puntos", pero no conocía la diferencia entre ambos. Por consiguiente, la visualización no se consideró eficaz para este participante.

El estudiante E5 descubrió la línea 21 de manera accidental y expresó la necesidad de que "tenía que escribir algo ahí, pero no sabía qué". En el segundo intento de crear el arreglo en memoria dinámica, escribió el operador `delete` en la línea 21. En el quinto intento activó la segunda animación de fuga de memoria, la cual explicó correctamente. Al finalizar la animación intentó eliminar cada elemento del arreglo con un ciclo, que posteriormente corrigió al operador `delete[]` con ayuda del navegador web. Por tanto, la visualización fue eficaz para este participante.

En síntesis, la primera animación de fuga de memoria (ruta 8 en la Figura 5.4, p.158) fue visualizada ocho veces cuando la memoria dinámica no fue eliminada del todo, y en siete de ellas ayudó a los participantes a detectar el problema, es decir, tuvo una eficacia del $7/8 = 87.5\%$. La segunda animación (ruta 9 en la Figura 5.4, p.158) se activó también 8 veces cuando el arreglo se eliminó con el operador `delete`, y ayudó a los ocho participantes a detectar el problema, es decir, su eficacia fue del 100%. La eficacia general de la visualización puede estimarse en $15/16 = 93.75\%$.

La segunda tarea del prototipo se diseñó para ayudar a la detección de errores y no tanto a la corrección de los mismos. Los resultados muestran que la visualización es evidentemente insuficiente para corregir la fuga de memoria. Por ejemplo, los participantes con conocimiento sobre el operador `delete[]` (profesores) corrigieron el problema de inmediato, mientras que los estudiantes requirieron el uso de herramientas adicionales (un navegador web). Por tanto,

el prototipo PowerPoint se considera una herramienta eficaz (93.75%) para detectar el problema de la fuga de memoria, pero limitado a la estrategia de prueba y error para corregirla. Este hecho llama la atención hacia investigación futura en apoyar otras estrategias de resolución de problemas en las visualizaciones de programa.

### *5.1.3.6 Opinión sobre la herramienta*

Tras finalizar las tareas se realizó una entrevista a cada estudiante siguiendo el cuestionario de 26 preguntas de la Figura 5.13 para conocer aspectos cualitativos de la herramienta. La cantidad de preguntas que se formuló varió en función de la disponibilidad de tiempo del estudiante. A continuación se presenta cada pregunta y un resumen de los resultados.

1. Para los siguientes bloques de código, marque aquellos que producen una fuga de memoria. Justifique.

(a)
```
{
    double value,
    double* ptr = &value,
    ptr = nullptr,
}
```

(b)
```
{
    double* ptr = new double,
    ptr = nullptr,
    delete ptr,
}
```

(c)
```
{
    double* ptr = nullptr,
    delete ptr,
}
```

(d)
```
{
    double* ptr = new double[4];
    delete ptr,
    ptr = nullptr,
}
```

(e)
```
{
    double arr[4];
    double* ptr = arr,
    ptr = nullptr,
}
```

(f)
```
{
    double* ptr = new double[4];
    delete [] ptr,
    ptr = nullptr,
}
```

2. ¿Cómo definiría en sus propias palabras qué es una fuga de memoria?
3. ¿Siente que la visualización le ayudó a resolver los ejercicios mejor?
4. ¿Cuál fue su reacción general?
5. ¿Qué fue lo que más le gustó?
6. ¿Siente que esto es un juego o no es un juego?
7. ¿Qué se acuerda de la historia?
8. ¿Qué le pareció la historia del juego? ¿Hace más interesante el reto o es superflua?
9. ¿Si encontrara esta aplicación en internet jugaría con ella?
10. ¿Si corriera su código, la usaría?
11. ¿Qué fue lo menos interesante? ¿qué fue lo más aburrido?
12. ¿Siente que los textos fueron muy extensos? ¿fueron útiles? ¿siente que era mucha información que no podía mantener en la mente?
13. ¿Fueron las animaciones muy lentas, muy rápidas? ¿era la velocidad adecuada para comprender lo que estaba pasando?
14. ¿Qué no fue claro?
15. ¿Siente que entiende mejor cómo trabaja la máquina luego de haber visto esta animación?
16. ¿Las clases "Progra2" mejorarían con esto?
17. ¿Qué fue lo que más le retó o intrigó?
18. ¿Cuál fue la emoción que sintió cuando vio un desbordamiento?
19. ¿Siente que el nivel de dificultad de los ejercicios está acorde a lo aprendido en el curso?
20. ¿Siente que las preguntas que realiza el robot fueron útiles?
21. ¿Siente que el cambio de la dificultad entre el nivel 1-1 a 1-2 fue muy brusco?
22. ¿Siente que dentro de la computadora hay tubos y robots? ¿Siente que esto podría confundir?
23. ¿Hubo algo nuevo para usted o que le sorprendiera?
24. ¿Pudo hacer todo lo que quería o hubo algo que hizo falta?
25. ¿Recomendaciones para mejorar la experiencia?
26. ¿Algo más que quiera agregar?

Figura 5.13. Cuestionario posterior a la evaluación de usabilidad de botNeumann++

1. *(Identificar bloques de código que producen una fuga de memoria y justificar).* Esta pregunta se realizó con el fin de evaluar informalmente si la visualización tuvo un efecto en el reconocimiento de fugas de memoria a partir de código fuente. Se asignó una calificación subjetiva entre 0 y 100 a cada respuesta, de tal forma que 0 indica una elección y justificación incorrectas, y 100 indica que tanto la elección como la justificación fueron correctas. El gráfico de cajas de la Figura 5.14 muestra el rendimiento de los seis estudiantes en cada uno de los seis bloques de código. Los únicos dos bloques con fugas de memoria fueron el B y el D. Aunque los rendimientos son mayoritariamente positivos, es notorio que los estudiantes dictaminaron con mayor acierto los bloques D y F, que son precisamente réplicas del código visualizado por el prototipo PowerPoint de botNeumann++.

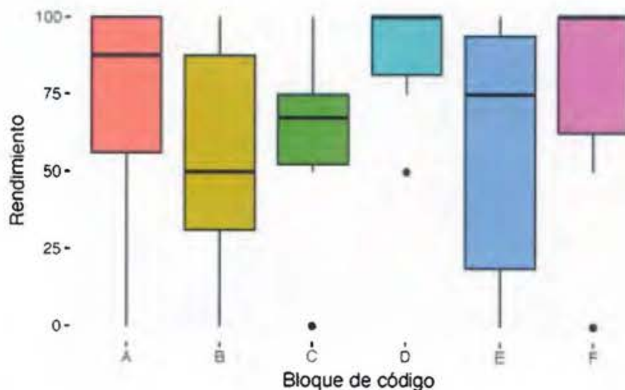



Figura 5.14. Rendimiento de los estudiantes detectando fugas de memoria en bloques de código

2. *¿Cómo definiría en sus propias palabras qué es una fuga de memoria?* Esta pregunta se formuló para tener un indicio si tras el proceso de análisis del concepto de fuga de memoria visualizado, los estudiantes podían sintetizar (generalizar) el concepto en función de otros. Cuatro de los cinco estudiantes que respondieron elaboraron definiciones aceptables en función de otros conceptos como puntero, asignación y liberación de memoria dinámica.
3. *¿Siente que la visualización le ayudó a resolver los ejercicios mejor?* Cuatro de los cinco estudiantes que respondieron la pregunta expresaron que la visualización les ayudó a responder la pregunta 1. Uno incluso declaró que sin la herramienta "habría tenido mala la pregunta". Un estudiante dijo que la herramienta no fue muy útil porque la superposición de las invocaciones de funciones es confusa.

4. *¿Cuál fue su reacción general?* Cinco de seis estudiantes expresaron reacciones positivas, indicando aspectos que les gustaron, como la utilidad, la belleza, limpieza, el formato de juego, y el reto, entre otras. Además ofrecieron ideas para mejorar como hacer las animaciones más rápidas y menos texto. Un estudiante expresó que era lento y tendría que ver la versión final con un control de velocidad implementado.
5. *¿Qué fue lo que más le gustó?* Los seis estudiantes indicaron que la animación gráfica de lo que pasa en la computadora es lo que más les gustó. Cuatro indicaron que habría sido muy útil ver estas animaciones cuando estaban aprendiendo los conceptos, dado que habría sido más fácil comprender y les habría tomado menos tiempo. Como aspectos negativos resurgió la lentitud y los textos extensos.
6. *¿Siente que esto es un juego o no es un juego?* Un estudiante sitió que era un juego, otro que no, y los otros cuatro sintieron que era entre una aplicación académica y un juego serio o educativo. La percepción de la mayoría se acerca a lo que realmente es, una visualización de programa (una aplicación) con ludificación.
7. *¿Qué se acuerda de la historia?* Esta pregunta se realizó dado que la narrativa teóricamente ayuda en el recuerdo de información. Los cuatro estudiantes a quienes se les realizó esta pregunta, recordaron el objetivo del jugador dentro del contexto de la historia: que debían ayudar a reprogramar código que se había perdido por un ataque de hackers. Los participantes recordaron varios detalles secundarios con niveles fluctuantes de precisión.
8. *¿Qué le pareció la historia del juego? ¿Hace más interesante el reto o es superflua?* Esta pregunta tiene el objetivo de confirmar si la historia del juego influye en el interés en la tarea. Sin una historia, el objetivo de "reparar código para ayudar a una sociedad", se reduce a una tarea tradicional similar a "corrija el siguiente código para que sume dos números". Tres de los estudiantes consideraron que la historia hacía más interesante el objetivo de reparar el código. Los otros tres se enfocaron en que al ser mucho texto se perdía el interés y sugirieron en reducir la cantidad de texto o expresar la historia con caricaturas.
9. *¿Si encontrara esta aplicación en internet jugaría con ella?* Esta pregunta busca un indicio de impacto. Un estudiante respondió afirmativamente, los restantes tres estudiantes que la respondieron condicionaron su potencial uso. Expresaron que utilizarían la aplicación si estuvieran en la fase de aprendizaje de C++ y sólo si la herramienta estuviera integrada como parte del curso de programación.

10. *¿Si corriera su código, la usaría?* Esta pregunta busca indicios de impacto como depurador visual. Los cuatro estudiantes que la respondieron lo hicieron afirmativamente. Dos estudiantes indicaron que así sería muy útil para encontrar errores difíciles en programas, aunque uno dejó claro que para la etapa de estudiante y no para la de desarrollador profesional.
11. *¿Qué fue lo menos interesante? ¿qué fue lo más aburrido?* Esta pregunta busca identificar los problemas de usabilidad que más hayan afectado el interés en la tarea. Los textos extensos del tutorial y la lentitud de las animaciones fueron los dos problemas más señalados. Otros problemas fueron la falta de rótulos sobre algunos elementos de la interfaz (como la basura en la memoria) y más acciones por parte del robot {hilo de ejecución}. Dos de los cinco estudiantes expresaron que lo más aburrido fue cuando "me quedé pegado y no sabía que hacer" por errores sintácticos.
12. *¿Siente que los textos fueron muy extensos? ¿fueron útiles? ¿siente que era mucha información que no podía mantener en la mente?* En general, la mayoría de estudiantes habían respondido esta pregunta desde antes. Cuatro estudiantes agregaron de que la información se podía mantener en la mente, pero tenían que releer por lo largo de los textos y el pequeño tamaño de la tipografía.
13. *¿Fueron las animaciones muy lentas, muy rápidas? ¿era la velocidad adecuada para comprender lo que estaba pasando?* De igual forma esta pregunta fue respondida con anterioridad. Tres participantes recalcaron la importancia del control de velocidad y la posibilidad de saltar líneas de código durante la animación.
14. *¿Qué no fue claro?* Los cinco estudiantes que contestaron la pregunta indicaron que no fue claro: el caso de prueba que difería del ejemplo al usado en la animación, el mensaje de "consulte con el profesor" que activaba el protocolo de mago de oz, los botones de control de la visualización no se distinguen del fondo, y no llamar la atención cuando se elimina algo de la memoria dinámica.
15. *¿Siente que entiende mejor cómo trabaja la máquina luego de haber visto esta animación?* Los cinco estudiantes que respondieron esta pregunta lo hicieron afirmativamente. Uno agregó que sería más útil cuando se está aprendiendo C++. El estudiante E4 favoreció la alegoría concreta: "a usted le pueden explicar abstractamente como funciona, pero... al verlo como algo más físico que uno puede como verlo mejor, es más fácil de comprender".

16. *¿Las clases "Progra2" mejorarían con esto?* Esta pregunta está ideada a obtener un segundo indicio de impacto y fue respondida por cinco estudiantes. Tres estudiantes consideraron que las clases mejorarían porque ayudan a comprender la memoria más fácilmente. Un estudiante indicó que al limitar los recursos (por ejemplo, con poca memoria) ayuda a aprender a programar bien. Un estudiante indicó que sería más útil en "Programación I" porque es cuando se aprenden los conceptos sobre memoria.
17. *¿Qué fue lo que más le retó o intrigó?* Cinco de los seis estudiantes coincidieron en la dificultad para crear el arreglo en memoria dinámica, dado que habían olvidado la sintaxis de C++. Un estudiante sugirió tener un botón de ayuda. Por otro lado, el estudiante E5 indicó que lo más retador fue que "no podía editar todo el código".
18. *¿Cuál fue la emoción que sintió cuando vio un desbordamiento?* De acuerdo a la teoría sociocultural, las emociones ayudan o inhiben la asociación de las nociones. Dos de cinco estudiantes expresaron sentir contradicción ("¿Por qué pasó eso si yo creía que lo estaba haciendo bien?"), uno expresó confusión (creyó que era un error gráfico hasta que comprendió que era realmente un desbordamiento), uno sintió lástima (de que falló), y el último decepción ("Diay lo hice mal").
19. *¿Siente que el nivel de dificultad de los ejercicios está acorde a lo aprendido en el curso?* Los seis estudiantes indicaron que los ejercicios estuvieron acordes al nivel de dificultad del curso que tomaron. Uno sugirió para la segunda tarea usar memoria dinámica pero no arreglos.
20. *¿Siente que las preguntas que realiza el robot fueron útiles?* Las preguntas se realizaron con el fin de incentivar el estado mental activo. Por ejemplo, tras un fallo de segmento el robot pregunta por qué ocurrió y cómo se puede corregir. Las respuestas de los participantes a estas preguntas se registraban en los videos. Los cinco estudiantes que respondieron esta pregunta las consideraron útiles, y uno indició que para responderlas debía pensar, o volver a leer para comprender. Dado que responder las preguntas no es obligatorio para avanzar, tres estudiantes dieron a entender que si no estuviera presente el moderador, ni siquiera habrían prestado atención a las preguntas.
21. *¿Siente que el cambio de la dificultad entre el nivel 1-1 a 1-2 fue muy brusco?* Las cinco respuestas brindadas coinciden en que no fue un cambio brusco, pero podría serlo para estudiantes principiantes en C++.

22. *¿Siente que dentro de la computadora hay tubos y robots? ¿Siente que esto podría confundir?* Ninguno de los cinco estudiantes que respondieron la pregunta consideró que pudiera confundir. Para ellos era evidente que la computadora es una máquina de circuitos, en especial porque el curso de "Circuitos digitales" es simultáneo en el plan de carrera a "Programación II". Tres estudiantes indicaron que puede ayudar a comprender mejor cómo trabaja la máquina, por ejemplo: "A menos de que el jugador tenga [menos] de 10 años, yo estoy seguro que va a entender que es una metáfora [...] el robot es el *ALU* y los tubos son *STREAMS*..."
23. *¿Hubo algo nuevo para usted o que le sorprendiera?* Tres de cinco estudiantes indicaron que era nuevo para ellos crear arreglos en memoria dinámica. Un participante indicó con inseguridad de que el juego en sí. El último participante indicó que nada era nuevo para él.
24. *¿Pudo hacer todo lo que quería o hubo algo que hizo falta?* Un estudiante indicó que no hubo algo que hizo falta, los restantes cuatro estudiantes indicaron: poderse devolver en el tutorial, ver cómo se representa un registro (clase o estructura), poder editar código arbitrariamente, y poder adelantar la animación a una línea particular.
25. *¿Recomendaciones para mejorar la experiencia?* Tres de cuatro estudiantes indicaron que ya habían expresado sus recomendaciones y el cuarto volvió a sugerir la reducción de los textos largos.
26. *¿Algo más que quiera agregar?* Ningún estudiante ofreció más detalles.

### 5.1.4 Discusión de la evaluación de usabilidad

La prueba de usabilidad se realizó con un propósito de efectividad y cuatro propósitos de usabilidad de acuerdo a la nomenclatura de [Dumas and Fox 2012]: explorar la usabilidad de conceptos iniciales de diseño, diagnosticar problemas de usabilidad, corregir problemas de usabilidad, y validar la usabilidad. Cada uno de los cinco propósitos se discute en los siguientes apartados.

#### *5.1.4.1 Efectividad de la visualización*

El propósito de efectividad fue evaluar si el prototipo PowerPoint de botNeumann++ ayuda a los participantes a detectar y corregir fugas de memoria. La fuga de memoria ha sido uno de los problemas más difíciles de abordar en el aprendizaje de la programación con C++ en la

Escuela de Ciencias de la Computación e Informática, potencialmente debido a la influencia de *JAVA* en el curso de programación previo y la ausencia de signos visibles en los programas para poder detectarlas. Por tanto, el prototipo se diseñó con dos tareas, una de familiarización de la herramienta y otra para detectar y corregir fugas de memoria.

Mediante observación se encontró que los dos tipos de animaciones de fuga de memoria implementados en el prototipo ayudaron a los seis estudiantes y tres profesores en el 93.75% de las veces a detectar una fuga de memoria cuando estaba presente en el código. Este es un resultado muy satisfactorio de eficacia para una visualización de programa, no sólo por su aporte al contexto de uno de los problemas más difíciles de la educación de la programación en la Escuela de Ciencias de la Computación e Informática, sino también porque responde al *objetivo ingenieril* de esta tesis planteado en la página 8, de mejorar las visualizaciones de programa con alegorías visuales y ludificación para ayudar a estudiantes a comprender la máquina nocional de un lenguaje de programación.

Una limitación del prototipo PowerPoint es que no puede compararse contra otras herramientas existentes, dado que ninguna visualización de programa de C/C++ visualiza fugas de memoria. Esta limitación sumada a la dependencia del protocolo de mago de oz, hace que el prototipo PowerPoint sea insuficiente para responder el objetivo científico de esta investigación, planteado en la página 9 que pretende comparar las visualizaciones lúdicas de programa contra herramientas existentes. Por estos motivos, se desarrolló un segundo prototipo presentado en la próxima sección.

Por otra parte, aunque el prototipo PowerPoint fue eficaz en la *detección* de fugas de memoria, no se diseñó para ayudar en la *corrección* de las mismas. Los estudiantes debieron recurrir a una herramienta externa (un navegador web) para obtener información sobre la eliminación de arreglos en memoria dinámica en C++. Este hecho señala una puerta abierta para investigación futura.

#### *5.1.4.2 Usabilidad de conceptos iniciales de diseño*

El prototipo PowerPoint es el primer prototipo de una visualización lúdico-alegórica de programa. Por tanto, implementa conceptos iniciales de diseño sobre alegorías visuales y ludificación con los que se conocieron los primeros resultados de usabilidad. Estos resultados sirvieron para ajustar el segundo prototipo en C++ y pueden ser útiles para futuras

visualizaciones lúdicas de programa. Se discuten a continuación los resultados generales de usabilidad sobre estos conceptos.

El prototipo se evaluó con tres poblaciones: estudiantes en calidad de usuarios finales, profesores de programación como expertos en el dominio, y profesionales en *HCI* como expertos en usabilidad. Debido a sistemas de conceptos inestables sobre la sintaxis de creación de arreglos en C++, los usuarios finales introdujeron una cantidad sustancial de errores de usabilidad que afectó la duración y completitud de la segunda tarea. Estos errores evidenciaron la dependencia del prototipo por las nociones previas de los participantes y confiere importancia a ofrecer ayuda sobre errores, especialmente sintácticos, a los usuarios finales de botNeumann++, con el fin de mejorar la experiencia de usuario e incrementar la eficiencia de la herramienta. Sin embargo, este es un resultado anecdótico que merece investigación futura. Elementos lúdicos como recompensa y realimentación podrían satisfacer esta necesidad, al permitir intercambiar "puntos" acumulados por "pistas" sobre los errores sintácticos. Sería muy interesante determinar si estos elementos lúdicos incrementan la eficiencia de la tarea en el corto y largo plazo (en inglés, *LEARNABILITY*) y mejoran la experiencia de usuario.

El prototipo PowerPoint se diseñó con tareas de comprensión y resolución de problemas en lugar de enseñanza de conceptos de programación. Aunque mediante observación se encontró que la mayoría de los participantes explicaron correctamente los conceptos de programación representados por la alegoría visual, no se controló rigurosamente si estas interpretaciones se hicieron sobre la animación o a partir del código fuente en C++. Este hecho deja abierto un espacio para investigación futura que podría explorar la interpretación de la alegoría sin presencia de código fuente, o los niveles de atención a los elementos alegóricos con tecnología de seguimiento de ojos (en inglés, *EYE TRACKING*).

En la entrevista posterior a la evaluación, se le pidió a los estudiantes reconocer fugas de memoria en seis bloques de código. Se encontró que los estudiantes tuvieron mayor precisión y acierto en los dos bloques con códigos similares a los visualizados por el prototipo. Este es un segundo resultado positivo de la efectividad de la visualización de programa botNeumann++. Por otra parte, los resultados menos positivos en los otros cuatro bloques de código confirman la teoría de aprendizaje sociocultural de que los estudiantes pueden aplicar las nociones asimiladas a contextos similares, pero requieren la aplicación reiterada a

diversos contextos para poder abstraer el principio fundamental del concepto de los contextos. Esta posición de la teoría es una invitación a extender el prototipo y evaluar su efectividad en investigación futura.

#### *5.1.4.3 Validación de la usabilidad*

El tercer propósito fue validar la usabilidad. Los participantes identificaron 169 problemas de usabilidad, 60% de ellos fueron de severidad baja, 33% de severidad media y ninguno fue considerado de severidad alta. Es decir, ninguno de los problemas identificados impedía a los participantes completar las dos tareas planteadas en la evaluación. Por tanto el prototipo se consideró usable y esta pudo ser una consecuencia positiva de la evaluación piloto.

La prueba se consideró confiable con un porcentaje general de acuerdo de 10.44%. Se encontró que hubo más acuerdo entre participantes del mismo grupo, a excepción de los estudiantes.

Los grupos de participantes se interesan en aspectos diferentes del software. Los expertos en *HCI* reportan más errores por la forma de comunicar información general, mientras que los estudiantes y profesores reportan más errores sobre información relacionada con programación en C++. Los profesores se diferencian de los estudiantes en reportar más problemas que limitan la edición natural del código, mientras que los estudiantes se inclinan más hacia aspectos generales de la interfaz. Este resultado apoya la teoría del efecto evaluador y sugiere evaluar las herramientas con diversidad de poblaciones para tener una mayor cobertura temática de los problemas de usabilidad.

#### *5.1.4.4 Diagnóstico y corrección de problemas de usabilidad*

El cuarto propósito de la prueba fue diagnosticar y corregir problemas de usabilidad. Los 75 problemas únicos de usabilidad fueron valorados por prioridad en proporción a su severidad y el número de participantes que los reportaron. Los problemas más prioritarios refirieron a texto ilegible por poco espacio en la pantalla, la imposibilidad de controlar la animación y su velocidad, controlar el tutorial, editar código naturalmente y hacer más claros los enunciados de los problemas. En la entrevista a los estudiantes se volvieron a mencionar algunos de estos problemas, y se les sumó los largos textos en el tutorial. Estos problemas fueron considerados para el prototipo C++ que se documenta en la siguiente sección.

## 5.2 Prototipo 2: C++

Para poder evaluar experimentalmente el diseño botNeumann++ se requirió un prototipo que fuese capaz de visualizar código C/C++ arbitrario, sin dependencia de un protocolo de mago de oz y que adicionalmente superara los problemas de usabilidad identificados en el prototipo PowerPoint. Este segundo prototipo se desarrolló en C++, y en este documento se le denomina **prototipo C++**. La implementación de este software enfrentó severas dificultades, lo que generó varias lecciones aprendidas, las cuales se discuten a continuación.

### 5.2.1 Implementación y lecciones aprendidas

Como se aprecia en la línea del tiempo de la Figura 5.15, la implementación del prototipo en C++ inició el 1º de octubre de 2014 y transcurrieron dos años y nueve meses de desarrollo hasta que alcanzara un nivel de funcionalidad apto para ser evaluado mediante un experimento controlado. En este período se enfrentaron varias dificultades procurando producir la cantidad de detalles que las animaciones de botNeumann++ requieren para representar alegóricamente la máquina nocional de C/C++. Esta subsección presenta un vistazo general del prototipo por razones de espacio. Los detalles técnicos de implementación y las lecciones aprendidas se recogen en el Anexo B .

Figura 5.15. Historial de cambios en el desarrollo de prototipo C++ (adaptado de *GITHUB*)

El prototipo C++ fue implementado en módulos que extendieron la arquitectura sugerida en [Lanza 2003] como se aprecia en la Figura 5.16. Un resumen de cada módulo se presenta los siguientes apartados.

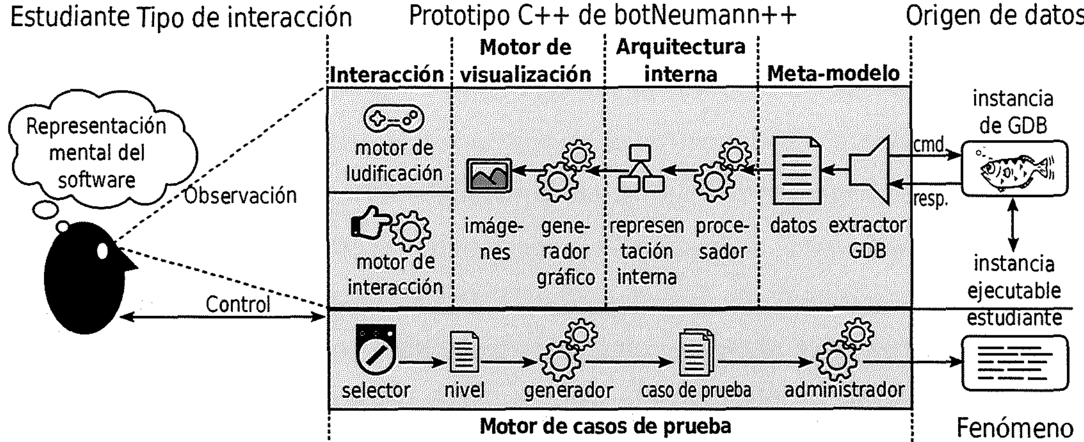



Figura 5.16. Arquitectura del prototipo C++ de botNeumann++

#### *5.2.1.1 El motor de visualización*

La implementación del prototipo C++ inició con el motor de visualización como puede verse en la parte superior de la Figura 5.15, al cual se le dio el nombre de `GameE` (por *GAME ENGINE*). Este módulo es el que se encarga de producir las imágenes que visualiza el usuario [Lanza 2003]. Se probaron varias tecnologías y se decidió usar una biblioteca de uso general e implementar los algoritmos gráficos faltantes, como transiciones entre escenas y posicionamiento de los elementos gráficos.

Una fortaleza de `GameE` es su administrador de posicionamiento (en inglés, *LAYOUT MANAGER*), que es capaz de ubicar elementos gráficos vectoriales en posiciones y proporciones en punto flotante. Gracias a esta capacidad el usuario puede re-dimensionar la visualización a su gusto, sin perder calidad de imagen ni la ubicación correcta de los elementos gráficos, incluso aunque se encuentren en movimiento.

#### *5.2.1.2 El motor de interacción*

El motor de interacción se encarga de permitir al usuario manipular directamente la visualización [Lanza 2003]. Se implementó un motor de interacción muy básico, que permite accionar los elementos gráficos con un dispositivo de apuntar (por ejemplo, un ratón) o un dispositivo táctil (una pantalla sensible). Esta decisión se debe a que la mayoría de la interacción con el usuario ocurre en componentes cuyos mecanismos de interacción ya están provistos por la biblioteca gráfica de uso general.

#### *5.2.1.3 El motor de ludificación*

El motor de ludificación es propio de botNeumann++ y se encarga de implementar los elementos lúdicos, como el menú de juego y niveles. La Figura 5.17 muestra la máquina de estados general del motor de ludificación implementado en el prototipo C++. Cuando la aplicación inicia se presenta la *pantalla de menú* donde el estudiante puede crear o escoger un usuario. Al accionar un modo de juego, como "Entrenar" se presenta la *pantalla de selección de nivel* (mapa de niveles). Al escoger un nivel se presenta la *pantalla de nivel*, donde el usuario puede solucionar un problema planteado y visualizar sus programas. El usuario puede regresar entre estas pantallas accionando un botón con forma de flecha hacia atrás, como es habitual en videojuegos casuales. Los detalles técnicos de implementación de estos estados se encuentran en el anexo 2 al final de esta tesis.

Figura 5.17. Máquina de estados general del prototipo C++

La pantalla de menú permite crear o cambiar el usuario al accionar el texto "Jugador" en la Figura 5.18. Un perfil de usuario es necesario para poder usar el prototipo, ya que permite a la aplicación registrar el progreso del usuario, y los eventos que genere en las bitácoras. Cuando un usuario es seleccionado, el botón de "Entrenar" se habilita y permite al jugador pasar al mapa de niveles.

Figura 5.18. Pantalla de menú en el prototipo C++

La pantalla de selección de nivel, también llamada mapa de niveles, muestra los niveles que el investigador establezca en tiempo de compilación, por ejemplo los cuatro niveles de la Figura 5.19 fueron los usados en el experimento al final de este capítulo. Cuando el usuario activa uno de los niveles, por ejemplo, el rotulado "1-1" se presenta la pantalla de nivel.

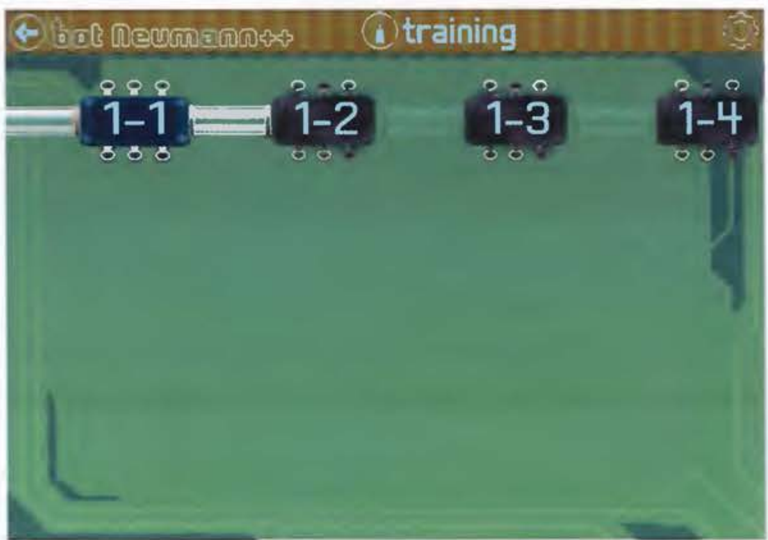



Figura 5.19. Pantalla de selección del nivel en el prototipo C++

Un nivel en botNeumann++ es un ejercicio de programación que el investigador diseña en un archivo *XML*. La pantalla de nivel carga el ejercicio desde el archivo *XML*, y permite al usuario resolverlo y visualizar su solución. La Figura 5.20 muestra la pantalla de nivel para un ejercicio de conversión de temperaturas (rotulado "1-1") en el experimento que se reporta al final de este capítulo.

La pantalla de nivel se ajusta a varias preferencias del diseñador indicadas en el archivo *XML* para el ejercicio. Por ejemplo, para el ejercicio de conversión de temperaturas en la Figura 5.20, el diseñador indicó no utilizar memoria dinámica por lo que las puertas de la bodega están cerradas ①, solicitó dos núcleos del procesador por lo que el prototipo visualiza dos estaciones de trabajo para robots ② ③, y especificó un tamaño de memoria total de 512 que el prototipo distribuye automáticamente entre los tres segmentos, manteniendo las proporciones visuales de los aposentos en múltiplos de ocho como se puede ver en el segmento de datos ④.

El diseñador del ejercicio debe especificar un enunciado del problema en al menos un idioma. El prototipo C++ muestra el enunciado en la pestaña "Descripción" rotulado ⑤ en la Figura

5.20. El usuario estudia este enunciado para comprender el problema que debe resolver. Dado que en el prototipo PowerPoint la legibilidad de estos enunciados fue uno de los problemas de usabilidad más relevantes de corregir, el prototipo C++ permite re-dimensionar y reubicar las partes de la interfaz. El rótulo ① de la Figura 5.21 muestra la descripción del problema reubicado en la esquina superior derecha con un tamaño de tipografía mayor. Puede notarse cómo el área de visualización se reajusta para ocupar el espacio libre, respecto a la Figura 5.20.

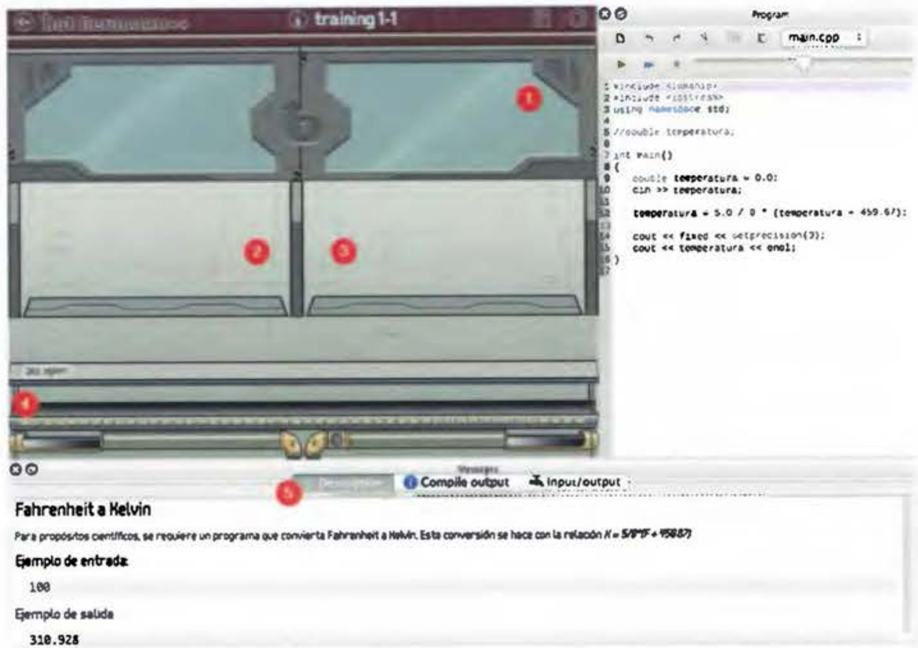


Figura 5.20. Pantalla de nivel con el ejercicio de conversión de temperaturas

El diseñador puede proveer cero o más códigos iniciales para efectos de investigación, como programas en C/C++ con errores. El prototipo C++ cargará uno al azar en el editor de código fuente (segmento de código) rotulado con ② en la Figura 5.21, donde el usuario podrá modificarlo. En los números de línea del editor, el usuario puede establecer puntos de parada (*BREAKPOINTS*). Encima del editor de código se ubican tres botones similares a los de un reproductor multimedia ⑤ que permiten al usuario probar y controlar la animación de su solución.

Cuando el usuario acciona el botón de reproducir (►), el prototipo ejecuta la solución del usuario contra varios casos de prueba e inicia la animación de uno de ellos. El usuario puede controlar la velocidad de la animación con la barra deslizable que es parte de los controles,

ubicada a la derecha de ⑤ en la Figura 5.21. La animación se pausa cuando el usuario presiona el botón correspondiente (❚❚) o cuando se alcanza un punto de parada (*BREAKPOINT*). En el estado de pausa, el usuario puede animar la ejecución de una instrucción a la vez (⏭) o reanudar la animación (▶). El usuario puede detener la animación (■) que regresa el prototipo al estado de edición, aunque el usuario puede corregir su código fuente mientras la animación está en progreso.

