Tesis de Licenciatura

Visualizaciones Gráficas de Datos y su Aplicación en Debuggers

Rossana Escobar (LU 579/87)
Pablo Gelaf (LU 478/86)
Hernán Otero (LU 399/87)

Director:
Lic. Roberto Bevilacqua

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Mayo-2003

Resumen

Durante la depuración de programas que manejan estructuras de datos complejas, es necesario proveer una forma cómoda de inspeccionar dichas estructuras. Las formas de visualización tradicionales funcionan bien para estructuras pequeñas, pero poseen restricciones a la hora de desplegar estructuras con gran cantidad de datos. Las vistas de tipo *fisheye* (ojo de pez) permiten al usuario manipular este tipo de estructuras enfocando una porción de las mismas y al mismo tiempo preservando el contexto general. Este trabajo implementa un *debugger* de programas Java y explora el uso de diferentes técnicas *fisheye* para la visualización y la navegación de estructuras de datos. Así mismo, se propone un mecanismo para permitirle al usuario la configuración de la representación gráfica de sus propias estructuras de datos durante la depuración.

Abstract

When debugging programs that deal with complex data structures, it is sometimes necessary to have convenient ways to inspect such structures. Traditional visualization techniques work well for small structures, but are limited when it comes to represent structures with a large number of elements. Fisheye views allows for manipulating these structures focusing on parts of these while at the same time preserving the overall context. This work implements a Java *debugger* and explores the use of different fisheye view techniques for data structure visualization and navigation. At the same time, we propose a mechanism for allowing users to use custom graphic representations of their own data structures during debugging.

TABLA DE CONTENIDOS

AGRADECIMIENTOS

Queremos agradecer a Roberto, nuestro director, por habernos guiado durante el transcurso de este trabajo así como a nuestras familias y amigos por brindarnos el apoyo y aliento necesarios para completarlo.

Durante el primer capítulo hablaremos de la importancia de la depuración dentro del ciclo de vida del desarrollo del software. Para ello, presentaremos los modelos más conocidos del ciclo de vida del software y haremos hincapié en uno de los más populares (cascada). Veremos como la etapa de prueba constituye una de las más costosas, la cual frecuentemente se convierte en el cuello de botella de todo el ciclo. Por esa razón, nuestro trabajo intentará aliviar algunos de los problemas que presenta ésta etapa utilizando herramientas de visualización gráfica de estructuras de datos.

En el segundo capítulo describiremos las distintas técnicas existentes para la visualización gráfica de estructuras de datos. Presentaremos una taxonomía para clasificarlas y entraremos en detalles en aquellas que consideramos más significativas y aplicables al área de *debugging* (depuración).

En el tercer capítulo presentaremos JPDA, la arquitectura provista por las últimas versiones de la plataforma Java para facilitar el desarrollo de *debuggers*. Explicaremos como haremos uso de esta herramienta para la implementación de un *debugger*. También introducimos *Reflection* y *Java Beans*, dos herramientas de la plataforma Java que utilizaremos durante el siguiente capítulo.

Por último, en el cuarto capítulo presentaremos una implementación de un *debugger* del lenguaje de programación Java donde utilizaremos la arquitectura JPDA descripta en el capítulo tres y dos de las técnicas de visualización de datos descriptas en el capítulo dos: *ZUI* o *Zoomable User Interfaces* (Interfaces de Usuario Magnificables) y *Tree Maps* (Mapas de Árboles).

Estado del Arte

Las herramientas de depuración prometen "ver" que es lo que esta sucediendo en un programa. No hace falta decir que esta visualización es muy limitada. Los *debuggers* más comunes utilizados en los comienzos de los 80' eran *debuggers* de línea de comando donde uno podía ingresar el nombre de una variable y ver su valor. Por supuesto eran útiles pero todavía existía la necesidad de inmersión y manipulación directa de los datos. Desde la época de los *debuggers* de línea de comando, las típicas herramientas de *debugging* han evolucionado en cuanto a la funcionalidad de sus interfaces de usuarios siendo ahora gráficas y haciendo uso de sus características que incluyen el despliegue de los datos a través de ventanas más amigables y la forma de acceder a estos a través del *mouse*, pero desafortunadamente la representación propia de los datos no ha evolucionado en absoluto.

Actualmente existe un *debugger* de dominio publico (*open source*) hecho por GNU llamado DDD que permite visualizar gráficamente objetos y referencias entre los mismos.

Si bien DDD mejora la visualización de la representación de los datos, el espacio de la pantalla se torna insuficiente cuando se trata de desplegar múltiples variables al mismo tiempo.

Otra limitación de DDD es la falta de extensibilidad por parte del usuario. No provee una interfaz para agregar representaciones gráficas alternativas. Si bien el proyecto GVD (http://libre.act-europe.fr/gvd/) se propuso eliminar algunas de estas limitaciones proveyendo una arquitectura más modular, la misma no ha sido implementada hasta la fecha. [ZEL01]

Capítulo 1

DEPURACIÓN DE ERRORES DENTRO DEL CICLO DE VIDA DEL SOFTWARE

Modelos de Ciclo de Vida del Software

El Ciclo de Vida del Software es el proceso completo de desarrollo de sistemas de información a través de un ciclo de múltiples pasos que va desde la investigación de los requerimientos iniciales hasta el análisis, diseño, implementación y mantenimiento. [KAY02]

Los Modelos de Ciclo de Vida del Software son modelos de resolución de problemas aplicados al proceso del desarrollo de Software. Ellos proveen al Ingeniero de Software los medios para entender, razonar, planear, monitorear y controlar el desarrollo del software. [HOR95]

Existen diferentes Modelos de Ciclo de Vida del Software,

- Modelo de Cascada

  Es el modelo más antiguo. Se divide en una secuencia de etapas en las cuales la salida de cada etapa se vuelve la entrada de la siguiente. Estas etapas se pueden caracterizar y dividir de distintas formas, incluyendo la siguiente:

  - Planeamiento del Proyecto y análisis de Factibilidad: Establece una vista de alto nivel del proyecto propuesto y determina sus objetivos.

- Análisis del Sistema y Definición de Requerimientos: Refina los objetivos del sistema en operaciones así como las funciones de la aplicación deseada. Analiza necesidades de información del usuario final.

- Diseño del Sistema: Describe características deseadas y operaciones en detalle, incluyendo diseño de pantallas, reglas de negocios, diagramas de procesos, seudo código y otra documentación.

- Implementación: la codificación de programas ocurre en esta etapa.

- Prueba, Ajuste y Depuración: Junta todas las piezas en un ambiente de prueba especial y se verifica la ausencia de errores y problemas de interoperabilidad.

- Aceptación de Usuario, Instalación y Puesta en Producción: Etapa final del desarrollo inicial donde el software es puesto en producción a cumplir sus funciones.

- Mantenimiento: Lo que sucede durante el resto de la vida del software, cambios, correcciones, agregados, migraciones a diferentes plataformas, etc. Esta etapa, la menos vistosa y quizás la más importante de todas, continua mientras el sistema es utilizado.

El modelo de cascada asume que el único rol de los usuarios es el de especificar los requerimientos y que todos los requerimientos pueden ser especificados por adelantado. Desafortunadamente los requerimientos crecen y cambian a medida que progresa el proyecto,

requiriendo retroalimentación y consultas iterativas. Debido a esto se han desarrollado muchos otros modelos de Ciclo de Vida del Desarrollo del Software. [KAY02]

- Modelo de Prototipación Rápida

  En el modelo de Prototipación Rápida – a veces llamado Desarrollo Rápido de Aplicaciones – el énfasis inicial esta en crear un prototipo que luzca y actúe como el producto deseado, para poder verificar su utilidad. El prototipo es una parte esencial de la fase de definición de los requerimientos, y puede ser creado usando herramientas distintas a las usadas para crear el producto final. Una vez que el prototipo es aprobado, se codifica el producto real. [KAY02]

- Modelo Espiral

  El modelo Espiral enfatiza la necesidad de volver a las primeras etapas y repetirlas varias veces a medida que el proyecto progresa. Es realmente una serie de ciclos de Cascada cortos, cada uno produciendo un prototipo preliminar representando una parte del proyecto. Este acercamiento permite mostrar una "prueba de concepto" en una etapa temprana del ciclo, y refleja más acertadamente la desordenada y casi caótica evolución de la tecnología [KAY02]

- Modelo Construir y Arreglar

  El modelo Construir y Arreglar es el más crudo de los métodos. Propone codificar sin planeamiento y luego modificar el código a medida que sea necesario hasta satisfacer los requerimientos del usuario.

Este es, obviamente, un modelo sin un final bien definido, por lo cual puede ser muy riesgoso. [KAY-02]

- Modelo Incremental

  El modelo Incremental divide al producto en distintas secciones, donde cada sección del proyecto es creada y verificada separadamente. Este acercamiento probablemente encontrara errores en los requerimientos de usuarios más rápidamente, dado que se solicita retroalimentación de los usuarios en cada etapa, y debido a que el código es verificado rápidamente después de haber sido escrito. [KAY02]

- Modelo Fuente

  El modelo Fuente, reconoce que aunque algunas actividades no pueden comenzar antes que otras – tales como que se necesita un diseño antes de comenzar a codificar – hay una superposición considerable de actividades a través del desarrollo del ciclo. [HOR95]

- Modelo Cuarto Limpio (Cleanroom)

  El modelo Cuarto Limpio es un proceso orientado a trabajo en equipo que hace el desarrollo más manejable ya que se realiza bajo un control de calidad estadístico. La filosofía detrás de este método es tratar de evitar la dependencia del costoso proceso de eliminación de errores codificando incrementalmente y verificando su correctitud antes de pasar a la etapa de prueba. [LIN94]

- Modelo SSADM (Structured Systems Analysis and Design Methodology)

Este modelo se basa en el uso de tres técnicas claves: Modelización de Datos Lógicos, Modelización de Flujo de Datos y Modelización de Entidades/Eventos. El mismo se descompone en 5 etapas: estudio de factibilidad, análisis de requerimientos, especificación de requerimientos, especificación lógica del sistema y diseño físico. [HUT96]

- Modelo Métrica Versión 3

  Este modelo utiliza un enfoque orientado al proceso.

  Usa como referencia el Modelo de Ciclo de Vida de Desarrollo propuesto en la norma ISO 12.207 "Information Technology – Software Life Cycle Processes". Siguiendo este modelo se ha elaborado la estructura en la que se distinguen procesos principales (Planificación, Desarrollo y Mantenimiento) e interfaces (Gestión de Proyectos, Control de Calidad y Seguridad) cuyo objetivo es dar soporte al proyecto en los aspectos organizativos.

  El modelo descompone cada uno de los procesos en actividades, y éstas a su vez en tareas. Para cada tarea se describe su contenido haciendo referencia a sus principales acciones, productos, técnicas, prácticas y participantes. [MAP03]

Dado que en el alcance de este trabajo estamos interesados en el estudio de la tarea de depuración dentro del ciclo de vida del software, elegimos el modelo de Cascada como base para estudiar las etapas que involucran a esta tarea, ya que encontramos que este es el modelo que más claramente define a esas etapas.

Prueba, Ajuste y Depuración

La fase de Prueba, Ajuste y Depuración consiste de dos sub-tareas principales, la *prueba de la funcionalidad* y la *evaluación de la performance* alcanzada. La primera examina si el comportamiento de entrada/salida de un programa se corresponde a su especificación. La segunda se aplica cuando se investiga la eficiencia y performance en tiempo de ejecución de un programa funcionalmente correcto. A partir de ahora nos enfocaremos principalmente en el proceso de prueba [KRA00]

El proceso de Prueba

A continuación se detallan los pasos básicos de la prueba entre el momento en que el código ha sido escrito y el momento en que se determina que es aceptable:

- Análisis de código estático
- Preparación de casos de prueba
- Monitoreo de la prueba y verificación de salida
- Aislamiento de fallas y *debugging* (depuración)
- Re-prueba (una vez que se han reparado los errores)
- Integración de rutinas

El análisis de código estático comienza con los diagnósticos de compilación y chequeo de tipos de datos.

Para la preparación de los casos de prueba, existen herramientas que ayudan a los ingenieros a elegir los valores de entrada para hacer que el programa se ejecute a través de un camino deseado, estas son suficientes solamente para generar una cantidad de entradas limitada, por lo cual las salidas deben todavía ser calculadas manualmente o con otros medios de soporte de computación.

Para poder realizar la mayor parte de la prueba automáticamente, se han desarrollado técnicas para varios tipos de verificación de tipos de datos dinámicos y de aseveraciones. Además, herramientas de prueba posterior al procesamiento ayudan a los usuarios con comparadores de salidas y reportes de excepciones.

En el área de aislamiento de fallas y depuración podemos distinguir métodos bien establecidos como vuelcos de memoria, *traces*, *snapshots*, *breakpoints*, *replay*, *backtracking*.

Luego de que se reparan los errores, se utiliza el re-testeo para comparar ciclos de testeo previos con los resultados del código corregido, aquí se pueden usar comparadores para verificar diferencias en el código, y entre entradas y salidas de ambas ejecuciones.

Problemas en el proceso de Prueba

Uno de los problemas principales de la fase de prueba es que en el caso general, la prueba solo puede demostrar la existencia de errores, pero nunca puede probar la ausencia de los mismos. Más aun, es usualmente imposible realizar una prueba exhaustiva debido al enorme espacio de posibilidades de entradas que tienen la mayoría de los sistemas. Debido a esta capacidad limitada de prueba, ciertos errores en el código pueden no ser detectados en los escenarios de prueba. Esto implica que la eliminación total de errores es un objetivo irrealista.

Otro problema principal es que muy a menudo, la prueba no es considerada hasta que el código ha sido ejecutado por primera vez y recién allí se descubre que este no funciona correctamente. Las estadísticas muestran que muchos errores se generan aun antes de comenzar a codificar, y cuanto más tarde se detecte el error, más alto será el costo de re-hacer el sistema para que funcione correctamente.

La tarea de prueba es además altamente tediosa y en la mayoría de los casos es realizada manualmente, lo que la lleva a ser en si misma propensa a errores.

Finalmente, es muy difícil determinar la cantidad suficiente de testeo. Es claro que una prueba exhaustiva es imposible. En principio existen dos formas para decidir acerca de la completitud de la prueba: estadística, cuando las estadísticas de errores muestran un grado suficientemente alto de confiabilidad, y sistemática, la cual intenta determinar métodos para identificar las propiedades funcionales y clases de posibles errores en los programas.

Depuración de Errores

Probablemente la parte más importante de la prueba es la depuración o localización de errores y su corrección. Es una parte fundamental del ciclo de vida del software dado que un programa es obviamente útil solamente si ejecuta correctamente y es lo suficientemente confiable. En términos de ingeniería de software, esta actividad podría definirse como:

El proceso de localización, análisis y corrección de errores.

Una gran parte del tiempo de desarrollo de los programadores es usado en depuración por lo cual se lo considera uno de los principales cuellos de botella dentro del ciclo de vida del software.

La fase de depuración comienza luego de que se han observado resultados incorrectos o fallas de ejecución durante la prueba. En tal caso, se deben detectar y corregir los errores y las fuentes de los mismos.

Mientras que la importancia del testeo y depuración es altamente aceptada en el dominio de la ingeniería del software, existe una sorprendente falta de literatura acerca de la depuración del software a un nivel práctico. Desde el punto de vista científico, la depuración esta muy poco estudiada, comparada, por ejemplo, a los compiladores. Además la depuración esta ausente en los planes actuales de ciencias de la computación, lo que deja a estudiantes y programadores sin demasiada guía. La comunidad entera de las ciencias de la computación ha ignorado durante mucho tiempo el problema de la depuración. Los ambientes de programación que se encuentran actualmente en el mercado proveen herramientas de depuración que son ligeramente mejores a las herramientas que acompañaban a los ambientes de programación de hace treinta años. Esto resulta del hecho de que los programas son objetos complejos, y consecuentemente su depuración también lo es. Por esta razón, la depuración es todavía una cuestión de "prueba y error".

Se pueden identificar los siguientes pasos generales durante la depuración de programas:

- Investigaciones preliminares: ¿es realmente un error?

- Depuración estática: verificación de los requerimientos, el diseño y análisis del código fuente.

- Observaciones de ejecución: testeo y análisis de los resultados de la ejecución.

- Manipulación del código fuente: instrumentación y ejecución parcial.

- Depuración dinámica: uso de breakpoints, ejecución de a pasos, monitoreo y testeo.

Implicancias de los Errores en el Software

Obviamente, los errores en el software son inevitables y en general, es muy probable que sucedan. La importancia e implicaciones de los errores son usualmente subestimadas, y generalmente sus consecuencias financieras no son tenidas en cuenta.

Resultados de un estudio estadístico realizado en 1995 sobre 175.000 desarrollos de software revelaron que el 31% de esos proyectos fueron cancelados antes de su finalización con una perdida financiera de 81.000 millones de dólares. Del resto de los proyectos, más del 52% sufrió en promedio un costo del 189% más de lo presupuestado, resultando en una pérdida de 59.000 millones de dólares.

Existe además la posibilidad de otras consecuencias que podrían provocar daños a las personas. Evidentemente los más críticos son los errores de *software* letales donde se puede llegar a herir o eventualmente causar la muerte a seres humanos (un ejemplo de estos fue el mal funcionamiento de las máquinas de aceleración lineal Therac-25 entre 1985 y 1987 en un Centro de Cáncer de Tyler, Texas donde al menos 4 personas recibieron dosis letales de radiación durante tratamiento médico).

