Capítulo 1

Introducción: Problema y Solución

Cuando se desarrollan los sistemas de software se busca que el producto final satisfaga las necesidades de los usuarios, que el buen funcionamiento perdure en el mayor tiempo posible y en caso de hacer una modificación para mejorarlo no implique grandes costos. Estos puntos conllevan a un software de alta calidad y la disciplina encargada de conseguirlo es la *Ingeniería de Software* (IS). Para que la IS logre las metas antes mencionadas se utilizan, entre otras tantas, tres temáticas importantes que están orientadas al desarrollo de buenos sistemas: el *mantenimiento del software*, la *evolución del software* y la *migración del software*.

1.1. Mantenimiento del Software

La etapa de mantenimiento es importante en el desarrollo del software, dado que los sistemas están sujetos a cambios y a una permanente evolución [32].

El mantenimiento del software según la IEEE standard [25], *es la modificación del software que se realiza posterior a la entrega del producto al usuario con el fin de arreglar fallas, mejorar el rendimiento, o adaptar el sistema al ambiente que ha cambiado.*

De la definición anterior se desprenden 4 tipos de cambios que se pueden

realizar en la etapa de mantenimiento. El *mantenimiento correctivo* cambia el software para reparar las fallas detectadas por el usuario. El *mantenimiento adaptativo* modifica el sistema para adaptarlo a cambios externos, como por ejemplo actualización del sistema operativo o el motor de base de datos. El *mantenimiento perfectivo* se encarga de agregar nuevas funcionalidades al software que son descubiertas por el usuario. Por último, el *mantenimiento preventivo* realiza cambios al programa para facilitar futuras correcciones o adaptaciones que puedan surgir en el futuro [28].

Por lo antedicho, entre otras diversas razones [7] indican que el mantenimiento del software consume mucho esfuerzo y dinero. En algunos casos, los costos de mantenimiento pueden duplicar a los que se emplearon en el desarrollo del producto. Las causas que pueden aumentar estos costos son, el mal diseño de la arquitectura del programa, la mala codificación, la ausencia de documentación.

No es descabellado pensar estrategias de automatización que puedan ser aplicadas en fases del mantenimiento del software que ayuden a reducir estos costos, lo cual requiere una comprensión del objeto que se va a modificar antes de realizar algún cambio que sea de utilidad.

Algunos autores [7, 30, 28, 32] concluyen que el mantenimiento del software y la comprensión de los programas son conceptos que están fuertemente relacionados. Una frase conocida en la jerga de la IS es: *"Mientras más fácil sea comprender un sistema, más fácil será de mantenerlo"*.

1.2. Evolución del Software

Los sistemas complejos evolucionan con el tiempo, los nuevos usuarios y requisitos durante el desarrollo del mismo causan que el producto final posiblemente no sea el que se planteó en un comienzo.

La evolución del software básicamente se atribuye al crecimiento de los sistemas a través del tiempo, es decir, tomar una versión operativa y generar una nueva versión ampliada o mejorada en su eficiencia [7].

Generalmente, los ingenieros del software recurren a los modelos evolutivos. Estos modelos indican que cuando se desarrolla un producto de software

es conveniente que se divida en distintas iteraciones. A medida que avanza el desarrollo, cada iteración retorna una version entregable cada vez más compleja [28].

Estos modelos son muy recomendados sobre todo si se tienen fechas ajustadas donde se necesita una versión funcional lo más rápido posible. De esta manera, no se requiere esperar una versión completa al final del proceso de desarrollo.

Es inevitable y fundamental comprender correctamente la iteración actual del producto antes de comenzar con una nueva. Nuevos requerimientos del usuario pueden aparecer, como así también nuevas dificultades en el desarrollo del sistema. Es por esto que la comprensión del sistema durante su evolución es crucial.

1.3. Migración del Software

Las tareas de mantenimiento de software no solo se realizan para mejorar cuestiones internas del sistema, como es el caso de arreglos de errores de programación o requisitos nuevos del usuario. También se necesitan para adaptar el software al contexto cambiante. Es aquí donde la *migración del software* entra en escena.

De acuerdo con la IEEE standard [24], *la migración del software es convertir o adaptar un viejo sistema (sistema heredado) a un nuevo contexto tecnológico sin cambiar la funcionalidad del mismo.*

Las migraciones de sistemas más comunes que se llevan a cabo son causadas por cambios en el hardware obsoleto, nuevos sistemas operativos, cambios en la arquitectura y nuevas base de datos. Sin duda, la que más se ha acentuado en los últimos años son la aparición de nuevas tecnologías web y las mobile (basadas en dispositivos móviles) [34].