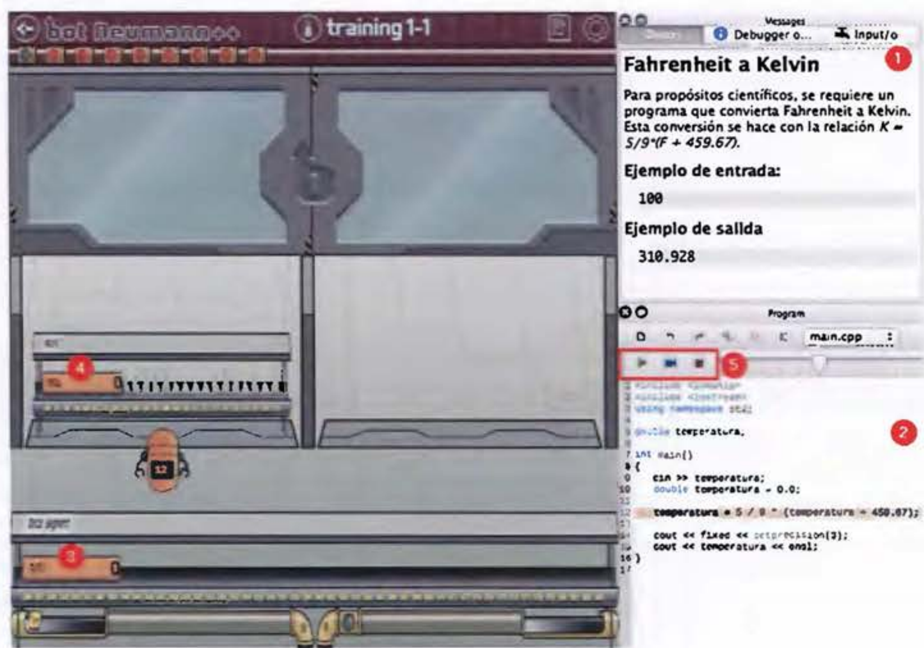


Figura 5.21. Reubicación de las ventanas en el prototipo C++

### *5.2.1.4 El motor de casos de prueba*

El motor de casos de prueba es propio de botNeumann++ y se encarga de ejecutar instancias del programa del usuario contra casos de prueba (Figura 5.16). Cuando el usuario acciona el botón de reproducir (▶), el prototipo compila su código fuente en C/C++ para generar un ejecutable. Si el programa del usuario tiene errores, el prototipo detiene el proceso y presenta los diagnósticos reportados por el compilador.

Simultáneamente el prototipo extrae casos de prueba literales a partir del archivo *XML*, y si el diseñador provee programas generadores de casos de prueba en C/C++, los compila y los corre para generar casos de prueba adicionales. Un caso de prueba consta de entradas que recibirá el programa del usuario y las salidas correctas que debe producir. El programa del usuario recibirá las entradas y las salidas que produzca serán comparadas con las esperadas.

Un selector (Figura 5.22) indica al usuario con colores de semáforo los casos de prueba en que su solución coincide con la salida esperada (verde), difiere (rojo), o está en proceso (gris). Si el usuario acciona un caso de prueba en el selector, el prototipo C++ anima la ejecución del programa con el caso seleccionado.

Figura 5.22. Selector con trece casos de prueba

#### *5.2.1.5 El meta-modelo*

El meta-modelo es el módulo que se encarga extraer y estructurar los datos que serán visualizados (Figura 5.16) [Lanza 2003]. La extracción de datos fue el proceso que generó más contratiempos en el desarrollo del prototipo C++ (Figura 5.15) por la falta de una infraestructura de depuración para C/C++. Se optó por utilizar el depurador *GDB*[18] para extraer información del programa del estudiante en tiempo de ejecución. Sin embargo, *GDB* impone serias limitaciones a una visualización de programa. Por ejemplo, la interfaz de comunicación de los programas con *GDB* es limitada y requiere un intérprete, por lo que se tuvo que adaptar uno existente. *GDB* facilita estudiar los hilos de ejecución del programa del estudiante y sus pilas de invocaciones de funciones, pero no los demás segmentos o archivos abiertos. Por tanto, se tuvo que crear una arquitectura interna voluminosa para suplir estos faltantes.

#### *5.2.1.6 La arquitectura interna*

La arquitectura interna es el módulo que se encarga de procesar los datos extraídos a través del meta-modelo y transformarlos en una representación interna que sea fácil de visualizar (Figura 5.16) [Lanza 2003]. Debido a que *GDB* sólo ofrece información sobre los hilos de ejecución y sus pilas de invocaciones de funciones en el programa del estudiante, se tuvo que recurrir a una considerable cantidad de mecanismos indirectos e imprecisos para obtener los datos sobre los demás segmentos y flujos de datos. Estos mecanismos se implementaron usando funcionalidad básica de *GDB* como puntos de parada (*BREAKPOINTS*) y vigilancia de expresiones (*WATCHES*).

[18] https://www.gnu.org/software/gdb/

De las 11,733 líneas de código C++ en que se implementó el prototipo, la arquitectura interna requirió más de la mitad, debido a los mecanismos alternativos para obtener detalles no provistos por *GDB*, como puede verse en la Figura 5.23. Aunque el prototipo C++ supera muchas limitaciones del prototipo PowerPoint, introduce otras. De ahí la importancia de evaluar su usabilidad antes de realizar el experimento.

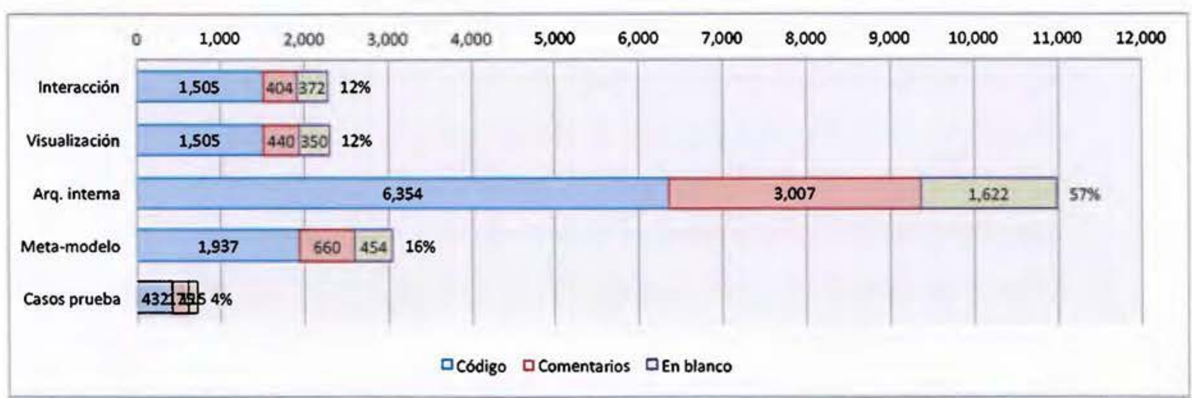



Figura 5.23. Cantidad de líneas de código de los módulos en el prototipo C++

## 5.2.2 Evaluación de usabilidad

La usabilidad es una propiedad clave que debe evaluarse de los sistemas interactivos [Urquiza-Fuentes and Velázquez-Iturbide 2009]. En el caso de esta tesis, conviene evaluarla antes del experimento con el fin de evitar sesgarlo por problemas severos de usabilidad. Para el prototipo C++ se condujo una evaluación con la escala de usabilidad *SUS* (del inglés, *SYSTEM USABILITY SCALE*) [Brooke 1996], por ser considerada la medición subjetiva más confiable, siempre y cuando se realice con diez o más participantes [Dumas and Fox 2012, p.1227].

La sesión de evaluación se realizó con 39 estudiantes del curso "Introducción a la computación", quienes tienen nociones incipientes de programación en lenguajes que no son C/C++. Estos estudiantes se dirigen hacia su primer curso de programación, y por tanto, podrían ser los mayores beneficiados por una visualización de programa, de acuerdo a los resultados cualitativos de la entrevista posterior a la evaluación de usabilidad de prototipo PowerPoint.

La sesión de evaluación se realizó en lecciones normales de cuatro grupos del curso en un laboratorio de computadoras entre el 6 y 7 de julio de 2017. Los estudiantes descargaron una

copia del prototipo C++ para Linux a través de un enlace provisto por el moderador. Se les entregó un instructivo que incluía una historia del juego textual, un tutorial de cómo interactuar con la visualización de programa, y una explicación de la alegoría visual. El prototipo incluyó los siguientes seis ejercicios de programación (niveles) y se les pidió que los corrigieran para que pasaran los casos de prueba.

1. *Entrada y salida (1-1).* El enunciado del problema explica con ejemplos cómo leer de la entrada e imprimir en la salida estándar, y solicita leer dos valores enteros e imprimirlos con textos indicando cuál es el primer valor y cuál es el segundo.
2. *Sumador simple (1-2).* Se solicita leer dos números enteros de la entrada estándar e imprimir el resultado de la suma de los mismos en la salida estándar.
3. *Índice de masa corporal (1-3).* Se solicita corregir un programa que debe calcular el índice de masa corporal ($imc = m/h^2$) a partir de la masa $m$ en kilogramos y la altura $h$ en metros, leídas de la entrada estándar. El código inicial tiene dos errores: declara variables enteras (`int`) en lugar de reales (`double`), y la prioridad de operaciones es incorrecta al calcular el cuadrado de la altura. El enunciado provee explicaciones de tipos de datos y prioridad de operadores en C++.
4. *Fahrenheit a Celsius (1-4).* El código inicial realiza la conversión aplicando la fórmula correcta $C = 5/9 * (F - 32)$ literalmente. Sin embargo, C++ calcula 5/9 como cero dado que sus operandos son enteros. El enunciado explica la diferencia entre la división entera y real en este lenguaje de programación.
5. *Cupones para el sorteo (2-1).* El problema está contextualizado a una compañía de telefonía celular que quiere dar un cupón para un sorteo a sus clientes por cada 5000 colones consumidos. Como código inicial se da una subrutina `main()` vacía. El enunciado provee pistas para leer el monto como un número flotante e imprimir el número de cupones que es resultado de la división entera.
6. *Fahrenheit a Kelvin (2-2)*: Solicita corregir un programa que debe convertir temperaturas, pero tiene interferencia de una variable local con una global. Es el ejemplo que se proveyó a lo largo de la subsección anterior.

Inicialmente sólo el nivel "1-1" está habilitado en el mapa de niveles. Cuando el participante aprueba todos los casos de prueba de un nivel, el prototipo habilita el siguiente nivel en el mapa. Los estudiantes dispusieron de las dos horas de la lección para resolver los ejercicios. Una vez finalizados o se venciera el tiempo de la sesión, los participantes respondieron el

cuestionario de la Figura 5.24, el cual incluye las diez preguntas originales de la escala *SUS* traducidas al español más seis preguntas adicionales sobre la experiencia del usuario. De las respuestas a este instrumento surgen los principales resultados de la evaluación de usabilidad del prototipo C++.

| ***Sobre el prototipo botNeumann++:*** | En desacuerdo | | | | De acuerdo |
|---|---|---|---|---|---|
| 1. Pienso que me gustaría usar este sistema con frecuencia | 1 | 2 | 3 | 4 | 5 |
| 2. Encuentro este sistema innecesariamente complejo | 1 | 2 | 3 | 4 | 5 |
| 3. Pienso que el sistema es fácil de usar | 1 | 2 | 3 | 4 | 5 |
| 4. Creo que necesitaría soporte técnico para hacer uso del sistema | 1 | 2 | 3 | 4 | 5 |
| 5. Encuentro las diversas funciones del sistema bastante bien integradas | 1 | 2 | 3 | 4 | 5 |
| 6. He encontrado demasiada inconsistencia en este sistema | 1 | 2 | 3 | 4 | 5 |
| 7. Creo que la mayoría de la gente aprendería a hacer uso del sistema rápidamente | 1 | 2 | 3 | 4 | 5 |
| 8. He encontrado el sistema bastante incómodo para usar | 1 | 2 | 3 | 4 | 5 |
| 9. Me he sentido muy seguro(a) haciendo uso del sistema | 1 | 2 | 3 | 4 | 5 |
| 10. Necesitaría aprender un montón de cosas antes de poder manejar el sistema | 1 | 2 | 3 | 4 | 5 |

11. ¿Qué fue lo que más le gustó? __________
12. ¿Qué fue lo menos interesante? ¿qué fue lo más aburrido? __________
13. ¿Siente que las animaciones sivieron para entender mejor cómo trabaja la máquina?
    ( ) Nada ( ) Poco ( ) Algo ( ) Bastante ( ) Mucho
14. ¿Qué recomendaría para mejorar la experiencia? __________
15. ¿Cuál lenguaje de programación es el que más domina? __________
    ¿En qué nivel? ( ) Principiante ( ) Intermedio ( ) Avanzado ( ) Experto

Figura 5.24. Cuestionario de evaluación de usabilidad del prototipo C++

### 5.2.3 Resultados de la evaluación de usabilidad

Los datos generados durante la evaluación de usabilidad fueron las respuestas al cuestionario de la Figura 5.24 y el progreso de los participantes registrado en las bitácoras del prototipo C++. Como se aprecia en la distribución de la Figura 5.25, la mayoría de participantes tardaron cerca de una hora en resolver los cinco ejercicios y en responder el cuestionario.

Después de interaccionar con el prototipo C++, los participantes contestaron el cuestionario de la Figura 5.24. Al primer grupo de participantes se les envió un enlace hacia una versión digital del cuestionario por correo electrónico, sin embargo, seis participantes no lo respondieron. Para incrementar la tasa de respuesta, el cuestionario se administró en papel para los restantes tres grupos. En total 33 de 39 participantes respondieron el cuestionario, lo que supera el mínimo de 10 reportes para considerar la medición *SUS* como confiable. El promedio de los 33 reportes de usabilidad fue de 70.68. De acuerdo a [Bangor et al. 2009], la

interpretación de este resultado es que los estudiantes de "Introducción a la computación" consideraron que el prototipo C++ tiene una "buena" usabilidad. El histograma de la Figura 5.26 muestra la distribución de la usabilidad percibida por los participantes.

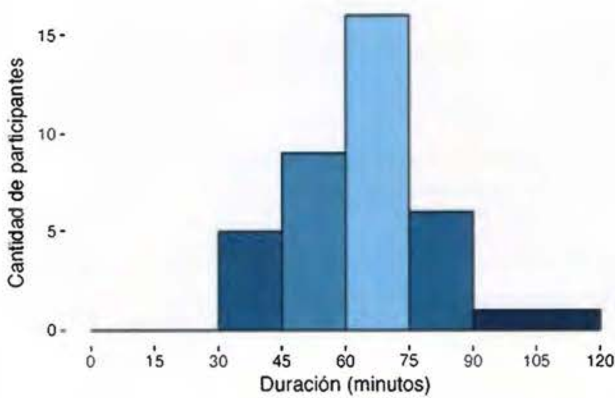



Figura 5.25. Duración de la evaluación de usabilidad del prototipo C++

La pregunta 11 del cuestionario, "¿Qué fue lo que más le gustó?" (Figura 5.24), permite identificar **amenidades de usabilidad** del prototipo (traducción escogida para *USABILITY FINDINGS*), que son características positivas del sistema que convienen identificar para asegurarse de que no se pierdan en las actualizaciones del sistema y futuras evaluaciones de usabilidad [Tullis and Albert 2013, p.101]. El Cuadro 5.5 agrupa las 42 amenidades respondidas en esta pregunta en cuatro categorías. Las categorías en la parte superior tuvieron más reportes de amenidad (R). En general, los estudiantes consideraron el prototipo como una herramienta entretenida y retadora, muy adecuada para el aprendizaje de C++, con una interfaz fácil de usar y una metáfora muy agradable, y que ayuda a comprender cómo el código es ejecutado.

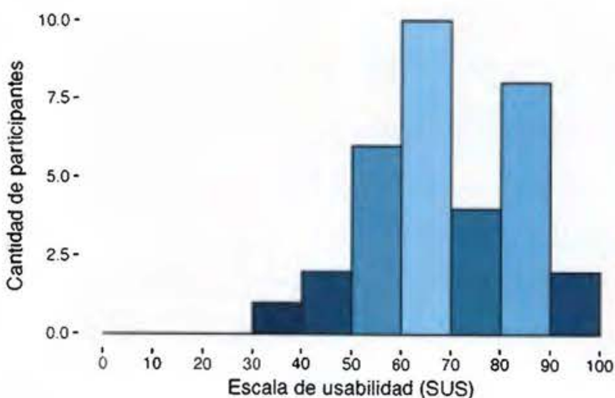



Figura 5.26. Distribución de la usabilidad percibida por los participantes

Dado que los estudiantes de "Introducción a la computación" tienen nociones muy elementales de programación y casi nulas de C++, las animaciones pueden resultar de utilidad o muy confusas. La pregunta 13 del cuestionario (Figura 5.24) pretende determinar la percepción de la utilidad de las animaciones. De acuerdo a la distribución de la Figura 5.27, la mayoría de estudiantes consideraron las animaciones del prototipo C++ como bastante útiles, lo cual es afín a las amenidades reportadas.

Cuadro 5.5. Amenidades del prototipo C++ por número de reportes (R)

| R | Categoría | Amenidades reportadas por los participantes |
|---|---|---|
| 16 | Herramienta de aprendizaje | La herramienta ayuda a aprender C++, a resolver problemas y aprender a leer los enunciados con cuidado. |
| 11 | Interfaz gráfica y alegoría visual | La interfaz gráfica es amigable, fácil de usar, y rápida en tiempo de respuesta. Los diseños y las animaciones gustaron mucho. |
| 9 | Ludificación | El sistema es entretenido, dinámico, interactivo, basado en una historia original, es retador, no regala las respuestas. Para resolver un nivel se requiere conocimiento de los anteriores. |
| 6 | Máquina nocional | Ver cómo funciona el código al ejecutarlo, en especial las variables y las operaciones. Poder detectar errores y corregirlos. |

La mayor utilidad de una evaluación de usabilidad es quizás detectar problemas de usabilidad y ofrecer recomendaciones [Tullis and Albert 2013, p.99]. Aunque el cuestionario de la Figura 5.24 no se diseñó para reporte de problemas, se acopió la lista del Cuadro 5.6 del análisis de las preguntas 12 y 14 del cuestionario.

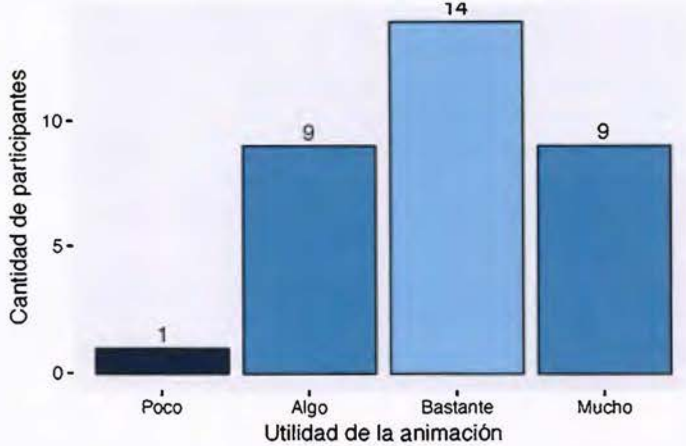



Figura 5.27. Utilidad percibida del prototipo C++ por los participantes

Durante la sesión, cuando un estudiante enfrentaba un problema de usabilidad, lo reportaba al moderador (el autor de esta tesis). El problema en la posición 1 del Cuadro 5.6 fue el más reportado durante la sesión y ocurrió en dos escenarios. En el primer escenario, el prototipo

C++ terminaba abruptamente su ejecución si se producían ciertos errores de compilación. Fue un problema de alta severidad dado que los estudiantes necesitaban los diagnósticos del compilador para ayudarse a corregir sus programas y así completar la tarea (pasar los casos de prueba). En tales casos, el moderador proveyó sugerencias a los estudiantes para que pudieran corregir los programas y poder avanzar en los niveles. El segundo escenario fue cuando los usuarios hacían doble clic en los niveles del mapa de niveles. El prototipo C++ reaccionaba tratando de cargar dos veces la pantalla de nivel y terminaba su ejecución de forma abrupta. El moderador les sugirió hacer clic una única vez en el nivel. Ambos errores fueron corregidos en el código fuente del prototipo inmediatamente después de finalizada la evaluación de usabilidad de los cuatro grupos.

El análisis de las bitácoras es trabajo futuro que permitirá obtener las duraciones, número de intentos, la tasa de completitud y eficiencia de cada uno de los seis niveles. Estos resultados serán reportados en un artículo tentativamente para la revista *INTERNATIONAL JOURNAL OF HUMAN–COMPUTER INTERACTION*[19].

Cuadro 5.6. Problemas de usabilidad del prototipo C++ reportados por estudiantes de "Introducción a la computación" ordenados por su número de reportes (R) y severidad (S)

| # | R | S | Problema | Recomendación |
|---|---|---|---|---|
| 1 | 11 | 2 | Terminación abrupta del programa cuando se generan ciertos errores de compilación o se hace doble clic en los niveles. | Corregir el código fuente del programa, lo cual se hizo una vez terminada la evaluación de usabilidad. |
| 2 | 12 | 1 | Los estudiantes encontraron aburrido, extenso o confuso de entender las instrucciones de la actividad o los enunciados de problemas | Ajustar las instrucciones y enunciados con estudiantes de esta población mediante una prueba piloto. |
| 3 | 8 | 1 | Ocho estudiantes indicaron que les tomó tiempo entender lo que debían hacer o cómo resolver los problemas por su falta de conocimiento de C++. Tenían que volver a niveles anteriores para leer las instrucciones. | Proveer instrucciones adicionales o pistas, por ejemplo, a cambio de puntos. Ajustar las instrucciones con esta población mediante una prueba piloto. |
| 4 | 4 | 1 | Cuatro participantes hicieron referencia a la dificultad de los ejercicios, sobre todo al ejercicio "1-1" que fue el que más les tomó tiempo. | Ordenar o reajustar los ejercicios de acuerdo a la dificultad con estudiantes de esta población mediante una prueba piloto. |
| 5 | 3 | 1 | Tres estudiantes reportaron que cuando se enfrentaban a un error del programa, no sabían cómo resolverlo. | Proveer pistas adicionales a los errores de compilación o lo que hace al programa fallar los casos de prueba. |
| 6 | 2 | 1 | Los valores de las variables flotantes no son legibles. No se explica cómo el robot lee los datos en la animación. | Formatear los valores flotantes o permitir al usuario escoger el formato. Proveer explicaciones textuales de la |

[19] https://www.tandfonline.com/toc/hihc20/current

| # | R | S | Problema | Recomendación |
|---|---|---|---|---|
| | | | | animación. |
| 7 | 2 | 0 | Dos participantes consideraron que el código fuente que había que escribir y otras tareas, se sintieron repetitivas, pero no era algo grave y se entiende que es necesario para aprender mejor. | Variar el contexto de los ejercicios sin incrementar el nivel de dificultad. |
| 8 | 2 | 0 | Dos estudiantes con conocimiento de programación en Java sugirieron agregar más niveles. | Se pueden agregar más niveles, pero sólo si son de carácter opcional, por la duración de la actividad. |
| 9 | 1 | 0 | Un participante indicó que la aplicación podría ser más divertida. | Implementar más elementos de ludificación. |
| 10 | 2 | -1 | Dos participantes indicaron que las animaciones y las pruebas tardan mucho. Los estudiantes no utilizaron algún mecanismo para controlar la velocidad. | Hacer explícita en las instrucciones los mecanismos para controlar la visualización y su velocidad. |

### 5.2.4 Discusión de la evaluación de usabilidad

Esta primera evaluación de usabilidad del prototipo C++ se realizó con estudiantes que tienen nociones incipientes de programación, y que podrían beneficiarse de una herramienta de aprendizaje. Pese al desconocimiento de programación en C++, la mayoría lograron completar las seis tareas propuestas en un lapso cercano a una hora, y encontraron el prototipo con una "buena usabilidad". Cualitativamente, los estudiantes describieron al prototipo como una herramienta muy adecuada para el aprendizaje del lenguaje de programación C++, con una interfaz fácil de usar, y una metáfora muy agradable. Estos resultados son muy positivos considerando las nociones elementales de programación de los participantes, quienes podrían haber encontrado las tareas como abrumadoras o ininteligibles.

Se identificaron dos problemas severos de usabilidad que provocaban que el prototipo terminara abruptamente su ejecución en el laboratorio de computadoras. Ambos se corrigieron de inmediato tras finalizada la sesión de evaluación. Estos problemas probablemente afectaron de forma negativa el promedio *SUS* obtenido de 70.68, el cual se convierte en la línea base (en inglés, *BASELINE*) para comparar futuras mediciones de este instrumento, como la que se hizo al finalizar el experimento reportado en la siguiente sección.

## 5.3 Evaluación experimental del prototipo C++

Esta sección emplea el prototipo C++ de botNeumann++ para responder la pregunta de conocimiento de esta tesis: "determinar si las visualizaciones de programa que aplican los constructos computacionales propuestos en esta investigación (alegorías visuales y ludificación), son más eficaces que herramientas existentes para ayudar a estudiantes universitarios de programación a comprender la máquina nocional de un lenguaje de programación" (p.9). Se realizó un experimento con estudiantes de "Programación II" a quienes se les pidió corregir programas con errores no triviales, ya que una comprensión de los programas es necesaria para poder corregirlos [McCauley et al. 2008, p.76].

Se proveyó a los participantes dos herramientas para estudiar los programas y corregirlos: el prototipo C++ de botNeumann++ (tratamiento), y los ambientes de desarrollo con depuradores integrados a los que están familiarizados en el curso (control). El experimento trató de evaluar si existe un efecto significativo de la herramienta asignada en las correcciones de los errores. Dos hipótesis podrían responder a esta pregunta:

1. Hipótesis nula: Las dos herramientas tienen el mismo efecto en la tasa de detección y corrección de errores no triviales.
2. Hipótesis alternativa: La visualización lúdica de programa influye en una mayor tasa de detección y corrección de errores que herramientas tradicionales.

En caso de rechazarse la hipótesis nula, se concluye que las visualizaciones lúdicas de programa ayudan a comprender mejor los programas y corregir errores en ellos que herramientas tradicionales. A continuación se explican los métodos para tomar o rechazar las hipótesis planteadas.

### 5.3.1 Método experimental

Se siguió el diseño experimental 6 de Campbell y Stanley, llamado "diseño de grupo control con pos-test únicamente" por tener las mismas o más ventajas que los otros dos diseños experimentales planteados por estos autores con menos intervenciones, lo cual es deseable para sesiones experimentales cortas y poblaciones reducidas [Campbell and Stanley 1973, p.22,55]. La codificación de este diseño se presenta en la Figura 5.28 siguiendo la

nomenclatura de Campbell y Stanley. La primera línea de la Figura 5.28 representa el grupo tratamiento y la segunda al grupo control, los cuales son concurrentes. De izquierda a derecha se leen los eventos en el orden temporal. Los participantes se distribuyen aleatoriamente a ambos grupos (representado con una R). El grupo tratamiento en la primera línea es expuesto al acontecimiento experimental (representado por una X), que para esta tesis es el prototipo C++ de botNeumann++. Por su parte, el grupo control en la segunda línea realiza las mismas tareas con las herramientas conocidas o tradicionales. Finalmente ambos grupos son observados o medidos (representado por una O) [Campbell and Stanley 1973, p.18].

Figura 5.28. Diseño de grupo control con post-test únicamente

La población escogida fueron estudiantes de "Programación II" porque es el curso del Bachillerato en Computación e Informática cuyo lenguaje de facto es C++ y coincide con la máquina nocional de botNeumann++. El experimento se realizó al finalizar el primer ciclo lectivo, momento en que los estudiantes tienen un dominio ya establecido del lenguaje de programación. Por tanto, se escogieron ejercicios de programación que involucraran nociones de C++ que no son cubiertas en el temario del curso, con el fin de hacer necesaria una herramienta para estudiar el comportamiento de los programas en tiempo de ejecución. También se incluyeron algunos errores comunes de programación, dado que aunque son conocidos, es difícil detectarlos y herramientas de visualización podrían ayudar en este proceso.

Se escogieron para la evaluación experimental, los cuatro ejercicios explicados en los siguiente apartados (los números entre paréntesis corresponden a los niveles de la Figura 5.19, p.193). Se les pidió a los participantes corregirlos para que pasaran los casos de prueba haciendo el mínimo número de modificaciones al código fuente dado.

#### *5.3.1.1 Fahrenheit a Kelvin (1-1)*

El ejercicio 1-1 solicitó corregir el programa conversor de temperaturas del Listado 5.1, con que se ejemplificó la sección 5.2 (Prototipo 2: C++). Este programa tiene dos errores. El primer error es la interferencia de las dos variables con el mismo nombre `temperatura`. La

línea 9 lee valores de la entrada estándar en la variable global declarada en la línea 5. A partir de la línea 10 se declara una variable local que oculta a la global en las líneas posteriores. Una visualización de programa debe mostrar la variable global de la línea 5 en el segmento de datos y cambiarla con la lectura de la línea 9. Al ejecutar la línea 10 debe animar la aparición de otra variable `temperatura` en el segmento de pila y los accesos en el resto de líneas con esta variable. Una corrección recomendada es eliminar la declaración de la variable global (línea 5) y mover la declaración de la variable local (línea 10) al inicio del cuerpo de la subrutina `main()`. Una segunda corrección es eliminar la declaración de la variable local (línea 10).

El segundo error en el código del Listado 5.1 es la división entera `5 / 9` de la línea 12, ya que se evalúa en cero y al multiplicarse por el resto de la expresión aritmética genera siempre el valor `0.0` indiferentemente del valor de la temperatura. Una solución recomendada es cambiar las constantes literales enteras por reales, de la forma `5.0 / 9.0`, ó utilizar conversión de tipos, por ejemplo `double(5) / 9`. Se considera un rodeo alterar la expresión aritmética modificando el orden de evaluación de los operadores, por ejemplo: `5 * (temperatura + 459.67) / 9`.

```
 1 #include <iomanip>
 2 #include <iostream>
 3 using namespace std;
 4
 5 double temperatura,
 6
 7 int main()
 8 {
 9   cin >> temperatura,
10   double temperatura = 0.0;
11
12   temperatura = 5 / 9 * (temperatura + 459.67);
13
14   cout << fixed << setprecision(3);
15   cout << temperatura << endl;
16   return 0;
17 }
```

Listado 5.1. Código C++ inicial del ejercicio 1-1 Fahrenheit a Kelvin

#### *5.3.1.2 Porcentaje de incremento (1-2)*

El ejercicio 1-2 solicitó corregir el programa del Listado 5.2, el cual trata de calcular el porcentaje de incremento entre una cantidad inicial $I$ y una cantidad final $F$ mediante la

relación $100(F - I)/I$. Se ofrece como contexto el porcentaje de incremento de muertes en accidentes de tránsito entre un año y otro.

El código del Listado 5.2 tiene dos errores. En la línea 29, para obtener los argumentos con que se invocará a la subrutina `imprimir()` se debe hacer dos invocaciones a la subrutina `leer()`. El orden de invocación de estas dos subrutinas es dependiente del compilador, ya que el estándar C/C++ no impone un orden. El compilador oficial de Linux, GCC[20], evalúa los argumentos de una invocación de derecha a izquierda, por tanto, lee al revés los valores para los parámetros `inicial` y `final` provenientes del caso de prueba. Este orden de evaluación no es un tema del curso "Programación II". Una herramienta hace visible este efecto en la invocación de la subrutina `imprimir()`, al revelar los valores de los parámetros.

```
1 #include <iomanip>
2 #include <iostream>
3 using namespace std;
4
5 double leer()
6 {
7   double valor;
8
9   cin >> valor,
10  return valor,
11 }
12
13 double incremento(double inicial, double final)
14 {
15   return 100 * (final - inicial) / inicial,
16 }
17
18 void imprimir(double inicial, double final)
19 {
20   if ( inicial < 0 || final < 0 )
21      cout << "error\n";
22   else
23      cout << incremento(inicial, final) << "%\n";
24 }
25
26 int main()
27 {
28   cout << fixed << setprecision(2);
29   imprimir(leer(), leer());
30 }
```

Listado 5.2. Código C++ inicial del ejercicio 1-2: Porcentaje de incremento

[20] https://gcc.gnu.org/

El error del orden de evaluación se corrige invocando la subrutina `leer()` y guardando el valor retornado en al menos una variable antes de invocar la subrutina `imprimir()` de la línea 29. De esta forma, se obliga a "serializar" la lectura de los valores en el orden correcto. Un rodeo consiste en invertir los nombres de los parámetros en la función `imprimir()` o en `incremento()`. Es un rodeo porque pasa los casos de prueba si se compila con GCC, pero fallará con compiladores que tengan otro orden de evaluación, como *CLANG*. El segundo error ocurre en la línea 20 del Listado 5.2. El enunciado establece que se debe reportar el mensaje de "error" si los valores son cero o negativos. Se corrige cambiando los operadores relacionales de "<" a "<=".

#### *5.3.1.3 ¿Es binario? (1-3)*

El ejercicio 1-3 solicita corregir el programa del Listado 5.3, el cual debe leer números de la entrada estándar e indicar si están compuestos únicamente de dígitos binarios o no. Se da como contexto un juego educativo hipotético.

```
1 #include <iostream>
2 using namespace std;
3
4 short is_binary(long long number){
5   if(number<0) number-=number;
6   bool bin, in_loop=number!=0;
7   do {
8      switch (number%10 ){
9         case 0:{bin=true;}
10        case 1:{bin=true;}
11        default:{
12           bin=false,
13        }
14     }
15
16     number/=10;
17     in_loop=number=!0;
18  }
19  while(in_loop);
20     return bin;
21 }
22
23 int main()
24 {
25   long long x;
26   cin>>x;
27   cout<<is_binary(x)<<endl;
28 }
29
```

Listado 5.3. Código C++ inicial del ejercicio 1-3: ¿Es binario?

El código dado del Listado 5.3 tiene cuatro errores y varias malas prácticas de programación que hace difícil comprenderlo, con el fin de incentivar a los participantes a usar la herramienta para estudiarlo. Los cuatro errores son:

1. En la línea 5, el número se resta a sí mismo con el operador combinado "`-=`" lo que produce cero. Se debe cambiar por alguna expresión aritmética que cambie el signo a la variable, como `number=-number` ó `number=-1*number`.
2. En las líneas 9 y 10 se debe romper los casos del `switch` con la palabra reservada `break`. Es un error común de programación.
3. En la subrutina `is_binary` se debe detener el ciclo `do-while` cuando se detecte que el número contiene un dígito que no es cero o uno. Hay varias formas de lograrlo, como hacer nula la variable `number` en el caso `default`, o retornar de la subrutina en este mismo caso.
4. En la línea 17 se asigna la negación de cero (`=!`) en las variables `number` e `in_loop` que causa un ciclo infinito. Se corrige con el operador de diferencia (`!=`).

#### *5.3.1.4 Sonido de animales (1-4)*

El ejercicio 1-4 solicita corregir el código del Listado 5.4, contextualizado a una aplicación para niños que asocia polimórficamente letras del alfabeto con sonidos de animales. El programa tiene un único error en la línea 8. En el constructor de la clase base `Animal()` se invoca el método virtual `makeSound()`. De acuerdo al estándar C++, la versión de la clase derivada (líneas 15, 20, y 25) no se debe invocar porque la parte base del objeto se está construyendo, lo que indica que la parte derivada está aún pendiente de construcción. Si el método virtual se invocara, tendría acceso a los miembros aún no creados de la clase derivada. Los compiladores actuales no generan algún tipo de mensaje por este error y es un concepto avanzado usualmente no cubierto en el temario del curso. Una herramienta de visualización no explica este fenómeno, pero ayuda al usuario a generar hipótesis al descubrir que el método de la clase base se invoca en lugar de la versión correspondiente en la clase derivada. Una solución es mover la invocación del método virtual `makeSound()` del constructor (línea 8) a un lugar donde el objeto haya terminado su construcción, por ejemplo, la línea 45.

Los cuatro ejercicios escogidos plantean conceptos nuevos, errores comunes de programación, y código confuso, con el fin de ameritar una herramienta para comprender el código y poder corregirlo.

```
1 #include <cctype>
2 #include <iomanip>
3 #include <iostream>
4 using namespace std;
5
6 class Animal {
7  public:
8   Animal() { cout << makeSound() << ".mp3\n"; }
9   virtual const char* makeSound() const { return ""; }
10  virtual ~Animal() { }
11 };
12
13 class Bird : public Animal {
14  public:
15   virtual const char* makeSound() const override { return "cucu"; }
16 };
17
18 class Cow : public Animal {
19  public:
20   virtual const char* makeSound() const override { return "mu"; }
21 };
22
23 class Duck : public Animal {
24  public:
25   virtual const char* makeSound() const override { return "cuak"; }
26 };
27
28 Animal* createAnimal(char letter) {
29   switch ( tolower(letter) ) {
30      case 'b': return new Bird(); break;
31      case 'c': return new Cow(); break;
32      case 'd': return new Duck(); break;
33   }
34   return nullptr;
35 }
36
37 int main() {
38   char letter;
39   cin >> letter;
40
41   Animal* animal = createAnimal(letter);
42
43   if ( ! animal )
44      cout << "unregistered\n";
45
46   delete animal;
47 }
```

Listado 5.4. Código C++ inicial del ejercicio 1-4: Sonido de animales

#### *5.3.1.5 Selección de las herramientas*

La herramienta en el grupo tratamiento fue el prototipo C++ de botNeumann++. En ella, los cuatro ejercicios estuvieron disponibles como niveles independientes, dado que no existe una linealidad temática entre ellos. Es decir, los cuatro niveles estuvieron siempre habilitados para evitar que un estudiante se viese limitado a continuar si no podía resolver uno de los ejercicios (Figura 5.19, p.193). Esta disponibilidad fue equivalente en el grupo control.

