

Máster Universitario en Sistemas Inteligentes
Trabajo de Fin de Máster
Junio, 2014

Representación Visual de Sistemas de Software: Evolución y Colaboración

Presentado por:
Antonio González Torres

Dirigido por:
Dr. Francisco J. García Peñalvo
Dr. Roberto Therón Sánchez



Departamento de Informática y Automática
Universidad de Salamanca
(<http://informatica.usal.es>)

Información del autor:

Antonio González Torres
agtorres@usal.es

Este documento puede ser libremente distribuido.

(c) 2014 Departamento de Informática y Automática - Universidad de Salamanca.

Índice

Índice de figuras	III
Índice de tablas	V
1. Introducción	1
1.1. Objetivo	2
1.2. Justificación	2
1.3. Organización del documento	4
2. Trabajo en equipo y colaboración	5
2.1. Desarrollo global de software	5
2.2. Trabajo en equipo	6
2.2.1. Comunicación, coordinación, control y cognición	7
2.2.2. Conciencia situacional del equipo	12
2.2.3. Conciencia situacional distribuida	14
2.3. Consideraciones para diseñar espacios para la conciencia situacional	16
3. Visualización de software	20
3.1. Revisión general de técnicas de visualización	20
3.2. Visualización de la arquitectura de los sistemas de software	23
3.2.1. Visualización en 3D	25
3.2.2. Metáforas de ciudad	26
3.2.3. Diseños basados en matrices	27
3.2.4. Otros tipos de visualizaciones	29
3.3. Visualización de la evolución de la arquitectura de los sistemas de software	30
3.3.1. Metáforas de ciudad	30
3.3.2. Diseños basados en matrices	33
3.3.3. Animación	34
3.3.4. Cartografía de software	35
3.4. Actividades de los programadores y colaboración	38
3.4.1. Trabajo en equipo	38
3.4.2. Conciencia situacional	39
3.4.3. Cooperación	44
3.4.4. Relaciones socio-técnicas	45
4. Visualización para la conciencia situacional y la colaboración	49
4.1. Framework: conciencia situacional y colaboración	49

4.2. GridMaster: Visualización de la estructura de los sistemas y colaboración	51
4.2.1. Descripción de la visualización	52
4.2.2. Estructura del proyecto	54
4.2.3. Líneas de vida, colaboración y relaciones socio-técnicas	54
4.2.4. Relaciones: herencia e implementación de interfaces	55
4.2.5. Métricas	55
4.3. Escenarios	56
4.3.1. Escenario 1	56
4.3.2. Escenario 2	57
5. Conclusiones	58
Bibliografía	59
Glosario	67

Índice de figuras

1.	Taxonomía de las técnicas de visualización para grandes cantidades de información	20
2.	Ampliación simple de una área de interés. Figura tomada de [56] . . .	21
3.	Técnica bifocal para la visualización de información. Figura tomada de [56]	22
4.	Visualización aumentada mediante la utilización de la técnica del ojo de pez. Figura tomada de [56]	23
5.	Ejemplo de la técnica polifocal	24
6.	Visualización usando una metáfora de ciudad [84, 85]. (a) Uso de niveles para representar elementos contenidos por otros elementos. (b) Representación visual de los métodos en una clase usando figuras de ladrillos. (c) Visualización de un proyecto de software completo. . .	27
7.	Características de visualización de Lattix. (a) Correlación de dependencias entre tareas (elementos de software). (b) Característica expandible de la visualización y número de dependencias de los elementos de software contenidos en los paquetes. (c) El paquete "project.es" expandido y representa a los elementos de software que contiene y las dependencias en las cuales los elementos están involucrados. [77] . . .	28
8.	Reglas de diseño: marcas de dependencias permitidas, no permitidas y violaciones al diseño del sistema [77].	29
9.	Relaciones de dependencia con Interactive Multi-Matrix Visualization (IMMV) [8].	29
10.	Visualización de relaciones de dependencia con Parallel Node-Link (PNL) [8].	30
11.	Visualización de dos revisiones de un sistema de software [85].	31
12.	Representación visual de la evolución de un elemento de software y sus métodos [85].	31
13.	EvoStreets: Evolución de la estructura de un sistema de software [80].	32
14.	Estructura de árbol H-V para la visualización de la estructura de sistemas de software [28].	33
15.	EvoStreets: Uso de niveles en la visualización para mostrar cuando se agrega un nuevo paquete [80].	33
16.	EvoStreets: Las propiedades de los elementos de software se representan con el ancho, alto y color de los edificios [80].	34
17.	Comparación de dependencias para dos revisiones de un sistema de software [8].	35
18.	Visualización animada de la evolución de la arquitectura de un sistema de software utilizando Yarn [36].	36

19.	Proceso de construcción de un mapa de un sistema de software con <i>Software Cartography</i> [50]. (a) Colocación de elementos en el plano de acuerdo a la distancia de los términos. (b) Área de influencia de los elementos según su proximidad y tamaño medido por el número de líneas. (c) Altura de los montículos calculada utilizando como referencia el tamaño de los elementos del sistema.	37
20.	Representación de la sucesión de revisiones de un sistema utilizando <i>Software Cartography</i> [50].	37
21.	Representación de los cambios y la propiedad de los elementos de software en <i>Ownership Map</i> [27].	39
22.	Visualización de los patrones de comportamiento de los programadores usando <i>Ownership Map</i> [27].	41
23.	<i>CodeTimeline</i> : (a) Notas de del equipo de desarrollo sobre la visualización <i>Ownership Map</i> . (b) Visualización de la frecuencia de términos por versión usando nubes de palabras y notas del equipo de desarrollo. [52].	42
24.	Visualización de las actividades llevadas realizadas por los programadores [73].	43
25.	Representación de los elementos de software y cambios que ha realizada cada programador [73].	43
26.	Share: Editor de texto que muestra los trozos de código que han sido reutilizados y representam, mediante el uso de colores, quien ha realizado el aporte original [5].	45
27.	Share: Navegador de relaciones utilizando una configuración radial para mostrar las relaciones entre elementos con base en la reutilización de código fuente [5].	46
28.	Share: Navegador de relaciones simplicado para mostrar las relaciones de un elemento particular [5].	47
29.	Visualización de una red de colaboración entre programadores a partir de los elementos de software que han cambiado en común [37].	48
30.	Framework sobre el trabajo colaborativo en los procesos de Desarrollo, Mantenimiento y Evolución de Software (DMES).	50
31.	Vista principal de GridMaster.	53
32.	Estructura del sistema y contenido de los archivos. (a) Representación de la estructura del sistema. (b) Elementos de software que contiene un archivo. (c) Relaciones de herencia e implementación de interfaces.	54
33.	Relaciones de herencia e implementación de interfaces, incluyendo la expansión de un año y valores de métricas.	56
34.	Representación de métricas.	56

Indice de tablas

1. Supuestos beneficios de adoptar un enfoque de Desarrollo Global de Software (DGS). 5

1. Introducción

En el contexto actual el Desarrollo, Mantenimiento y Evolución de Software (DMES) se lleva a cabo, en su mayoría, utilizando un enfoque de Desarrollo Global de Software (DGS). A las dificultades que supone el desarrollo de software distribuido en diferentes localidades se suma que los cambios a los sistemas son frecuentes y la documentación se desactualiza en poco tiempo.

Poner al día la documentación de un sistema de software durante los procesos de DMES es un reto que puede tomar muchas horas e incluso días, lo que usualmente obliga a actualizarla de forma programada con el fin de tomar en cuenta un número suficiente de cambios. Sin embargo, por las características dinámicas de DMES, la documentación se vuelve obsoleta durante el tiempo que se requiere para actualizarla, y el equipo de trabajo requiere contar con conocimiento actualizado sobre los cambios que se han realizado al sistema para comprenderlo y poder continuar con su desarrollo o mantenimiento. A lo que se debe agregar que contar con la documentación actualizada tampoco resuelve el problema de los individuos que requieren contar con conocimiento actualizado de forma oportuna.

En un escenario ideal existen herramientas que ofrecen acceso a información sobre las actividades y cambios que realizan los programadores durante los procesos de DMES. Pero teniendo en cuenta el problema que se discutió anteriormente, se requieren mecanismos efectivos para poner a disposición de los interesados la información en el momento que se produce. De ahí que el análisis automático de los elementos asociados a los proyectos de software sean de gran utilidad para extraer hechos relevantes sobre los cambios recientes e históricos que se han efectuado al sistema. En este escenario, el uso de la visualización permite mostrar los cambios que se han realizado al sistema, pero además facilita comprender su estado actual a partir de la evolución, tomando como base un periodo de tiempo reciente o la evolución completa del sistema. De forma adicional conviene considerar que mediante el uso de la visualización es posible crear un espacio de trabajo para poner a disposición de los equipos de trabajo la información sobre las actividades que se llevan a cabo durante los procesos de DMES para tengan conocimiento sobre el estado de las cosas ¹ en torno al sistema.

Cabe tener presente que las tareas de los programadores forman parte de un ciclo que requiere la comprensión del estado actual del sistema, la programación de código fuente, la realización de cambios y la evaluación de los cambios efectuados, para volver nuevamente al punto de partida: comprender el estado actual del sistema para lograr que el sistema siga siendo susceptible de mantenimiento y evolucione. En ese ciclo los programadores hacen uso de herramientas para modificar el sistema, para que de acuerdo con los requerimientos que este busca cumplir, el resultado satisfaga las necesidades de los usuarios [69]. Entre las herramientas usadas por los programadores en las tareas de los procesos de DMES se encuentran los Entornos de Desarrollo Integrados (IDEs), compiladores, depuradores y las herramientas de Administración

¹En inglés se utiliza la palabra awareness para referirse al conocimiento de lo que sucede en un sistema o entorno.

de la Configuración de Software (SCM).

En este punto cabe puntualizar la definición de Gracanin [29] sobre la **Visualización de Software (VS)** como una disciplina que hace uso de diversas formas de imágenes para proporcionar conocimiento, facilitar la comprensión y reducir la complejidad de los sistemas de software bajo desarrollo, mantenimiento y evolución. En términos generales la visualización de sistemas de software [17] se divide en las siguientes categorías:

1. Visualización de programas (e.g. visualización de arquitecturas de software, **Evolución de Software (ES)**, métricas de software y cambios al código fuente).
2. Animación de algoritmos.
3. Programación visual.
4. Programación por ejemplos.
5. Representación visual de artefactos derivados de los requerimientos, documentación del diseño del sistema e informes de errores de los programas.
6. Visualización de la evolución de software.

La visualización de la evolución de software es una categoría especial que se encarga de representar diferentes elementos de los sistemas de software para un periodo de tiempo o la evolución completa de un sistema. Con respecto a las categorías mencionadas anteriormente, esta categoría especial tiene relación con la visualización de programas, la animación de algoritmos y la representación visual de artefactos.

1.1. Objetivo

Este trabajo de investigación se enfoca en la visualización de programas y su evolución, y tiene por objetivo facilitar la colaboración entre los miembros del equipo de desarrollo brindando información sobre las actividades que se llevan a cabo en torno a la arquitectura de los sistemas de software y su evolución, teniendo en cuenta:

1. El escenario actual del desarrollo de software en diferentes localidades geográficas (**DGS**).
2. Las necesidades de información de los **Administrador de Proyectos (APs)** y programadores sobre las actividades que realizan los miembros del equipo de trabajo (para controlar y monitorear las actividades) y la evolución de la calidad del sistema y sus elementos (medida con el uso de métricas).

1.2. Justificación

En la actualidad las empresas realizan el desarrollo y mantenimiento de software de forma distribuida, con los miembros del equipo de trabajo localizados en diferentes

áreas geográficas [22, 35, 38, 64], lo cual afecta la fluidez de la comunicación, la comprensión del estado del proyecto y contar con conocimiento oportuno sobre las actividades que son llevadas a cabo [71, 33, 66, 81].

Las distancias que involucra el DGS son geográficas, temporales y socio-culturales, y por lo tanto los retos a los que se enfrenta involucra problemas de comunicación, coordinación y control [11, 34, 15, 62]. Lo cual requiere del uso de mecanismos efectivos [11, 63, 71] como:

- * Documentación actualizada de los sistemas (requerimientos, especificación, diseño y manuales).
- * Metodologías de desarrollo apropiadas.
- * Email.
- * Teléfono.
- * Mensajería instantánea.
- * Sistemas para video-conferencias.
- * Tecnología y herramientas que apoyen el trabajo colaborativo al permitir compartir y obtener conocimiento sobre las actividades que llevan a cabo los miembros del equipo de trabajo.
- * Estrategias de formación de equipos.

En este contexto otro factor a tener en cuenta es el tipo de organización usada por equipos de trabajo que operan en entornos DGS (de forma independiente a la estructura organización usada por la compañía [61]), siendo los siguientes tipos los más comunes:

- * Equipos virtuales [40, 11].
- * Equipos coherentes y con ubicación compartida de programadores [18, 19].
- * Equipos débilmente acoplados [34, 70].
- * Desarrollo global de software con un enfoque disperso [82].

Teniendo en cuenta lo anterior existe la necesidad de definir un framework para describir la interacción entre los aspectos que intervienen durante los procesos de DMES y su relación con la necesidad de diseñar espacios de conocimiento compartido para facilitar la toma de decisiones y colaboración entre los miembros del equipo de trabajo. Asimismo, se debe tomar en cuenta que la arquitectura es el elemento central de un sistema de software y que su comprensión, y la de los elementos que la componen, es determinante durante los procesos de DMES, y la mayoría de tareas que realizan los equipos de trabajo giran en torno a esta. Por lo tanto, la visualización de la evolución de la arquitectura de los sistemas de software, así como las relaciones de herencia entre clases, implementación de interfaces, métricas, colaboración entre programadores y relaciones socio-técnicas son características deseables de una herramienta que busque apoyar la conciencia situacional y la colaboración en un entorno de DGS.

1.3. Organización del documento

Esta investigación inicia con una discusión que se desarrolla en la sección 2 sobre los elementos que intervienen durante el trabajo en equipo y la colaboración que tiene lugar en los procesos de DMES en un entorno DGS. A continuación, en la sección 3, se presenta los resultados de una revisión efectuada a algunas visualizaciones de relevancia en la visualización de las arquitecturas de los sistemas de software y su evolución. Posteriormente se define un framework que emerge como resultado de la discusión desarrollada en la sección 2 y se presenta un prototipo que se implementó en Java como un plugin de Eclipse. Finalmente se presenta las conclusiones en la sección 5.

2. Trabajo en equipo y colaboración

La realización de los procesos de DMES de forma distribuida a nivel global es un reto para los departamentos de desarrollo y la industria del software. Estos procesos son de gran complejidad y requieren del trabajo en equipo, y la colaboración de un gran número de personas separadas por las distancias geográficas, horarias y culturales que imponen los modelos DGS. Lo cual intensifica las necesidades de contar con mecanismos de comunicación, coordinación y control eficientes, pero además de medios para que los miembros de los equipos de trabajo adquieran conciencia situacional sobre las tareas que llevan a cabo y del estado de las actividades que los afecta de forma directa e indirecta.

Esta sección está dedicada a discutir sobre los factores mencionados y la relación que existe entre ellos, así como algunas consideraciones que conviene tomar en cuenta al diseñar espacios para facilitar la creación de conciencia situacional y la colaboración entre los individuos.

2.1. Desarrollo global de software

Un gran número de empresas se ha visto motivado en llevar a cabo el desarrollo de proyectos de software usando un modelo de DGS por los beneficios que suponen que van a obtener. Por lo que es aconsejable tener en cuenta el trabajo realizado por Conchúir que analizó la extensión con la cual se cumplen en la práctica [15] algunos de los beneficios que suelen ser considerados como los más importantes al utilizar dicho modelo. En dicha investigación Conchúir determinó que algunos de esos beneficios se cumplen de forma parcial mientras otros son mitos que no se pueden comprobar como tales en la realidad (see Table 1).

Tabla 1: Supuestos beneficios de adoptar un enfoque de DGS.

Supuesto beneficio	Realidad
Reducción de costos	Beneficio parcial
Aprovechamiento de las zonas horarias	Mito
Distribución por módulos del trabajo de desarrollo	Beneficio parcial
Acceso a un gran número de personas calificadas	Beneficio parcial
Innovación y mejores prácticas compartidas	Mito
Proximidad a los mercados y clientes	Beneficio parcial

Para entender mejor el posible impacto que el modelo DGS puede tener en el proceso de desarrollo y mantenimiento de software conviene considerar el estudio efectuado por Ågerfalk sobre las oportunidades y amenazas que ofrece el enfoque DGS. En ese trabajo se hace una correlación de los factores de distancia temporal, geográfica y sociocultural con los factores de comunicación, coordinación y control [2] para mostrar las oportunidades y amenazas asociadas. En síntesis, el principal reto de la industria de software cuando se usa el modelo DGS es como sobrellevar la distancia en proyectos globales tomando en cuenta las variables y correlaciones mencionadas.

2.2. Trabajo en equipo

El desarrollo de un sistema de software es un proceso complejo que usualmente se divide etapas y tareas, las cuales se encuentran interrelacionadas y pueden realizarse utilizando enfoques muy diversos. El proceso de desarrollo puede requerir de varios años y de la intervención de un gran número de personas, las cuales por lo general son organizadas en grupos de trabajo de dos o más personas (en algunos casos muy especializadas). Dichos grupos de trabajo de forma frecuente se encuentran distribuidos de forma global [44]. Conviene por lo tanto tener presente la diferencia que existe entre un grupo de personas y un grupo de trabajo: un grupo de personas es una colección de personas con funciones que pueden variar de forma considerable y cuyo trabajo no depende necesariamente de los demás miembros del grupo, mientras que un grupo de trabajo se encuentra formado por un grupo de individuos diferenciados e interdependientes entre sí [47]. En este contexto también es importante tener presente que cuando los individuos hacen trabajo en equipo se comunican, interactúan y coordinan entre sí, e incluso se controlan el trabajo mutuamente de forma independiente al rol que cumplan. Esto se diferencia del trabajo enfocado en tareas, en el cual los individuos llevan a cabo tareas independientes a las que realizan los otros miembros del equipo. Sin embargo, las habilidades para el trabajo en equipo y el trabajo enfocado en tareas son complementarias para lograr cumplir con las tareas y objetivos asignados al team [76].

Las organizaciones han adoptado el trabajo en equipo para el desarrollo de software, con preferencia sobre el trabajo individual, porque consideran que el funcionamiento efectivo de los grupos de trabajo puede proporcionar buenos resultados [23] por la diversidad de sus miembros, en términos de experiencia y especialización, aunque conviene recordar las diferencias de nacionalidad, cultura y localización geográfica de estos cuando se trabaja con el modelo de DGS (estas diferencias pueden constituir ventajas y desventajas) [31, 44].

Varios argumentos a favor del trabajo en equipo es que las tareas que le son asignadas las pueden llevar a cabo de forma más rápida, efectiva y eficiente [23, 44] que un solo individuo:

- * Pueden detectar, reconocer y resolver problemas de forma más rápida.
- * Pueden planificar, adquirir conocimiento y diseñar soluciones o productos en menos tiempo.
- * Es posible que se adapten a los cambios con rapidez.
- * Son capaces de evaluar una situación, y tomar mejores decisiones al considerar la combinación de conocimientos y experiencias de sus miembros.
- * Tienen la capacidad para gestionar mejor el estrés y las tareas, en general en periodos de cargas de trabajo muy pesadas.
- * La actuación coordinada de los grupos de trabajo, como una propiedad intrínseca, hace que actúen en bloque ante las situaciones que se les presentan y tengan mayor capacidad de reacción.

Para que un equipo de trabajo funcione de forma adecuada quienes lo conforman deben ser capaces de trabajar juntos de forma efectiva [31]: es necesario que los miembros del equipo se comuniquen, coordinen y trabajen juntos de forma efectiva, particularmente cuando se usa un modelo DGS. Por lo que es importante que los miembros de los equipos, y los equipos como tales, cuenten con la información que requieren mediante una distribución adecuada de acuerdo con las tareas que llevan a cabo [44]. La importancia de lo anterior radica en que al final del día los módulos y diferentes elementos del sistema deben integrarse y funcionar en conjunto, y por lo general la forma como se realiza la asignación de tareas a los grupos de trabajo e individuos es tomando como base la arquitectura del sistema y los módulos que la componen [63].

2.2.1. Comunicación, coordinación, control y cognición

Los factores de comunicación, coordinación y control se encuentran estrechamente relacionados entre sí, pero a estos es importante agregar el factor “cognición” de acuerdo con el trabajo efectuado por Comfort [14]. La gestión de emergencias [14] tiene varias similitudes con la naturaleza cambiante y dinámica (a veces poco predecible) del desarrollo de sistemas de software: el desarrollo de software requiere de la rápida adaptación de los actores de acuerdo a los eventos que se presentan y los nuevos escenarios que se originan, particularmente cuando su desarrollo se lleva a cabo de forma distribuida. Conviene tener en consideración que estos factores intervienen cuando se utiliza un enfoque vertical u horizontal. Lo cual implica que el administrador del proyecto realiza las acciones (comunicación, coordinación y control) siguiendo un enfoque jerárquico con el primer enfoque, mientras que cualquier miembro del equipo lleva a cabo las acciones cuando se usa el segundo enfoque, el cual tiene en cuenta que los trabajadores son profesionales con habilidades y capacidad para reaccionar y tomar acciones cuando lo consideran conveniente [12]. Los cuatro factores mencionados se discuten a continuación y posteriormente, en la siguiente sección, se discute su relación con los conceptos de cognición de equipo y conciencia de equipo.

Cognición: Usando con algunas modificaciones las definiciones proporcionadas por Comfort [14], la cognición se define como un proceso que depende de un modelo mental claro sobre como debe funcionar el sistema bajo observación, y por lo tanto es el elemento que activa los procesos de comunicación, coordinación y control cuando se detectan discrepancias entre lo que es visto por los individuos como un desempeño normal y los cambios de estado de los indicadores claves que alertan sobre desviaciones potenciales del proceso.

Comunicación: El fin de la comunicación es “comunicar” o hacer partícipe a otros de algo mediante un lenguaje común. El diccionario The Merriam-Webster define comunicación como “un proceso mediante el cual la información es intercambiada entre individuos a través de un sistema común de símbolos, señales o comportamiento” [60]. Los tipos de comunicación que se pueden dar en el contexto de DGS entre los individuos son diversos e incluyen los siguientes:

- * Cara a cara o a distancia (e.g. email, teléfono, video-conferencia).
- * Síncrona o asíncrona.
- * Formal e informal.
- * Comunicaciones controladas de forma centralizada y descentralizada.

El principal problema durante el desarrollo de proyectos de software es la falta de comunicación [2] entre los miembros de los equipos de desarrollo, sin importar el modelo utilizado (en la misma ubicación or DGS). Al respecto David Parnas [2] señala que existe una comunicación pobre en diferentes niveles:

- * Usuarios y desarrolladores.
- * Arquitectos y programadores.
- * Entre los programadores del equipo de trabajo.

Con respecto a la falta de comunicación entre los miembros del equipo de desarrollo Ramesh [72] resume los siguientes puntos:

- * Dificultades para iniciar la comunicación.
- * Falta de entendimiento.
- * Reducción dramática de la frecuencia de la comunicación entre los miembros del equipo.
- * Incrementos en el costo de la comunicación en términos de tiempo, personal y dinero.
- * Diferencias horarias.