La migración de grandes sistemas es fundamental, sin embargo está demostrado que es costosa y compleja [34]. Para llevarla a cabo reduciendo estos costos se recomienda hacer previamente una buena comprensión del sistema antiguo.

Una técnica de comprensión de sistemas ideada recientemente para la

migración de software, consiste en un modelado de datos [22]. Este modelado tiene como finalidad representar el sistema en distintos niveles de abstracción. De esta forma, el grupo encargado de la migración solo debe preocuparse por convertir el antiguo código a la nueva tecnología.

Nuevamente la comprensión de programas se hace presente en temáticas relacionadas al desarrollo de proyectos de software.

1.4. Comprensión de Programas

Como se explicó en las secciones anteriores, es crucial lograr comprender el sistema para llevar adelante las tareas de mantenimiento, evolución y migración del software. Por esta razón, existe un área de la IS que se dedica al desarrollo de técnicas de inspección y comprensión de software. Esta área se conoce con el nombre de *Comprensión de Programas* (CP).

La CP se define como: *Una disciplina de la IS encargada de ayudar al desarrollador a lograr un entendimiento acabado del software. De forma tal que se pueda analizar disminuyendo en lo posible el tiempo y los costos* [9].

La CP asiste al equipo de desarrollo de software proveyendo métodos, técnicas y herramientas que faciliten el entendimiento del sistema de estudio.

Las investigaciones realizadas en el área de la CP determinaron que el foco de estudio se centra en relacionar el *Dominio del Problema* con el *Dominio del Programa* [10, 9, 3, 17]. El primero hace referencia a los elementos que forman parte de la salida del sistema, mientras que el segundo indican las componentes del programa empleados para generar dicha salida. Esta relación es compleja de realizar y representa el principal desafío de la CP.

Una aproximación para relacionar ambos dominios consiste en construir:

I) Armar una representación del *Dominio del Problema*. II) Construir una representación para el *Dominio del Programa*. III) Unir ambas representaciones con una *Estrategia de Vinculación*.

Para llevar a cabo los 3 pasos mencionados, se necesitan diferentes temáticas las cuales se mencionan a continuación:

- Los *Modelos Cognitivos* son relevantes para la CP porque destacan co-

mo el programador utiliza procesos mentales para comprender el software.

- La *Visualización del Software* analiza las distintas partes del sistema y genera representaciones visuales que agilizan la CP.
- La *Interconexión de Dominios* trata de como interrelacionar los elementos de un dominio con otro. Es útil para la CP en la reconstrucción de la relación entre dominio del problema y el dominio del programa.
- La *Extracción de Información* en los sistemas de software, es necesaria para las estrategias de cognición, visualización, interconexión de dominios, etc.
- Al extraer gran cantidad de información se necesita la *Administración de Información*. La misma brinda técnicas de almacenamiento y acceso eficiente a la información extraída.

Todos estos temas se explican con mayor precisión en el próximo capítulo. En la siguiente sección se amplia la *extracción de la información* ya que es la antesala al *análisis de identificadores*, eje central de este trabajo final.

1.5. Extracción de Información

En la CP, la *Extracción de la Información* se define como *el uso/desarrollo de técnicas que permitan extraer información desde el sistema de estudio.* Esta información puede ser: Estática o Dinámica, dependiendo de las necesidades del ingeniero de software o del equipo de trabajo.

La extracción de información estática recupera los distintos elementos en el código fuente de un programa [1].

Por otro lado, la extracción dinámica de información implica obtener los elementos que se utilizaron en la ejecución del sistema [6]. Generalmente no todas las partes del código son ejecutadas, por lo tanto no todos los elementos se pueden recuperar en una sola ejecución.

La principal diferencia que radica entre ambas técnicas, es que las estáticas se basan en la información presente en el código, mientras que las dinámi-

cas es obligatorio ejecutar el sistema.

En este trabajo final se hace énfasis en la extracción de información estática. Es importante aclarar que la información dinámica es tan importante como la información estática, sin embargo su extracción requiere del estudio de otro tipo de aproximaciones y escapan al alcance de este trabajo.

Como se mencionó previamente, en los códigos de software abundan componentes que conforman la información estática. Alguno de ellos son, nombres de variables, tipos de las variables, los métodos de un programa, las variables locales a un método, constantes. Todos estos componentes se representan mediante los identificadores (ids). Los ids abarcan gran parte de los elementos de un código por lo tanto su análisis no debe pasarse por alto.