El diseño experimental 6 (Figura 5.28, p.205) compara el tratamiento X contra las actividades que el grupo control realiza durante la sesión experimental [Campbell and Stanley 1973, p.31], que en esta tesis corresponden a corregir los mismos programas usando las herramientas que los estudiantes han aprendido en el curso para este fin. Convenientemente, el diseño experimental 6 permite establecer un segundo tratamiento $X_2$ para el grupo control [Campbell and Stanley 1973, p.31]. Se estudió la posibilidad de que fuese una visualización de programa existente que utilizara metáforas tradicionales sin ludificación, y se obtuvieron los siguientes resultados.

Para poder asignar una herramienta a los participantes del grupo control, se necesita naturalmente una versión funcional que puedan utilizar. El Cuadro 5.7 lista las visualizaciones de programa disponibles para la máquina nocional de C/C++, obtenidas de la revisión de literatura del capítulo 3. La cuarta columna lista los problemas enfrentados con cada herramienta. En síntesis, *JGRASP* sólo visualiza código en *JAVA*, mientras que *THE TEACHING MACHINE*, *VIP* y *PROVIT* no se ejecutan en navegadores o sistemas operativos modernos, como los disponibles en los laboratorios de la Escuela de Ciencias de la Computación e Informática. *ONLINE PYTHON TUTOR* no implementa entrada estándar para C++, lo que impide evaluar un programa con casos de prueba. Por tanto, ninguna de las visualizaciones de programa para C++ pudieron usarse en el grupo control.

Las dos herramientas funcionales del Cuadro 5.7 fueron SeeC y Virtual-C IDE para la máquina nocional de C. La opción de realizar el experimento con ejercicios de programación en C se declinó por la siguiente razón. De los dos profesores que estuvieron anuentes a colaborar con en el experimento, uno hace una separación explícita entre C y C++ en el curso mientras que el otro no. Por tanto, este factor sería una causa importante de variabilidad en el experimento, que podría provocar que los ejercicios sean aproximadamente triviales para los estudiantes de un profesor y cercanamente inviables en el tiempo de la sesión para los otros.

Cuadro 5.7. Visualizaciones de programa disponibles para C/C++

| # | Nombre | Máquina | Problema |
|---|---|---|---|
| 1 | *JGRASP* | *JAVA*, C, C++, ... | Sólo visualiza la ejecución de código *JAVA*. Para C++ actúa como un depurador tradicional con limitaciones. |
| 2 | *THE TEACHING MACHINE* | *JAVA*, C++ | Es una aplicación en *JAVA* para navegadores web. *JAVA* no es soportado en navegadores modernos. No es claro cómo se pueden integrar casos de prueba. |
| 3 | *VIP* | C++ | No se logró hacer correr la versión disponible (del año 2008). |
| 4 | *ONLINE PYTHON TUTOR* | *PYTHON*, C, C++, ... | No soporta entrada estándar en C++. Al momento del experimento, el sitio web experimentaba caídas frecuentes. |
| 5 | *SEEC* | C | No soporta C++. No soporta edición de código. |
| 6 | *VIRTUAL-C IDE* | C | No soporta C++. Usa el intérprete *SIMPLEC*, el cual produce algunas animaciones erróneas de los ejercicios planteados. |
| 7 | *PROVIT* | C | No soporta C++. Es una aplicación en *JAVA* para navegadores web, lo cual no es soportado en navegadores modernos. Está disponible sólo en japonés. |

Ante la ausencia de visualizaciones de programa apropiadas para el grupo control, se tomó la resolución de mantener el diseño experimental con las herramientas tradicionales del curso para comprender programas. Esta es la misma resolución tomada por [Byckling and Sajaniemi 2006], que consiste en utilizar depuradores visuales. Un **depurador visual** (en inglés, *VISUAL DEBUGGER*) es una aplicación de interfaz gráfica de usuario (por ejemplo, *GEDE*) que permite controlar a un depurador textual (por ejemplo, *GDB*) en segundo plano. Los depuradores visuales no ofrecen toda la funcionalidad del depurador textual sino un subconjunto, como es el caso también de botNeumann++. Sin embargo, la funcionalidad de un depurador visual es más amplia que la de botNeumann++, dado que son herramientas diseñadas para desarrolladores profesionales y abundantemente utilizadas en la industria.

El uso de depuradores visuales es también una práctica común en ambientes de enseñanza. Por ejemplo, en [Byckling and Sajaniemi 2006] se comparó el efecto de enseñar un curso con el depurador visual del ambiente de desarrollo integrado (*IDE*) Turbo Pascal, contra la visualización de programa *PLANANI*. En el caso de ECCI, el aprendizaje de un depurador es un objetivo y tema obligatorio del curso "Programación I" y se continúa en el temario de "Programación II". Los profesores optan por enseñar depuradores integrados en ambientes de desarrollo, y por tanto *depuradores visuales*. Los dos profesores que colaboraron en el experimento utilizaron los depuradores visuales de *CODE::BLOCKS*[21] y *QTCREATOR*[22]. Ambos

[21] http://www.codeblocks.org/

ambientes son ideados con fines industriales y altamente probados por su uso masivo. A diferencia de *PLANANI* que fue enseñado a lo largo del curso en el grupo tratamiento en [Byckling and Sajaniemi 2006, p.415], botNeumann++ no lo fue. Los estudiantes tuvieron una exposición a botNeumann++ de unos minutos, contra los ambientes de desarrollo usados a lo largo del curso de "Programación II". Esto crea una diferencia de familiaridad y usabilidad en contra del tratamiento.

#### *5.3.1.6 Tarea experimental*

En el caso de botNeumann++ los cuatro ejercicios estaban integrados en el ejecutable. Para los depuradores visuales, los ejercicios se proveyeron como carpetas con los archivos del Cuadro 5.8. Para corregir los ejercicios, el participante escogió uno de los dos ambientes de desarrollo (*CODE::BLOCKS* o *QTCREATOR*) y abrió el archivo de proyecto respectivo. El ambiente de desarrollo permite editar el código fuente, correrlo, y depurarlo. Cuando el participante quería probar sus correcciones contra los casos de prueba, podía hacerlo con el comando `make test` que utiliza el archivo `Makefile`. Una explicación detallada de este proceso se proveyó a los participantes en un manual durante el experimento.

Cuadro 5.8. Archivos de un ejercicio de programación para el depurador visual

| Archivo | Descripción |
|---|---|
| `problem.es.html` | Enunciado del problema. Es el mismo que botNeumann++ despliega en el panel de información. |
| `solution.cpp` | Código fuente inicial. Es el mismo que botNeumann++ carga en el editor de código y que debe ser corregido por el participante. |
| `input01.txt` `output01.txt` | Un caso de prueba compuesto de una entrada y la salida esperada. Se provee una pareja enumerada de estos dos archivos por cada caso de prueba del ejercicio. |
| `solution.cbp` | Archivo de proyecto de *CODE::BLOCKS*. Al abrirlo, *CODE::BLOCKS* permite editar y depurar el código inicial en `solution.cpp`. |
| `solution.pro` | Archivo de proyecto de *QTCREATOR*. Al abrirlo, *QTCREATOR* permite editar y depurar el código inicial en `solution.cpp`. |
| `Makefile` | Compila la solución del estudiante y la corre contra todos los casos de prueba. |

La Figura 5.29 muestra una sesión de depuración en *CODE::BLOCKS* del primer ejercicio sobre conversión de temperaturas. El usuario puede establecer puntos de parada (*BREAKPOINTS*) ①, iniciar la sesión de depuración ②, interaccionar con el programa a través de su entrada y

[22] https://www.qt.io/

salida estándar ③, estudiar el segmento de pila ④, inspeccionar los valores de variables ⑤, y controlar el depurador ⑥. La Figura 5.30 muestra el proceso en el mismo orden para *QTCREATOR*, quien por defecto agrega la lista de hilos de ejecución ④.

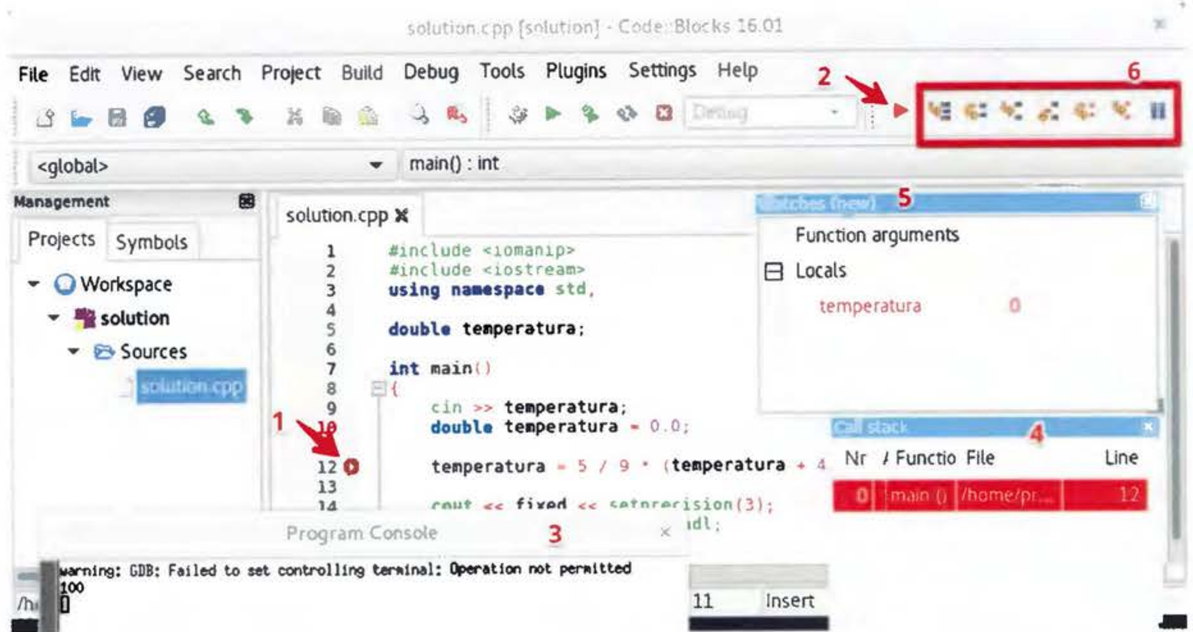



Figura 5.29. Captura de pantalla de una sesión de depuración en *CODE::BLOCKS*

## 5.3.2 Participantes del experimento

Dos profesores de "Programación II" colaboraron brindando crédito académico extra en el curso en proporción al rendimiento de los estudiantes en el experimento. Uno de los profesores impartió dos grupos del curso. En total participaron 35 estudiantes. La evaluación tuvo lugar una vez concluido el ciclo lectivo, en concreto, el miércoles 19 de julio de 2017. La actividad se anunció a los estudiantes como un taller de depuración en lugar de un experimento. Antes de realizar la actividad se hizo un sondeo del interés en participar y las posibilidades de horario de los estudiantes. Acorde a los resultados del sondeo, se realizaron dos sesiones experimentales el mismo día con el fin de maximizar la cantidad de participantes, la primera a las 9:00 a.m., inmediatamente seguida de la segunda a la 1:00 p.m. Ambas sesiones estuvieron planeadas para una duración de tres horas con un límite máximo de tiempo de cuatro horas. Esta duración fue suficiente para la totalidad de los participantes por lo que no representó un elemento de presión.

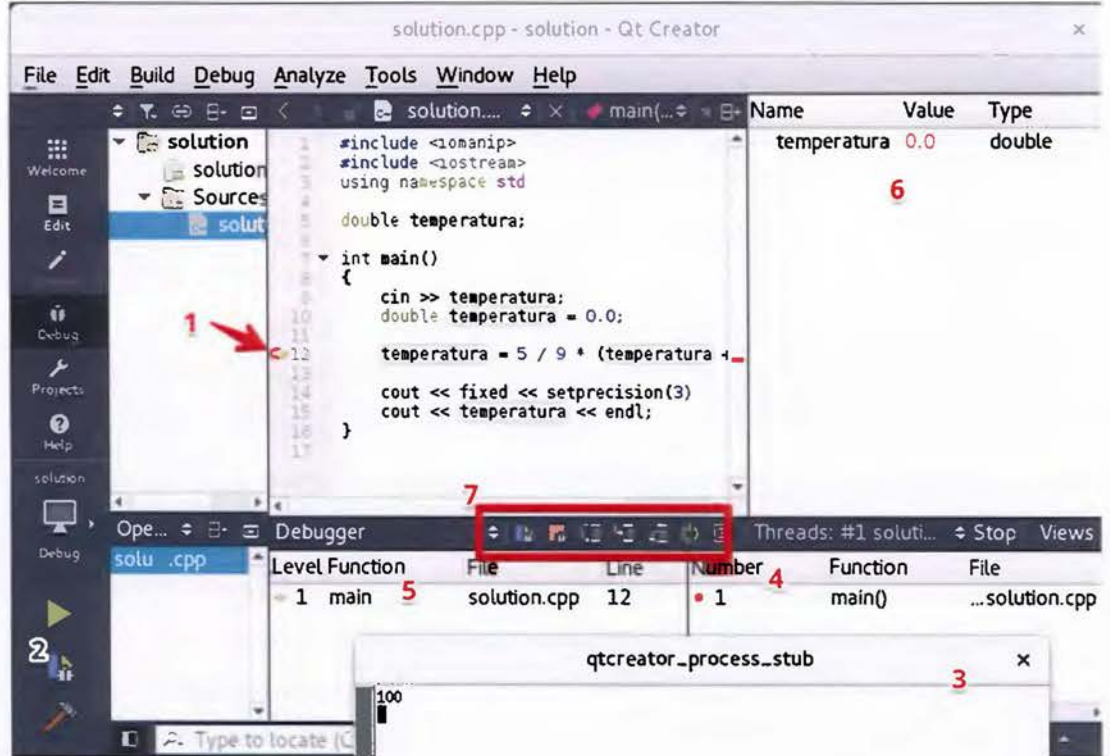



Figura 5.30. Captura de pantalla de una sesión de depuración en QtCreator

Se siguió un diseño entre-sujetos para el experimento. Dos laboratorios de computadoras se reservaron, uno para el grupo control (202-IF) y otro para el tratamiento (204-IF), asignados al azar. Instructivos impresos para cada grupo, se intercalaron iniciando con el grupo control en la parte superior. La pila de instructivos se colocó cerca de la entrada de los laboratorios. Cada estudiante que llegó a la actividad tomó la siguiente hoja disponible en la parte superior de la pila. De esta forma la asignación de los participantes a los grupos tratamiento y control fue balanceada y al azar, dado que no se controla el orden de llegada de los estudiantes.

El instructivo indicaba al estudiante dirigirse al laboratorio asignado, tomar una computadora libre, iniciarla en Linux (*FEDORA*), descargar un archivo comprimido con los materiales y extraerlos. El hiperenlace hacia el archivo estaba impreso en el instructivo y era distinto para cada grupo. El contenido del archivo comprimido fue:

1. Un manual en formato de documento portable (*PDF*, del inglés *PORTABLE DOCUMENT FORMAT*) que consta de dos partes. La primera parte fue una introducción, idéntica para ambos grupos. En la introducción se incluyó una motivación a la importancia de la depuración y se estableció el objetivo de la sesión: corregir los programas para que pasen

los casos de prueba y explicar las causas que los hacían fallar. La segunda parte del manual fue diferente para cada grupo. Su objetivo fue explicar la herramienta asignada y el proceso para detectar y corregir errores con los programas brindados. En el caso del grupo tratamiento se proveyó un tutorial de botNeumann++, una narrativa del juego ambientada en los apagones centroamericanos ocurridos en las dos semanas previas al experimento, y una explicación de la alegoría visual. Para el grupo control se proveyó una explicación del contenido de las carpetas de ejercicios (Cuadro 5.8, p.213), cómo correr el programa contra los casos de prueba usando el `Makefile` en la línea de comandos, y un tutorial de depuración tanto para *QtCreator* como *Code::Blocks*. Se pidió a cada estudiante del grupo control escoger sólo uno de estos dos depuradores visuales.

2. Un documento de justificaciones en formato *OpenDocument*, editable en software de oficina como *LibreOffice*[23]. El documento fue idéntico para ambos grupos. Incluyó instrucciones que solicitan al estudiante: (1) identificar únicamente los errores que impiden que los programas pasen los casos de prueba; (2) usar de forma exclusiva la herramienta provista (*botNeumann++*, *Code::Blocks*, o *QtCreator*) para estudiar los programas e identificar errores; (3) explicar la causa del error en los espacios provistos para los cuatro ejercicios en el mismo documento; (4) valorar qué tan útil fue la herramienta para detectar y comprender el error; (5) si utilizó otra herramienta o consultó un sitio web, indicarlo en la justificación; (6) corregir el error haciendo los *cambios mínimos* en el código fuente usando la herramienta provista. En el documento se proveyó un ejemplo y se hizo énfasis en los cambios mínimos a los programas provistos.
3. Los ejercicios de programación. Para el grupo tratamiento se proveyó el ejecutable del prototipo C++ de *botNeumann++* con los ejercicios incorporados. Para el grupo control los ejercicios se incluyeron en subcarpetas.

Durante la sesión los estudiantes usaron las herramientas provistas para detectar los errores, justificar sus causas en el documento editable, valorar la utilidad de la herramienta, y corregir los programas para que pasaran los casos de prueba. La actividad ocurrida en la pantalla de cada participante fue capturada sin sonido utilizando la funcionalidad incorporada de *Screencast* de *Fedora*. Una vez finalizada la sesión, el estudiante evaluó la usabilidad de la

[23] https://www.libreoffice.org/

herramienta asignada con el cuestionario *SUS* (Figura 5.24, p.199). Los datos que se generaron en estas actividades son analizados en la siguiente subsección.

### 5.3.3 Resultados del experimento

De los 35 participantes, se descartó del análisis los datos de dos de ellos que no completaron los ejercicios. Por tanto, la población final fue de 33 estudiantes, 17 en el grupo control y 16 en el grupo tratamiento. Los datos generados por los participantes fueron los siguientes:

1. El código fuente en C++ de los programas corregidos.
2. Documento con las causas que impedían a los programas pasar los casos de prueba y las valoraciones de utilidad de la herramienta asignada.
3. Evaluación de usabilidad (cuestionario *SUS*) y opinión de la experiencia.
4. Video grabado de la sesión mediante captura de pantalla sin sonido.
5. Bitácoras de eventos de los participantes que usaron botNeumann++.

Las correcciones en el código fuente (punto 1 en la lista anterior) se utilizaron para responder la pregunta de investigación y se discuten junto con las justificaciones (punto 2) en el siguiente apartado. En el apartado que le continúa se analiza la usabilidad (punto 3) percibida por esta población. El análisis cualitativo de los videos (punto 4) y de las bitácoras de eventos (punto 5), permitirá determinar si hay diferencias en otros indicadores, como el número de intentos de los participantes para corregir los errores y las duraciones de los mismos. Estos resultados quedan como trabajo futuro a reportar en artículos derivados de esta tesis.

#### *5.3.3.1 Efecto en corrección de errores*

Para determinar si un participante corrigió un error, no es suficiente con compilar el código fuente que modificó y probar si pasa los casos de prueba. A través de soluciones alternativas (rodeos), un programa se puede modificar para pasar los casos de prueba, en lugar de corregir directamente el error. Se requiere criterio experto de profesores para evaluar en qué grado los errores fueron realmente corregidos por los participantes. Dado que el criterio experto es una medida subjetiva, se procuró tener valoraciones hechas por varios profesores. Dos profesores del curso "Programación II" estuvieron dispuestos a revisar las soluciones de los

participantes: un profesor ajeno al experimento quien no impartió el curso durante el semestre en que se realizó el experimento, y el autor de esta tesis.

Los archivos fuente modificados por los participantes y sus justificaciones necesitaron procesamiento previo para facilitar la labor de los expertos y para asegurar la calidad de los resultados. Por ejemplo, las justificaciones se debieron anonimizar y los identificadores de los participantes se debieron aleatorizar. Se escribió un guión para el intérprete de comandos de Linux (en inglés, *BASH SCRIPT*) que toma los datos producidos por los participantes y genera páginas web como el extracto mostrado en la Figura 5.31. Las partes rotuladas de la página se explican a continuación.

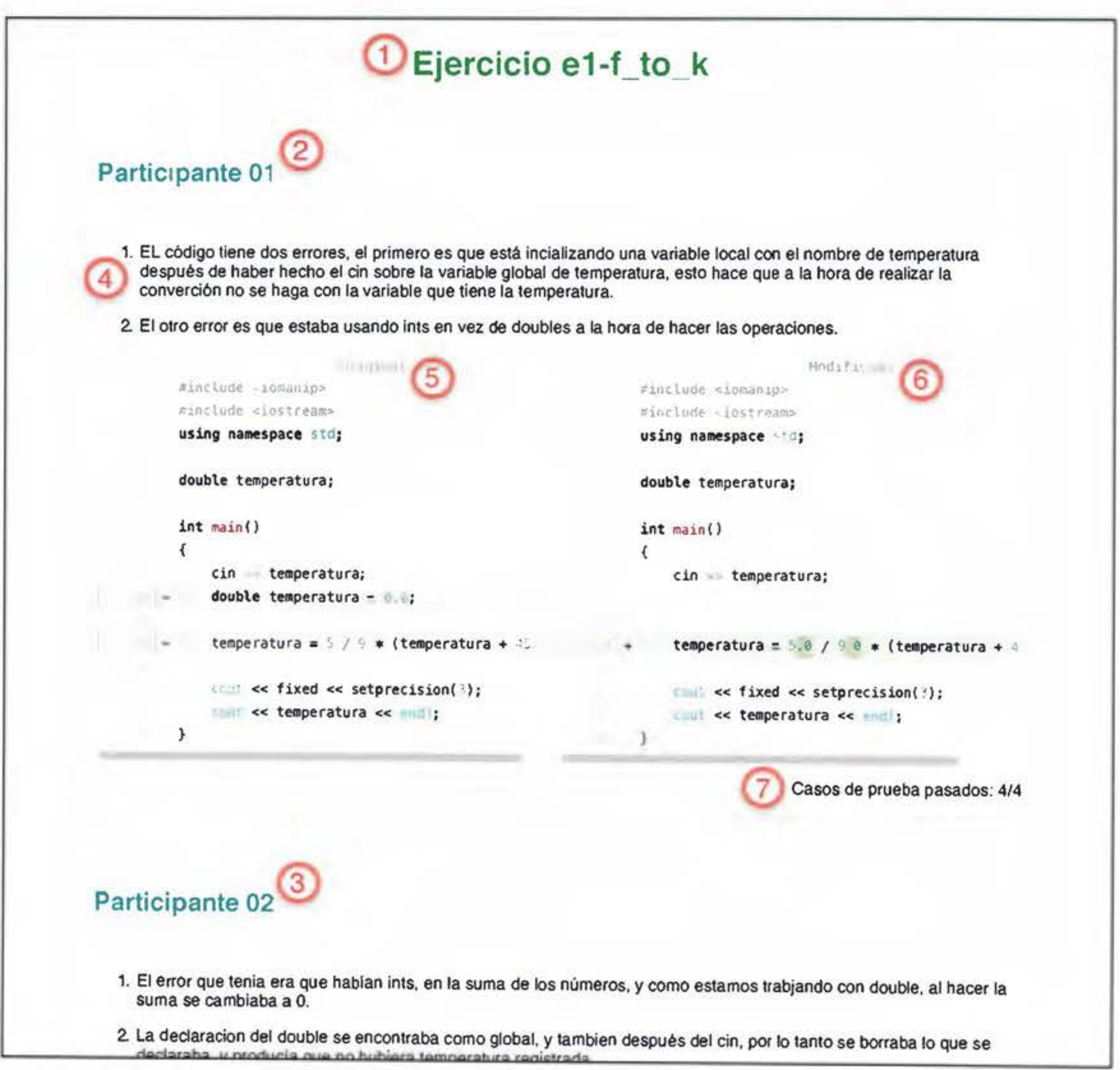



Figura 5.31. Extracto de un ejercicio para revisión de expertos

Los datos de cada participante están en una carpeta identificada con un número. La estructura de directorios y archivos en ella varían de acuerdo a la herramienta usada en el experimento. Si estos archivos se dieran a los expertos, harían explícito el grupo asignado de cada participante (control o tratamiento), por tanto, el guión uniformiza los archivos en las carpetas. Para evitar que los expertos tuvieran que revisar participante por participante, lo cual es menos natural que revisar ejercicio por ejercicio, el guión crea cuatro páginas web, una por cada ejercicio. El nombre del ejercicio es el título de la página como la rotulada ① en la Figura 5.31.

Una página web generada consta de todas las soluciones presentadas por los participantes a un ejercicio. En el extracto de la Figura 5.31 se puede apreciar dos participantes rotulados ② y ③. El orden y los identificadores de los participantes son aleatorizados por el guión, de tal forma que no coinciden para los expertos. El guión incluye en la página web las explicaciones verbales de los errores hechas por el participante, como las marcadas con ④. Estas explicaciones fueron anonimizadas para evitar que permitan identificar al participante o al grupo al que fue asignado (control o tratamiento).

Para ayudar al experto a evaluar las correcciones, el guión incluye el código fuente original del ejercicio, etiquetado con ⑤ en la Figura 5.31, y las modificaciones que el participante le efectuó, etiquetadas con ⑥. El guión resalta con colores las modificaciones entre el archivo original ⑤ y el final en ⑥. Las barras de desplazamiento horizontales están sincronizadas por código *JavaScript*, para comodidad del experto. Finalmente el guión indica la cantidad de casos de prueba que las correcciones del estudiante lograron pasar, rotuladas con ⑦. El guión, de 277 líneas de código, realizó estas tareas concurrentemente en cerca de 3 minutos en una computadora con un procesador de ocho núcleos.

Para la evaluación, a cada experto se le entregó cuatro páginas web similares a la Figura 5.31, una hoja de cálculo, y las carpetas con los archivos normalizados de los estudiantes. La hoja de cálculo tuvo la estructura presentada en el Cuadro 5.9. En las filas estaban los identificadores de los participantes en secuencia de 01 a 35. En las columnas se ubicaban los cuatro ejercicios. Cada ejercicio estaba dividido en la cantidad de errores presentes en el código fuente original dado a los participantes (explicados en la subsección 5.3.1). En total sumaron nueve errores. Para cada error se proveyó dos columnas, una para las explicaciones textuales (J) y otra para las correcciones en código (C), como se aprecia en el Cuadro 5.9.

Cuadro 5.9. Estructura de la hoja de cálculo usada por los expertos

| | 1-1 | | | | 1-2 | | | | 1-3 | | | | | | | | 1-4 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Err1 | | Err2 | | Err3 | | Err4 | | Err5 | | Err6 | | Err7 | | Err8 | | Err9 | |
| **Participante** | J | C | J | C | J | C | J | C | J | C | J | C | J | C | J | C | J | C |
| 01 | | | | | | | | | | | | | | | | | | |
| 02 | | | | | | | | | | | | | | | | | | |
| ... | | | | | | | | | | | | | | | | | | |
| 35 | | | | | | | | | | | | | | | | | | |

Se le pidió a los expertos valorar las explicaciones y las correcciones de los participantes para cada uno de los nueve errores usando una escala Likert entre 0 a 5, cuya semántica se lista en el Cuadro 5.10. El significado de los códigos estaba presente en la hoja de cálculo en forma de comentarios. Cuando hubo discrepancias significativas (tres o más puntos de la escala Likert) entre las valoraciones de los expertos, éstos deliberaron hasta llegar a un acuerdo.

Cuadro 5.10. Escala Likert para calificar las justificaciones y correcciones de los participantes

| | Cód. | Significado |
|---|---|---|
| **Justificación** | 0 | El estudiante no justificó el error |
| | 1 | La justificación no tiene relación con el error |
| | 2 | Da un vago indicio sobre el error |
| | 3 | Mezcla ideas correctas con equivocadas sobre el error |
| | 4 | Tiene una idea válida sobre el error con pocas nociones equivocadas |
| | 5 | Da un claro indicio sobre el error |
| **Corrección** | 0 | No hizo cambios en el código |
| | 1 | Los cambios no tienen relación con el error |
| | 2 | Solución forzada/indebida para pasar los casos de prueba |
| | 3 | Rodea el error con una solución aceptable o con cambios innecesarios |
| | 4 | Corrige el error junto con otros cambios innecesarios, da indicio parcial de dominio |
| | 5 | Corrigió directamente el error, da indicio de dominio |

Cada uno de los dos expertos calificó las soluciones y justificaciones de los estudiantes en una hoja de cálculo como la del Cuadro 5.9. Los datos de ambas hojas de cálculo se consolidaron en una, tras recuperar la identidad inicial de los participantes. Para el análisis de datos se promediaron las calificaciones de ambos profesores para cada participante en cada una de las celdas del Cuadro 5.9.

Se definió la eficacia de detección y corrección de errores de un participante como el promedio de las calificaciones obtenidas en las correcciones hechas al código fuente (columnas "C" del Cuadro 5.9). Una eficacia del 100% indica que el participante logró detectar

los nueve errores y corregirlos directamente. Esta eficacia es la variable respuesta a la pregunta de investigación.

Se realizó un análisis de varianza de un factor para comparar el efecto de la herramienta utilizada en la eficacia de detección y corrección de errores. El resultado reportado por el software estadístico R se encuentra en el Cuadro 5.11. Se encontró sustento estadístico para rechazar la hipótesis nula de igualdad de los efectos de ambos tipos de herramientas. En otras palabras, hubo un efecto significativo de la herramienta en la eficacia de detección y corrección de errores de los participantes (valor p=0.03166, α=0.05). Se bloqueó la variabilidad introducida por el profesor del curso en el análisis de varianza, la cual no resultó tener un efecto significativo.

Cuadro 5.11. Análisis de varianza de las correcciones de errores en código fuente

| Fuente | GL | SC | CM | Valor f | Valor p |
|---|---|---|---|---|---|
| Herramienta | 1 | 745.8 | 745.77 | 5.0802 | 0.03166 * |
| Bloqueo (profesor) | 1 | 322.1 | 322.14 | 2.1944 | 0.14894 |
| Residuos | 30 | 4404.0 | 146.80 | | |

(GL=grados de libertad, SC=suma de cuadrados, CM=cuadrados medios, *p<0.05)

Las estadísticas descriptivas del Cuadro 5.12 permiten ver que en promedio los estudiantes que utilizaron la visualización lúdica de programa (botNeumann++) lograron una eficacia mayor (81.88%) de detección y corrección de errores, que los estudiantes que usaron la herramienta tradicional del curso (depurador visual) para el mismo propósito (72.09%). El gráfico de cajas de la Figura 5.32 confirma visualmente este resultado con los valores medios.

Cuadro 5.12. Estadísticas descriptivas de las correcciones de errores en código fuente

| Herramienta | n | mediana | promedio | desviación estándar | Intervalo de confianza | |
|---|---|---|---|---|---|---|
| | | | | | inferior | superior |
| Depurador visual | 17 | 78.89 | 72.09 | 14.4 | 64.82 | 79.63 |
| botNeumann++ | 16 | 85.00 | 81.88 | 9.69 | 76.57 | 86.90 |

Para verificar el supuesto de normalidad se realizó una prueba de Shapiro-Wilk. No se encontró evidencia estadística para rechazar la hipótesis de que los residuos están normalmente distribuidos (W=0.9558, valor p=0.196). De la misma forma, una prueba de Bartlett no encontró sustento estadístico para rechazar la homogeneidad de varianzas entre el grupo control y el grupo tratamiento (valor p=0.1321).

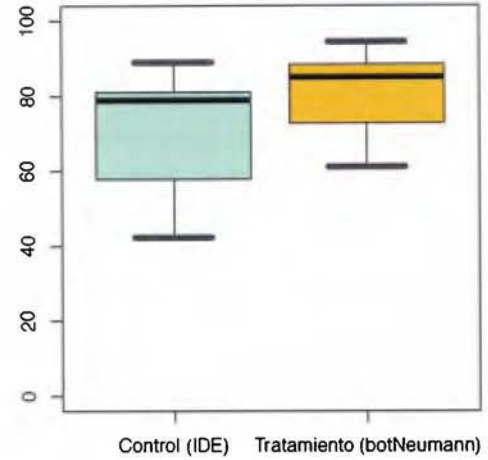



Figura 5.32. Rendimiento por correcciones de los errores en el código fuente

La Figura 5.33 compara la eficacia que los participantes alcanzaron en detectar y corregir errores en el experimento (eje y) contra la nota final que obtuvieron en el curso de "Programación II" (eje x). La gráfica permite ver que los estudiantes de alto y bajo rendimiento en el curso, estuvieron proporcionalmente distribuidos entre los grupos control y tratamiento del experimento.

El análisis de conglomerados identificó tres poblaciones en la Figura 5.33. Los tres grupos presentan el mismo comportamiento: el rendimiento que los estudiantes obtuvieron en el curso se refleja en su eficacia para detectar y corregir errores. Sin embargo, hay una diferencia notoria en los estudiantes que reprobaron el curso, con promedios entre 30 y 60 en el eje x de la Figura 5.33. Aquellos que usaron herramientas tradicionales (depuradores visuales) mostraron una baja eficacia de detección y corrección de errores. Por el contrario, estudiantes que reprobaron el curso pero que utilizaron la visualización lúdica de programa, lograron corregir más del 85% de los errores adecuadamente. Este es un resultado anecdótico que realmente merece investigación futura, y que en caso de confirmarse podría sugerir que las visualizaciones lúdicas de programa serían especialmente útiles para ayudar a estudiantes de bajo rendimiento en los cursos de programación a lograr resultados similares a los estudiantes de alto rendimiento.

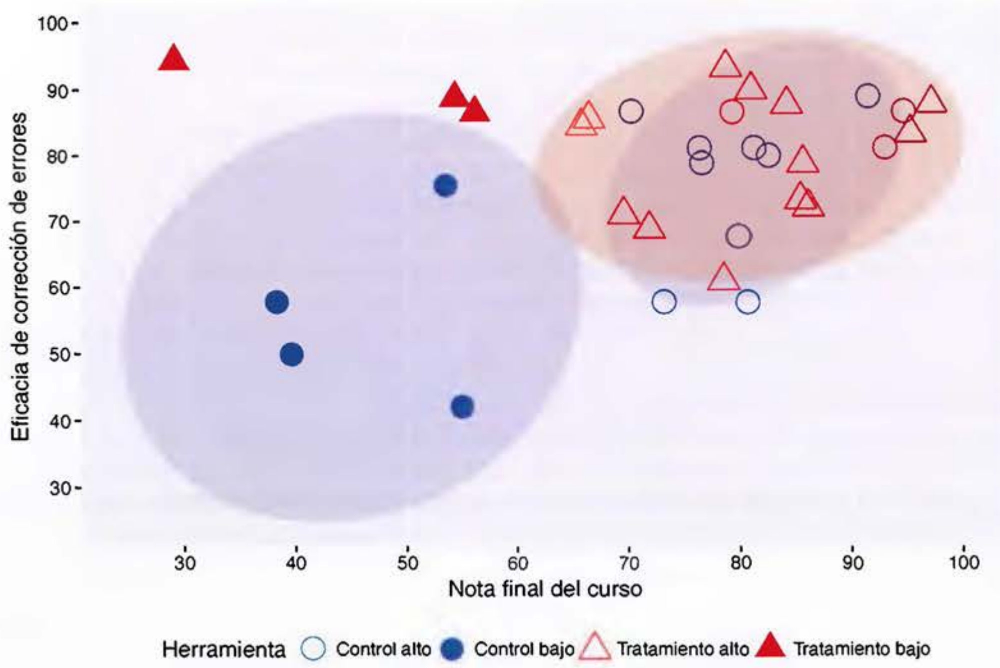



Figura 5.33. Distribución de participantes de acuerdo a su nota del curso (eje x) y su eficacia en detección y corrección de errores en el experimento (eje y)

A los participantes se les solicitó explicar la causa de cada error que corrigieron en el código, y a los expertos se les solicitó valorar esas explicaciones (columnas "J" en el Cuadro 5.9, p.220). Sin embargo, la tasa de no respuesta en las explicaciones fue de un 20%, quizá por olvido de los participantes o por considerar las causas triviales. Es decir, casi la totalidad de los participantes hicieron correcciones en el código fuente para pasar los casos de prueba, pero en uno de cada cinco errores, no proveyeron una justificación del todo. Lastimosamente esta alta tasa de no respuesta invalidaría los resultados de un análisis de varianza. Se intentó imputar las respuestas ausentes, pero los datos resultantes incumplieron el supuesto de normalidad. Por lo tanto, el análisis de las justificaciones debió omitirse por este problema de calidad de datos.

La distribución de la Figura 5.34 muestra que la mayoría de participantes tardaron cerca de tres horas en completar el experimento. Un análisis de varianza de un factor no encontró un efecto significativo de la herramienta en la duración (valor $p=0.67413$, $\alpha=0.05$). Por tanto, la eficiencia de la visualización de programa fue mayor que de las herramientas tradicionales, ya que los participantes en el grupo tratamiento lograron corregir más errores en el mismo tiempo que los participantes en el grupo control.

