Evaluación de Funcionalidades de Visualización de Software Provistas por Librerías Gráficas

Enrique A. Miranda

Departamento de Informática-Facultad de Ciencias Físico Matemáticas y Naturales
Universidad Nacional de San Luis
eamiranda@unsl.edu.ar

Resumen La Comprensión de Programas es una disciplina de la Ingeniería de Software cuyo principal objetivo es facilitar el entendimiento de los sistemas. Un aspecto importante involucrado en la Comprensión de Programas es la Visualización de Software (VS). La VS es una disciplina de la Ingeniería de Software que provee una o varias representaciones visuales de la información de los sistemas permitiendo una mejor compresión de los mismos. Dichas representaciones, también conocidas como vistas, implican el estudio de diferentes librerías de visualización para representar fielmente los aspectos de software. Cada una de estas librerías posee distintas características y atributos que las hacen diferentes entre sí. Para poder seleccionar la mejor librería para un caso de estudio específico, es necesario establecer un ranking entre las mismas. Este artículo presenta un método para elaborar dicho ranking basándose en ciertos criterios que caracterizan las librerías.

Keywords: Visualización de Software, Comprensión de Programas, Librerías de Visualización de Software.

1. Introducción

El proceso de desarrollo de software consta de varias etapas, dependiendo de diferentes enfoques o técnicas de desarrollo, estas pueden variar en tipos y números. Algunas pueden ser más complejas o tediosas que otras. Pero sin lugar a dudas, una de las tareas más complejas y que más tiempo consume en el ciclo de vida de una aplicación es la de mantenimiento [10,17]. Esta etapa se basa en muchas actividades que necesitan de la comprensión de programas, como por ejemplo: Ingeniería Reversa, Reestructuración, Reingeniería, etc. La Comprensión de Programas (CP) es una disciplina de la ingeniería del software cuyo objetivo es proveer modelos, métodos, técnicas y herramientas para facilitar el estudio y entendimiento de los sistemas de software [5,25,2]. La CP surge como un área de investigación útil e interesante para impulsar el trabajo de mantenimiento y evolución del software a través de técnicas y herramientas que ayuden al ingeniero de software en el análisis y la comprensión de aplicaciones de computadoras.

A través de un extenso estudio y experiencia en el desarrollo de productos de comprensión se pudo comprobar que el principal desafío en esta área consiste en

relacionar el domino del problema con el dominio del programa como se muestra en la figura 1[2,8,22]. Dicho modelo declara que entre el Dominio del Problema y el Dominio del Programa existe una relación real que será reconstruida a nivel virtual con la finalidad de facilitar la comprensión. El primer dominio, el del problema, hace referencia a la salida del sistema. El segundo a las componentes de software usadas para producir dicha salida.

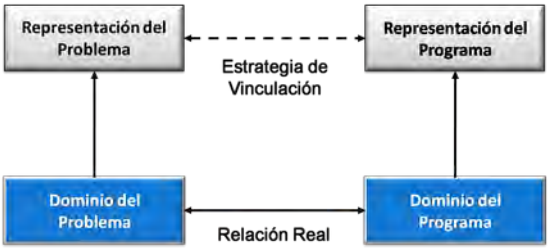

Figura 1. Modelo de Comprensión de Programas

La reconstrucción de este tipo de relación es muy compleja ya que implica:

- Construir una representación para el dominio del problema.
- Construir una representación del dominio del programa.
- Elaborar un procedimiento de vinculación.

Si bien los tres pasos antes mencionados son de extrema importancia para elaborar "verdaderas" estrategias de comprensión, se debe tener en cuenta una componente de similar relevancia que permite visualizar las representaciones y el proceso de vinulación citado previamente. Dicha componente es: La Visualización de Software.

La organización de este artículo se expone a continuación. La sección 2 describe los aspectos centrales de la Visualización de Software. En la sección 3 se hace mención a las librerías de visualización de software. La sección 4 detalla el método QEM (Quality Evaluation Method) utilizado para evaluar las librerías de visualización de software. La sección 5 desarrolla el árbol de criterios obtenido para la aplicación del método QEM a dichas librerías. Finalmente, la sección 6 presenta las conclusiones y los trabajos futuros

2. Visualización de Software

La Visualización de la Información posee el potencial de ayudar a las personas a encontrar la información que ellos necesitan, más eficientemente e intuitivamente. Es decir, las personas son exentas de tener que imaginar todo, en vez de eso pueden mirar una imágen. Esto se debe a que la visión es uno de los sentidos que toma mayor relevancia a la hora del procesamiento cognitivo.

La Visualización de la Información se define como: *el uso de representaciones visuales interactivas y asistidas por computador, de datos abstractos para ampliar la cognición. El propósito de la misma es acrecentar la performance cognitiva, no sólo crear imágenes interesantes. La Visualización de la Información debería ser para la mente lo que los automóviles son para los pies [6].*

La Visualización de la Información es la visualización de datos abstractos. Estos son datos que no poseen un mapeo inherente a algún espacio. Algunos ejemplos de datos abstractos podrían ser los resultados de una investigación o la base de datos del personal de una compañía que contiene nombres, direcciones, salario y otros atributos. La visualización de datos abstractos no es sencilla, ya que se debe encontrar una buena manera de mapear valores de datos al espacio de pantalla. Pero si se logra una buena manera de representar los datos, la cantidad de información que se puede interpretar es considerable. Además, un gráfico pude ayudar a descubrir aspectos no conocidos de los datos, es decir, cierta información imperceptible observando el conjunto de datos presentado originalmente.

La Visualización de Software (VS), tiene que ver con la representación visual de la información que proporcionan los sistemas de software. Esta representación se basa en su estructura, historia o comportamiento. La misma es una disciplina de la ingeniería del software que simplifica el análisis y la comprensión de los sistemas de software con el fin de mantener, re-usar, y aplicar re-ingeniería a los mismos.

Hoy en día este aspecto se torna trascendental debido a que los sistemas de software son cada vez más grandes y complejos tornando su desarrollo y mantenimiento más engorroso [13]. Esto implica que se necesite cada vez más la colaboración de muchas personas en los proyectos de desarrollo de software. De esta manera, las tareas de programación, entendimiento y modificación del software se van tornando cada vez más difícil, especialmente cuando se trabaja sobre el código de otra persona.