1.6. Análisis de Identificadores

Estudios realizados [14, 11, 20, 19] indican que gran parte de los códigos están conformados por identificadores (ids), por ende abundante información estática está representada por ellos (Ver Capítulo 3).

Generalmente, los ids están compuestos por más de una palabra en forma de abreviaturas. Varios autores coinciden [13, 26, 23, 18] que detrás de estas abreviaturas, se encuentra oculta información que es propia del dominio del problema.

Una vía posible para exponer la información oculta consiste en traducir las abreviaturas antes mencionadas, en sus correspondientes palabras expresadas en lenguaje natural. Para lograrlo: I) primero se extraen los identificadores del código, II) luego se les aplica técnicas de división en donde se descompone al identificador en las distintas palabras abreviadas que lo componen. Por ejemplo: inp_fl_sys $\rightarrow$ inp fl sys. Finalmente, III) se emplea estrategias de expansión a las abreviaturas para transformar las mismas en palabras completas. Siguiendo con el ejemplo anterior: inp fl sys $\rightarrow$ input file system.

Normalmente, los nombres de los ids son elegidos en base a criterios del programador [26, 18]. Lamentablemente, estos criterios son desconocidos para las técnicas que expanden abreviaturas en los ids. Una forma de afrontar esta dificultad, es recurrir a fuentes de información informal que se encuen-

tran disponible en el código fuente. Por información informal se entiende aquella contenida en los comentarios de los módulos, comentarios de las funciones, literales strings, documentación del sistema y todos lo demás recursos descriptivos del programa que estén escritos en lenguaje natural.

Sin duda, los comentarios tienen como principal finalidad ayudar a comprender un segmento de código [21, 4, 5]. Por esta razón, se puede ver a los comentarios como una herramienta natural para entender el significado de los ids en el código, como así también el funcionamiento del sistema.

Por otro lado para poder entender la semántica de los ids, se toman literales o constantes strings. Estos representan un valor constante formado por secuencias de caracteres. Ellos son generalmente utilizados en la muestra de carteles por pantalla, y comúnmente se almacenan en variables de tipo string.

Los literales string como los comentarios pueden brindar indicios del significado de las abreviaturas que se desean expandir.

En caso de que estas fuentes de información informal sean escasas dentro del mismo sistema, se puede acudir a alternativas externas como es el caso de los diccionarios predefinidos de palabras en lenguaje natural.

Las estrategias de análisis de ids, explicadas en esta sección, se han ido mejorando a lo largo del tiempo. Al principio contaban con diccionarios extensos y ocupaban mucho espacio, luego estos diccionarios se fueron reemplazando por listados de palabras que son acordes a la ciencias de la computación. Estos listados son más eficientes que los diccionarios, ya que contienen palabras más precisas y ocupan menos espacio de almacenamiento (Ver Capítulo 3).

1.7. Problema y la Solución

En síntesis, un camino para tratar de comprender los sistemas, es fijar la atención en los ids. En la actualidad entender el significado de los ids en los programas se realiza generalmente de manera manual leyendo el código, esto implica grandes esfuerzos para el lector sobre todo si el código del sistema es grande. Las herramientas automáticas que analizan ids simplifican en gran medida estas arduas tareas. Normalmente estas herramientas para realizar sus tareas, efectúan la traducción de ids, e involucran los siguientes pasos:

I) capturar los ids del código, II) dividir las palabras abreviadas que componen un id, III) expandir las abreviaturas en base a lo observado en comentarios/literales y/o diccionarios, etc. Desafortunadamente, hasta el momento se pudieron encontrar pocas herramientas que lleven a cabo los tres pasos antedichos.

Para abordar las iniciativas planteadas y hacer algunos aportes a la solución del problema descripto en el párrafo anterior, se pretende:

- Construir un analizador sintáctico en lenguaje Java que permite extraer los ids, comentarios y literales encontrados en el código fuente del sistema de estudio. La herramienta a seleccionar para realizar esta tarea es ANTLR[1].
- Armar un diccionario de palabras en lenguaje natural que sirva como alternativa a las fuentes de información informal.
- Investigar técnicas de división de ids y técnicas expansión de abreviaturas. Algunas de estas utilizan como principal recurso la información brindada en los comentarios y los literales extraídos del código. Como segunda posibilidad se recurre al diccionario.
- Implementar empleando el lenguaje JAVA algunas de las técnicas del ítem anterior en una herramienta denominada *Identifier Analyzer* (IDA). Esta herramienta también permite visualizar atributos del id (ambiente, tipo de identificador, número de línea, etc) mostrando la parte del código donde se encuentra ubicados.
- Probar la herramienta IDA con casos de estudios, para demostrar la utilidad de la misma.
- Mediante un gráfico de barras, comparar el desempeño de cada técnica implementada en IDA y sacar las conclusiones pertinentes.