Otro tipo de error de *software* que siempre obtiene la atención de la prensa son los *accidentes aéreos*, a pesar de que estos son a menudo declarados como los medios de transporte público más seguros. La mayoría de los problemas generalmente no suceden debido a un solo origen, sino que se deben a la

cooperación de varios factores, un ejemplo de este tipo de accidente fue cuando un Boeing 757 de American Airlines choco contra una montaña en Colombia, en ese caso, la computadora del avión estaba operando con una base de datos de navegación corrupta la cual dio información posicional errónea a la tripulación resultando en una catástrofe con 160 víctimas.

Otra área importante donde se encuentran errores a menudo es en las aplicaciones comerciales y gubernamentales. A pesar de que estos afectan a mucha gente, afortunadamente sus consecuencias no suelen ser letales. Un ejemplo se dio en 1985, cuando un sistema de impuestos envió avisos de impuestos impagos a 27.000 compañías, las cuales ya habían pagado sus impuestos. O en 1987 cuando un banco Ingles transfirió 2 millones de libras extra a algunos de sus clientes por error. Otro caso de grandes dimensiones se dio en 1990, cuando uno de los sistemas telefónicos de AT&T en la ciudad de Nueva York tuvo un problema menor, el cual fue el comienzo de una cadena de fallas resultando en el más severo colapso del sistema telefónico de los Estados Unidos. [KRA00]

Evolución del Debugging

Primera Generación

Al comienzo de la edad de la computación los programadores tuvieron la difícil tarea de persuadir a las computadoras para obtener alguna salida de los programas que ejecutaban. Estos se vieron forzados a inventar diferentes formas de obtener información de los programas que usaban, no solamente tenían que arreglar errores sino que también debían construir la herramienta a usar para encontrarlos. Las primeras técnicas de depuración utilizadas

fueron dispositivos tales como instrumentos de visualización (*scopes)* y bombitas de luz (*bulbs*) controladas por programas.

Segunda Generación

Eventualmente los programadores comenzaron a detectar errores en el código poniendo instrucciones de impresión dentro de los programas. De esta forma podían rastrear el camino recorrido por el programa y el valor de variables clave. El uso de sentencias de impresión libero a los programadores de la tediosa tarea de construir sus propias herramientas de depuración que consumía enormes cantidades de tiempo. Esta tarea es aun hoy comúnmente usada y es realmente buena para cierta clase de problemas.

Tercera Generación

Aunque las sentencias de impresión fueron una mejora en las técnicas de *debugging*, ellas todavía requieren de una cantidad considerable de tiempo y esfuerzo por parte del programador. Lo que los programadores necesitaban era una herramienta que pudiera ejecutar instrucciones de programas paso a paso e imprimir valores de cualquier variable del programa. Esto liberaría al programador de tener que decidir de antemano donde poner las sentencias de impresión, dado que esto sucedería a medida que se realiza la ejecución paso a paso. De esta manera nacieron los *debuggers* en tiempo de ejecución. En principio, un *debugger* en tiempo de ejecución no es más que una impresión automática de sentencias. Permite al programador seguir las variables y el camino del programa sin la necesidad de agregar las sentencias de impresión en el código.

Actualmente virtualmente todos los compiladores en el mercado incluyen un *debugger* en tiempo de ejecución. El *debugger* es implementado como una opción pasada al compilador del programa. Generalmente esta opción es

"-g". Esta opción le dice al compilador que construya suficiente información dentro del programa ejecutable capacitándolo así para correr con el *debugger* en tiempo de ejecución. [KOL02]

Estado Actual de la Depuración

Los *debuggers* en tiempo de ejecución fueron una gran mejora sobre las instrucciones de impresión ya que le permitieron al programador compilar y ejecutar con una sola compilación en vez de tener que modificar el código fuente y re-compilar, disminuyendo así las posibilidades de introducir nuevos errores. Estos facilitaron la detección de errores en los programas pero todavía existen clases de errores que son difíciles de detectar con los *debuggers* que se encuentran actualmente en el mercado.

La tecnología de *debugging* no evolucionó demasiado durante las últimas décadas con respecto a la visualización de datos. En la actualidad, uno de los problemas principales en la tarea de *debugging* es la falta de ambientes de depuración con interfaces de usuario lo suficientemente flexibles que brinden características de visualización de estructuras de datos complejas.

Nuestro trabajo apunta a mejorar las características de visualización de los datos para así brindar una más rápida y eficiente detección de errores.

Capítulo 2

VISUALIZACIONES GRÁFICAS DE DATOS

Herramientas de Visualización de Datos

La visualización de datos es una de las tareas que se realizan durante la depuración de programas. Si el programa depurado utiliza estructura de datos complejas, dichas estructuras deben ser representadas de forma tal que su visualización y navegación sea lo más cómoda e intuitiva posible desde el punto de vista del usuario.

Las vistas de tipo *fisheye* son adecuadas para el despliegue de estructuras con gran cantidad de datos. En esta sección se introduce el concepto de vistas de tipo *fisheye* y se describe su taxonomía, luego se presenta una serie de técnicas existentes y como estas se relacionan con la taxonomía descrita. Por último, se describen en más detalle aquellas técnicas que son consideradas relevantes en el contexto de este trabajo.

Introducción

Uno de los aspectos que ha recibido mayor atención en el trazado de grafos es el problema de escalabilidad: a medida que el grafo crece en tamaño, dibujar el grafo se convierte en una tarea que requiere mayor procesamiento (debido a la complejidad computacional del algoritmo utilizado para el trazado), y a su vez es más difícil de entender (debido a las limitaciones de las tecnologías utilizadas para el trazado como así también de las habilidades humanas para su reconocimiento). En la práctica, el dibujo de grafos se convierte en una tarea

compleja en grafos que contienen a partir de doscientos a trescientos nodos o vínculos.

Se han propuesto distintas soluciones a este problema. Una de ellas es mostrarle al usuario dos vistas simultáneas del grafo: una ventana conteniendo solo una parte del grafo en formato aumentado y otra ventana con el contenido del grafo completo en formato reducido. La vista reducida contiene una versión simplificada de todo el grafo, además de un rectángulo que representa el contenido actualmente desplegado en la vista aumentada. La vista aumentada provee una visión completa y detallada del área representada por el rectángulo contenido por la vista reducida.

Desafortunadamente, esta técnica sacrifica el contexto global del usuario con el objetivo de brindar acceso a la información local, complicando la percepción del usuario quien debe integrar mentalmente y de manera continua la vista detallada (aumentada) con la vista general (reducida).

Fisheye Views es una técnica de visualización que integra vistas con información detallada con el contexto global. [NOI96]

Fisheye Views

Definición

El término *Fisheye View* (FEV) es utilizado para agrupar una serie de técnicas de visualización basadas en la analogía del ojo de pez o los lentes gran angulares utilizados en fotografía. El objetivo principal de las técnicas FEV es proveer integración intuitiva de vistas con información detallada dentro del contexto global. [NOI96]

La Figura 4 muestra un ejemplo de FEV:

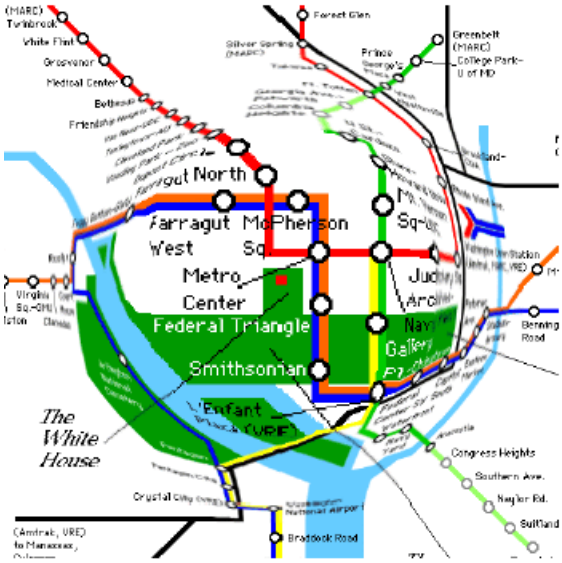


Figura 4. [STO93]

Una estrategia básica para dibujar estructuras con gran cantidad de datos utiliza la función *grado de interés* (GDI), esta función asigna un número o prioridad a cada nodo de la estructura, dicho número cuantifica el interés del usuario en dicho nodo teniendo en cuenta la tarea realizada por el usuario en un momento dado. Las prioridades pueden ser asignadas por el usuario o pueden ser calculadas por un algoritmo.

La función GDI está compuesta por dos componentes: la *importancia a priori* (IAP) que representa la importancia global de cada nodo dentro de la estructura, y la distancia (DIST) que representa la distancia conceptual entre dos nodos. Si un nodo es seleccionado como el foco de interés o *nodo focal* (NF), el GDI del usuario para un nodo 'n' dado un NF 'f' se calcula como GDI(n,f) = IAP(n) – DIST(n, f), basado en esta fórmula, el GDI crece a medida que IAP crece y se reduce a medida que DIST aumenta. [NOI96]

Ejemplos

La Figura 5 representa un grafo con 134 nodos y 338 vínculos. Los nodos representan las ciudades más importantes de los Estados Unidos y los vínculos representan los caminos entre dos ciudades vecinas. El valor IAP asignado a cada nodo es proporcional a la población de la ciudad representada por el nodo. [SAR92A]

Figura 5

La Figura 6 es un FEV del grafo mostrado en la Figura 5. El NF esta en la ciudad de St. Louis. [SAR92A]

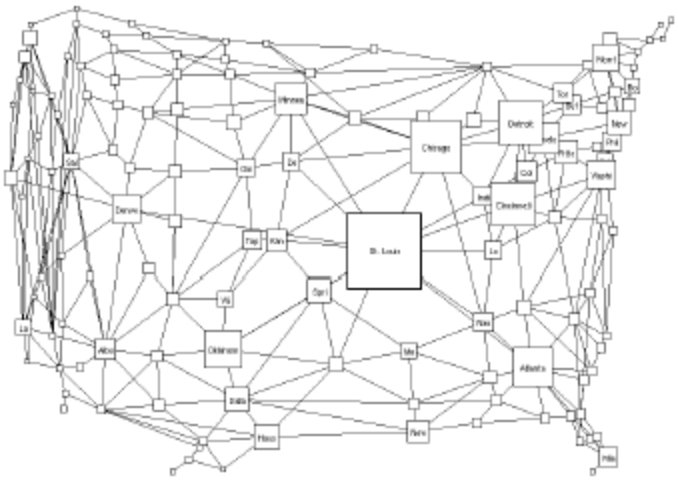


Figura 6

La siguiente figura es un FEV de la primer Figura 5 pero esta vez con el NF en la ciudad de Salt Lake City. [SAR92A]

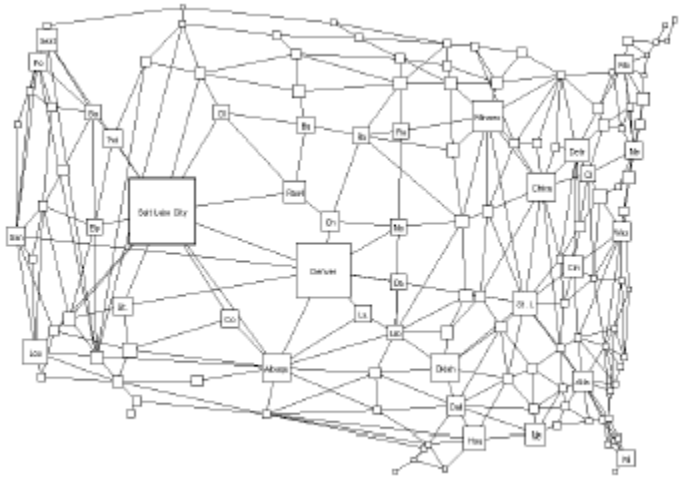


Figura 7

Taxonomía de Vistas de Tipo Grafo

Las vistas pueden ser categorizadas en los subgrupos de *vistas normales* y *vistas enfatizadas*.

Una *vista normal* es una vista en donde todos los elementos tienen la misma prioridad o, lo que es lo mismo, es una vista sin NFs.

Una FEV *implícita* surge como efecto de aplicar una perspectiva de tres dimensiones a la *vista normal*, en donde nodos que están más cerca del foco son desplegados en tamaño grande y nodos que están más lejos aparecen en tamaño pequeño. Las FEV implícitas son generalmente *estáticas*, pero pueden ser *dinámicas* si se altera el área de trazado (Ej.: moviendo los nodos con mayor prioridad al frente).

Las *vistas enfatizadas* (VE) se pueden clasificar de la siguiente manera:

- Una *vista filtrada* despliega solo un subconjunto de elementos suprimiendo el resto, esto puede ser realizado aumentando (*zooming*) o filtrando (*filtering*) el grafo de acuerdo a las prioridades de los elementos (desplegando elementos cuya prioridad es mayor a determinados valores y suprimiendo el resto).

- Una *vista distorsionada* resalta los elementos asignando diferentes tamaños, formas y posiciones. La distorsión puede ser *polar* u *ortogonal* (donde la DIST es geométrica), puede ser *no geométrica* (donde DIST no es geométrica) o puede ser *escalada local* o *globalmente* (en un solo nivel o en varios niveles).

- Una *vista adornada* resalta los elementos cambiando otras características de la visualización como el color, la sombra, el estilo del dibujo, etc.

A continuación usaremos el gráfico de la taxonomía para ejemplificar los distintos tipos de vistas.

La siguiente figura muestra una vista normal de esta taxonomía:

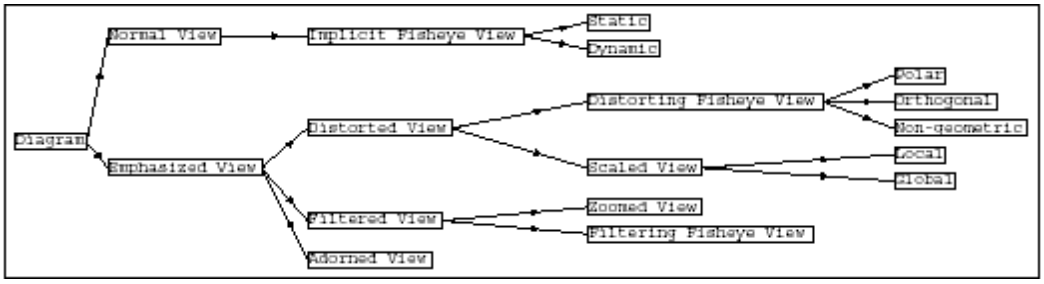


Figura 8

La próxima figura muestra una FEV filtrada con NF = Distorting Fisheye Views:

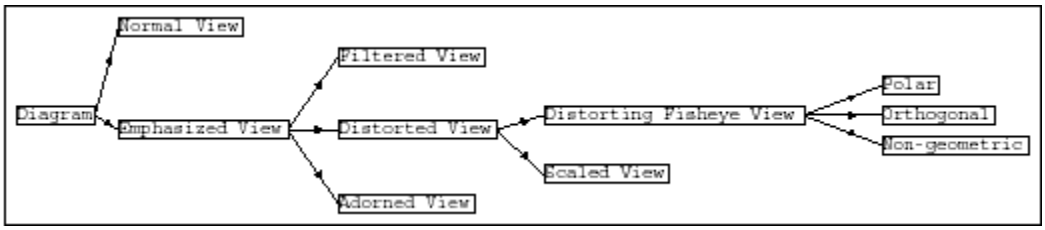


Figura 9

La próxima figura muestra una FEV distorsionada (ortogonalmente) con NF = Distorting Fisheye View:

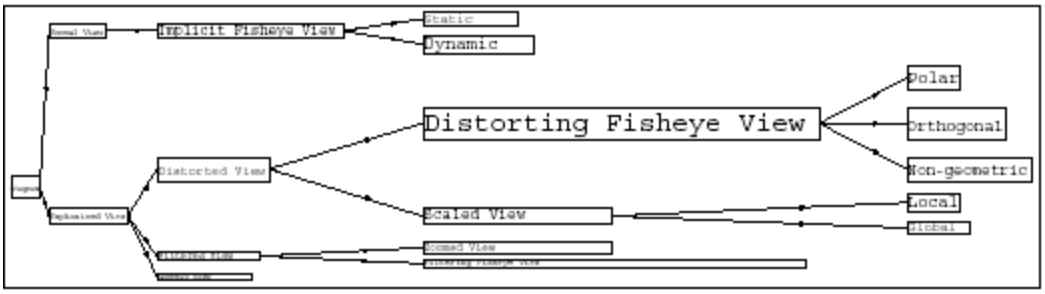


Figura 10

La próxima es una FEV distorsionada (en forma no geométrica), a la izquierda NF = Distorting Fisheye View a la derecha NFs = Dynamic y Global:

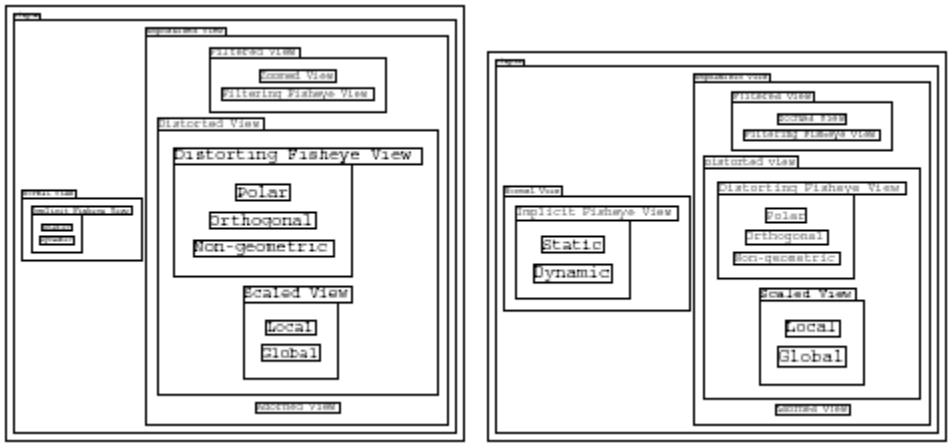


Figura 11

En la siguiente sección describiremos ejemplos de técnicas existentes para el trazado de Ves. Muchos de estos algoritmos generan vistas híbridas utilizando propiedades de dos o más de las vistas básicas presentadas.