Es importante agregar que en un entorno de desarrollo distribuido la intervención de más personas suma tiempos de retraso a la realización de las tareas por las distancias y aspectos de comunicación mencionados [58, 63], por lo que se deben considerar medidas adicionales para mitigar el impacto negativo [63, 34, 33], entre las que se encuentran las siguientes:

1. Dividir las tareas de forma óptima entre los sitios.
2. Incrementar la comunicación entre los miembros del equipo utilizando herramientas adecuadas para ese propósito.
3. Contratar o identificar de forma interna el personal con las destrezas y experiencia necesaria.
4. Herramientas para brindar información sobre las actividades que están realizando los miembros del equipo de trabajo.

Coordinación: La coordinación puede ser implícita o explícita. La coordinación implícita depende del conocimiento del equipo de trabajo y su capacidad para tomar decisiones en situaciones críticas con un nivel reducido de comunicación, por lo que los miembros del equipo ajustan sus comportamientos de forma dinámica para anticiparse a las acciones y atender de forma proactiva las tareas que así lo requieren [42, 58]. La coordinación implícita puede proporcionar ventajas cuando la carga de trabajo es muy alta, debido a que se requiere menos comunicación, y está asociada a equipos con alto desempeño en los cuales los miembros entienden de forma clara las necesidades y responsabilidades de sus tareas [58]. En tanto la coordinación explícita es el proceso de organizar las cosas, personas o grupos para que trabajen juntos de forma apropiada [60].

Para efectuar las labores de coordinación es necesario contar y utilizar mecanismos que permitan los procesos de intercambio efectivo de información y entendimiento para alinear las prioridades y acciones de los diferentes actores para alcanzar una meta compartida [14, 48, 58]. Las ventajas y desventajas de usar uno u otro tipo de coordinación depende de las circunstancias y las tareas que se llevan a cabo [58]. Cabe agregar que los mecanismos de coordinación adquieren una importancia particular cuando se utiliza un enfoque DGS por las distancias (geográficas, culturales y horarias) que existen entre los diferentes sitios involucrados en el desarrollo del proyecto [33].

Durante el desarrollo de software el proceso de coordinación implica que las personas que trabajan en un proyecto han acordado una serie de elementos a partir de las especificaciones de un diseño detallado que permitirán construir y organizar las partes de un sistema para que trabajen juntas de acuerdo a las necesidades de los usuarios y los requerimientos de la organización. De acuerdo con el trabajo realizado por Kraut [49] los siguientes son factores que inciden en la coordinación del desarrollo de sistemas:

Escala: Los proyectos de gran tamaño requieren la intervención de un gran número de personas y no es posible que solo una persona o un pequeño grupo conozca todos los detalles [13]. La coordinación de un proyecto se vuelve más difícil conforme aumenta su tamaño y complejidad [49], lo cual lleva a una necesaria división y especialización del trabajo.

Tiempo: El desarrollo de los grandes proyectos de software usualmente se extiende por varios años y su mantenimiento por muchos años más, por lo que continúa requiriendo de la intervención de un gran número de personas una vez que ha finalizado la etapa de desarrollo.

Incertidumbre: El desarrollo de muchos sistemas de software se torna en un proceso impredecible por algunas o varias de las siguientes razones:

- * Las especificaciones del sistema son incompletas por la pérdida de información al traducir las necesidades de los usuarios y el negocio en especificaciones. Dicha pérdida de información se produce porque los analistas, arquitectos y diseñadores en muchos casos no son especialistas del dominio del problema y no pueden comprender todos los detalles o porque aunque los entienden no los incluyen en las especificaciones.
- * No se desarrolla un prototipo que permita capturar la funcionalidad básica del sistema para luego efectuar modificaciones sucesivas con base en las especificaciones y la retroalimentación de los usuarios.
- * El entorno de las organizaciones en general es cambiante, por lo que la funcionalidad del sistema requiere cambiar con el tiempo. Esto produce cambios en las especificaciones durante el proceso de desarrollo o posteriormente cuando el sistema ha entrado en operación [53, 13].
- * Falta de definición de metodología, ausencia de medición de la calidad, detección de problemas, errores y fallas (algunas veces de forma tardía) durante el desarrollo del proyecto [74].
- * Problemas con los miembros del equipo debido a la falta de idoneidad o debido a un número excesivo o limitado de participantes [74].

- * Las diferencias de los puntos de vista entre los diferentes actores que intervienen en el desarrollo del sistema.
- * La complejidad propia de un sistema de gran tamaño que debe desarrollarse durante un largo periodo de tiempo [53, 13].

Interdependencia: Los sistemas se construyen a partir de componentes, en algunos casos miles, que luego deben ser integrados de forma precisa.

Comunicación formal e informal: La comunicación es clave para realizar la coordinación entre los miembros del equipo de trabajo y entre los subgrupos que se encuentran dispersos en diferentes localidades geográficas, por lo que se requiere hacer uso de la comunicación formal e informal [53] de acuerdo al tipo de problema que se está abordando.

En el mismo trabajo Kraut enumera una lista de técnicas de coordinación, entre las que se encuentran las siguientes:

- * Procedimientos impersonales formales (documentos de proyectos y memorandos, peticiones de modificación, procedimientos de seguimiento de errores y diccionarios de datos).
- * Procedimientos interpersonales formales (reuniones para revisar el estado y el diseño e inspecciones de código).
- * Procedimientos interpersonales informales (reuniones de grupo y ubicación común de requerimientos y personal de desarrollo).
- * Comunicaciones electrónicas (email and boletines electrónicos).

Control: Tomando en consideración los diversos factores que causan incertidumbre durante el proceso de desarrollo de software, el control se puede definir como la capacidad de mantener las acciones enfocadas en alcanzar las metas y objetivos fijados [14, 60] para producir un producto de software correcto, a tiempo y dentro del presupuesto [74]. Lo anterior, de acuerdo a los requerimientos, especificaciones, plazos, costos, estándares, políticas, niveles de calidad y otros factores inherentes al proceso de software que al ser atendidos de forma oportuna facilitan lograr el éxito del proyecto. Además, conviene considerar la dificultad para realizar el control cuando se utiliza el modelo DGS [72] por los factores discutidos en los apartados anteriores.

El control usualmente es clasificado en dos categorías: formal e informal. El control formal consiste en el monitoreo y evaluación del comportamiento y sus resultados, en donde el comportamiento del control consiste en controlar como se comportan los individuos y el control de los resultados consiste en medir el efecto de los resultados del comportamiento. Para utilizar el control del comportamiento, se requiere conocer de forma precisa las acciones que se deben realizar durante el proceso de desarrollo del proyecto para transformar una serie de entradas (i.e. requerimientos y diseños) en salidas (i.e. sistema que funciona correctamente). Por lo que se toman como base las acciones que se deben realizar y las acciones llevadas a cabo por los individuos se pueden determinar si su comportamiento ha sido el correcto. En tanto que el control de los resultados se puede utilizar cuando es factible medir el desempeño de los individuos de acuerdo a los resultados producidos y los que se esperaba obtener, de forma independiente a los comportamientos de estos [65, 45].

En el caso del control informal, los dos tipos más conocidos son el control del clan y el autocontrol. Un clan es un grupo de personas que tienen dependencias mutua, metas, objetivos, valores, filosofía y creencias comunes, además de un alto sentido de identidad y pertenencia de grupo con comportamientos que no son conocidos a priori y cuyos resultados pueden evolucionar con el tiempo. En este tipo de control el grupo se autocontrola, ejerciendo en muchos casos presión de grupo. Por lo que la selección de los individuos que participan en estos grupos debe ser cuidadosa y se requiere que cuenten con la capacitación adecuada. En lo que respecta al autocontrol, en este tipo de control el individuo se pone sus propias metas, monitorea, evalúa su propio progreso y se encuentra motivado para llevar a cabo su trabajo; por lo que este tipo de control resulta de utilidad en tareas que requieren de autonomía, creatividad y trabajo intelectual [45].

Haciendo un paralelismo entre la incertidumbre que está presente durante el desarrollo de los proyectos de software con la gestión de emergencias, resulta de utilidad tomar en cuenta el enfoque de Comfort con relación a la actuación individual y grupal de los miembros del equipo en las tareas de control. Comfort [14] considera que se puede mantener el control en situaciones cambiantes y de alta complejidad si los siguientes factores están presentes:

- * Conocimiento compartido de la situación.
- * Habilidades comunes.
- * Ajuste recíproco de las acciones de acuerdo a la evolución de la situación.

El fin de la actividad de control es habilitar y facilitar la toma de decisiones comparando la información obtenida (en la forma de informes de progreso y estado) de las actividades de desarrollo del proyecto, verificación, validación y prueba, [Aseguramiento de la Calidad de Software \(SQA\)](#) [25, 46, 39], [SCM](#) [74], monitoreo de errores, incidentes y sistemas de control de cambio [6], con los resultados que se espera obtener del proyecto de acuerdo a:

- * La planificación del sistema.
- * Los estándares y procedimientos del proyecto.
- * Modelos y análisis de riesgo.
- * Planes y procedimientos de administración de la configuración.
- * Requerimientos del sistema.
- * Diseño detallado y de alto nivel.
- * Requerimientos y planes de [SQA](#).
- * Plan general y detallado de pruebas del sistema.

El control, en su forma ideal, es un proceso continuo de retroalimentación que busca identificar y eliminar riesgos potenciales a partir de:

Detección y control de cambios imprevistos: Consiste de la detección de cambios que se deben realizar por variaciones en los requerimientos del sistema, de forma que estos deben ser controlados para mantener la integridad del diseño y deben ser incorporados de forma ordenada.

Detección y corrección de errores: Uno de los principales objetivos del proceso de control es detectar y corregir desviaciones o errores para alinear el desarrollo de acuerdo a los requerimientos, especificaciones, metas y objetivos.

Monitoreo: Tiene como fin medir el avance del proyecto mediante reuniones, herramientas especializadas y el uso de técnicas de SQA como métricas, revisiones técnicas e inspecciones de software.

Evaluación: Consiste en analizar la información de control disponible, efectuar la evaluación de las consecuencias de las alternativas posibles, elegir una de esas alternativas (toma de decisiones) y trazar el curso a seguir.

2.2.2. Conciencia situacional del equipo

El trabajo en equipo permite que los integrantes de un equipos se familiaricen con los demás miembros y conozcan sobre los conocimientos, habilidades, experiencia, background, personalidades y hábitos de cada uno. Ese conocimiento mutuo varía con el tiempo y se incrementa conforme este pasa, lo que hace posible una mejor planificación del trabajo [23].

Desde un punto de vista cognitivo, los equipos de trabajo construyen un modelo mental² a partir del entendimiento compartido de las tareas e involucra conocer los procedimientos, acciones y estrategias para llevarlas a cabo. Por lo que los miembros del equipo de trabajo deben tener expectativas y conceptualizaciones comunes de las tareas [31]. De ahí que un modelo mental compartido (que no consiste en la suma de sus modelos mentales) es la representación del conocimiento de forma organizada con relación a las tareas, situaciones, patrones de respuesta, metas, estrategias y relaciones de trabajo. Por lo que se puede decir que el modelo mental es la forma como el equipo de trabajo piensa de forma colectiva y caracteriza las situaciones de acuerdo a creencias, supuestos y percepciones comunes [47].

Como se mencionó el modelo mental de un equipo de trabajo no es la suma de los modelos mentales de los individuos, pero si toma en cuenta el conocimiento relevante de sus miembros y lo transforma en el conocimiento del equipo con el fin de guiar la toma de decisiones y la realización de acciones para alcanzar las metas y objetivos de las tareas que le han sido encomendadas [23]. De acuerdo con Cooke entre los elementos que pueden incluir los procesos cognitivos de los equipos de trabajo en este contexto [16] se encuentran:

- * El aprendizaje.
- * La planificación.
- * El razonamiento.
- * La habilidad toma de decisiones.
- * La capacidad para la resolución de problemas y recordar.
- * La habilidad para evaluar situaciones.

²El modelo mental de un individuo es la forma como el conocimiento y la información son representadas por su mente y reflejan la tendencia a categorizar lo que saben [47].

La cognición de los equipos de trabajo se construye a partir de las interacciones entre los miembros del equipo [58, 31] cuando:

- * Trabajan juntos.
- * Clarifican los roles de cada uno en cada tarea.
- * Se distribuyen las sub-tareas.
- * Se comunican por diferentes medios para construir y mantener un modelo mental compartido de la situación [58].
- * Se reúnen, de forma presencia o virtual, para compartir puntos de vista y preocupaciones.
- * Coordinan las actividades del proyecto.
- * Observan el trabajo de los demás y aprenden de ellos.
- * Monitorean el avance de actividades.

La cognición de un equipo de trabajo es su capacidad para adquirir, procesar, almacenar y utilizar el conocimiento durante la realización de las tareas, en especial cuando se requiere la colaboración de un gran número de personas o la intervención de sus miembros en tiempo real para resolver alguna situación urgente [44]. Algunas características relevantes que se deben considerar con respecto a la cognición de los equipos de trabajo [23, 24, 31] son las siguientes:

- * No es la suma de la cognición individual de los miembros, pero sí hace uso de la cognición individual de sus miembros.
- * Consiste en la suma de comportamientos del equipo durante la comunicación, coordinación y control de actividades.
- * El equipo hace uso modelos mentales compartidos en torno a los aspectos relacionados con el proyecto y las tareas.
- * Tienen conciencia de las metas y objetivos del trabajo que realizan, así como del estado de las actividades y tareas (conciencia situacional) [58].
- * Los miembros del equipo de trabajo exhiben comportamientos y actitudes que evidencian la actuación coordinada entre ellos [24].

Endsley define la conciencia situacional como la percepción de los elementos en el ambiente en un periodo de tiempo y espacio determinado, así como su comprensión y la proyección de esos elementos en un futuro cercano [21, 78, 76]. En esta investigación conciencia situacional es definida como el grado de conocimiento sobre el estado de las tareas, actividades, cambios y resolución de problemas relacionados con el proceso de desarrollo, mantenimiento y evolución de un sistema de software. En esta investigación se considera que la conciencia situacional tiene importancia tanto a nivel individual como del equipo de trabajo en los procesos de software. Por lo que los siguientes puntos merecen especial consideración:

Perspectiva individual: Desde una perspectiva individual, la conciencia situacional es la conciencia sobre el estado de las cosas de un miembro particular del equipo de trabajo. El estado de conciencia y conocimiento con el que cuenta

el individuo sobre las tareas que le han sido asignadas y los factores que las afectan para llevar a cabo su desarrollo de forma exitosa.

Perspectiva del equipo de trabajo: Desde una perspectiva del equipo de trabajo, la conciencia situacional es la conciencia y conocimiento compartido por los miembros del equipo sobre el estado del sistema en términos generales y específicos. Cada miembro del equipo cuenta con el conocimiento específico de los factores relacionados con sus tareas pero también con una visión, en términos de conciencia sobre el estado general de las cosas que le permite colaborar con el resto de miembros del equipo para contribuir con el sistema. A este tipo de estado de conciencia se le conoce como conciencia situacional del equipo de trabajo. Es importante notar que la conciencia situacional del equipo se encuentra estrechamente relacionada con los modelos mentales y la cognición del equipo, como se explica más adelante. La construcción de la conciencia situacional del equipo, al igual que la construcción de los modelos mentales compartidos y la cognición del equipo, no es la suma conciencia situacional individual, pero sí es la composición que se origina de cada perspectiva individual y los puntos de coincidencia de los miembros del equipo sobre el estado de las cosas en términos del sistema en general. En este contexto se considera que la conciencia situacional forma parte de un entorno de trabajo colaborativo en el que además intervienen otros elementos relacionados con el trabajo en equipo.

La forma como los individuos procesan la información y construyen su modelo mental de conciencia situacional depende de sus objetivos, habilidades, experiencia, capacitación y rol desempeñado en los procesos de DMES. Por lo que los administradores de proyectos están interesados en los aspectos de más alto nivel [55] y los programadores en los detalles más específicos.

La experiencia de los individuos y el equipo de trabajo en sistemas similares son de gran valor en la construcción de la conciencia situacional, principalmente en cuanto al conocimiento de las habilidades de los demás miembros del equipo y la asignación de tareas [31], pero también en torno a la mecánica de funcionamiento del equipo. Sin embargo, cada proyecto de software es único por las diferencias de los problemas que buscan resolver, los enfoques que se usan para resolverlos y la interdependencia de los elementos internos del sistema. Por lo que aunque un sistema de software tenga elementos comunes en relación con otro, los problemas a los que se enfrentan los individuos y el equipo de trabajo, así como las variables a tomar en cuenta, son diferentes. De forma que la construcción de la conciencia situacional requiere de tiempo, por ser necesaria la acumulación de experiencia con el proyecto en que trabajan el equipo y quienes lo conforman.

2.2.3. Conciencia situacional distribuida

Otro enfoque más reciente, que es complementario con el anterior, es conocido como Espacio de Trabajo de Conciencia Situacional Distribuida (ETCSD), el cual tiene una orientación basada en sistemas en la cual los individuos y los elemen-

tos tecnológicos son considerados como agentes, que interáctuan, tienen diferentes propósitos (por las tareas y actividades que llevan a cabo) y su propia **Conciencia Situacional (CS)** sobre las tareas que realizan y el proyecto en general. La idea detrás es que los miembros de un equipo de trabajo no necesitan conocer todos los detalles sobre el proyecto en que trabajan, sino solo aquellos detalles que les permita realizar sus tareas. Pero implica que deben tener conciencia sobre lo que sucede con el proyecto de forma general y saber la información que los demás requieren y necesitan conocer sobre las tareas que tiene bajo su responsabilidad [79].

Cabe mencionar que el acceso a la misma información no produce la misma conciencia situacional en los miembros de un equipo, por las metas, tareas, roles y experiencia de cada uno, que hace que la usen de forma diferente a los demás. Por lo que el conocimiento se encuentra distribuido en el entorno y la **CS** de un agente puede ser diferente pero compatible con la **CS** de otros agentes. La realización de determinadas tareas que se encuentran relacionadas entre sí requieren la colaboración entre los agentes para que las lleven a cabo de forma coordinada o conjunta, por lo que la compatibilidad de la **CS** de los agentes resulta de gran utilidad.

De acuerdo con los principios de **ETCSD** enunciados por Stanton *et al.* [79], los siguientes puntos son de importancia:

- * Tanto los agentes humanos como no-humanos cuentan con su propia **CS**.
- * Cada agente tienen su propio punto de vista, el cual puede ser diferente al punto de vista de otros agentes, sobre la misma situación.
- * Los agentes tienen roles protagónicos en el desarrollo y mantenimiento de la **CS** de los demás agentes a través de la interacción que tiene lugar entre ellos.
- * Los agentes compensan entre sí la falta de **CS** de uno o varios agentes en determinadas situaciones.
- * Los puntos de vista que tienen los agentes sobre el sistema cambian con el tiempo de acuerdo a las tareas que llevan a cabo.
- * El conocimiento que conforma **ETCSD** se activa en diferentes momentos de acuerdo a las metas, objetivos y requerimientos de las situaciones, tareas y actividades que surgen y se llevan a cabo en el tiempo.
- * Las coincidencias de **CS** entre agentes depende de las metas que estos tengan.
- * La comunicación entre agentes puede ser no verbal, con mecanismos electrónicos u otras formas personalizadas.
- * **ETCSD** ayuda a la cohesión de sistemas poco acoplados.
- * Las perspectivas de los agentes pueden ser redundantes pero siempre son complementarias en los entornos colaborativos.
- * **ETCSD** ve a **CS** como una propiedad que emerge del sistema y sus partes, y no como algo que existe en la mente de los individuos.

- * En los entornos de trabajo colaborativo **ETCSD** puede ser visto como una propiedad y un producto de la interacción entre los agentes y el comportamiento de estos en el entorno.

2.3. Consideraciones para diseñar espacios para la conciencia situacional

Tomando en consideración los conceptos y definiciones enunciadas, realizar el diseño de un espacio para facilitar la construcción de la conciencia situacional del equipo de trabajo requiere considerar que dicho espacio, además de proporcionar detalles que permitan comprender la situación actual, debe tomar en cuenta los hechos pasados para obtener información sobre como se llegó a un determinado estado, pero además debe apoyar la realización de proyecciones futuras a partir del análisis de esa información.

El diseño de un **Espacio de Conciencia Situacional (ECS)** requiere identificar los elementos que el individuo o equipo de trabajo deben conocer, de acuerdo a los objetivos y metas del proyecto, y las tareas o actividades que les han sido encomendadas. Entonces, para apoyar el proceso de mantenimiento de software, por ejemplo, resulta de utilidad que dicho espacio brinde información que ayude a comprender los cambios que se realizan en la arquitectura del sistema, lo cual puede contribuir a mejorar el desempeño de los individuos y el equipo en general [75, 23].

Para realizar el diseño de un **ECS** es importante tener en consideración los puntos que se presentan a continuación:

- * Es común que los proyectos de software se lleven a cabo utilizando un modelo **DGS**.
- * La meta de un **ECS** es apoyar la construcción de un **Conciencia Situacional del Equipo de Trabajo (CSET)** con el fin de facilitar la colaboración entre los miembros del equipo cuando llevan a cabo las tareas y actividades bajo su responsabilidad.
- * **DMES** es un proceso dinámico cuyo estado cambia de forma constante, lo que dificulta mantener actualizada la información sobre las actividades, tareas y patrones de progreso que hayan sido determinadas como de interés durante el diseño del **ECS**.
- * En entornos complejos y cambiantes, como el de **DMES**, las decisiones se deben tomar en poco tiempo, por lo que se requiere de información actualizada.
- * Contar con información actualizada sobre el estado de las cosas es importante incluso cuando los miembros del equipo realizan tareas triviales.
- * Un **ECS** debe brindar información sobre el estado presente de las cosas, pero es necesario considerar aquellos elementos que puedan revelar cambios futuros

del estado del proceso con el fin es ayudar a decidir mejor el curso de las acciones a realizar.

- * Contar con información errónea sobre el estado de las cosas puede llevar a tomar decisiones equivocadas.
- * Los miembros del equipo requieren conocer lo que ocurre en relación con sus tareas y el proyecto en general, pero además deben ser capaces de interpretar correctamente las diferentes situaciones de acuerdo a las metas del proyecto para que con base en estas realicen la toma de decisiones y lleven a cabo las acciones pertinentes de forma oportuna (entender la situación →tomar decisiones →llevar a cabo acciones).
- * La falta de capacitación de los miembros del equipo o que no cuenten con las habilidades requeridas puede conducir a una mala interpretación de CS [75, 21], lo cual puede tener como consecuencia la toma incorrecta de decisiones.
- * La conformación de equipos de trabajo con estructuras cohesivas en términos de conocimiento, habilidades y control de actividades es un objetivo difícil de alcanzar.

El uso de la visualización de software surge, entonces, como una alternativa viable para construir un ECS por sus capacidades para transmitir información y proporcionar mecanismos de interacción con los usuarios. Las herramientas de visualización pueden hacer posible la interacción entre usuarios mediante métodos de anotación o simplemente al proporcionar información sobre lo que sucede a los miembros de equipo de trabajo para que inicien las tareas de comunicación, coordinación y control a partir de los procesos de cognición que se activan al adquirir conciencia sobre la situación.

Conviene recordar que en esta investigación los términos *representación* y *visualización* se usan de forma indistinta, pero en lo que resta de este capítulo se hace la distinción entre ambos para explicar de forma adecuada los elementos que se deben considerar al diseñar una herramienta de visualización para sistemas de software. En términos fundamentales una representación es un elemento gráfico simple (*e.g.* símbolos, glifos y figuras geométricas simples) que sirven para transmitir detalles concretos relacionados con un evento o dato y forman parte de una visualización, como un nodo o una arista en un grafo, mientras que una visualización es un conjunto de representaciones individuales que de forma conjunta dan mayor significado a la información, por ejemplo, al utilizar relaciones, como en un grafo [86].