[1] ANother Tool for Language Recognition - http://www.antlr.org

1.8. Contribución

Como se mencionó previamente, las herramientas que ayudan a comprender los ids son importantes para la CP, dado que facilitan encontrar indicios sobre las funcionalidades del sistema. Teniendo en cuenta la escasez de este tipo de herramientas en el ámbito de la CP, el correspondiente trabajo final consiste en el diseño y la implementación de una nueva herramienta que analiza ids en programas escritos en JAVA. Esta herramienta llamada *Identifier Analyzer* (IDA), le permite al usuario ingresar un programa escrito en JAVA, luego IDA captura los ids de forma automática y finalmente mediante la ejecución de técnicas específicas, ayuda al usuario a encontrar el significado de los ids en el programa ingresado. La herramienta IDA, además de analizar ids, puede ser utilizada como base para construir futuras implementaciones en este mismo contexto.

1.9. Organización del Trabajo Final

El trabajo está organizado de la siguiente manera. El Capítulo 2 define conceptos teóricos relacionados a la comprensión de programas. El Capítulo 3 describe el estado del arte asociado a las técnicas de análisis de identificadores conocidas. El Capítulo 4 trata sobre la herramienta *Identifier Analyzer* (IDA), en donde se explican los distintos módulos que la herramienta posee y que técnicas de análisis de identificadores tiene implementadas. En el Capítulo 5, se describen algunos casos de estudio que demuestran la utilidad de la herramienta IDA. Finalmente, en el Capítulo 6 se explican las conclusiones elaboradas y se proponen trabajos futuros.

Capítulo 2

Comprensión de Programas

En el capítulo 1 se explicó una aproximación para ayudar a comprender programas. Esta solución se basa en el análisis de los identificadores encontrados en el código del sistema de estudio.

En este capítulo se definen algunos conceptos importantes sobre comprensión de programas (CP). Al final se redactan algunos comentarios sobre los temas tratados en el correspondiente capítulo.

2.1. Introducción

La CP es un área de la Ingeniería de Software que tiene como meta primordial desarrollar métodos, técnicas y herramientas que ayuden al programador a comprender en profundidad los sistemas de software.

Diversos estudios e investigaciones demuestran que el principal desafío en la CP es vincular el *Dominio del Problema* y el *Dominio del Programa* [10, 9, 8, 17]. El primero se refiere al resultado de la ejecución del sistema, mientras que el segundo indica todos los componentes de software que causaron dicho resultado.

La comunidad de CP sostiene, basado en estudios de los modelos cognitivos, que un programador entiende un programa cuando puede relacionar el comportamiento del mismo con su operación. En otras palabras cuando puede vincular el *Dominio del Problema* y el *Dominio del Programa*. El in-

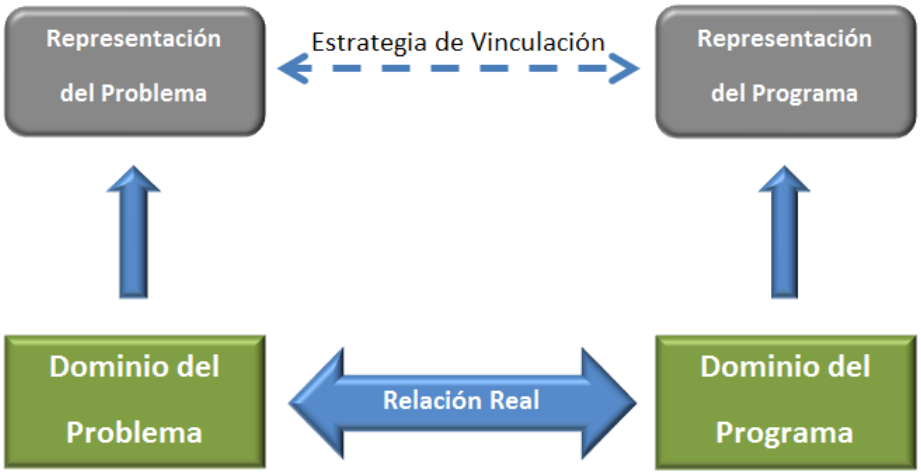

Figura 2.1: Modelo de Comprensión de Programas

conveniente es que este vínculo es complejo de llevar a cabo, por ende se puede bosquejar una aproximación a través de los siguientes pasos: I) Elaborar una representación para el Dominio del Problema; II) Construir una representación del Dominio del Programa y III) Elaborar un procedimiento de vinculación. La Figura 2.1 muestra un modelo de comprensión que refleja lo mencionado previamente.