Para abreviar, nos referiremos con la sigla AE al algoritmo utilizado para calcular el énfasis en vistas enfatizadas, y AP al algoritmo utilizado para el cálculo de la prioridad. [NOI96]

Ejemplo de Técnicas Existentes

Fisheye Views Generalizados

Es una técnica que reduce la complejidad de visualización de alto nivel de detalle desplegando un subconjunto de los datos más relevantes, generando FEVs de tipo filtradas. Como se menciono anteriormente, en una FEV generalizada, GDI(n,f) = IAP(n) – DIST(n, f). [NOI96]

The Perspective Wall

Este tipo de AE, que es utilizada para la visualización de información lineal como por ejemplo información relacionada con el tiempo, genera vistas globalmente escaladas. Esta técnica posee un panel central para el despliegue de detalles y dos paneles en perspectiva para el despliegue de información de contexto. La vista en perspectiva genera un efecto tipo FEV: enfatiza los detalles de los datos cerca del punto explorado agrandando dicha área respecto a otras que están más alejadas dentro de la vista contextual. El balance entre el detalle y la información contextual es controlando modificando el grado de plegamiento y el ancho del panel de detalles. El NF es modificado moviendo la pared (*scrolling*) como si fuera una partitura de piano. [NOI96]

Ejemplo:

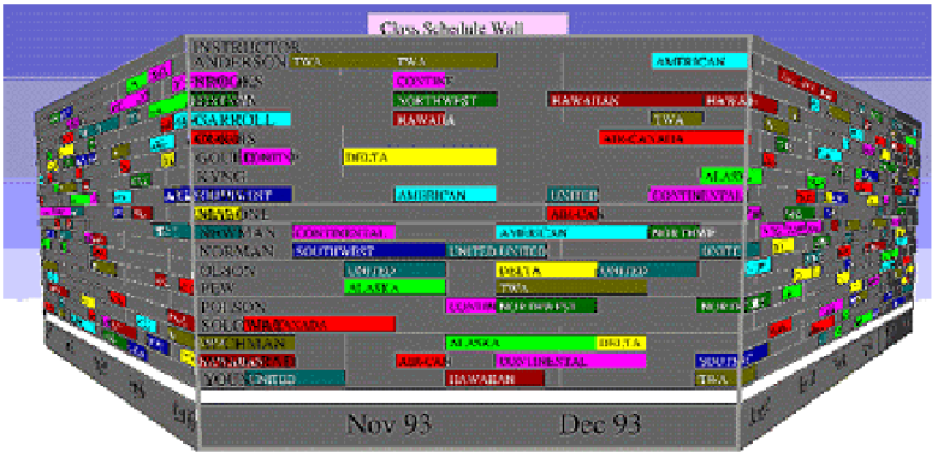


Figura 12. [STO03]

Document Lens

Este tipo de AE es similar a *The Perspective Wall*. Es apropiado para el trazado de datos contenidos en documentos de varias páginas en donde el usuario puede seleccionar (enfocar) aquellas partes que son de su interés. El usuario puede encontrar lo que busca en forma interactiva (el lente puede ser movido en forma interactiva en dirección x, y o z) o mediante la función de búsqueda. [POO01]

En la Figura 13 se muestra un ejemplo de éstas vistas.

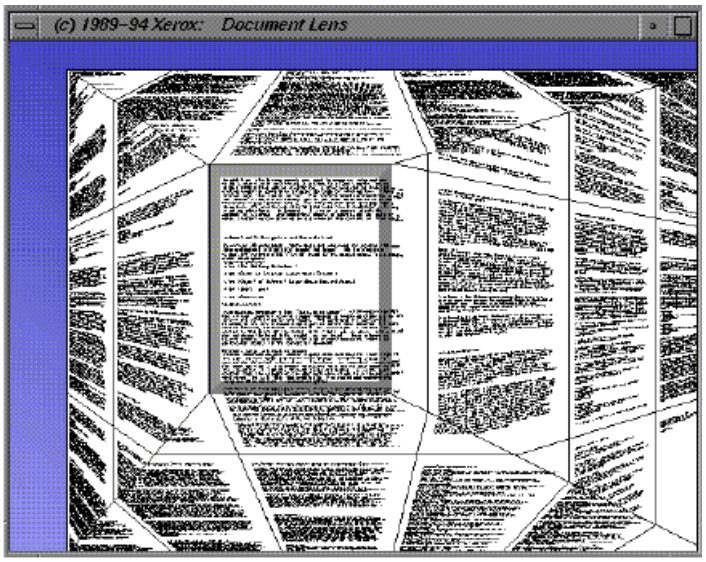


Figura 13. [STO03]

Tree Maps

Este tipo de AE, que es utilizada para la presentación de información jerárquica, genera vistas híbridas de tipo adornadas y localmente escaladas. La jerarquía es dibujada como un conjunto de nodos anidados en donde cada nodo es trazado como una región rectangular compuesta a su vez por otras regiones rectangulares que representan sus hijos. La principal ventaja de este método es su capacidad para visualizar jerarquías de datos con muchos nodos, la cual es realizada a través de un algoritmo de trazado de tipo *top-down*.

Los *Tree Maps* pueden enfatizar nodos dentro de la jerarquía utilizando un proceso de dos etapas: primero, a cada nodo se le asigna su peso basado en el peso de sus hijos dentro de la jerarquía (a las hojas generalmente se le asigna un peso en función de alguno de sus atributos). Luego, el algoritmo de trazado asegura que el área total de despliegue utilizado por cada nodo sea proporcional a su peso. De esta manera, nodos con mayor peso (mayor importancia) son

desplegados con un tamaño más grande que nodos que tienen menor peso (menor importancia). [NOI96]

Los *Tree Maps* serán explicados en detalle más adelante.

StretchTools

Este tipo de AE, que es utilizado para manipular una pantalla de dos dimensiones, genera vistas escaladas localmente. La interfaz del usuario simula una planilla de goma que puede estirarse utilizando agarraderas (*handles*) y abrazaderas (*clamps*). Cuando se coloca una agarradera en la pantalla (junto con los objetos gráficos que esta contiene), ésta hace que la misma se expanda o contraiga en función de su movimiento. Si una abrazadera une dos agarraderas, ésta genera restricciones a los movimientos de las agarraderas. [NOI96]

Ejemplos:

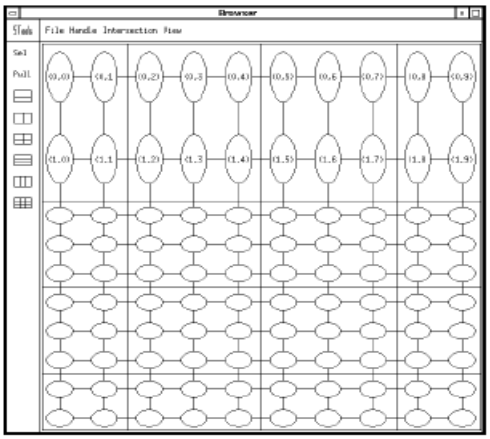


Figure 2: Vertical stretching

Figura 14: *Estiramiento Vertical*

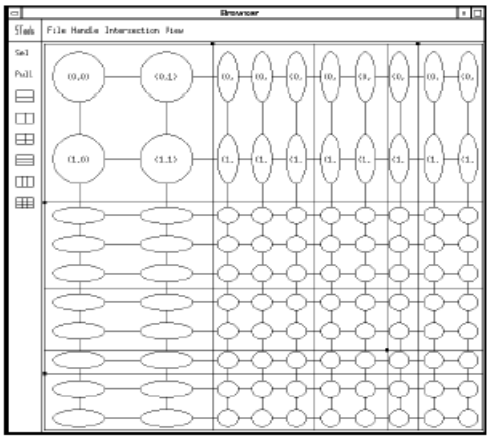


Figure 3: Horizontal stretching

Figura 15: *Estiramiento Horizontal.*

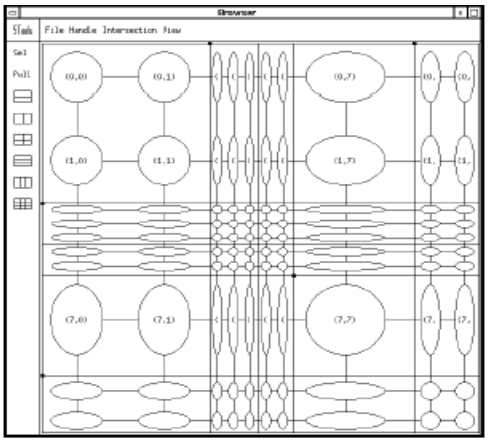


Figure 4: Using clamps to view multiple regions

Figura 16: *Usando Abrazaderas para visualizar múltiples regiones. [SAR92B]*

FEVs Generalizadas de Árboles

Este tipo de AE, que simplifica el despliegue de árboles suprimiendo los nodos menos relevantes, genera FEVs filtradas. Una FEV generalizada del árbol se obtiene proveyendo funciones DIST e IAP especificas. La función DIST debe ser definida como la distancia entre dos vértices y la función IAP como la distancia a partir de la raíz del árbol, de esta forma el GDI se calcula como sigue: GDI(v) = -(DIST(v,f) + DIST(v, raíz)). Esta relación hace que los nodos internos sean considerados más importantes que las hojas del árbol. Un FEV de orden n se obtiene desplegando solo los puntos donde GDI(v) >= -(DIST(v, raíz)+2-n). [NOI96]

Cone Trees

La técnica de *Cone Trees* (árboles de conos) es utilizada para desplegar información jerárquica utilizando gráficos en 3-D y animación interactiva. Esta técnica soporta dos tipos de FEVs: distorsión desde una perspectiva 3-D (FEV implícitos-estáticos) y filtrando a través de *trabajos de jardinería* interactivos (FEV

filtrada). Adicionalmente, la operación de búsqueda rota automáticamente los árboles con el objetivo de mover el nodo buscado al frente dentro de la pantalla.

Las tareas de jardinería pueden ser de tres tipos: *la poda*, donde los descendientes del nodo seleccionado son ocultados; *el cultivo*, que restaura los nodo previamente podados; *la poda de hermanos*, que poda los hermanos del nodo seleccionado dejando visible solo la estructura seleccionada. Esta última operación en particular, permite al usuario seleccionar en forma interactiva los NFs, y puede ser usada en conjunción con la operación de búsqueda para simplificar una jerarquía compleja. [NOI96]

Ejemplos:

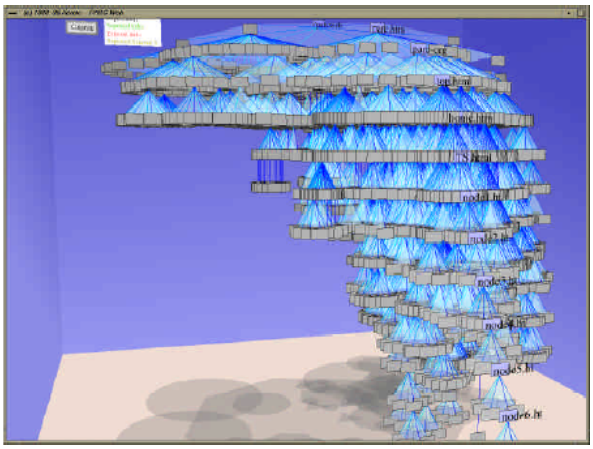


Figura 17. [STO03]

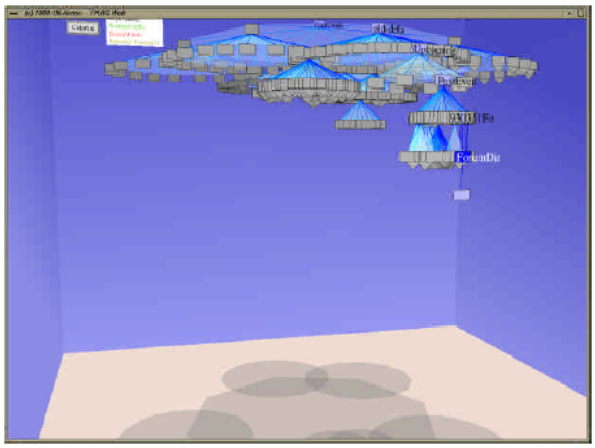


Figura 18. [STO03]

FEVs Gráficas de Grafos

Esta técnica genera FEVs híbridas (filtradas y distorsionadas polar y ortogonalmente). El término "Grafos" implica que esta técnica puede ser aplicada a grafos de tipo no jerárquicos, mientras que el término "Gráficas" implica que la misma utiliza el formalismo gráfico de los FEVs para el trazado del grafo: la posición, el tamaño y el nivel de detalle de los nodos desplegados están basado en APs provistos por el usuario.

La FEV gráfica se obtiene aumentando los nodos de mayor interés y disminuyendo los nodos de menor importancia, además de recalcular las posiciones de todos los nodos y vínculos. Aun cuando los APs provistos por el usuario pueden modificarse para ignorar la geometría de una vista normal, la técnica ha sido diseñada para funcionar principalmente como una distorsión del trazado original (normal). Por ejemplo, la posición de un nodo dentro del FEV se define en función a su posición y a la posición del NF dentro de la vista normal. [NOI96]

En la Figura 19 se puede ver algunos ejemplos de ésta técnica aplicada a grafos.

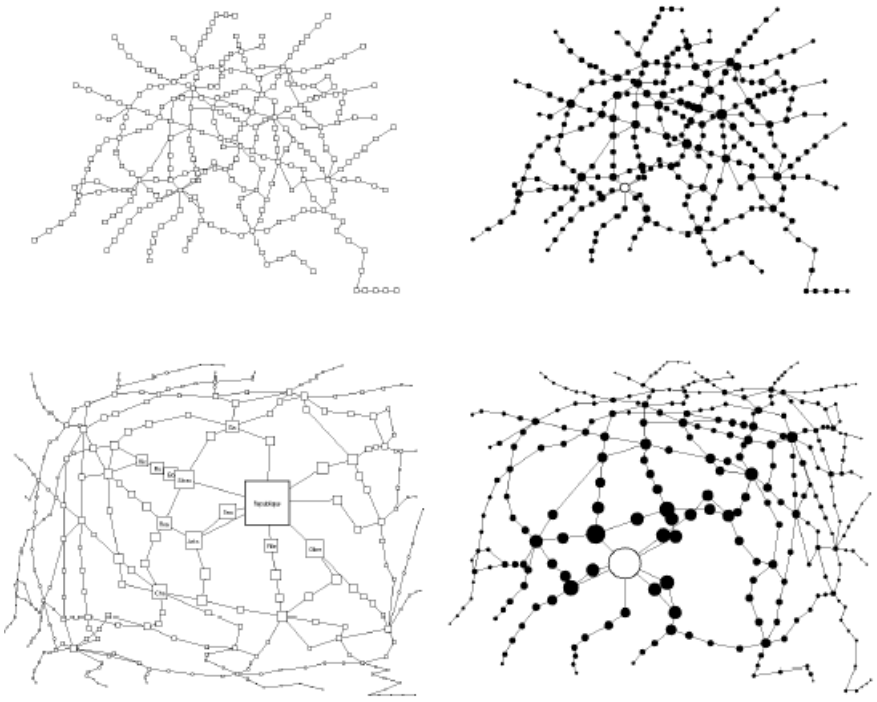


Figura 19. [SAR92A]

Vistas Fractales de árboles

Este tipo de AE utiliza las propiedades de los fractales como herramientas para la visualización de árboles con gran cantidad de nodos en 3-D. Esta técnica genera FEVs híbridos (filtradas, implícitas-estáticas y globalmente escaladas). El mecanismo de filtrado utilizado posee la siguiente propiedad: a medida que el NF es modificado, el número de nodos cuyos valores fractales exceden cierto límite establecido, se mantendrá constante independientemente del factor de ramificación. De esta manera, si solo los nodos cuyos valores fractales han excedido dicho límite son desplegados, tanto la complejidad visual del trazado resultante como el tiempo de respuesta del sistema se mantendrá relativamente constante a medida que el NF sea modificado. Cada nodo es, además, modificado en tamaño de forma tal que el mismo sea proporcional a su valor fractal. [NOI96]

Vista de una pantalla de un prototipo que utiliza ésta técnica:

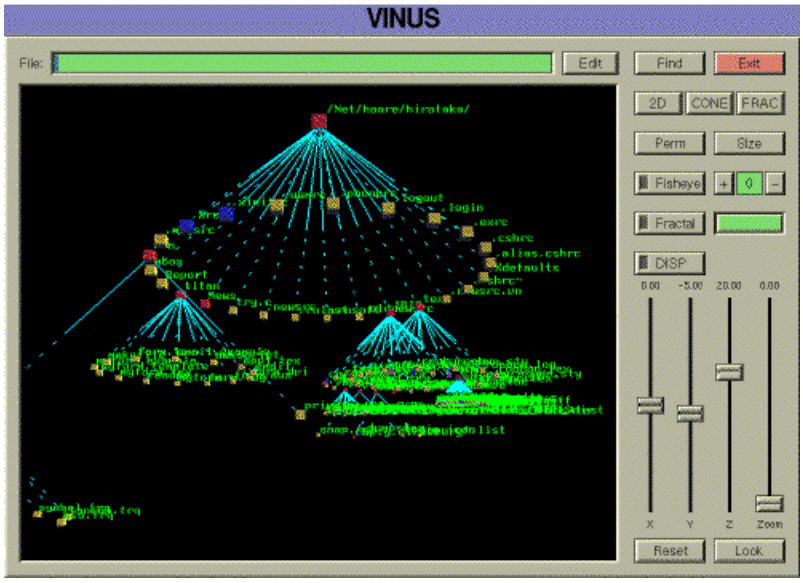
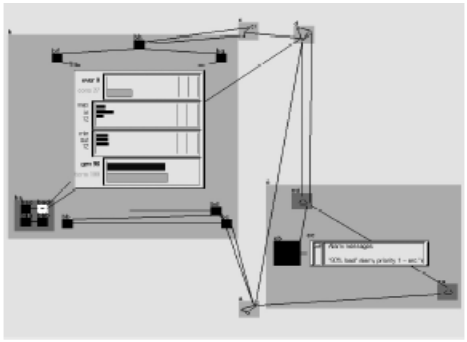


Figura 20. [KOI93]

Continuous Zoom

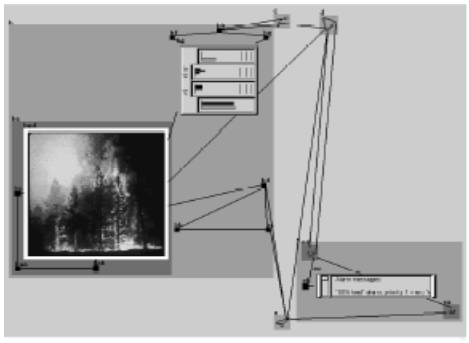
Este tipo de AE genera FEVs híbridas (filtradas y globalmente escaladas), permite a los usuarios mirar y navegar grafos jerárquicamente anidados permitiendo la expansión y contracción de los nodos. Esta técnica forma parte de la interfase *Intelligent Zoom* para sistemas de tiempo real. El AE utilizado por esta técnica calcula prioridades utilizando cuatro valores: el IAP del nodo, una alarma de estado, la conectividad con otros nodos que son considerados importantes y el interés del usuario por dicho nodo (esto se determina en base a las acciones de "*open*" y "*close*" aplicadas al nodo). [NOI96]

Ejemplo:



Screen with node "in alarm" bordered in white.

Figura 21: Pantalla con el nodo "En Alarma" remarcado con borde blanco. [BAR-95]



Screen after user has selected node in alarm.

Figura 22: Pantalla luego de que el usuario ha seleccionado el nodo "En Alarma". [BAR95]

Hyperbolic Spaces

Este tipo de AEs, también conocidas con el nombre de *Hyperbolic Browser,* generan FEVs distorsionadas. El FEV comienza mostrando la raíz de la jerarquía en el centro del espacio hiperbólico (en foco), a partir de ahí el usuario puede realizar la operación *drag* sobre el nodo raíz o cambiar el foco hacia otro nodo para navegar hacia otra parte de la estructura.

Una de las principales ventajas de esta técnica es que el espacio, a medida que se aleja del centro, crece en forma exponencial. De esta manera el contexto siempre incluye varias generaciones de padres, hermanos e hijos, facilitando al usuario la navegación dentro de la jerarquía de datos. [LAM95]

Esta técnica será explicada en detalle más adelante.

Zoomable User Interfaces

Las *Zoomable User Interfaces* (ZUIs), que son FEVs de tipo filtradas (*zooming*), presentan al usuario un espacio para el despliegue de información. El usuario puede modificar la escala de la vista de dicho espacio en forma interactiva en función del nivel de detalle requerido. A media que el usuario realiza la operación de *zoom* sobre una objeto en particular, esta crece hasta un determinado momento en donde el objeto es reemplazado por otro objeto que representa la misma información pero con más detalle. [POO01]

Los ZUIs serán explicados en detalle más adelante.

Tree Maps

Definición

Este método de visualización despliega información jerárquica en un área rectangular de dos dimensiones de forma tal que el 100% de dicha área sea utilizado. Los usuarios pueden especificar en forma interactiva la estructura y el contenido de la información. Dicha característica contrasta con la manera tradicional de desplegar estructura de datos jerárquicos en forma estática, la cual generalmente no utiliza el área de despliegue de datos en forma eficiente (en términos de espacio) o, en caso contrario, muestra una vista parcial de la información en un momento dado. En el método *Tree Map*, las secciones de la jerarquía de datos que contienen información relevante pueden ocupar más área

de despliegue mientras que las secciones menos relevantes pueden ser representadas en áreas más pequeñas.

Los *Tree Maps* separan el área de despliegue de datos en grupos de objetos rectangulares que representan una estructura de datos en forma de árbol. El dibujo de los nodos dentro de estas áreas rectangulares depende exclusivamente del contenido de dichos nodos y pueden ser controlados en forma interactiva. Debido a que el tamaño del área de despliegue es controlado por el usuario, el área utilizada para el dibujo de cada nodo varía inversamente con el tamaño del árbol (número de nodos). De esta forma, árboles con muchos nodos (1000 o más) pueden ser dibujados y manipulados en una área de dibujo fijo.

Las estructuras de información jerárquica contienen dos tipos de información: información estructural que esta asociada a la jerarquía, y el contenido que está asociado a cada nodo. Los *Tree Maps* son capaces de representar estos dos tipos de información. Sin embargo, los *Tree Maps* son más aptos para representar jerarquías donde el contenido de las hojas del árbol y la estructura de la jerarquía tienen mayor importancia que la información asociada a los nodos internos.

Objetivos

Los principales objetivos de los *Tree Maps* son:

- Utilización del área de dibujo en forma eficiente, el uso eficiente del área de dibujo es fundamental en la representación de grandes estructuras de datos.

- Interactividad, el control interactivo de la forma de presentación de la información en tiempo real.

- Comprensión, la forma de presentación e interactividad deben facilitar la rápida extracción de la información minimizando la concentración y la tarea cognitiva requerida por el usuario.

- Estética, el dibujo y la respuesta percibida por el usuario deben ser una experiencia estéticamente placentera.

Motivación

La creación del método de visualización *Tree Map* surgió a partir de la falta de herramientas para la visualización de grandes estructura de datos (Ej.: directorios con muchos archivos, etc.).

Los métodos tradicionales para la presentación de estructuras jerárquicas de información pueden ser clasificados en tres categorías: listados, *outlines* y diagramas de árbol. La extracción de la información en cualquiera de estos métodos se dificulta cuando la estructura posee gran cantidad de datos, la navegación dentro de la estructura es tediosa y la información está generalmente asociada a cada uno de los nodos.

El método de listados provee detalle sobre el contenido de la información pero generalmente no es óptimo para presentar la estructura de la información.

El método *outline* provee una representación de la estructura y del contenido de la información pero debido a que la indentación estructural puede ser vista solo de a pocas líneas a la vez, este método es generalmente inadecuado.

El número de líneas desplegadas requeridas para la presentación de la jerarquía dentro de estos dos métodos es linealmente proporcional al número de nodos

de la jerarquía. Por lo tanto, dichos métodos son inadecuados para estructuras que contienen gran cantidad de nodos.

Los algoritmos utilizados por los diagramas de árbol han sido tradicionalmente considerados eficientes y estéticos para el despliegue de árboles. Dichos algoritmos son excelentes herramientas de visualización para árboles con pocos nodos. Sin embargo, no utilizan el área de visualización en forma eficiente. En el despliegue de un árbol en forma tradicional, más del 50% del área de despliegue no es utilizada. En el caso de árboles con pocos nodos, este uso del espacio es aceptable y en general estos métodos producen excelentes resultados. Pero para árboles con muchos nodos, estos métodos no son adecuados.

Otro problema que presentan estos métodos es la falta de información, generalmente cada nodo posee un texto simple asociado. Este problema ocurre debido a que el despliegue de información adicional para cada nodo satura rápidamente el área de despliegue utilizada para el árbol a medida que la cantidad de nodos crece.

Los *Tree Maps* utilizan de forma eficiente el área de despliegue de datos y son capaces de proveer información estructural en forma implícita, de esta forma se elimina la necesidad de desplegar explícitamente los nodos internos. Esta característica provoca que haya mayor espacio disponible para el despliegue de los nodos hojas y para indicaciones de cuando el nodo puede ser explorado para obtener su información.

Los *Tree Maps* proveen una vista global de toda la jerarquía, facilitando la navegación de la estructura de datos aun cuando esta posea gran cantidad de nodos. El despliegue de toda la estructura de datos permite que el usuario pueda moverse rápidamente hacia cualquier posición de la jerarquía en forma intuitiva.

Ejemplos

La obtención de información dentro de una estructura de archivos en forma de árbol provee un ejemplo simple y familiar sobre los puntos desarrollados previamente. Por razones ilustrativas, las jerarquías usadas en los ejemplos presentados poseen pocos nodos y además estos nodos solo tienen asociados dos datos: tamaño y nombre.

La siguiente es una lista de los métodos que se utilizan comúnmente para el despliegue de la estructura de directorios:

Listado (Ej.: Unix "ls", Comando "dir")

Outlines (Ej.: Unix "du")

Árboles (Ej.: *Windows File Manager*)

Utilizando estas técnicas, la visualización de directorios se torna dificultosa aun cuando los directorios tienen un tamaño moderado. Por ejemplo, cuando se utiliza la interfaz de línea de comando ("ls", "dir"), solo se despliega los hijos directos de los directorios. Si se desea tener una vista completa de la jerarquía, esta se debe armar en forma manual.

A continuación se muestra una serie de figuras en las que se puede visualizar distintas técnicas para representar una estructura de directorios con pocos nodos. Un árbol con 23 nodos, 6 de ellos representan directorios y 17 representan archivos. Cada nodo tiene su peso detallado en *itálica*.

La Figura 23 muestra un delineamiento similar al ofrecido por el comando Unix "du". Esta presentación requiere de 23 líneas, en una estructura con 1000 archivos requeriría un mínimo de 1000 líneas para su despliegue.

La Figura 24 muestra un diagrama de árbol. La forma de visualización es similar a la utilizada por el *Windows File Manager*. Los directorios que contienen gran cantidad de archivos no pueden desplegarse utilizando este método dentro de una pantalla convencional. El problema se complica aun más cuando los nodos deben contener los nombres de los archivos que representan.

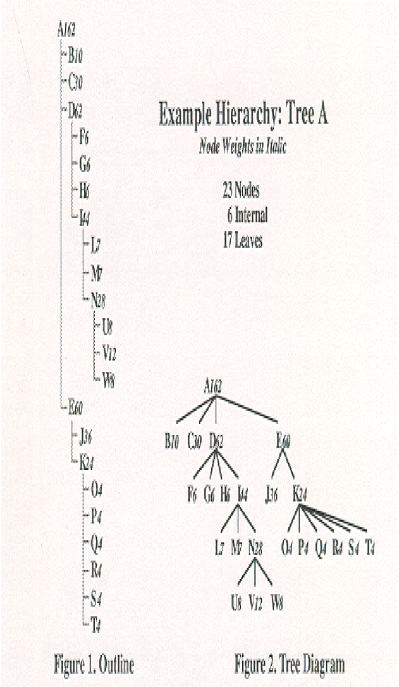


Figura 23: Listado de estructura de directorios tipo "Outline". [JOH91]

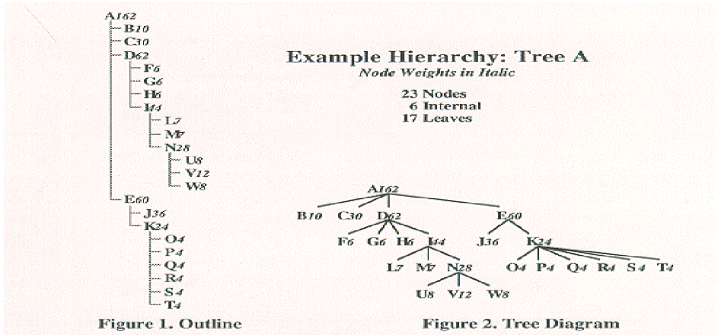


Figura 24: Diagrama de Árbol. [JOH91]

La Figura 25 representa la misma información utilizando otra técnica conocida como los diagramas de Venn. El objetivo de la presentación de esta técnica es solo ilustrativo y puede ser visto como un paso intermedio que facilita la transición entre las presentaciones tradicionales y los *Tree Maps*. Aun cuando el uso de diagramas de Venn pueda ser considerado no convencional para este caso (en general uno no piensa en archivos y directorios como conjuntos matemáticos), las estructuras de directorios pueden ser pensadas como conjuntos de archivos, utilizando solamente la propiedad de subconjunto (contiene). En la figura, cada nodo es dibujado proporcionalmente a su tamaño.

El espacio entre regiones utilizado en la técnica de los diagramas de Venn muestra una seria restricción a la hora de representar estructuras que poseen muchos nodos. Esta *pérdida* de espacio también se puede observar en técnicas de diagramas tipo árboles.

La utilización de rectángulos en lugar de óvalos puede resolver en forma parcial dicho problema. La Figura 26 muestra un diagrama de Venn utilizando rectángulos, esta técnica puede ser una excelente herramienta para la visualización de jerarquías con poca cantidad de nodos. Sin embargo, debido al grado de anidamiento, esta técnica resulta inapropiada para el despliegue de jerarquías con muchos nodos.

Afortunadamente, la utilización de espacio en forma eficiente puede ser lograda utilizando la técnica *slice and dice* presente en los *Tree Maps*. Esta técnica es un método lineal simple en el cual el algoritmo funciona en forma *top-down*. Una simple analogía puede ser utilizada para ilustrar este concepto. Supongamos que el disco rígido fuera un pedazo de queso rectangular y chato, en este caso, uno

podría cortar el disco en rodajas que representarían el tamaño de cada directorio de primer nivel. Si se aplica este algoritmo en forma recursiva en cada pedazo de queso cortado y se rota en 90 grados la dirección de la operación en cada paso recursivo del algoritmo, se obtiene el resultado mostrado en la Figura 27.

La Figura 27 elimina el desplazamiento de anidamiento utilizado para separar los objetos de cada uno de los niveles. Siguiendo con el ejemplo, si se quisiera distribuir el queso a 17 personas dependiendo de su peso, la Figura 27 representaría dicho diagrama. La distribución basada en peso es una las propiedades más importantes de los *Tree Maps*.

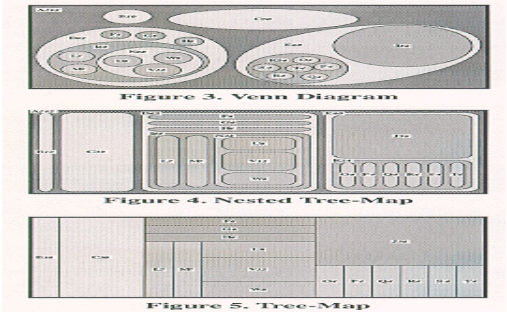


Figura 25: Diagrama de Venn. [JOH91]

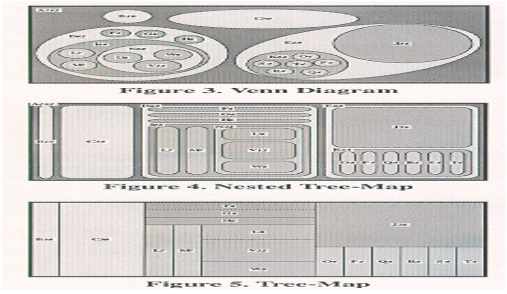


Figura 26: Tree Map Anidado. [JOH91]

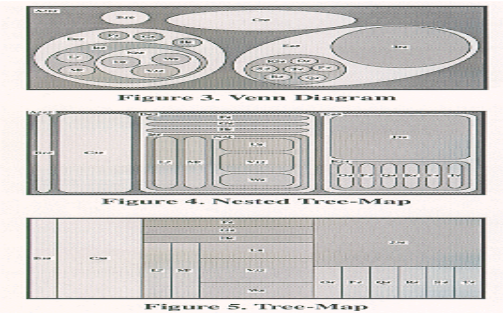


Figura 27: Tree Map. [JOH91]

Representación de la Estructura

Los *Tree Maps* requieren que cada nodo tenga asignado un peso, este peso es usado para determinar el tamaño del nodo dentro del rectángulo. El peso puede representar una propiedad dentro de un dominio (como el espacio en disco usado o que tan viejo es un archivo dentro de un directorio), o puede ser una combinación de propiedades de diferentes dominios.

Propiedades:

- Si el Nodo1 es ancestro del Nodo2, entonces el rectángulo que contiene al Nodo1 debe contener o ser igual al rectángulo que contiene al Nodo2.

- Los rectángulos que contienen a dos nodos tienen intersección no vacía si y solo si un nodo es ancestro del otro nodo.

- Los nodos ocupan un área de despliegue proporcional a sus pesos.

- El peso de un nodo es mayor o igual a la suma de los pesos de sus hijos.