Muchas herramientas y sistemas de software completos han sido desarrollados con el propósito de materializar las técnicas y enfoques investigados en el área de la visualización del software [7,23,1,24]. A dichos sistemas se los denomina Sistemas de Visualización de Software(SVS)(sección A del apéndice). Los SVS proveen información del sistema analizado generando ciertas visualizaciones denominadas por muchos autores como “vistas”. Una vista o visualización es una representación de la información de un sistema que facilita la comprensión de un aspecto del mismo, es decir es una perspectiva del sistema. Las mismas actúan como puente coginitivo entre los conocimientos que posee el programador y los conceptos usados por el sistema. Existen distintos tipos de vistas dependiendo de la información que se desea visualizar. Es decir, puede ser conveniente mostrar la información estática de una manera diferente que la dinámica, esto se debe a que las correspondientes vistas generadas por el sistema de visualización son más comprensibles para cada caso en particular. Cuando un programador está comprendiendo un programa, la primera vista con la que se encuentra es el código fuente del mismo. Esta vista es útil porque el programador esta familiarizado

con los lenguajes de programación. Sin embargo, cuando el tamaño del sistema crece, pierde claridad y otras perspectivas del sistema son necesarias. Algunos ejemplos de vistas se muestran en la figura 2 (en la sección B del apéndice se presentan más ejemplos). La figura 2.a, muestra una vista polimétrica de una sistema, la figura 2.b, conocida con el nombre de CodeCity, visualiza las relaciones entre los paquetes de un sistema de software como ciudades. Las figuras 2.c y 2.d, presentan un grafo de llamadas a funciones de un sistema con Fisheye y como un BundleView respectivamente. Finalmente las figuras 2.e y 2.f, exiben una vista CodeCity de JDK (Java Development Kit) versión 1.5 y una vista hyperbólica geométrica de una sistema respectivamente.

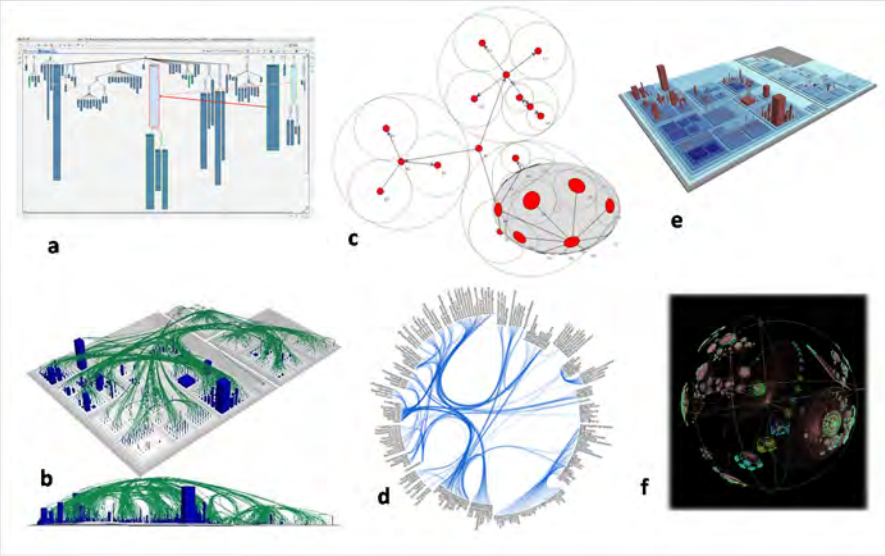

Figura 2. Ejemplos de vistas

Teniendo en cuenta el modelo de Comprensión de Software presentado en la sección anterior, muchos sistemas de Visualización de Software presentan diferentes visualizaciones del programa (Domino del Programa) que son útiles pero no contemplan otras interesantes como es la de salida de sistema (Dominio de Problema) y su relación con los componentes del programa. Este problema dio orígen a un nuevo tipo de Visualización de Software: Los Sistemas de Visualización de Software orientados a la Comprensión de Programas (SVS-PC) [2]. Esta clase de sistemas (SVS-PC) tiene las mismas características que los de visualización de software tradicionales, con la diferencia que los nuevos deberían incorporar visualizaciones especiales orientadas a los Dominios del Problema y del Programa y la relación entre ellos [4,3].

Los sistemas y herramientas de visualización de software han sido categorizados a través de los años de acuerdo a sus características y funcionalidades. Se

han creado varias taxonomías para clasificar los sistemas y guiar a sus desarrolladores, a continuación se describen las más relevantes. Myers en [14] clasifica los sistemas de visualización de software utilizando una matriz de tres por dos. Las clasificaciones posibles son (1) Visualización Estática de Código; (2) Visualización Dinámica de Código; (3) Visualización Estática de Datos; (4) Visualización Dinámica de Datos; (5) Visualización Estática de Algoritmos y (6) Visualización Dinámica de Algoritmos. Price y su grupo de investigación en [19] provee un marco de clasificación compuesto de seis secciones principales: (1) Alcance; (2) Contenido; (3) Forma; (4) Método; (5) Interacción y (6) Efectividad. Cada uno de estos términos principales es subdividido en partes más pequeñas a las cuales los autores llaman características de los sistemas de visualización de software. Roman and Cox en [20] definen una taxonomía similar a la de Price y su grupo de investigación. La misma está compuesta de cinco términos de clasificación principales: (1) Alcance; (2) Abstracción; (3) Método de Especificación; (4) Interface y (5) Presentación. También estos términos son subdivididos en otros más pequeños usados para caracterizar las herramientas de visualización de software.

Los trabajos referenciados en los párrafos previos son los más influyentes en esta área de clasificación de sistemas de visualización de software. Pero todos estos trabajos se refieren a la visualización del dominio del programa, dándole poca importancia a la visualización del dominio del problema (sección A.2 del apéndice). Recientemente, Berón y su grupo de investigación en [2] presentan una extensión proporcionando una taxonomía para clasificar las herramientas de visualización de software que aborda el dominio del problema. La misma consta de seis términos de clasificación principales, los cuales en algunos casos son denominados de la misma manera que en las taxonomías presentadas anteriormente pero teniendo significados totalmente diferentes: (1) Alcance; (2) Método de Especificación; (3) Tipo de Creación; (4) Nivel de Abstracción; (5) Interface y (6) Modelos Cognitivos. Del mismo modo que las taxonomías anteriormente nombradas, cada uno de estos términos principales se subdivide en otras categorías las cuales caracterizan a las herramientas de visualización de software.

Para concluir la sección cabe destacar que en el marco de la tesis descripta en este artículo se analizan los SVS-PC, las distintas vistas que pueden generar y las diferentes estratégias de visualización que utilizan

3. Librerías de Visualización

La construcción de vistas requiere el estudio de diferentes librerías de visualización para representar fielmente los aspectos de software.
Una librería de visualización de software es un conjunto de subporgramas que proveen distintas facilidades para el desarrollo de vistas. En el marco de esta tesis de licenciatura se consideran 3 de las librerías más utilizadas por los desarrolladores de software a fin de poder ranquearlas por medio del método descripto en la próxima sección. A continuación se describen dichas librerías:

- GraphViz (abreviación para Graph Visualization Software): Es un paquete de herramientas *open source* para el dibujo de grafos.
  Los programas de diseño de GraphViz reciben descripciones de grafos en un lenguaje textual simple (referenciado como DOT) y construye diagramas en formatos útiles, tales como: imágenes y SVG para páginas webs, PDF o Poscript para inclusión en otros documentos, o mostrar en un navegador gráfico interactivo [11].
- JUNG (Java Universal Network/Graph Framework): Es una librería de software que provee un lenguaje común y extensible para modelado, análisis y visualización de datos que pueden ser expresados como un grafo o red.
  La arquitectura de Jung soporta una variedad de representaciones de entidades y sus relaciones, tales como grafos dirigidos y no dirigidos, grafos multimodales, grafos con bordes paralelos e hipergrafos. JUNG también provee un framework de visualización que hace más fácil la construcción de herramientas para la exploración interactiva de datos. Los usuarios pueden usar uno de los algoritmos de diseño provistos para construir sus aplicaciones o usar dicho framework para crear sus propios diseños [12].
- Prefuse: Es conjunto de herramientas de software para la creación de visualizaciones de datos interactivas. El conjunto original de herramientas Prefuse, provee un framework de visualización para el lenguaje de programación Java. Prefuse soporta un conjunto poderoso de características para el modelado de datos, visualización e interacción. Dicha librería provee estructuras de datos optimizadas para tablas, grafos y árboles, técnicas de diseño y codificación visual, soporte para animación, consultas dinámicas, búsqueda integrada y conectividad a base de datos. Prefuse está escrito en Java, usando la librería gráfica Java 2D, y está fácilmente integrada en aplicaciones Java Swing o applets web [18].