Para lograr con éxito los pasos antedichos se deben tener en cuenta conceptos muy importantes que son los cimientos sobre los cuales está sustentada la CP: los *modelos cognitivos*, la *extracción de información*, la *administración de la información*, la *visualización de software* y la *interconexión de dominios*. Estos conceptos fueron descriptos brevemente en el capítulo anterior, a continuación se desarrolla una explicación más amplia de cada uno.

2.2. Modelos Cognitivos

Los *Modelos Cognitivos* se refieren a las estrategias de estudio y las estructuras de información usadas por los programadores para comprender los programas [33, 35, 12, 31]. Estos modelos son importantes porque indican de que forma el programador comprende los sistemas y como incorpora nuevos conocimientos (en términos generales aprende nuevos conceptos). Los Mo-

delos Cognitivos están compuestos por: *Conocimientos*, *Modelos Mentales* y *Procesos de Asimilación* [8]. Estos ítems se detallan a continuación.

Conocimientos: Existen dos tipos de conocimientos, por un lado está el conocimiento interno el cual se refiere al conocimiento que el programador ya posee antes de analizar el código del sistema de estudio. Por otro lado existe el conocimiento externo que representa un soporte de información que le brinda al programador nuevos conceptos. Este conocimiento externo es el que proporciona el sistema que se quiere entender, con todos los artefactos asociados al mismo. Los ejemplos más comunes son la documentación del sistema, otros desarrolladores que conocen el dominio del problema, códigos de otros sistemas con similares características, entre otras tantas fuentes de información.

Modelo Mental: Este concepto hace referencia a representaciones mentales que el programador elabora al momento de estudiar el código del sistema. Algunos ejemplos de modelos mentales son: el grafo de llamadas a funciones, el grafo de dependencias de módulos, diagramas de flujo, etc. Es importante mencionar que algunos modelos mentales permiten una representación visual que posibilita que dicho modelo sea exteriorizado. Esta demás decir que esta característica ayuda al programador a entender un programa.

Proceso de Asimilación: Está compuesto por estrategias de aprendizaje que el programador usa para llevar adelante la comprensión de un programa. Los procesos de asimilación se pueden clasificar en tres grupos: Bottom-up, Top-Down e Híbrido [27, 30]. A continuación se explica cada uno.

Bottom-up: El proceso de comprensión *Bottom-up*, indica que el desarrollador primero lee el código del sistema. Luego, mentalmente las líneas de código leídas se agrupan en distintas abstracciones de más alto nivel. Estas abstracciones intermedias se van construyendo hasta lograr una abstracción comprensiva total del sistema.

Top-down: Brooks explica el proceso teórico *top-down*, donde el programador primero elabora hipótesis generales del sistema en base a su conocimiento del mismo. En ese proceso se elaboran nuevas hipótesis las cuales se intentan validar y en ese proceso se generan nuevas hipótesis. Este proceso se repite hasta que se encuentre un trozo de código que implementa la funcionalidad deseada.

Para resumir, el proceso de aprendizaje bottom-up procede de lo más específico (código fuente) a lo más general (abstracciones). Mientras que el top-down es a la inversa.

Híbrido: Este proceso combina los dos conceptos mencionados anteriormente *top-down* y *bottom-up*. Durante este proceso de aprendizaje del sistema, el programador combina libremente ambas políticas (*top-down* y *bottom-up*) hasta lograr comprender el sistema.

La teoría descripta en esta sección sobre Modelos Cognitivos, marca la importancia de esta temática en el ámbito de la CP y de lo difícil que es desarrollar métodos, técnicas y herramientas en este ámbito.

2.3. Extracción de Información

En la Ingeniería del Software existe un área que se encarga de la *Extracción de la Información* desde los sistemas se software. Existen dos tipos de informaciones: la Estática y la Dinámica. A continuación se explica cada una y de que manera pueden ser extraídas desde los programas.

Información Estática: Está presente en los componentes del código fuente del sistema. Alguno de ellos son identificadores, procedimientos, comentarios, documentación. Una forma para extraer la información estática consiste en utilizar técnicas tradicionales de compilación [1]. Estas técnicas, utilizan un analizador sintáctico similar al empleado por un compilador. Este analizador sintáctico, por medio de acciones semánticas específicas procede a capturar elementos presentes en el código durante la fase de compilación. En la actualidad, existen herramientas automáticas que ayudan a construir este tipo de analizador sintáctico.