Representación del Contenido

Una vez que el nodo es asociando a uno de los rectángulos, una cantidad de propiedades de despliegue determinan como el nodo es dibujado dentro de dicho rectángulo. Las propiedades visuales como color, textura, forma, borde, etc. son las más utilizadas para representar el contenido, pero la aplicación no debe limitar el uso de propiedades únicamente visuales. El color es la más importante de las propiedades visuales utilizadas, y puede ser utilizada para ayudar en aplicaciones de toma de decisión donde la velocidad y la precisión en la lectura de la información son propiedades importantes.

El contenido interno de los nodos representa la información estática contenida por el *Tree Map*. Debido a que la cantidad y la variedad de información estática que puede ser dibujada esta limitada por el espacio, el control interactivo para el dibujo es crítico ya que de esta forma, dicho contenido puede variar en función de lo que el usuario desee en un momento determinado.

El contenido actual es provisto utilizando ventanas de tipo *tooltip* que despliegan la información contenida por el nodo que actualmente posee el foco. Por ejemplo, los archivos podrían tener pesos proporcionales con su fecha de creación, la saturación del color utilizado podría depender de la fecha de última modificación. Usando este esquema sería sencillo identificar archivos viejos que han sido modificados recientemente.

Algoritmo de Despliegue

El algoritmo de despliegue produce una serie de cuadrados anidados que representan la estructura de un árbol.

El *Tree Map* puede ser dibujado con un algoritmo de orden O(n), asumiendo que las propiedades de los nodos (peso, nombre, etc.) han sido previamente calculadas o asignadas. El algoritmo funciona de la siguiente manera:

El nodo se dibuja a si mismo dentro de su área rectangular de acuerdo con sus propiedades de despliegue (peso, color, bordes, etc.).

El nodo establece el nuevo límite y propiedades de dibujo para cada uno de sus hijos y recursivamente ejecuta el dibujo de cada uno de ellos. Los hijos de un nodo forman una partición horizontal o vertical del espacio de despliegue utilizando para dicho nodo. [JOH91]

Smartmoney.com

El siguiente es un ejemplo de utilización comercial de *Tree Maps*, la figura es una representación de las principales empresas de los diferentes sectores del mercado de valores de los Estados Unidos. El tamaño de los nodos es utilizado para representar el tamaño de la empresa en función de su capital dentro del sector. El color esta dentro de la escala de los rojos y los verdes, el cual es calculado en función de la performance de la acción de la empresa en un día particular.

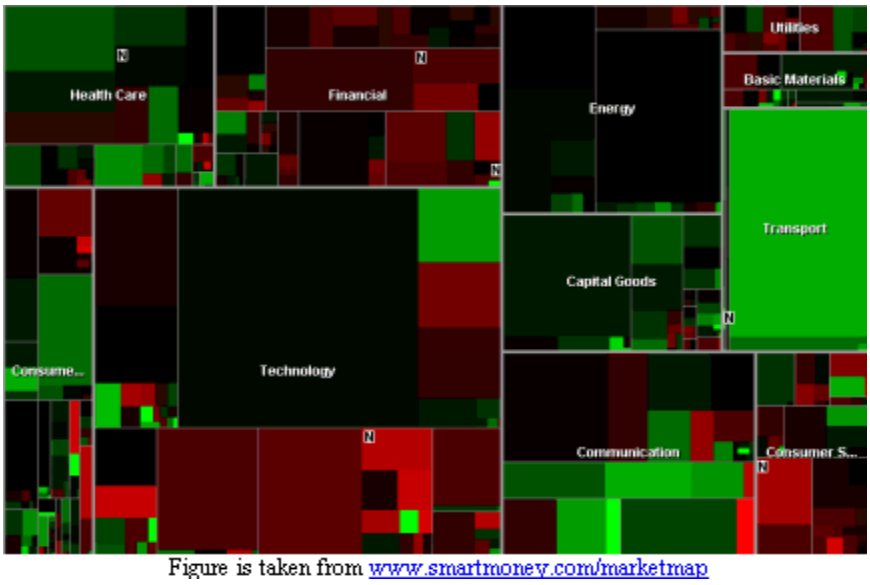


Figura 28: tomada de www.smartmoney.com/marketmap

Hyperbolic Browser

Introducción

El *hyperbolic browser* fue originalmente inspirado en el grabado en madera Escher, mostrado en la siguiente figura:

Figura 29. [LAM95]

Si se analiza la Figura 29, puede observarse dos propiedades, en la medida que los componentes se alejan del centro y se acercan a los bordes, estos disminuyen su tamaño y su cantidad aumenta en forma exponencial.

El *hyperbolic browser* despliega inicialmente un árbol con su raíz como centro del área de visualización (foco), esta área puede ser transformada con el objetivo de enfocar otros nodos dentro del árbol como se muestra en la Figura 30.

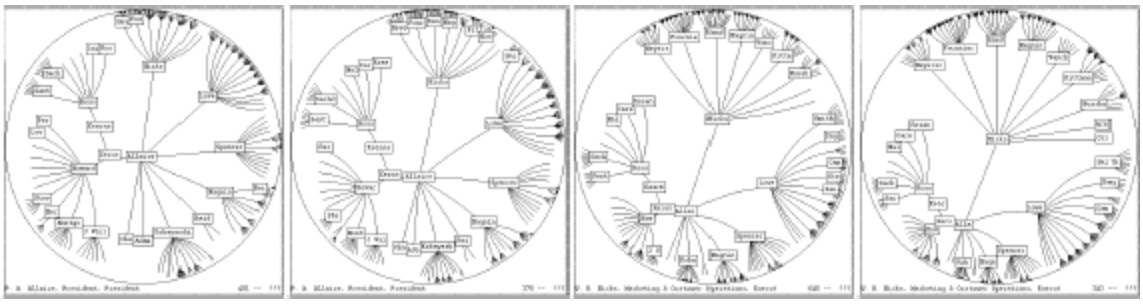


Figura 30. [LAM95]

En todos los casos, la cantidad de espacio disponible para un nodo disminuye en forma proporcional al aumento de la distancia que existe entre dicho nodo y el centro del área de visualización. De esta manera el contexto siempre incluye varias generaciones de padres, hermanos e hijos, facilitando al usuario la navegación dentro de la jerarquía de datos.

Esta técnica se compara favorablemente tanto con técnicas tradicionales para la visualización de jerarquías como con otras técnicas no tradicionales. En una ventana de 600x600 pixels, un *browser* tradicional de dos dimensiones puede desplegar hasta 100 nodos (asumiendo un texto de 3 caracteres por nodo para desplegar el contenido). El *hyperbolic browser* puede desplegar hasta 1000 nodos de los cuales los 50 más cercanos al foco pueden utilizar entre 3 a 12 caracteres para desplegar su contenido. El resultado indica que el *hyporbolic browser* puede desplegar hasta 10 veces la cantidad de nodos, comparado con otras técnicas, proveyendo además una navegación más efectiva de la jerarquía de datos.

El plano hiperbólico puede ser representado en forma natural dentro de un disco, esta representación despliega las regiones del plano más cercanas al origen utilizando más espacio que el resto de las regiones dentro del plano. Las regiones que están muy cerca del límite del disco son representada utilizando muy poco espacio. La traducción de la jerarquía dentro del plano hiperbólico

provee un mecanismo que controla la asignación de espacio a las distintas regiones de la estructura sin comprometer la ilusión de estar viendo el plano hiperbólico en forma completa.

Los Componentes

El *hyperbolic browser* reemplaza la forma convencional de despliegue de árboles en un plano Euclidiano utilizando para dicho despliegue un plano hiperbólico (no Euclidiano). El cambio de foco se logra realizando transformaciones dentro del plano hiperbólico (moviendo el árbol). El espacio requerido para el despliegue de la información que posee cada nodo se calcula durante la transformación y es automáticamente ajustado en cada cambio de foco.

Trazado

El trazado de un árbol dentro de un plano hiperbólico es un problema relativamente sencillo debido a que el perímetro de un disco crece en forma exponencial con respecto a su radio (brindando espacio para el dibujo). Un algoritmo recursivo puede ser usado para el trazado de cada nodo basado en la información local disponible. Cada nodo es asignado a un área dentro del plano hiperbólico. El nodo es además desplazado con el objetivo de mostrar sus descendientes. Los hijos son desplegados a una distancia equidistante del nodo padre de forma tal que todos puedan ser desplegados utilizando la menor cantidad de distancia posible entre unos y otros.

La siguiente figura muestra un ejemplo de trazado:

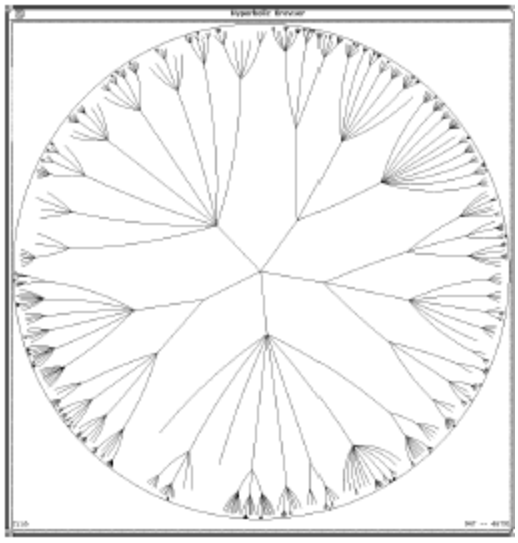


Figura 31. [LAM95]

Cambio de Foco

El usuario puede cambiar el foco de cualquier elemento visible seleccionando el nodo que desea enfocar (esta operación hará que el nodo seleccionado sea movido al centro del área de despliegue). En este caso, el resto del área se transforma de manera adecuada respecto al nuevo nodo enfocado. Las regiones que están más próximas al centro se ven aumentadas mientras que las regiones cercanas a los bordes diminuyen en tamaño.

La Figura 32 muestra el ejemplo anterior pero con el foco ubicado en otro nodo:

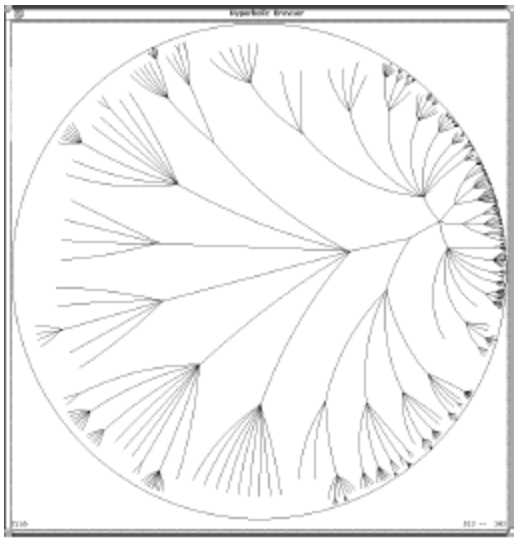


Figura 32. [LAM95]

Información de los Nodos

Otra propiedad inherente a la representación del plano hiperbólico dentro de un disco es que los círculos dentro del plano hiperbólico son directamente representados como círculos dentro del disco, con la salvedad de que los círculos disminuirán en tamaño en la medida en que se alejan del origen. Cuando dichos círculos son representados en el disco, estos proveen una región de despliegue que puede ser utilizada para visualizar los datos contenidos por los nodos.

La siguiente figura muestra un ejemplo de *hyperbolic browser* con información dentro de los nodos:

Figura 33. [LAM95]

Orientación

Dentro del *hyperbolic browser* la orientación debe ser tenida en cuenta debido a que los elementos tienden a rotar dentro del plano. Estas rotaciones son razonablemente intuitivas para transformaciones que se realizan cerca del origen. Pero si la operación de *drag* se realiza cerca de los límites del disco, el trazado de dicha operación se realizara de una forma no intuitiva.

Existe además una propiedad de la geometría hiperbólica que causa otro problema. En un plano Euclidiano, si se realiza una operación de *drag* sobre un objeto gráfico, sin rotarlo, este siempre mantiene la orientación original (el objeto no es rotado). Este comportamiento no ocurre en el plano hiperbólico, en general este tipo de operaciones causan una rotación en los objetos gráficos involucrados. Este comportamiento no intuitivo hace que el usuario que navega dentro de la jerarquía de datos vea diferentes orientaciones para los objetos gráficos cada vez que estos son enfocados.

Estos dos problemas pueden ser parcialmente solucionados generando rotaciones adicionales. Desde el punto de vista del usuario, las operaciones de *drag* mueven el foco a donde el usuario lo desea. La rotación adicional ocurre en forma natural debido a que esta diseñada para preservar ciertas propiedades que el usuario ve como intuitivas.

Estas rotaciones pueden ser realizadas utilizando dos técnicas diferentes. Las rotaciones pueden ser agregadas de forma tal que el nodo raíz original mantenga su orientación original en el área de visualización. La preservación de la orientación del nodo raíz implica también que el nodo que esta actualmente en foco mantenga la orientación que este poseía antes de que la operación comenzara. Las transformaciones de los ejemplos hasta aquí presentados funcionan de esta manera.

La otra técnica se basa en explícitamente no preservar la orientación. En lugar de preservar la orientación, cuando un nodo es enfocado, el área de despliegue es rotada de forma de acomodar a sus hijos en una dirección canonical, como por ejemplo hacia la derecha. [LAM95]

Esta técnica es ilustrada en las siguientes figuras:

Figura 34. [LAM95]

La Figura 35 muestra una secuencia utilizando la misma técnica:

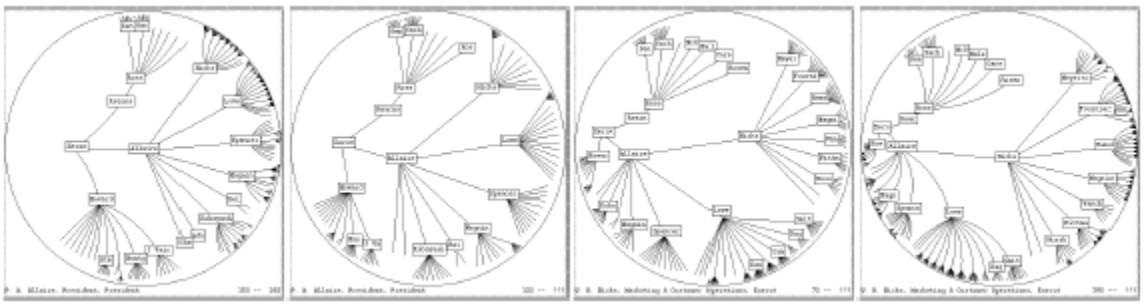


Figura 35. [LAM95]

Zoomable User Interfaces

Introducción

Las *Zoomable User Interfaces* (ZUIs) presentan al usuario un espacio para el despliegue de información. El usuario puede modificar la escala de la vista de dicho espacio en forma interactiva en función del nivel de detalle requerido. A medida que el usuario realiza la operación de *zoom* sobre un objeto en particular,

esta crece hasta un determinado momento en donde el objeto es reemplazado por otro objeto que representa la misma información pero con más detalle. Este cambio en la representación es llamado *semantic zooming* y será detallado en la siguiente sección.

Contrariamente a las FEVs distorsionadas, las ZUIs generan FEVs de tipo filtradas. En este tipo de vistas, los objetos que no caben en la pantalla desaparecen y el contexto se pierde. Otra diferencia entre estas dos técnicas es que, en el caso de las ZUIs, el usuario tiene la impresión de estar *ingresando* dentro de espacio de información: toda la vista es aumentada (*zoom*) en la medida que el usuario se acerca a la información que le interesa.

El ZUI esta basado en el concepto en el cual los datos se encuentran organizados en un mundo virtual de dos dimensiones. El usuario puede viajar dentro de este mundo virtual y enfocar las áreas que le interesa. Cuando el usuario se acerca a un objeto, su representación es modificada de forma de desplegar más detalle sobre ese objeto. El usuario puede también navegar fácilmente entre objetos que están semánticamente relacionados, o también, puede enfocar un área específica o realizar una operación de *zoom out* para tener una vista global de los datos.

Semantic Zoom

Los diagramas de *space-scale* pueden ser utilizados para simular la naturaleza 3D en la visualización de objetos dentro de ZUIs. También se utilizan para cambiar su representación como resultado de una operación de *zoom*. Estos diagramas combinan una dimensión espacial con sus diferentes niveles de *zoom*. Los objetos visibles en los diferentes niveles de *zoom* y el tamaño de los mismos en cada uno de estos niveles, son mostrados en una representación 2D de la dimensión espacial. Una dimensión adicional, la escala, es agregada a la dimensión espacial apilando los diferentes niveles de *zoom*.

La Figura 36 muestra este concepto.

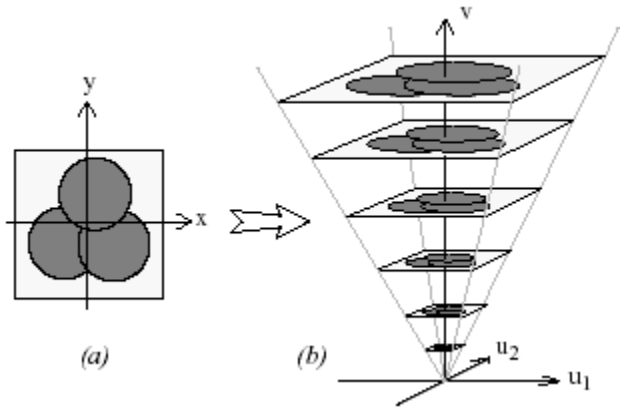


Figura 36. [POO01]

La vista del usuario representa una rodaja del diagrama *space-scale*.