Cada una de estas librerías posee distintas características y atributos que las hacen diferentes entre sí, por lo cual la elección de una librería antes que otra es dependiente de la aplicación en donde se va a emplear y las distintas características de la misma. Para poder seleccionar la mejor librería para un caso de estudio específico, es necesario establecer un ranking entre las mismas. Teniendo en cuenta esta observación, se escogió el método QEM (Quality Evaluation Method) debido a que dicho método utiliza para su funcionamiento un conjunto de criterios definidos por el especialista en visualización de software

4. Método QEM

En esta sección se describe un *Método de Evaluación de Calidad* (QEM-**Q**uality **E**valuation **M**ethod[1]) [16,15] el cual será adaptado para evaluar Librerías de Visualización. El objetivo de esta tarea es introducir al lector en el método y el proceso de ranking utilizado por el mismo logrando de esta manera que el artículo quede autocontenido.

[1] QEM fue concebido y utilizado para evaluar sitios web.

4.1. Pasos Requeridos para la Aplicación de QEM

Basicamente QEM consta de los siguientes pasos:

- **Seleccionar un conjunto de herramientas competitivas para evaluar o comparar:** esta tarea, en el contexto de las librerías de visualización, consiste en la selección de las librerías que provean mayores facilidades para la construcción de vistas de software.
- **Especificar los objetivos y el punto de vista del usuario**: en este paso los evaluadores definen los objetivos y alcance de la evaluación. El propósito principal es evaluar un conjunto de características, las cuales serán definidas en las siguientes secciones, que las librerías de visualización deben poseer para cumplir con su objetivo eficientemente. Los usuarios que utilizarán el software de estudio serán expertos en Ingeniería de Software. Por simplicidad en la aplicación del método no se considerarán diferentes niveles académicos de usuarios, lo que si se requiere es que los mismos posean un conocimiento razonable en programación.
- **Definir las características y atributos de calidad y el árbol de requerimientos de atributos**: en esta etapa los evaluadores definen las características de calidad y atributos (de las librerías de visualización) que deberán ser categorizados en un árbol de requerimientos para su posterior uso en el proceso de evaluación. Para cada una de las características se especifican, si es necesario, subcaracterísticas y atributos que se pueden medir por medio de variables cuyos valores permiten establecer un grado de preferencia. Básicamente, el trabajo realizado en esta sección se basa en la definición de características de calidad de las librerías de visualización.
- **Definir una función de criterio para cada atributo y aplicar una estrategia para medir los atributos**: para cada variable de medición es necesario la definición de un rango aceptable de valores y una función de criterio elemental. Dicha función mapea un valor en un dominio específico en un valor numérico que indica un grado de preferencia, el cual varía desde 0 % (totalmente insatisfactorio) hasta 100 % (totalmente satisfactorio).
- **Calcular las preferencias elementales para producir la preferencia de calidad de las librerías de visualización**: este trabajo se lleva a cabo por medio de funciones que permiten obtener un indicador de preferencia de los sistemas en evaluación. Para cada característica, subcaracterística o atributo se producirá, a través de un proceso de agregación, un valor de preferencia que se utilizará para producir una preferencia de calidad global. Este último valor representa el grado de satisfacción de todos los requerimientos utilizados en la evaluación.
- **Analizar, verificar y comparar los resultados parciales y globales**: en este paso los evaluadores analizan y comparan los resultados (elementales, parciales y totales) obtenidos durante el proceso de evaluación

4.2. Definición de los Criterios Elementales

Cada característica, subcaracterística o atributo se asocia con una variable de preferencia que tomará valores de acuerdo a la función de criterio elemen-

tal que debe ser definida por los evaluadores. El resultado de la aplicación de esta función representará una preferencia de calidad elemental. Dicha preferencia se puede clasificar en rangos de satisfacción, por ejemplo: [0 %,40 %] no satisfactorio; (40 %,60 %) satisfactorio y [60 %,100 %] totalmente satisfactorio

4.3. Cálculo de las Preferencias de Calidad Elementales

En esta fase del proceso de evaluación se recolectan los datos necesarios para la aplicación de las funciones de preferencia elemental. Esos valores forman la base para el cálculo de los valores de las variables de más alto nivel. Es importante notar que la recolección de datos se puede realizar en forma: *Automática*; *Semi-automática* o *Manual*. Para el caso de las librerías de visualización la recolección de datos se realizaría en forma manual debido a la carencia de herramientas que puedan automatizar este paso

4.4. Cálculo de las Preferencias Globales

En este paso, se define y prepara el proceso de evaluación que permitirá obtener un indicador de la competitividad del sistema. Esta tarea se realiza a través del cálculo de una preferencia global que indica el grado de satisfacción del programador producido por la librería de visualización. Generalmente, para alcanzar este objetivo se utiliza una aproximación similar a la empleada por el método LSP, el lector interesado en conocer el funcionamiento de LSP puede ver [9].
La temática abordada por las secciones siguientes está relacionada con la elaboración de los criterios que se utilizarán para la construcción de un *Árbol de Requerimientos* utilizado por QEM en el proceso de evaluación

5. Resultado: un Árbol de Requerimientos

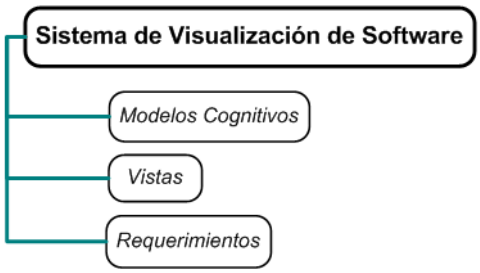

Figura 3. Árbol de criterios generales

Los resultados obtenidos hasta el momento, relacionados con los datos necesarios para aplicar QEM a las librerías de visualización, constan de un cojunto de criterios que caracterizan dichas librerías. Los mismos están clasificados en criterios generales, que a su vez pueden contener subcriterios más específicos que permiten cuantificar el grado de satisfacción de los criterios más generales.

A continuación se describen la desagregación de los criterios más generales que se ha obtenido hasta el momento. La figura 3 muestra el árbol general (en la sección B del apéndice, la figura 11 presenta una imágen del árbol completamente desagregado).