Información Dinámica: Se basa en elementos del programa presentes durante una ejecución específica del sistema [6]. Un ejemplo conocido de una técnica que extrae información dinámica es la instrumentación de código. Esta técnica inserta sentencias nuevas en el código fuente. La finalidad de las nuevas sentencias es registrar las actividades realizadas durante la ejecución del programa. Estas sentencias nuevas no deben modificar la funcionalidad original del sistema, por ende la inserción debe realizarse con sumo cuidado y de forma estratégica para no alterar el flujo normal de ejecución.

Tanto las estrategias de extracción de información estáticas, como las dinámicas son importantes ya que permiten elaborar técnicas para comprender programas. A veces, se recomienda complementar el uso de ambas estrategias para obtener mejores resultados, sobre todo si el sistema que se está analizando es grande y complejo [15].

Cabe destacar que extraer información de los sistemas implica tomar ciertos recaudos. Si la magnitud de información es demasiado grande se puede dificultar el acceso y el almacenamiento de la misma. Es por esto que se recurre a las técnicas de administración de información. En la próxima sección se explica con más detalles esta afirmación.

2.4. Administración de la Información

Teniendo cuenta que lo sistemas son cada vez más amplios y complejos. El volumen de la información extraída de los sistemas crece notoriamente, por lo tanto se necesita administrar la información.

Las técnicas de administración de información se encargan de brindar estrategias de almacenamiento y acceso eficiente a la información recolectada de los sistemas.

Dependiendo del tamaño de la información se utiliza una determinada estrategia. Estas estrategias indican el tipo de estructura de datos a utilizar y las operaciones de acceso sobre ellas [2, 29]. La eficiencia en espacio de almacenamiento y tiempo de acceso son claves a la hora de elegir una estrategia. Cuando la cantidad de datos son de gran envergadura, se recomienda emplear

una base de datos con índices adecuados para realizar las consultas [16].

Después que se administra la información, se aconseja que la misma sea representada por alguna técnica de visualización. Esta representación, permite esclarecer la información del sistema de una mejor manera para que sea interpretada por el programador. El área encargada de llevar adelante esta tarea es la *visualización de software*.

2.5. Visualización de Software

La *Visualización de Software* es una disciplina de la Ingeniería del Software. Esta disciplina, se encarga de visualizar la información presente en los programas, con el propósito de facilitar el análisis y la comprensión de los mismos. Esta cualidad es interesante ya que en la actualidad los sistemas son cada vez más amplios y complicados de entender. Esta disciplina además brinda apoyo en lo que respecta a la comprensión de las distintas etapas involucradas en el desarrollo de los sistemas, como es el caso del análisis, diseño, implementación y mantenimiento [8]. Por ende, la Visualización de Software colabora en la CP.

Para lograr lo descripto en el párrafo anterior, la Visualización de Software provee distintos sistemas de visualización. Estos sistemas son herramientas útiles que se encargan de analizar los distintos módulos de un programa y generar vistas. Las vistas son una representación visual de la información contenida en el software. Generalmente, una herramienta de comprensión de programas posee varias vistas que ayudan a comprender un programa, dependiendo de la información que se requiera visualizar existe una vista adecuada a cada caso. Las vistas en el contexto de la CP, representan puentes cognitivos que achican la brecha entre los conocimientos del programador y los conceptos usados en el software de estudio.

Para concluir, el objetivo primordial de la visualización de software orientada a la CP es generar vistas (representaciones visuales), que ayuden a reconstruir el vinculo entre el Dominio del Problema y el Dominio del Programa. De manera más sencilla, el objetivo es representar visualmente la salida del sistema, los componentes utilizados para dicha salida y la relación que existe entre ambos.

2.6. Estrategias de Interconexión de Dominios

Los conceptos explicados en los puntos anteriores como la *visualización de software* y la *extracción de la información* forman la base para elaborar estrategias de interrelación de dominios.

La *Interconexión de Dominios* en la Ingeniería del Software, tiene como objetivo principal la transformación y vinculación de un dominio específico en otro dominio. Este último dominio puede estar en un nivel de abstracción alto o bajo. Lo primordial aquí, es que cada componente de un dominio se vea reflejado en uno o más componentes de otro dominio y viceversa.

Un ejemplo sencillo de *Interconexión de Dominios*, es cuando el dominio del código fuente de un programa, se puede transformar en un Grafo de Llamadas a Funciones (Dominio de Grafos). Cada nodo del grafo representa una función particular y cada arco las funciones que puede invocar. En este ejemplo la relación entre ambos dominios (código y grafo) es clara y directa.