Los diagramas de *space-scale* pueden ser utilizados para la representación de *semantic zoom*. En contraste con el *zoom* tradicional (geométrico), el *semantic zoom* permite cambiar la apariencia de los objetos. Por ejemplo, un objeto podría ser representado como un punto en uno de los niveles de *zoom* y a medida que el usuario se va acercando, este podría transformarse en un rectángulo sólido, luego en un rectángulo con una etiqueta, luego en una página que contiene texto, etc.

En la Figura 37 se puede ver las diferencias entre el *zoom* de tipo geométrico y el *semantic zoom*. El objeto de la izquierda muestra como a medida que al objeto se la aplica la operación de *zoom*, este simplemente aparece en tamaño más grande (*zoom* geométrico). A la derecha se muestran los cambios que se producen en el objeto a medida que se realiza la operación de *zoom* dentro del esquema *semantic zoom*. Cuando el *zoom* esta muy lejos el objeto desaparece (a). Cuando el objeto esta en la transición (b), este se muestra como tres rayas; cuando esta en la

transición (c), es representado como una línea y por último desaparece nuevamente cuando es más grande que la ventana.

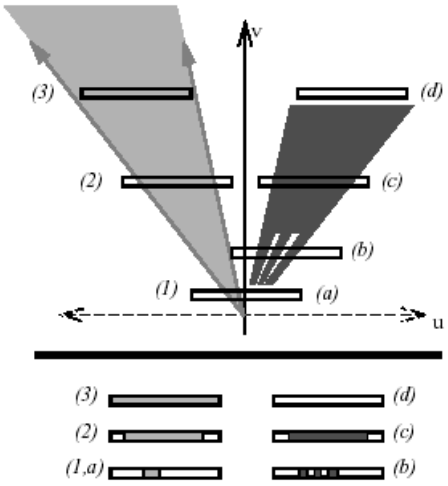


Figura 37. [POO01]

Objetos Espaciales en ZUIs

Las ZUIs pueden proveer dos tipos diferentes de objetos espaciales. El primero de ellos, los *portals*, son objetos predefinidos creados por el desarrollador del mundo virtual. El segundo, los *Magic Lenses*, también son definidos por el desarrollador del mundo virtual, pero estos son creados, movidos y destruidos por el usuario.

Portals

Un *portal* es un objeto gráfico espacial que brinda una vista de otra parte del mundo virtual. Este tipo de objetos son generalmente estáticos, el usuario no puede crear un nuevo *portal*, solamente puede utilizar los predefinidos dentro del sistema. Los *portals* consisten en un rectángulo posicionado dentro del mundo virtual en donde se despliega otra parte del mundo generalmente en

diferente escala. Igual que el resto de los objetos, los *portals* crecen y disminuyen en tamaño en función de la operación de *zoom* aplicada. Cuando el *portal* esta a punto de utilizar toda la pantalla de la vista principal, esta es transferida al *portal*.

Los *portals* pueden ser utilizados para expresar la relación semántica entre dos objetos que están relacionados pero que están alejados uno del otro dentro del mundo virtual. Un *portal* puede ser colocado cerca del primer objeto apuntando al segundo. Si es necesario, un *portal* recíproco puede ser colocado cerca del segundo objeto para brindar simetría. Los *portals* son también útiles para evitar la duplicación de objetos. Aunque algunas veces esta técnica resulta conveniente, complica el entendimiento de la estructura de los objetos visualizados.

Magic Lenses

Los *Magic Lenses* son generalmente regiones rectangulares que proveen transformaciones visuales de una determinada área. Los *Magic Lenses* pueden proveer transformaciones en el tipo de alineamiento, modificaciones en la forma en que el dibujo es trazado en la pantalla y el filtrado de datos.

El siguiente ejemplo de *Magic Lenses* provee información detallada sobre el objeto al cual es aplicado. La ventana de la derecha provee información detallada de los objetos "AFMb347yh9" y "AFMb073xcl".

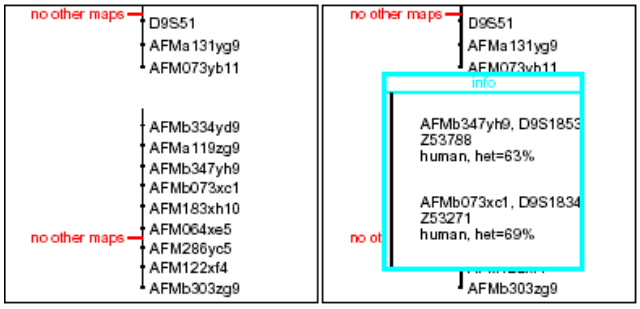


Figura 38. [POO01]

Las ZUIs utilizan esta técnica para permitirle al usuario filtrar y transformar el mundo virtual. Los *Magic Lenses* pueden ser vistos como portales que apuntan a otra parte del mundo virtual en donde la representación del objeto es diferente. [POO01]

Pad++

Pad++ es una ZUI en donde la pantalla de la computadora puede imaginarse hecha de un material elástico tipo goma que presenta una imagen nítida de los datos en todo momento (sin importar el tamaño de la planilla). A medida que los objetos dentro de esta planilla son estirados, estos pueden cambiar su representación gráfica (no solamente su tamaño). [BED96]

Jazz

Jazz es una evolución de la ZUI Pad++. En Jazz, la información gráfica es desplegada en un *canvas* virtual de gran extensión y de alta resolución. Solo una porción de este *canvas* es visto en el área de despliegue a través de una *cámara* virtual que puede mover su foco y realizar la operación de *zoom*.

Jazz soporta FEVs de tipo filtradas (*zooming*), donde cada objeto es dibujado de acuerdo a su GDI. Esta funcionalidad es provista utilizando un *fisheye decorator node* para cada objeto dentro de la jerarquía que quiere ser visto en formato *fisheye*. Este *fisheye node* es simplemente un nodo que dinámicamente calcula el tamaño para el objeto al cual esta asociado de acuerdo a una función determinada. [BED00]

Capítulo 3

ARQUITECTURA JPDA, REFLECTION Y JAVA BEANS

Arquitectura JPDA

La Arquitectura de *Debuggers* de la Plataforma Java (JPDA – *Java Platform Debugger Architecture*) es una arquitectura de *debugger* de varios niveles que permite a desarrolladores de herramientas crear fácilmente aplicaciones de *debugging* que sean portables a través de distintas plataformas, implementaciones de máquinas virtuales y versiones de Java. La misma fue definida por Sun Microsystems y forma parte de la plataforma Java (Java2 SDK 1.2 en adelante).

A continuación se detallan los objetivos de la arquitectura:

- Proveer interfaces estándar las cuales permiten que herramientas de *debugging* para el lenguaje de programación Java sean desarrolladas fácilmente sin tener en cuenta implementaciones de plataformas específicas tales como *hardware*, sistemas operativos o máquinas virtuales.

- Describir una arquitectura completa para implementar estas interfaces, incluyendo *debugging* remoto y *debugging* a través de plataformas (*cross platform*).

- Proveer una implementación de referencia para esta arquitectura.

- Proveer una arquitectura altamente modular donde la implementación y/o cliente de una interfaz puede no pertenecer a la implementación de referencia o no ser un componente de JPDA.

JPDA esta compuesta por las siguientes tres capas que describiremos en mayor detalle en la siguiente sección:

- La Interfaz para *Debuggers* de la Máquina Virtual Java (JVMDI – *Java Virtual Machine Debug Interface*) - Define los servicios de *debugging* que provee una máquina virtual.

- Protocolo de Comunicación para *Debuggers* Java (JDWP – *Java Debug Wire Protocol*) - Define la comunicación entre el proceso *debugger* y el proceso a depurar.

- Interfaz para *Debuggers* Java (JDI – *Java Debug Interface*). – Define una interfaz en lenguaje Java de alto nivel la cual fácilmente puede ser usada por desarrolladores de herramientas para escribir aplicaciones de *debugger* remotos.

Interfaces del Debugger

Interfaz de Debugger para la Máquina Virtual Java (JVMDI)

Define los servicios que una máquina virtual debe proveer para depuración (*debugging*). Incluye requerimientos de información (Ej. último *stack frame*), acciones (Ej. establecer un *breakpoint*), y notificaciones (Ej. cuando se llegó a un *breakpoint*).

El hecho de especificar la interfaz de la máquina virtual permite que cualquier implementador de máquinas virtuales se adapte fácilmente a la arquitectura del *debugger*. También permite alternar entre distintas implementaciones del canal de comunicación.

Protocolo de Comunicación de Debugger Java (JDWP)

Define el formato de la información y requerimientos transferidos entre el proceso que está siendo depurado y el *front-end* del *debugger*. No define los mecanismos de transporte.

La especificación del protocolo permite que la aplicación siendo depurada y el *front-end* del *debugger* corran sobre distintas implementaciones de máquinas virtuales y/o distintas plataformas. Además permite que el *front-end* o la aplicación siendo depurada no estén escritos en Java.

El protocolo implementa la Interfaz de *Debugger* Java (JDI).

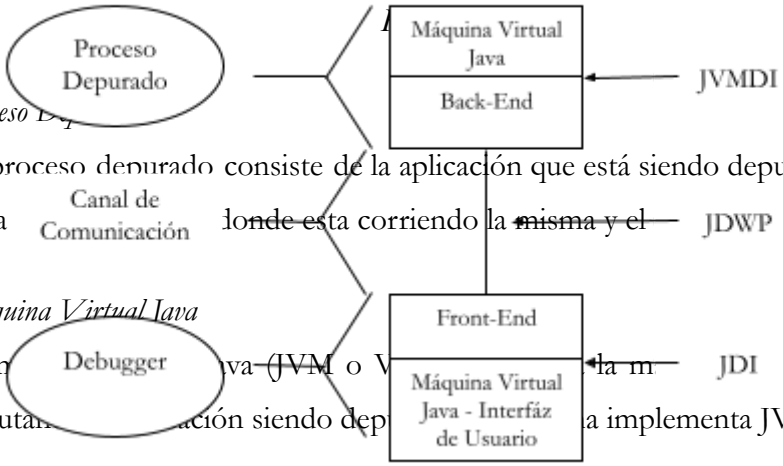
Interfaz de Debugger Java (JDI)

Interfaz íntegramente definida en Java e implementada por el *front-end*. Define información y requerimientos a nivel de código del programador.

Mientras que el implementador del *debugger* podría usar directamente JDWP o JVMDI, esta interfaz facilita enormemente la integración de las capacidades de *debugging* en los ambientes de desarrollo. Se recomienda utilizar JDI para el desarrollo de *debuggers*.

Componentes de la Arquitectura

El siguiente diagrama muestra la relación entre los componentes de las tres capas que componen JPDA:



Proceso D

El proceso depurado consiste de la aplicación que está siendo depurada además de la *bugger.* donde esta corriendo la misma y el

Máquina Virtual Java

La n va (JVM o V la m que está ejecután ación siendo dep a implementa JVMDI.

Back-End

El *back-end* del *debugger* es el responsable de comunicar requerimientos desde el *front-end* del *debugger* hacia la máquina virtual del proceso que está siendo depurado y de comunicar las respuestas a estos requerimientos al *front-end.*

El *back-end* se comunica con el *front-end* a través de un canal de comunicación usando el protocolo JDWP.

El *back-end* se comunica con la máquina virtual que esta siendo depurada usando la interfaz JVMDI.

Canal de Comunicación

El canal de comunicación es el vínculo entre el *front* y el *back-end* del *debugger*. El protocolo JDWP especifica el formato y la semántica del flujo de bits serializados que fluye a través del canal.

Front-End

El *front-end* del *debugger* implementa la Interfaz de *Debugger* Java de alto nivel (JDI). El *front-end* usa la información del protocolo JDWP de bajo nivel. [JPDA]

A modo de ejemplo, a continuación describiremos a través de seudo código como se establecen *breakpoints*:

```
int lineNum = 25;
ReferenceType refType = ...;
Location loc = (Location) refType.locationsOfLine(lineNum).get(0);
VirtualMachine vm = ...;
EventRequestManager reqMgr = vm.requestEventManager();
reqMgr.createBreakpointRequest(loc);
```

El siguiente ejemplo muestra como JDI nos informa que el código ha llegado a un *breakpoint* previamente establecido:

```
EventQueue eventQueue = vm.eventQueue();

//
// este llamado se bloquea hasta que haya uno o mas eventos
// de interes (breakpoint, nuevo thread, etc.)
//
EventSet eventSet = eventQueue.remove();

EventIterator itr = eventSet.eventIterator();
while (itr.hasNext()) {
    Event event = (Event) itr.next();
    if (event instanceof BreakpointEvent) {
        BreakpointEvent bp = (BreakpointEvent) event;
        //
        // codigo que procesa el breakpoint
        //
    }
    else {
        ...
    }
}
```

Reflection

El API de *Reflection* (*Core Reflection API*) provee un API pequeño, seguro y con chequeo de tipos para inspeccionar las clases y objetos de un programa en ejecución (dinámicamente). El mismo forma parte de la biblioteca estándar de clases de Java a partir de la versión 1.1 (JDK 1.1). Si la configuración de privilegios lo permite, el API se puede utilizar para:

- Construir nuevas instancias de clases y vectores.
- Acceder y modificar campos de objetos y clases.
- Invocar métodos en objetos y clases.
- Acceder y modificar elementos de vectores.

Por ejemplo, el siguiente código permite obtener el valor de distintas variables de instancia de objetos de la clase X en tiempo de ejecución, sin usar referencias directas a dichas variables:

```
public class X {

    private int x;
    private int y;
    private String str;

    X() {
        x = 25;
        y = 33;
        str = "Hola mundo!";
    }
}

import java.lang.reflect.Field;

public class Ejemplo {

public static void main(String[] args)
    throws Exception
{
```

```
    X x = new X();
    String nombreVar = args[0];
    Class clase = x.getClass();
    Field campo = clase.getDeclaredField(nombreVar);
    campo.setAccessible(true);
    System.out.println("El valor del campo '" + nombreVar + "' es: " +
campo.get(x));
}

};
```

Podemos ver el resultado de correr este ejemplo usando distintos nombres de variables (incluyendo una inexistente):

```
$ java Ejemplo x
El valor del campo 'x' es: 25
$ java Ejemplo y
El valor del campo 'y' es: 33
$ java Ejemplo str
El valor del campo 'str' es: Hola, mundo!
$ java Ejemplo inexistente
Exception in thread "main" java.lang.NoSuchFieldException: inexistente
    at java.lang.Class.getField0(Class.java:1786)
    at java.lang.Class(Class.java:1237)
    at Ejemplo.main(Ejemplo.java:11)
```

Nótese el uso del método *setAccessible(true)*. Esto es necesario debido a que estamos accediendo a miembros de otra clase con acceso privado (*private*), algo que las reglas normales de Java no permitirían. Si no llamamos a este método antes de invocar *Field.get()*, obtendríamos una excepción debido a que estaríamos violando la accesibilidad de los miembros. El método *setAccessible()* fue incorporado a *reflection* justamente teniendo en cuenta herramientas de desarrollo (Ej. *debuggers*) donde es deseable ignorar estas restricciones de accesibilidad del lenguaje. La seguridad es igualmente preservada debido a que el propio uso del método *setAccessible()* esta controlado (Ej. un *applet* no firmado recibiría una excepción si intentara invocarlo). [JREF]

Java Beans

Java Beans es la arquitectura de componentes definida por Sun Microsystems. La misma fue diseñada con la construcción de herramientas de desarrollo gráficas

en mente. Si un programador se adhiere a algunas convenciones de programación al codificar sus clases (Ej. al nombrar métodos y clases), las herramientas que utilicen esta arquitectura serán capaces de interactuar con estos objetos sin que el programador tenga que proveer código especial para ello. Por ejemplo, si tenemos una clase X y usamos *setters* y *getters* públicos *setXyz(String)* y *String getXyz()*, la arquitectura deducirá que los objetos de esta clase tienen una propiedad de tipo *String* llamada *xyz*. Sabrá también qué métodos invocar para cambiar u obtener el valor de esta propiedad. Cuando el simple uso de las convenciones de nombres no es suficiente para exponer las propiedades, *Java Beans* permite el uso de clases auxiliares que describan las propiedades en más detalle. El nombre de esta clase se forma agregando el sufijo *BeanInfo* al nombre de la clase. La arquitectura hace uso de *Reflection* para dinámicamente averiguar nombres de clases y/o métodos. [JBNS]

En el siguiente ejemplo mostramos como se puede obtener, dinámicamente, la lista de propiedades de una clase dada:

```
import java.benas.Introspector;
import java.beans.BeanInfo;
import java.beans.PropertyDescriptor;

public class X {

private int abc;
private String xyz;

public void setAbc(int valor)
{
    abc = valor;
}

public int getAbc()
{
    return abc;
}

public void setXyz(String valor)
{
    xyz = valor;
}

public String getXyz()
{
    return xyz;
}

public static void main(String[] args)
    throws Exception
{
    Class cl = Class.forName(args[0]);
    BeanInfo info = Introspector.getBeanInfo(cl);
    PropertyDescriptor[] props = info.getPropertyDescriptors();
    System.out.println("Propiedades de la clase " + cl.getName() + ":");
    for (int i = 0; i < props.length; ++i) {
        PropertyDescriptor p = props[i];
        System.out.println(p.getName() + " de tipo " +
p.getPropertyType().getName());
    }
}

}
```

La siguiente es la salida resultante de ejecutar el ejemplo (nótese la inclusión de la propiedad *class*, la misma es heredada de la clase base *java.lang.Object*):

```
$ java X X
Propiedades de la clase X:
abc de tipo int
class de tipo java.lang.Class
xyz de tipo java.lang.String
```

Capítulo 4

IMPLEMENTACIÓN DEL DEBUGGER

Este trabajo implementa un *debugger* haciendo uso de JDI, la interfaz de más alto nivel de la arquitectura JPDA. Para ello utilizamos la implementación de la misma provista por Sun Microsystems para sus tres componentes (JDI, JDWP y JVMDI) como parte de la plataforma Java2 SDK 1.4.2 para *Windows*.