En cualquiera de los campos relacionados con *Visualización de la Información (VI)*, y de acuerdo al interés de esta investigación con la visualización de sistemas de software es de importancia tener en consideración que el diseño adecuado de una visualización debe tener en consideración las capacidades sensoriales, de percepción y cognición de los usuarios [10]. Por lo que se deben tener en cuenta aquellos elementos que permitan a los APs y programadores comprender la información que se transmite visualmente de forma inmediata (procesamiento preatentivo), sin entrenamiento previo y de forma independiente a su cultura u origen. Para ello se puede

tomar en cuenta el uso de representaciones basadas en convenciones internacionales así como uso de patrones visuales y colores basados en los principios de las leyes de Gestalt [83].

En la literatura científica existen un gran número de artículos que hacen referencia a elementos que deben ser considerados durante el diseño de herramientas de visualización para apoyar las tareas del proceso DMES. En esa línea, Young and Munro [86] identificaron 6 factores a tener en cuenta cuando se diseñan herramientas usando $3D$ (representación, abstracción, navegación, interacción, correlación and automatización), pero por su relevancia se aplican también son aplicables a visualizaciones en $2D$. Por su parte, Maletic *et al.* [59] enumeraron 5 aspectos (tareas, audiencia, datos objetivo, representación and medio). Tomando como referencia los factores identificados por ambas investigaciones, para el diseño de una herramienta para la visualización de sistemas de software podría ser de interés tomar en consideración los elementos que se presentan a continuación.

Audiencia: El diseño de la visualización y las técnicas de interacción que se utilicen deben tener en consideración quienes serán los usuarios y las representaciones del contenido de acuerdo a su interés y necesidades.

Tareas: Es necesario identificar de forma precisa las tareas que la visualización busca apoyar, para determinar las representaciones visuales más adecuadas.

Datos objetivo: La identificación de los datos es un aspecto crítico, porque con base en los datos disponibles, las tareas y la audiencia que se pretende apoyar se deberán definir las visualizaciones que se requieren y el diseño de cada una.

Correlación: Consiste no solo en la correlación de la información de diferentes fuentes de datos, sino también en la posibilidad de enlazar los elementos visuales con documentos y código fuente.

Representación: Un problema fundamental en la visualización de los cambios de los sistemas de software es la sección apropiada de los elementos gráficos, metáforas y colores, así como hacer una combinación efectiva de estos elementos para mostrar diferentes perspectivas de los datos [20]. Esto es de gran importancia para diseñar una visualización intuitiva que transmita la información de forma efectiva haciendo uso de un bajo nivel de complejidad cognitiva.

Abstracción: Para que la información que se busca transmitir sea interpretada de forma efectiva se requiere determinar que información y nivel de detalle será presentado (se puede dejar al usuario escoger el nivel de detalle mediante interacción), así como las representaciones y visualizaciones a utilizar.

Navegación e interacción: DMES es un proceso que genera grandes cantidades de información. Lo cual requiere la utilización de varias visualizaciones enlazadas entre sí y que estas sean implementadas utilizando mecanismos de navegación e interacción (e.g. foco + contexto, vista general + detalle, puntos de referencia, zoom, historial y filtrado). El objetivo es que el usuario, de forma simple, tenga la posibilidad de explorar detalles, a la vez que se encuentra debidamente ubicado en el contexto de la

visualización y en el análisis que lleva a cabo. Como parte de la interacción se pueden incorporar elementos que permitan a los usuarios cierto grado de flexibilidad para hacer personalizaciones, como el uso del color.

Automatización: La visualización de grandes sistemas de software, de forma ideal, se debe realizar de forma automática a partir de las fuentes de datos seleccionadas, e incluso es conveniente que incorpore nueva información conforme esta es creada. Young and Munro [86] consideran que se debe permitir la creación de la visualización de forma interactiva, con el usuario tomando las decisiones sobre los elementos del sistema que quiere representar para que mediante el ejercicio práctico obtenga una mejor comprensión del sistema bajo estudio.

3. Visualización de software

En esta sección se realiza una revisión de diferentes técnicas de visualización utilizadas para la representación de la arquitectura de los sistemas de software y su evolución, así como de las actividades que llevan a cabo los programadores y la colaboración que tiene lugar en los procesos de DMES.

3.1. Revisión general de técnicas de visualización

El problema de la visualización consiste en desplegar una gran cantidad de información en un espacio reducido. Por lo que es necesario implementar mecanismos de interacción que permitan navegar a través de los datos sin perder de vista el contexto, pero que además proporcionen medios que faciliten la interpretación de elementos particulares [56].

Las técnicas de visualización se pueden categorizar, de forma general, como orientadas a la distorsión o no orientadas a la distorsión. La figura 1 muestra una taxonomía preparada por Leung [56] para las técnicas orientadas a la visualización de grandes cantidades de información basadas en esas dos categorías.

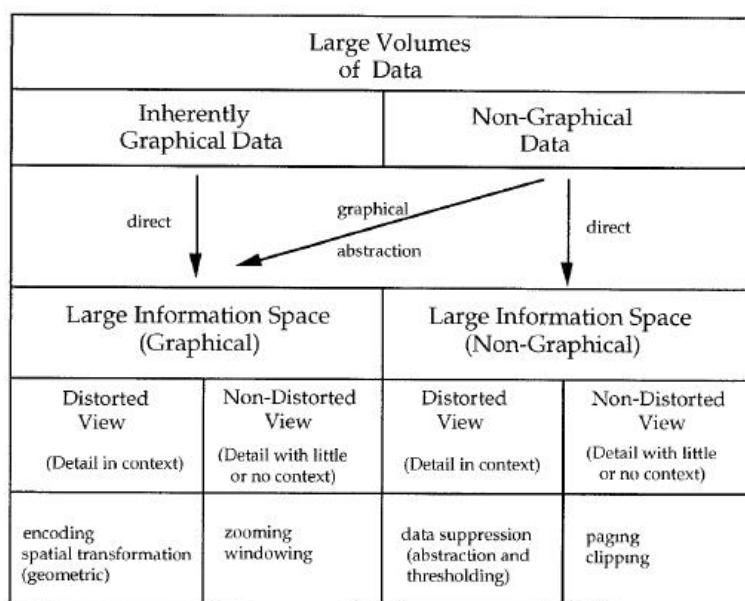


Figura 1: Taxonomía de las técnicas de visualización para grandes cantidades de información

Las técnicas no orientadas a la distorsión son adecuadas para aplicaciones de pequeño tamaño que se encuentran basadas en texto pero no proporcionan un contexto adecuado para soportar la navegación en un conjunto de información a gran escala. En tanto que las técnicas orientadas a la distorsión permiten que el usuario examine de forma detallada, dinámica e interactiva los datos en una sección de la pantalla, mientras se le presenta una vista global adicional del espacio en que se en-

cuentra inmersa esa sección. Dentro de los mecanismos de interacción que se pueden proporcionar al usuario se encuentran:

- * Selección
- * Navegación
- * Filtrado
- * Enlace
- * Zoom

Las técnicas orientadas a la distorsion se utilizan en conjunto con funciones de transformación que definen la forma como la información será presentada al usuario y la interacción que tendrá lugar durante el proceso de obtención de conocimiento. Algunas de las técnicas orientadas a la distorsion que se pueden mencionar son las siguientes:

- * Técnicas de ampliación –Bifocal – Polifocal – Fisheye –
- * Table Lens
- * Perspective Wall
- * Espacios hiperbólicos
- * Circular
- * Jerárquica – Tree map – Cone Trees
- * Líneas de vida
- * Líneas de planificación

Demagnification in both X and Y dimensions	Demagnification in Y dimension	Demagnification in both X and Y dimensions
Demagnification in X dimension	Central 'Focus' Region no demagnification	Demagnification in X dimension
Demagnification in both X and Y dimensions	Demagnification in Y dimension	Demagnification in both X and Y dimensions

Figura 2: Ampliación simple de una área de interés. Figura tomada de [56]

Estas técnicas de visualización se combinan en la etapa de diseño de una solución de visualización de información con el objetivo de brindar mayores posibilidades de visualización y navegación. En este contexto las técnicas de ampliación revisten gran importancia debido a que permiten revisar fácilmente elementos específicos de información que se encuentran inmersos u ocultos en grandes cantidades de información.

Éstas técnicas se utilizan para ampliar un punto concreto, una zona vertical u horizontal o bien para ampliar al mismo tiempo una zona horizontal y una zona vertical que se intersecan en un punto determinado.

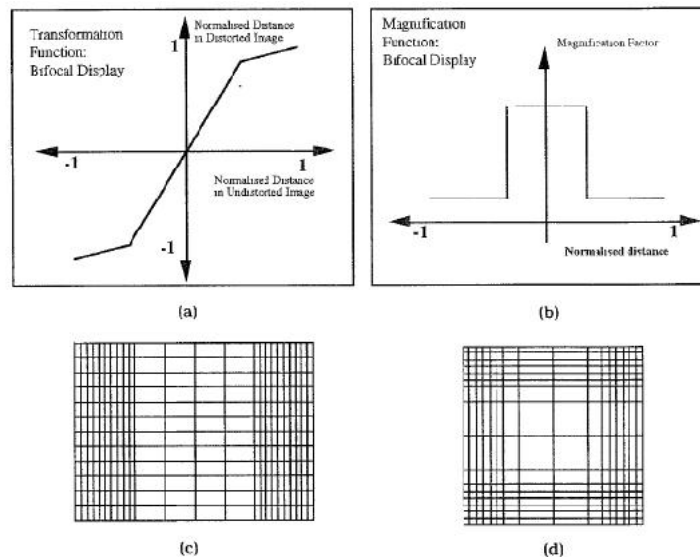


Figura 3: Técnica bifocal para la visualización de información. Figura tomada de [56]

La figura 2 sirve para ilustrar el concepto de ampliación de un punto concreto en una visualización, mientras las figuras 3c y 3d muestran la ampliación de una y dos dimensiones. A la ampliación en una y dos dimensiones se le denominó originalmente como bifocal, fue propuesta por Robert Spence y Mark Apperley en 1982 y posteriormente fue denominada como fisheye (ojo de pez) en 1986 por Furnas [26]. Las figuras 4c y 4d muestran dos configuraciones diferentes de la técnica del ojo de pez.

La aplicación de las técnicas de visualización para ampliar una área concreta resulta de gran utilidad, pero cuando se navega en un conjunto con gran cantidad de elementos es frecuente que resulte necesario comparar dos o más elementos. Para ello se utilizan de forma conjunta las técnicas de ampliación e interacción con el fin de poder observar de forma ampliada y simultánea varias áreas a la vez. Las figuras 5c y 5d muestran la ampliación de varias áreas de interés de forma simultánea y se corresponden con la técnica propuesta por Naftali Kadmon y Eli Shlomi en 1978 para la presentación de datos estadísticos sobre mapas de cartografía.

El “ojo de pez” permite mantener el contexto a la vez que se visualiza una área específica y se puede aplicar en conjunto con cualquier otra técnica de visualización; textual, tabular, jerárquica, circular e hiperbólica, así como a líneas de vida y de planificación, entre otras.

Es importante hacer la distinción entre visualizar información con el ojo de pez y hacerlo con el zoom. El zoom geométrico permite que el usuario especifique la escala del aumento cada vez que amplíe o reduzca el tamaño del área de interés y por lo general se encuentra fijada a un punto concreto y no conserva el contexto.

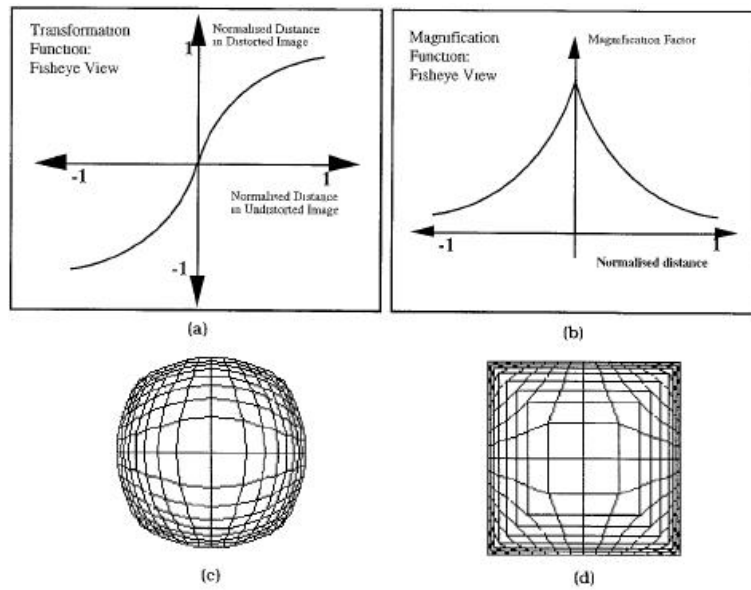


Figura 4: Visualización aumentada mediante la utilización de la técnica del ojo de pez. Figura tomada de [56]

El zoom semántico cambia la forma o contexto en el cual la información está siendo presentada. Un ejemplo de gran utilidad para entender esta técnica es el que proporciona Stephens y en base al cual se desarrolla el siguiente ejemplo: Un individuo cuenta con una agenda electrónica en la cual observa el calendario anual de forma global, cuando el puntero se detiene sobre un mes este se amplía y visualizan los días de ese mes, y al concentrarse en un día particular se muestran las diferentes horas de ese día, y finalmente cuando se posiciona sobre una hora obtiene información sobre las citas o tareas programadas para esa hora.

3.2. Visualización de la arquitectura de los sistemas de software

La arquitectura de un sistema de software tiene como fin mostrar una vista abstracta del sistema y puede ser diseñada usando capas con diferentes niveles de detalle. El nivel de detalle de las capas depende del propósito de la arquitectura, los requerimientos del sistema y el entorno en que funcionará el sistema. Una arquitectura de alto nivel es útil para comunicar una visión general a los gerentes, administradores de proyectos y usuarios, mientras que una arquitectura de bajo nivel sirve para orientar el diseño detallado del sistema y apoyar a los programadores cuando aún no se han familiarizado con el sistema [41], al mostrar, por ejemplo, información sobre las relaciones entre los elementos de software y las estructuras de datos a utilizar. Tomando en cuenta lo anterior, es posible hacer varios diseños de la arquitectura para el sistema, según su propósito. En consecuencia, esta investigación adopta la definición proporcionada por Bass *et al.* [7] con respecto a la arquitectura de un sistema de software:

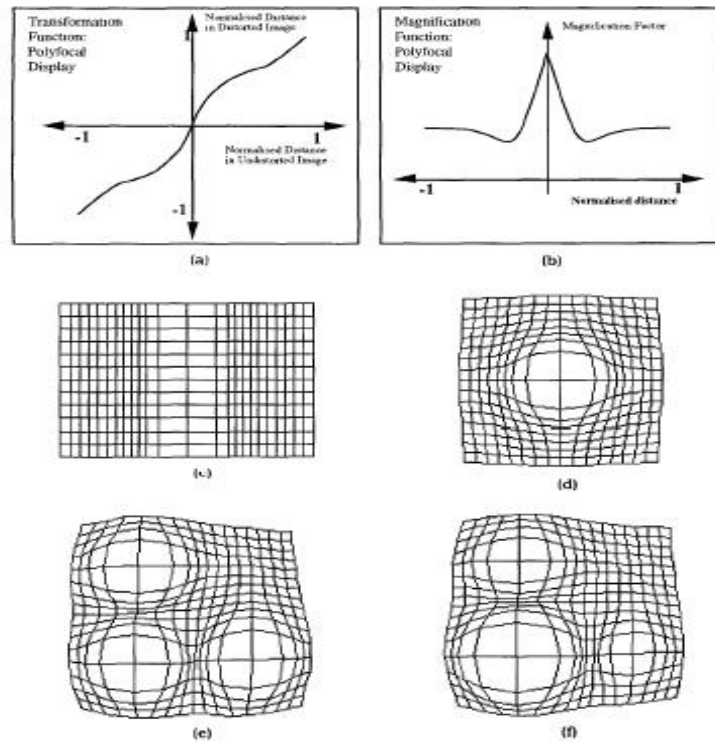


Figura 5: Ejemplo de la técnica polifocal

The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.

Es importante resaltar que a partir de la arquitectura es posible comprender como está organizado un sistema de software en términos de modularidad, componentes, relaciones entre elementos, estructuras de datos, acceso a datos y la distribución física de los componentes del sistema en servidores y equipos de usuarios.

En la actualidad la mayoría de sistemas de software se desarrollan usando lenguajes orientados a objetos y utilizan una estructura jerárquica (packages, classes, methods and attributes). Como consecuencia, el trabajo realizado en esta investigación está basado en el análisis de sistemas de software desarrollados con el paradigma de programación orientada a objetos.

Otra consideración con respecto a la arquitectura de los sistemas es que el proceso de aseguramiento de la calidad del software mide mediante el uso de métricas los elementos que la componen (e.g. tamaño, complejidad, dependencias y relaciones), y el objetivo particular de las métricas de evolución es permitir la comparación y evolución de la calidad de varias revisiones o periodos de tiempo.

En línea con esto, Khan *et al.* [43] realizaron una investigación sobre el uso de la visualización para representar los elementos relacionados con la arquitectura de los sistemas. De acuerdo con esa investigación los elementos más comunes que son representados por las visualizaciones de software, tomando en cuenta una o más

revisiones del sistema, son los siguientes:

- * Acoplamiento y acoplamiento lógico.
- * Dependencias.
- * Métricas y métricas de evolución.
- * Relaciones entre elementos de software (e.g. herencia e implementación de interfaces).
- * Estructura y cambios en la estructura.
- * Vocabulario.

3.2.1. Visualización en 3D

MUDRIK [3] es un sistema 3D desarrollado en Java y OpenGL para apoyar a los programadores en la comprensión de las bibliotecas externas (escritas en Java) que son usadas por los proyectos de software y con las cuales no se encuentran familiarizados. El gran tamaño de algunas bibliotecas dificultan su comprensión, pero también la forma como se ha organizado su desarrollo. Como es señalado por Ali, en la actualidad un gran número de bibliotecas de código abierto son desarrolladas por programadores que contribuyen de forma voluntaria usando mecanismos de coordinación desacoplados, por que algunas bibliotecas no cuenta con una organización óptimas, como sí la podrían tener las bibliotecas desarrolladas usando mecanismos de coordinación más rigurosos [3].

Mudrik representa la estructura de las bibliotecas haciendo uso de 3 visualizaciones y proporciona mecanismos de exploración y búsqueda para facilitar la obtención de información sobre dicha estructura, y la funcionalidad de las clases que la componen. Las visualizaciones utilizadas son una representación de árbol con la estructura completa de la biblioteca, la cual ha sido llamada *Class Browser*, una estructura del tipo *Cone Tree* y una matriz de dependencias (ambas en 3D).

Class Browser es una visualización simple de un árbol (similar al *JTree* de Java) que muestra la estructura de la biblioteca (paquetes, clases e interfaces). Esta visualización permite seleccionar los elementos que el usuario quiere explorar en la visualización del tipo *Cone Tree*. Cuando un elemento individual es seleccionado en *Class Browser*, dependiendo de la opción de visualización seleccionada, el sistema presenta una vista para los detalles de ese elemento, para las subclasses del elemento o para las relaciones que ese elemento tiene con otros elementos. En caso de que el elemento seleccionado sea la raíz de la biblioteca (de forma posterior el sistema permite filtrar las clases mediante un control de filtrado), el sistema presenta, de acuerdo a la opción seleccionada, la jerarquía de la herencia de todas las clases (usando la visualización del *Cone Tree*) o las relaciones entre todas las clases (usando una matriz de dependencias con histogramas en las celdas para representar el número de referencias en las relaciones de dependencia).

3.2.2. Metáforas de ciudad

El uso de metáforas de ciudad se ha vuelto popular en la visualización de software en años recientes, después de que Panas *et al.* propusieron su uso para representar la arquitectura de sistemas de software [67, 68]. Conviene indicar que esta investigación se refiere a este tipo de visualizaciones con el nombre de “software city” de ahora en adelante.

La forma como la representación “software city” es utilizada para visualizar los elementos de un sistema difiere según el investigador o grupo de investigación que la utiliza. Por ejemplo, Panas *et al.* representan un paquete haciendo uso de la visualización completa y usan los distritos para representar las clases y los edificios para los métodos. En tanto los trabajos de investigación realizados por Wettel y Lanza usan los distritos y subdistritos para representar los paquetes y sub-paquetes, los edificios para describir los elementos de softwares (clases e interfaces) y los ladrillos para representar los métodos [84, 85]. En general ambos enfoques consideran el uso de la altura, ancho y color de los edificios para representar diferentes tipos de métricas de los elementos de software. Con relación a la estructura, el primer enfoque difiere del segundo en que este último representa el sistema completo, mientras que el primero solo representa un paquete. De forma que la estructura visual del enfoque de Panas *et al.* comienza con la representación de los sub-paquetes, mientras que el enfoque usado por Wettel y Lanza inicia con la visualización de los subpaquetes. Así, en el primer enfoque los distritos (sub-paquetes) y bloques (elemento de software) son ramas del árbol, mientras que los edificios (métodos) son las hojas; mientras que en el segundo enfoque los distritos (paquetes), subdistritos (sub-paquetes), bloques (sub-paquetes del último nivel) y edificios (elemento de software) son ramas del árbol, y los ladrillos (métodos) son las hojas.

En la versión de “software city” de Wettel y Lanza el anidamiento de los distritos, subdistritos y bloques es resaltado mediante el uso de elevaciones, en donde la elevación más alta representa los elementos que se encuentra más cerca de las hojas en la estructura del árbol. La Figura 6(a) muestra el uso de niveles para representar el anidamiento de los elementos en un sistema (sub-paquetes que forman parte de otros sub-paquetes o paquetes); la Figura 6(b) ilustra la representación de los métodos en una clase y la figura 6(c) muestra la representación completa de un sistema de software.

Las principales ventajas de “software city” es la escalabilidad y capacidad para visualizar la estructura y métricas de sistemas de software de gran tamaño. Asimismo este tipo de visualización brinda información de gran valor a primera vista y permite la exploración y obtención de detalles adicionales de forma rápida y simple mediante el uso de técnicas de interacción.