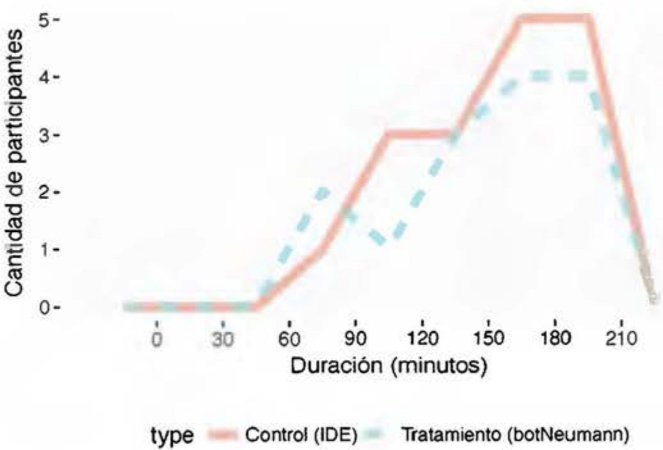



Figura 5.34. Duración de los participantes en el experimento por grupo control y tratamiento

La variabilidad debida al profesor fue bloqueada, y tuvo un efecto significativo en la duración. Como se aprecia en la Figura 5.35, los estudiantes del profesor 1 tardaron más tiempo que los estudiantes del profesor 2 indiferentemente de la herramienta que usaron. Esta diferencia podría explicarse porque el profesor 2 emplea un juez automático en línea como parte de la metodología su curso. Por tanto, sus estudiantes tienen una mayor familiaridad con los ejercicios en este formato.

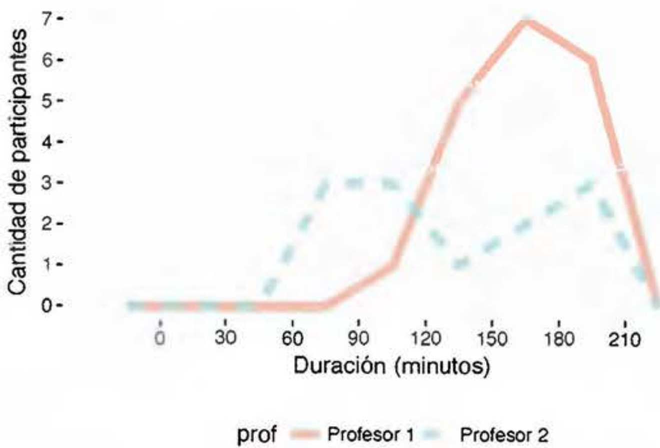



Figura 5.35. Duración de los participantes en el experimento de acuerdo al profesor

#### *5.3.3.2 Evaluación de usabilidad*

Cuando un participante terminaba el experimento, respondía el cuestionario *SUS* de la Figura 5.24 (p.199) en formato digital. El experimentador se aseguró que cada participante hubiese

respondido el cuestionario, en el momento de copiar los datos generados por el participante por los cuales recibiría crédito en el curso de "Programación II". De esta forma, el cuestionario *SUS* fue respondido por los 35 participantes, incluso por los dos que no completaron los ejercicios. De los 35 participantes, 17 usaron el prototipo C++ de botNeumann++ y 18 usaron depuradores visuales, por lo que cada grupo supera el mínimo de 10 evaluaciones para considerar confiables los resultados del *SUS*.

En promedio los 17 estudiantes del grupo tratamiento consideraron que el prototipo C++ de botNeumann++ tiene una usabilidad de 76.47. La interpretación de este resultado es que los estudiantes de "Programación II" consideraron que el prototipo C++ tiene una "excelente" usabilidad de acuerdo a las categorías de [Bangor et al. 2009]. Los 18 estudiantes del grupo control consideraron que en promedio los depuradores visuales tienen una usabilidad de 69.34, cuya categoría es de una "buena usabilidad" de acuerdo a [Bangor et al. 2009].

Si se analizan los dos depuradores visuales por separado, la usabilidad promedio de *CODE::BLOCKS* fue de 72.75 y la de *QTCREATOR* fue de 65.94. Ambas herramientas se mantienen en el rango de "buena" usabilidad de acuerdo a las categorías de [Bangor et al. 2009]. Sin embargo, *QTCREATOR* fue evaluado por ocho participantes, por lo que no alcanza el mínimo para considerarse una medición confiable. El histograma de la Figura 5.36 muestra en el eje-x los valores de usabilidad en rangos de 10, y en el eje-y la cantidad de participantes que valoraron en cada rango a las herramientas. Puede notarse que el prototipo de la visualización de programa fue considerada la herramienta más usable, seguida de *CODE::BLOCKS*, y en último lugar *QTCREATOR*.

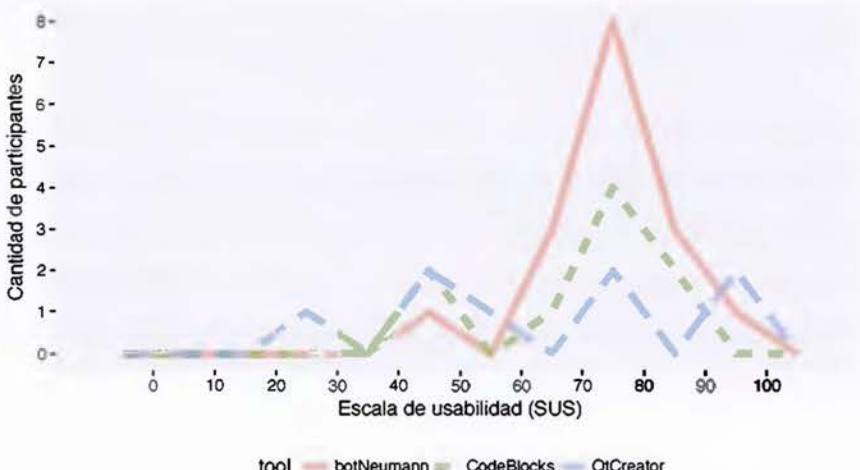



Figura 5.36. Comparación de la usabilidad entre el prototipo C++ y depuradores visuales

Los participantes reportaron 57 amenidades en respuesta a la pregunta 11 del cuestionario (Figura 5.24, p.199). El Cuadro 5.13 agrupa estas amenidades en seis categorías. A la derecha de cada categoría se incluye un gráfico proporcional al número de amenidades reportadas sobre el prototipo C++ de botNeumann++ ("bN") y sobre los depuradores visuales (DV) usados en el experimento. En general los participantes consideraron que ambos tipos de herramientas les permiten observar y entender lo que ocurre en sus programas, además de ser fáciles de usar. A modo de diferencia, los participantes percibieron los depuradores visuales como útiles, mientras que botNeumann++ destacó en amenidades diversas, relacionadas con los jueces automáticos, su capacidad de controlar la visualización, y el diseño elaborado de su interfaz, entre otros.

Cuadro 5.13. Amenidades reportadas de botNeumann++ (bN) y depuradores visuales (DV)

| Categorías de amenidades | bN | DV | Total |
|---|---|---|---|
| *Visualización.* Permite observar el comportamiento del programa en tiempo real, en especial el estado de sus variables, lo cual sería invisible de otro modo. | 8 | 8 | 16 |
| *Utilidad.* Permite encontrar y corregir errores en los programas. | 4 | 10 | 14 |
| *Facilidad de uso.* La herramienta es fácil de usar. Facilitó alcanzar el objetivo de la sesión. Es fácil de comprender. | 5 | 6 | 11 |
| *Juez automático.* Saber rápidamente cuáles casos de prueba habían fallado y estudiar su entrada y salida. | 4 | 1 | 5 |
| *Interacción.* Poder controlar la visualización y su velocidad. | 4 | 0 | 4 |
| *Otras.* Es divertida. Hace su trabajo correctamente. Interfaz gráfica muy bien diseñada. La tarea propuesta en el experimento (corregir errores en código de otros) es muy útil para el futuro. | 5 | 2 | 7 |

Al finalizar el curso, los estudiantes de "Programación II" ya tienen sistemas de conceptos creados sobre cómo funciona la máquina nocional de C++. En una escala de 1 a 5, donde 1 es nada útil y 5 muy útil, los estudiantes consideraron en promedio que el prototipo C++ de botNeumann++ es "bastante útil" (4.13) para comprender la máquina nocional, mientras que los depuradores visuales como "algo útil" (3.88) para el mismo fin. La Figura 5.37 refleja esta percepción de utilidad por los participantes.

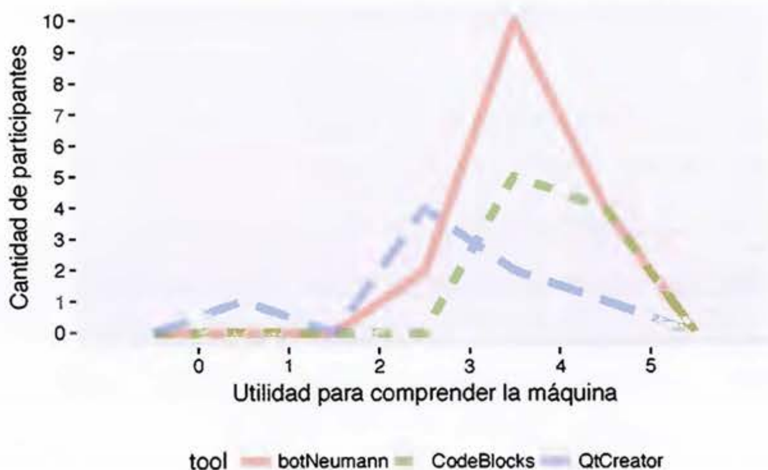



Figura 5.37. Utilidad percibida de la herramienta para comprender la máquina

El Cuadro 5.14 lista los problemas de usabilidad de mayor nivel de severidad reportados por los participantes. Ninguno de ellos fue de alta severidad, dado que no impidieron a los estudiantes completar las cuatro tareas propuestas en el experimento. El problema más reportado fue a nivel de los manuales, considerados poco claros para las tres herramientas. Se sugiere adaptarlos para esta población en futuras evaluaciones. En el caso de botNeumann++ se sugiere que este manual sea parte del tutorial del juego en implementaciones futuras.

Cuadro 5.14. Problemas más severos de usabilidad de las tres herramientas en el experimento

| # | **Problema y recomendaciones** | bN | CB | Qt | |
|---|---|---|---|---|---|
| 1 | Instrucciones poco claras, insuficientes, o muy extensas. Se sugiere mejorar y probar los manuales, y proveer mas ayuda en la herramienta misma. Integrar la herramienta en los cursos de programación. | 4 | 5 | 3 | 12 |
| 2 | Visualización ilegible, la herramienta presenta códigos hexadecimales o valores de variables difíciles de leer, o en espacios muy pequeños. | 4 | | | 6 |
| 3 | El control de los casos de prueba no es intuitivo. Investigar con esta población diferentes diseños de interfaz. Indicar en la salida de *MAKE* el número del caso de prueba. Explicar qué son los casos de prueba generados. Poder omitir las animaciones si sólo se quiere probar. | 3 | 1 | | 4 |
| 4 | Herramienta difícil o complicada de usar con muchas opciones (*QTCREATOR*). Explicar opciones avanzadas con un tutorial opcional. | | | 3 | 3 |
| 5 | Lentitud de *QTCREATOR*. A veces no inicia la depuración o la hace muy lenta. Identificar la causa del problema, puede deberse a un defecto de instalación o configuración en el laboratorio usado. | | | 2 | 2 |

R=reportes, S=severidad, bN=*BOTNEUMANN++*, CB=*CODE::BLOCKS*, Qt=*QTCREATOR*

El segundo problema de usabilidad más reportado fue la presencia de códigos ilegibles (en hexadecimal) en valores de variables, o valores incompletos debido al poco espacio de la ventana. El tercer problema más reportado afecta al prototipo C++ de botNeumann++, dado que para algunos estudiantes el control de los casos de prueba no fue intuitivo.

Los dos problemas siguientes afectaron a *QTCREATOR* y pueden explicar por qué fue la herramienta con la menor valoración de usabilidad. Los estudiantes consideraron esta herramienta muy compleja de usar. Además, reportaron que *QTCREATOR* a veces no iniciaba el proceso de depuración o lo hacía de forma muy lenta. Este comportamiento podría deberse a un defecto de instalación o de configuración en el laboratorio de computadoras utilizado para el experimento.

### 5.3.4 Discusión del experimento

El objetivo ingenieril de esta tesis ha sido "mejorar las visualizaciones de programa, mediante la introducción de otros constructos computacionales que satisfagan requerimientos de software de alto nivel derivados de una teoría de aprendizaje, con el fin de ayudar a estudiantes universitarios de programación a comprender la máquina nocional de un lenguaje de programación" (p. 8). Se propuso que los requerimientos podrían satisfacerse con alegorías visuales y ludificación, además de aspectos de un juez automático. A la propuesta conceptual resultante se le llamó visualización lúdica de programa y se diseñaron dos de ellas: *PUPPETEER++* y botNeumann++. De la segunda se implementaron dos prototipos, uno en *POWERPOINT* y otro en C++. El prototipo C++ se utilizó para responder a la pregunta de investigación.

La pregunta de investigación de esta tesis pretende determinar si las visualizaciones lúdicas de programa son más eficaces que herramientas tradicionales para ayudar a estudiantes a comprender la máquina nocional de un lenguaje de programación. La comprensión es un fenómeno psicológico que no se evalúa directamente, si no a través de evidencia indirecta. Dado que la comprensión es un fenómeno cognitivo necesario para poder corregir errores en un programa (en inglés, *SOFTWARE BUG*) [McCauley et al. 2008, p.76], se trató en el experimento de evaluarla mediante la detección y corrección de errores en programas (depuración).

La variable respuesta fueron las correcciones hechas por los participantes a errores presentes en programas de C++. Los errores escogidos fueron de carácter semántico o lógico, que deben ser estudiados en tiempo de ejecución. Este tipo de errores se consideran más difíciles de detectar [Denny et al. 2014; McCauley et al. 2008]. Se escogieron con el fin de ameritar el uso de una herramienta que ayude a comprender el comportamiento en tiempo de ejecución de los programas.

La herramienta se asignó al azar a los participantes. A los del grupo control se les asignó un depurador visual, que es parte del ambiente de desarrollo al que estaban habituados en el curso. A los del grupo tratamiento se les asignó el prototipo C++ de la visualización de programa botNeumann++. Se les plantearon los mismos programas en C++ con errores a ambos grupos y se les pidió estudiarlos con la herramienta asignada para detectar y corregir los errores presentes en ellos.

Los resultados del experimento mostraron que hubo un efecto estadísticamente significativo de la herramienta en la eficacia de corrección de errores, y de forma indirecta, en la comprensión de los programas. Los participantes que utilizaron la visualización lúdica de programa alcanzaron una mayor eficacia que aquellos que continuaron usando el ambiente de desarrollo que conocían en el curso. La presencia de este efecto responde positivamente a la pregunta científica de esta tesis, confirmando que las visualizaciones lúdicas de programa tienen una mayor eficacia que herramientas tradicionales para ayudar a estudiantes a comprender cómo los programas son ejecutados por una máquina nocional.

Mediante un análisis de conglomerados se encontró que, la eficacia en corregir los errores planteados en el experimento fue proporcional al rendimiento de los participantes en el curso de "Programación II". Es decir, los estudiantes que tuvieron un alto rendimiento en el curso alcanzaron una alta eficacia de corrección de errores en el experimento, y los que tuvieron un bajo rendimiento en el curso mostraron una baja eficacia de corrección de errores. Sin embargo, hubo una excepción. Los estudiantes de bajo rendimiento, quienes reprobaron el curso, tuvieron una baja eficacia de corrección de errores en el experimento si continuaron utilizando el ambiente de desarrollo del curso. Pero, los estudiantes de bajo rendimiento que utilizaron el prototipo de botNeumann++ alcanzaron una eficacia de corrección de errores superior al 85%, mejor incluso que el promedio de participantes que aprobaron el curso. Este es un resultado sumamente interesante, que no es reportado en ningún estudio analizado en

la revisión de literatura de este trabajo, y que merece investigación futura para determinar si es un efecto generalizable. En caso de serlo, sería el aporte ingenieril más importante de las visualizaciones lúdicas de programa al contexto del aprendizaje de la programación.

Dado que botNeumann++ era una herramienta nueva para los participantes, mientras que el ambiente de desarrollo era una herramienta conocida, se esperaba que la duración en corregir los programas fuera mayor para los participantes del grupo tratamiento que el grupo control debido a la curva de aprendizaje. Este efecto no se confirmó. En promedio los estudiantes de ambos grupos lograron resolver los ejercicios cerca de una hora con cuarenta minutos. Por consiguiente, la eficiencia de la visualización de programa fue superior a las herramientas tradicionales, dado que los participantes lograron una mayor tasa de errores corregidos en aproximadamente el mismo tiempo que el grupo control.

En cuanto a las valoraciones de usabilidad, el prototipo de botNeumann++ fue la herramienta considerada como más usable, seguida por *CODE::BLOCKS* y finalmente por *QTCREATOR*. Este es un resultado interesante, ya que botNeumann++ era una herramienta desconocida, mientras que los ambientes de desarrollo integrados eran herramientas familiares para los participantes. Estas valoraciones podrían indicar que las visualizaciones de programa son herramientas más aptas para estudiantes de la programación que las herramientas industriales más ampliamente empleadas en ambientes de aprendizaje. Este resultado se reafirmó cuantitativamente usando una escala Likert. Los participantes percibieron una utilidad ligeramente mayor de botNeumann++ para comprender la máquina nocional que los depuradores visuales.

El análisis cualitativo de las amenidades encontró que los estudiantes destacaron la utilidad de los depuradores visuales, posiblemente al proyectarlas como herramientas industriales que usarán en su vida profesional, mientras que de botNeumann++ resaltaron amenidades diversas relacionadas con su funcionalidad de juez automático, control de la visualización, y su diseño elaborado. El error de usabilidad más reportado fue sobre los tutoriales de las tres herramientas, los cuales se trataron de mantener abreviados para lograr una sesión experimental cercana a tres horas. Los estudiantes alertaron sobre valores de variables ilegibles en las tres herramientas, control no intuitivo de los casos de prueba en botNeumann++, y problemas de *QTCREATOR* que pueden explicar su menor grado de

usabilidad. Estos reportes son motivadores de investigación futura para las visualizaciones de programa, y recomendaciones para los docentes que utilizan depuradores visuales en sus cursos.

La usabilidad del prototipo C++ de botNeumann++ fue evaluada dos veces. Estudiantes del curso "Introducción a la computación" consideraron que tenía una "buena usabilidad" con un promedio de 70.68. Por su parte, estudiantes del curso "Programación II" la consideraron "excelente" con un promedio de 76.47. Una explicación científica de este incremento sobrepasa los objetivos de esta investigación. El cambio más sobresaliente en la herramienta fue la dificultad de los ejercicios, los cuales pasaron de problemas triviales a programas con errores difíciles de detectar. Un incremento en la dificultad de la tarea no es suficiente para explicar un incremento en la usabilidad. Factores externos a la herramienta, como la experiencia de programación o el uso de herramientas tradicionales, podrían explicar mejor este incremento.

# 6 CONCLUSIONES

Los resultados de los objetivos de esta investigación se han presentado y discutido a lo largo de los capítulos de este documento por conveniencia de presentación. En este capítulo se hace una discusión general de estos resultados (sección 6.1), se listan los aportes al conocimiento hechos por esta investigación (sección 6.2), se sugieren ideas de investigación futura que ayudarían a comprender mejor el problema atendido en este trabajo (sección 6.3), y finalmente se listan las publicaciones científicas realizadas (sección 6.4).

## 6.1 Discusión general

La motivación que inició esta investigación fue la experiencia del autor como docente del curso "Programación II", al confirmar que los métodos tradicionales de enseñanza resultan insuficientes para que los estudiantes comprendan cómo los programas son ejecutados por las computadoras [Hidalgo-Céspedes et al. 2016b]. Al consultar sobre este problema en la literatura científica, se encontró que los investigadores critican, además, la eficiencia de los materiales estáticos ampliamente usados para explicar cómo las computadoras ejecutan los programas [Sorva et al. 2013, p.4], y en respuesta proponen el uso de visualizaciones de programa como una solución [Sorva 2012]. Sin embargo, la efectividad de estas herramientas es incierta y su uso es raro en los ambientes de enseñanza-aprendizaje [Sorva et al. 2013; Naps et al. 2003; Isohanni and Järvinen 2014]. Por tanto, se determinó que existía una necesidad de conocimiento que permita construir herramientas *eficaces* para la visualización de programas. Aportar a este conocimiento fue el objetivo general de esta tesis.

Siguiendo la metodología de la ciencia del diseño, el autor se propuso el objetivo ingenieril de hacer mejoras en las visualizaciones de programa, y el objetivo científico de evaluar si esas mejoras a las visualizaciones de programa influyen en una mayor eficacia para ayudar a los estudiantes a comprender cómo los programas son ejecutados por una máquina nocional. Para conseguir estos objetivos generales, se plantearon cuatro objetivos específicos que se discuten en las siguientes subsecciones.

### 6.1.1 Requerimientos de software

Dado que las visualizaciones de programa son herramientas de aprendizaje, esta investigación conjeturó que si se aplican las recomendaciones de una teoría de aprendizaje sobre la mediación de herramientas, se podría incrementar su eficacia para ayudar a estudiantes a comprender máquinas nocionales. Por tanto, el primer objetivo de la investigación fue identificar requerimientos de software de alto nivel, a partir de una teoría de aprendizaje, que no son atendidos por las visualizaciones de programa existentes. Mediante un razonamiento deductivo se infirió una lista de 16 requerimientos de software de alto nivel a partir de la teoría de aprendizaje del constructivismo sociocultural.

Los requerimientos inferidos podrían estar ya satisfechos por las visualizaciones de programa existentes. Para poder determinar en qué medida los requerimientos eran satisfechos ya, se necesitó primero conocer la lista de visualizaciones de programa. La única revisión de literatura exhaustiva afín reportó 46 herramientas de visualización entre 1979 y 2012 [Sorva et al. 2013]. El autor de esta tesis realizó una revisión sistemática de literatura para actualizar esta lista hasta el año 2016. Se encontraron siete nuevas visualizaciones de programa, para sumar 53 herramientas.

Se realizó la verificación de software con las herramientas que estuvieron disponibles para descarga y que pudieron ser ejecutadas. Catorce herramientas cumplieron este requisito. Se encontró que sólo dos de los 16 requerimientos estaban medianamente satisfechos por las visualizaciones disponibles de programa. El primero de ellos era satisfecho porque son herramientas interactivas que promueven un estado activo donde el usuario escribe código, provee parámetros de ejecución, o controla la ejecución de la visualización. Un segundo requerimiento era satisfecho porque proveen realimentación inmediata sobre el estado del programa mientras se ejecuta en la máquina nocional.

Seis requerimientos eran satisfechos en alguna medida. Es decir, pocas visualizaciones de programa estimulaban el interés del usuario por aprender, contraponían conceptualmente las nociones, asociaban visualmente los nuevos conceptos con nociones adquiridas en la experiencia de vida, enseñaban y evaluaban más el proceso que el producto final, ayudaban a construir sistemas de conceptos, y organizaban los conceptos para que reflejaran relaciones naturales.

Los restantes ocho requerimientos no eran satisfechos por las visualizaciones disponibles. Es decir, ninguna visualización investigaba qué motivaba a los estudiantes a aprender, evaluaba nociones o emociones previas, comparaba conceptos nuevos contra conocidos, ejercitaba un nuevo concepto aplicándolo a varios contextos, resolvía problemas reales que requirieran la aplicación del nuevo concepto hasta que los estudiantes desarrollaran un hábito, evaluaba que las nociones y habilidades de los estudiantes fueran estables en el tiempo, y fomentaba a que los estudiantes transfirieran este conocimiento a nuevas situaciones.

La primera conclusión a la que llegó esta investigación fue que las visualizaciones de programa tradicionales poco atienden recomendaciones de una teoría de aprendizaje constructivista que podrían influir en su eficacia como herramientas de aprendizaje. Este resultado abre una puerta de oportunidades para investigación futura.

### 6.1.2 Propuesta conceptual

Conociendo que la mayoría de requerimientos no ha sido satisfecha por las visualizaciones de programa disponibles, esta investigación procedió a ingeniar teóricamente cómo lograrlo. Mediante un proceso de abducción [Vaishnavi and Kuechler Jr. 2015, p.18] se conjeturó que los requerimientos se podían satisfacer con el uso de alegorías visuales concretas en lugar de metáforas abstractas inconexas, y el uso de ludificación.

Antes de formular una propuesta conceptual, se verificó en la revisión de literatura el soporte de alegorías visuales y ludificación por parte de las visualizaciones de programa disponibles. Se encontró que sólo una herramienta, *JELIOT CONAN*, ha sido utilizada en ambientes lúdicos, donde equipos compiten por encontrar errores en animaciones conflictivas. Sin embargo, cabe destacar que la ludificación ocurre en el contexto de uso de *JELIOT CONAN*, más que en la implementación en la herramienta misma de los elementos lúdicos sugeridos por [Kapp 2012].

En cuanto a las alegorías visuales, la revisión de literatura encontró que no eran utilizadas del todo por las visualizaciones de programa disponibles. Por el contrario, en lo que respecta al ámbito de las metáforas, las visualizaciones de programa existentes empleaban metáforas extendidas inconexas para representar visualmente los conceptos de programación. En

cuanto al origen de las metáforas, todos los sistemas, a excepción de *PLANANI* y *METAPHOR-BASED OO VISUALIZER*, usaban metáforas abstractas para representar los conceptos abstractos de programación. Sin embargo, existe la tendencia histórica opuesta a representar lo abstracto con lo concreto [Smith et al. 1981], ya que ayuda a las personas a comprender lo desconocido con lo familiar [Lakoff 1993; Luria et al. 2011], como debería ser el caso de los conceptos computacionales.

La carencia de ludificación y de alegorías visuales concretas en las visualizaciones de programa existentes, generó mucho interés de investigar si éstas tienen un efecto positivo en la eficacia de herramientas para ayudar a los estudiantes a comprender máquinas nocionales. Pese a que la revisión de literatura no encontró visualizaciones de programa que implementaran alegorías visuales, se detectaron advertencias de efectos negativos de alegorías en interfaces de usuario de otros tipos de herramientas [Hsu 2006; Blackwell 2006]. Por tanto, antes de continuar con la investigación, se determinó vital confirmar estas advertencias en otros estudios y en los estudiantes de la Escuela de Ciencias de la Computación e Informática. En caso de confirmarse, se tendría que descartar la propuesta conceptual de esta tesis.

Para confirmar las advertencias detectadas, se realizó una segunda revisión de literatura y se condujo un experimento. La revisión de literatura recabó trabajos sobre alegorías en el campo de la computación, no limitados a visualizaciones de programa. Gracias a que casi la totalidad de trabajos encontrados realizaron comparaciones experimentales, se pudo contar la cantidad de variables de respuesta que encontraron efectos a favor de las alegorías y de las metáforas tradicionales. De las 55 variables medidas, el 73% no detectó diferencias significativas entre las alegorías y las metáforas, mientras que un 22% encontró que las alegorías influyen en mejores resultados que las metáforas inconexas. Por tanto, los resultados de la revisión de literatura no sustentaron las advertencias de efectos negativos reportados por [Hsu 2006; Blackwell 2006], sino que en computación, las metáforas y alegorías tienen prácticamente los mismos efectos, y en algunas situaciones las alegorías superan las metáforas tradicionales, y muy rara vez lo opuesto.

Para determinar si las advertencias negativas de las alegorías o los resultados de la revisión de literatura se reflejan en los estudiantes de la ECCI, se condujo un experimento que comparó el efecto de la enseñanza con alegorías y con metáforas tradicionales en el rendimiento de

resolución de problemas de estudiantes de "Programación I" y en la percepción motivacional de esta enseñanza. Los resultados encontraron que los estudiantes tuvieron el mismo rendimiento y motivación, indiferentemente de si las nociones fueron explicadas con alegorías o metáforas tradicionales. Por tanto, se confirmó el resultado más reportado en la literatura científica también en estudiantes de la ECCI, en lugar de las advertencias negativas. Como un aporte adicional, se encontró que la igualdad de efectos entre alegorías y metáforas se mantiene en sus distintas modalidades (visual, textual, oral, o multimodal). Como un segundo aporte, se confirmó que las alegorías son más verbosas [Hsu 2007] y difíciles de presentar que las metáforas tradicionales [Blackwell 2001], pero estos rasgos no afectaron la percepción motivacional de los estudiantes.

La falta de evidencia empírica, en la literatura científica de computación y en los estudiantes de la ECCI, de efectos negativos de las alegorías, permitió continuar con el segundo objetivo específico de la investigación. Este objetivo propuso cambios conceptuales a las visualizaciones de programa para satisfacer los requerimientos de software de alto nivel, dado que no son atendidos por las visualizaciones tradicionales de programa.

La propuesta conceptual hecha por esta tesis establece la alegoría visual como el principal mecanismo para abstraer la realidad, es decir, para representar la máquina nocional. La alegoría sustenta los once elementos lúdicos restantes porque provee las abstracciones a las que serán aplicados los elementos. La propuesta sugiere que la elección de la alegoría visual sea la primera tarea de diseño de una visualización lúdica de programa. Además se sugiere que las entidades visuales sean concretas debido a la naturaleza abstracta de los conceptos computacionales, y que tengan características comunes con ellos para incrementar su éxito, de acuerdo a las recomendaciones encontradas en la literatura [Smith et al. 1981; Lakoff 1993; Smilowitz 1995]. El mayor cuerpo de la propuesta conceptual es el mapeo entre los requerimientos de software y los elementos lúdicos que pueden satisfacerlos, los cuales se explicaron junto con dos diseños elaborados en el próximo objetivo.

### 6.1.3 Diseño de una solución

Dado que ninguna visualización de programa existente implementaba alegorías visuales ni ludificación, surgió naturalmente la pregunta ¿serán las visualizaciones lúdicas de programa

más eficaces que herramientas tradicionales para comprender la máquina nocional de un lenguaje de programación? La hipótesis de acuerdo a las teorías en que se basa la propuesta conceptual es que lo son. Sin embargo, se requieren datos empíricos para poder confirmar o rechazar esta hipótesis. Por tanto, el tercer objetivo específico fue diseñar una visualización lúdica de programa para recabar evidencia que permitiera confirmar o rechazar empíricamente la hipótesis de investigación.

El diseño de la visualización lúdica de programa se hizo pensando en su contexto de uso, que fueron los cursos de programación de la ECCI, y los estudiantes de estos cursos fueron sus usuarios finales. Mediante una encuesta se encontró que C++ es el lenguaje de programación más usado por los estudiantes durante la carrera, y que los conceptos de concurrencia y administración de memoria fueron considerados como los más difíciles y relevantes de comprender.

Se diseñó una visualización lúdica de programa para C++, ideada para los conceptos de concurrencia y administración de memoria. Se utilizó una alegoría de un teatro de títeres, donde los titiriteros son hilos de ejecución y el escenario es la memoria dinámica. Elementos de ludificación fueron aplicados, similares a los de un videojuego casual. El modelo resultante, llamado *PUPPETEER++*, fue validado con expertos (profesores de programación) tratando de predecir su interacción con el contexto. Los expertos se mostraron divididos en cuanto a sus posiciones, lo que impidió predecir una interacción efectiva. Sin embargo, las debilidades reportadas fueron más numerosas y enfáticas que las fortalezas señaladas. Por tanto, se prefirió hacer una segunda iteración del ciclo de diseño con una nueva alegoría.

Se diseñó una segunda visualización de programa usando robots en una fábrica futurista como alegoría visual. Se aplicaron elementos de ludificación de forma similar al modelo previo. Al diseño resultante se le llamó botNeumann++ y fue validado mediante entrevistas con dos investigadoras de *CARNEGIE MELLON UNIVERSITY*. Las expertas sugirieron mejoras al diseño, y desarrollar prototipos incrementales debido a la complejidad de la herramienta. Sin embargo, fueron más numerosas y enfáticas las opiniones positivas sobre su interacción con los usuarios finales. Dado el éxito con que se validó el modelo botNeumann++, se procedió a implementar un prototipo para evaluar su usabilidad y eficacia experimentalmente

### 6.1.4 Evaluación de un prototipo

Como un prototipo es requerido para poder recabar evidencia empírica que rechazara o confirmara la hipótesis de investigación, y siguiendo la recomendación de crear prototipos incrementales, se implementó primero un prototipo en *MICROSOFT POWERPOINT*. Esta herramienta de ofimática permitió crear un prototipo en un tiempo considerablemente más corto que C++, y facilitó el posicionamiento de los elementos gráficos y la construcción de las animaciones que luego se replicarían en C++. Sin embargo, la imposibilidad de invocar herramientas de compilación y depuración, limitó la natural edición de código y se tuvo que recurrir a un protocolo de mago de oz para responder a los errores de código introducidos por los participantes.

Se recomienda que la usabilidad de una visualización de programa se evalúe por ser un sistema interactivo [Urquiza-Fuentes and Velázquez-Iturbide 2009, p.9:2-3]. botNeumann++ es probablemente la primera visualización de programa cuya usabilidad se evalúa con tres poblaciones: usuarios finales (estudiantes), expertos en el dominio (profesores), y expertos en usabilidad (*HCI*). Se confirmó que estas tres poblaciones se interesan en aspectos diferentes del software, lo que sustenta la teoría del efecto evaluador. Por tanto, se sugiere que para tener una mayor cobertura temática de los problemas de usabilidad, conviene evaluar con grupos de distinta naturaleza.

El prototipo PowerPoint se diseñó con dos tareas en mente. La primera tarea fue de familiarización con la herramienta. La segunda tarea requirió el uso de memoria dinámica. Dado que la fuga de memoria es uno de los problemas más difíciles de abordar en la enseñanza y aprendizaje de C++ en la ECCI [Hidalgo-Céspedes et al. 2016b, p.14], se prepararon animaciones para visualizar este problema. Mediante observación se encontró que el prototipo PowerPoint fue eficaz como herramienta para detectar fugas de memoria, ya que cerca del 95% de las veces que los participantes advirtieron este error fue gracias a una animación del prototipo. Este es un aporte valioso a la eficacia de las visualizaciones de programa, en especial al considerar los cuestionamientos hechos a la eficacia de estas herramientas en la literatura científica que motivaron esta investigación.

Ninguno de los problemas de usabilidad reportado fue de alta severidad, por lo que se consideró el prototipo PowerPoint como usable. Los problemas de usabilidad más prioritarios sirvieron para ajustar el segundo prototipo, implementado en C++.

La implementación del prototipo C++ generó muchas lecciones aprendidas, en especial por la carencia de una infraestructura de depuración para C/C++. Se tuvo que crear una enorme capa de código para suplir esta carencia a través de un depurador tradicional. La usabilidad del prototipo C++ fue evaluada por estudiantes que estaban por iniciar su aprendizaje de la programación, quienes la consideraron como "buena" de acuerdo a la interpretación del promedio *SUS* de 70.68. Se detectaron y corrigieron dos problemas severos de usabilidad antes de realizar la evaluación experimental.

Responder la pregunta de investigación, sobre determinar si las visualizaciones lúdicas de programa son más eficaces para ayudar a comprender una máquina nocional que herramientas tradicionales, fue el objetivo específico 4. Se comparó mediante un experimento la eficacia del prototipo (tratamiento) contra herramientas usadas habitualmente en los cursos de programación para el mismo fin (control). Los participantes fueron estudiantes del curso "Programación II" a quienes se les pidió usar la herramienta asignada para detectar y corregir errores sutiles en cuatro programas en C++. Se escogió esta tarea porque la comprensión de los programas es ineludible para poder corregir errores en ellos.

Se encontró un efecto estadísticamente significativo de la herramienta en la eficacia de corrección de errores de los participantes, y de forma indirecta, en la comprensión de los programas. Los participantes que usaron la visualización lúdica de programa corrigieron satisfactoriamente más errores que aquellos que continuaron usando el ambiente de desarrollo que conocían en el curso. La presencia de este efecto responde positivamente a la pregunta de investigación de esta tesis, confirmando que las visualizaciones lúdicas de programa tuvieron una mayor eficacia que herramientas tradicionales para ayudar a estudiantes a comprender cómo los programas son ejecutados por una máquina nocional.

Se encontró además que los estudiantes de bajo rendimiento, quienes reprobaron el curso de "Programación II", tuvieron una baja eficacia de corrección de errores en el experimento si continuaron utilizando el ambiente de desarrollo del curso, pero una eficacia superior a los que aprobaron el curso si utilizaron el prototipo de botNeumann++. Ningún trabajo revisado en la literatura reporta un resultado similar. Si investigación futura logra determinar que es un efecto generalizable de las visualizaciones lúdicas de programa, sería probablemente el aporte ingenieril más importante al contexto del aprendizaje de la programación.

Dado que los participantes corrigieron una mayor tasa de errores con el prototipo C++ en el mismo tiempo que los estudiantes con herramientas tradicionales, la eficiencia de la visualización lúdica de programa fue mayor. Pese que el prototipo era una herramienta desconocida para los participantes, su usabilidad fue considerada "excelente" de acuerdo al promedio 76.47 en la escala *SUS*, mientras que los ambientes de desarrollo tradicionales fueron considerados como "buenos" con un promedio de 69.34. Estas diferencias perfilan a las visualizaciones lúdicas de programa como aplicaciones más apropiadas para estudiantes que las herramientas usadas convencionalmente en la industria.

