Organización del Conocimiento para un Curso de Programación mediante Patrones de Diseño

J. M. Burgos[1], J. Galve, J. García, M. Sutil
Facultad de Informática
Universidad Politécnica de Madrid
Boadilla del Monte, 28660 - Madrid
{jmburgos, jgalve, juliog}@fi.upm.es

Resumen

La búsqueda de un modelo más expresivo para la representación del conocimiento es un problema que surge cuando se busca una manera de estructurar los contenidos de un curso de programación para organizarlos y transmitirlos de manera más efectiva. La presente propuesta aplica la tecnología de los patrones de diseño para resolver este problema.

1. Antecedentes

Desde hace algún tiempo, venimos trabajando en la organización de los contenidos de un primer curso universitario de programación. A partir del concepto de *esquema de conocimiento* de la sicología cognitiva [AKB79] y basándonos en los trabajos sobre el aprendizaje en el campo de la programación hemos establecido un repertorio de esquemas que modelan dichos contenidos [BGG00b]. Para la definición de nuestros esquemas, hemos utilizado el modelo descriptivo ofrecido por los *patrones de diseño* [BGG00a]. Surgido originalmente en el contexto de la arquitectura [ALE79], en los últimos años este modelo ha venido demostrando su efectividad en la representación de sistemas y procesos complejos como el desarrollo de software [GHJV95], la enseñanza [CL99], la organización de empresas [BEE00] o la gestión documental [RÜP98].

El lenguaje de patrones *De los Problemas a los Programas* es un intento de formalizar mediante patrones el proceso de programación para ir del problema al programa. Este modelo establece un marco para categorizar, especificar y resolver los problemas enseñados comúnmente en un curso de programación básica, enfocado sobre las relaciones que se establecen entre un problema, un esquema de solución y un programa.

2. Motivación

En el enfoque *guiado por la sintaxis*, se enseña a programar al mismo tiempo que se enseña un lenguaje de programación, con lo que se mezclan los conceptos de la computabilidad con la sintaxis del propio lenguaje. Los problemas se muestran como los ejemplos necesarios para ilustrar las construcciones sintácticas y las soluciones se presentan como programas acabados que resuelven los problemas. De este modo, esta colección de problemas deslabazados y programas de última versión se convierten en el material del que disponen los estudiantes para resolver problemas nuevos.

Esta organización de curso no promueve la integración del conocimiento pues ni muestra cómo los problemas se relacionan entre sí, ni cómo los problemas se relacionan con las construcciones del lenguaje de programación, ni tampoco cómo es el vínculo entre el problema y el programa a lo largo del proceso de programación. Se presta muy poca atención a las habilidades que

[1] Este trabajo ha sido financiado por el proyecto de Innovación Educativa: *"Herramientas Metodológicas e Instrumentales para la Enseñanza de la Programación"* (F.G.UPM-43700000190).

maneja un programador experto para desarrollar un programa, desde que se plantea el problema como objetivos, establece una abstracción intermedia de solución o esquemas y llega finalmente al programa.

El objetivo central de este trabajo es el de encontrar una buena organización del conocimiento que se transmite en un primer curso de programación, que lo estructure y además facilite su aprendizaje y transmisión a programadores noveles. Las investigaciones hechas sobre el modo en que se aprende y se maneja en la mente el conocimiento sugieren que su organización debe reunir tres características:

- Que sea *abstracto*: Debe permitir tener representaciones profundas, semánticas, abstractas y conceptuales del conocimiento. Cuanto más abstractas y conceptuales sean las categorías que se manejen, más pequeño será su número, su nivel de organización será más alto y permitirá detectar más fácilmente las similitudes profundas entre los esquemas.
- Que esté altamente *cohesionado*: Debe permitir interconectar de diversas maneras el conocimiento, pues esto ayuda extrordinariamente a asimilar, memorizar, e integrar lo aprendido, así como acceder a dicho conocimiento en nuestra mente y utilizarlo de manera más versátil y fructífera.
- Que sea generativo: Debe facilitar la generación de soluciones de mayor dimensión mediante la combinación de los esquemas, así como la incorporación de otros nuevos.