5.1. Modelos Cognitivos

Lo componen un conjunto de criterios que cubren los aspectos más importantes relacionados directamente con los procesos mentales usados por el ingeniero de software cuando intenta comprender un sistema. En la figura 4 se muestra la desagregación de este criterio general.

Como se puede observar en la figura 4, este criterio se desagrega en 5 subcriterios más específicos. Los mismos se describen a continuación:

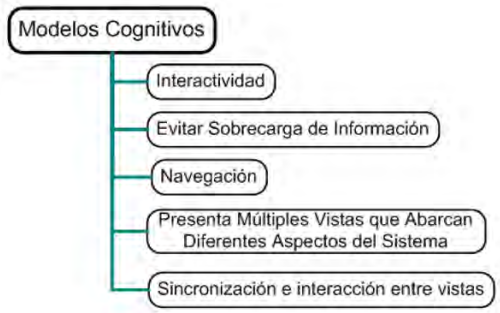

Figura 4. Desagregación de Modelos Cognitivos

- Interactividad: Representa la posibilidad de interacción que provee el sistema para con el usuario. Esta caracterizado por otros criterios presentados, como las Capacidades de Zoom, Capacidades de Navegación, Capacidades de Personalización, etc.
- Evita Sobrecarga de Información: La sobrecarga de información se produce cuando la visualización se ve perjudicada debido a la cantidad de información que la misma presenta. Es decir, la visualización "pierde" su principal objetivo: Facilitar el entendimiento del sistema al usuario. Para estos casos el sistema debería poseer aptitudes para reducir la información mostrada. Un ejemplo específico son las capacidades de diseño automático que posean mecanismos de abstracción.

- Navegación: Incluye los mecanismos para navegar entre el código fuente y las diferentes vistas gráficas de la estructura de software del programa analizado. El sistema puede proveer Navegación Direccional o Arbitraria.
- Presenta Múltiples Vistas que Abarcan Diferentes Aspectos del Sistema: Si el sistema provee esta característica, el mismo da soporte a la construcción de diferentes modelos mentales, aspecto fundamental para la comprensión del programa en cuestión.
- Sincronización e Interacción entre Vistas: El usuario a menudo intercambia de un modelo cognitivo a otro durante la comprensión, por lo tanto es importante que el sistema brinde soporte para dicho cambio. La sincronización e interacción entre las vistas ayuda al usuario en el proceso de cambio de modelo cognitivo. Por ejemplo, en algunos sistemas, muchas de las vistas que estos presentan, se "conectan visualmente" resaltando las instancias de un mismo objeto en cada visualización. Muchos sistemas también dan soporte a este criterio actualizando todas las vistas cuando una es alterada de alguna manera

5.2. Vistas

Cubre los aspectos más importantes relacionados con las vistas que se pueden generar con las librerías de visualización. La figura 5 presenta la desagregación de este criterio general. El mismo se subdivide en dos categorías descriptas en las subsecciones siguientes.

Calidad de los Artefactos Esta compuesto por los criterios que caracterizan la calidad de los grupos de artefactos que se pueden generar con las librerías de visualización, ya sea tomando en cuenta el artefacto individualmente como también las carácteristicas relacionadas a conjutos de estos. Este criterio se subdivide en dos categorías:

- Atómicos: Describe las características individuales vinculadas a los artefactos. Es decir, lo componen los criterios que caracterizan a cada artefacto visualizado. Se subdivide en dos categorías:
  - Diseño: Conjunto de criterios que caracterizan el diseño de cada artefacto. Estos son: i) Utiliza Colores, ii) Utiliza Texturas. El uso de colores y texturas se ha tornado esencial en las visualizaciones a medida que los sistemas han crecido en tamaño. Además de la "mejora visual" que estos aspectos le dan al artefacto, también es importante los diferentes significados que pueden otorgar al mismo. Por ejemplo, en algunas vistas, ciertos artefactos pueden visualizarse usando una textura transparente denotando inactividad (por ej. en vistas dinámicas).
  - Operaciones: Conjunto de criterios que caracterizan las operaciones que se pueden realizar sobre/por medio de los artefactos. Estos son:
    – Mapear Artefactos entre Vistas: Permite al usuario navegar entre diferentes vistas "observando" el mismo artefacto. Dicha característica es

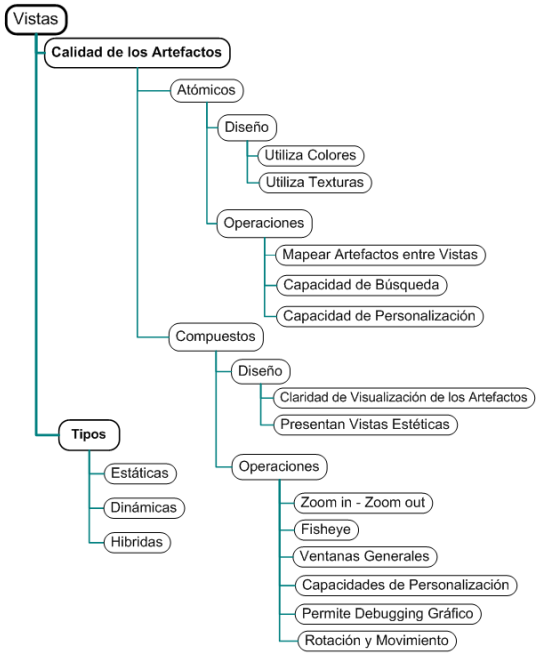

Figura 5. Desagregación de Vistas

importante debido a que permite al usuario mantener el foco de interés y de esta manera reducir overhead cognitivo. Un claro ejemplo es poder dirigirse del sector de código que conforma una determinada función a su respectivo nodo en el grafo de funciones y viceversa.

– Capacidades de Búsqueda: Permite al usuario realizar una búsqueda de algún artefacto visual en las vistas o textual en el caso del código fuente. Es importante debido a que, en algunos casos, se dificulta la interacción con los objetos que son de interés para el usuario. Generalmente, cuando el sistema es de un tamaño considerable.

– Capacidades de Personalización: Permite al usuario poder modificar algunos aspectos de los artefactos de acuerdo a su necesidad. Dichos aspectos pueden ser el color, la textura, la ubicación, etc. Esta funcionalidad hace el sistema más flexible para el usuario, de manera tal que puede realizar ciertos cambios al mismo que mejoren su comprensión.

- Compuestos: Describe las características generales vinculadas a conjuntos de artefactos o vistas. Es decir, lo componen los criterios que caracterizan a la visualización en sí. De la misma manera que Atómicos, este crtierio se subdivide en dos:
  - Diseño: Conjunto de criterios que caracterizan el diseño de las visualizaciones. Estos son:

– Claridad de Visualización de los Artefactos: Este criterio abarca diferentes funcionalidades del sistema. Una de ellas es la capacidad de diseño automático que posee el sistema para las vistas. Otra es la capacidad de abstracción de la información. Por ejemplo, un grafo de llamadas a funciones es más claro si no existen arcos cruzados. La manera de que no existan, es que el algoritmo de diseño no los genere. Lo mismo sucede si el grafo posee demasiados nodos. La manera de que se reduzca la cantidad, es que algún mecanismo de abstracción pueda agruparlos en nodos más generales. Es claro que un diseño pobre de las vistas, induce overhead cognitivo extra para el usuario.
– Presentan Vistas Estéticas: Análogamente, como en los artefactos atómicos, es importante que las diferentes vistas sean estéticas. Es decir, no sólo se requiere vistas que abarquen diferentes rasgos del sistema, sino también que la mismas presenten aspectos estéticos, los cuales facilitan al usuario la comprensión del programa.