Es relevante tener en cuenta que existe un gran número de trabajos de investigación que han propuesto variantes adicionales de “software city”, pero la mayoría son herramientas standalone para la visualización de sistemas desarrollados en Java. Entonces llama la atención el trabajo desarrollado por Limberger *et al.* [57] que utilizó tecnologías web para desarrollar una herramienta que usa este tipo de re-

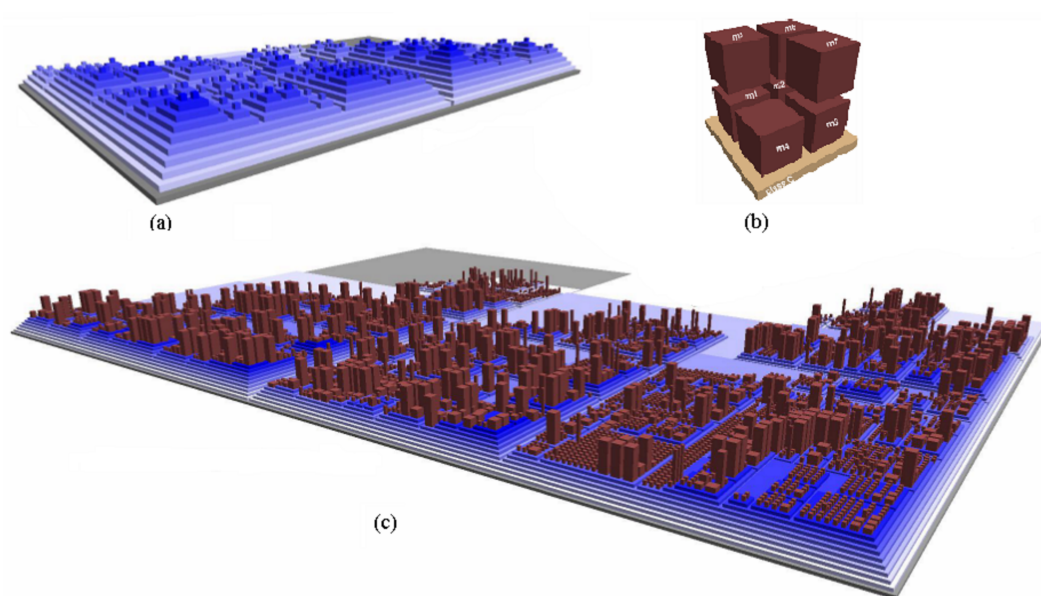


Figura 6: Visualización usando una metáfora de ciudad [84, 85]. (a) Uso de niveles para representar elementos contenidos por otros elementos. (b) Representación visual de los métodos en una clase usando figuras de ladrillos. (c) Visualización de un proyecto de software completo.

presentación. La utilización de tecnología tiene como ventaja que la herramienta es independiente del hardware del dispositivo en que se ejecuta y solo requiere de un navegador de Internet (en un ordenador convencional o tableta, por ejemplo).

También cabe resaltar que “software city” permite representar cualquier tipo de estructura jerárquica, por lo que es una representación que puede ser utilizada para visualizar cualquier sistema de software desarrollado con un lenguaje que cuente con una estructura de este tipo. El trabajo realizado por Bentrand and Melasti [9] muestra las posibilidades de este tipo de visualización para representar los elementos de un sistema de software desarrollado con AspectJ, un lenguaje orientado a aspectos.

La investigación llevada a cabo por Bentrand and Melasti propone una herramienta que se integra a Eclipse, como un plugin. En este caso “software city” es usada para la visualización de la estructura de un sistema de la siguiente forma: los paquetes son representados como distritos, y los aspectos y elementos de software como edificios. Al igual que las otras implementaciones de “software city”, el ancho, alto y color de los edificios son usados para la representación de métricas.

3.2.3. Diseños basados en matrices

El diseño basado en matrices es un tipo de visualización que ha sido utilizado de forma amplia para representar la estructura de sistemas por su escalabilidad y facilidad de comprensión. Sangal *et al.* [77] proponen Lattix, una herramienta que extrae las dependencias entre los elementos directamente del código fuente mediante análisis estático y que utiliza como método de visualización [Dependency Structu-](#)

re Matrix (DSM). El método consiste en colocar los elementos en las filas de la matriz y enumerarlas, para luego generar una correspondencia entre las filas y las columnas indicando en la casilla que se crea con la intersección de estas el número de dependencias entre los elementos, según corresponda. La Figura 7(a) muestra la dependencia entre elementos marcando una “X” en la casilla correspondiente.

La jerarquía de la estructura del sistema (paquetes y elementos de software) se representa usando un número variable de columnas que está en función del número de niveles de la profundidad del árbol de la estructura: cada columna es usada para ubicar los elementos del nivel correspondiente a la jerarquía como muestra la Figura 7(b). La estructura del árbol se puede expandir y contraer según las necesidades del usuario de acuerdo al espacio en pantalla. Cuando la visualización está representando la estructura a nivel de paquetes indica el número de dependencias que existe entre paquetes, como se puede ver en la Figura 7(b). Sin embargo, cuando la estructura ha sido expandida hasta el último nivel de la jerarquía, donde se encuentran los elementos de software, los valores de las dependencias entre elementos serán igual a 1, como se puede observar en la Figura 7(c), porque la relación de dependencia es entre elementos atómicos del sistema.

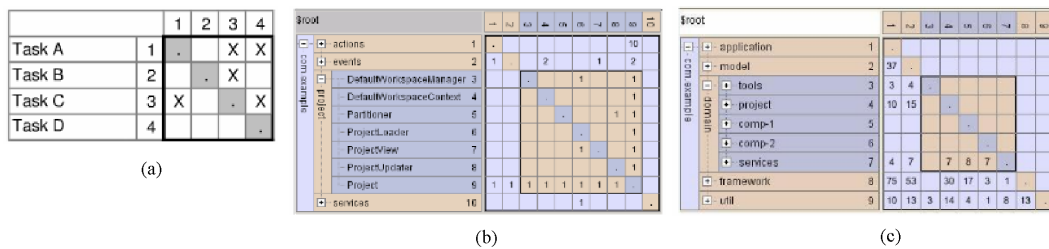


Figura 7: Características de visualización de Lattix. (a) Correlación de dependencias entre tareas (elementos de software). (b) Característica expandible de la visualización y número de dependencias de los elementos de software contenidos en los paquetes. (c) El paquete "project.es" expandido y representa a los elementos de software que contiene y las dependencias en las cuales los elementos están involucrados. [77]

Lattix proporciona la posibilidad de especificar reglas para describir las dependencias consideradas como aceptables de acuerdo al diseño del sistema. La visualización hace uso de dichas reglas para mostrar posibles violaciones al diseño, usando para ello marcas de colores que son colocadas en las celdas que se crea por la intersección entre los elementos: las marcas de color verde indican que se puede establecer una relación de dependencia, las marcas de color negro muestran una prohibición para establecer una dependencia entre los elementos y las marcas de color rojo indican la violación a las reglas de dependencias (ver Figura 8).

La visualización de múltiples tipos de relaciones y dependencias entre los elementos de software es un reto difícil de abordar. Los grafos tienen limitaciones para señalar varias relaciones a la vez y además sufren de problemas de escalabilidad para representar varios cientos de relaciones. Un enfoque que se basa en una matriz de adyacencia y que permite representar varios tipos de dependencia a la vez es el propuesto por Abuthawabeh *et al.* [1] que fue nombrado como **Interactive Multi-Matrix**

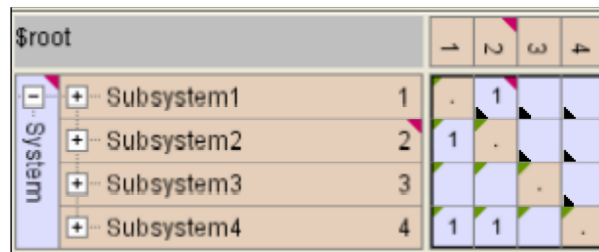


Figura 8: Reglas de diseño: marcas de dependencias permitidas, no permitidas y violaciones al diseño del sistema [77].

Visualization (IMMV). Esta visualización utiliza una representación *icicle-plot* para la estructura del sistema y divide las celdas de la matriz en sub-celdas que se rellenan de diferentes colores para representar distintos tipos de dependencia. Entonces cuando existe un tipo de dependencia se rellena un sub-celda con el color que corresponde a ese tipo de dependencia, como muestra la Figura 9.

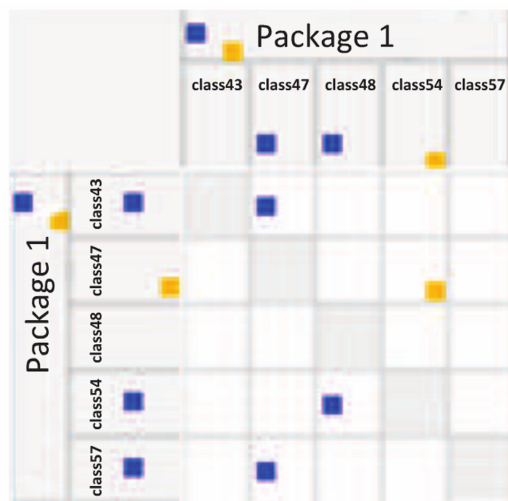


Figura 9: Relaciones de dependencia con IMMV [8].

3.2.4. Otros tipos de visualizaciones

Una alternativa al uso de **DSM** para la representación de dependencias es la visualización propuesta por Abuthawabeh *et al.* [1] que fue denominada como **Parallel Node-Link (PNL)**, y que permite la representación de n tipos de dependencias, cuyo límite es fijado por el espacio de visualización de la pantalla.

La estructura del sistema es visualizada por **PNL** mediante una representación del tipo *icicle-plot*, la cual es colocada en el lado izquierdo de la visualización, de forma similar a como lo hace **DSM**. Para mostrar las relaciones de dependencia entre los elementos de software, **PNL** representa el último nivel de la estructura del sistema (según como haya sido expandido el árbol por la interacción que ha realizado el usuario) mediante estructuras paralelas, una para cada tipo de dependencia.

Entonces las dependencias se muestran mediante líneas que enlazan los elementos en la estructura representada por el *icicle-plot* y los elementos en las estructuras paralelas, como muestra la Figura 18.

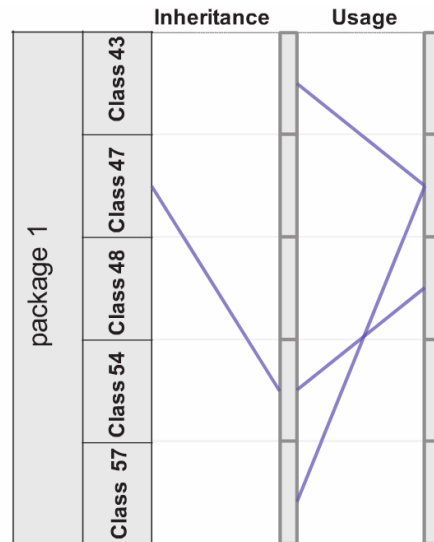


Figura 10: Visualización de relaciones de dependencia con PNL [8].

3.3. Visualización de la evolución de la arquitectura de los sistemas de software

La comprensión de un sistema de software requiere comprender los cambios y la evolución, y por lo tanto también es necesario comprender la evolución de la arquitectura y los cambios en las métricas asociadas a los elementos que la componen. En línea con esto, Wettel y Lanza [85] apoyan la comprensión de la evolución usando “software city” con dos niveles de representación: el sistema en general y los elementos de software en particular. Para representar la evolución del sistema y los elementos de software utilizan una sucesión de visualizaciones que permiten notar los cambios que han ocurrido entre una y otra revisión. Como se puede ver en la figura 11 entre una y otra revisión del sistema han cambiado los elementos señalados por las flechas y el círculo de color negro. Mientras que en el caso de un elemento de software específico, para cada revisión se muestra el tamaño y antigüedad de los métodos, por medio de la altura del edificio y haciendo uso de colores (el color más oscuro representa el método más antiguo), como se muestra en la figura 12. En general, los dos métodos mencionados cuentan con limitaciones de escalabilidad para representar un gran número de revisiones, incluso para sistemas de pequeña y mediana escala.

3.3.1. Metáforas de ciudad

La implementación de “software city” que fue realizada por Wettel y Lanza [85] calcula la estructura del sistema para todas las revisiones del sistema desde su crea-

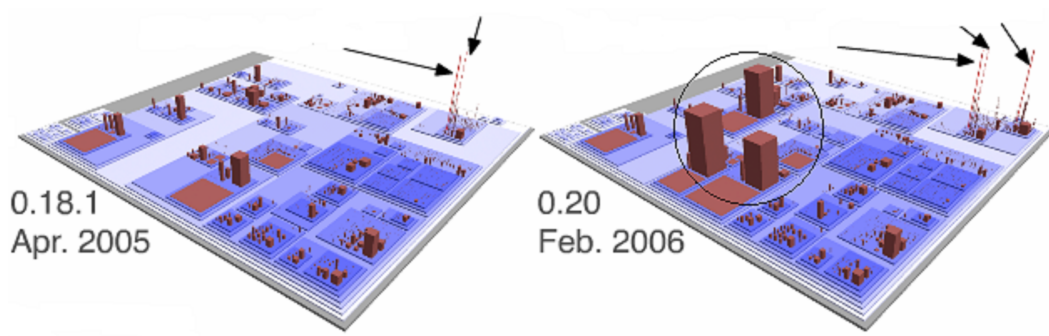


Figura 11: Visualización de dos revisiones de un sistema de software [85].

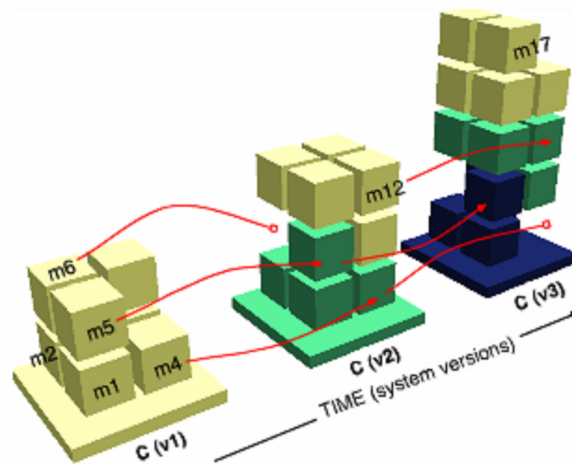


Figura 12: Representación visual de la evolución de un elemento de software y sus métodos [85].

ción hasta un punto determinado en el tiempo, luego grafica el mapa que es producido con base en el cálculo realizado. Por lo que la visualización es utilizada como un espacio de exploración para ver los cambios que han ocurrido entre revisiones o periodos de tiempo, pero no contempla la posibilidad de incorporar cambios a la estructura después de que el mapa ha sido calculado. Para incorporar cambios podría ser necesario reorganizar la visualización y cambiar las posiciones de los elementos. Esto implica que el modelo mental que los usuarios se hayan formado sobre la estructura del proyecto se vería alterada y también acarrearía dificultades para realizar comparaciones entre las revisiones y periodos de tiempo posteriores a las comprendidos en el cálculo del mapa inicial.

Las dificultades mencionadas anteriormente fueron puntualizadas por Steinbrückner and Lewerentz [80] al sugerir que la visualización de “software cities” que están orientadas a representar la evolución de los sistemas sufren de problemas al lidiar con la representación de cambios en la estructura de los sistemas de software. Por esa razón propusieron una visualización denominada como *EvoStreets*, que también se basa en la metáfora de una ciudad que se construye conforme el sistema evoluciona.

El concepto de *EvoStreets* gira en torno a las calles de la ciudad. En esta vi-

sualización la calle principal representa al sistema completo, las calles secundarias representan a los paquetes o sub-paquetes y los edificios a los elementos de software. El ancho de las calles es inversamente proporcional a la profundidad que tiene el paquete o subpaquete en la jerarquía de la estructura del sistema. El tamaño de los edificios es utilizado, al igual que en las otras implementaciones de “software city”, para representar propiedades o métricas de los elementos de software. Los principales elementos que representa *EvoStreets* son los paquetes, clases, herencia, dependencias, tipo y tamaño de los elementos de software, entre otros.

Para construir la representación visual de *EvoStreets* se toma punto de partida la estructura de la primera revisión del sistema de software o bien la estructura que se obtiene del análisis acumulado de las revisiones que se han realizado durante un periodo de tiempo, el cual puede tener como final el momento actual. Los elementos de la estructura inicial son fijados en unas determinadas coordenadas que no cambian durante la evolución del sistema, por lo que los nuevos elementos que se agreguen a la estructura se acomodan como parte de una calle existente o se agregan a una nueva calle. La visualización crece del centro hacia afuera, por lo que las calles que se desprenden de la calle principal se alargan hacia la periferia de la visualización y se le agregan calles secundarias conforme se crean nuevos sub-paquetes. Las calles también se van alargando conforme se van agregando elementos de software a ambos lados de estas (ver Figura [80]). Los elementos una vez que son agregados no son movidos a otra posición, cuando un elemento es removido en la estructura se resalta como tal pero no se elimina y cuando un elemento es cambiado a una nueva posición es resaltado como removido (pero no se mueve el elemento gráfico) y se agrega en la nueva posición como un nuevo elemento. Previo a la propuesta de *EvoStreets* un enfoque similar en 2D había sido propuesto en [28] como parte de los esfuerzos realizados a inicios de la investigación de esta trabajo. La figura 14 muestra una secuencia de la evolución de una estructura con el diseño de visualización que fue propuesto.

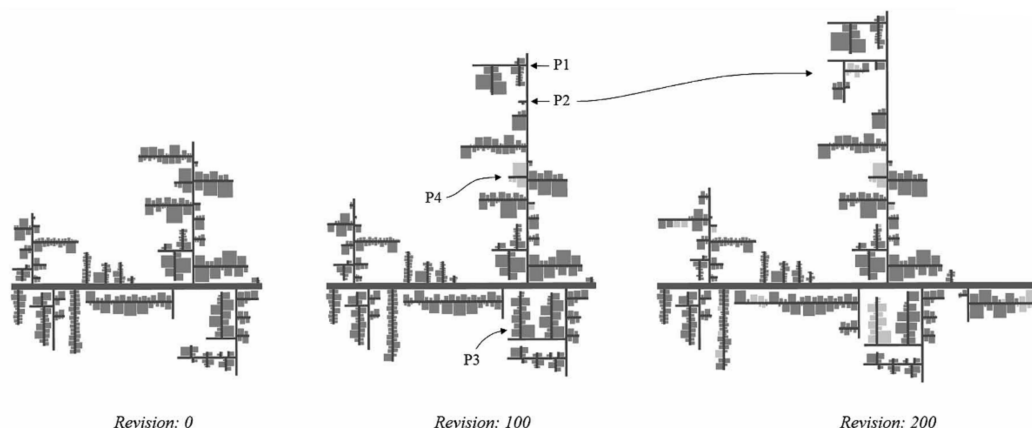


Figura 13: *EvoStreets*: Evolución de la estructura de un sistema de software [80].

La representación de la evolución se hace mediante el uso de niveles y contornos: los elementos más antiguos se ubican en los niveles superiores y los más nuevos en los niveles inferiores. La Figura 15 muestra el uso de niveles en la visualización: el

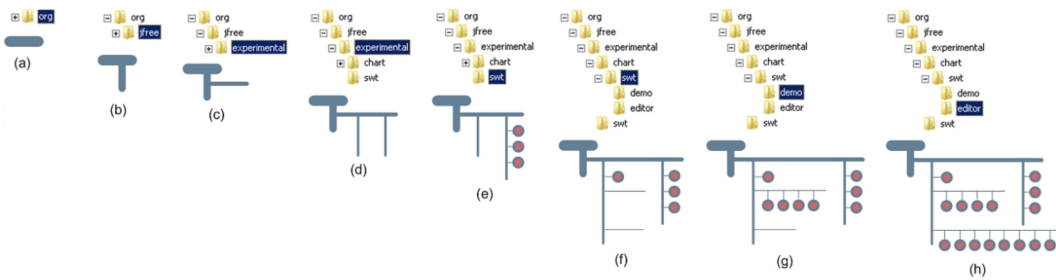


Figura 14: Estructura de árbol H-V para la visualización de la estructura de sistemas de software [28].

paquete P es agregado a la estructura (a la izquierda de la figura) y se puede observar el efecto en los niveles que produce la adición del nuevo elemento (a la derecha de la figura).

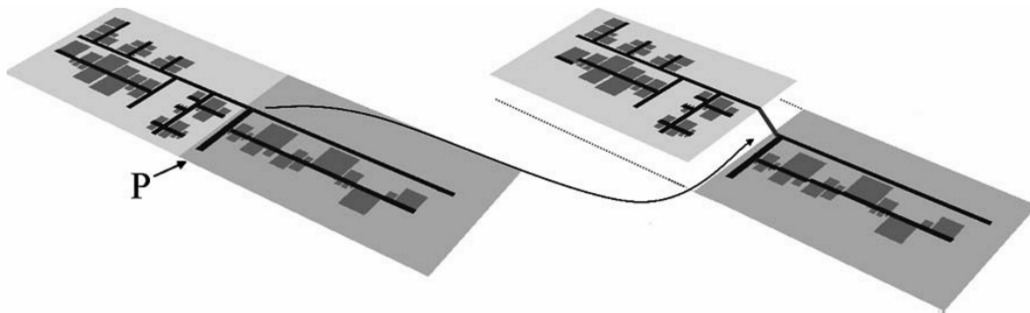


Figura 15: EvoStreets: Uso de niveles en la visualización para mostrar cuando se agrega un nuevo paquete [80].

La altura de los edificios, el ancho de los edificios, el uso de varios colores en los edificios, y el uso de los colores en general son utilizados en *EvoStreets* para indicar propiedades (ver Figura 16) como nombres de los programadores y volumen de cambios que han realizado, cobertura de las pruebas del sistema y métricas, por ejemplo.

3.3.2. Diseños basados en matrices

En línea con lo anterior, Beck y Diehl [8] proponen una visualización basada en DSM para encontrar las diferencias de la estructura y las dependencias de los elementos de software entre dos revisiones. Para visualizar la estructura del sistema se utilizan el lado izquierdo y la parte superior de la matriz, con la estructura correspondiente a una revisión en cada una de esas áreas de la representación matricial. Las posibles diferencias entre las estructuras son gestionadas mediante un algoritmo que ordena los elementos tomando en cuenta los ascendentes y parientes que no son comunes entre una y otra estructura [8]. Una vez que las estructuras han sido representadas se realiza la comparación entre revisiones utilizando un código de colores: cuando existe una relación de dependencia entre dos elementos, la celda que

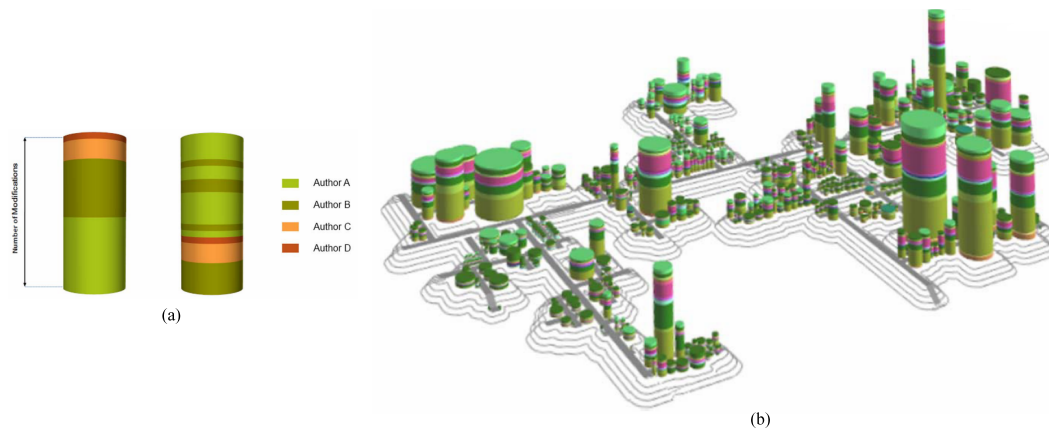


Figura 16: EvoStreets: Las propiedades de los elementos de software se representan con el ancho, alto y color de los edificios [80].

representa dicha relación es señalada utilizando un color determinado. El código de colores se compone del azul, púrpura y rojo. El color azul es usado para señalar las dependencias que existen en la primera revisión, mientras que el color púrpura se utiliza para indicar las dependencias existentes en la segunda revisión y el color rojo para mostrar las dependencias que son comunes a ambas revisiones (ver Figura 17).

3.3.3. Animación

Yarn es una visualización que representa la evolución de la arquitectura de los sistemas de software a partir de los cambios en el código fuente y haciendo uso animación [36]. Esta visualización consiste de un grafo cuyos nodos representan a los elementos de software y cuyas aristas utilizan pesos para indicar el número de dependencias entre los elementos. La evolución del sistema se presenta como un grafo animado que conserva la posición de sus elementos y muestra de forma progresiva los cambios que ocurren en las dependencias.

