Capítulo 2

Comprensión de Programas

2.1. Introducción

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 los conceptos más importantes sobre comprensión de programas (CP). Estos conceptos fueron encontrados en textos que narran sobre CP. Al final se redacta un breve comentario de los temas tratados en el capítulo.

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 está enmarcado en vincular el dominio del problema y el dominio del programa [**?**, **?**, **?**, **?**]. El primero se refiere al resultado de la ejecución del sistema, mientras que el segundo indica todos los componentes de software involucrados que causaron dicho resultado. La figura 2.1 muestra un modelo de comprensión que refleja lo mencionado previamente.

El inconveniente es que la relación real mostrada en la figura 2.1, es compleja de reconstruir, por ende se puede bosquejar una aproximación a través de los siguientes pasos: I) Elaborar una representación para el Dominio

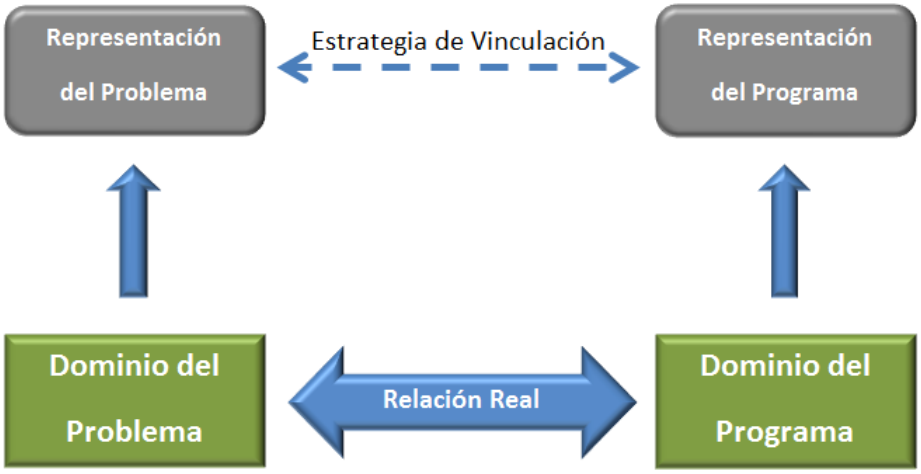

Figura 2.1: Gráfico de Comprensión de Programas

del Problema; II) Construir una representación del Dominio del Programa y III) Elaborar un procedimiento de vinculación.

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. Estos modelos [**?**, **?**, **?**, **?**] son importantes porque indican de que forma el programador comprende los sistemas y como incorpora nuevos conocimientos (aprende nuevos conceptos). Los Modelos Cognitivos están compuestos por: *Conocimientos*, *Modelos Mentales* y *Procesos de Asimilación* [**?**].

Existen 2 tipos de *conocimientos*, uno es 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 externo que le brinda al programador nuevos conceptos. 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 externas.

El concepto *modelo mental* hace referencia a representaciones mentales que el programador elabora al momento de estudiar el código del sistema. Algunos modelos construidos por los arquitectos de software, como por ejemplo el diagrama de clases o el diagrama de casos de uso (entre otros), pueden verse como representaciones visuales de modelos mentales.

El *proceso de asimilación* son las 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 híbrida [**?**, **?**].

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.

Por otro lado, 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 intentar 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.

Por último, el proceso *híbrido* combina los dos conceptos mencionados *top-down* y *bottom-up*. Durante el proceso de aprendizaje del sistema, con el proceso *híbrido*, el programador combina libremente ambas políticas (*top-down* y *bottom-up*) hasta lograr comprender el sistema.

Para concluir sobre la temática de modelos cognitivos: el modelo men-

tal es una representación mental que el programador tiene sobre el sector del sistema que esta analizando. Si esta representación se la vincula con los conocimientos que el propio programador posee, se logra un entendimiento de esa parte del sistema, como así también incrementa los conocimientos del programador. Con esto se deja en claro la importancia de modelos cognitivos en el proceso de entendimiento de los sistemas.

2.3. Extracción de Información

En la Ingeniería del Software existe un área que se encarga de la *Extracción de la Información*. Esta extracción se realiza en los sistemas de software y las técnicas utilizadas para tal fin, se clasifican según el tipo de información que extraen. Existen dos tipos de informaciones: la Estática y la Dinámica.

La información estática está presente en los componentes del código fuente del sistema. Alguno de ellos son identificadores, procedimientos, comentarios, documentación. Una estrategia para extraer la información estática consiste en utilizar técnicas tradicionales de compilación [**?**]. 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.

Por otro lado, la información dinámica se basa en elementos del programa presentes durante una ejecución específica del sistema [**?**]. Una técnica encargada de extraer 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 modifican 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 es-

trategias para obtener mejores resultados, sobre todo si el sistema que se está analizando es grande y complejo [**?**].

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 [**?**, **?**]. 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 [**?**].

Después que se administra la información, se recomienda 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 importante 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.

La *Visualización de Software* provee varias técnicas para implementar 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. Esto permite, interpretar los programas de manera más clara y ágil. Dependiendo de la información que se desea visualizar existe una vista específica [**?**].

En la actualidad, existen distintos tipos de sistemas de visualización, algunos autores son: Myers, Price, Roman, Kenneth, Storey [**?**]. Sin embargo, todos estos sistemas de visualización tienen algo en común, las vistas que son generadas por estos sistemas, representan conceptos situados solo en el dominio del programa, restando importancia al dominio del problema y a la relación entre ambos dominios.

Debido a esta falencia Berón [**?**] propone variantes en los sistemas de visualización orientados a la CP. Estas variantes contemplan la representación visual de los conceptos en el dominio del problema, inclusive la visualización de la relación entre los dos dominios mencionados.

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 (figura 2.1).

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 armar estrategias de interrelación de dominios.

La *Interconexión de Dominios* [**?**, **?**] en la Ingeniería del Software básí-

camente se atribuye a la transformación y vinculación del dominio de la aplicación de software (dominio del problema) con el dominio del programa (ver figura 2.1). Esta afirmación, como se explicó en secciones anteriores es clave en la CP.

El objetivo es que cada componente de un dominio se vea reflejado en uno o más componentes de otro dominio y viceversa.

Un ejemplo 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 (ver figura 2.1). Como se dijo previamente, esta relación es de mucha importancia porque diagrama la salida del sistema y que elementos vinculados intervinieron.

Una de las técnicas es *Simultaneous Visualization Strategy* SVS, esta se encarga de mostrar los distintos componentes de un programa en plena ejecución, mediante distintas vistas, usando un inspector de sentencias, de esta manera se obtiene una visualización cuando el sistema se está ejecutando. Esta estrategia usa un esquema de instrumentación de código, donde las acciones semánticas le van indicando a un visualizador la traza de invocaciones a funciones durante la ejecución del sistema [**?**, **?**, **?**].

La otra estrategia se denomina *Behavioral-Operational Relation Strategy* BORS, que a diferencia del SVS, espera a que termine la ejecución del sistema y luego la información recopilada por el instrumentador de código es procesada por BORS. Una vez procesada la información, BORS retorna una abstracción gráfica del código ejecutado. BORS vincula los conceptos del código capturados en tiempo de ejecución con la información asociada al

dominio del problema [?, ?, ?].

Existe también, a nivel conceptual una técnica que combina ambas estrategias mencionadas llamada *Simultaneous Visualization Strategy Improved* SVSI. Esta técnica disminuye los problemas que manifiestan tanto SVS como BORS y con ello se logra un mejor desempeño en cuanto a los resultados esperados [?, ?, ?].

2.7. 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. Como es demasiado complicado este vínculo, 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.