- Operaciones: Conjunto de criterios que caracterizan las operaciones que se pueden realizar sobre/por medio de las vistas. Estos son:
– Zoom in-Zoom out y Fisheye: Básicamente, los dos describen capacidades de zoom. El primero hace referencia al zoom "común". El segundo hace referencia a una clase especial de zoom donde se mantiene el contexto, lo cual disminuye el overhead cognitivo.
– Ventanas Generales: Permite al usuario lograr un panorama más amplio de algunos aspectos del sistema al proveer ciertas ventanas que contienen visualizaciones más generales con información reducida. Las mismas se pueden presentar en conjunto con algunas vistas.
– Capacidades de Personalización: Como en el caso de los artefactos, permiten al usuario redefinir algunos aspectos de las visualizaciones de acuerdo a su necesidad. Por ej, se podría alterar el diseño base de alguna visualización con el fin de mejorar el aspecto visual de la misma. Esta funcionalidad hace el sistema más flexible para el usuario, de manera tal que pueda realizar ciertos cambios al mismo que mejoren su comprensión.
– Permite el Debugging Gráfico: Mantiene la idea del debugging ya conocido, sólo que se aplica a la/las vista/s. Es decir, a medida que se ejecuta el programa, la visualización se va "construyendo". Es útil debido a que nos ofrece las mismas capacidades que el clásico debugging, pero en conjunto con la visualización. Se debe tener en cuenta muchos aspectos para lograr esta capacidad, debido a que se ve afectado por: Los mecanismos de abstracción, las capacidades de diseño automático, las características estéticas de la vista, etc.
– Rotación y Movimiento: Se aplica a ciertas visualizaciones, como las 3D o las que son demasiado grandes. Es importante que el sistema posea dichas funcionalidades para mejorar la interacción usuario-sistema. Muchas veces, en las vistas 3D, la perspectiva en la que esta diseñada la vista no es la más apropiada, por lo tanto es necesario que el usuario pueda rotar o mover el conjunto de artefactos para lograr una mejor percepción

Tipos Está compuesto por los criterios que caracterizan los tipos de vistas que se pueden generar. Las vistas pueden ser:

- Estáticas: Visualizaciones que abarcan los aspectos estáticos del programa. Un sistema de visualización incluye este aspecto si posee vistas como: program slicing (estático), visualizaciones en 3D (codecity), grafos dirigidos (dependencia de funciones), etc.
- Dinámicas: Visualizaciones que abarcan los aspectos dinámicos del programa. Un sistema de visualización incluye este aspecto si posee vistas como: Program slicing (dinámico), visualizaciones en 3D dinámicas (Dynamic Feature Trace View), vistas animadas (code-swarm), etc.
- Híbridas: Visualizaciones que abarcan los aspectos estáticos y dinámicos del programa.

5.3. Requerimientos

Lo componen los criterios que abarcan los requisitos básicos de hardware y software para las librerías de visualización. En la figura 6 se muestra el subárbol correspondiente.

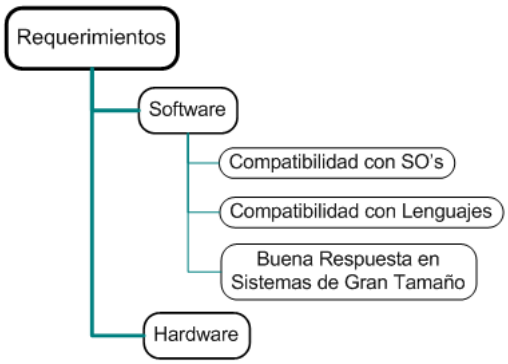

Figura 6. Desagregación de Requerimientos

- Software: Está compuesto por los criterios que describen la compatibilidad y requerimientos de software de las librerías de visualización de software. Estos son:
  – Compatibilidad con Sistemas Operativos y Lenguajes de Programación: Comprenden la compatibilidad del sistema de visualización con respecto a los sistemas operativos y los lenguajes que soporta.
  – Desempeño en Sistemas de Gran Tamaño: Muchos sistemas de visualización comprenden la mayoría de los criterios antes mencionados (Vistas estéticas, capacidades de diseño automático, mecanismos de abstracción, buena

compatibilidad, etc.), pero su rendimiento decae en tiempo y espacio cuando los sistemas a visualizar son grandes. Por lo tanto, este criterio es sumamente importante a la hora de la implementación del sistema de visualización.

- Hardware: Los requisitos de HW incluyen: Memoria RAM, espacio en disco duro, tarjeta de video, ancho de banda mínimo, etc. Es importante considerar dichos requisitos, ya que muchas veces el entorno en donde se utilizará el sistema podría no ser ideal; más teniendo en cuenta ciertas vistas que poseen un desenvolvimiento gráfico considerable como las que incluyen animaciones u otras que requieren de mucha memoria a la hora de visualizar sistemas grandes como las 3D

6. Conclusión y Trabajo Futuro

En este artículo se mencionó que el principal desafío en el contexto de la Comprensión de Programas consiste en relacionar el Dominio del Problema con el Dominio del Programa. No obstante, para comprender un programa, es necesario mostrar adecuadamente dicha relación. Para realizar la tarea mencionada previamente, es necesario introducirse en los conceptos de Visualización de Software. Esta disciplina estudia la mejor forma de crear vistas de un sistema. La creación de dichas vistas, se lleva a cabo utilizando librerías de visualización de software. Esto lleva al programador a tener que seleccionar la librería de visualización más conveniente para cada caso en particular, dependiendo de la vista que se quiere construir. La selección de la mejor librería de visualización para la construcción de una vista determinada, es una tarea compleja que conduce al programador a gastar demasiado tiempo en ella. Lo antes mencionado, hace que el programador centre su antención en la selección de la librería y no en la construcción de la vista en sí. Para solucionar ese problema, se seleccionó un mecanismo que permite establecer un ranking de librerías de visualización de forma semi automática. El mecanismo se basa: i) En la definición de los criterios que una librería de visualización debe poseer para simplificar la creación de vistas de software útiles para comprender programas y ii) En la aplicación de un proceso de agregación.

En este artículo se propuso una solución al ítem i) es decir se elaboró un árbol de requerimientos, en donde se encuentran plasmadas las principales características/atributos que una librería debe poseer para facilitar la construcción de vistas de software orientadas a la comprensión. Es importante notar que para realizar esta tarea, se debió hacer un profundo estudio del estado del arte de la temática subyacente abordada en el artículo. Dicho estudio, permitió comprobar que: i) Existen criterios para construir sistemas de visualización, los cuales pueden ser usados para seleccionar la librería de visualización de software más apropiada, ii) Los criterios observados en la literatura no se encuentran organizados y tampoco se encuentran bien desagregados, iii) No se pudo encontrar un método de selección de librerías de visualización de software.