Es importante aclarar que existe una amplia gama de transformaciones entre dominios. La más escasa y difícil de conseguir es aquella que relaciona el Dominio del Problema con el Dominio del Programa (Ver Figura 2.1).

Sin embargo, actualmente existen técnicas recientemente elaboradas, que conectan visualmente el Dominio del Problema y el Dominio del Programa mediante la información estática y dinámica que se extrajo del sistema de estudio, algunos ejemplos son *Simultaneous Visualization Strategy* (SVS), *Behavioral-Operational Relation Strategy* (BORS) y *Simultaneous Visualization Strategy Improved* (SVSI) [10, 9, 8]. Estas técnicas son muy útiles para la CP, ya que diagraman la salida del sistema con los componentes que intervinieron en dicha salida.

2.7. Notas y Comentarios

Para resumir las ideas tratadas en este capítulo, el área de la CP le da mucha importancia a la relación entre el Dominio del Problema y el Dominio del Programa. Esta relación ayuda al programador a entender con facilidad los programas, ya que encuentra las partes del sistema que produjeron una determinada salida. Para llevar adelante, la difícil tarea de unir ambos dominios, un camino es construir una representación de cada uno y luego unirlos con un estrategia de relación. Para llevar a cabo estos pasos, se necesitan estudiar las temáticas pertinentes: los *modelos cognitivos*, la *extracción de información*, la *administración de la información*, la *visualización de software* y la *interconexión de dominios*.

Los conceptos antedichos son claves para la CP y sirven como punto de partida para la construcción de Herramientas de Comprensión de Sistemas. Estas herramientas presentan diferentes perspectivas del sistema, facilitando su análisis y su inspección. De esta manera, evitan que el programador invierta tiempo y esfuerzo en entender los módulos de los sistemas. Por ende, se agilizan las tareas de evolución y mantenimiento del software.

En el próximo capítulo se presentan distintas estrategias que se encargan de extraer información de los sistemas, y luego en analizar la información extraída. Este análisis, puntualmente es sobre los identificadores presentes en el código del sistema de estudio. Algunas de estas estrategias están implementadas en forma de Herramientas de CP.

Bibliografía

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.

[2] Alfred V. Aho, Jeffrey D. Ullman, and John E. Hopcroft. *Data structures and algorithms / Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman*. Addison-Wesley Reading, Mass, 1983.

[3] José Luís Albanes, Mario Berón, Pedro Henriques, and Maria João Pereira. Estrategias para relacionar el dominio del problema con el dominio del programa para la comprensión de programas. *XIII Workshop de Investigadores en Ciencias de la Computación*, pages 449–453, 2011.

[4] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering traceability links between code and documentation. *Software Engineering, IEEE Transactions on*, 28(10):970–983, 2002.

[5] Javier Azcurra, Mario Berón, Pedro Henriques, and Maria João Pereira. Análisis de información informal para facilitar la comprensión de programas. *XIV Workshop de Investigadores en Ciencias de la Computación*, pages 597–601, 2012.

[6] Thomas Ball. The concept of dynamic analysis. In Oscar Nierstrasz and Michel Lemoine, editors, *ESEC / SIGSOFT FSE*, volume 1687 of *Lecture Notes in Computer Science*, page 216–234. Springer, 1999.

[7] K. Bennett and V. Rajlich. Software maintenance and evolution: a roadmap. In *ICSE - Future of SE Track*, page 73–87, 2000.

[8] M. Beron, P. Henriques, and R. Uzal. *Inspección de Programas para Interconectar las Vistas Comportamentaly Operacional para la Comprensión de Programas.* PhD thesis, Universidade do Minho, Braga. Portugal, 2010.

[9] Mario Berón, Pedro Henriques, Maria João Pereira, and Roberto Uzal. Program inspection to interconnect behavioral and operational view for program comprehension. University of York, 2007.

[10] Mario Berón, Daniel Eduardo Riesco, Germán Antonio Montejano, Pedro Rangel Henriques, and Maria J Pereira. Estrategias para facilitar la comprensión de programas. In *XII Workshop de Investigadores en Ciencias de la Computación*, 2010.

[11] Dave Binkley, Marcia Davis, Dawn Lawrie, Jonathan Maletic, Christopher Morrell, and Bonita Sharif. The impact of identifier style on effort and comprehension. *Empirical Software Engineering*, 18(2):219–276, 2013. (early access).