En síntesis, los positivos resultados obtenidos sobre las visualizaciones lúdicas de programa apoyan la propuesta conceptual de que las teorías escogidas para satisfacer los requerimientos, alegorías visuales concretas y ludificación, ayudan a la construcción de visualizaciones de programa más eficaces.

## 6.2 Compendio de los aportes al conocimiento

Los principales aportes de esta investigación al conocimiento científico e ingenieril sobre las visualizaciones de programa son los siguientes:

1. Se actualizó la única revisión de literatura exhaustiva sobre visualizaciones de programa hasta el año 2016. Se encontraron y analizaron siete nuevas herramientas en el período 2012 a 2016.
2. Se encontró que las visualizaciones de programa tradicionales poco satisfacen recomendaciones de una teoría de aprendizaje constructivista sobre la mediación de estas herramientas en los procesos educativos. Este resultado abre una puerta para futuras investigaciones.
3. Se propuso una clasificación de metáforas y alegorías para computación, ya que las metáforas en esta disciplina no cumplen estrictamente la teoría de metáfora conceptual de Lakoff [Lakoff 1993].
4. Se encontró mediante una revisión de literatura que las alegorías tienen en general los mismos efectos que las metáforas inconexas tradicionales en interfaces gráficas de usuario y procesos de enseñanza y aprendizaje de la disciplina. Este resultado se confirmó

mediante un experimento con estudiantes de la Escuela de Ciencias de la Computación e Informática en una tarea de resolución de problemas.

5. Se formuló una propuesta conceptual que enriquece las visualizaciones de programa tradicionales, con alegorías visuales como mecanismo de abstracción para representar la máquina nocional, y con ludificación de la alegoría para satisfacer los requerimientos de software. A la propuesta se le llamó *visualizaciones lúdicas de programa* y es el aporte teórico más valioso de esta investigación, presentada en la sección 4.1 (Propuesta conceptual).
6. Se encontró mediante una encuesta que C++ es el lenguaje de programación más utilizado por los estudiantes a lo largo del bachillerato en computación de la ECCI. Para esta población los conceptos considerados más difíciles y útiles de aprender en C++ fueron concurrencia y administración de memoria.
7. Se diseñaron dos visualizaciones lúdicas de programa, que sirven de ejemplo de la propuesta conceptual: *PUPPETEER++* y botNeumann++. De la segunda se construyeron dos prototipos, que constituyen dos pruebas de que la propuesta conceptual es factible.
8. Se confirmó la teoría del efecto evaluador, ya que los usuarios finales (estudiantes), expertos en el dominio (profesores), y expertos en usabilidad (*HCI*), se interesaron en aspectos diferentes de una visualización lúdica de programa. Por tanto, se sugiere contar con estas poblaciones para alcanzar una mayor cobertura temática de los problemas de usabilidad en las evaluaciones de estas herramientas.
9. Se verificó que la visualización lúdica de programa fue eficaz para ayudar a los usuarios a detectar uno de los errores más sutiles de la enseñanza y aprendizaje de la programación en C++ en la ECCI: la fuga de memoria.
10. Se encontró experimentalmente que las visualizaciones lúdicas de programa fueron más eficaces, eficientes, y usables, que las herramientas usadas tradicionalmente en un curso de programación para corregir errores sutiles en programas. Este resultado provee evidencia empírica a favor de la propuesta conceptual realizada en esta investigación.

## 6.3 Trabajo futuro

La investigación realizada propuso usar alegorías y ludificación para incrementar la eficacia de las visualizaciones de programa, y comprobó empíricamente que esta eficacia fue mayor que las herramientas tradicionalmente usadas para comprender programas en un curso de la ECCI. Más investigación sería especialmente útil para comprender sobre las visualizaciones lúdicas de programa propuestas, y en general sobre la creación de herramientas exitosas para ayudar a comprender máquinas nocionales. Las siguientes son algunas sugerencias.

1. Este estudio evaluó el efecto en corto plazo de las visualizaciones lúdicas de programa en la eficacia de detección y corrección de errores. Sin embargo, los efectos podrían ser aún más significativos en variables de largo plazo, como el recuerdo, el aprendizaje, y la transferencia. Métodos cualitativos y longitudinales, como estudios de caso, podrían ayudar a los científicos a tener un conocimiento más holístico de los efectos de las visualizaciones lúdicas de programa. Por ejemplo, estudios longitudinales podrían comparar el efecto en los estudiantes que realizan la transición de visualizaciones de programa a los depuradores visuales industriales, respecto a aquellos que sólo utilizaron depuradores industriales.
2. Este estudio comparó una visualización lúdica de programa contra depuradores visuales por ser las herramientas tradicionales para estudiar (comprender) programas y corregir errores en ellos. Sería sumamente útil comparar el efecto las visualizaciones lúdicas contra visualizaciones tradicionales de programa que no implementan alegorías ni ludificación. Dada la carencia de estas herramientas para la máquina nocional de C++ en un estado funcional, se sugiere adaptar botNeumann++ en las variantes indicadas en el Cuadro 4.1 (p.111), y comparar sus efectos experimentalmente. El proyecto de investigación 326-B7-130 se estableció con este fin, y se espera que permita conocer en qué medida las alegorías visuales y la ludificación, por separado y en sus interacciones, colaboran a la eficacia de las visualizaciones lúdicas de programa.
3. De su experiencia docente, el autor de esta tesis ha notado cómo el uso de jueces automáticos en la enseñanza de la programación incrementa el aprendizaje pero a la vez incrementa también la necesidad de los estudiantes por ayuda, lo que satura a los instructores. Esta necesidad se comprobó durante la evaluación de los prototipos de botNeumann++, y algunos participantes debieron recurrir a la búsqueda de información a

través de una herramienta externa (un navegador web). Se sugiere que la aplicación de técnicas de inteligencia artificial usadas en los sistemas de tutoría inteligente (en inglés, *INTELLIGENT TUTORING SYSTEM*) podrían ayudar a las visualizaciones lúdicas de programa a satisfacer esta necesidad. Por ejemplo, el tutor podría personalizar los niveles acordes al progreso del estudiante, y ofrecer realimentación explicada en términos de la alegoría visual a cambio de recompensas que el estudiante acumularía con su progreso.

4. Cuando se usa una alegoría multi-modal para representar los conceptos de una máquina nocional, es probable que se usen metáforas visuales y auditivas para representar las entidades en el dominio origen. Pero si se agrega el modo verbal ¿produce un efecto distinto si el texto usa los términos del dominio origen, los del dominio destino, o una combinación de ambos? Si se tiene una narración de juego ¿Afecta la eficacia de la visualización lúdica si la historia usa términos de un dominio u otro en variables como la comprensión o el recuerdo? Por ejemplo, en botNeumann++ un robot representa un hilo de ejecución, ¿influye si la historia del juego usa exclusivamente el término "robot" en lugar de "hilo de ejecución"?
5. En la vida cotidiana es común que cuando se dispone de abundancia de recursos, se tienda al desperdicio, y por el contrario, cuando hay carencia de recursos, se tienda a la optimización. La visualización de programa botNeumann++ impone limitaciones estrictas de memoria y de procesamiento a los programas. Estas limitaciones son menos estrictas en los ambientes de ejecución tradicional, como un sistema operativo. ¿Influyen estas limitaciones en que los estudiantes desarrollen el hábito de escribir programas más eficientes?
6. Aunque el diseño de botNeumann++ contempla trabajo colaborativo en los modos de juego "Cooperar" y "Crear", esta característica no ha sido evaluada en visualizaciones lúdicas de programa. La colaboración puede usarse para que los estudiantes requieran unir recursos para resolver problemas de rendimiento crítico, o para resolver problemas complejos, y puede involucrar al docente. De acuerdo a la teoría sociocultural, la colaboración es uno de los escenarios más ricos para el aprendizaje y podría influir positivamente en la eficacia de las visualizaciones de programa.

## 6.4 Publicaciones

Las siguientes son las publicaciones derivadas de esta investigación. Cada referencia se acompaña de una pequeña reseña sobre su contenido.

1. Hidalgo-Céspedes, Jeisson; Marín-Raventós, Gabriela; Lara-Villagrán, Vladimir. "*PLAYING WITH METAPHORS: A METHODOLOGY TO DESIGN VIDEO GAMES FOR LEARNING ABSTRACT PROGRAMMING CONCEPTS*". *PROCEEDINGS OF THE 19TH CONFERENCE ON INNOVATION & TECHNOLOGY IN COMPUTER SCIENCE EDUCATION* (ITiCSE'14), ACM, Uppsala, Suecia, 2014. http://dx.doi.org/10.1145/2591708.2602661

   Póster sobre la alegoría *PUPPETEER++* y la metodología con que se obtuvo. Muestra ideas iniciales de la investigación, ubicada como una metodología para diseñar videojuegos que luego se refinaría a ludificación. Alojado en *ACM DIGITAL LIBRARY*. Indexado por *SCOPUS*.

2. Hidalgo-Céspedes, Jeisson. "*ALLEGORIES FOR LEARNING ABSTRACT PROGRAMMING CONCEPTS*". *PROCEEDINGS OF THE TENTH ANNUAL CONFERENCE ON INTERNATIONAL COMPUTING EDUCATION RESEARCH* (ICER'14), ACM, Glasgow, Escocia, 2014. http://dx.doi.org/10.1145/2632320.2632326

   Consorcio doctoral que recibió realimentación de expertos internacionales sobre la propuesta de tesis. Es la primera publicación de esta investigación en introducir el concepto de alegoría. Alojado en *ACM DIGITAL LIBRARY*. Indexado por *SCOPUS*.

3. Hidalgo-Céspedes, Jeisson; Marín-Raventós, Gabriela; Lara-Villagrán, Vladimir. "*STUDENT UNDERSTANDING OF THE C++ NOTIONAL MACHINE THROUGH TRADITIONAL TEACHING WITH CONCEPTUAL CONTRAPOSITION AND PROGRAM MEMORY TRACING*". Conferencia Latinoamericana en Informática (CLEI 2015). Arequipa, Perú. http://dx.doi.org/10.1109/CLEI.2015.7360049

   Artículo de conferencia que comprueba empíricamente que los métodos estáticos de visualización usados en la enseñanza tradicional son insuficientes para que los estudiantes comprendan cómo los programas corren en una máquina nocional. Fue el

artículo con la mejor evaluación recibida en el XXIII Simposio Iberoamericano de Educación Superior en Computación, y fue invitado a ser extendido para su publicación en la revista CLEIej. Alojado en *IEEE XPLORE*. Indexado por *SCOPUS*. Publicado en español.

4. Hidalgo-Céspedes, J., Marín-Raventós, G. and Lara-Villagrán, V., 2016. "*UNDERSTANDING NOTIONAL MACHINES THROUGH TRADITIONAL TEACHING WITH CONCEPTUAL CONTRAPOSITION AND PROGRAM MEMORY TRACING*". *CLEI ELECTRONIC JOURNAL*, 19(2). http://dx.doi.org/10.19153/cleiej.19.2.2

   Artículo de revista que extiende el artículo de conferencia anterior. Indexado por *WEB OF KNOWLEDGE (SCIELO)*.

5. Hidalgo-Céspedes, J., Marín-Raventós, G. and Lara-Villagrán, V., 2016. "*EXPERIENCES DESIGNING AND VALIDATING A GAMIFIED DEVELOPMENT ENVIRONMENT FOR LEARNING PROGRAMMING*". In M. Larrondo & H. Álvarez, eds. *PROCEEDINGS OF THE 14TH LATIN AMERICAN AND CARIBBEAN CONFERENCE FOR ENGINEERING AND TECHNOLOGY* (LACCEI'16). San José, Costa Rica. http://laccei.org/LACCEI2016-SanJose/meta/RP182.html

   Artículo de conferencia que publica: (1) los resultados de la encuesta hecha a los estudiantes de la ECCI, que determinó que C++ es el lenguaje más usado y que los conceptos de concurrencia y administración de memoria son considerados los más útiles y difíciles de aprender. (2) El diseño de *PUPPETEER++* para visualizar dichos conceptos usando el teatro de títeres como alegoría. (3) Los resultados de la validación de *PUPPETEER++* con expertos a través de grupos focales.

6. Hidalgo-Céspedes, J., Marín-Raventós, G. and Lara-Villagrán, V., 2016. "*LEARNING PRINCIPLES IN PROGRAM VISUALIZATIONS: A SYSTEMATIC LITERATURE REVIEW*". In 2016 *IEEE FRONTIERS IN EDUCATION CONFERENCE* (FIE). Erie, PA, USA, IEEE, pp. 1–9. http://ieeexplore.ieee.org/document/7757692/

   Artículo de conferencia que publica: (1) La extensión de la revisión de literatura exhaustiva sobre visualizaciones de programa a los años 2012-2016. (2) La verificación

que permitió conocer el nivel de satisfacción de los 16 requerimientos de software. (3) El nivel de soporte de alegorías visuales y ludificación por parte de las visualizaciones de programa disponibles. Alojado en *IEEE XPLORE*. Indexado por *SCOPUS*.

7. Hidalgo-Céspedes, J., Marín-Raventós, G. and Lara-Villagrán, V., Villalobos-Fernández, L. "*EFFECTS OF ORAL METAPHORS AND ALLEGORIES ON PROGRAMMING PROBLEM SOLVING*". *COMPUTER APPLICATIONS IN ENGINEERING EDUCATION* (ISI *IMPACT FACTOR* 0.694) Nov. 17th, 2017.

   Artículo de revista que publica: (1) Una taxonomía propuesta de metáforas para computación, ya que las metáforas en la disciplina no cumplen la teoría de la metáfora conceptual de Lakoff. (2) Una revisión de literatura que encontró una igualdad de efectos de las alegorías y las metáforas inconexas tradicionales en la disciplina de computación. (3) Los resultados de un experimento que comparó el efecto de la instrucción alegórica contra la instrucción metafórica tradicional en la resolución de problemas en estudiantes de la Escuela de Ciencias de la Computación e Informática. Alojado en *WILEY ONLINE LIBRARY*. Indexado por *WEB OF SCIENCE* y *SCOPUS*.

Publicaciones futuras:

1. Artículo con los resultados de la evaluación de usabilidad del prototipo PowerPoint y del prototipo C++ de botNeumann++. A enviar a *INTERNATIONAL JOURNAL OF HUMAN-COMPUTER INTERACTION* (*ISI IMPACT FACTOR* 0.688).

2. Artículo con los resultados de la comparación experimental del prototipo C++ contra las herramientas usadas tradicionalmente en el curso. A enviar a la revista más prestigiosa en el tema: *JOURNAL OF VISUAL LANGUAGES AND COMPUTING* (*ISI IMPACT FACTOR*, 1.171).

3. Artículo con las lecciones aprendidas en la implementación de los prototipos de botNeumann++. Estas lecciones son de utilidad para otros ingenieros interesados en construir visualizaciones lúdicas de programa. A enviar a una conferencia sobre educación de la computación.

# Anexo A Elementos lúdicos

El Cuadro 6.1 resume los doce elementos lúdicos identificados por [Kapp 2012]. Estos elementos son referidos en la sección de ludificación (2.5) del marco teórico.

Cuadro 6.1. Resumen de elementos lúdicos identificados por [Kapp 2012]

| **1. Abstracción de conceptos y realidad** |
|---|
| Los juegos no son la realidad, sino una abstracción o simplificación de la realidad, sea esta física o imaginaria. Se eliminan detalles de la realidad para permitir al jugador concentrarse en los principios que se pretenden aprender o desarrollar. Por ejemplo, el juego "Monopolio" (en inglés, *Monopoly*), permite al jugador concentrarse en reglas básicas de economía, abstrayendo detalles reales como el papeleo legal por la compra y venta de una propiedad. |
| **2. Objetivos** |
| Los juegos tienen objetivos específicos y no ambiguos, que dan propósito al juego, y fácilmente permiten saber si este propósito se ha alcanzado o no. En el juego "Monopolio" gana el jugador que sobrevive después de que los demás se han declarado en quiebra. El objetivo de un jugador de "Serpientes y escaleras" es llegar primero al final de la travesía. Los objetivos sustentan el juego y mantienen a los jugadores progresando en él.<br>Si un juego tiene varios objetivos, deben estructurarse y secuenciarse. Normalmente uno será el objetivo principal del juego, y los demás objetivos secundarios que el jugador debe ir completando para alcanzar el principal. Por ejemplo, el objetivo principal de *Super Mario Bros* es rescatar a la princesa; pero para desafiar al villano final, el jugador debe aprender primero a vencer dragones de menor dificultad. Una vez que el jugador alcance el objetivo principal del juego, el juego concluye. |
| **3. Reglas** |
| Las reglas son restricciones para limitar las acciones de los jugadores con el fin de mantener al juego manejable. Hay 4 tipos de reglas:<br>1. Las reglas operacionales indican cómo se juega. Por ejemplo, si en "Serpientes y escaleras" el jugador cae en una cabeza de serpiente, debe descender hasta su cola.<br>2. Las reglas constitutivas o reglas fundacionales son restricciones formales que conocen los diseñadores del juego y están implícitas u ocultas a los jugadores. Por ejemplo, la probabilidad de que la bola de una ruleta se detenga en una celda debe ser uniforme para cada celda.<br>3. Las reglas implícitas o reglas de comportamiento gobiernan el contrato social entre dos o más jugadores. No se escriben, sino que se asumen.<br>4. Las reglas de instrucción que se pretende el jugador aprenda en el caso de un juego educativo.<br>Idealmente las reglas deben otorgar libertad a los jugadores de alcanzar los objetivos utilizando métodos diversos. |
| **4. Conflicto, competición o cooperación** |
| Un juego ofrece un ambiente de conflicto. Un conflicto es una situación de reto que impide el progreso de los jugadores. Para superar el conflicto, los jugadores deben competir, cooperar o una mezcla de ambas. Por ejemplo, en el fútbol el objetivo es anotar más goles que el contrincante. Sin embargo, dado que sólo hay un único balón, se genera un conflicto por dominarlo. Jugadores de diferentes equipos compiten por dominar el balón. Jugadores de un mismo equipo colaboran para mantener el dominio o recuperar el balón. La competición se da entre jugadores que dedican su atención a optimizar su propio rendimiento con el fin de superar el rendimiento de sus oponentes. La cooperación o colaboración se da cuando jugadores trabajan con otros para lograr un resultado deseable y beneficioso para todos. |
| **5. Tiempo** |
| Los juegos utilizan el tiempo principalmente para motivar a los jugadores a tomar acciones. Un |

contador de tiempo regresivo o progresivo tiende a incrementar en los jugadores el nivel de estrés y la necesidad de optimizar acciones para alcanzar el objetivo del juego. El tiempo es un valioso recurso en una ludificación si también lo es en la actividad seria. Por ejemplo, el tiempo de respuesta a una llamada telefónica en un departamento de soporte técnico. El tiempo de los juegos no necesita coincidir con el de la realidad. En unos segundos un juego puede construir un edificio que tarda meses en la realidad.

**6. Reconocimiento**

Con el fin de incrementar la motivación por alcanzar los objetivos, los juegos recompensan a los jugadores por sus logros, utilizando puntos, insignias, premios u otros objetos de valor real o ficticio. Niveles mayores de motivación pueden alcanzarse cuando el reconocimiento es social. Un ejemplo común en los videojuegos son las tablas de marcadores con el registro de los jugadores que han alcanzado mayores niveles de progreso.

En algunos juegos las recompensas van más allá de objetos coleccionables: se pueden canjear por beneficios para el jugador que le ayudan en su progreso o aprendizaje. La escogencia de los logros del jugador a premiar debe ser cuidadosa. En una ludificación se deben premiar aquellos logros que representan un progreso del jugador en el conocimiento o habilidades que se pretenden aprender.

**7. Realimentación**

Los juegos ofrecen al jugador realimentación en tiempo real de su progreso hacia el objetivo. Por ejemplo, información sobre energía disponible, ubicación, tiempo restante, y progreso de otros jugadores. En una ludificación, la realimentación se provee para evocar el comportamiento, los pensamientos, o acciones correctas en el aprendiz. La realimentación informa de inmediato si el jugador hizo algo correcto, incorrecto, o un intermedio de los dos anteriores. La realimentación no dice al jugador cómo corregir una acción errónea, a lo sumo debe guiarlo hacia una acción correcta, por ejemplo, proveyéndole pistas.

Algunos criterios que ayudan a proveer realimentación efectiva para el jugador: Cuando hay realimentación en el juego, el jugador debe saber que ocurrió. El jugador la desea y trabaja para conseguirla. La realimentación es repetible, aparece una y otra vez ante las mismas circunstancias. Es continua, ocurre en la interacción del jugador, no sólo en situaciones impredecibles. Es emergente, integrada con el ambiente del juego y no es de distracción. Balanceada, no sobrecarga al jugador de información. A veces es sorpresiva, provee algo útil al jugador que no esperaba y motiva su aprendizaje.

**8. Niveles**

Los diseñadores de juegos utilizan niveles para dar estructura y evitar que el jugador vague sin rumbo por el juego. En cada nivel el jugador cumple un pequeño conjunto de objetivos, antes de pasar al siguiente. Los niveles se ordenan de tal forma que cada nivel:

1. Agrega un trozo a un relato general, lo que despierta el interés en el jugador por averiguar qué sigue.
2. Fortalece conocimientos o habilidades desarrollados en niveles previos, o introduce nuevos.
3. Incrementa ligeramente el nivel de dificultad, de tal forma que el jugador pueda alcanzar los objetivos del nivel.

Los primeros niveles explican la navegación del juego y una habilidad básica a la vez. Los niveles intermedios requieren recordar y utilizar conocimientos y habilidades adquiridos en niveles previos con mayor rapidez o bajo mayor presión. Los niveles finales requieren que los jugadores utilicen varias habilidades aprendidas en niveles previos en combinaciones únicas para ganar el juego.

Un juego muy fácil aburre al jugador, mientras que uno muy difícil lo frustra. Lograr un equilibrio es el ideal, pero es difícil diseñar un juego que se adapte a una audiencia con jugadores de distintos niveles de experiencia e intereses. Una solución es proveer niveles fáciles, intermedios y difíciles, e ir revelándolos al jugador de acuerdo a su pericia. Es común que los niveles provean recompensas acordes a su nivel de dificultad.

**9. Narración**

La narración provee contexto con el fin de darle relevancia y significado al juego. No todos los juegos tienen una narración, pero en la ludificación de actividades educativas es esencial. Las historias han sido usadas durante centurias para pasar información de una persona a otra. Las historias son más

| fáciles de recordar que listas de elementos, lo cual es importantísimo en la ludificación de la educación. La narración debe ayudar al jugador a comprender los objetivos del juego y a aprender conductas deseadas, acciones y patrones de pensamiento. Se debe involucrar al jugador en la narrativa del juego. Normalmente una historia incluye: (1) caracteres, de los cuales uno se identifica con el jugador; (2) un problema que ocurrió; (3) tensión porque el carácter no sabe qué hacer o reacciona de forma equivocada; y (4) una solución, la cual es planteada por el carácter quien decide ejecutarla, pero necesita la ayuda del jugador. |
|---|
| **10. Curva de interés** |
| La curva de interés es el curso de eventos que mantiene al jugador interesado en el juego a través del tiempo. Inicia con el punto de entrada, con la motivación que hace al jugador llegar al juego por primera vez. El juego debe provocar la atención del jugador e ir incrementándola conforme avanza hacia el final del juego. Al probar un juego es conveniente pedir a sus jugadores reportar la curva de interés y comparar los resultados contra lo que el diseñador espera. |
| **11. Estética** |
| La estética abarca el arte, los detalles visuales y la belleza de los componentes del juego. En el ajedrez se incurre en aspectos estéticos para distinguir un alfil de un peón y a las piezas de un jugador de las piezas de su contrincante. Sin belleza estética, un juego tiene alta probabilidad de ser considerado como aburrido. La estética no consiste en proveer gráficos foto-realistas, sino en el uso de detalles que comunican claramente un mensaje. Por ejemplo, en algunos juegos de estrategia una fábrica en construcción se presenta en gris y con niveles incrementales de detalles. Cuando adquiere color y emana humo, comunica al jugador que está lista sin recurrir a un discurso verbal. |
| **12. Reintento** |
| En los juegos, el fallo es una opción aceptada. En comparación a una actividad seria, fallar en un juego con consecuencias mínimas incentiva la exploración, la curiosidad y el aprendizaje basado en el descubrimiento. Proveer un botón de reintentar ofrece un sentido de libertad y los jugadores lo utilizan para poner sus caracteres en peligro y ver qué ocurre, formular y probar hipótesis, y recordar cuáles fueron exitosas y cuáles fallaron.<br>El fallo debe verse parte del proceso normal de progreso por el juego, y se debe prestar atención a los casos extremos. El fallo sin progreso es una señal de alerta. Un jugador estancado en un nivel es un candidato a abandonar el juego. En el extremo opuesto, avanzar por el juego sin fallos es una experiencia insatisfactoria para el jugador. |

# Anexo B Implementación del prototipo C++

En este anexo se incluyen detalles técnicos de implementación del prototipo C++, dado que son lecciones aprendidas especialmente útiles para otros ingenieros interesados en desarrollar visualizaciones lúdicas de programa. Las secciones de este anexo están organizadas siguiendo la arquitectura implementada en el prototipo.

## B.1 Arquitectura del prototipo

El prototipo C++ fue implementado en módulos siguiendo las recomendaciones de [Lanza 2003]. Según este autor, toda *visualización de software* debe, de alguna u otra forma, tener los cuatro componentes arquitectónicos resaltados con fondo gris en la Figura 6.1 [Lanza 2003, p.2]. Estructurar el código de la visualización en estos componentes, que se describen a continuación, aísla a unos de otros y mitiga los efectos de los cambios en los requerimientos de la visualización [Lanza 2003, p.3].

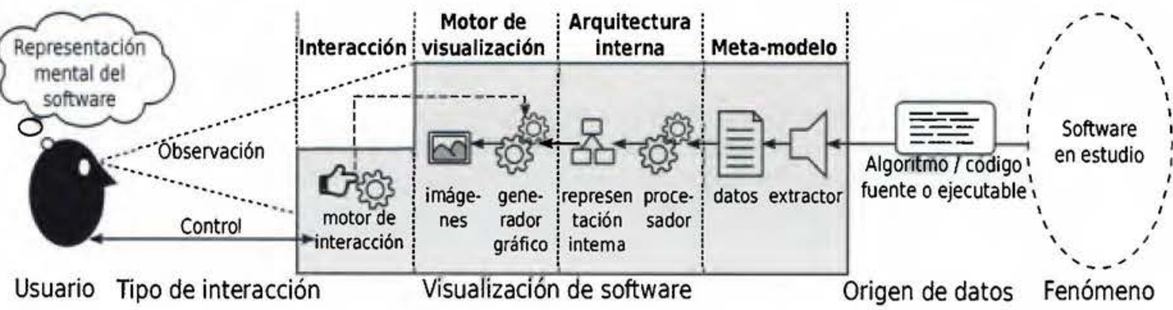



Figura 6.1. Arquitectura de una visualización de software (adaptado de [Lanza 2003, p.3])

1. El **meta-modelo** se encarga de extraer y estructurar los datos que deben ser visualizados. Los datos se extraen del código fuente, programas en tiempo de ejecución, algoritmos, o una combinación de ellos. En el caso de una visualización de código, se requiere un analizador estático de código, usualmente provisto en forma de biblioteca de software (por ejemplo, *MOOSE*[24]) [Lanza 2003]. En el caso de una visualización de programa, se requiere de un analizador dinámico de programa o de un depurador (por ejemplo, *GDB*[25]),

[24] http://www.moosetechnology.org/
[25] https://www.gnu.org/software/gdb/

en forma de biblioteca o programa ejecutable que pueda ser invocado y controlado por la visualización [Egan and McDonald 2013].

2. La **arquitectura interna** se encarga de procesar los datos extraídos por el meta-modelo y transformarlos a alguna forma de representación interna que sea fácil de visualizar [Lanza 2003].
3. El **motor de visualización** se encarga de convertir la representación interna en información gráfica y desplegarla al usuario. Incluye algoritmos gráficos que producen imágenes. El diseñador debe escoger entre tres alternativas: (1) utilizar una biblioteca de visualización de uso general, la cual ahorra trabajo pero puede limitar libertad potencialmente requerida por la visualización; (2) escribir una biblioteca de visualización desde cero, cuyo trabajo es tedioso pero provee el máximo de flexibilidad; o (3) un intermedio de las dos anteriores, como alterar o extender una biblioteca de uso general. Diseñar el motor de visualización requiere la combinación cuidadosa de principios artísticos (diseño gráfico, animación y cinematografía), semiótica visual, y tecnología de gráficos por computadora. [Lanza 2003]
4. El **motor de interacción** se encarga de permitir al usuario manipular directamente la visualización. De acuerdo a [Lanza 2003], este es el componente que requiere más trabajo de implementación, y el que finalmente dicta si la herramienta es usable o exitosa. El diseño de este componente requiere la aplicación de técnicas modernas de interacción humano-computador (*HCI*).

La arquitectura del prototipo C++ de botNeumann++ extiende la arquitectura de una visualización de software propuesta por Lanza, como se ve en la Figura 6.2, al agregar dos módulos: el *motor de ludificación* y el *motor de casos de prueba*. El **motor de ludificación** emplea al motor de interacción y el de visualización para proveer elementos del juego a la visualización de programa, por ejemplo, niveles.

El **motor de casos de prueba** proviene del juez automático (apartado 4.4.2.4, p.149). Se encarga de dos tareas: (1) generar casos de prueba y (2) correr instancias del programa del estudiante contra los casos de prueba generados. Como se ve en la parte inferior de la Figura 6.2, el flujo de información inicia en el motor de casos de prueba, ya que es el módulo que alimenta de datos al programa del estudiante. La ejecución de una de las instancias del programa del estudiante es la animada por los demás módulos de la arquitectura en la parte

superior de la Figura 6.2. El primero de estos seis módulos en ser implementado en el prototipo C++ fue el motor de visualización.

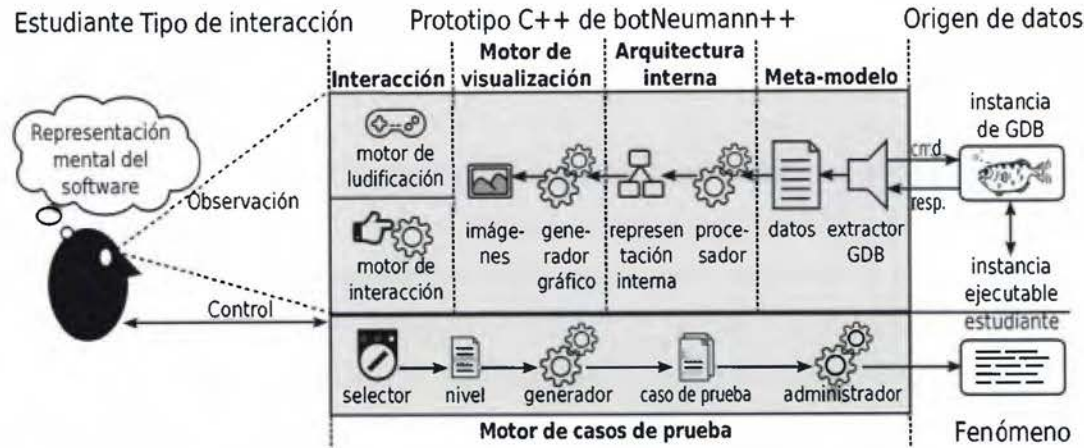


Figura 6.2. Arquitectura del prototipo C++ de botNeumann++

## B.1.1 El motor de visualización

La implementación del prototipo C++ inició con el motor de visualización. Como se indica en [Lanza 2003], este módulo requiere reusar, modificar o crear una biblioteca que ayude a generar las gráficas de la visualización. La primera biblioteca evaluada fue Cocos2d-x[26], lo cual se hizo a partir del 1º de octubre de 2014. Esta biblioteca fue seleccionada por ser un motor de videojuegos en C++, popular, libre y por contar con amplia documentación. Como ventajas permitió el desarrollo multiplataforma y facilitó el trabajo con animaciones. Sin embargo, se descartó luego de dos semanas de pruebas por sus limitados componentes de interfaz gráfica de usuario. botNeumann++ requiere componentes avanzados que no están disponibles en esta biblioteca, como un editor de texto enriquecido para proveer edición natural de código fuente, y un motor web para formatear los enunciados de los problemas.

La segunda tecnología evaluada fue V-Play[27], por ser tanto un motor de videojuegos como de aplicaciones multiplataforma basado en Qt (discutido más adelante), cuya licencia académica es gratuita. Su evaluación inició el 28 de octubre de 2014. V-Play usa un lenguaje basado en *JavaScript* para crear los elementos gráficos y animarlos, lo cual se considera una ventaja para acelerar el desarrollo. Después de tres semanas de pruebas y comunicaciones con los autores

[26] http://www.cocos2d-x.org/
[27] https://v-play.net/

de V-Play, se descartó para el desarrollo del prototipo. La principal razón fue que botNeumann++ requería funcionalidad provista por Qt a través de C++ pero que no está disponible para *JAVASCRIPT*, como la capacidad de invocar y controlar programas (`QProcess`) y desplegar imágenes vectoriales en formato *SVG* (del inglés, *SCALAR VECTOR GRAPHICS*). Para poder acceder a esta funcionalidad desde *JAVASCRIPT*, se debía crear adaptadores de código en C++. En tal caso se prefirió escribir el código del prototipo en C++ y acceder directamente a la funcionalidad de Qt.

Qt fue la tercera tecnología evaluada y la escogida el 5 de noviembre de 2014 para implementar el motor de visualización y de interacción. Qt[28] es un marco para desarrollar aplicaciones multiplataforma (en inglés, *APPLICATION FRAMEWORK*). Qt incluye bibliotecas de software que aíslan al desarrollador de la plataforma. De las bibliotecas disponibles en Qt, el prototipo C++ de botNeumann++ empleó componentes de interfaz gráfica (*GUI*), multimedia (ej.: sonido), redes, lenguaje de marcado extensible (*XML*, del inglés *EXTENSIBLE MARKUP LANGUAGE*), y gráficos vectoriales (*SVG*). Qt tiene una licencia comercial y dos de software libre: *GPL* (del inglés, *GNU GENERAL PUBLIC LICENSE*) y *LGPL* (*GNU LESSER GENERAL PUBLIC LICENSE*). El prototipo de botNeumann++ aplica la licencia *GPL* y por tanto es software libre. Su código fuente está disponible al mundo en un repositorio de control de versiones de *GITHUB*[29]. El histograma de la Figura 6.3, generado por *GITHUB*, ilustra de la cantidad de cambios (*COMMITS*) hechos en el tiempo en el repositorio. Se rotuló en la parte superior de la Figura 6.3 las fases o productos de desarrollo que se mencionan en esta sección.

Figura 6.3. Historial de cambios en el desarrollo de prototipo C++ (adaptado de *GITHUB*)

Una limitación de Qt es que, al ser una biblioteca de uso general, no dispone de facilidades que proveen los motores de juegos como realizar transiciones entre escenas, o posicionar y

[28] https://www.qt.io/

[29] https://github.com/citic/botNeumann

actualizar elementos gráficos en la pantalla y ajustarlos cuando ésta cambia sus dimensiones. Esta funcionalidad es necesaria para permitir edición natural de código. Como se confirmó en la prueba de usabilidad del prototipo PowerPoint, para los usuarios es importante ajustar el tamaño del editor de código para que las líneas se vean completas, lo que estruja o expande el área de visualización.

Se creó entonces un motor de visualización a partir de una biblioteca de uso general (el caso 3 señalado por [Lanza 2003]), lo que implicó un contratiempo en el desarrollo del prototipo C++. Para propósitos internos se le dio el nombre en clave `GameE` (por *GAME ENGINE*) y su código fuente se encuentra en la subcarpeta `source/gamee` del repositorio de control de versiones. Para poder ubicar los elementos gráficos en una interfaz que puede re-dimensionarse, se implementó un administrador de posicionamiento (en inglés, *LAYOUT MANAGER*). A diferencia de los administradores de posicionamiento disponibles en Qt que usan proporciones enteras, `GameE` utiliza porcentajes en punto flotante para la posición y proporción de los elementos gráficos, y tiene además la capacidad de superponerlos sobre otros (eje z). Estas funcionalidades más el diseño gráfico hecho con imágenes vectoriales, provee la versatilidad al usuario de re-dimensionar la visualización a su gusto, incluso cuando una animación está en progreso, sin perder calidad de la imagen ni la ubicación correcta de los elementos gráficos en movimiento. Este administrador de posicionamiento podría ser un aporte innovador al diseño de interfaces gráficas animadas, pero confirmar esta hipótesis requiere de investigación futura.

### B.1.2 El motor de interacción

Se implementó en el prototipo C++ un motor de interacción muy básico, que permite accionar los elementos gráficos con un dispositivo de apuntar (por ejemplo, un ratón) o un dispositivo táctil (una pantalla sensible). Esta decisión se debe a que la mayoría de la interacción con el usuario ocurre en el editor de código, los controles de la animación, y las ventanas acoplables (en inglés, *DOCK WIDGETS*). Estos componentes en Qt ya implementan mecanismos de interacción y no requieren de un motor adicional.