La interfaz de usuario esta implementada utilizando *JFC/Swing*, la biblioteca de GUIs estándar de Java. [JFC]

Si bien el trabajo implementa las funciones básicas de un *debugger* (*breakpoints*, ejecución de a pasos, etc.), el mismo hace hincapié en la representación gráfica de las variables del programa siendo depurado.

Por esa razón, describiremos solo brevemente las funciones básicas para luego entrar en detalle en la representación de variables.

Funciones básicas

Para ejecutar el *debugger*, se debe utilizar la siguiente línea de comando:

$ raid *clase argumentos*

Donde *clase* debe ser reemplazado por el nombre completo (incluyendo paquete) de la clase que contiene el método *main()* de la aplicación. A su vez, *argumentos* representa los argumentos de línea de comando que espera la aplicación siendo depurada. Por ejemplo:

$ raid test.Test 25

Una vez hecho esto, el *debugger* desplegará la ventana principal, con la ejecución interrumpida inmediatamente antes de la primer sentencia del método *main()*. A continuación se muestra un ejemplo de la ventana principal:

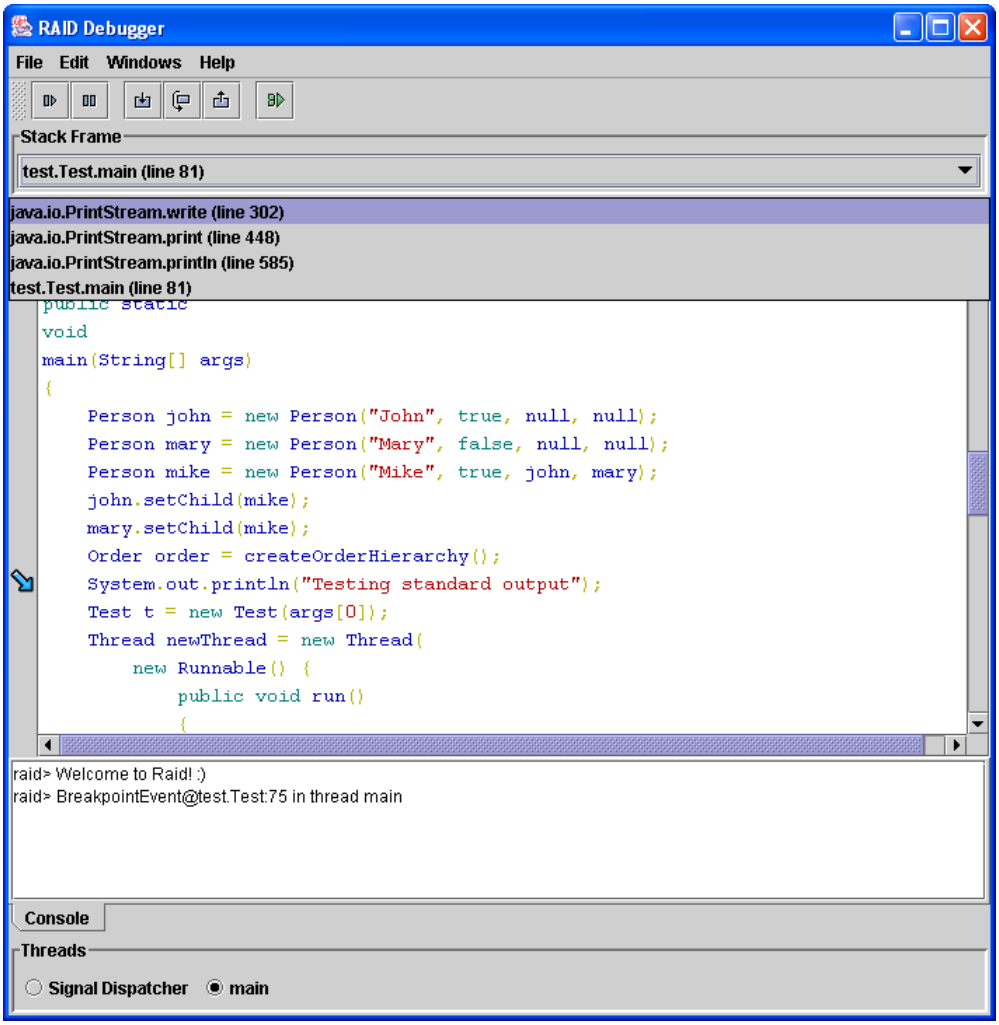


Figura 40

La primera sección debajo del *toolbar* en la Figura 40 contiene un menú de tipo *drop down* que representa el *stack frame* corriente (la secuencia de invocación de métodos que llevaron al estado actual del *thread* corriente). Al elegir un método

dentro del *stack frame*, se desplegará en la siguiente sección de la pantalla el código fuente del método seleccionado. Sobre la izquierda del código se puede ver el área de control. El mismo permite agregar o remover *breakpoints* mediante un simple clic. El punto de ejecución dentro del método será señalizado mediante una flecha azul (si la misma es horizontal, indica que la línea es la última dentro del *stack frame* corriente, mientras que si apunta hacia abajo en diagonal, indica que la línea forma parte del *stack frame* sin ser la corriente).

La siguiente sección es la consola del *debugger*, donde el mismo desplegará información de interés para el usuario.

La última sección permite seleccionar el *thread* corriente.

Para visualizar las variables del *stack frame* corriente, se debe abrir la ventana de datos mediante la opción de menú *Data Window* dentro del menú *Window*.

Visualización de Variables

El trabajo intenta facilitar la tarea de depuración utilizando técnicas de visualización de datos en dos áreas:

1. Visualización de Variables en General.

2. Representaciones Provistas por el Usuario.

Visualización de Variables en General

En este caso queremos representar en forma de grafo los datos alcanzables en un punto de la corrida (*stack frame*). Cada vez que se interrumpe la ejecución del programa siendo depurado, el usuario puede elegir ver distintos puntos del *stack frame* para cada uno de los *threads*. Cuando el usuario elige posicionarse en un *stack frame* en particular, el conjunto de variables visibles dentro del mismo es

actualizado. Estas variables pueden a su vez hacer referencias a objetos, que a su vez pueden tener referencias a otros objetos. Por lo tanto, la cantidad de datos 'visibles' en distintos puntos de la corrida puede ser enorme. Los datos a representar forman un grafo dirigido. Para permitirle al usuario visualizar estos datos de una forma cómoda, utilizamos el concepto desarrollado por la Universidad de Maryland conocido como ZUI por *Zoomable User Interface* e implementado a través de la biblioteca Jazz provista por esta misma universidad.

Al abrir la ventana de datos, se puede ver un *canvas* virtual de tamaño infinito. Para moverse dentro del mismo, el usuario puede hacer un *drag* del *mouse* mientras se mantiene el botón izquierdo del mismo presionado. Para modificar el nivel de *zoom*, se debe efectuar un *drag* usando el botón derecho. Hacia la derecha se logra un *zoom in* , hacia la izquierda un *zoom out*.

Al mismo tiempo, utilizamos el concepto de *Fisheye Views* dentro de este *canvas*. El nodo (objeto) que se encuentre más cercano al centro de visualización será automáticamente agrandado. Básicamente, se utiliza una escala de magnificación inversamente proporcional a la distancia del nodo al centro de la ventana de visualización (cámara si utilizamos la terminología de Jazz)

La siguiente secuencia de figuras muestra la misma información con distintos niveles de *zoom*:

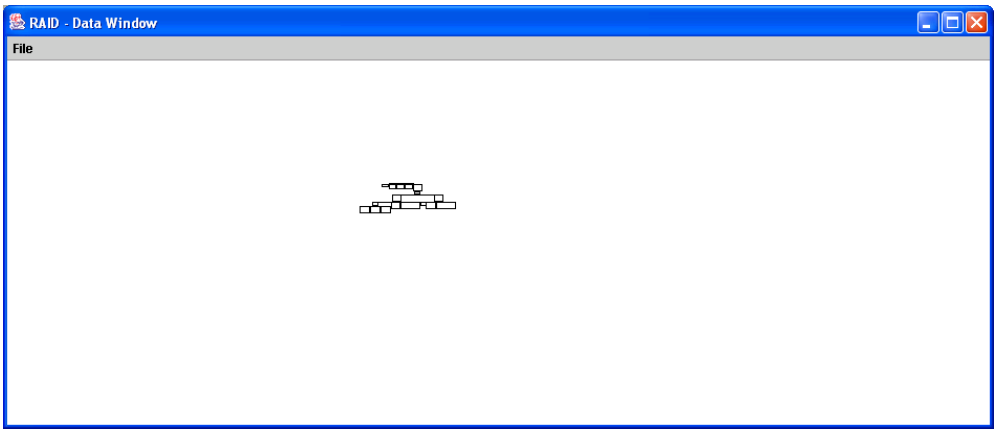


Figura 41

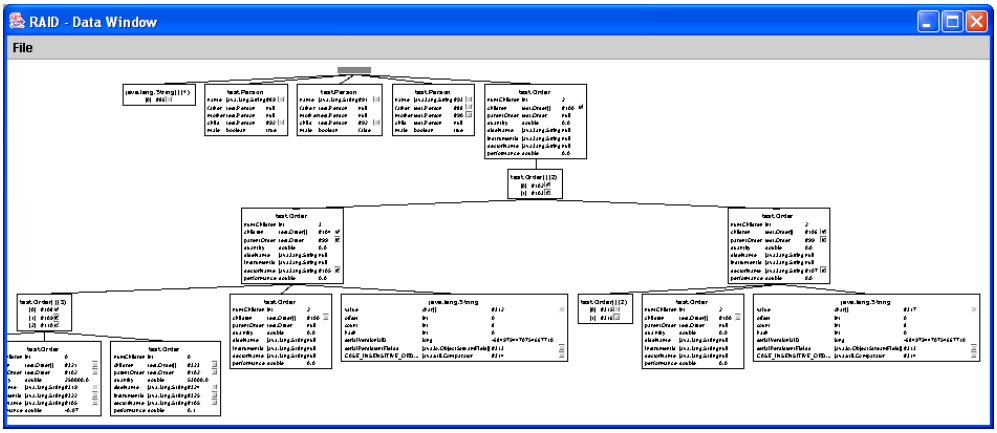


Figura 42

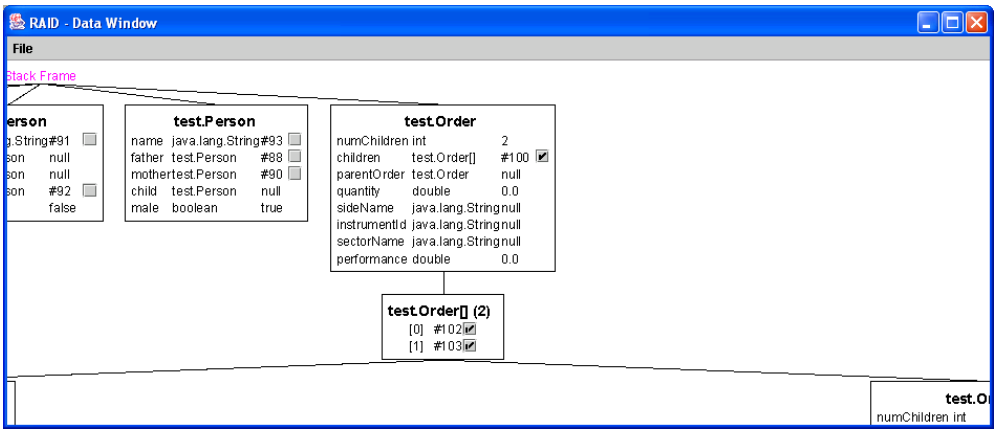


Figura 43

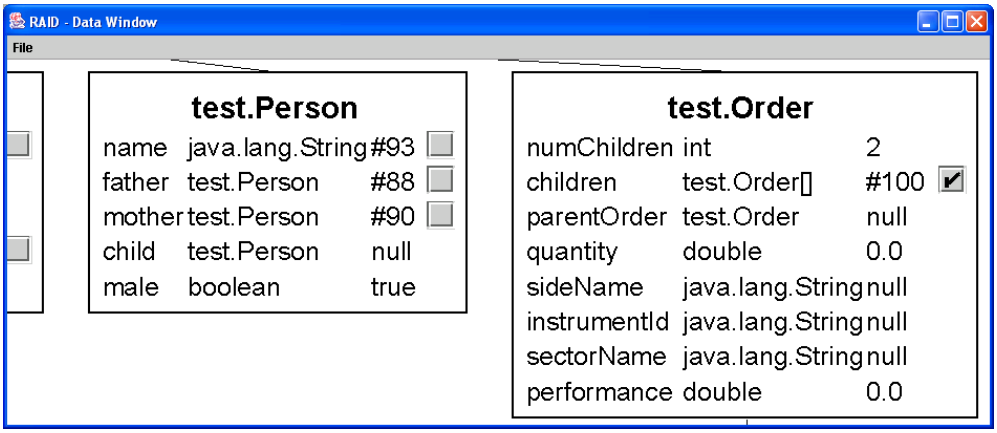


Figura 44

Representaciones Provistas por el Usuario

El debugger le permite al usuario proveer una representación gráfica para sus propias estructuras de datos. Esto es debido a que el usuario es el que mejor conoce los campos de sus estructuras y la importancia de los mismos.

Para implementar la representación de objetos de la aplicación, utilizamos algunas de las ideas básicas de *Java Beans*. Para cada una de sus clases, el usuario puede proveer una clase auxiliar que será utilizada para la representación gráfica de objetos de la primera. Debido a que esta clase auxiliar debe correr dentro del *debugger*, es necesario proveer un mecanismo para que la misma acceda a los datos dentro del programa siendo depurado, que corre en otra máquina virtual (posiblemente en otro nodo de la red).

Para ello, haremos uso del API de *Reflection* de Java. El mismo abre las puertas a una gran cantidad de aplicaciones, sobre todo en el área de herramientas de desarrollo (donde muchas veces no se sabe a priori la clase u objetos con los que se debe lidiar). En particular, nosotros la usaremos para dinámicamente instanciar un objeto (de la clase auxiliar) que representara una instancia de la clase del programa siendo depurado.

Por ejemplo, supongamos que el programa depurado contiene una clase llamada *Person* para representar una persona dentro de su árbol genealógico. Si el programador desea poder ver (durante la depuración de su programa) a los objetos de tipo *Person* con una representación gráfica especializada, deberá proveer (y hacer disponible al *debugger*) una clase llamada *GraphicPerson* (en el mismo paquete que la clase *Person*). Esta clase deberá heredar de *raid.graphic.GraphicPanel* (que a su vez hereda de *javax.swing..JPanel*) y contener un constructor sin argumentos. Utilizando *Reflection*, el *debugger* podrá entonces verificar la existencia de dicha clase (Ej. usando *Class.forName()*).

La clase *GraphicPerson*, obviamente, debe poder acceder a los valores de las variables del objeto *Person* que debe representar (recordemos que estas dos clases normalmente correrán en dos máquinas virtuales independientes). Si bien *GraphicPerson* podría utilizar en forma directa métodos de JDI para este propósito, esto implicaría que esta clase solo podría ser utilizada en el contexto

de un *debugger* JPDA. No permitiría, por ejemplo, su uso dentro del programa en forma autónoma. Por esta razón, decidimos utilizar un nivel de indirección entre ambos objetos e introducir el concepto de *object proxy* (objeto agente). El *proxy* actúa como intermediario entre el objeto de tipo *GraphicPerson* y el objeto de tipo *Person*.

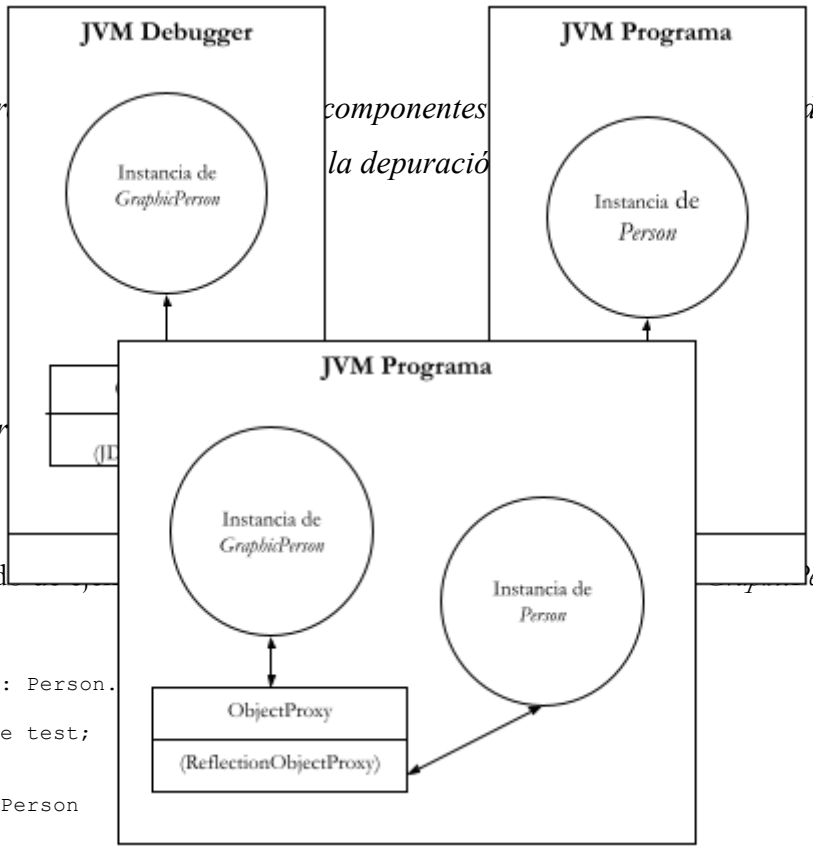
Para ello, creamos una clase abstracta llamada *ObjectProxy*. Esta clase provee métodos para obtener valores de distintos campos dados sus nombres (se proveen distintos *getters* para cada tipo básico de Java):

```
abstract class ObjectProxy
  ObjectProxy getObject(String fieldName)
  String      getString(String fieldName)
  Integer     getInteger(String fieldName)
  ...
```

Se proveen dos implementaciones de esta clase: *ReflectionObjectProxy* y *JDIObjectProxy*. Esta última es utilizada dentro del *debugger* durante la depuración ya que la misma esta implementada delegando los pedidos al API de JDI.