[12] R. Brook. A theoretical analysis of the role of documentation in the comprehensionof computer programs. *Proceedings of the 1982 conference on Human factors in computingsystems.*, pages 125–129, 1982.

[13] Bruno Caprile and Paolo Tonella. Nomen est omen: analyzing the language of function identifiers. In *Proc. Sixth Working Conf. on Reverse Engineering*, page 112–122. IEEE, October 1999.

[14] Florian DeiBenbock and Markus Pizka. Concise and consistent naming. In *IWPC*, page 97–106. IEEE Computer Society, 2005.

[15] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Aiding program comprehension by static and dynamic feature analysis. In *ICSM*, pages 602–611, 2001.

[16] Ramez A. Elmasri and Shankrant B. Navathe. *Fundamentals of Database Systems.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 1999.

[17] David W Embley. Toward semantic understanding: an approach based on information extraction ontologies. In *Proceedings of the 15th Australasian database conference-Volume 27*, pages 3–12. Australian Computer Society, Inc., 2004.

[18] E. Enslen, E. Hill, L. Pollock, and K. Vijay-Shanker. Mining source code to automatically split identifiers for software analysis. In *6th IEEE International Working Conference on Mining Software Repositories.*, page 71–80. IEEE, may. 2009.

[19] Henry Feild, David Binkley, and Dawn Lawrie. An empirical comparison of techniques for extracting concept abbreviations from identifiers. In *Proceedings of IASTED International Conference on Software Engineering and Applications.*, 2006.

[20] Henry Feild, David Binkley, and Dawn Lawrie. Identifier splitting: A study of two techniques. In *Proceedings of the Mid-Atlantic Student Workshop on Programming Languages and Systems.*, pages 154–160, 2006.

[21] José Luís Freitas, Daniela da Cruz, and Pedro Rangel Henriques. The role of comments on program comprehension. In *INForum*, 2008.

[22] Wilhelm Hasselbring, Andreas Fuhr, and Volker Riediger. First international workshop on model-driven software migration (mdsm 2011). In *CSMR '11 Proceedings of the 2011 15th European Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering (CSMR 2011)*, pages 299–300, Washington, DC, USA, März 2011. IEEE Computer Society.

[23] Emily Hill, Zachary P. Fry, Haley Boyd, Giriprasad Sridhara, Yana Novikova, Lori Pollock, and K. Vijay-Shanker. Amap: Automatically mining abbreviation expansions in programs to enhance software maintenance tools. In *Proceedings of the 5th Int'l Working Conf. on Mining Software Repositories.*, page 79–88. ACM, 2008.

[24] IEEE. Ieee standard for software maintenance. *IEEE Std 1219-1998*, page i–, 1998.

[25] IEEE. *Standard Glossary of Software Engineering Terminology 610.12-1990.*, volume 1. IEEE Press, 1999.

[26] D. Lawrie, H. Feild, and D. Binkley. Extracting meaning from abbreviated identifiers. In *Source Code Analysis and Manipulation, 2007. SCAM 2007. Seventh IEEE International Working Conference on*, page 213–222, Sept 2007.

[27] Michael P O'brien. Software comprehension–a review & research direction. *Department of Computer Science & Information Systems University of Limerick, Ireland, Technical Report.*, 2003.

[28] Roger S. Pressman. *Software Engineering - A Practitioner's Approach.* McGraw-Hill, 5 edition, 2001.

[29] T.A. Standish. *Data structure techniques.* Computer Sciences. Addison-Wesley, 1980.

[30] M. Storey. Theories, methods and tools in program comprehension: past, present and future. In *Program Comprehension, 2005. IWPC 2005. Proceedings. 13th International Workshop on.*, page 181–191, May 2005.

[31] M. A. Storey, F. D. Fracchia, and H. A. Müller. Cognitive design elements to support the construction of a mental model during software exploration. *The Journal of Systems & Software.*, 44(3):171–185, 1999.

[32] P.F. Tiako. Maintenance in joint software development. In *Computer Software and Applications Conference, 2002. COMPSAC 2002. Proceedings. 26th Annual International.*, page 1077–1080, 2002.

[33] Tim Tiemens. Cognitive models of program comprehension, 1989.

[34] Marco Torchiano, Massimiliano Di Penta, Filippo Ricca, Andrea De Lucia, and Filippo Lanubile. Software migration projects in italian indus-

try: Preliminary results from a state of the practice survey. In *ASE Workshops.*, page 35–42. IEEE, 2008.

[35] A von Mayrhauser and AM. Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, Aug 1995.