Por lo expuesto en el párrafo precedente se puede mencionar que el trabajo presentado en este artículo es una contribución importante para la Comprensión de Programas y la Visualización de Software.

Como trabajo futuro, se pretende: i) Adapatar los criterios presentados en este artículo a un estandard internacional, ii) Construir una herramienta que implemente QEM para librerías de visualización de software

APÉNCIDE

En este apéndice se describen algunos conceptos necesarios para comprender ciertos aspectos de los sistemas de visualización. También se describe el problema existente con las taxonomías de sistemas de visualización de software. Por otro lado se prensentan imágenes de ejemplos de vistas y una imágen del árbol de requerimientos completo.

A. Sistemas de Visualización de Software

Un Sistema de Visualización de Software (SVS) es una aplicación cuyo objetivo es proveer diferentes perspectivas visuales del software. Los SVS están normalmente compuestos por los siguientes subsistemas: *Ambiente de Visualización*, *Administrador de la Información* y *Extracción de la Información*.

El primero contiene diferentes estrategias visuales que muestran la información. El segundo está relacionado con el acceso eficiente a los datos requeridos para la generación de las visualizaciones. El tercero es usualmente una componente compartida por otra clase de sistemas tales como las Herramientas de Comprensión de Programas (HCP) y su función consiste en la recuperación de la información desde el software.

En este artículo, el interés se centra en los *Sistemas de Visualización de Software Orientados a la CP* (SVS-PC) esta clase de aplicaciones son una subclase de los SVS que normalmente se integra con HCP. La principal diferencia entre SVS y los SVS-PC es que los primeros solamente se preocupan por la visualización de software, en otras palabras por el Dominio del Programa, mientras que los segundos consideran la visualización de los Dominios del Problema y Programa y la relación existente entre ambos. Esta distinción es substancial porque fuerza al diseñador a: i) Crear representaciones para otros objetos además del software y ii) Pensar estrategias de visualización de la relación entre la especificación del problema y la representación del software. El lector puede percibir que la consideración expuesta previamente se basa en la concepción de CP descripta en la sección 1 del artículo.

En este apartado se describen los *Ambientes de Visualización*

A.1. Ambiente de Visualización

Un *Ambiente de Visualizaicón* (AV) es una componente de los SVS o SVS-PC que contiene los elementos necesarios para simplificar la manipulación de los objetos que forman parte de la visualización. Las principales funcionalidades y componentes de AV se refieren a: *AV en sí mismo* y las *Vistas*. Las próximas subsecciones describen cada uno de ellos

Caracterización de los Ambientes de Visualización

Las principales partes de un AV [21] son:

Encabezado: Muestra el nombre de la aplicación, la identificación de la componente de software, etc.

Título: Presenta el tema de estudio.

Objetivo: Explica la finalidad de la visualización.

Descripción de Artefacto: Describe las caracterísitcas principales de los artefactos usados en la visualización.

Área Historia: Almacena la historia de las operaciones de usuario.

Área de Comando: Provee un acceso fácil y eficiente a las operaciones de los artefactos usados en la visualización.

Área de Visualización: Se refiere a la parte de la interfaz gráfica donde se ubican los artefactos.

Área de Intercambio: Algunas veces un artefacto puede mostrar muchos atributos, sin embargo, observarlos en su totalidad en una simple visualización no es buena idea porque promueve la desorientación. Una aproximación interesante para solucionar este inconveniente consiste en proporcionar diferentes visualizaciones del mismo artefacto pero con un número limitado de atributos y proveer un mecanismo de navegación entre esas visualizaciones. Este mecanismo se puede implementar usando un área de intercambio que permita un acceso fácil y rápido a las diferentes perspectivas del mismo objeto. Es importante destacar que el Área de Intercambio también facilita el estudio del software cuando el usuario trabaja con dos o más artefactos al mismo tiempo.

Área de Relación: Generalmente, los artefactos de visualización ocupan más espacio que el proporcionado por la pantalla. En esas situaciones, es posible tener una vista parcial de la representación del sistema. Este problema produce desorientación porque se pierde el contexto. Para paliar esta dificultad, es útil proveer un Área de Mapeo que contenga una vista global del artefacto. Esta área se puede mostrar como una pequeña ventana en alguna esquina del objeto contenedor.

Layout and Look: Considera atributos tales como: *Color*, *Tamaño*, *Posición*, etc. Esas cualidades son muy dependientes de la clase de objetos usados para mostrar atrefactos; sin embargo es importante considerar su utilidad para representar las características del sistema.

Anotación: Se usa para hacer comentarios acerca de alguna de las características del/los objeto/s observado/s en la visualización.

Base de Conocimiento: Se emplea para mostrar los conceptos recientemente usados, o para extraer la terminología relacionada con un trabajo específico. En términos cognitivos, asiste al proceso de recuperación de conceptos desde la estructura mental del programador.

Clase de Proceso de Asimilación: Posibilita conocer la estrategia de aprendizaje usada y provee funciones que simplifican la operación del proceso de asimilación corriente. Por ejemplo, si la estrategia de aprendizaje es Top-down entonces la herramienta debe proporcionar funciones de navegación desde los aspectos más abstractos hasta los más específicos.

Área de Leyenda: Facilita la identificación de los atributos de los artefactos más importantes empleados en la visualización.

Pie: Es beneficioso exhibir alguna información en esta parte. Por ejemplo: *Cantidad de vistas abiertas*, *Coordenadas del ratón*, *Información de estado en general*, etc.

Para finalizar el apartado, es importante notar que:

- La información provista por las componentes del AV no deben sobrecargar la percepción del usuario.
- Si el número de componentes del AV es muy grande entonces el diseñador debe eliminar o filtrar alguna de ellas.
- Además de las funcionalidades descriptas previamente, los AV tienen funciones similares a las de las clásicas interfaces de usuario (Abrir, guardar, guardar como, redimensionar, etc.), porque tienen como objetivo la operación de SVS-PC en sí mismo y no la manipulación de artefactos.

Caracterización de las Vistas Las *Vistas* son una componente fundamental de los AV. Una *Vista* es una forma de visualizar las componentes del sistema y sus relaciones que permite mostrar alguno de sus aspectos.

Las vistas pueden ser *Estáticas*, *Dinámicas* o *Híbridas*.

Una vista estática muestra la información sin efectos de animación, en otras palabras las características estáticas del sistema o problema; un ejemplo de este tipo de vistas es el Grafo de Funciones. Las vistas dinámicas visualizan el flujo de ejecución del programa o animaciones del Dominio del Problema. El árbol de ejecución de funciones y el proceso usado para construirlo forman parte de esta clase de perspectivas. Las vistas híbridas son una combinación de las vistas estáticas y dinámicas. Esta clase de vistas son interesantes porque la combinación adecuada de la información posibilita la identificación de las partes del sistema usadas para un propósito específico. Por ejemplo, conocer cuales funciones del Grafo de Funciones se utilizaron para un escenario de ejecución particular simplifica la inspección del sistema.