Explorar técnicas modernas de interacción del usuario con una visualización lúdica de programa es un amplio horizonte para investigación futura. Por ejemplo, permitir al usuario interaccionar con los elementos gráficos de la visualización y obtener información como su

significado y las relaciones entre ellos. Otro ejemplo es permitir al usuario completar código fuente del programa creando o arrastrando elementos gráficos de la visualización hacia el editor de código, lo cual sería especialmente útil para dispositivos móviles.

### B.1.3 El motor de ludificación

Como se ilustra en la Figura 6.2, sobre el motor de interacción se creó el motor de ludificación que se encarga de implementar los elementos de juego, como el menú de juego y niveles. La Figura 6.4 muestra la máquina de estados del motor de ludificación implementado en el prototipo C++. Cuando la aplicación inicia se presenta la *pantalla de menú* donde el estudiante puede crear o escoger un usuario. Al accionar un modo de juego, como "Entrenar" se presenta la *pantalla de selección de nivel* (mapa de niveles). Al escoger un nivel se presenta la *pantalla de nivel*, donde el usuario puede solucionar un problema planteado y visualizar sus programas. El usuario puede regresar entre estas pantallas accionando un botón con forma de flecha hacia atrás, como es habitual en videojuegos casuales. En las siguientes secciones se detalla cada una de estas pantallas por separado.

Figura 6.4. Máquina de estados general del prototipo C++

## B.2 Pantalla de menú

Existen pocas diferencias entre la pantalla de menú en el prototipo C++ (Figura 6.5) con su correspondiente en el prototipo PowerPoint. Dado que en el prototipo C++ no se implementó el tutorial por delimitaciones de tiempo, la pregunta del nombre del jugador (véase la Figura 5.2, p.157) se sustituyó por un administrador de jugadores (Figura 6.6), el cual se activa cuando se presiona el nombre del jugador en la pantalla de menú (el texto "Jugador" en la Figura 6.5). El administrador de jugadores permite crear, renombrar, eliminar y cambiar el jugador activo.

Figura 6.5. Pantalla de menú en el prototipo C++

El nombre del jugador es un dato necesario en el prototipo C++ para identificar su progreso, datos (ej.: código fuente), preferencias y eventos que se registran en la bitácora. Sin los nombres de los jugadores, sería muy difícil para el investigador analizar los datos y las bitácoras, por ejemplo, tras reunir los archivos de un experimento en una computadora para su análisis. Por este motivo, crear o seleccionar un jugador es un requisito para que botNeumann++ habilite el botón de entrenamiento (rotulado *TRAINING* en la Figura 6.5).

Figura 6.6. Administrador de jugadores en el prototipo C++

Como una amenidad, el prototipo C++ permite que dos o más jugadores puedan usar una misma máquina sin confundir sus datos y progreso. Esta característica es útil para realizar pruebas de usabilidad en una misma computadora con varios participantes sin tener que eliminar el progreso o las bitácoras de los participantes anteriores. Antes de pasar a la pantalla de niveles, se explican las bitácoras generadas por el prototipo.

### B.2.1 Registro de eventos

El prototipo C++ registra eventos en bitácoras siguiendo las recomendaciones de [Chung and Kerr 2012] para videojuegos educativos. El **registro de eventos** (en inglés, *DATA LOGGING*) corresponde a la especificación sistemática, codificación, captura y almacenamiento de eventos que ocurren durante la ejecución de una aplicación con el fin de recabar evidencia para apoyar o rechazar hipótesis de investigación (adaptado de [Chung and Kerr 2012, p.1]).

El aspecto crítico del registro de eventos es la especificación de qué comportamientos de los jugadores o estados del juego registrar. El principal reto es mapear comportamientos de los jugadores durante el juego con sus procesos cognitivos o emocionales, porque los segundos no son observables. Idealmente el diseño de la interfaz y de la mecánica del juego, debe permitir sólo a los aprendices que hayan adquirido un conocimiento o habilidad *X* poder ejecutar una mecánica *x*, de tal forma que la mecánica *x* se convierta en una medida potencial de *X* [Chung and Kerr 2012, p.2]. Por ejemplo, en caso de que se tengan varios niveles encadenados donde uno es requisito de otro, si un participante alcanzó un nivel superior, puede considerarse un indicio de conocimiento sobre los niveles previos. Por la corta duración de la sesión experimental, este encadenamiento de niveles no se evaluó en el prototipo C++ y queda como una oportunidad de investigación futura.

De acuerdo a las recomendaciones de Chung y Kerr, el prototipo C++ se diseñó para registrar al más fino nivel de granularidad usable. Es decir, que el evento tenga suficiente información para describir el contexto en el que ocurrió y pueda usarse para análisis sin tener que recurrir a eventos cercanos que le den sentido. Por ejemplo "el usuario hizo clic" no es usable, pero sí "el usuario 17 hizo clic en el botón de reproducir" [Chung and Kerr 2012, p.3]. El registro de eventos se procuró que sea completo, de tal forma que si se ordenan permiten reconstruir lo ocurrido durante las sesiones de juego y que es relevante para los análisis.

En cuanto el formato de los registros, se implementó la preferencia de Chung y Kerr de la facilidad de uso de los usuarios finales de los datos (analistas, investigadores) sobre la eficiencia de almacenamiento. Al igual que [Chung and Kerr 2012, pp.4–5] se escogió el formato *CSV* (del inglés *COMMA-SEPARATED VALUES*), que consiste de un archivo de texto cuyas líneas son registros y sus campos se separan por tabuladores. Los campos recomendados por estos autores se listan en el Cuadro 6.2.

De acuerdo a Chung y Kerr, los datos almacenados por el prototipo en los campos del Cuadro 6.2 son descriptivos, no ambiguos y contextualizados. Los datos son *descriptivos* si describen objetivamente el evento y no son interpretaciones o inferencias. Por ejemplo, "el usuario obtuvo el error de compilación 134 en la línea 18" es descriptivo mientras que "el usuario no sabe programar" es inferencial. Los datos son *no ambiguos* si hay una correspondencia 1:1 entre los datos registrados y un evento ocurrido, y además permiten reproducir la sesión que los generó. Por ejemplo "el usuario hizo clic en un botón" es ambiguo, contrario a "el usuario hizo clic en el botón reproducir" dado que sólo hay un botón "reproducir" en el prototipo. Los datos son contextualizados si contienen suficiente información relevante sobre las condiciones en las cuales el evento se generó. Esta información ayuda a los analistas a generar o descartar hipótesis o explicaciones que ayuden a entender porqué el evento ocurrió durante el juego. [Chung and Kerr 2012, pp.5-6]

Cuadro 6.2 Sugerencia de campos para registro de eventos (adaptado de [Chung and Kerr 2012])

| Campo | Descripción | Implementado |
|---|---|---|
| `num` | Número de registro. Es un entero que incrementa de uno en uno. | No |
| `timestamp` | Fecha y hora en que ocurrió el evento. | Sí |
| `session_time` | El tiempo en segundos desde que la aplicación fue cargada. | Sí |
| `user_id` | Un identificador del jugador activo. | Sí |
| `level` | El nivel donde se generó el evento. Si el juego está separado en mundos (*STAGE*), se puede incluir en un campo aparte. | Pendiente |
| `event_type` | Un código numérico que identifica el tipo de evento. Los campos que continúan varían dependiendo de este código. | Parcial |
| `description` | Un texto descriptivo que varía dependiendo del tipo de evento. | Sí |
| `data_field[N]` | Varios campos cuya semántica depende del tipo de evento. N es un número entero que se obtiene con el tipo de evento que tiene más campos. | Pendiente |
| `game_state` | Es una lista de valores que indica el estado del reto que el jugador trata de resolver. Por ejemplo, en una visualización de programa podría incluir el código fuente provisto por el jugador. | Parcial |

Cada vez que el usuario cambia de pantalla, inicia la animación de un programa (un intento), ajusta la velocidad de la animación, u otras acciones, el prototipo C++ registra un evento en la bitácora que permite estudiar la interacción ocurrida. Cuando en el menú principal activa el botón de "Entrenamiento", el prototipo pasa a la pantalla de selección de nivel.

## B.3 Pantalla de selección de nivel

Una vez que el usuario presiona el botón *TRAINING* en la pantalla de menú (Figura 6.5), el prototipo muestra la pantalla de selección del nivel (Figura 6.7), también llamada mapa de niveles. El prototipo PowerPoint se pre-estableció con dos niveles, implementados como una serie de diapositivas. El prototipo C++, en cambio, no tiene niveles pre-establecidos en el código fuente, sino que el investigador puede crear una cantidad arbitraria de niveles en archivos *XML* y escoger cuáles de ellos el prototipo debe cargar en tiempo de ejecución. Por tanto, el mapa de niveles variará en función de los niveles que el investigador haya habilitado. Por ejemplo, la Figura 6.7 muestra los cuatro niveles usados en el experimento del capítulo 5.

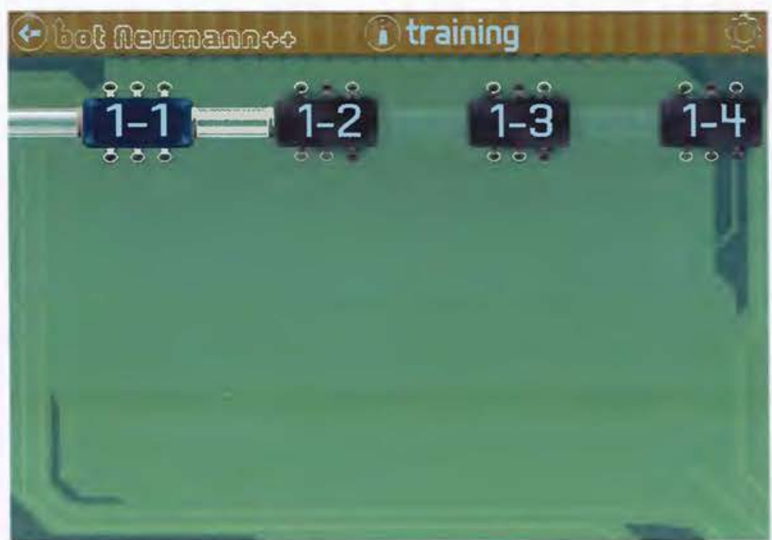



Figura 6.7. Pantalla de selección del nivel en el prototipo C++

En la alegoría inicial, los niveles se consideraron unidades de procesamiento y por tanto se representaron como circuitos integrados (chips) unidos por tubos neumáticos, como se aprecia en la Figura 6.7. Los chips fundidos representan niveles cuyo código fuente no ha sido corregido por el jugador. Los chips en buen estado que procesan y permiten el paso de cápsulas neumáticas representan niveles completados por el jugador. Posteriormente la alegoría varió a fábricas de datos, como se presentó en la sección 4.4.1 (Alegoría visual). Sin embargo, la pantalla de selección de nivel mantuvo la metáfora de los chips por delimitaciones de tiempo.

El prototipo C++ carga los niveles desde archivos *XML*, los cuales se encuentran en la carpeta `units/` del repositorio de control de versiones. Para saber cuáles de ellos deben mostrarse en

la pantalla de niveles y en qué orden, un archivo *XML* de recursos de Qt se usa para referirlos. El Listado 6.1 muestra la lista de niveles usados en el experimento controlado. El prototipo asigna un chip a cada entrada `<file>` del archivo de niveles. Por ejemplo, el circuito integrado "1-1" en la Figura 6.7 corresponde al archivo `e1-f_to_k.botnu` referido en el Listado 6.1, el cual contiene un ejercicio de conversión de Fahrenheit a Kelvin que se muestra a lo largo de esta sección.

```
<RCC>
   <qresource prefix='/training/ndebug">
      <file>e1-f_to_k.botnu</file>
      <file>e2-increment_percent.botnu</file>
      <file>e3-is_binary.botnu</file>
      <file>e4-animal_sound.botnu</file>
   </qresource>
</RCC>
```

Listado 6.1. Ejemplo de una lista de niveles a cargar en el prototipo C++

El prototipo asume que todos los archivos agrupados en un `<qresource>` están relacionados y los presenta conectados por un tubo neumático y numerados en secuencia ("1-1", "1-2", ...). Si el investigador quiere agregar otro grupo de niveles, los puede encerrar en otro elemento `<qresouce>`; el prototipo los mostrará en una segunda hilera de chips unidos por otro tubo neumático y numerados con una nueva secuencia ("2-1", "2-2", ...) como puede verse en la Figura 6.8.

El atributo `prefix="modo/grupo"` (Listado 6.1) indica el modo de juego del que serán mostrados los niveles y el grupo al que pertenecen. Los modos de juego disponibles son entrenamiento (`training`), misiones (`missions`), colaboración (`collaboration`) y crear niveles (`create`). Corresponden a los botones que se encuentran en el menú del juego (Figura 6.5), pero los últimos tres fueron deshabilitados en el prototipo para el experimento, siguiendo las recomendaciones de la evaluación de usabilidad del prototipo PowerPoint. Por su parte, el grupo en el prefijo es cualquier identificador que guste usar el investigador para agrupar niveles relacionados.

Una ventaja de los archivos de recursos de Qt es que son independientes de la plataforma, y los archivos referidos estarán disponibles en el ejecutable incluso aunque el prototipo corra en un dispositivo móvil. Una limitación de estos archivos es que se agregan al ejecutable en tiempo de compilación, por lo que cambiar los niveles requiere recompilar el prototipo. Trabajo futuro puede superar esta limitación al implementar un repositorio de niveles

accesible a través de internet, donde los usuarios pueden crear sus propios niveles a través del modo de juego "Crear". De los cuatro niveles del experimento se usará el primero de ellos en los siguientes apartados para ejemplificar la estructura de un archivo de nivel y cómo es visualizado.

Figura 6.8. Varios niveles agrupados por tubos neumáticos

## B.4 Archivo y pantalla de nivel

Un nivel en botNeumann++ es un **ejercicio de programación**, el cual consta de un problema usualmente contextualizado, junto con parámetros, casos de prueba, y otros recursos que permite a un juez automático plantearlo a los usuarios y ofrecer realimentación sobre las soluciones que éstos someten. No existe consenso entre jueces automáticos, cursos en línea y otras herramientas que usan ejercicios de programación sobre el formato para especificarlos [Queirós and Leal 2013]. La mayoría de formatos adoptan alguna forma de *XML* para representar los ejercicios, como es el caso de *CATS*[30] y *PExIL* (del inglés, *PROGRAMMING EXERCISES INTEROPERABILITY LANGUAGE*). Aunque estos dos formatos son bastantes completos, son muy complejos de implementar. Por tanto, para el prototipo C++ se implementó una simplificación, también en *XML*, cuya estructura general se presenta en el Listado 6.2.

Un ejercicio de programación se representa con el elemento raíz `botnu` (del inglés, *BOTNEUMANN++ UNIT*, en alusión a las unidades de procesamiento), como en la línea 4 del

[30] https://imcs.dvfu.ru/cats/docs/format.en.html

Listado 6.2. El diseñador puede especificar 12 atributos que controlan cómo botNeumann++ despliega el ejercicio. El atributo `id` es el único obligatorio usado para identificar el ejercicio en eventos de bitácora y el progreso del jugador.

```
48 <?xml version="1.0" encoding="utf-8"?>
49 <!DOCTYPE botnu SYSTEM "../botnu-1.0.dtd">
50
51 <botnu id="e1-f_to_k" ...>
52   <description lang="es">...</description>
53   <initial-code lang="cpp">...</initial-code>
54   <solution lang="cpp">...</solution>
55   <standard-generator lang="cpp" default-runs="5">...</standard-generator>
56   <file-generator lang="cpp" default-runs="5">...</file-generator>
57   <test-case>
58      <input>...</input>
59      <output>...</output>
60   </test-case>
61 </botnu>
```

Listado 6.2. Estructura de un archivo de nivel para el prototipo C++

El Listado 6.3 muestra el elemento raíz del ejercicio de conversión de temperaturas. Los atributos indican la versión del ejercicio (`version`), si el código fuente en C/C++ del jugador debe compilarse en 32 ó 64 bits (`architecture`), la cantidad de núcleos del procesador a visualizar (`cpu-cores`), el tamaño de memoria primaria de la máquina nocional en la que debe correr la solución del jugador (`ram`), si se debe o no visualizar el segmento de memoria dinámica (`heap-segment`), la cantidad mínima de hilos de ejecución que el estudiante debe implementar para pasar el nivel (`min-threads`), la cantidad máxima de milisegundos que dispone el programa del usuario para pasar un caso de prueba (`timeout`), y si se debe ignorar espacios en blanco adicionales al comparar casos de prueba (`ignore-whitespace`), entre otros.

```
1 <botnu id="e1-f_to_k" version="1.0" architecture="32" cpu-cores="2"
2   ram="512" heap-segment="no" min-threads="1" timeout="1000"
3   ignore-whitespace="yes">
4   ...
5 </botnu>
```

Listado 6.3. Elemento raíz del ejercicio de conversión de temperaturas

La Figura 6.9 es una captura de pantalla de cómo el prototipo C++ visualiza el ejercicio de conversión de temperaturas. Como en el Listado 6.3 se indicó que el problema debe resolverse sin usar memoria dinámica (atributo `heap-segment`), el prototipo C++ cierra las puertas de la bodega, etiquetadas con ① en la Figura 6.9. El Listado 6.3 solicita dos núcleos de

procesador, por lo que el prototipo visualiza dos estaciones de trabajo para robots, etiquetadas con ② y ③ en la Figura 6.9.

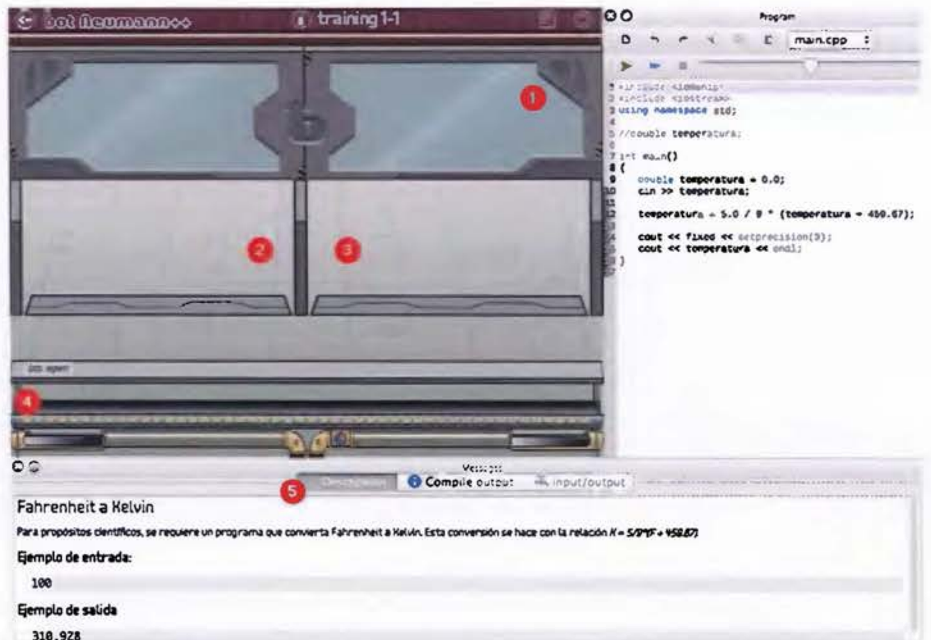



Figura 6.9. Pantalla de nivel con el ejercicio de conversión de temperaturas

El atributo más complejo de implementar fue el tamaño de la memoria (`ram`). La Figura 6.10 muestra parte del modelo matemático para distribuir los bytes entre los cuatro segmentos, de tal manera que los estantes tengan tamaños múltiplos de ocho bytes (como ④ en Figura 6.9) y proporcionales a la distribución visual de los aposentos.

Una vez especificado los atributos del elemento `botnu`, se indica su contenido. De acuerdo al extracto de la definición del tipo de documento (*DTD*, del inglés *DOCUMENT TYPE DEFINITION*) del Listado 6.4, un ejercicio de programación debe contener al menos una descripción del problema, códigos iniciales opcionales, soluciones opcionales, y al menos un caso de prueba o un generador de casos de prueba[31].

```
<!ELEMENT botnu (description+, initial-code*, solution*,
   (standard-generator|file-generator|test-case)+) >
```

Listado 6.4. Contenido de un ejercicio de programación según el DTD

[31] El *DTD* contiene una descripción detallada de cada elemento del ejercicio de programación. Se puede acceder en https://github.com/citic/botNeumann/tree/master/units

BotNeumann — Memory distribution model — 2016-02-10(1)

| var | description | units | restrict |
|---|---|---|---|
| (R) | RAM size | bytes | input |
| Rv | visible RAM ▨ | bytes | |
| Ri | invisible RAM □ | bytes | |
| (C) | CPU cores | cores | input |
| (A) | architecture width | bits | input |
| (a) | larger data type (double) | bytes | input |
| K | colums | colums | |
| Kc | colums / core | colums/core | |
| r | rows | rows | |
| rv | visible rows | rows | |
| ri | invisible rows | rows | |
| h / H | heap segment | rows / bytes | |
| d / D | data segment | rows / bytes | |
| Sv / Sv | visible stack segment | rows / bytes | |
| Si / Si | invisible stack segment | rows / bytes | |
| Svc / Svc | visible stack segment / core | rows/core / bytes/core | |

Our goal is distributing R bytes to each segment: heap (H), stack (S), and data (D). The stack is divided in visible memory ($S_v$) of the current running function, and available invisible memory for future function calls ($S_i$). Each CPU core must have a complete stack segment ($S_c$). Because we want to avoid storing large datatypes, such as double or long long (a), we store the memory in columns (K) of a bytes. Each CPU core requires at least 1 column ($K_c$). The whole area is

$$rKa = R \quad (0)$$
$$r = r_v + r_i \quad (1)$$
$$r_v = h + S_v + d \quad (2)$$
$$r_i = S_i$$

Let's do some simplifications. The critical area is the visible stack segment. Now the other segments in function of $r_i$, according to subjective convenience criteria of the author. I may be very wrong.

$$h = 2S_v$$
$$S_i = 4S_v$$
$$d = S_v/2$$

substitution on (2)

$$r_v = h + S_v + d = 2S_v + S_v + S_v/2 = \frac{7}{2}S_v$$
$$r_i = S_i = 4S_v$$
$$r = r_v + r_i = \frac{7}{2}S_v + 4S_v = \frac{15}{2}S_v$$

substitution on (0)

$$rKa = R$$
$$\frac{15}{2}S_v K a = R$$
$$\boxed{S_v = \frac{2R}{15Ka}} \quad (3)$$

a is a constant, R, but K is not. We don't want $S_v$ in function of K, this says us something we know: the number of rows depends on the number of colums both. We can simplify it. Let's assume the rows and cols have a 4:3 proportion. Here we work only with visible memory.

By proportions

$$\frac{r_v}{3} = \frac{K}{4}, \quad r_v = \frac{3}{4}K, \quad K = \frac{4}{3}r_v$$

By area

$$r_v K a = R_v$$
$$\frac{3}{4}K K a = R_v$$
$$K^2 = \frac{4R_v}{3a}$$
$$K = \sqrt{\frac{4R_v}{3a}}, \quad K > 0$$
$$K = 2\left\lfloor\sqrt{\frac{R_v}{3a}}\right\rfloor, \quad K \in \mathbb{N} \quad (4)$$

We need K in function of R instead of $R_v$. $R_v$ and $R_i$ are proportions of R.

$$R_i = S_i K a = 4S_v K a$$
$$R_v = r_v K a = \frac{7}{2}S_v K a$$
$$R = R_i + R_v = \left(4 + \frac{7}{2}\right)S_v K a = \frac{15}{2}S_v K a$$
$$\%R_v = \frac{R_v}{R} = \frac{7/2\, S_v K a}{15/2\, S_v K a} = \frac{7}{15} \qquad S_v > 0,\ a > 0.$$
$$R_v = \frac{7}{15}R, \quad R_i = \left(1 - \frac{7}{15}\right)R = \frac{8}{15}R$$

Substituting in (4)

$$K' = 2\left\lfloor\sqrt{\frac{R_v}{3a}}\right\rfloor = 2\left\lfloor\sqrt{\frac{7R}{15 \cdot 3a}}\right\rfloor =$$
$$\boxed{K' = 2\left\lfloor\sqrt{\frac{7R}{45a}}\right\rfloor} = \boxed{\left\lfloor\frac{2}{3}\sqrt{\frac{7R}{5a}}\right\rfloor} \quad \boxed{K = \left\lfloor\frac{K'}{c}\right\rfloor c} \quad (5)$$

substituting in (3)

$$S_v = \frac{2R}{15Ka} = \frac{2R}{15\frac{2}{3}\left\lfloor\sqrt{\frac{7R}{5a}}\right\rfloor a} = \frac{R\sqrt{\frac{7R}{5a}}}{5a\frac{7R}{5a}} = \frac{1}{7}\sqrt{\frac{7R}{5a}} = \sqrt{\frac{R}{35a}}$$
$$\boxed{S_v = \left\lfloor\sqrt{\frac{R}{35a}}\right\rfloor} \quad S_v = S_v K a =$$
$$h = 2S_v =$$

Figura 6.10. Modelo matemático de distribución de memoria en el prototipo C++ (imagen escaneada, rotar la página para leer)

Para presentar la estructura general, el Listado 6.2 muestra estos elementos con su contenido reemplazado por puntos suspensivos. Se describe cada elemento con su contenido y cómo botNeumann++ lo visualiza en este y el siguiente apartado.

El elemento `description` especifica el enunciado del problema. Un problema puede tener varios de estos elementos, uno por cada idioma (atributo `lang`) al que se traduzca el ejercicio. El Listado 6.5 muestra el enunciado en español del problema de convertir temperaturas. El prototipo C++ soporta enunciados en texto puro o un subconjunto de *HTML* (del inglés, *HYPERTEXT MARKUP LANGUAGE*). La forma en que el prototipo C++ presenta este enunciado al usuario se rotula con ⑤ en la Figura 6.9.

```
1 <description lang="es"><![CDATA[
2 <h2>Fahrenheit a Kelvin</h2>
3 <p>Para propósitos científicos, se requiere un programa que convierta
4 Fahrenheit a Kelvin. Esta conversión se hace con la relación
5 <i>K = 5/9*(F + 459.67)</i>.</p>
6
7 <h3>Ejemplo de entrada:</h3>
8 <pre>100</pre>
9
10 <h3>Ejemplo de salida</h3>
11 <pre>310.928</pre>
12 ]]></description>
```

Listado 6.5. Descripción de un ejercicio de programación (enunciado del problema)

Uno de los problemas de usabilidad más relevantes de corregir en el prototipo PowerPoint fue la legibilidad del enunciado del problema. El prototipo C++ permite al usuario desplazarse por el enunciado (en inglés, *SCROLLING*), re-dimensionar la ventana del enunciado, reubicar las ventanas en la interfaz, o ajustar el tamaño de la tipografía. El rótulo ① de la Figura 6.11 muestra la descripción del problema reubicado en la esquina superior derecha con un tamaño de tipografía mayor. Puede notarse cómo el área de visualización se reajusta para ocupar el espacio libre, respecto a la Figura 6.9.

Tras la descripción del problema en el archivo de nivel, el diseñador del ejercicio puede proveer cero o más códigos iniciales, como el del Listado 6.6. El prototipo C++ escogerá uno al azar para el usuario y lo cargará en el editor de código (rotulado ② en la Figura 6.11). Si no se proveen códigos iniciales, el prototipo mostrará un editor vacío.

Figura 6.11. Reubicación de las ventanas en el prototipo C++

Un código inicial permite al investigador proveer programas con errores o con otras características para estudiar conceptos de programación. Por ejemplo, el código inicial del Listado 6.6 tiene interferencia de dos variables `temperatura`, una global (línea 6 del Listado 6.6, visualizada ③ en el segmento de datos de la Figura 6.11) y otra local (línea 11 del Listado 6.6, visualizada ④ en el segmento de pila de la Figura 6.11). La funcionalidad de asignar códigos iniciales al azar se diseñó para facilitar al investigador la evaluación del efecto de estos códigos en experimentos controlados. Una vez que se ha asignado un código inicial, el usuario puede modificarlo en el editor de código, por ejemplo, para corregir sus errores.

A diferencia del prototipo PowerPoint que sólo permite modificar tres líneas en dos niveles, el prototipo C++ implementa un editor de código fuente tradicional. Para conveniencia de la experiencia de usuario, éste dispone de edición texto arbitrario, un portapapeles, una pila de cambios (en inglés, *UNDO STACK*), resaltado de sintaxis (en inglés, *SYNTAX HIGHLIGHTING*), autoguardado, y establecer puntos de parada (en inglés, *BREAKPOINTS*) en los números de línea. Una vez que usuario ha escrito código, puede activar la animación lo que cambia el estado de la pantalla de nivel.

```
 1 <initial-code lang="cpp"><![CDATA[
 2 #include <iomanip>
 3 #include <iostream>
 4 using namespace std;
 5
 6 double temperatura;
 7
 8 int main()
 9 {
10   cin >> temperatura;
11   double temperatura = 0.0;
12
13   temperatura = 5 / 9 * (temperatura + 459.67);
14
15   cout << fixed << setprecision(3);
16   cout << temperatura << endl;
17 }
18 ]]></initial-code>
```

Listado 6.6. Código C++ inicial dado al usuario en el ejercicio de conversión de temperaturas

## B.4.1 Estados de la pantalla de nivel

La pantalla de nivel puede estar en uno de varios estados, como editar código o animar el programa del usuario. Encima del editor de código se ubican tres botones que cambian la pantalla de nivel de un estado a otro (⑤ en la Figura 6.11). La máquina de estados de la pantalla de nivel se comporta como un reproductor multimedia y se diagrama en la Figura 6.12. Sobre cada estado de la Figura 6.12 se indican los botones de control que están habilitados al usuario.

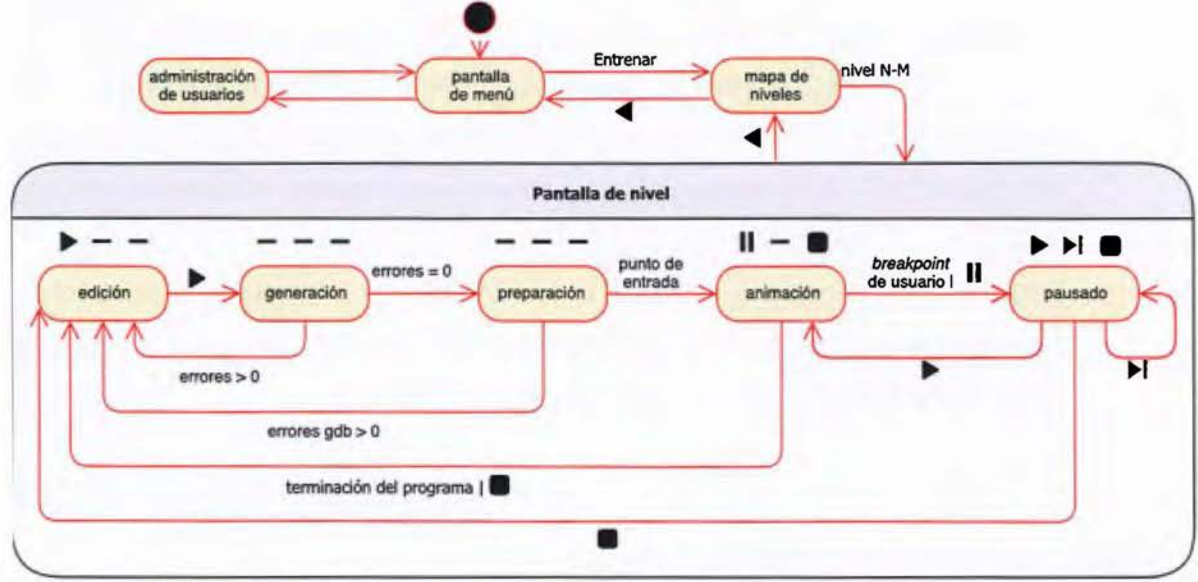



Figura 6.12. Máquina de estados de la pantalla de nivel en el prototipo C++

Mientras el usuario modifica el código fuente para resolver el problema, se dice que la pantalla de nivel se encuentra en **estado de edición**. Es el estado descrito hasta el momento en esta subsección. El único botón de control habilitado en estado de edición es el de reproducir (▶) que permite pasar a la generación de archivos.

### B.4.2 Estado de generación

Cuando el usuario acciona el botón de reproducir (▶) en el estado de edición, la pantalla de nivel pasa al **estado de generación** (en inglés, *BUILDING*). Durante este estado se generan los archivos necesarios para realizar la animación del programa del estudiante. Cuando el botón de reproducir se presiona, el prototipo realiza tres tareas concurrentemente como se resume en el diagrama de flujo de la Figura 6.13:

1. *Generar el ejecutable.* Para cada archivo fuente que forme parte de la solución del estudiante, el prototipo invoca al compilador de C o C++ de acuerdo a la extensión de los archivos con el fin de generar sus respectivos archivos objeto (con extensión ".o"). Si ninguna invocación del compilador reporta errores, se ejecuta el enlazador (en inglés, *LINKER*) para que ensamble el ejecutable a partir de los archivos objeto. El prototipo lista los errores o advertencias que generen los compiladores y el enlazador en el área de mensajes, ubicado por defecto en la parte inferior de la pantalla de nivel. Por ejemplo, la Figura 6.14 es una captura de pantalla del error de compilación reportado al eliminar la variable global en el ejercicio de conversión de temperaturas. Si el usuario activa un reporte, el prototipo C++ selecciona la línea que lo generó en el editor de código.
2. *Extraer símbolos globales.* Consiste en extraer los nombres de las subrutinas y variables globales de los archivos encabezado y fuente de la solución del usuario con la herramienta *UNIVERSAL CTAGS*[32]. Estos nombres son necesarios para visualizar el segmento de datos y animar invocaciones de subrutinas.
3. *Generar casos de prueba.* Consiste en generar archivos de entrada y sus respectivas salidas esperadas para probar la solución del estudiante. Es la primera tarea del motor de casos de prueba y a continuación se explica cómo es realizada.