La animación utilizada por Yarn puede enfatizar la acumulación de los cambios o solo los cambios que ocurrieron de forma reciente. Para ello hace uso de los colores y el grosor de las aristas. La visualización permite conocer las dependencias de la evolución completa con una vista que muestra el acumulado de las dependencias, la cual toma en cuenta el intervalo de tiempo entre el instante en que estas fueron creadas hasta el momento en que fue creada la última revisión del sistema. Además, esta representación permite obtener información sobre las dependencias que han cambiado de forma reciente utilizando colores para resaltarlas y opacando las dependencias más antiguas. De forma adicional conforme la animación avanza muestra información sobre el número de revisión y la fecha en que ha sido creada (ver Figura).

Entre las limitaciones que presenta Yarn se encuentra la falta de escalabilidad para representar sistemas de gran tamaño, así como la oclusión al intentar representar un gran número de relaciones. A lo cual se debe agregar las limitadas capacida-

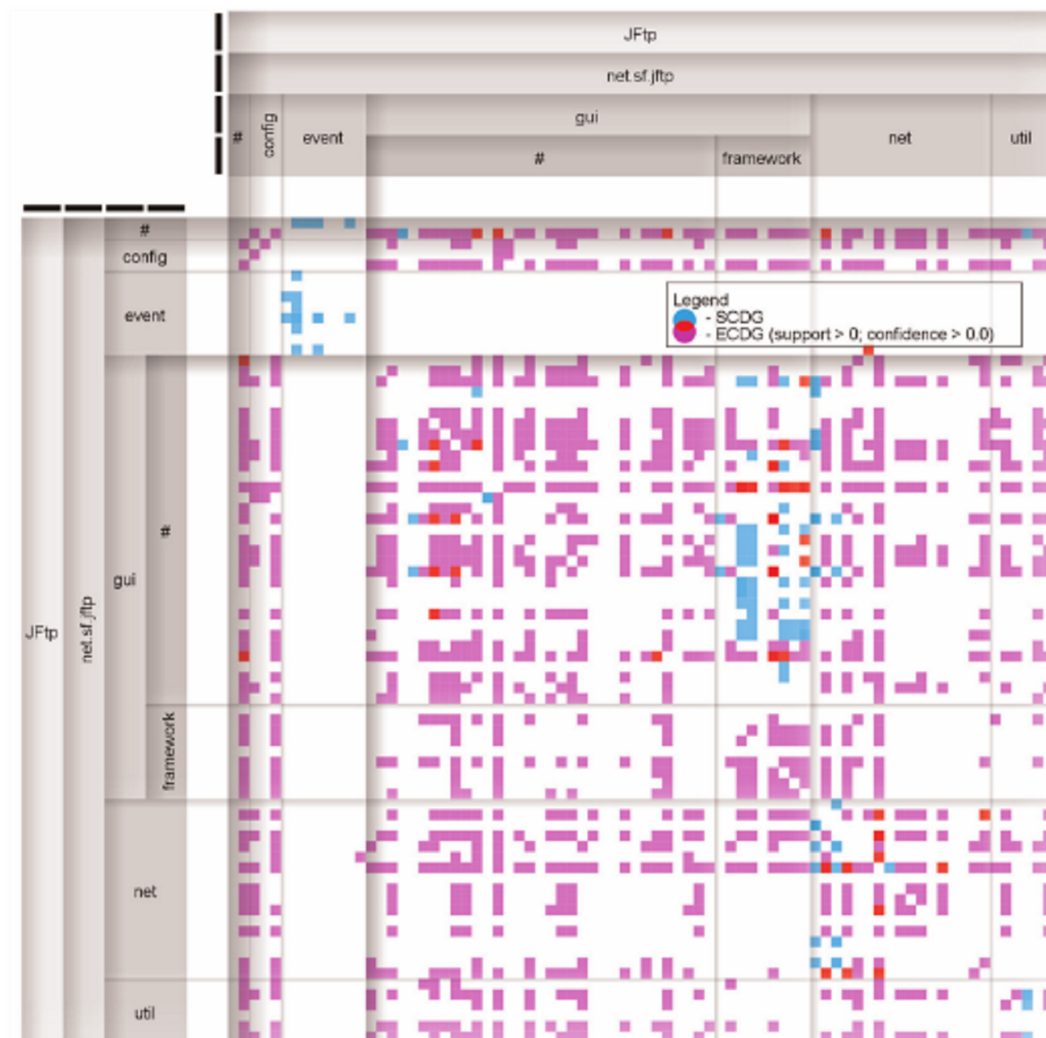


Figura 17: Comparación de dependencias para dos revisiones de un sistema de software [8].

des de interacción al ser una animación que ofrece información en su forma final que no propociona la posibilidad de navegación y descubrimiento de interactivo de conocimiento.

3.3.4. Cartografía de software

Khun *et al.* [50] al igual que otros investigadores señalan la necesidad de conservar la configuración de las posiciones de los elementos de la visualización en el tiempo, con el fin de hacer posible la comparación de información. Por lo que tomando esto en cuenta propusieron una visualización a la cual llamaron *Software Cartography*. En esta visualización los elementos de software son representados por gráficos cuya posición dentro del espacio es calculada utilizando como base la similitud y el uso del vocabulario de los elementos de software. De acuerdo con los investigadores, el léxico que utilizan los sistemas de software tiende a crecer con el tiempo, pero no cambia

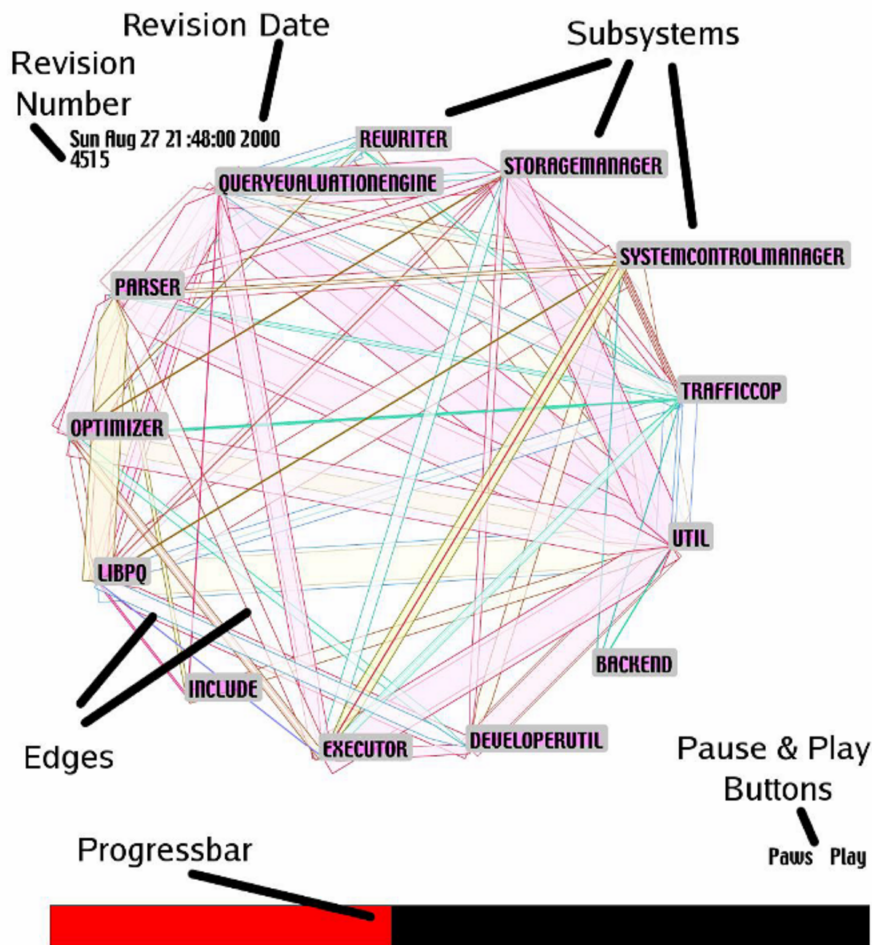


Figura 18: Visualización animada de la evolución de la arquitectura de un sistema de software utilizando Yarn [36].

de forma tan dinámica como su estructura. Lo cual permite crear una visualización robusta y consistente que permite representar diferentes aspectos de los sistemas de software, incluida su evolución. Esto facilita la comprensión entre diferentes vistas porque los usuarios conservan el mismo modelo mental del sistema.

La construcción del mapa de *Software Cartography* se puede llevar a cabo procesando de forma inicial todas las revisiones disponibles o utilizando un enfoque incremental revisión por revisión. Con el primer método se calcula el mapa de toda la evolución, y de cada revisión, una vez que todos los datos han sido procesados, mientras que el segundo método calcula el mapa de forma acumulativa conforme cada revisión es procesada. En cualquiera de los dos casos el resultado es un mapa consistente que conserva la posición de los elementos durante toda la evolución del sistema.

El procesamiento de los datos para crear la visualización toma en cuenta los términos que aparecen en archivos fuentes del sistema, los cuales son colocados en una matriz de ocurrencia de términos para luego se indexados y clasificados de

acuerdo a su frecuencia de uso. Los términos del léxico incluyen los nombres de las clases, métodos, parámetros, variables, métodos invocados, palabras en comentarios y valores literales. De forma posterior se calcula la distancia entre los elementos del sistema utilizando como base la similitud de los términos.

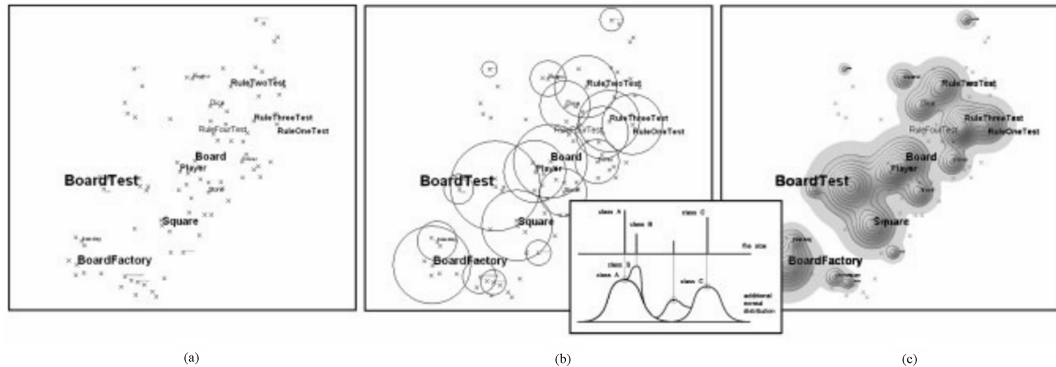


Figura 19: Proceso de construcción de un mapa de un sistema de software con *Software Cartography* [50]. (a) Colocación de elementos en el plano de acuerdo a la distancia de los términos. (b) Área de influencia de los elementos según su proximidad y tamaño medido por el número de líneas. (c) Altura de los montículos calculada utilizando como referencia el tamaño de los elementos del sistema.

En la Figura 19 se puede apreciar que durante el proceso de construcción de un mapa de *Software Cartography* primero se colocan en el plano los elementos de acuerdo a la distancia entre los términos, luego se determina su área de influencia de acuerdo a su proximidad con otros elementos y tamaño del archivo, y finalmente se calcula la altura del montículo con base en el tamaño del archivo o clase. La altura del montículo cambia conforme el tamaño del elemento se reduce o aumenta, pero en cualquier caso la configuración de los elementos en el plano no se altera. La Figura 20 muestra la sucesión de cuatro revisiones de un sistema en las que se puede apreciar la consistencia de la visualización, y las variaciones en las áreas de influencia y altura de los elementos.

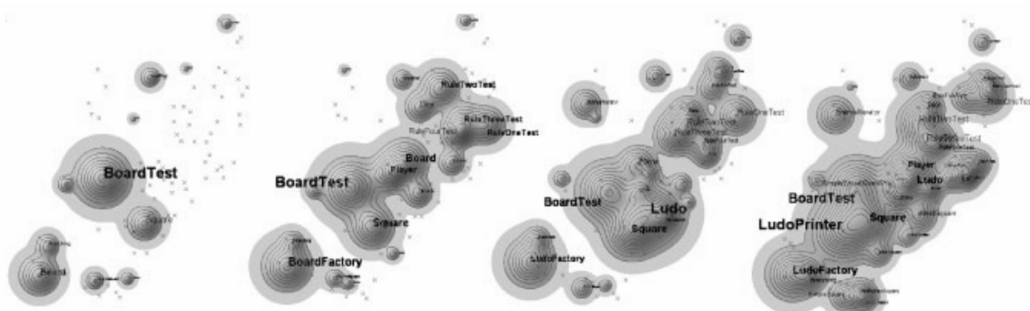


Figura 20: Representación de la sucesión de revisiones de un sistema utilizando *Software Cartography* [50].

Software Cartography es una herramienta que se integra como un plugin de Eclipse para proporcionar presencia continua y fácil acceso al programador en su entorno, de acuerdo con sus autores [51] persigue los siguientes objetivos:

- * Permitir la exploración y rápida comprensión del sistema.
- * Facilitar la comparación de las métricas.
- * Apoyar la construcción de un marco de comprensión común del sistema y facilitar la colaboración entre los miembros del equipo de desarrollo al permitir la conexión de dos o más programadores y sus entornos de programación (IDEs) para brindar información sobre las actividades que se llevan a cabo en los elementos de software.

Una característica en común de estas visualizaciones es su escalabilidad en la visualización de estructuras, dependencias y métricas.

3.4. Actividades de los programadores y colaboración

Las herramientas SCM son utilizadas de forma amplia por los departamentos de desarrollo de software y, por lo tanto, también por los programadores. Esto implica que se registran datos sobre las actividades y modificaciones de código que realizan los programadores, lo cual refleja la posible comunicación que tiene lugar entre ellos a partir de los elementos que cambian. Adicionalmente, la información almacenada por las herramientas SCM se puede acceder y analizar utilizando mecanismos automáticos. Por estas razones un gran número de investigaciones hacen uso de los datos almacenados por estas herramientas.

Los aspectos que las herramientas de visualización buscan apoyar con la creación de espacios de conocimiento compartido (para facilitar la comunicación, colaboración y control) durante el proceso de desarrollo y mantenimiento de sistemas de software son diversos. Mientras algunas de estas herramientas tienen por objetivo brindar información a los programadores otras tienen como fin apoyar a los administradores de proyectos en la toma de decisiones. Por lo que toman en cuenta factores como el estado de los proyectos en términos de calidad (medida con el uso de métricas), los cambios del sistema (incluyendo el código fuente, dependencias, relaciones y estructura), la comprensión de aspectos de diseño del sistema, las relaciones socio-técnicas y las relaciones de colaboración entre los miembros del equipo.

3.4.1. Trabajo en equipo

Algunas herramientas de visualización como [System Hotspots View \(SHV\)](#) [4] tienen doble propósito: por un lado brindar información sobre aspectos técnicos del sistema que faciliten su evolución (desarrollo y mantenimiento) y por otro lado apoyar la colaboración entre los miembros del equipo.

SHV es una herramienta de visualización basada en Polymetric Views que tiene por objetivo apoyar la comprensión de la estructura de un sistema de software y la detección de problemas con el uso de métricas aplicadas a los paquetes, clases y dependencias. La principal contribución de esta herramienta es el despliegue de la visualización en una gran pantalla de pared que permite compartir el conocimiento sobre el sistema para propiciar la discusión, coordinación y colaboración entre

los miembros del equipo de desarrollo. Esta herramienta, por lo tanto, puede ser utilizada tanto por los programadores como por los administradores de proyectos.

La efectividad de SHV fue probada mediante un estudio con usuarios. Los resultados del estudio mostraron que los participantes disfrutaron la visualización con el uso del gran panel visual y la posibilidad de observar un gran número de detalles a la vez, aunque echaron en falta no contar con la posibilidad de interactuar con la visualización porque las pantallas no eran táctiles. Algunos participantes hicieron observaciones en torno a la altura de las pantallas; en algunos casos porque la estatura de los usuarios era muy baja no pudieron observar de forma adecuada las partes superiores de la visualización, y en otros casos porque los participantes eran muy altos les resultó incómodo ver la información en la parte inferior de las pantallas. Otra observación realizada por los participantes tuvo relación con la interrupción de la continuación visual de la representación por los bordes de las pantallas.

3.4.2. Conciencia situacional

Mientras que *Ownership Map* [27, 30] es una visualización que brinda información a los administradores de proyectos sobre la colaboración que ha tenido lugar durante el desarrollo del sistema. El objetivo de dicha representación es apoyar la toma de decisiones y ofrece detalles sobre:

- * El número de programadores que han participado en el desarrollo del sistema.
- * Las modificaciones o partes del sistema que han sido desarrolladas por cada programador.
- * El comportamiento de los programadores durante el desarrollo y mantenimiento del sistema.

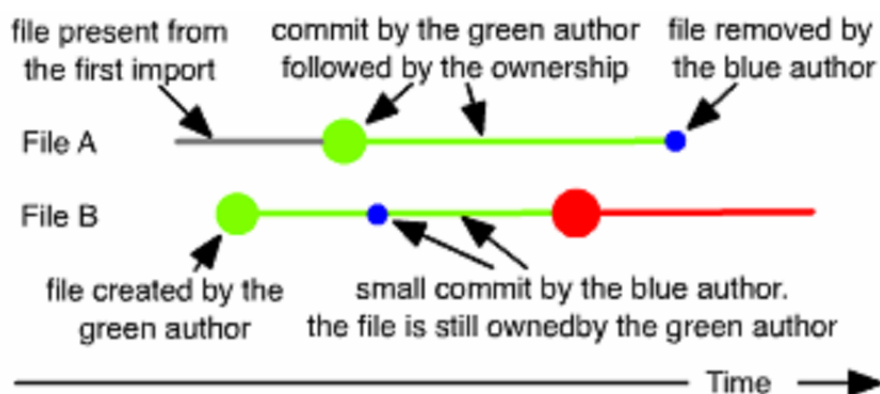


Figura 21: Representación de los cambios y la propiedad de los elementos de software en *Ownership Map* [27].

Los datos usados por *Ownership Map* se extraen de los logs de *Concurrent Versioning System (CVS)* y contemplan detalles sobre la asociación de los cambios, a nivel de líneas de código, y los programadores. Con base en esos datos se determina

cuál programador es el propietario de un elemento de software durante un intervalo de tiempo o durante la evolución completa de ese elemento. La representación de los elementos de software y programadores se lleva a cabo mediante el uso de líneas, círculos y colores. Las líneas representan a los elementos de software, los círculos la magnitud del cambio realizado y los colores están asociados a los programadores. La alternancia en el color de una línea muestra los intervalos de tiempo y programadores que han tenido la propiedad del elemento de software. En tanto, el color gris de la línea representa a un programador desconocido o la importación inicial de un elemento de software en el repositorio de software, y un círculo con el color del programador al final de una línea indica que el elemento ha sido borrado (ver Figura 21).

El uso de esta visualización permite establecer diferentes patrones que se derivan de las actividades y la colaboración entre los programadores, de acuerdo con Gîrba *et al.*. La siguiente lista muestra esos patrones y los relaciona, cuando aplica, con la Figura 22:

Monólogo: Consiste en la actividad realizada por un único programador en la mayoría de archivos y durante un periodo de tiempo, como se puede observar en el lado izquierdo de la figura, en donde los cambios fueron realizados por el programador asociado al color verde (señalado por *R5*).

Familiarización: Este patrón muestra como un programador de forma progresiva realiza cambios cada vez en más elementos de software, hasta que prácticamente toma posesión de todos los elementos, como se puede ver en el recuadro *R5* de la figura y que hace notar el comportamiento del programador representado por el color azul.

Expansión: Este patrón está asociado con la adición de nuevos archivos al sistema por parte de un programador, como es el caso del programador identificado por el color azul de acuerdo con los recuadros *R8* y *R12* de la figura.

Edición: Es el patrón asociado a cambios asociados a la limpieza de comentarios, renombrado de identificadores u otros cambios necesarios que no agregan funcionalidad al sistema. Este tipo de patrón se puede observar como una columna vertical de cambios realizados por el mismo programador, tal y como aparece indicado en la figura por los recuadros *R7*, *R11* y *R15*.

Toma de posesión: El nombre de este patrón se debe a que un programador toma posesión de la mayoría de elementos del sistema en un periodo muy corto de tiempo, como es señalado por los recuadros *R13* y *R14* en la visualización.

Trabajo en equipo: Este patrón se identifica porque un grupo de programadores toman propiedad de múltiples elementos de forma sucesiva y en un corto periodo de tiempo. En la figura se puede ver la intervención de varios programadores en múltiples periodos que son resaltados por los recuadros *R1*, *R3 – 4* y *R12*.

Corrección de errores: Consiste en una intervención puntual para corregir un error y que por los pocos cambios que implica, el programador asociado toma

la propiedad del elemento de software por un periodo muy corto, a veces imperceptible como se muestra en tres puntos de la visualización en donde un punto de color amarillo es señalado por un círculo (ver recuadro R10).

Limpieza: Es el patrón contrario al patrón **Expansión**, y consiste en la eliminación de un número notable de elementos del sistema, como indica el recuadro R2.

Silencio: Es un periodo de tiempo en el cual se muestran pocos o ningún cambio y se identifica como un rectángulo en donde los elementos de software no cambian de color.

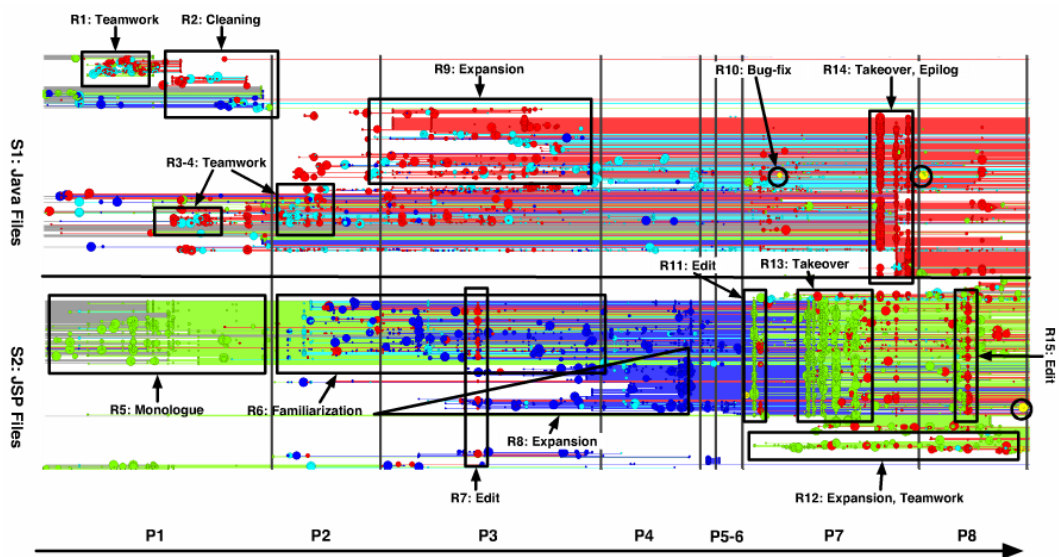


Figura 22: Visualización de los patrones de comportamiento de los programadores usando *Ownership Map* [27].