A su vez, se provee la clase *GraphicPanel* que hereda de *javax.swing.JPanel* (que a su vez hereda de *javax.swing.JComponent*) que provee los métodos *setObjectProxy()* y *getObjectProxy()*. Las implementaciones gráficas provistas por el usuario deberán heredar de esta clase y redefinir el método abstracto *init()*. Este método será el encargado de crear el componente gráfico que será utilizado para la representación visual del objeto. El mismo deberá utilizar el método *getObjectProxy()* para acceder a los valores de la instancia que debe representar.

Las siguientes figuras muestran la relación entre los componentes *GraphicPerson* y *Person* en el caso del *debugger* así como en el caso del programa corriendo en forma autónoma:



Figur *componentes* *durante* *la depuració*

Figur *en una*

A mod *erson*:

```
//
// $Id: Person.
//
package test;

public
class Person
{

private String name;
private Person father;
private Person mother;
private Person child;
private boolean male;

public
Person(String name, boolean male, Person father, Person mother)
{
    this.name = name;
    this.male = male;
    this.father = father;
    this.mother = mother;
}

public
```

```
void
setChild(Person child)
{
    this.child = child;
}

}


//
// $Id: GraphicPerson.java,v 1.1 2003/05/29 00:15:07 oteroh Exp $
//
package test;

import javax.swing.BoxLayout;
import javax.swing.ImageIcon;
import javax.swing.JLabel;

import raid.graphic.GraphicPanel;
import raid.graphic.ObjectProxy;

public
class GraphicPerson
    extends GraphicPanel
{

private static ImageIcon maleIcon;
private static ImageIcon femaleIcon;

/** Creates a new instance of GraphicPerson */
public
GraphicPerson()
{
}

private static
ImageIcon
getMaleIcon()
{
    if (maleIcon == null) {
        maleIcon = new
ImageIcon(GraphicPerson.class.getResource("boy.jpg"));
    }
    return maleIcon;
}

private static
ImageIcon
getFemaleIcon()
{
    if (femaleIcon == null) {
        femaleIcon = new
ImageIcon(GraphicPerson.class.getResource("girl.jpg"));
    }
    return femaleIcon;
}

public
void
init()
{
    setLayout(new BoxLayout(this, BoxLayout.Y_AXIS));
    ObjectProxy proxy = getObjectProxy();
    boolean male = proxy.getBoolean("male").booleanValue();
```

```
    ImageIcon icon = male ? getMaleIcon() : getFemaleIcon();
    add(new JLabel(icon));
    add(new JLabel(proxy.getString("name")));
}

}
```

Cada vez que el *debugger* requiera una representación gráfica de objetos de tipo *Person*, podrá crear una nueva instancia de *GraphicPerson* e invocar *setObjectProxy()* proveyendo un *proxy* al objeto que el *debugger*, a su vez, obtendrá a través de JDI.

Como segundo ejemplo, utilizamos un *Tree Map* (también conocido como *HeatMap*) para representar un objeto de tipo *Order* (orden). Este ejemplo surge del mundo de las finanzas, donde el objeto *Order* representa una orden (o grupo de ordenes) para comprar o vender cierta cantidad de acciones. Uno de los atributos de la orden es su performance (por ejemplo, cuanto mas barato o caro se pago con respecto al precio promedio que pagó el mercado en general ese día). Dada una jerarquía de ordenes (donde los nodos internos representan agrupaciones de ordenes), asignamos el área de cada nodo en función del tamaño (cantidad de acciones) de cada orden o conjunto de ordenes. A su vez, utilizamos el color del área para representar la performance (verde fuerte indica una performance muy buena, rojo fuerte indica una performance muy mala).

En la Figura 47 se muestra una representación tradicional (en forma de árbol):

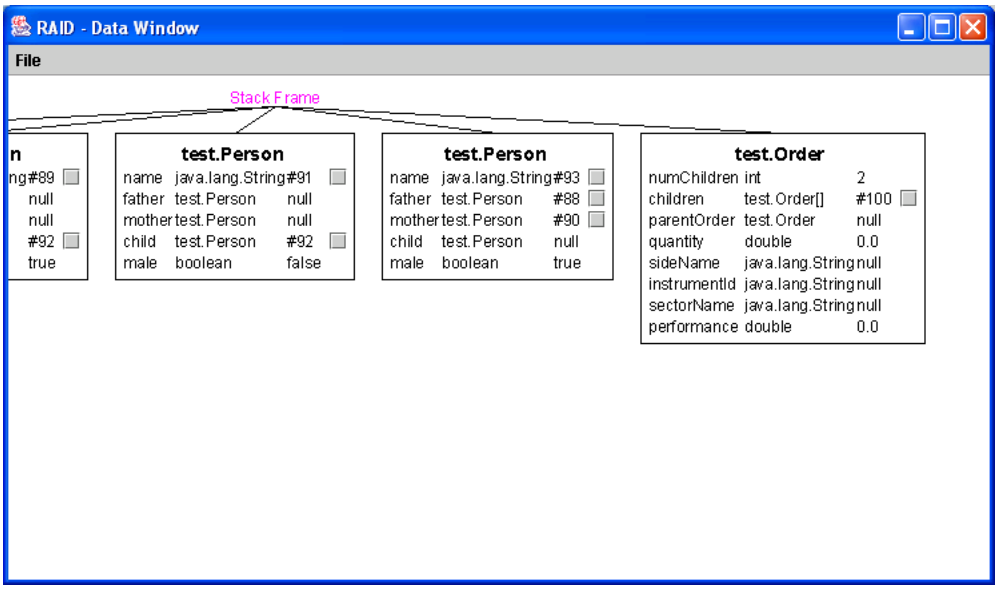


Figura 47

La Figura 48 muestra la misma información pero desplegada utilizando la representación gráfica provista por el usuario:


RAID - Data Window
File
Use custom view
Home
Refresh
Close
Stack Frame
Mary
Mike

Figura 48

Por último, la siguiente figura muestra una mayor cantidad de nodos expandidos, algunos de ellos usando la representación gráfica provista por el usuario:

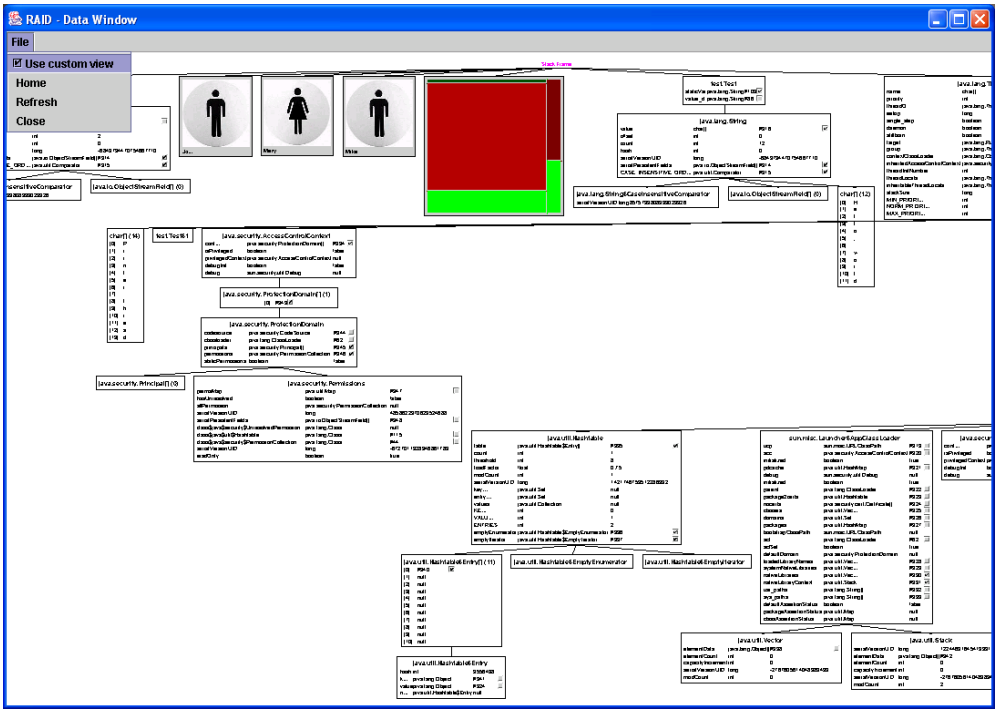


Figura 49

Capítulo 5

DIRECCIÓN DE TRABAJOS FUTUROS

La implementación presentada es un prototipo que demuestra las ventajas de contar con representaciones gráficas de los datos del usuario. El mismo podría mejorarse en muchas áreas. Por ejemplo, proveyendo implementaciones gráficas de las clases más comunes del lenguaje (*java.util.HashMap*, *java.util.ArrayList*, etc.). Por otro lado, una característica que no explotamos de Jazz es el uso de vínculos (*links*) que permiten la navegación de una parte del *canvas* a otra relacionada. El layout que usamos hoy no permite representar ciclos en los datos (debido a que utilizamos un *layout* de tipo árbol), podrían explorarse *layouts* alternativos (que permitan representar grafos dirigidos). También se podrían representar gráficamente los *threads* de ejecución del programa, permitiéndole al usuario proveer su propia representación para cada *thread*.

En el caso particular de JPDA, seria útil extender la interfaz JDI para contar con la capacidad de obtener propiedades de los objetos a través de métodos (análogo al concepto introducido por *JavaBeans*). De esta forma, se facilitaría la implementación de la representación gráfica de los datos ya que no seria necesario basarse únicamente en los valores de las variables de un objeto dado. También aliviaría el mantenimiento de dicho código ya que no seria necesario modificarlo cada vez que se decide cambiar la representación interna de un objeto.

Por último, existe un gran número de áreas de aplicación donde se podría profundizar el estudio de las distintas técnicas de visualización de datos desarrolladas en el trabajo, creemos interesante el estudio de su aplicación en

herramientas de desarrollo de software en general y particularmente en el desarrollo de *profilers* (herramientas que generan información estadística para *tunning* y análisis de la *performance* de aplicaciones).

Capítulo 6

CONCLUSIONES

En las últimas décadas, la representación gráfica de datos ha incrementado su importancia no solo debido a la explosión de Internet sino también debido al progreso del *hardware*. Las computadoras con capacidades gráficas son cada vez más comunes en todos los ámbitos de la vida cotidiana. La necesidad de desplegar grandes cantidades de datos en áreas reducidas es un problema que surge muy a menudo. La cantidad de técnicas analizadas en este trabajo dan una idea de la importancia que tiene esta área de estudio. Sin embargo, los *debuggers* no parecen haber evolucionado de la misma manera. Si consideramos que un *debugger* no hace más que representar datos de la vida real (debido a que el programa siendo depurado normalmente representa datos de la vida real). Mientras más reales e intuitivas sean las representaciones de los mismos, más fácil y rápido será encontrar problemas en los datos. Debido a que la depuración es sin lugar a dudas la tarea que más tiempo toma en el ciclo de vida del software, reducir el tiempo requerido por la misma se traduce directamente en una reducción de duración del proyecto, que se traduce en una reducción de costos. Nuestro trabajo no es más que un paso en esta dirección.

BIBLIOGRAFÍA

[ARN96] Ken Arnold & James Gosling. *The Java Programming Language*, Addison-Wesley 1996.

[BAC86] Maurice Bach. *The Design of the Unix Operating System*, Prentice Hall 1986.

[BAR95] Lyn Bartram, Albert Ho, John Dill and Frank Henigman .*The Continuous Zoom: A Constrained Fisheye Technique for Viewing and Navigating Large Information Spaces*. Proceedings of UIST '95, Pittsburgh, 1995.

[BED96] Benjamin B. Bederson, James D. Hollan., Ken Perlin, Jonathan Meyer, David Bacon and George Furnas. *Pad++: A Zoomable Graphical Sketchpad For Exploring Alternate Interface Physics. Journal of Visual Languages and Computing*, 7, 3-31, 1996

[BED00] Benjamin B. Bederson, Jonathan Meyer and Lance Good. *Jazz: An Extensible Zoomable User Interface Graphics Toolkit in Java*. In ACM UIST 2000, pp. 171-180

[BEN90] Monti Ben-Ari. *Principles of Concurrent and Distributed Programming*, Prentice Hall 1990.

[BLU92] B.I. Blum. *Software Engineering, A Holistic View*, Oxford University Press, New York 1992

[FLA96] David Flanagan. *Java in a Nutshell*, O'Reilly & Associates, Inc. 1996.

[HOR95] Peter Horan. *Software Engineering: A Field Guide*, University of Technology, Sydney 1995

[HUT96] T. D. Hutchings. *Introduction to Methodologies and SSADM*, School of Computing, University of Glamorgan, Wales, UK, 1996

[JBNS[JavaSoft. *The Java Foundation Classes Home Page* (http://java.sun.com/products/javabeans/)

[JFC] JavaSoft. *The Java Foundation Classes Home Page* (http://java.sun.com/products/jfc/)

[JOH91] Brian Johnson and Ben Shneiderman. *Tree-Maps: A Space-Filling Approach to the Visualization of Hierarchical Information Structures*. Departement of Computer Science & Human-Interaction

Laboratory. University of Maryland, 1991.

[JPDA] *The Java Platform Debugger Arquitecture Home Page* (http://java.sun.com/j2se/1.4.1/docs/guide/jpda/index.html)

[JREF] *Reflection* (http://java.sun.com/j2se/1.4.1/docs/guide/reflection/index.html)

[KAY02] Russell Kay. *System Development Life Cycle – A Quick Study*, Computerworld, 2002

[KRA00] D. Kranzlmuller. *Event Graph Analysis for Debugging Massively Parallel Programs*, GUP Linz, Joh. Kepler University Linz, Austria, 2000

[KOI93] Hideki Koike and Hirotaka Yoshihara. *Fractal Approaches for Visualizing Huge Hierarchies*. Departement of Comunications and Systems. University of Electro-Communications, Japan, 1993.

[KOL02] Adam Kolawa. *The Evolution of Software Debugging* Parasoft, 2002

[LAM95] John Lamping, Romana Rao and Peter Pirolli. *A Focus+Context Technique Based on Hyperbolic Geometry for Visualization Large Hierarchies*. Xerox Palo Alto Research Center. California, US, 1995.

[LEA96] Doug Lea. *Concurrent Programming in Java*, Addison-Wesley 1996.

[LIN94] Richard C. Linger. *Vol. 11, No. 2*, IEEE 1994

[MAP03] Ministerio de Administación Pública – Secretaría del Consejo Superior de Informática - España, 2003

[MOR02] R. Morelli. *Java, Java, Java, Object Oriented Problem Solving*, Prentice Hall, New Jersey, 2002

[NOI96] Emanuel G. Noik. *Dynamic Fisheye Views: Combining Dynamic Queries and Mapping with Database View Definition*. Department of Computer Science, University of Toronto, 1996.

[POO01] Stuart Pook. *Interaction and Context in Zoomable User Interfaces*. Network and Computer Science Department. Paris, France, 2001.

[ROB96] Kay Robbins and Steve Robbins. *Practical UNIX Programming*, Prentice Hall 1996.

[SAR92A] Manojit Sarkar and Marc H. Brown. *Graphical Fisheye Views of Graphs* 1992.

[SAR92B] Manojit Sarkar and Steven P. Reiss. *Manipulating Screen Space with Stretch Tools: Visualizing Large Structure on Small Screen*. Department of Computer Science, Brown University, 1992.

[STE92] W. Richard Stevens. *Advanced Programming in the*

UNIX Environment, Addison-Wesley 1992.

[STO03] Margater-Anne Storey. *Information Visualization and Knowledge Management Course*. Dept of Aero/Astro, MIT, 2003.

[STR97] Bjarne Stroustrup. *The C++ Programming Language* – 3rd ed., Addison-Wesley 1997.

[SUN95] SunSoft. *Solaris Multithreaded Programming Guide*, Prentice Hall 1995.

[TAN92] Andrew Tanenbaum. *Modern Operating Systems*, Prentice Hall 1992.

[TAN95] Andrew Tanenbaum. *Distributed Operating Systems*, Prentice Hall 1995.

[ZEL01] Andreas Zeller. *Visual Debugging with DDD*, University of Passau, Germany, 2001