[32] https://ctags.io/

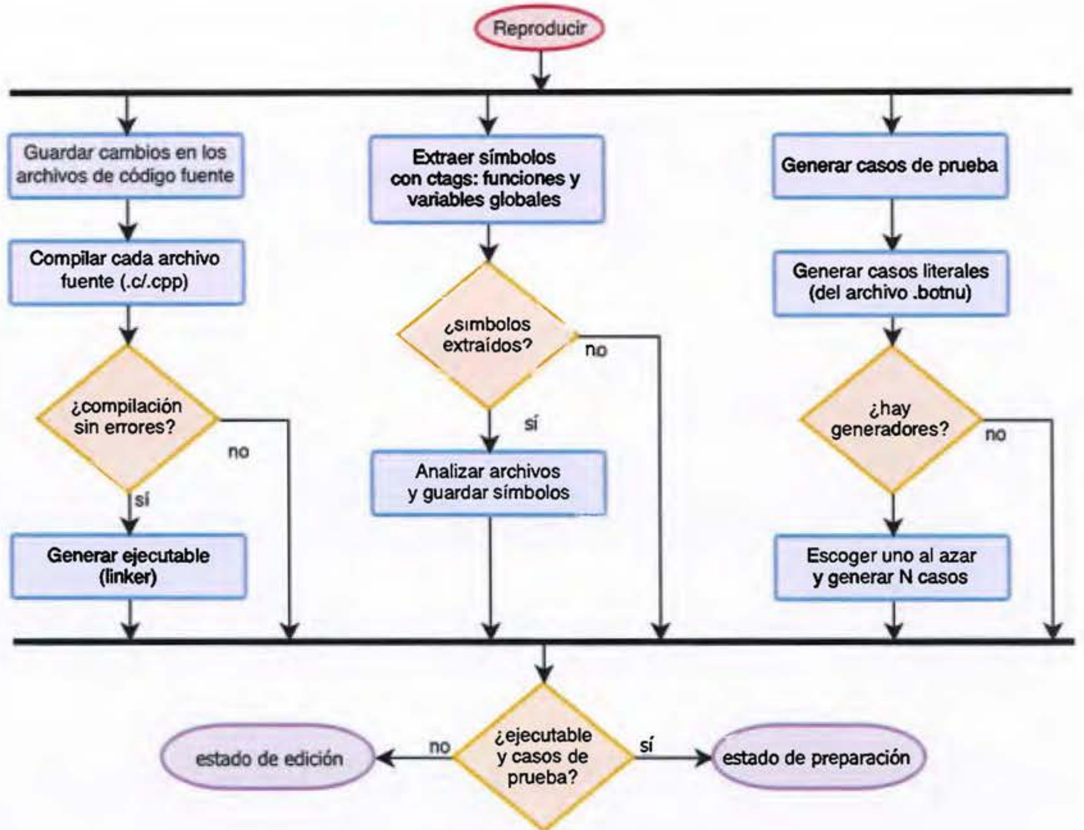



Figura 6.13. Proceso de generación de archivos para la animación

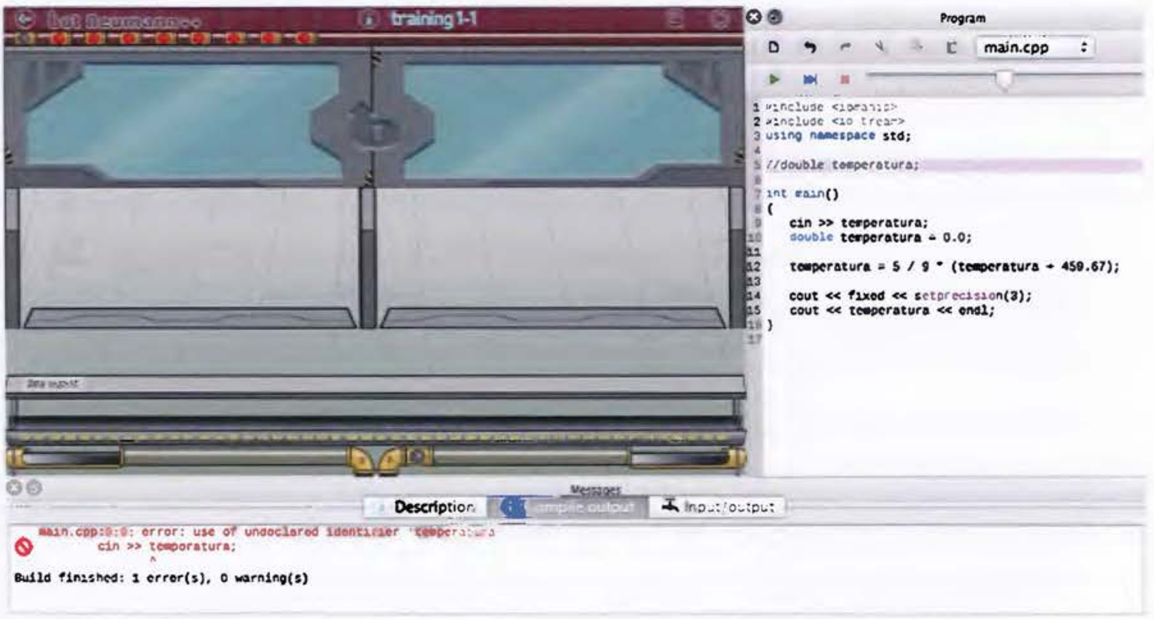



Figura 6.14. Reporte de un error de compilación durante la generación de archivos

El diseñador del ejercicio puede incluir en el archivo de nivel (con extensión "`.botnu`") casos de prueba literales o generadores de casos de prueba. Los casos de prueba literales, como los del Listado 6.7, pueden contener un atributo `args` opcional que indica los argumentos en línea de comandos y constan de tres elementos: una entrada obligatoria (`input`), una salida esperada obligatoria (`output`), y una salida de error esperada opcional (`error`).

```
1 <test-case>
2   <input>100</input>
3   <output>310.928</output>
4 </test-case>
5 <test-case>
6   <input>32</input>
7   <output>273.150</output>
8 </test-case>
9 <test-case>
10   <input>0</input>
11   <output>255.372</output>
12 </test-case>
13 <test-case>
14   <input>-273.15</input>
15   <output>103.622</output>
16 </test-case>
```

Listado 6.7. Cuatro casos de prueba literales para el ejercicio de conversión de temperaturas

El motor de casos de prueba extrae el contenido de cada elemento de un caso de prueba a un archivo de texto en la misma carpeta donde se genera el ejecutable del usuario. Por ejemplo, el prototipo genera el archivo `bn_02_output_ex.txt` para la salida esperada del segundo caso de prueba con el contenido 273.150 de acuerdo al Listado 6.7. La solución del estudiante se prueba contra todos los casos de prueba literales que provea el diseñador. Por tanto, son especialmente útiles para incluir los casos usados en el enunciado del problema y para probar valores extremos o interesantes.

Opcionalmente el archivo de nivel puede especificar generadores y soluciones. Ambos contienen código fuente de programas en C/C++ que botNeumann++ compila cuando se presiona el botón de reproducir. El prototipo C++ admite dos tipos de generadores: estándar y de archivo. Un **generador estándar** es un programa que cuando corre, escribe en su salida estándar (de ahí su nombre) el texto que será la entrada de un caso de prueba. Por ejemplo, cuando el programa del Listado 6.8 corre, genera una temperatura al azar entre -100.0 y 150.9 en la salida estándar. botNeumann++ envía al generador estándar por parámetro el número de caso de prueba y la cantidad total de casos de prueba que serán generados. El

generador puede usar esta información, por ejemplo, para crear casos de prueba de dificultad progresiva. Si el generador estándar imprime algún texto en su error estándar, botNeumann++ lo considerará como los argumentos en línea de comandos que recibirá la solución del estudiante.

```
1 <standard-generator lang="cpp" default-runs="5"><![CDATA[
2 #include <cstdlib>
3 #include <ctime>
4 #include <iostream>
5
6 double grand(int min, int max)
7 {
8   return rand() % (max - min) + min + rand() % 10 / 10.0;
9 }
10
11 int main()
12 {
13   srand( time(0) + clock() );
14   std::cout << grand(-100, 150) << std::endl;
15
16   return 0;
17 }
18 ]]></standard-generator>
```

Listado 6.8. Un programa generador de casos de prueba

Si hay varios generadores en el archivo de nivel, botNeumann++ escoge uno al azar, lo compila y el ejecutable es invocado tantas veces como el atributo `default-runs` indique para generar esta cantidad de casos de prueba. Así, una solución del estudiante será probada contra casos de prueba cambiantes cada vez que presione el botón de reproducir, lo que evita que la solución se adapte a casos de prueba estáticos.

Un generador estándar produce la entrada y los argumentos del caso de prueba. Para generar la salida esperada correspondiente, se requiere al menos una solución al problema. Una **solución** es un programa en C/C++ que resuelve el problema planteado en el ejercicio. El Listado 6.9 muestra una solución al ejercicio de conversión de temperaturas. Si hay varias soluciones, botNeumann++ escoge una al azar, la compila y la ejecuta re-direccionándole su entrada estándar a la entrada producida por el generador estándar. La salida que produzca la solución se considerará la salida esperada del caso de prueba.

Los generadores estándar tienen la limitación de que no pueden producir la salida de error estándar esperada. Un generador de archivo resuelve este problema. Un **generador de archivo** es un programa en C/C++ que recibe los cuatro nombres de archivo por argumentos

de línea de comandos, los crea y escribe en ellos el texto esperado. botNeumann++ provee una interfaz de programación de aplicaciones (*API*, del inglés *APPLICATION PROGRAMMING INTERFACE*) para C y C++ que facilita la manipulación de estos archivos. Los generadores de archivo son los más versátiles y están documentados en el *DTD* en el repositorio de control de versiones del prototipo[33].

```
1 <solution lang="cpp"><![CDATA[
2 #include <iomanip>
3 #include <iostream>
4 using namespace std;
5
6 int main()
7 {
8   double temperatura = 0.0;
9
10   cin >> temperatura;
11   temperatura = 5.0 / 9 0 * (temperatura + 459.67);
12
13   cout << fixed << setprecision(3);
14   cout << temperatura << endl;
15 }
16 ]]></solution>
```

Listado 6.9. Un programa en C++ que resuelve el problema de conversión de temperaturas

La generación de los casos de prueba es una tarea concurrente a la generación del ejecutable a partir de la solución del estudiante y a la extracción de símbolos (Figura 6.13). Cuando termina la última de estas tres tareas, el prototipo C++ revisa si se generó un ejecutable y al menos un caso de prueba. En caso negativo, la pantalla de nivel retorna automáticamente al estado de edición. En caso de éxito, la pantalla pasa silenciosamente al estado de preparación.

### B.4.3 Estado de preparación

En el **estado de preparación** (Figura 6.12, p.267) botNeumann++ toma los archivos producidos durante el estado de generación e invoca los programas requeridos para iniciar la animación de la solución del estudiante. En este estado, el motor de casos de prueba (Figura 6.2, p.252) realiza su segunda tarea: correr una instancia del ejecutable del estudiante con cada caso de prueba generado. Es decir, si en la etapa de generación se crearon *N* casos de prueba, el motor correrá *N* instancias del programa del estudiante concurrentemente en el estado de preparación.

---

[33] https://github.com/citic/botNeumann/blob/master/units/botnu-1.0.dtd

La Figura 6.15 muestra los cuatro archivos que conforman un caso de prueba encerrados en un rectángulo punteado. Cada instancia del programa del usuario se invoca con los argumentos de línea de comandos y el contenido del archivo de entrada re-direccionado a su entrada estándar. La salida estándar y los errores reportados por el programa del estudiante son capturados y comparados contra la salida esperada y el error esperado del caso de prueba. Si ambos coinciden, la solución del estudiante pasa el caso de prueba. Un selector (Figura 6.16) indica con colores al estudiante los casos de prueba que pasaron (verde), fallaron (rojo), o están en proceso (gris).

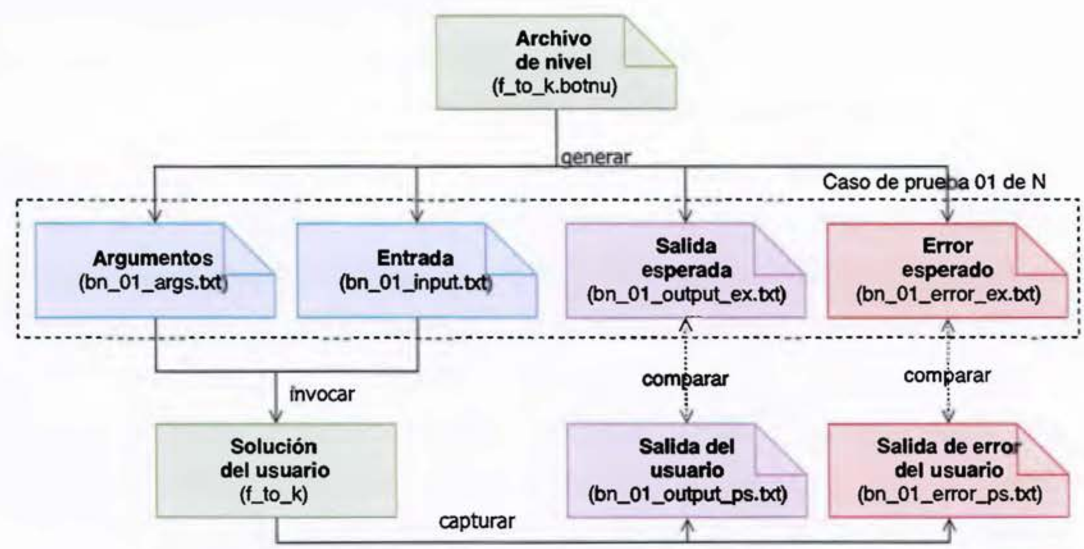



Figura 6.15. Ejecución de una instancia del programa del usuario contra un caso de prueba

El primer caso de prueba es escogido por defecto para visualizar. El usuario puede cambiarlo al accionar el dispositivo luminoso de cualquier otro caso de prueba en el selector (Figura 6.16). Es probable que el usuario realice este cambio para estudiar un caso de prueba fallido. Indiferentemente de que se visualice el caso por defecto o uno escogido por el usuario, la pantalla de nivel pasa al estado de animación.

Figura 6.16. Selector con trece casos de prueba

## B.4.4 El estado de animación

En el estado de animación (Figura 6.12, p.267), una instancia del programa del usuario que corre contra un caso de prueba es visualizada por el prototipo C++. Aunque todos los módulos

de la arquitectura intervienen durante la animación, sólo dos lo hacen en este estado: (1) el meta-modelo que se encarga de extraer información de la instancia del programa del usuario y (2) la arquitectura interna que usa la información extraída para crear las animaciones.

## B.5 El meta-modelo

Siguiendo la convención de [Lanza 2003], el meta-modelo es el módulo que se encarga extraer y estructurar los datos que serán visualizados (véase la Figura 6.1, p.250). Como puede apreciarse en la línea de tiempo (Figura 6.3, p.253), este módulo fue el que requirió más esfuerzo de desarrollo y el que generó más lecciones aprendidas.

De acuerdo a Egan y McDonald, los desarrolladores de SeeC (apartado 3.1.2.2, p.68), las visualizaciones de programa aprovechan infraestructuras existentes para la depuración de código, como es el caso de la *JAVA PLATFORM DEBUGGER ARCHITECTURE*[34] [Egan and McDonald 2013, p.5]. Estas infraestructuras permiten a una visualización de programa, obtener información sobre un programa de usuario en ejecución de forma conveniente a través de una *API*. Como indican estos autores, dado que al año 2013 no existía una infraestructura de depuración para C/C++, las visualizaciones de programa de esta máquina nocional tenían que recurrir a alguna alternativa, como las cuatro siguientes [Egan and McDonald 2013, p.5]:

1. Crear un intérprete de C/C++ hecho a la medida, usualmente de un subconjunto del lenguaje, lo cual limita severamente la utilidad y adopción de la herramienta.
2. Utilizar un depurador como el *GNU DEBUGGER* (*GDB*)[35]. Tiene la ventaja de no limitar el lenguaje de programación. *GDB* permite a la visualización obtener información en tiempo de ejecución del programa del usuario, si éste se compila con información de depuración. Sin embargo, esta información no es tan precisa como la visualización requiere y *GDB* ofrece una interfaz limitada para comunicarse con otros programas.
3. Utilizar instrumentalización binaria. Herramientas como *VALGRIND*[36], *DYNAMORIO*[37], o *PIN*[38] permiten a otras estudiar y modificar código binario sin tener que recompilarlo. Esta

[34] https://docs.oracle.com/javase/8/docs/technotes/guides/jpda/architecture.html
[35] https://www.gnu.org/software/gdb/
[36] http://valgrind.org/
[37] http://dynamorio.org/

alternativa tiene la desventaja de que mucha de la información semántica del programa necesaria para la visualización se pierde al generar el código binario, incluso aunque se almacene información de depuración en él.

4. Utilizar una infraestructura de compilación, en particular *LLVM*[39]. Es la alternativa que Egan y McDonald tomaron para construir el meta-modelo de SeeC. Estos autores adaptaron el compilador de C (*CLANG*) para que genere un ejecutable modificado a partir del código fuente del usuario. Cuando el ejecutable modificado corre, además de realizar su trabajo habitual, genera un archivo con extensión `.seec` que contiene las instrucciones de alto nivel que fueron ejecutadas. Este archivo `.seec` puede considerarse un video que es reproducido por el visor de SeeC (Figura 6.17). Este visor permite algunas operaciones básicas de un reproductor de medios, como adelantar o retroceder, uno, varios o todos los pasos a la vez con los botones que dispone en la parte superior. La Figura 6.17 reproduce una ejecución del código fuente inicial del ejercicio de conversión de temperaturas traducido a C y al que se le dio el valor **`100.0`** en la entrada estándar.

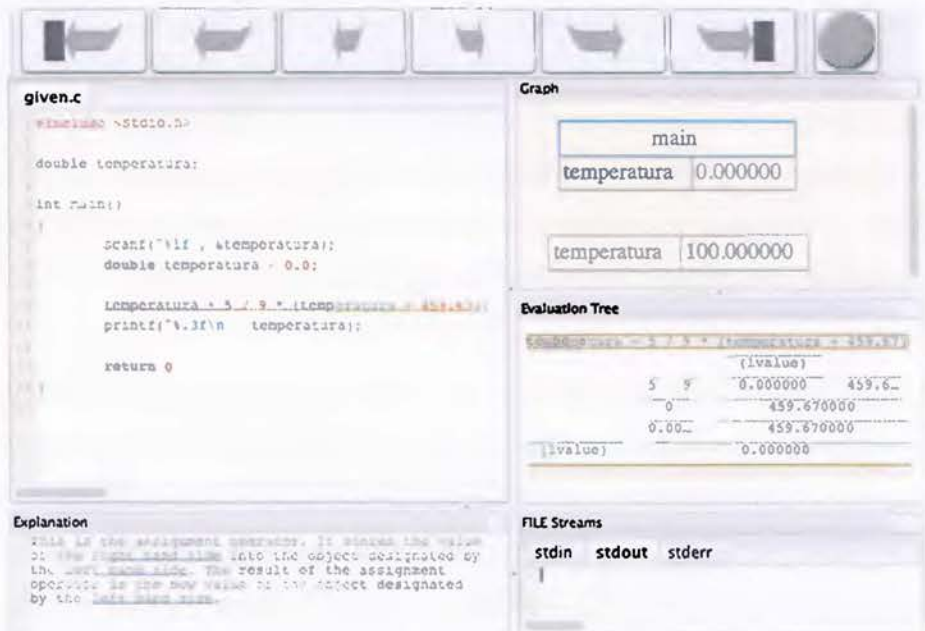



Figura 6.17. Programa de conversión de temperaturas en C visualizado por SeeC

[38] https://software.intel.com/en-us/articles/pintool
[39] https://llvm.org/

Se infiere del panorama esbozado en [Egan and McDonald 2013, p.5], que una *infraestructura de depuración* es la herramienta más conveniente para implementar el meta-modelo de una visualización de programa. En 2013, el mismo año que dicho artículo fue publicado, Apple Inc. liberó con la quinta versión de su ambiente de desarrollo (*XCODE*) con un nuevo depurador, *LLDB*[40], construido desde cero con tecnología de *LLVM* para reemplazar a *GDB*. Al momento de iniciar la implementación del meta-modelo de botNeumann++, en 2015, *LLDB* debió declinarse debido a que el depurador sólo estaba implementado en condición funcional para *MACOS* y la Escuela de Ciencias de la Computación e Informática no dispone de un laboratorio de computadoras con este sistema operativo donde pudiera efectuarse un experimento controlado. Al momento de finalizar esta tesis (2018), la mayoría de la funcionalidad de *LLDB* ha sido portada a Linux y Microsoft Windows, lo que abre una oportunidad para investigación futura.

La segunda tecnología que se evaluó para implementar el meta-modelo fue la *infraestructura de compilación LLVM*, como optaron los autores de SeeC, pero con un enfoque distinto. La estrategia de modificar los ejecutables de los estudiantes para generar videos no se consideró apropiada debido a que limita la naturaleza interactiva de la visualización de programa (requerimiento 2: estado activo), como ocurre con el visor de SeeC (Figura 6.17), limitado a ser un reproductor de video simplificado. Sin embargo, se conjetura que esta limitación podría superarse si se guarda en el video suficiente información para responder a un conjunto de potenciales interacciones del usuario. Por tanto, es una oportunidad que puede explorarse en el futuro.

La evaluación de *LLVM* se inclinó hacia uno de sus intérpretes como se explica a continuación. *LLVM* consta de dos grandes módulos, la parte frontal (en inglés, *FRONT-END*) que traduce código de un lenguaje de programación (por ejemplo, C/C++) a una representación intermedia, y la parte trasera (en inglés, *BACK-END*) que realiza operaciones con la representación intermedia, como análisis estático o traducción a código objeto. Entre las herramientas del segundo módulo, *LLVM* dispone de un compilador justo a tiempo (*JIT*, del inglés *JUST IN TIME COMPILER*) que puede interpretar código en representación intermedia. Sin embargo, tuvo también que descartarse debido a que este intérprete evalúa subrutinas y obtiene su valor de retorno [Cardoso Lopes and Auler 2014, p.178], pero no ejecuta

---

[40] http://lldb.llvm.org/

instrucciones de alto nivel, una a la vez, lo cual es necesario para una visualización de programa.

Ante la imposibilidad de usar herramientas de depuración y compilación modernas para C/C++, en mayo de 2015 se decidió implementar el meta-modelo utilizando el depurador *GDB*. Dadas sus limitaciones conocidas para el desarrollo de visualizaciones de programa, se realizó un proceso de diseño del meta-modelo y la arquitectura interna que tardó cerca de un año, como puede verse en la línea de tiempo de la Figura 6.3 (p.253). Durante este período se realizaron las siguientes macro-actividades:

1. *Familiarización con la herramienta a través de su documentación oficial*[41]. *GDB* permite a un programa en ejecución, llamado **superior** (la visualización de programa), controlar la ejecución de otro programa, llamado **inferior** (el programa del estudiante), a través de *GDB*. En este modelo, la visualización de programa ejecuta una instancia de *GDB* y se comunica con ella a través de flujos de entrada y salida estándar, en lugar de una *API*[42]. En forma muy general *GDB* permite observar, y hasta cierto punto controlar, los hilos de ejecución del inferior y sus respectivos segmentos de pila. *GDB* no provee facilidades para estudiar los demás segmentos ni flujos de entrada o salida, lo cual es una limitación importante para una visualización de programa.
2. *Conocimiento de la interfaz de máquina*. *GDB* reconoce dos interfaces a través de los flujos de entrada y salida: (1) la interfaz de usuario diseñada para humanos, y (2) la interfaz para máquinas llamada *GDB/MI* (del inglés *GDB MACHINE INTERFACE*). Desdichadamente *GDB/MI* soporta sólo un subconjunto limitado de la interfaz de usuario, y por tanto, de la funcionalidad de *GDB* [Egan and McDonald 2013, p.5]. El Listado 6.10 muestra un fragmento de la comunicación entre el prototipo C++ y *GDB* usando la interfaz de máquina, cuando se inicia la animación del ejercicio de conversión de temperaturas[43]. Los comandos en negrita en el Listado 6.10 los envía botNeumann++ a la entrada estándar de *GDB* y los textos más claros son las respuestas que *GDB* genera en la salida estándar. Los comandos y las respuestas son asincrónicas, por lo que se suelen desfasar. Se preceden

---

[41] Libro *DEBUGGING WITH GDB*, disponible en https://sourceware.org/gdb/onlinedocs/
[42] Una iniciativa de crear una *API* para *GDB*, llamada `libgdb`, fue abandonada en 1993: https://www.gnu.org/software/gdb/papers/libgdb/libgdb.html
[43] Una descripción de los comandos y las respuestas se encuentra en el documento `docs/gdb/session_example.html` del repositorio de control de versiones.

con números para ayudar a re-asociarlas. Como puede verse, especialmente en la línea 11 del Listado 6.10, la sintaxis de las respuestas de *GDB/MI* es similar a la notación de objetos de *JAVA*Script (*JSON*, del inglés, *JAVASCRIPT OBJECT NOTATION*), aunque ambas tecnologías son distantes en el tiempo. Para poder enviar comandos a *GDB*, reordenar, y analizar (en inglés, *PARSING*) sus respuestas, se requiere una capa de comunicación.

3. *Adaptación de una capa de comunicación existente*. Dado que el prototipo C++ se construyó sobre Qt, se intentó primero adaptar la capa de comunicación de QtCreator con el depurador. QtCreator es el ambiente de desarrollo integrado (*IDE*, del inglés *INTEGRATED DEVELOPMENT ENVIRONMENT*) oficial de Qt. Sin embargo, se excluyó por su enorme complejidad debida a la compatibilidad con otros depuradores soportados por este *IDE*. Se intentó luego con la capa de comunicación de *GEDE*[44], una aplicación de interfaz gráfica minimalista para *GDB* en Linux, la cual fue implementada en Qt. El código de la capa de comunicación con *GDB* tuvo que ser reestructurado para aislarlo de *GEDE*, optimizarlo, y aplicarle patrones de desarrollo de Qt. La adaptación de esta capa de comunicación fue exitosa, incluso en el proceso de reingeniería se logró portar a *MACOS* y *MICROSOFT WINDOWS*.

```
1 gdb -q -i mi
2 =thread-group-added,id="i1"
3 (gdb)
4 1-file-exec-and-symbols "/tmp/bn_Jugador/e1-f_to_k/e1-f_to_k"
5 1^done
6 (gdb)
7 2-exec-arguments
      < "/tmp/bn_Jugador/e1-f_to_k/bn_01_input.txt"
      > "/tmp/bn_Jugador/e1-f_to_k/bn_01_output_ps_v.txt"
     2> "/tmp/bn_Jugador/e1-f_to_k/bn_01_error_ps_v.txt"
8 2^done
9 (gdb)
10 3-break-insert "/tmp/bn_Jugador/e1-f_to_k/main.cpp:7"
11 3^done,bkpt={number="1",type="breakpoint",disp="keep",enabled="y",addr="0x00000001
   000009c9",func="main()",file="/tmp/bn_Jugador/e1-
   f_to_k/main.cpp",fullname="/tmp/bn_Jugador/e1-f_to_k/main.cpp",line="7",thread-
   groups=["i1"],times="0",original-location="/tmp/bn_Jugador/e1-f_to_k/main.cpp:7"}
12 (gdb)
13 4-exec-run
14 ...
```

Listado 6.10. Fragmento de interacción entre botNeumann++ y *GDB/MI*. La línea 1 inicia una instancia de *GDB*. Línea 4 indica el ejecutable del estudiante a depurar. Línea 7 re-direcciona los flujos al caso de prueba. Línea 10 establece un *BREAKPOINT*. Línea 13 inicia la ejecución del programa del estudiante.

[44] http://gede.acidron.com/

Con el conocimiento sobre *GDB* y su interfaz para máquinas, se adaptó la capa de comunicación de *GEDE* y además se construyó una aplicación minimalista para probar la capa resultante. La aplicación de pruebas se llamó `GdbTest` y elaboró entre junio y octubre de 2016 como se aprecia en la línea de tiempo de la Figura 6.3 (p.253). Está disponible en la carpeta `tools/GdbTest` del repositorio de control de versiones. Dado que las pruebas fueron exitosas, se continuó en la línea del tiempo con la elaboración de la arquitectura interna.

## B.6 La arquitectura interna

La arquitectura interna es el módulo que se encarga de procesar los datos extraídos a través del meta-modelo y transformarlos en una representación interna que sea fácil de visualizar (véase la Figura 6.1, p.250) [Lanza 2003]. El principal obstáculo que se enfrentó para implementar la arquitectura interna es que *GDB* sólo ofrece información sobre los hilos de ejecución y sus pilas de invocaciones de funciones en el programa del estudiante. Para los demás segmentos y flujos de datos se tuvo que recurrir a mecanismos indirectos e imprecisos para obtener los datos que el motor de visualización requiere, aunque fueran aproximados.

El diseño de la arquitectura interna se escribió como un algoritmo, el cual usa los mecanismos de depuración de *GDB*, como puntos de parada (*BREAKPOINTS*) y vigilancia de expresiones (*WATCHES*) para tratar de obtener indirectamente los datos del programa. El algoritmo resultante no se incluye ya que ocuparía 44 páginas de esta tesis, sin embargo se puede consultar en el repositorio de control de versiones[45].

La Figura 6.18 muestra la máquina de estados de la animación. Los estados a la izquierda de la línea punteada provienen de la Figura 6.12 (p.267). Durante la *preparación* se corre una instancia del programa del estudiante por cada caso de prueba generado. Uno de esos casos es el animado a través de los estados de la Figura 6.18. En lugar de correr directamente, la instancia del programa del estudiante corre como el proceso inferior de *GDB*. En la *preparación* se invoca la instancia de *GDB*, se le indica la ruta del programa del estudiante, sus parámetros, se le re-direcciona la entrada y salida, y se establecen dos tipos de puntos de parada (*BREAKPOINTS*): los definidos por el usuario en el editor de código y las definiciones de cada subrutina (función) extraída con la herramienta *UNIVERSAL CTAGS*. Los puntos de parada

[45] https://github.com/citic/botNeumann/blob/master/docs/gdb/session_use_case.adoc

de subrutinas son un mecanismo indirecto para detectar cuándo se invoca una subrutina, ya que *GDB* no provee esta información directamente.

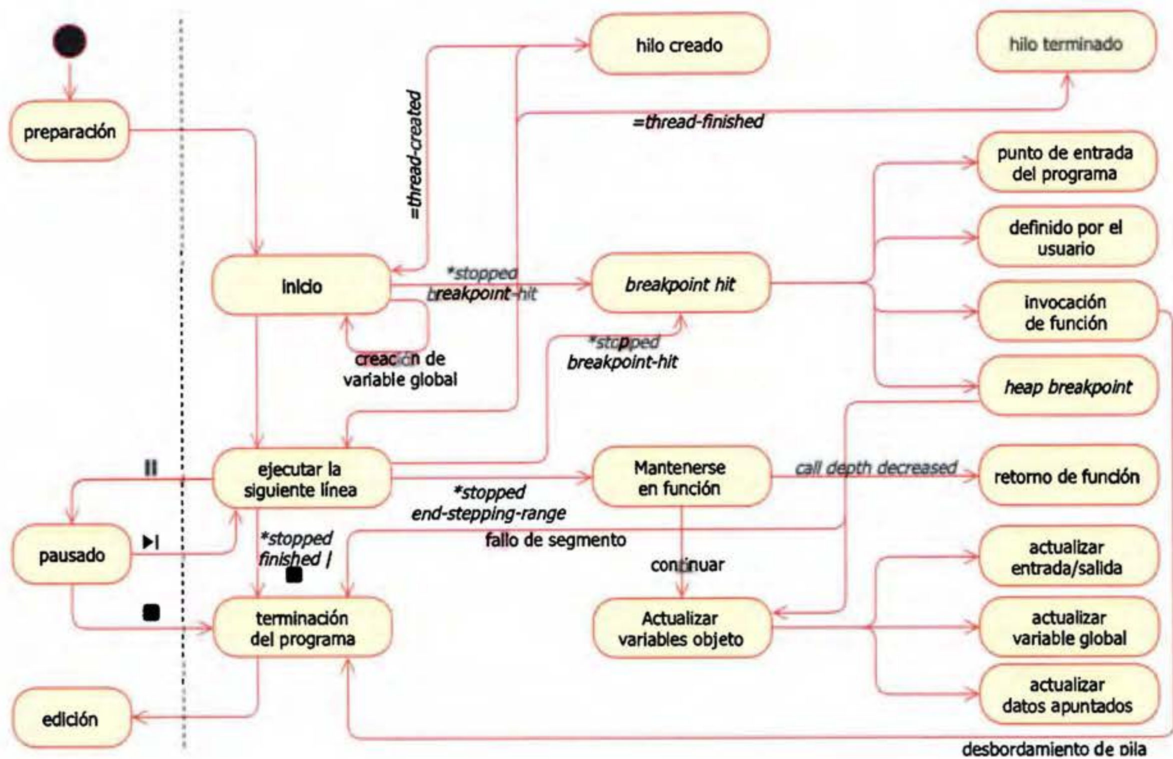



Figura 6.18. Máquina de estados de la animación

Una vez que *GDB* está preparado, se le indica que corra la instancia el programa del estudiante (inferior) en el estado *inicio* de la Figura 6.18. Mientras el programa del estudiante corre, *GDB* genera mensajes como la creación de un hilo de ejecución o su terminación. El prototipo C++ reacciona a estos mensajes animando la creación de un robot en una estación de trabajo libre (estado *hilo creado*), como se ve en la Figura 6.19, o removiéndolo de estas áreas (estado *hilo terminado*). Estos son mensajes directos de *GDB* que no requieren un mecanismo indirecto.

*GDB* corre el programa del estudiante silenciosamente, hasta que éste termina su ejecución. *GDB* no lo detiene a menos de que reciba una señal del sistema operativo o alcance un punto de parada. Como el prototipo C++ establece puntos de parada para cada declaración de subrutina reportada por *CTAGS*, *GDB* se detendrá cada vez que el control ingrese en una de ellas y reportará el mensaje `*stopped` que indica que el programa del estudiante no está corriendo sino que está detenido. Si la razón por la que el programa se detuvo es un punto de parada (`breakpoint-hit`), el prototipo C++ pasará al estado *BREAKPOINT-HIT* en la Figura 6.18. En este estado se revisa el tipo de punto de parada que detuvo al programa del estudiante. Si

es la primera invocación de subrutina que ocurre, el prototipo la considerará el punto de entrada del programa. En C siempre es la función `main()`, pero en C++ puede ser un constructor de un objeto global y botNeumann++ es la primera visualización de programa en animarlos.

Figura 6.19. Problema de conversión de temperaturas resuelto con dos hilos de ejecución

La animación del punto de entrada del programa se realiza como cualquier otra invocación de subrutina. Sin embargo, antes de animar el hilo de ejecución principal ejecutando el punto de entrada del programa, botNeumann++ realiza varias operaciones adicionales en el estado de *inicio*, como las siguientes:

1. Establece más puntos de parada, para las funciones de biblioteca de administración de memoria dinámica: `malloc`, `calloc`, `realloc`, y `free`.
2. Crea variables objeto (en inglés, *WATCHES*) para las variables globales reportadas por *CTAGS* y anima la creación de las variables en el segmento de datos.
3. Crea variables objeto de punteros en los flujos de entrada, salida, y error estándar, con el fin de detectar operaciones de lectura o escritura.
4. Por cada carácter en el archivo de entrada del caso de prueba, anima un carácter desplazándose por el tubo de entrada hasta que se acaben o se llene el tubo (en inglés, *BUFFERING*).

Una vez concluido el estado de inicio, el prototipo entra en el ciclo de animación en el que se *ejecuta la siguiente línea* del programa del usuario. Toda instrucción de un programa C/C++

tiene un efecto en la memoria, por tanto, un cambio debe animarse. Al ejecutar la siguiente línea puede ocurrir lo siguiente:

1. Se crea o finaliza un hilo de ejecución, que se anima con la creación o desaparición de un robot en la fábrica. Si no hay estaciones de trabajo suficientes el robot se alinea en espera por una de ellas.
2. Se alcanza un punto de parada. Si es un punto de parada definido por el usuario en el código fuente, la animación pasa al estado *pausado*. Si es una invocación de una subrutina, se anima una estantería surgiendo del sótano de la fábrica para el robot que inicia la nueva tarea, y se crean variables objeto (*WATCHES*) para cada variable local. Si es una subrutina de una función de biblioteca de administración de memoria dinámica, se anima la creación o destrucción de cápsulas en la bodega (esta animación no se llegó a implementar por delimitaciones de tiempo).
3. Cada hilo de ejecución mantiene el rastro de la cantidad de invocaciones de funciones en su pila (en inglés, *STACK DEPTH*). Si este número disminuye, el hilo de ejecución animará un retorno de función. Su estantería se regresa al sótano. Todas las variables objeto (*WATCHES*) asociadas a las variables locales son eliminadas.
4. Se actualizan todas las variables objeto de *GDB*. Las variables objeto (en inglés, *WATCHES*) sirven para detectar si algún lugar de la memoria cambia de valor, lo cual es necesario para los siguientes tres casos.
5. Si una variable objeto asociada a la entrada, salida, o error estándar cambia de valor, se invoca una subrutina inyectada por botNeumann++ en el código fuente del estudiante para determinar la cantidad de bytes leídos o escritos, con el fin de animar una lectura desde el tubo de entrada hacia el robot que la realizó o desde el robot hacia el tubo de salida. Desdichadamente *GDB* no reporta cuál hilo de ejecución realizó la operación de lectura o escritura. Por tanto, el prototipo C++ recurre a análisis estático de código tratando de identificar funciones (como `fprintf`) u objetos de biblioteca estándar (como `cout`) en las líneas circundantes a la ejecutada por el hilo. Aunque este mecanismo indirecto funciona en la mayoría de casos, es muy impreciso y podría realizar una animación incorrecta.
6. Si una variable objeto asociada a una variable global o local cambia de valor, se anima su cápsula reemplazando el viejo valor por el nuevo.

7. Si una variable objeto asociada a un puntero o referencia cambia de valor, botNeumann++ actualiza ambas partes: el dato apuntado y el puntero usando una contabilidad de punteros y referencias. No se implementó en el prototipo por delimitaciones de tiempo.
8. El programa del usuario terminó su ejecución. Puede ocurrir por alcanzar la última línea de código (terminación normal), por un error (ejemplo, un desbordamiento de pila o fallo de segmento), por una señal del sistema operativo, o que el usuario presionara el botón para detener la animación.
9. Ninguna de las anteriores. En tal caso, *GDB* reporta el mensaje `*stopped, reason=end-stepping-range` que indica que el programa del estudiante ejecutó una línea y se detuvo de nuevo antes de ejecutar la próxima.

En los nueve eventos anteriores, excepto el ocho, la animación retorna al estado de ejecutar la siguiente línea, donde el usuario puede pausar o controlar la velocidad de la animación.

### B.6.1 El estado de pausa y velocidad de la animación

El usuario puede controlar la animación y ajustar su velocidad con los controles de la Figura 6.20. Por defecto, si el usuario no establece ningún punto de parada en el código fuente, el prototipo C++ corre el programa del usuario desde el punto de entrada hasta su finalización a la velocidad indicada por la barra deslizable de velocidad. Si el usuario desliza esta barra, el efecto en la velocidad de la animación es casi inmediato. A nivel de implementación, la barra de velocidad es un multiplicador invertido de las duraciones de las animaciones.

Si el usuario establece al menos un punto de parada (*BREAKPOINT*) en su código fuente, la animación correrá a máxima velocidad desde el punto de entrada del programa hasta que se alcance el punto de parada, momento en que la animación pasará al estado de pausa. En este estado, la animación se detiene hasta que el usuario presione uno de los tres botones de control. Si presiona el botón de reproducir, la animación se reanuda a la misma velocidad que tenía antes de entrar en el estado de pausa. Si se presiona el botón del siguiente paso, se animará la ejecución de una instrucción y la animación entrará en estado de pausa de nuevo. Si se presiona el botón de detener, la animación pasará al estado de terminación de programa con el fin de finalizar la sesión con *GDB* y retornar al estado de edición donde el usuario puede

modificar código fuente. Cabe aclarar que el prototipo C++ permite al usuario modificar su solución durante la animación, con el fin de maximizar la interacción con la herramienta.