En el caso de *CodeTimeline* [52], es una herramienta de visualización que utiliza dos representaciones visuales. Una de las visualizaciones utilizadas evoluciona el concepto utilizado por *Ownership Map* al permitir que los programadores puedan agregar notas y fotos con comentarios sobre la representación, en los puntos de la evolución donde consideren relevante mantener la memoria de sucesos que han ocurrido durante el proceso de desarrollo (ver Figura 23). Mientras que la otra visualización utiliza una línea de tiempo para representar el uso de términos y su frecuencia de utilización en el código fuente del programa. Esto lo logra mediante el uso de nubes de palabras para cada versión del sistema, en donde el tamaño de las palabras y el color representan la frecuencia de uso del término. En esta visualización el color azul representa un incremento de la frecuencia del término y el color rojo una disminución, mientras que el tamaño del término refleja la magnitud de su utilización en relación los demás términos en la nube. De forma adicional esta visualización permite el uso de anotaciones como en la primera visualización (ver Figura 23).

Otra visualización que permite mostrar las actividades que llevan a cabo los programadores es un visor de actividades que Ripley *et al.* [73] construyeron sobre

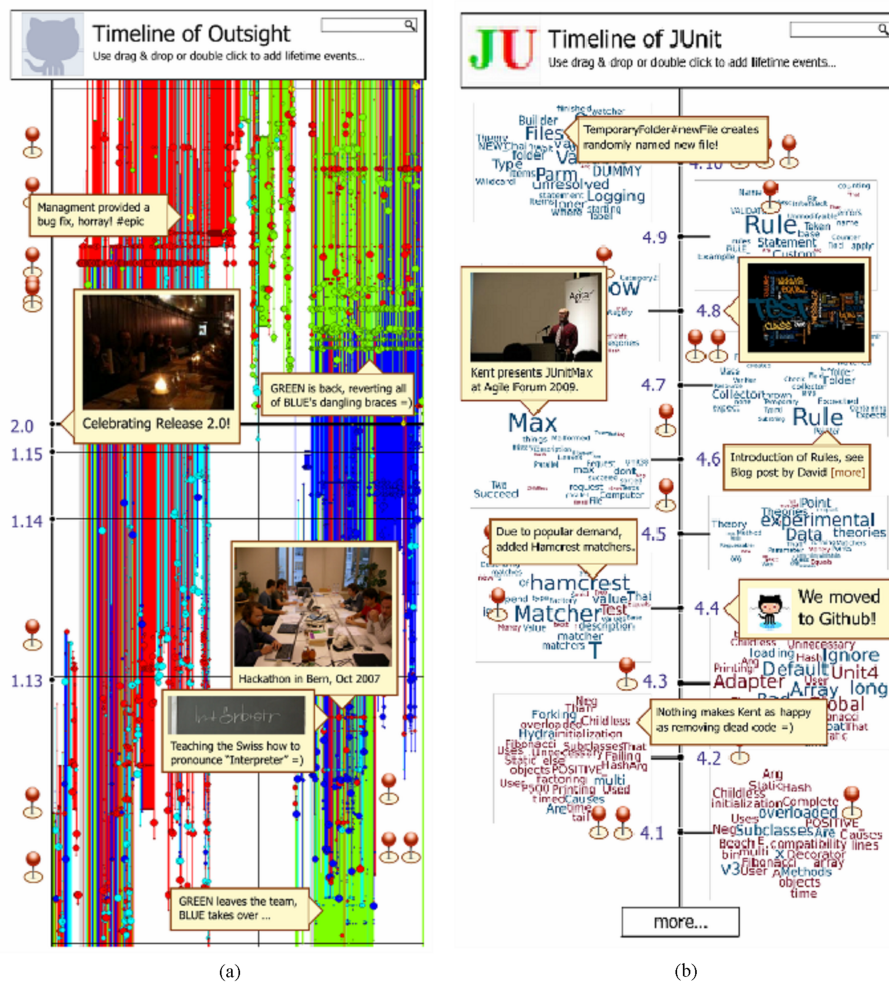


Figura 23: *CodeTimeline*: (a) Notas de del equipo de desarrollo sobre la visualización *Ownership Map*. (b) Visualización de la frecuencia de términos por versión usando nubes de palabras y notas del equipo de desarrollo. [52].

Palantír. Dicho visor fue programado utilizando tecnología 3D y utiliza como fuente de datos los logs de las herramientas SCM y de los espacios de trabajo local de cada programador. Las actividades sobre las cuales se obtiene y representa información incluye las operaciones de *check-in*, *check-out*, sincronización del espacio de trabajo local con el repositorio de software, así como la edición y borrado de elementos de software del espacio de trabajo local del programador.

Esta visualización muestra información sobre los espacios de trabajo y los elementos de software que se encuentran activos mediante el uso de dos representaciones visuales que utilizan pilas de cilindros como su principal elemento gráfico. Dichas representaciones se centran en visualizar las actividades de los programadores y en mostrar detalles sobre los cambios efectuados sobre los elementos de software.

La representación de las actividades de los programadores visualiza el espacio de trabajo local de cada programador usando pilas de cilindros, en donde cada cilindro representa un elemento de software. La altura de la pilas de cilindros reflejan el

número de actividades llevadas a cabo por los programadores (ver Figura 24). Las pilas de cilindros se desplazan al frente o hacia atrás en la visualización en función del tiempo transcurrido desde que se hicieron los últimos cambios en los espacios de trabajo que representan. Las pilas de cilindros con los cambios más recientes son ubicadas al frente de la visualización, mientras que aquellas con los cambios más antiguos, son localizadas en la parte de atrás de la visualización.

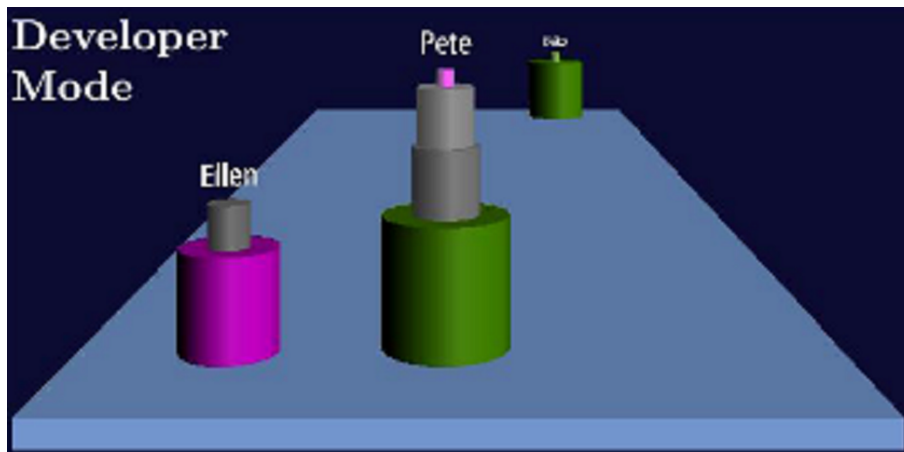


Figura 24: Visualización de las actividades llevadas a cabo por los programadores [73].

En cuanto a la representación de los elementos de software, se utiliza la pila de cilindros para representar un elemento de software y los cambios que ha realizado cada programador desde su espacio de trabajo. En este contexto cada cilindro se corresponde con un programador y su tamaño refleja la magnitud de los cambios que ha realizado, como se muestra en la Figura 25. Cabe mencionar que, en general, cuando un elemento de Palantir es seleccionado, es posible obtener información sobre la magnitud de los cambios efectuados, los nombres de los programadores y los valores de varios tipos de métricas.

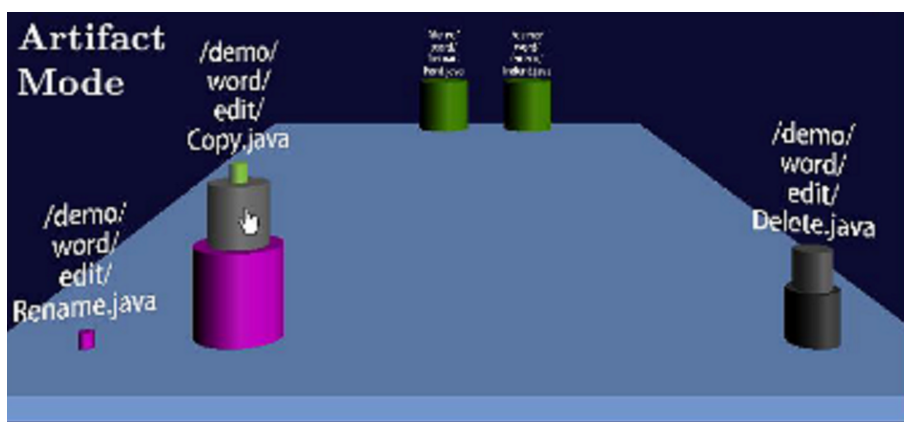


Figura 25: Representación de los elementos de software y cambios que ha realizada cada programador [73].

3.4.3. Cooperación

En la mayoría de los casos, el objetivo de las herramientas que apoyan la construcción de espacios comunes de conocimiento es mejorar la cooperación entre los miembros del equipo. Dicha cooperación usualmente es necesaria porque los colaboradores trabajan en un proyecto común y el trabajo de cada uno tiene relación con el trabajo de otros colaboradores. Pero es conveniente considerar que la cooperación entre individuos también está presente cuando varias personas aportan a la solución de determinados problemas, trabajando de forma independiente y sin tener como fin la consecución de objetivos o metas comunes. Un ejemplo notable son los sitios web utilizados por los programadores para solicitar la colaboración de otros programadores en la solución de problemas particulares [5]. Los programadores que participan en este tipo de sitios comparten la pertenencia a una comunidad de individuos que trabajan en temas sobre los cuales tienen conocimiento, pero que necesariamente no forman parte de su trabajo diario.

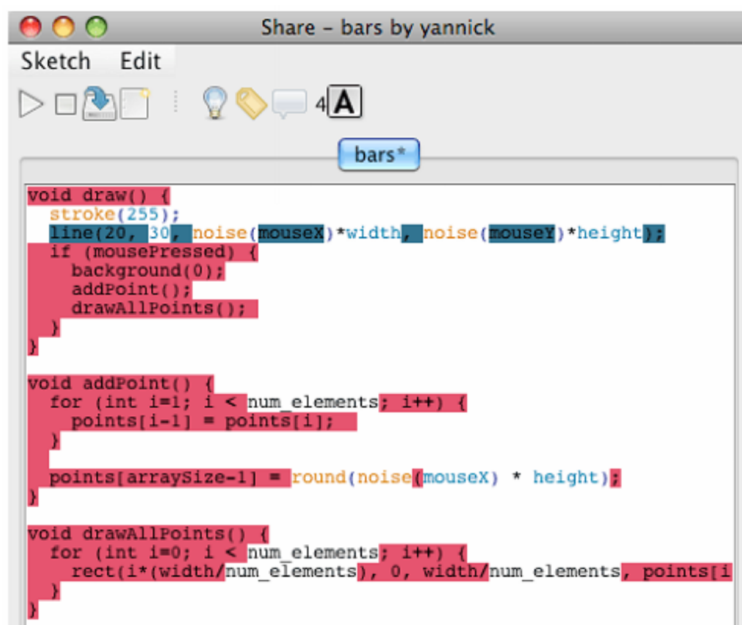
Assogba y Donah [5] dieron el nombre de “loosely bound cooperation” a este tipo de cooperación y lo definieron como “una forma de cooperación, a veces indirecta, entre miembros de una comunidad que les da libertad para perseguir sus metas individuales a la vez que les permite ayudarse entre sí”. Como un resumen de todo lo anterior y de las características que los mismos investigadores resaltan, los siguientes son los puntos centrales de este tipo de cooperación:

- * Los individuos no adquieren ningún tipo de obligación de ayudar a otros.
- * Cada participante tiene sus propias metas y de forma mayoritaria no tienen metas compartidas.
- * La cooperación puede ser desde casual hasta continua y comprometida e involucra la solución o desarrollo de un determinado elemento de software.
- * Los participantes de este modelo de cooperación forman parte de una comunidad de individuos que practican de forma activa su profesión.

Durante el trabajo de investigación realizado por Assogba y Donah desarrollaron una herramienta en Processing a la cual denominaron como Share [5]. El objetivo de esa herramienta es apoyar el intercambio de código fuente entre los miembros de una comunidad de programadores. Para su implementación utilizaron una arquitectura cliente/servidor, la cual del lado servidor proporciona la autenticación y almacenamiento de datos, mientras que del lado cliente es una aplicación de escritorio en donde el usuario realiza las tareas de programación. A cada programador que utiliza esta herramienta se le asigna un color que lo representa en todos los proyectos que participa.

El lado cliente de Share proporciona un navegador de archivos, un editor de programas, un gestor de referencias, un buscador, la visualización de la red de relaciones (navegador de relaciones) y mecanismos para la sincronización con el servidor. Para los efectos de esta investigación resultan de interés el editor de programas y la visualización de la red de relaciones, por lo que a continuación son explicados.

El editor de programas proporcionado por Share utiliza el color asignado a cada programador para indicar quien es el autor de cada trozo de código reutilizado que forma parte de un programa, pero no usa ningún color para el código nuevo que ha sido desarrollado en ese programa (ver Figura 26).



```

void draw() {
  stroke(255);
  line(20, 30, noise(mouseX)*width, noise(mouseX)*height);
  if (mousePressed) {
    background(0);
    addPoint();
    drawAllPoints();
  }
}

void addPoint() {
  for (int i=1; i < num_elements; i++) {
    points[i-1] = points[i];
  }
  points[arraySize-1] = round(noise(mouseX) * height);
}

void drawAllPoints() {
  for (int i=0; i < num_elements; i++) {
    rect(i*(width/num_elements), 0, width/num_elements, points[i]
  }
}

```

Figura 26: Share: Editor de texto que muestra los trozos de código que han sido reutilizados y representan, mediante el uso de colores, quien ha realizado el aporte original [5].

El navegador de relaciones de Share tiene como fin facilitar el rastreo de la reutilización de código fuente, para lo cual utiliza un grafo que representa las correspondencias usuario – usuario, usuario – elemento de software y elemento de software – elemento de software. Esta representación visual tiene dos variantes que permiten determinar con facilidad quien ha contribuido con código a un proyecto y quienes han reutilizado ese código.

La primera variante del navegador de relaciones utiliza una configuración radial que ubica como elemento central aquel que ha sido seleccionado por el usuario, y proporciona una vista general de todas las contribuciones efectuadas usando la herramienta. La Figura 27 muestra las contribuciones realizadas por cada programador y la reutilización de código fuente entre elementos mediante el uso de flechas, en donde una flecha indica el elemento desde el cual se ha reutilizado código. La segunda variante del navegador busca brindar información sobre cuales elementos de software están prestando o tomando código prestado de un determinado elemento, como es ilustrado por la Figura 28.

3.4.4. Relaciones socio-técnicas

La visualización de la colaboración entre programadores permite que un programador obtenga información sobre quienes le pueden prestar colaboración a partir de

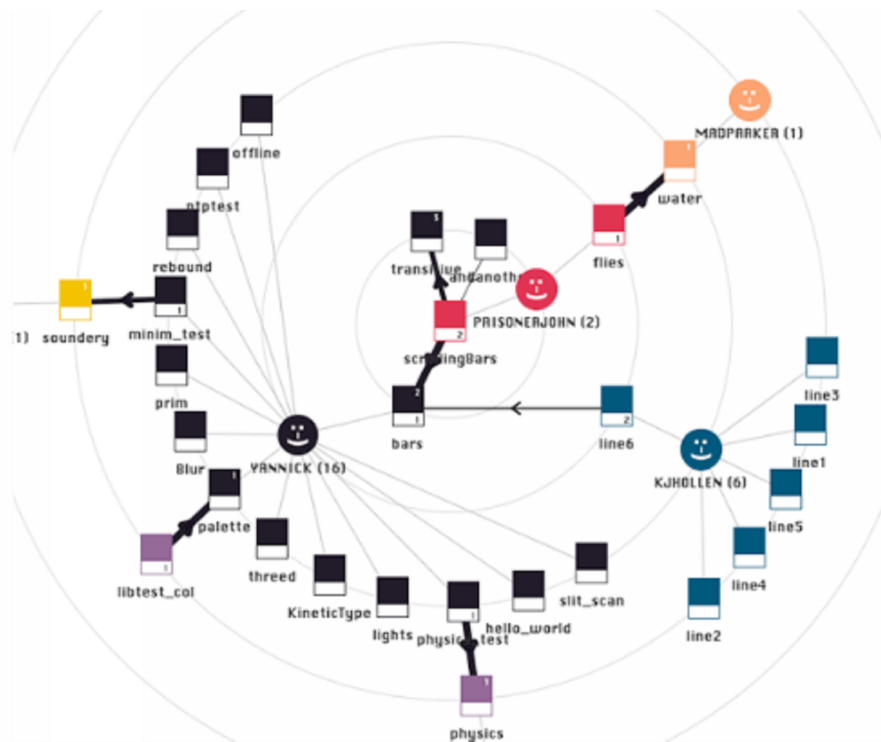


Figura 27: Share: Navegador de relaciones utilizando una configuración radial para mostrar las relaciones entre elementos con base en la reutilización de código fuente [5].

los elementos que han cambiado y las relaciones que existen entre ellos. Este tipo de visualización ofrece información a los administradores de proyectos para tomar decisiones sobre cuál programador puede sustituir a otro programador en caso de baja por enfermedad, accidente, renuncia o despido, pero además también les puede ayudar a conformar equipos de trabajo de acuerdo a las relaciones de colaboración previa entre los individuos [37].

Jermakovics *et al.* [37] construyen la red de colaboración que tiene lugar entre los desarrolladores usando información que extraen del repositorio de software sobre los cambios que se realizan a los elementos de software. Dicha red de colaboración emerge como producto del análisis de las similitudes entre los programadores con base en los elementos de software que han cambiado en común.

La red que resulta del análisis de los cambios es visualizada haciendo uso de un grafo de fuerzas, en donde los nodos representan a los programadores y las aristas reflejan las relaciones entre ellos, en función con la similitud. El tamaño de los nodos es usado para representar el número de “commits” realizados por los programadores, mientras que las fuerzas del grafo son calculadas de acuerdo con la similitud entre los programadores y el número de conexiones entre ellos. La medida de similitud también es usada como criterio de filtrado, lo que permite que el usuario pueda elegir un umbral para filtrar aristas que no reúnan el criterio seleccionado. Otro elemento interesante de esta representación es que además de ofrecer información

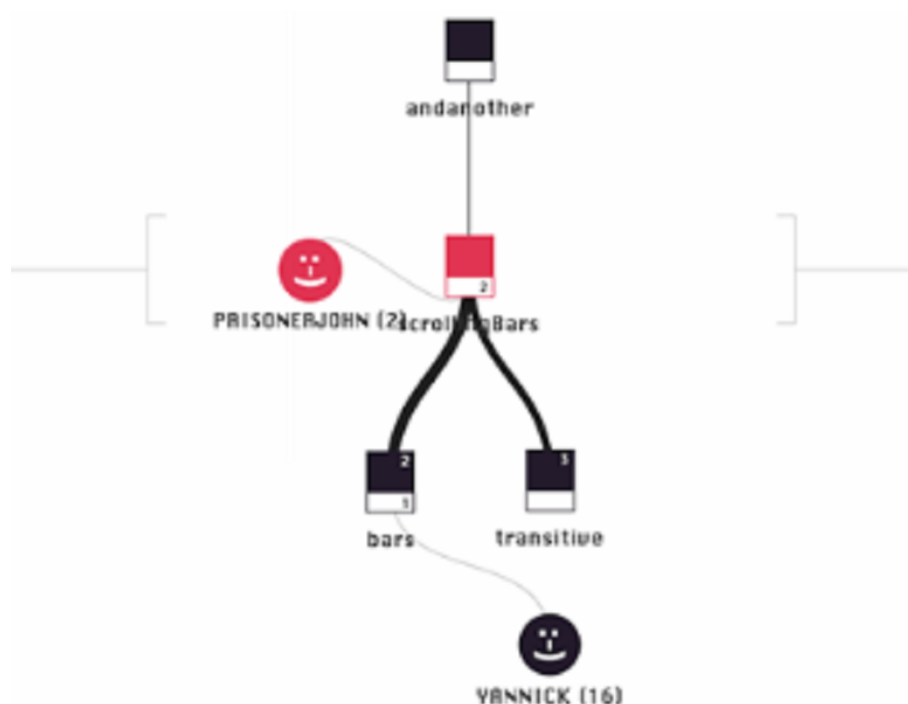


Figura 28: Share: Navegador de relaciones simplificado para mostrar las relaciones de un elemento particular [5].

sobre la relación entre programadores también brinda detalles sobre la pertenencia de los programadores a un grupo de trabajo y además la relación entre esos grupos de trabajo, como ilustra la Figura 29.

Una propuesta complementaria a las anteriores es la que realizan Heller *et al.* [32] sobre la visualización de la colaboración entre programadores, pero tomando en consideración su ubicación geográfica. En esta visualización se representan datos que son obtenidos de GitHub por medio de un grafo que se dibuja sobre un mapa para mostrar las relaciones entre los programadores, lo que además permite mostrar la densidad de programadores por países o regiones.

Elaborar estrategias para incrementar el nivel de conocimiento sobre las actividades que realizan los programadores puede contribuir con la coordinación en entornos de desarrollo distribuido. Teniendo esto en consideración, conviene recordar que las herramientas SCM han sido ampliamente difundidas y utilizadas para contribuir en la coordinación del desarrollo paralelo de software y que en los entornos distribuidos han sido de gran utilidad. Además, por la riqueza y gran cantidad de información que gestionan, un gran número de herramientas de visualización hacen uso de esa información. Pero las herramientas SCM tienen como desventaja que los programadores se dan cuenta de las modificaciones de los demás programadores hasta que han sido enviadas al repositorio de software mediante una operación “check-in”, y no en el momento en que se realizan los cambios [54].

Considerando lo anterior, Lanza *et al.* proponen una arquitectura que utiliza un plugin de Eclipse para grabar y transmitir los cambios de código que se llevan

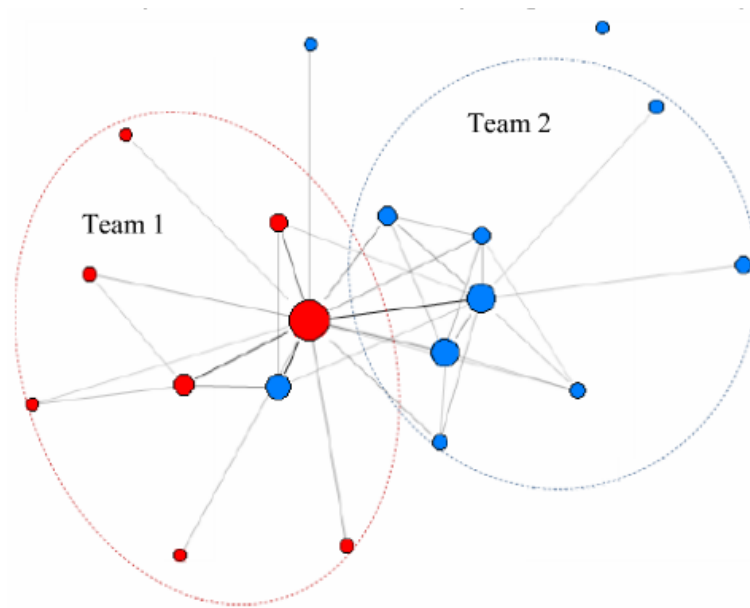


Figura 29: Visualización de una red de colaboración entre programadores a partir de los elementos de software que han cambiado en común [37].

a cabo en el entorno de un programador, a todos los programadores. Para apoyar la comprensión y reacción frente a los cambios desarrollaron otro plugin que hace uso de tres visualizaciones simples que se actualizan en tiempo real conforme llega la información de los cambios (para más detalles sobre las visualizaciones puede referirse a [54]).