Otra forma de conceptualizar las vistas es considerar su contenido. En este caso, es posible obtener vistas de: *Datos*, *Código* o *Funcionalidades*, las cuales se pueden concretizar con versiones *estáticas*, *dinámicas* o *híbridas*. El diseñador de la visualización decide el método de combinación de acuerdo a las características del objeto de estudio.

Es importante notar que la elaboración de vistas implica responder preguntas tales como: *¿Por qué se necesita la visuazliación? ¿Quién usará la visualización? ¿Qué representa la fuente de datos? ¿Cómo se representan los datos? ¿Dónde se representa la visualización?*

Los temas tratados en este apartado sirven como complemento de lo desarrollado en la sección 2.

A.2. Incompletitud de las Taxonomías Corrientes

En la sección 2, se describieron las principales taxonomías de los Sistemas de Visualización de Software (SVS). En términos generales, esas clasificaciones describen las características principales de los SVS. Sin embargo, como se menciona

en la sección 2, las taxonomías dejan de lado aspectos de CP muy importantes, como por ejemplo: *La visualización del Dominio del Problema y la relación existente con el Dominio del Programa.*

El proceso de Comprensión de Programas se basa sobre Modelos Cognitivos y cuando el programador quiere comprender un programa dos dominios del conocimiento son escenciales: el Dominio del Programa y el Dominio del Problema.

El *Dominio del Programa* analiza pasos tecnológicos como el lenguaje de programación (sentencias, funciones, módulos) y como se ejecuta el programa para producir una salida; mientras que el *Dominio del Problema* estudia los efectos de la ejecución del programa (El resultado final producido y el impacto a nivel del problema que se resolvió).

La principal desventaja con las taxonomías corrientes es que *las mismas no están basadas en una concepción de Comprensión de Programas* porque sólo consideran el Dominio del Programa. Por esta razón, se pierden muchos aspectos sustanciales que determinan la calidad del SVS.

En la sección 1 se declaró que:

La CP se simplifica si se representan e interconectan los Dominios del Problema y Programa.

Esta declaración es el punto de partida para afirmar que las clasificaciones actuales no tienen en cuenta algunas características escenciales que los SVS-PC deben poseer.

Las investigaciones en VS realizadas en [2] revelan que las taxonomías de SVS describen muy bien el Dominio del Programa, pero no sucede lo mismo con el Dominio del Problema, ni la relación existente entre ellos. Preguntas tales como: *¿Cómo se puede caracterizar el Dominio del Problema? ¿Cuál es la mejor forma de describir la relación entre los Dominios del Problema y Programa? ¿Por qué los factores cognitivos no se mencionan en esas taxonomías? ¿Las características de los SVS están bien organizadas?*, etc. motivan la elaboración de una clasificación de los SVS cuya finalidad es caracterizar apropiadamente a los SVS orientados a la Comprensión de Programas

B. Ejemplos de Vistas y Árbol de Criterios Completo

En este apartado se muestran imágenes de algunos ejemplos de vistas particulares y la imágen del árbol de requerimiento completo.

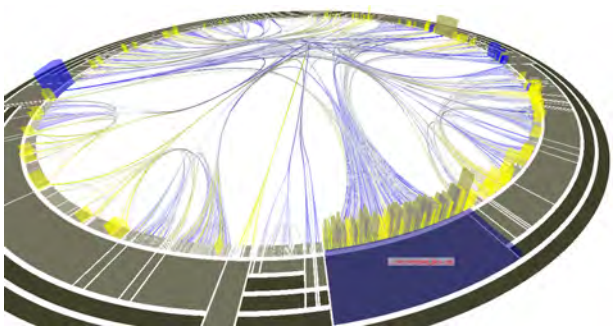

Figura 7. Static Circular Bundle View de un aspecto de Notepad++

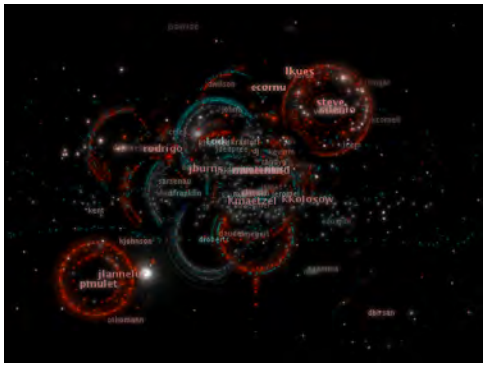

Figura 8. Captura de una animación Code-Swarm de Eclipse

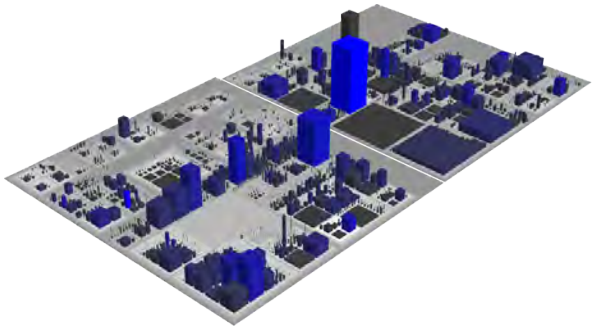

Figura 9. CodeCity de Java Development Kit (JDK) v1.5

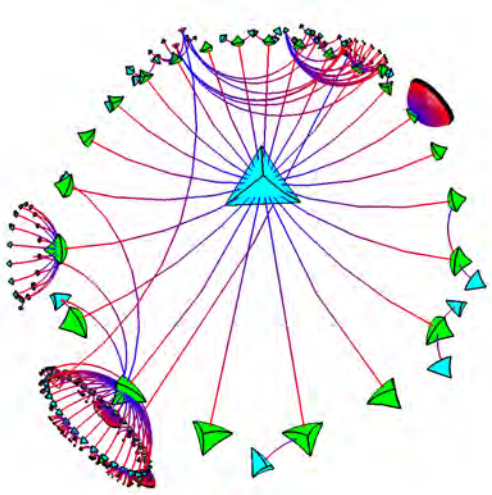

Figura 10. 3D Hyperbolic Tree del sistema de archivo de la Web(1995)