Figura 6.20. Controles de la animación y su velocidad

# Referencias

ABERG, J., 2010. Challenges with teaching HCI early to computer students. In *Proceedings of the fifteenth annual conference on Innovation and technology in computer science education - ITiCSE '10*. Ankara, Turkey: ACM Press, pp. 3–7.

ACM AND IEEE COMPUTER SOCIETY, 2013. *Computer Science 2013: Curriculum Guidelines for Undergraduate Programs in Computer Science*,

AL-FEDAGHI, S. AND ALRASHED, A., 2014. Visualization of Execution of Programming Statements. In *2014 11th International Conference on Information Technology: New Generations*. IEEE, pp. 363–370.

ALLEVATO, A., EDWARDS, S.H. AND PÉREZ-QUIÑONES, M.A., 2009. Dereferee: exploring pointer mismanagement in student code. *ACM SIGCSE Bulletin*, 41(1), pp.173–177.

ALVAREZ, M.I. ET AL., 1998. *Computers in schools: a qualitative study of Chile and Costa Rica*,

ANTCZAK, M., BADURA, J.A.N., LASKOWSKI, A. AND STERNAL, T., 2018. A Survey on Online Judge Systems and Their Applications. *ACM Computing Surveys*, 51(1), pp.1–34.

BADILLA SAXE, E., 1991. Informática educativa en Costa Rica a partir de 1987. *Revista Educación*, 15(1).

BANGOR, A., KORTUM, P. AND MILLER, J., 2009. Determining what individual SUS scores mean: Adding an adjective rating scale. *Journal of usability studies*, 4(3), pp.114–123.

BEDNARIK, R., MYLLER, N., SUTINEN, E. AND TUKIAINEN, M., 2005. Effects of Experience on Gaze Behavior during Program Animation. In *17th Workshop of the Psychology of Programming Interest Group, Sussex University, June 2005*. pp. 49–61.

BEN-ARI, M. ET AL., 2011. A decade of research and development on program animation: The Jeliot experience. *Journal of Visual Languages & Computing*, 22(5), pp.375–384.

BENNEDSEN, J. AND CASPERSEN, M.E., 2007. Failure rates in introductory programming. *ACM SIGCSE Bulletin*, 39(2), p.32.

BERRY, M. AND KÖLLING, M., 2016. Novis: A notional machine implementation for teaching introductory programming. In *Fourth International Conference on Learning and Teaching in Computing and Engineering (LATICE 2016)*.

BERRY, M. AND KÖLLING, M., 2013. The design and implementation of a notional machine for teaching introductory programming. In *Proceedings of the 8th Workshop in Primary and Secondary Computing Education on - WiPSE '13*. New York, New York, USA: ACM Press, pp. 25–28.

BERRY, M. AND KÖLLING, M., 2014. The state of play: a notional machine for learning programming. In *Proceedings of the 2014 Conference on Innovation & ....* pp. 21–26.

BLACK, M., 1955. Metaphor. In *Proceedings of the Aristotelian Society*. Oxford University Press, pp. 273–294.

BLACK, M., 1977. More about metaphor. *Dialectica*, 31(3–4), pp.431–457.

BLACKWELL, A.F., 1998. *Metaphor in Diagrams*. University of Cambridge.

BLACKWELL, A.F., 1996. Metaphor or Analogy: How Should We See Programming Abstractions? In P. Vanneste, K. Bertels, B. De Decker, & J.-M. Jaques, eds. *Proceedings of the 8th Annual Workshop of the Psychology of Programming Interest Group*. pp. 105–113.

BLACKWELL, A.F., 2001. Pictorial Representation and Metaphor in Visual Language Design. *Journal of Visual Languages & Computing*, 12(3), pp.223–252.

BLACKWELL, A.F., 2006. The Reification of Metaphor as a Design Tool. *ACM Transactions on Computer-Human Interaction*, 13(4), pp.490–530.

BLACKWELL, A.F. AND GREEN, T.R.G., 1999. Does metaphor increase visual language usability? In *Proceedings 1999 IEEE Symposium on Visual Languages*. IEEE, pp. 246–253.

BOGOIAVLENSKI, D.N. ET AL., 1963. *Educational Psychology in the U.S.S.R* Print 2008. B. Simon & J. Simon, eds., Routledge and Kegan Paul Ltd.

BOGOYAVLENSKY, D.N. AND MENCHINSKAYA, N.A., 2011a. La psicología del aprendizaje desde 1900 a 1960. In *Psicología y pedagogía*. Sevilla: Ediciones Akal, pp. 119–188.

BOGOYAVLENSKY, D.N. AND MENCHINSKAYA, N.A., 2011b. Relación entre aprendizaje y desarrollo psicointelectivo del niño en edad escolar. In *Psicología y pedagogía*. Ediciones Akal, pp. 59–80.

BONK, C.J. AND CUNNINGHAM, D.J., 1998. Searching for Learner-Centered, Constructivist, and Sociocultural Components of Collaborative Educational Learning Tools. In C. J. Bonk & K. S. King, eds. *Electronic Collaborators: Learner-centered Technologies for Literacy, Apprenticeship, and Discourse*. New York, NY, USA: Routledge, pp. 25–50.

BOULAY, B. DU, O'SHEA, T. AND MONK, J., 1981. The black box inside the glass box: presenting computing concepts to novices. *International Journal of Man-Machine Studies*, 14(3), pp.237–249.

BROOKE, J., 1996. SUS: A "Quick and Dirty" Usability Scale. In P. W. Jordan, B. Thomas, B. A. Weerdmeester, & I. L. McClelland, eds. *Usability evaluation in industry*. London: Taylor and Francis, pp. 189–194.

BYCKLING, P. AND SAJANIEMI, J., 2006. Roles of variables and programming skills improvement. In *Proceedings of the 37th SIGCSE technical symposium on Computer science education*. Houston, Texas, USA: ACM, pp. 413–417.

CAFARO, F., LYONS, L., KANG, R., RADINSKY, J., ROBERTS, J. AND VOGT, K., 2014. Framed Guessability: Using Embodied Allegories to Increase User Agreement on Gesture Sets. In *Proceedings of the 8th International Conference on Tangible, Embedded and Embodied Interaction - TEI '14*. New York, New York, USA: ACM Press, pp. 197–204.

CAMERON, L., 2003. *Metaphor in educational discourse* 1st ed., New York: Continuum.

CAMPBELL, D.T. AND STANLEY, J.C., 1973. *Diseños experimentales y cuasiexperimentales en la investigación social*, Argentina: Amorrotu editores.

CARDOSO LOPES, B. AND AULER, R., 2014. *Getting Started with LLVM Core Libraries* 1st ed., Birmingham: Packt Publishing Ltd.

CARROLL, J.M. AND MACK, R.L., 1985. Metaphor, computing systems, and active learning. *International Journal of Man-Machine Studies*, 22(1), pp.39–57.

CARROLL, N., 1994. Visual Metaphor. In J. Hintikka, ed. *Aspects of Metaphor*. Dordrecht:

Springer, Dordrecht, pp. 189–218.

CARSTON, R. AND WEARING, C., 2011. Metaphor, hyperbole and simile: A pragmatic approach. *Language and Cognition*, 3(2), pp.283–312.

CASASOLA, E.E., 2004. Elaboración de material educativo para la formación de profesionales en desarrollo de software. In M. Solar, D. Fernández-Baca, & E. Cuadros-Vargas, eds. *Actas de la XXX Conferencia Latinoamericana de Informática (CLEI2004)*. Arequipa, pp. 1148–1157.

CASPERSEN, M.E. AND BENNEDSEN, J., 2007. Instructional design of a programming course: a learning theoretic approach. In *Proceedings of the third international workshop on Computing education research - ICER '07*. New York, New York, USA: ACM Press, p. 111.

CATES, W.M., 2002. Systematic selection and implementation of graphical user interface metaphors. *Computers & Education*, 38(4), pp.385–397.

CECCHINI, M., 2011. Introducción. In *Psicología y pedagogía*. Sevilla: Ediciones Akal, pp. 7–20.

CHUNG, G.K.W.K. AND KERR, D.S., 2012. *A Primer on Data Logging to Support Extraction of Meaningful Information from Educational Games: An Example from Save Patch. CRESST Report 814.*, National Center for Research on Evaluation, Standards, and Student Testing (CRESST). 300 Charles E Young Drive N, GSE&IS Building 3rd Floor, Mailbox 951522, Los Angeles, CA 90095-1522. Tel: 310-206-1532; Fax: 310-825-3883; Web site: http://www.cresst.org.

COLBURN, T.R. AND SHUTE, G.M., 2008. Metaphor in computer science. *Journal of Applied Logic*, 6(4), pp.526–533.

COTO CHOTTO, M. AND DIRCKINCK-HOLMFELD, L., 2007. Diseño Para Un Aprendizaje Significativo. *Teoría de la Educación. Educación y Cultura en la Sociedad de la Información*, 8(3), pp.135–148.

COTO CHOTTO, M. AND MORA RIVERA, S., 2012. El aula virtual como modelo de democratización del conocimiento. *Uniciencia*, 26(1–2), pp.169–178.

COTO CHOTTO, M., MORA RIVERA, S. AND ALFARO SALAZAR, G., 2013. Giving more autonomy to computer engineering students: Are we ready? In *IEEE Global Engineering Education Conference, EDUCON*. Berlin, Germany, pp. 618–626.

COTO CHOTTO, M., MORA RIVERA, S. AND LYKKE, M., 2012. Design considerations for introducing PBL in computer engineering. In *38th Latin America Conference on Informatics, CLEI 2012*. Medellin, Colombia.

CRAIG, M. AND PETERSEN, A., 2016. Student difficulties with pointer concepts in C. In *Proceedings of the Australasian Computer Science Week Multiconference on - ACSW '16*. New York, New York, USA: ACM Press, pp. 1–10.

CRISP, P., 2001. Allegory: conceptual metaphor in history. *Language and Literature*, 10, pp.5–19.

CRISP, P., 2008. Between extended metaphor and allegory: is blending enough? *Language and Literature*, 17(4), pp.291–308.

DADIC, T., STANKOV, S. AND ROSIC, M., 2008. Meaningful learning in the tutoring system for programming. In *ITI 2008 - 30th International Conference on Information Technology*

*Interfaces*. IEEE, pp. 483–488.

DENNY, P., LUXTON-REILLY, A. AND CARPENTER, D., 2014. Enhancing Syntax Error Messages Appears Ineffectual. In *Proceedings of the 19th ACM Conference on Innovation & Technology in Computer Science Education (ITiCSE'14)*. Uppsala, Sweden, pp. 273–278.

DØRUM, K. AND GARLAND, K., 2011. Efficient electronic navigation: A metaphorical question? *Interacting with Computers*, 23(2), pp.129–136.

DUMAS, J.S. AND FOX, J.E., 2012. Usability Testing. In J. A. Jacko, ed. *The human-computer interaction handbook: fundamentals, evolving technologies, and emerging applications*. CRC Press, pp. 1221–1241.

DUMAS, J.S. AND REDISH, J., 1999. *A Practical Guide to Usability Testing*, Oregon, USA: Intellect Books.

EBEL, G. AND BEN-ARI, M., 2006. Affective effects of program visualization. In *Proceedings of the 2006 international workshop on Computing education research - ICER '06*. New York, New York, USA: ACM Press, p. 1.

EBERTS, R.E. AND BITTIANDA, K.P., 1993. Preferred mental models for direct manipulation and command-based interfaces. *International Journal of Man-Machine Studies*, 38(5), pp.769–785.

EGAN, M.H. AND MCDONALD, C., 2014. Program visualization and explanation for novice C programmers. In *Proceedings of the Sixteenth Australasian Computing Education Conference - Volume 148*. Australian Computer Society, Inc., pp. 51–57.

EGAN, M.H. AND MCDONALD, C., 2013. Runtime error checking for novice C programmers. In *4th Annual International Conference on Computer Science Education: Innovation and Technology (CSEIT 2013)*. pp. 1–9.

FINCHER, S. AND PETRE, M. EDS., 2004. *Computer Science Education Research*, Netherlands: CRC Press.

FLESHNER, E.A., 2011. Psicología del aprendizaje y de la aplicación de algunos conceptos de física. In *Psicología y pedagogía*. Ediciones Akal, pp. 213–231.

FORCEVILLE, C.J. AND URIOS-APARISI, E., 2009. *Multimodal metaphor*, Berlin: M. de Gruyter.

FRANCIS, S., 2012. *El conocimiento pedagógico del contenido como modelo de mediación docente*, Coordinación Educativa y Cultural.

GHASSEMZADEH, H., 2005. Vygotsky's mediational psychology: A new conceptualization of culture, signification and metaphor. *Language Sciences*, 27(3), pp.281–300.

GRAČANIN, D., MATKOVIĆ, K. AND ELTOWEISSY, M., 2005. Software visualization. *Innovations in Systems and Software Engineering*, 1(2), pp.221–230.

GREENING, T., 2000. Emerging Constructivist Forces in Computer Science Education: Shaping a New Future? In T. Greening, ed. *Computer Science Education in the 21st Century*. New York, NY: Springer New York, pp. 47–80.

HARMON-JONES, E. AND MILLS, J., 1999. An introduction to cognitive dissonance theory and an overview of current perspectives on the theory. In E. Harmon-Jones & J. Mills, eds. *Cognitive dissonance: Progress on a pivotal theory in social psychology*. Washington, DC: American Psychological Association, p. 411.

HEATH, D., NORTON, D. AND VENTURA, D., 2014. Conveying Semantics through Visual Metaphor. *ACM Transactions on Intelligent Systems and Technology*, 5(2), pp.1–17.

HERNÁNDEZ SAMPIERI, R., COLLADO FERNÁNDEZ, C. AND BAPTISTA LUCIO, M. DEL P., 2010. *Metodología de la investigación* 5th ed., McGraw-Hill.

HERTZ, M. AND JUMP, M., 2013. Trace-based teaching in early programming courses. In *Proceeding of the 44th ACM technical symposium on Computer science education - SIGCSE '13*. New York, New York, USA: ACM Press, p. 561.

HERTZUM, M. AND JACOBSEN, N.E., 2001. The Evaluator Effect: A Chilling Fact About Usability Evaluation Methods. *International Journal of Human-Computer Interaction*, 13(4), pp.421–443.

HEVNER, A.R., MARCH, S.T., PARK, J. AND RAM, S., 2004. Design science in information systems research. *MIS Quarterly*, 28(1), pp.75–105.

HIDALGO-CÉSPEDES, J., MARÍN-RAVENTÓS, G. AND LARA-VILLAGRÁN, V., 2016a. Learning principles in program visualizations: A systematic literature review. In *2016 IEEE Frontiers in Education Conference (FIE)*. IEEE, pp. 1–9.

HIDALGO-CÉSPEDES, J., MARÍN-RAVENTÓS, G. AND LARA-VILLAGRÁN, V., 2016b. Understanding Notional Machines through Traditional Teaching with Conceptual Contraposition and Program Memory Tracing. *CLEI Electronic Journal*, 19(2).

HSU, Y., 2006. The effects of metaphors on novice and expert learners' performance and mental-model development. *Interacting with Computers*, 18(4), pp.770–792.

HSU, Y., 2000. *The effects of varying levels of interface cues derived from metaphors on computer users' information search performance*. Indiana University.

HSU, Y., 2007. The Effects of Visual Versus Verbal Metaphors on Novice and Expert Learners' Performance. In J. A. Jacko, ed. *HCI 2007: Human-Computer Interaction. HCI Applications and Services*. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 264–269.

HSU, Y., 2005. The long-term effects of integral versus composite metaphors on experts' and novices' search behaviors. *Interacting with Computers*, 17(4), pp.367–394.

HSU, Y. AND SCHWEN, T.M., 2003. The effects of structural cues from multiple metaphors on computer users' information search performance. *International Journal of Human Computer Studies*, 58(1), pp.39–55.

HUNDHAUSEN, C.D., DOUGLAS, S.A. AND STASKO, J.T., 2002. A Meta-Study of Algorithm Visualization Effectiveness. *Journal of Visual Languages & Computing*, 13(3), pp.259–290.

HUNG, D.W.L., 2002. Metaphorical ideas as mediating artifacts for the social construction of knowledge. *International Journal of Instructional Media*, 29(2), pp.197–214.

ISOHANNI, E., 2013. *Visualizations in Learning Programming Building a Theory of Student Engagement*. Tampere University of Technology.

ISOHANNI, E. AND JÄRVINEN, H.-M., 2014. Are visualization tools used in programming education?: by whom, how, why, and why not? In *Proceedings of the 14th Koli Calling International Conference on Computing Education Research - Koli Calling '14*. New York, New York, USA: ACM Press, pp. 35–40.

KAPP, K.M., 2012. *The Gamification of Learning and Instruction: Game-based Methods and*

*Strategies for Training and Education* 1st ed., Pfeiffer.

KELLEHER, C. AND PAUSCH, R., 2005. Lowering the Barriers to Programming: A Taxonomy of Programming Environments and Languages for Novice Programmers. *ACM Computing Surveys*, 37(2), pp.83–137.

KERNIGHAN, B.W. AND RITCHIE, D.M., 1988. *The C Programming Language* 2nd ed., United States: Prentice Hall.

KOSTIUK, G.S., 2011. Algunos aspectos de la relación recíproca entre educación y desarrollo de la personalidad. In *Psicología y pedagogía*. Sevilla: Ediciones Akal, pp. 41–58.

KUITTINEN, M. AND SAJANIEMI, J., 2004. Teaching roles of variables in elementary programming courses. *ACM SIGCSE Bulletin*, 36(3), p.57.

KUNIAVSKY, M., 2010. *Smart Things: Ubiquitous Computing User Experience Design* 1st ed., China: Morgan Kaufmann.

LAHTINEN, E., AHONIEMI, T. AND SALO, A., 2007. Effectiveness of Integrating Program Visualizations to a Programming Course. In *Koli Calling '07 Proceedings of the Seventh Baltic Sea Conference on Computing Education Research - Volume 88*. pp. 195–198.

LAHTINEN, E., ALA-MUTKA, K. AND JÄRVINEN, H.-M., 2005. A study of the difficulties of novice programmers. In *Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education - ITiCSE '05*. New York, New York, USA: ACM Press, p. 14.

LAKOFF, G., 1993. The Contemporary Theory of Metaphor. In A. Ortony, ed. *Metaphor and Thought*. Cambridge University Press, pp. 202–251.

LAKOFF, G. AND JOHNSON, M., 2003. *Metaphors we live by* 1st ed., Chicago: University of Chicago Press.

LANZA, M., 2003. CodeCrawler-lessons learned in building a software visualization tool. In *Seventh European Conference onSoftware Maintenance and Reengineering, 2003. Proceedings*. IEEE Comput. Soc, pp. 409–418.

LEE, J., 2007. The effects of visual metaphor and cognitive style for mental modeling in a hypermedia-based environment. *Interacting with Computers*, 19(5–6), pp.614–629.

LIN, L.-Y., 1989. *Learning to use hypertext systems with metaphors: An interface design perspective*. University of Illinois at Urbana-Champaign.

LURIA, LEONTIEV AND VIGOTSKY, 2011. *Psicología y pedagogía* 4th ed., Sevilla, España: Ediciones Akal.

LYKKE, M., COTO CHOTTO, M., JANTZEN, C., MORA RIVERA, S. AND VANDEL, N., 2015. Motivating Students Through Positive Learning Experiences : A Comparison of Three Learning Designs for Computer Programming Courses. *Journal of Problem Based Learning in Higher Education*, 3(2), pp.80–108.

LYKKE, M., COTO CHOTTO, M., MORA RIVERA, S., VANDEL, N. AND JANTZEN, C., 2014. Motivating programming students by problem based learning and LEGO robots. In *2014 IEEE Global Engineering Education Conference (EDUCON)*. Istanbul, Turkey: IEEE, pp. 544–555.

MALETIC, J.I., MARCUS, A. AND COLLARD, M.L., 2002. A task oriented view of software visualization. In *Proceedings First International Workshop on Visualizing Software for Understanding*

*and Analysis*. IEEE Comput. Soc, pp. 32–40.

DI MARE, A., 2010a. Aprendizaje Java acelerado por casos de prueba JUnit. In *XVIII Congreso Iberoamericano de Educación Superior en Computación (CIESC)*. Asunción, Paraguay.

DI MARE, A., 2013a. Enseñanza de C++ al Estudiante Java. In *XV Congreso Internacional de Informática en la Educación*. La Habana, Cuba.

DI MARE, A., 2010b. Introducción de la programación concurrente en el primer curso de programación. In *Octava Conferencia del Latin American And Caribbean Consortium Of Engineering Institutions LACCEI-2010*. Arequipa, Perú.

DI MARE, A., 1996. Tres formas diferentes de explicar la recursividad. *Revista Ingeniería, Facultad de Ingeniería, Universidad de Costa Rica*, 6(2), pp.31–44.

DI MARE, A., 2008. Uso de Doxygen para especificar módulos y programas. In *I Congreso Internacional de Computación y Matemática (CICMA)*. Universidad Nacional (UNA), Costa Rica, pp. 1–5.

DI MARE, A., 2013b. Uso de la Visualización Jeliot para Apoyar el Aprendizaje Acelerado de la Programación. In *Congreso Internacional de Informática en la Educación (InforEdu-2013)*. La Habana.

MAYER, R.E., 1981. The Psychology of How Novices Learn Computer Programming. *ACM Computing Surveys*, 13(1), pp.121–141.

MCCAULEY, R. ET AL., 2008. Debugging: a review of the literature from an educational perspective. *Computer Science Education*, 18(2), pp.67–92.

MCCRACKEN, M. ET AL., 2001. A multi-national, multi-institutional study of assessment of programming skills of first-year CS students A framework for first-year learning objectives. *ACM SIGCSE Bulletin, Volume 33 Issue 4*, pp.128–180.

MILERYAN, E.A., 2011. Características psicológicas de la transferencia de capacidades técnicas en los estudiantes de escuelas superiores. In *Psicología y pedagogía*. Sevilla, España: Ediciones Akal, pp. 257–268.

MILNE, I. AND ROWE, G., 2002. Difficulties in Learning and Teaching Programming — Views of Students and Tutors. *Education and Information Technologies*, 7(1), pp.55–66.

MORA RIVERA, S., SOTO CHOTTO, M. AND ALFARO SALAZAR, G., 2014. A proposal for implementing PBL in programming courses. In *Proceedings of the 2014 Latin American Computing Conference, CLEI 2014*. Montevideo, Uruguay.

MORENO, A., JOY, M., MYLLER, N. AND SUTINEN, E., 2010. Layered architecture for automatic generation of conflictive animations in programming education. *IEEE Transactions on Learning Technologies*, 3(2), pp.139–151.

MORENO, A. AND MYLLER, N., 2003. Producing an Educationally Effective and Usable Tool for Learning, The Case of the Jeliot Family. In *Proceedings of International Conference on Networked e-learning for European Universities (EUROPACE'03)*. Granada: EUROPACE.

MORENO, A., SUTINEN, E. AND JOY, M., 2014. Defining and evaluating conflictive animations for programming education. In *Proceedings of the 45th ACM technical symposium on Computer science education - SIGCSE '14*. New York, New York, USA: ACM Press, pp. 629–634.

MORENO, A., SUTINEN, E. AND SEDANO, C.I., 2013. A game concept using conflictive animations for learning programming. In *2013 IEEE International Games Innovation Conference (IGIC)*. IEEE, pp. 175–178.

MURILLO RIVERA, M., 2011. Explorando el proceso de enseñanza y de aprendizaje en el área de la programación de computadoras. *Actualidades Investigativas en Educación*, 6(1).

MYERS, B.A., 1990. Taxonomies of visual programming and program visualization. *Journal of Visual Languages & Computing*, 1(1), pp.97–123.

NAPS, T.L. ET AL., 2003. Exploring the role of visualization and engagement in computer science education. *ACM SIGCSE Bulletin*, 35(2), p.131.

NEALE, D.C. AND CARROLL, J.M., 1997. The Role of Metaphors in User Interface Design. In M. G. Helander, T. K. Landauer, & P. V. Prabhu, eds. *Handbook of Human-Computer Interaction (Second Edition)*. Amsterdam, The Netherlands: Elsevier Ltd, pp. 441–462.

VON NEUMANN, J., 1945. *First Draft of a Report on the EDVAC*,

NIELSEN, J. AND MOLICH, R., 1990. Heuristic Evaluation of user interfaces. In *CHI '90 Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. Seattle, Washington, pp. 249–256.

OSGOOD, C.E., 1964. Semantic Differential Technique in the Comparative Study of Cultures1. *American Anthropologist*, 66(3), pp.171–200.

PADOVANI, S. AND LANSDALE, M., 2003. Balancing search and retrieval in hypertext: context-specific trade-offs in navigational tool use. *International Journal of Human-Computer Studies*, 58(1), pp.125–149.

PAWELCZAK, D. AND BAUMANN, A., 2014. Virtual-C - a programming environment for teaching C in undergraduate programming courses. In *2014 IEEE Global Engineering Education Conference (EDUCON)*. IEEE, pp. 1142–1148.

PEFFERS, K., TUUNANEN, T., ROTHENBERGER, M.A. AND CHATTERJEE, S., 2007. A Design Science Research Methodology for Information Systems Research. *Journal of Management Information Systems*, 24(3), pp.45–77.

PETIT, J., GIMÉNEZ, O. AND ROURA, S., 2012. Jutge.org: An Educational Programming Judge. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education - SIGCSE '12*. New York, New York, USA: ACM Press, p. 445.

POWELL, K.C. AND KALINA, C.J., 2009. Cognitive and Social Constructivism: Developing Tools for an Effective Classroom. *Education*, 130(2), pp.241–250.

PRICE, B.A., BAECKER, R.M. AND SMALL, I.S., 1993. A Principled Taxonomy of Software Visualization. *Journal of Visual Languages & Computing*, 4(3), pp.211–266.

QUEIRÓS, R. AND LEAL, J.P., 2013. BabeLO — An Extensible Converter of Programming Exercises Formats. *IEEE Transactions on Learning Technologies*, 6(1), pp.38–45.

RAMADHAN, H., 1992. An intelligent discovery programming system. In *Proceedings of the 1992 ACM/SIGAPP Symposium on Applied computing technological challenges of the 1990's - SAC '92*. New York, New York, USA: ACM Press, pp. 149–159.

RAMADHAN, H.A., 2001. Programming by discovery. *Journal of Computer Assisted Learning*, 16(1), pp.83–93.

ROBERTSON, G. ET AL., 2000. The Task Gallery: a 3D window manager. In *Proceedings of the SIGCHI conference on Human factors in computing systems - CHI '00*. New York, New York, USA: ACM Press, pp. 494–501.

RODRÍGUEZ-AROCHO, W., 2002. Herramientas culturales y transformaciones mentales: De los jeroglíficos a la internet. *Ciencias de la Conducta*, 17, pp.12–19.

ROMAN, G.-C. AND COX, K.C., 1992. Program visualization: the art of mapping programs to pictures. In *Proceedings of the 14th international conference on Software engineering - ICSE '92*. New York, New York, USA: ACM Press, pp. 412–420.

RUBIO-FERNÁNDEZ, P., CUMMINS, C. AND TIAN, Y., 2016. Are single and extended metaphors processed differently? A test of two Relevance-Theoretic accounts. *Journal of Pragmatics*, 94, pp.15–28.

SAJANIEMI, J., BYCKLING, P. AND GERDT, P., 2007. Animation Metaphors for Object-Oriented Concepts. *Electronic Notes in Theoretical Computer Science*, 178, pp.15–22.

SAJANIEMI, J. AND KUITTINEN, M., 2003. Program animation based on the roles of variables. In *Proceedings of the 2003 ACM symposium on Software visualization - SoftVis '03*. New York, New York, USA: ACM Press, p. 7.

SANFORD, J.P., TIETZ, A., FAROOQ, S., GUYER, S. AND SHAPIRO, R.B., 2014. Metaphors we teach by. In *Proceedings of the 45th ACM technical symposium on Computer science education - SIGCSE '14*. New York, New York, USA: ACM Press, pp. 585–590.

SCHULTE, C. AND BENNEDSEN, J., 2006. What do teachers teach in introductory programming? In *Proceedings of the 2006 international workshop on Computing education research - ICER '06*. New York, New York, USA: ACM Press, p. 17.

SEASE, R., 2008. Metaphor's Role in the Information Behavior of Humans Interacting with Computers. *Information Technology and Libraries*, 27(4), p.9.

SENSALIRE, M., OGAO, P. AND TELEA, A., 2009. Evaluation of Software Visualization Tools: Lessons Learned. In *2009 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT 2009)*. IEEE, pp. 19–26.

SMILOWITZ, E.D., 1995. *Metaphors in user interface design: An empirical investigation*. New Mexico State University.

SMITH, M.K., POLLIO, H.R. AND PITTS, M.K., 1981. Metaphor as intellectual history: conceptual categories underlying figurative usage in American English from 1675-1975. *Linguistics*, 19, pp.911–935.

SORVA, J., 2013. Notional machines and introductory programming education. *ACM Transactions on Computing Education*, 13(2), pp.1–31.

SORVA, J., 2010. Reflections on threshold concepts in computer programming and beyond. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research - Koli Calling '10*. New York, New York, USA: ACM Press, pp. 21–30.

SORVA, J., 2008. The same but different students' understandings of primitive and object variables. In *Proceedings of the 8th International Conference on Computing Education Research - Koli '08*. New York, New York, USA: ACM Press, p. 5.

SORVA, J., 2012. *Visual Program Simulation in Introductory Programming Education*. Aalto

University.

SORVA, J., KARAVIRTA, V. AND MALMI, L., 2013. A Review of Generic Program Visualization Systems for Introductory Programming Education. *ACM Transactions on Computing Education*, 13(4), pp.1–64.

SORVA, J. AND SIRKIÄ, T., 2010. UUhistle: a software tool for visual program simulation. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research - Koli Calling '10*. New York, New York, USA: ACM Press, pp. 49–54.

STASKO, J.T. AND PATTERSON, C., 1992. Understanding and characterizing software visualization systems. In *Proceedings IEEE Workshop on Visual Languages*. IEEE Comput. Soc. Press, pp. 3–10.

STEFFE, L.P. AND GALE, J.E. EDS., 1995. *Constructivism in Education*, Lawrence Erlbaum.

TAM, M., 2000. Constructivism, Instructional Design, and Technology: Implications for Transforming Distance Learning. *Journal of Educational Technology & Society*, 3(2), pp.50–60.

TEPLOV, R.M., 2011. Aspectos psicológicos de la educación artística. In *Psicología y pedagogía*. Sevilla, España: Ediciones Akal, pp. 289–314.

TULLIS, T. (THOMAS) AND ALBERT, B. (WILLIAM), 2013. *Measuring the User Experience: Collecting, Analyzing, and Presenting Usability Metrics* 2nd ed., United States of America: Elsevier Inc.

URQUIZA-FUENTES, J. AND VELÁZQUEZ-ITURBIDE, J.Á., 2009. A Survey of Successful Evaluations of Program Visualization and Algorithm Animation Systems. *ACM Transactions on Computing Education*, 9(2), pp.1–21.

VAISHNAVI, V.K. AND KUECHLER JR., W., 2015. *Design science research methods and patterns: Innovating Information and Communication Technology* 2nd ed., Florida, UAS: CRC Press.

VIHAVAINEN, A., AIRAKSINEN, J. AND WATSON, C., 2014. A systematic review of approaches for teaching introductory programming and their influence on success. In *Proceedings of the tenth annual conference on International computing education research - ICER '14*. New York, New York, USA: ACM Press, pp. 19–26.

VYGOTSKY, L.S., 2011. Aprendizaje y desarrollo intelectual en la edad escolar. In *Psicología y pedagogía*. Sevilla, España: Ediciones Akal, pp. 23–39.

WAGUESPACK, L.J., 1989. Visual metaphors for teaching programming concepts. In *ACM SIGCSE Bulletin*. ACM, pp. 141–145.

WATSON, C. AND LI, F.W.B., 2014. Failure rates in introductory programming revisited. In *Proceedings of the 2014 conference on Innovation & technology in computer science education - ITiCSE '14*. New York, New York, USA: ACM Press, pp. 39–44.

WATSON, C., LI, F.W.B. AND GODWIN, J.L., 2014. No Tests Required: Comparing Traditional and Dynamic Predictors of Programming Success. In *Proceedings of the 45th ACM technical symposium on Computer science education - SIGCSE '14*. New York, New York, USA: ACM Press, pp. 469–474.

WIERINGA, R.J., 2014. *Design Science Methodology for Information Systems and Software Engineering*, Berlin, Heidelberg: Springer.

WOHLIN, C., RUNESON, P., HÖST, M., OHLSSON, M., REGNELL, B. AND WESSLÉN, A., 2012. *Experimentation in Software Engineering*, Springer Berlin Heidelberg.

WU, W.-H., CHIOU, W.-B., KAO, H.-Y., ALEX HU, C.-H. AND HUANG, S.-H., 2012. Re-exploring game-assisted learning research: The perspective of learning theoretical bases. *Computers & Education*, 59(4), pp.1153–1161.

XINOGALOS, S., 2013. Using flowchart-based programming environments for simplifying programming and software engineering processes. In *2013 IEEE Global Engineering Education Conference (EDUCON)*. IEEE, pp. 1313–1322.

YAN, Y., HIROTO, N., KOHEI, H., SHOTA, S. AND HE, A., 2014. A C Programming Learning Support System and Its Subjective Assessment. In *2014 IEEE International Conference on Computer and Information Technology*. IEEE, pp. 561–566.

YANG, J., LEE, Y., HICKS, D. AND CHANG, K.H., 2015. Enhancing Object-Oriented Programming Education using Static and Dynamic Visualization. In *2015 IEEE Frontiers in Education Conference*.

# Índice de materias

alegoría, 54
  visual, 56, 64
  visual abstracta, 64
  visual concreta, 64

ambiente de ejecución, 126

amenidad de usabilidad, 200

animación por computadora, 27

aprendizaje, 32

arquitectura interna, 251

artefacto, 8

audiencia pretendida de una visualización, 29

caso de prueba, 151

ciclo
  de diseño, 11
  de evaluación, 12

ciencia del diseño, 8

constructivismo sociocultural, 31

constructo, 109

contraposición conceptual, 36

efecto del evaluador, 175

eficiencia de la tarea, 169

ejercicio de programación, 261

elemento
  del juego, 58
  lúdico, 58

enseñanza
  alegórica, 91
  metafórica, 91

entrada estándar, 129

error de usabilidad, 169

error estándar, 130

estado
  de edición, 268
  de generación, 268
  de preparación, 272

estudio observacional, 166

evaluación heurística, 166

expresión metafórica, 47

generador de casos de prueba
  de archivo, 271
  estándar, 270

hilo de ejecución, 133

inferior (GDB), 277

juego, 58

juez automático, 149

ludificación, 57

máquina nocional, 3, 22
  principio de simplicidad, 24
  principio de visibilidad, 24

mediador, 32

metáfora, 46
  clasificación por ámbito, 50
  clasificación por modalidad, 49
  compleja, 55
  compuesta, 55
  conceptual, 46, 47
  de imagen, 47
  expresión metafórica, 47
  extendida, 53
  individual, 52
  múltiple, 55
  principio de generalización, 48
  principio de invariancia, 48
  teoría de comparación y sustitución, 45
  teoría de interacción, 45
  visual, 56
  visual extendida, 56, 64
  visual extendida abstracta, 64
  visual extendida concreta, 64
  visual individual, 56

meta-modelo, 250

modelo, 109

motor
- de casos de prueba, 251
- de ludificación, 251

motor de interacción, 251

motor de visualización, 251

objetivo
- científico, 9
- ingenieril, 8

pila de invocaciones, 135

problema de usabilidad, 169

problema único de usabilidad, 173

prototipo
- C++, 190
- PowerPoint, 156

rastreo de memoria de programa, 4

registro de eventos, 257

requerimiento
- aplicación del concepto, 40
- asimilación, 38
- comparación de conceptos, 39
- contraposición conceptual, 37
- ejercicios conceptuales, 40
- enfoque en el proceso, 41
- estado activo, 34
- evaluación, 44
- formación de hábitos, 42
- interés, 34
- motivación, 33
- nociones previas, 36
- organización de conceptos, 44
- realimentación, 35
- sistemas de conceptos, 43
- transferencia, 45

salida estándar, 130

segmento
- de código, 127
- de memoria, 127
- de memoria dinámica, 137
- de pila, 135
- de texto, 127

simulación por computadora, 27

solución, 271

superior (GDB), 277

teoría de diseño, 11

usabilidad, 163

valor, 131

variable, 131

visualización, 25
- activa, 63
- audiencia, 29
- automática, 26
- de algoritmo, 28
- de código, 28
- de datos, 27
- de programa, 4, 28, 30
- de software, 27
- dinámica, 26
- disponible, 63
- especializada, 26, 29
- estática, 26
- genérica, 26, 29
- interactiva, 27
- lúdica de programa, 111
- lúdica-alegórica concreta de programa, 111
- manual, 26
- por computadora, 26