4. Visualización para la conciencia situacional y la colaboración

En esta sección se introduce un framework para describir la relación de los aspectos que intervienen en la creación de conciencia situacional de los individuos y los equipos de trabajo, así como de los elementos que toman parte en los procesos de colaboración. Posteriormente se presenta GridMaster, una visualización que toma en cuenta el framework mencionado y que busca apoyar la creación de conciencia situacional tanto en los administradores de proyectos como programadores, así como facilitar la colaboración entre los miembros del equipo.

4.1. Framework: conciencia situacional y colaboración

Como se mencionó con anterioridad en este trabajo, los procesos de DMES son complejos, dinámicos, impredecibles y es común que se lleven a cabo en diferentes localidades geográficas y durante varios años. Lo que requiere que los miembros de los equipos de trabajo establezcan relaciones de colaboración de largo plazo o puntuales para realizar tareas y solucionar problemas concretos. En cualquiera de los casos cuando una tarea ha sido terminada o un problema ha sido resuelto, los individuos pasan a trabajar en lo siguiente que tienen en la agenda de acuerdo a la planificación del proyecto o en la lista de problemas pendientes de resolución. Esto conlleva que la colaboración no se lleva a cabo siempre entre los mismos miembros del equipo, sino en función de especialización y las tareas que acometen en un momento determinado.

La colaboración entre los miembros de un equipo de trabajo puede iniciar por diferentes circunstancias o situaciones, lo que implica que no existe una relación de orden entre los aspectos relacionados con ese proceso (ver la sección 2). De acuerdo con esto, el framework que se ilustra en la Figura 30 tiene como fin orientar sobre una posible configuración de las relaciones entre esos aspectos, así como definir el papel que pueden jugar los ECSs en los procesos de colaboración. En la Figura 30, las palabras de enlace compuestas (e.g. activa / apoya) sobre las líneas de enlace con flechas bidireccionales se leen siguiendo un orden descendente / ascendente. En el caso del orden descendente se inicia por el concepto en el extremo superior de la línea y se termina con el concepto en el extremo inferior (e.g. Cognición distribuida del equipo de trabajo – activa – Colaboración), mientras que en el caso del orden ascendente, se inicia a la inversa (e.g. Colaboración – apoya – Cognición distribuida del equipo de trabajo).

El framework de la Figura 30 ha sido definido tomando en consideración un enfoque de cognición distribuida en el que los miembros de los equipos de trabajo hacen uso de su cognición individual, de acuerdo a su especialización y experiencia, y la complementan con la cognición de los demás individuos para actuar en la realización de tareas y la resolución de problemas. Con base en esto, el proceso de colaboración puede ser activado por uno o varios miembros del equipo de trabajo haciendo uso

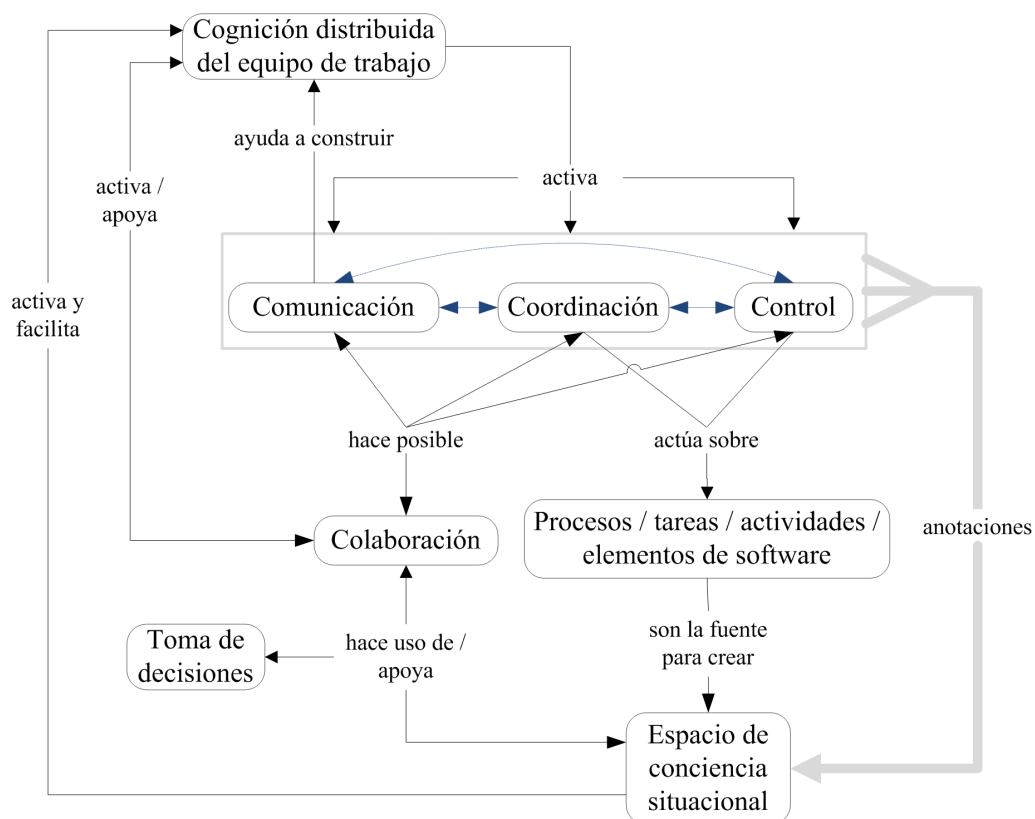


Figura 30: Framework sobre el trabajo colaborativo en los procesos de DMES.

de sus capacidades cognitivas cuando requiere llevar a cabo una tarea o detecta una situación ante la cual debe actuar.

En este contexto, para que la colaboración sea posible se puede requerir de las actividades de comunicación, coordinación o control entre los miembros del equipo de trabajo. Es importante tener presente que la colaboración entre individuos requiere de información para la distribución de tareas y para determinar las acciones que se deben seguir de acuerdo al estado del proyecto. Por lo que las facilidades e información que proporcionan los ECSs sobre la evolución y el estado de los procesos, tareas, actividades y cambios que se han realizado a los elementos de software del proyecto son de gran utilidad. La relación que existe entre el proceso de colaboración y las actividades de comunicación, coordinación y control es recíproca, como muestra la Figura 30.

De forma análoga a la activación de la colaboración, uno o varios miembros del equipo de trabajo pueden activar las actividades de comunicación, coordinación o control, para que a su vez se active el proceso de colaboración entre las personas adecuadas, y en caso necesario se haga uso de las ventajas de la utilización de los ECSs.

En este punto es relevante mencionar que el fin de los ECSs es ofrecer información en torno a los procesos, tareas, actividades y artefactos de los sistemas de software, pero además brindar la posibilidad de interacción a los usuarios para que puedan

efectuar anotaciones en las visualizaciones durante las actividades de comunicación, coordinación y control (ver Figura 30). El propósito de los ECSs es apoyar los procesos de colaboración y toma de decisiones, por lo que su diseño puede tomar en consideración varios de los siguientes objetivos:

- * Proporcionar información sobre el estado del sistema para facilitar la realización de una tarea o la resolución de un problema de mantenimiento.
- * Facilitar la construcción de la cognición individual y grupal en torno a los procesos de DMES de un proyecto determinado al brindar información de seguimiento del estado del sistema.
- * Mostrar advertencias o presentar patrones que dejen al descubierto un comportamiento no esperado de los cambios o actividades que se han llevado a cabo en el sistema para activar los mecanismos de colaboración entre los miembros del equipo de trabajo.
- * Apoyar a los administradores de proyectos en los procesos de toma de decisiones, que pueden estar relacionadas con la asignación de tareas a los miembros del equipo, las actividades y tareas en proceso o con aspectos muy concretos y técnicos de los elementos de software del sistema.
- * Asistir a los programadores en su trabajo individual y el cumplimiento de tareas, metas y objetivos que le han sido asignados al ofrecer detalles sobre el progreso de las actividades que están realizando los demás miembros del equipo y el estado general del proyecto.

Cabe resaltar que las actividades de coordinación y control tienen efectos directos sobre los procesos, actividades y elementos de software del sistema, los cuales son la fuente que se utiliza para alimentar las visualizaciones de los ECSs.

4.2. GridMaster: Visualización de la estructura de los sistemas y colaboración

En esta sección se presenta GridMaster, una visualización para la evolución de proyectos de software que fue programada en Java como un plugin de Eclipse. El objetivo de GridMaster es apoyar a los administradores de proyectos y programadores, y proporcionar información para facilitar la colaboración entre los miembros de los equipos de trabajo. El desarrollo de esta herramienta se llevó a cabo tomando en cuenta los siguientes requerimientos:

Audiencia: Apoyar tanto a los administradores de proyectos como a los programadores mediante información pertinente de acuerdo a las tareas que desempeñan.

Tareas: Dar soporte a las siguientes tareas para apoyar a administradores de proyectos y programadores:

Toma de decisiones: Facilitar la toma de decisiones sobre la asignación de tareas a los programadores.

Comprensión de relaciones: Proporcionar información que permita comprender las relaciones entre los programadores y los elementos de software del sistema, así como la relación entre los mismos programadores.

Entender la estructura: Apoyar la comprensión de los cambios y la evolución de la estructura de los sistemas de software.

Comprensión de dependencias: Brindar información sobre las relaciones de herencia e implementación de interfaces y los cambios de esas relaciones en el tiempo.

Apoyo a la calidad: Ofrecer detalles sobre las variaciones de la calidad de los elementos de software mediante el uso de métricas de evolución.

Datos: Realizar la recuperación y análisis automático de los logs y el código fuente almacenado por los repositorios de software gestionados por las herramientas SCM.

Representación: Hacer uso de una representación sencilla y fácil de usar que transmita la información de forma rápida y con poca carga cognitiva, y que además soporte técnicas de interacción para la navegación y exploración de los datos. De forma adicional la visualización debe ofrecer:

Escalabilidad: La visualización debe ser capaz de representar la estructura de grandes sistemas de software y de acomodar el crecimiento de esa estructura conforme el sistema evoluciona.

Estructura estable: La visualización debe conservar la representación de los elementos de la estructura del sistema en la misma posición desde las primeras revisiones hasta las últimas para brindar al usuario una visualización uniforme durante la evolución del proyecto.

4.2.1. Descripción de la visualización

Con el uso de estructuras matriciales (ampliamente conocidas por los ingenieros de software) es posible diseñar representaciones visuales sencillas, escalables, poderosas e intuitivas que permiten correlacionar un gran número de elementos de datos y facilitan el descubrimiento de conocimiento. Las visualizaciones que utilizan este tipo de estructuras utilizan las filas y las columnas para representar datos que son correlacionados mediante la colocación de elementos gráficos en las celdas que se forman con su intersección.

GridMaster utiliza una estructura matricial cuyas celdas correlacionan la estructura con la evolución del sistema (ver Figura 31). En esta visualización la primera columna de la matriz es usada para representar la estructura del sistema, en donde los paquetes y elementos de software son distribuidos entre las filas de la matriz. Mientras que la primera fila de la estructura matricial es usada para representar la línea de tiempo de la evolución del sistema, y las unidades de tiempo que la conforman son distribuidas entre las columnas de la representación.

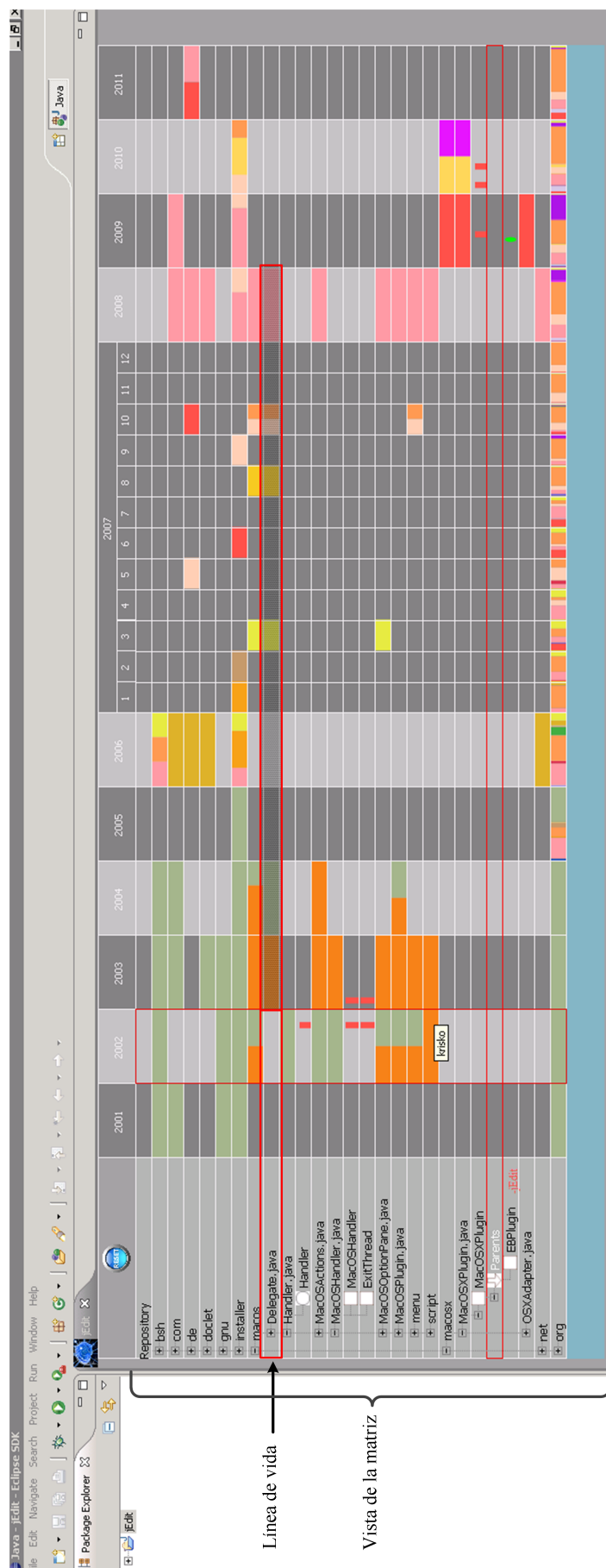


Figura 31: Vista principal de GridMaster.

Las celdas de GridMaster son usadas para correlacionar en el tiempo la estructura de un sistema con las métricas asociadas a sus elementos, las contribuciones de los programadores, la creación de los elementos de software, las relaciones socio-técnicas, y las creación o eliminación de las relaciones de herencia e implementación de interfaces.

Las técnicas de interacción utilizadas por esta visualización incluyen el uso del zoom, la distorsión Fisheye, la utilización de filtros para quitar nodos de la estructura del sistema y la posibilidad de seleccionar entre la representación absoluta o relativa de los valores de las métricas y contribuciones efectuadas por los programadores.

4.2.2. Estructura del proyecto

La representación visual de la estructura del proyecto se compone de los paquetes y elementos de software que han sido agregados durante su evolución. Si un paquete o elemento de software ha sido eliminado o movido de posición en la estructura del sistema este sigue siendo representado visualmente en su posición original, pero también es representado en su nueva posición. La correlación que se hace entre estos elementos y la línea de tiempo permite observar sus actividades en el tiempo, así si el elemento no existe en una determinada posición no se observará más actividad asociada a él. Esta característica construye una representación visual estable de la estructura del sistema, de forma que el usuario cuenta con una misma representación para toda la evolución del sistema. La Figura 32(a) muestra del lado izquierdo la estructura actual del sistema jEdit en el entorno de Eclipse, mientras que del lado derecho despliega la representación visual de la estructura del sistema incluyendo los paquetes y elementos de software que en la actualidad no forman parte del sistema.

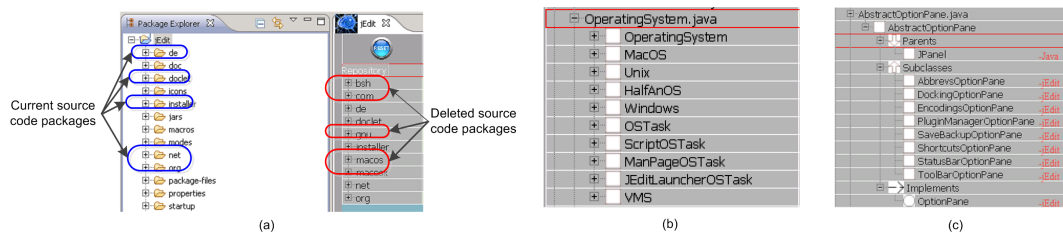


Figura 32: Estructura del sistema y contenido de los archivos. (a) Representación de la estructura del sistema. (b) Elementos de software que contiene un archivo. (c) Relaciones de herencia e implementación de interfaces.

4.2.3. Líneas de vida, colaboración y relaciones socio-técnicas

GridMaster representa las líneas de vida de los paquetes y elementos de software haciendo una correlación entre la estructura del sistema y la línea de tiempo, para lo cual rellena las celdas de la matriz en donde existe actividad de forma proporcional al volumen de actividades. El relleno que es colocado en las celdas de la representación usa colores para indicar cuales programadores han llevado a cabo las actividades (las cuales son operaciones de “check-in” sobre los archivos usando

una herramienta SCM). La proporcionalidad del relleno de la celda está en función del número de actividades asociadas a la unidad de tiempo que se está correlacionando y la representación que se está utilizando (absoluta o relativa), además cada programador se encuentra representado mediante el uso de un color.

La Figura 31 muestra la línea de vida y las actividades que se han llevado a cabo sobre el archivo “macos.Delegate.java” entre el 2003 and 2008. Sin embargo, al revisar los paquetes que forman parte de la última revisión del sistema se puede notar que el paquete “macos” en la actualidad no forma parte esta, lo cual se puede corroborar al revisar la estructura del proyecto en Eclipse, como muestra la Figura 32(a). De acuerdo con lo anterior la línea de tiempo de un paquete o elemento de software está representada por la presencia de actividad en la fila correspondiente. Así, se puede distinguir su creación cuando aparece actividad asociada por primera vez y se puede deducir que la evolución del elemento se ha estabilizado, o bien este ha sido movido o eliminado cuando no se observan más actividades relacionadas con este. Mientras que el uso de los colores permite ver cuáles programadores han realizado cambios a un elemento y por lo tanto es posible conocer quiénes han colaborado con su evolución. Esta misma característica de la visualización permite establecer las relaciones socio-técnicas al relacionar de forma directa a los programadores con los elementos del sistema. Pero además también permite obtener detalles sobre las relaciones entre los programadores por los elementos que han cambiado en común, lo que resulta de utilidad a los administradores de proyectos para la asignación de tareas y la sustitución de un programador cuando sucede algún evento que lo aparta de forma temporal o permanente del proyecto.

4.2.4. Relaciones: herencia e implementación de interfaces

La estructura del sistema es una estructura plegable que permite expandir los paquetes para ver los archivos que contiene, y los archivos a su vez se pueden expandir para observar los elementos de software que contienen (ver Figura 32 (b)). Al continuar expandiendo la estructura, los elementos de software permiten ver con cuales otros elementos de software ha establecido en algún momento relaciones de herencia e implementación de interfaces. Las relaciones de herencia que han sido tomadas en cuenta consideran tanto a los ascendientes como a los descendientes (ver Figura 32 (c)).

La Figura 33 muestra que el establecimiento de las relaciones de herencia e implementación de interfaces es representada por un óvalo de color y que la terminación de esas relaciones es indicada por un óvalo de color rojo. De forma adicional también se indica de forma explícita la localización de los elementos de software asociados (API de Java, el sistema bajo análisis o una biblioteca externa).

4.2.5. Métricas

Las métricas asociadas a los elementos de software son representadas haciendo uso de gráficas de barras con el fin de resaltar los cambios en sus valores (see

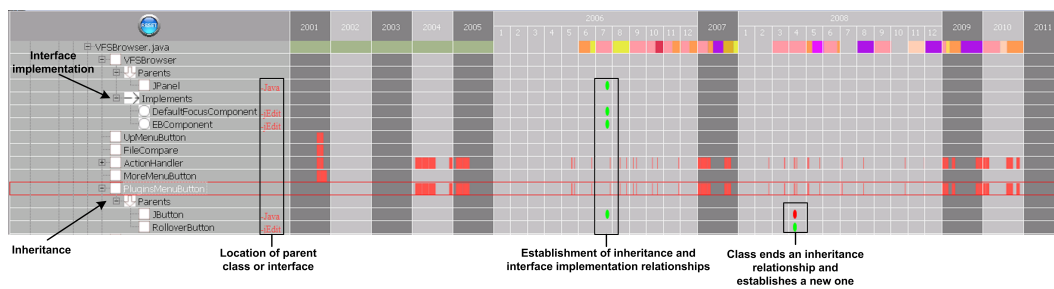


Figura 33: Relaciones de herencia e implementación de interfaces, incluyendo la expansión de un año y valores de métricas.

Figure 34). De forma similar a la representación de las contribuciones de los programadores los valores de las métricas son mostrados usando áreas relativas y absolutas. La representación relativa utiliza el valor de la métrica más alto, tomando en cuenta todos los valores de las métricas asociadas a los elementos de software del sistema, y con base en ese valor calcula el alto de la gráfica, mientras que la representación absoluta solo toma en cuenta el valor de la métrica más alto para el elemento de software que está siendo analizado.

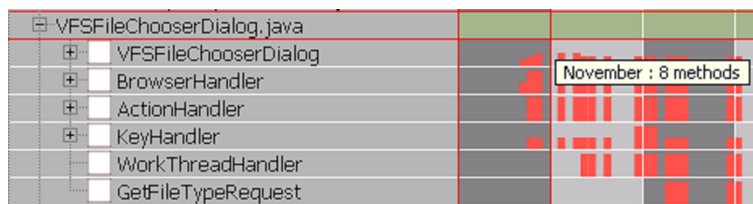


Figura 34: Representación de métricas.

4.3. Escenarios

Los siguientes escenarios buscan mostrar algunos ejemplos prácticos en los cuales GridMaster puede ser aplicada.

4.3.1. Escenario 1

Un error fue reportado y el administrador del proyecto debe asignar un programador para resolverlo. Una vez que el administrador del proyecto analizó el problema y determinó quien es el programador más indicado para resolverlo por estar a cargo de los componentes en los cuales es probable que se localice el error, se da cuenta que ese desarrollador se encuentra de vacaciones. Entonces el administrador del proyecto sigue los siguientes pasos utilizando GridMaster:

1. Abre la visualización mostrando el sistema completo, sin filtrar ningún paquete.

2. Selecciona una unidad de tiempo reciente que contiene un número considerable de actividades.
3. Revisa los elementos de sistema en los cuales considera que se puede encontrar localizado el error e inspecciona quienes han sido los programadores que han realizado cambios sobre esos elementos, con base en las relaciones socio-técnicas.
4. El administrador del proyecto determina cuál programador podría hacerse cargo de corregir el error y le asigna la tarea.

4.3.2. Escenario 2

Los programadores están en medio de una refactorización del sistema y necesitan revisar como la herencia y la implementación de interfaces ha sido afectada por los cambios realizados por los demás programadores. Por lo que los programadores siguen los siguientes pasos después de abrir GridMaster para revisar el sistema completo o solo un paquete que les resulta de su interés:

1. El programador selecciona un archivo, lo expande y selecciona el elemento de su interés.
2. Expande el elemento de software y examina las relaciones de herencia e implementación de interfaces para determinar si se ha establecido o terminado alguna de esas relaciones.