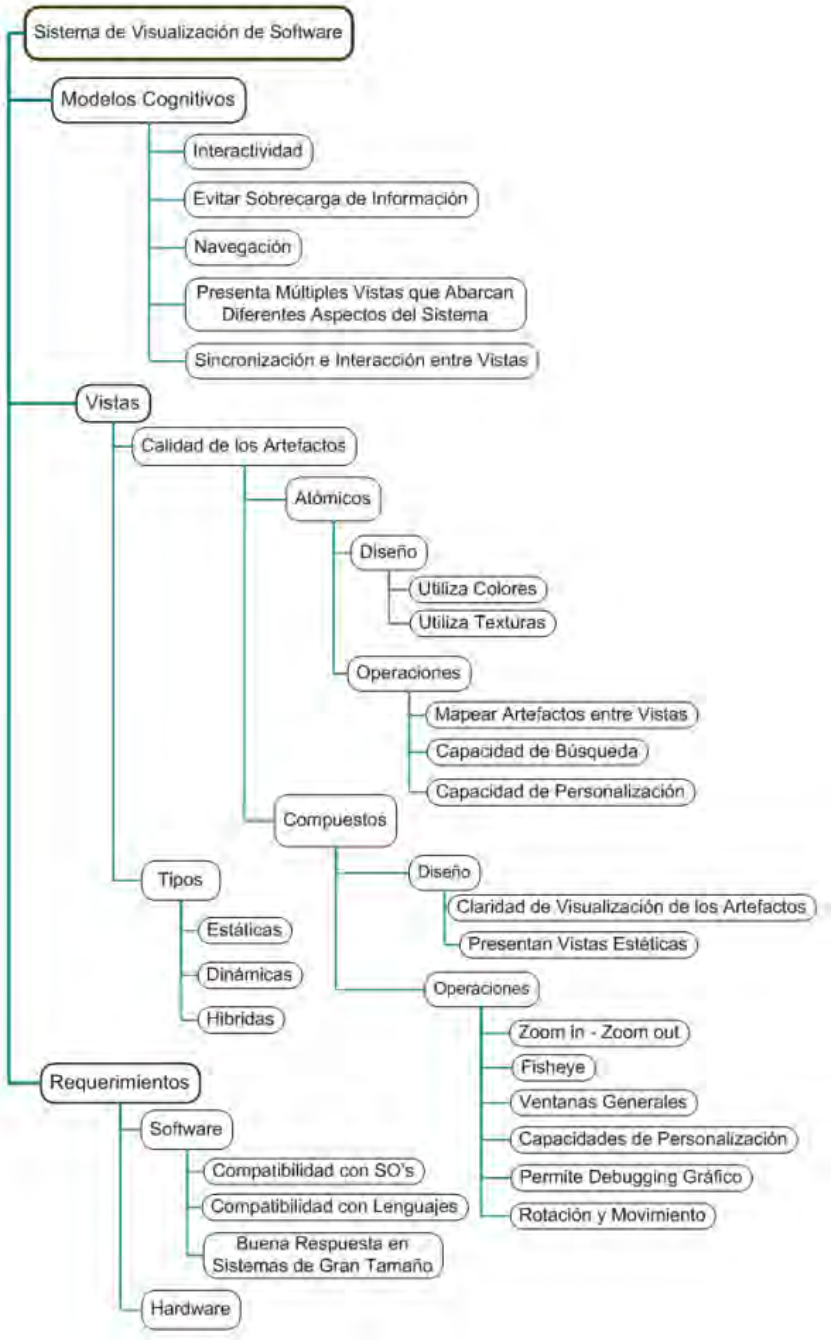

Figura 11. Árbol de Requerimientos Completo

Referencias

1. T. Ball and SG Eick. Software visualization in the large. *Computer*, 29(4):33–43, 1996.
2. M. Beron, P. Henriques, and R. Uzal. Program Inspection to interconnect Behavioral and Operational Views for Program Comprehension. *Ph.D Thesis Dissertation at University of Minho. Braga. Portugal*, 2010.
3. M. Beron, P. Henriques, M. Varanda, and R. Uzal. A Language Processing Tool for Program Comprehension. *Congreso Argentino de Ciencias de la Computacion (CACIC06)*, 2006.
4. M. M. Beron, D. Cruz, M. J. Varanda Pereira, P. R. Henriques, and R. Uzal. Evaluation criteria of software visualization system used for program comprehension. *3a Conferencia Nacional em Interacção Pessoa-Máquina*, 03:285, 2008.
5. K. Bertels, P. Vanneste, and C. De Backer. A cognitive approach to program understanding. *Reverse Engineering, 1993., Proceedings of Working Conference on Program Comprehension*, pages 1–7, 1993.
6. Stuart K. Card, Jock D. Mackinlay, and Ben Shneiderman, editors. *Readings in information visualization: using vision to think*. Morgan Kaufmann Publishers Inc., 1999.
7. B. Cornelissen and Leon Moonen. Visualizing Similarities in Execution Traces. *Program Comprehension through Dynamic Analysis*, 1:6–10, 2007.
8. D. Cruz, P. Henriques, and J. Pinto. Code analysis: Past and present. *Proceeding of the Third International Workshop on Foundations and Techniques for Open Source Software Certification*, 00:1–10, 2009.
9. J. J. Dujmovic. A Method for Evaluation and Selection of Complex Hardware and Software Systems. *The 22nd Conference for the Resource Management and Performance Evaluation of Enterprise CS.*, 01:368–278, 1996.
10. T. Eisenbarth, R. Koschke, and D. Simon. Aiding program comprehension by static and dynamic feature analysis. In *ICSM'01: Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, page 602, Washington, DC, USA, 2001.
11. GraphViz-Team. http://www.graphviz.org/, 2011.
12. JUNG-Team. http://jung.sourceforge.net/, 2011.
13. K. Mens, T. Mens, and M. Wermelinger. Supporting software evolution with intentional software views. *Proceedings of the International Workshop on Principles of Software Evolution*, pages 138–142, 2002.
14. B. Myers. Taxonomies of Visual Programming and Program Visualization. *Journal of Visual Languages and Computing*, 1(1):97–123, 1990.
15. L. Olsina, D. Godoy, G. Lafuente, and G. Rossi. Assessing the Quality of Academic Web Sites: a Case Study. *New Review Hypermedia Journal*, 05:81–103, 1999.
16. L. Olsina and G. Rossi. Measuring Web Application Quality with WebQEM. *IEEE MultiMedia, 2002*, 09(4):20–29, 2002.
17. M. Petrenko, V. Rajlich, and Vanciu R. Partial Domain Comprehension in Software Evolution and Maintenance. *International Conference on Program Comprehension (ICPC08)*.
18. Prefuse-Team. http://prefuse.org/, 2011.
19. B. A. Price, I. S. Small, and R. M. Baecker. A taxonomy of software visualization. *System Sciences, 1992. Proceedings of the Twenty-Fifth Hawaii International Conference on*, 2, 1992.

20. G. C. Roman and K. C. Cox. A taxonomy of program visualization systems. *Computer*, 26(12):11–24, 1993.
21. J. Stasko, J. Domingue, M. Brown, and B. Price. *Software Visualization: Programming as a Multimedia Experience*. The MIT Press, 1998.
22. M. A. Storey. Theories, methods and tools in program comprehension: past, present and future. *Proceedings of the 13th International Workshop on Program Comprehension*, pages 181–191, 2005.
23. M. A. Storey, F. D. Fracchia, and H. A. Muller. Cognitive design elements to support the construction of a mentalmodel during software visualization. *Program Comprehension, 1997. IWPC'97. Proceedings., Fifth Iternational Workshop on*, pages 17–28, 1997.
24. S. Tilley and S. Huang. On selecting software visualization tools for program understanding in an industrial context. *iwpc*, 00:285, 2002.
25. A. Von Mayrhauser and A. M. Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, 1995.