

Capítulo 2

Comprensión de Programas

2.1. Introducción

En este capítulo se definen algunos conceptos importantes sobre comprensión de programas (CP). 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 inconveniente 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



Figura 2.1: Modelo de Comprensión de Programas

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

Conocimiento: 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 además 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 descrita 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.

Para llevar a cabo estrategias involucradas en Modelos Cognitivos, es importante extraer información desde los códigos de estudio, en el próximo apartado se hace hincapié en este tema.

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 de software. Existen dos tipos de informaciones: la Estática y la Dinámica. A continuación se explica cada una y se da un ejemplo de como 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 los 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 vínculo 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 más alto o más bajo de abstracción. Lo primordial aquí, es que cada componente de un dominio se vea proyectado 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 es clave, por que ayuda al programador a entender con facilidad los programas, ya que encuentra las partes del sistema que produjeron una determinada salida. Dado que los especialistas determinan que es complicado este vínculo [3, 9, 8, 17], un camino para aproximarse a la difícil solución de unir ambos dominios, es construir una representación de cada uno y luego unirlos con una estrategia de relación. Para llevar a cabo estos pasos, se necesitan estudiar las temáticas pertinentes: los *modelos cognitivos* indican como el programador emplea procesos mentales para comprender el sistema; para implementar estos modelos se necesita *extraer información* del software de estudio; si esta información es mucha se recomienda *administrar la información*; generalmente esta información extraída de los sistemas, es más fácil de interpretarla si se emplea la *visualización de software*; por último la *interconexión de dominios* es una pieza clave para elaborar estrategias de CP, mediante la extracción y visualización de la información propia de los sistemas.

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 comprehension of computer programs. *Proceedings of the 1982 conference on Human factors in computing systems.*, 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.