5. Conclusiones

El desarrollo, mantenimiento y evolución de sistemas de software en entornos distribuidos de forma global requiere de enfoques que contribuyan a mejorar la colaboración y la productividad de los equipos de trabajo. En este trabajo se ha discutido de forma amplia sobre la necesidad de comunicación, coordinación y control, así como de la importancia de crear espacios que permitan desarrollar la conciencia situacional tanto de los individuos como de los equipos. También se ha analizado el papel que juegan los procesos cognitivos individuales y grupales en la activación de los diferentes mecanismos que inician los procesos de colaboración.

Como resultado se ha definido un framework para describir la relación entre todos esos elementos y factores, y se ha desarrollado una herramienta de visualización que busca apoyar el desarrollo de la conciencia situacional y la colaboración entre los miembros del equipo de trabajo.

GridMaster, la herramienta que fue implementada, no ha sido evaluada formalmente. Pero su uso en escenarios concretos y situaciones controladas muestra su utilidad para proporcionar información sobre la arquitectura los sistemas, las relaciones entre los elementos de la arquitectura, patrones de colaboración y las relaciones socio-técnicas. Por lo que como trabajo futuro se plantea la evaluación formal de esta visualización con usuarios y la realización de mejoras para permitir que los usuarios puedan hacer anotaciones cuando encuentran información relevante, y que además puedan compartir sus descubrimientos con otros usuarios.

Bibliografía

- [1] Ala Abuthawabeh, Fabian Beck, Dirk Zeckzer, and Stephan Diehl. Finding structures in multi-type code couplings with node-link and matrix visualizations. In *First IEEE Working Conference on Software Visualization (VIS-SOFT), 2013*, pages 1–10, 2013. [Citado en págs. 28 y 29.]
- [2] Pär J. Agerfalk and Brian Fitzgerald. Flexible and distributed software processes: Old petunias in new bowls? *Communications of the ACM*, 49(10):26–34, October 2006. [Citado en págs. 5 y 8.]
- [3] Jauhar Ali. Cognitive support through visualization and focus specification for understanding large class libraries. *Journal of Visual Languages & Computing*, 20(1):50 – 59, 2009. [Citado en págs. 25.]
- [4] Craig Anslow, Stuart Marshall, James Noble, Ewan Tempero, and Robert Biddle. User evaluation of polymetric views using a large visualization wall. In *Proceedings of the 5th International Symposium on Software visualization, SOFT-VIS '10*, pages 25–34, New York, NY, USA, 2010. ACM. [Citado en págs. 38.]
- [5] Yannick Assogba and Judith Donath. Share: a programming environment for loosely bound cooperation. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '10*, pages 961–970, New York, NY, USA, 2010. ACM. [Citado en págs. IV, IV, IV, 44, 45, 46 y 47.]
- [6] A. Kitchenham Barbara. Controlling software projects. *Electronics and Power*, 33(5):312–315, May 1987. [Citado en págs. 11.]
- [7] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice, Second Edition*. Addison-Wesley Professional, April 2003. [Citado en págs. 23.]
- [8] Fabian Beck and Stephan Diehl. Visual comparison of software architectures. *Information Visualization*, 12(2):178–199, 04 2013. [Citado en págs. III, III, III, 29, 30, 33 y 35.]
- [9] Sassi Bentradi and Djamel Meslati. Visualizing and analyzing the structure of aspectj software under the eclipse platform. *International Journal of Software Engineering and Its Applications*, 7(3):353–376, May 2013. [Citado en págs. 27.]
- [10] Stuart K. Card, Jock D. Mackinlay, and Ben Shneiderman, editors. *Readings in information visualization: using vision to think*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999. [Citado en págs. 17.]
- [11] Erran Carmel. *Global Software Teams: Collaborating Across Borders and Time Zones*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1999. [Citado en págs. 3.]
- [12] Erran Carmel and Ritu Agarwal. Tactical approaches for alleviating distance in global software development. *IEEE Software*, 18(2):22–29, March 2001. [Citado en págs. 7.]

- [13] Marcelo Cataldo, Matthew Bass, James D. Herbsleb, and Len Bass. On coordination mechanisms in global software development. In *Second IEEE International Conference on Global Software Engineering, 2007. ICGSE 2007.*, pages 71–80, Aug 2007. [Citado en págs. 9 y 10.]
- [14] Louise K. Comfort. Crisis management in hindsight: Cognition, communication, coordination, and control. *Public Administration Review*, 67:189–197, 2007. [Citado en págs. 7, 9, 10 y 11.]
- [15] Eoin Ó. Conchúir, Par J. Agerfalk, Helena H. Olsson, and Brian Fitzgerald. Global software development: Where are the benefits? *Communications of the ACM*, 52(8):127 – 131, 2009. [Citado en págs. 3 y 5.]
- [16] Nancy J. Cooke, Jamie C. Gorman, Christopher W. Myers, and Jasmine L. Duran. Interactive team cognition. *Cognitive Science*, 37(2):255–285, 2013. [Citado en pág. 12.]
- [17] Stephan Diehl. *Software Visualization Visualizing the Structure, Behaviour, and Evolution of Software*. Springer Berlin Heidelberg New York, 2007. [Citado en pág. 2.]
- [18] Christof Ebert and Philip De Neve. Surviving global software development. *IEEE Software*, 18(2):62–69, March 2001. [Citado en pág. 3.]
- [19] Christof Ebert, Casimiro Hernandez Parro, Roland Suttels, and Harald Kolarczyk. Improving validation activities in a global software development. In *Proceedings of the 23rd International Conference on Software Engineering, ICSE '01*, pages 545–554, Washington, DC, USA, 2001. IEEE Computer Society. [Citado en pág. 3.]
- [20] Stephen G. Eick, Todd L. Graves, Alan F. Karr, Audris Mockus, and Paul Schuster. Visualizing software changes. *IEEE Transactions in Software Engineering*, 28(4):396–412, 2002. [Citado en pág. 18.]
- [21] Mica R. Endsley. Toward a theory of situation awareness in dynamic systems. *Human Factors: The Journal of the Human Factors and Ergonomics Society*, 37(1):32–64, 1995. [Citado en págs. 13 y 17.]
- [22] Jacky Estublier. Distributed objects for concurrent engineering. In *Proceedings of the 9th International Symposium on System Configuration Management, SCM-9*, pages 172–185, London, UK, UK, 1999. Springer-Verlag. [Citado en pág. 3.]
- [23] Stephen M. Fiore and Eduardo Salas. *Team cognition: Understanding the factors that drive process and performance*, chapter Advances in measuring team cognition., pages 83–106. American Psychological Association, Washington, DC, US, 2004. [Citado en págs. 6, 12, 13 y 16.]

- [24] Stephen M. Fiore and Eduardo Salas. *Team cognition: Understanding the factors that drive process and performance*, chapter Why we need team cognition., pages 235–248. American Psychological Association, Washington, DC, US, 2004. [Citado en pág. 13.]
- [25] Kurt F. Fischer. Software quality assurance tools: Recent experience and future requirements. *SIGSOFT Software Engineering Notes*, 3(5):116–121, January 1978. [Citado en pág. 11.]
- [26] George W. Furnas. Generalized fisheye views. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '86, pages 16–23, New York, NY, USA, 1986. ACM. [Citado en pág. 22.]
- [27] Tudor Girba, Adrian Kuhn, Mauricio Seeberger, and Stéphane Ducasse. How developers drive software evolution. In *Proceedings of the Eighth International Workshop on Principles of Software Evolution, IWPSE '05*, pages 113–122, Washington, DC, USA, 2005. IEEE Computer Society. [Citado en págs. IV, IV, 39 y 41.]
- [28] Antonio González, Roberto Therón, Alexandru Telea, and Francisco J. García. Combined visualization of structural and metric information for software evolution analysis. In *Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops, IWPSE-Evol '09*, pages 25–30, New York, NY, USA, 2009. ACM. [Citado en págs. III, 32 y 33.]
- [29] Denis Gracanin, Kresimir Matkovic, and Mohamed Eltoweissy. Software visualization. *Innovations in Systems and Software Engineering*, Volumen 1(Number 3):221 – 230, 1994. [Citado en pág. 2.]
- [30] LilePalma Hattori, Michele Lanza, and Romain Robbes. Refining code ownership with synchronous changes. *Empirical Software Engineering*, 17(4-5):467–499, 2012. [Citado en pág. 39.]
- [31] Jun He, Brian Butler, and William King. Team cognition: Development and evolution in software project teams. *Journal of Management Information Systems*, 24(2):261–292, October 2007. [Citado en págs. 6, 7, 12, 13 y 14.]
- [32] Brandon Heller, Eli Marschner, Evan Rosenfeld, and Jeffrey Heer. Visualizing collaboration and influence in the open-source software community. In *Proceedings of the 8th Working Conference on Mining Software Repositories, MSR '11*, pages 223–226, New York, NY, USA, 2011. ACM. [Citado en pág. 47.]
- [33] James D. Herbsleb and Audris Mockus. An empirical study of speed and communication in globally distributed software development. *IEEE Transactions in Software Engineering*, 29(6):481–494, June 2003. [Citado en págs. 3, 8 y 9.]
- [34] James D. Herbsleb, Audris Mockus, Thomas A. Finholt, and Rebecca E. Grinter. An empirical study of global software development: Distance and speed. In *Proceedings of the 23rd International Conference on Software Engineering*,

- ICSE '01, pages 81–90, Washington, DC, USA, 2001. IEEE Computer Society. [Citado en págs. 3 y 8.]
- [35] James D. Herbsleb and Deependra Moitra. Global software development. *IEEE Software*, 18(2):16–20, Mar 2001. [Citado en pág. 3.]
- [36] Abram Hindle, Zhen Ming Jiang, Walid Koneilat, Michael W. Godfrey, and Richard C. Holt. Yarn: Animating software evolution. In *4th IEEE International Workshop on Visualizing Software for Understanding and Analysis, 2007. VIS-SOFT 2007.*, pages 129–136, 2007. [Citado en págs. III, 34 y 36.]
- [37] Andrejs Jermakovics, Alberto Sillitti, and Giancarlo Succi. Mining and visualizing developer networks from version control systems. In *Proceedings of the 4th International Workshop on Cooperative and Human Aspects of Software Engineering*, CHASE '11, pages 24–31, New York, NY, USA, 2011. ACM. [Citado en págs. IV, 46 y 48.]
- [38] Miguel Jiménez, Mario Piattini, and Aurora Vizcaíno. Challenges and improvements in distributed software development: A systematic review. *Advances in Software Engineering*, 2009, 2009. [Citado en pág. 3.]
- [39] Stephen H. Kan. *Metrics and Models in Software Quality Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002. [Citado en pág. 11.]
- [40] Dale W. Karolak. *Global Software Development: Managing Virtual Teams and Environments*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1st edition, 1999. [Citado en pág. 3.]
- [41] Rick Kazman, Gregory Abowd, Len Bass, and Paul Clements. Scenario-based analysis of software architecture. *IEEE Software*, 13(6):47–55, Nov 1996. [Citado en pág. 23.]
- [42] Mumtaz Muhammad Khan, Sulaiman Aziz Lodhi, and Muhammad Abdul Majid Makk. Measuring team implicit coordination. *Australian Journal of Basic and Applied Sciences*, 4(6):1211–1136, 2010. [Citado en pág. 8.]
- [43] Taimur Khan, Henning Barthel, Achim Ebert, and Peter Liggesmeyer. Visualization and Evolution of Software Architectures. In Christoph Garth, Ariane Middel, and Hans Hagen, editors, *Visualization of Large and Unstructured Data Sets: Applications in Geospatial Planning, Modeling and Engineering - Proceedings of IRTG 1131 Workshop 2011*, volume 27 of *OpenAccess Series in Informatics (OASIS)*, pages 25–42, Dagstuhl, Germany, 2012. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. [Citado en pág. 24.]
- [44] Preston A . Kiekel and Nancy J . Cooke. *Handbook of Human Factors in Web Design, Second Edition*, chapter Human Factor Aspects of Team Cognition, pages 107 – 123. CRC Press, 2011. [Citado en págs. 6, 7 y 13.]

- [45] Laurie J. Kirsch. The management of complex tasks in organizations: Controlling the systems development process. *Organization Science*, 7(1):1–21, 1996. [Citado en págs. 10 y 11.]
- [46] Barbara A. Kitchenham and John G. Walker. A quantitative approach to monitoring software development. *Software Engineering Journal*, 4(1):2–13, Jan 1989. [Citado en pág. 11.]
- [47] Richard Klimoski and Susan Mohammed. Team mental model: construct or metaphor? *Journal of Management*, 20(2):403 – 437, 1994. A Special Issue of The Journal of Management. [Citado en págs. 6 y 12.]
- [48] Julia Kotlarsky, Paul C. van Fenema, and Leslie P. Willcocks. Developing a knowledge-based perspective on coordination: The case of global software projects. *Information and Management*, 45(2):96 – 108, 2008. [Citado en pág. 9.]
- [49] Robert E. Kraut and Lynn A. Streeter. Coordination in software development. *Communications of the ACM*, 38(3):69–81, March 1995. [Citado en pág. 9.]
- [50] Adrian Kuhn, David Erni, Peter Loretan, and Oscar Nierstrasz. Software cartography: thematic software visualization with consistent layout. *Journal of Software Maintenance and Evolution: Research and Practice*, 22(3):191–210, 2010. [Citado en págs. IV, IV, 35 y 37.]
- [51] Adrian Kuhn, David Erni, and Oscar Nierstrasz. Embedding spatial software visualization in the ide: an exploratory study. In *Proceedings of the 5th international symposium on Software visualization*, SOFTVIS '10, pages 113–122, New York, NY, USA, 2010. ACM. [Citado en pág. 37.]
- [52] Adrian Kuhn and Mirko Stocker. Codetimeline: Storytelling with versioning data. In *34th International Conference on Software Engineering (ICSE), 2012*, pages 1333–1336, 2012. [Citado en págs. IV, 41 y 42.]
- [53] Su-Ying Lai, Richard Heeks, and Brian Nicholson. Uncertainty and coordination in global software projects: A uk/india-centred case study. In *Development informatics working paper series*, number 17 in Development informatics working paper series. Manchester : Institute for Development Policy and Management, University of Manchester, 2003. [Citado en págs. 9 y 10.]
- [54] Michele Lanza, Lile Hattori, and Anja Guzzi. Supporting collaboration awareness with real-time visualization of development activity. In *Proceedings of the 2010 14th European Conference on Software Maintenance and Reengineering*, CSMR '10, pages 202–211, Washington, DC, USA, 2010. IEEE Computer Society. [Citado en págs. 47 y 48.]
- [55] Piritta Leinonen, Sanna Järvelä, and Päivi Häkkinen. Conceptualizing the awareness of collaboration: A qualitative study of a global virtual team. *Computer Supported Cooperative Work*, 14(4):301–322, August 2005. [Citado en pág. 14.]

- [56] Ying K. Leung and Mark D. Apperley. A review and taxonomy of distortion-oriented presentation techniques. *ACM Transactions in Computer-Human Interaction*, 1(2):126–160, June 1994. [Citado en págs. III, III, III, 20, 21, 22 y 23.]
- [57] Daniel Limberger, Benjamin Wasty, Jonas Trümper, and Jürgen Döllner. Interactive software maps for web-based source code analysis. In *Proceedings of the 18th International Conference on 3D Web Technology, Web3D '13*, pages 91–98, New York, NY, USA, 2013. ACM. [Citado en pág. 26.]
- [58] Jean MacMillan, Elliot E. Entin, and Daniel Serfaty. *Team cognition: Understanding the factors that drive process and performance*, chapter Communication overhead: The hidden cost of team cognition., pages 61–82. American Psychological Association, Washington, DC, US, 2004. [Citado en págs. 8, 9 y 13.]
- [59] Jonathan I. Maletic, Andrian Marcus, and Michael L. Collard. A task oriented view of software visualization. *IEEE Workshop of Visualizing Software for Understanding and Analysis (VISSOFT)*, 26:32–40, 2002. [Citado en pág. 18.]
- [60] Merriam-Webster Online. Merriam-Webster Online Dictionary, 2009. [Citado en págs. 7, 8 y 10.]
- [61] Henry Mintzberg. The effective organization: Forces and forms. *Mit Sloan Management Review*, January, 15 1991. [Citado en pág. 3.]
- [62] Sanjay Misra, Ricardo Colomo-Palacios, Tolga Pusatli, and Pedro Soto-Acosta. A discussion on the role of people in global software development. *Tehnicki vjesnik / Technical Gazette*, 20(3):525 – 531, 2013. [Citado en pág. 3.]
- [63] Audris Mockus and David M. Weiss. Globalization by chunking: A quantitative approach. *IEEE Software*, 18(2):30–37, March 2001. [Citado en págs. 3, 7 y 8.]
- [64] Michael Ogawa and Kwan-Liu Ma. code_swarm: A design study in organic software visualization. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1097–1104, nov 2009. [Citado en pág. 3.]
- [65] William G. Olchi. The transmission of control through organizational hierarchy. *Academy of Management Journal*, 2(2):173–192, June 1978. [Citado en pág. 10.]
- [66] Inah Omoronyia, John Ferguson, Marc Roper, and Murray Wood. A review of awareness in distributed collaborative software engineering. *Software: Practice and Experience*, 40(12):1107–1133, 2010. [Citado en pág. 3.]
- [67] Thomas Panas, Rebecca Berrigan, and John Grundy. A 3d metaphor for software production visualization. In *Proceedings. Seventh International Conference on Information Visualization, 2003. IV 2003.*, pages 314–319, July 2003. [Citado en pág. 26.]
- [68] Thomas Panas, Rüdiger Lincke, and Welf Löwe. Online-configuration of software visualizations with vizz3d. In *Proceedings of the 2005 ACM Symposium on Software Visualization, SoftVis '05*, pages 173–182, New York, NY, USA, 2005. ACM. [Citado en pág. 26.]

- [69] Wim De Pauw, Doug Kimelman, and John Vlissides. Modeling object-oriented program execution. In Mario Tokoro and Remo Pareschi, editors, *Object-Oriented Programming*, volume 821 of *Lecture Notes in Computer Science*, pages 163–182. Springer Berlin Heidelberg, 1994. [Citado en pág. 1.]
- [70] David Pinelle and Carl Gutwin. A groupware design framework for loosely coupled workgroups. In *Proceedings of the Ninth Conference on European Conference on Computer Supported Cooperative Work, ECSCW'05*, pages 65–82, New York, NY, USA, 2005. Springer-Verlag New York, Inc. [Citado en pág. 3.]
- [71] Rafael Prikladnicki1, Jorge Luis Nicolas Audy, and Roberto Evaristo. Global software development in practice lessons learned. *Software Process: Improvement and Practice*, 8(4):267–281, 2003. [Citado en pág. 3.]
- [72] Balasubramaniam Ramesh, Lan Cao, Kannan Mohan, and Peng Xu. Can distributed software development be agile? *Communications of the ACM*, 49(10):41–46, October 2006. [Citado en págs. 8 y 10.]
- [73] Roger M. Ripley, Anita Sarma, and Andre van der Hoek. A visualization for software project awareness and evolution. In *Visualizing Software for Understanding and Analysis, 2007. VISSOFT 2007. 4th IEEE International Workshop on*, pages 137–144, june 2007. [Citado en págs. IV, IV, 41 y 43.]
- [74] Paul Rook. Controlling software projects. *Software Engineering Journal*, 1(1):7–, January 1986. [Citado en págs. 9, 10 y 11.]
- [75] Eduardo Salas, Carolyn Prince, David P. Baker, and Lisa Shrestha. Situation awareness in team performance: Implications for measurement and training. *Human Factors: The Journal of the Human Factors and Ergonomics Society*, 37(1):123–136, 1995. [Citado en págs. 16 y 17.]
- [76] Paul M. Salmon and Neville A. Stanton. Situation awareness and safety: Contribution or confusion? situation awareness and safety editorial. *Safety Science*, 56(0):1–5, 2013. Situation Awareness and Safety. [Citado en págs. 6 y 13.]
- [77] Neeraj Sangal, Ev Jordan, Vineet Sinha, and Daniel Jackson. Using dependency models to manage complex software architecture. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA '05*, pages 167–176, New York, NY, USA, 2005. ACM. [Citado en págs. III, III, 27, 28 y 29.]
- [78] Neville A. Stanton, P.R.G Chambers, and J Piggott. Situational awareness and safety. *Safety Science*, 39(3):189–204, 2001. [Citado en pág. 13.]
- [79] Neville A. Stanton, Rebecca Stewart, Don Harris, Robert J. Houghton, Chris Baber, Richard McMaster, Paul Salmon, Geoff Hoyle, Guy Walker, Mark S. Young, Mark Linsell, Roy Dymott, and Damian Green. [Citado en pág. 15.]
- [80] Frank Steinbrückner and Claus Lewerentz. Understanding software evolution with software cities. *Information Visualization*, 12(2):200–216, 04 2013. [Citado en págs. III, III, III, 31, 32, 33 y 34.]

- [81] Amir Talaei-Khoei, Pradeep Ray, Nandan Parameshwaran, and Lundy Lewis. A framework for awareness maintenance. *Journal of Network and Computer Applications*, 35(1):199 – 210, 2012. [Citado en pág. 3.]
- [82] Simon L. R. Vrhoveca, Marina Trkmana, Ales Kumera, Marjan Krispera, and Damjan Vavpotica. Outsourcing as an economic development tool in transition economies: Scattered global software development. *Information Technology for Development*, 0(0):1–15, 0. [Citado en pág. 3.]
- [83] Colin Ware. *Information Visualization, Second Edition: Perception for Design (Interactive Technologies)*. Morgan Kaufmann, 2 edition, April 2004. [Citado en pág. 18.]
- [84] Richard Wettel and Michele Lanza. Visualizing software systems as cities. In *4th IEEE International Workshop on Visualizing Software for Understanding and Analysis, 2007. VISSOFT 2007.*, pages 92–99, 2007. [Citado en págs. III, 26 y 27.]
- [85] Richard Wettel and Michele Lanza. Visual exploration of large-scale system evolution. In *15th Working Conference on Reverse Engineering, 2008. WCRE '08.*, pages 219–228, Oct 2008. [Citado en págs. III, III, III, 26, 27, 30 y 31.]
- [86] Peter Young and Malcolm Munro. Visualising software in virtual reality. In *Proceedings., 6th International Workshop on Program Comprehension, 1998. IWPC '98.*, pages 19–26, Jun 1998. [Citado en págs. 17, 18 y 19.]

Glosario

AP	Administrador de Proyectos	2, 17
CS	Conciencia Situacional	14–16
CSET	Conciencia Situacional del Equipo de Trabajo.....	16
CVS	Concurrent Versioning System	38
DGS	Desarrollo Global de Software.....	1–10, 16
DMES	Desarrollo, Mantenimiento y Evolución de Software	1, 3–5, 14, 16–19, 48–50
DSM	Dependency Structure Matrix	27, 28, 32
ECS	Espacio de Conciencia Situacional.....	16, 17, 48–50
ES	Evolución de Software	2
ETCSD	Espacio de Trabajo de Conciencia Situacional Distribuida.	14, 15
IDE	Entornos de Desarrollo Integrado.....	1, 37
IMMV	Interactive Multi-Matrix Visualization	28
PNL	Parallel Node-Link.....	28, 29
SCM	Administración de la Configuración de Software.	1, 11, 41, 51, 53
SHV	System Hotspots View	37, 38
SQA	Aseguramiento de la Calidad de Software.....	11, 12
VI	Visualización de la Información	17
VS	Visualización de Software	2