Tras presentar algunos conceptos e ideas generales sobre los patrones y lenguajes de patrones como modelo de conocimiento en la sección 3, en la sección 4 se presenta la propuesta del lenguaje de patrones *De los Problemas a los Programas* [BG00a]. En la sección 5 se dan pautas de utilización de este lenguaje en un curso de programación para terminar con las conclusiones en la sección 6.

3. Patrones y Lenguajes de Patrones

En los últimos años, se ha generado una fuerte corriente de trabajo sobre patrones y lenguajes de patrones en el mundo del desarrollo de software. Este movimiento en informática se basa en los trabajos del arquitecto y profesor Christopher Alexander, promotor de estas ideas en el mundo de la arquitectura [ALE79].

Un patrón de diseño es una solución a un problema en un contexto. Para tener el grado de generalidad que haga al patrón aplicable a muchos problemas distintos, se enuncia mediante una abstracción de una forma concreta de solución que se repite en contextos específicos. Pero un patrón es algo más que una solución probada a un problema recurrente. El problema ocurre en un determinado contexto y en presencia de diversas restricciones o ligaduras que muchas veces se contradicen. La solución propuesta conlleva una estructura que compromete a las restricciones para establecer un equilibrio entre estas “fuerzas”. Usando la forma del patrón, la descripción de la solución intenta capturar el conocimiento detallado de resolución de problemas en un dominio poseído por un experto a otros que se podrían beneficiar de ese conocimiento para aplicarlo en otro dominio. Hasta cierto punto, un patrón es un intento de establecer de una manera clara y expresiva, el modo de actuar frente a un problema o una clase de problemas dictado por el “buen oficio”. Al patrón se le refiere por un nombre que sirve como un “asa conceptual” por donde agarrarlo y facilitar el diálogo sobre la perla de información que guarda, estableciendo un vocabulario sencillo y unificado.

Los patrones se pueden aglutinar formando colecciones que constituyen un vocabulario para comprender y comunicar ideas. Cuando tales colecciones se tejen con habilidad para formar un todo cohesionado que revele las estructuras y las relaciones de sus componentes para cumplir un objetivo compartido, entonces estamos hablando de lo que Alexander bautiza como *lenguaje de patrones*. Un lenguaje de patrones define una colección de patrones y las reglas y pautas para combinarlos con un estilo que podríamos denominar “arquitéctónico”, pues define una forma de construir estructuras a todos los niveles de escala y en todos los grados de diversidad. En realidad, un lenguaje de patrones se compone de un léxico de patrones y una gramática que establece cómo unirlos para formar estructuras

sintácticas. Idealmente, los buenos lenguajes de patrones son *generativos*, es decir, capaces de generar todas las posibles frases a partir de un vocabulario de patrones rico y expresivo. En palabras de Alexander: *"No solamente nos dicen qué reglas hay que seguir en las soluciones sino que nos muestran cómo construir soluciones –tantas como queramos- que satisfagan las reglas"* [ALE79].

Un lenguaje de patrones forma una entidad *orgánica*, en el sentido de que, además de que los patrones colaboran a resolver un problema de orden superior que no es explícitamente resuelto por ninguno de ellos individualmente, existe un equilibrio "perfecto" entre las necesidades de las partes y las necesidades del todo, formando un todo morfológica y funcionalmente completo. Los lenguajes de patrones definen un sistema *hologramático* [MOR90] en el sentido de la idea formulada por Pascal: *"No puedo concebir al todo sin concebir a las partes y no puedo concebir a las partes sin concebir al todo"*. El conocimiento aportado por cada patrón reentra sobre el lenguaje y lo que aprehendemos sobre las cualidades emergentes del lenguaje reentra sobre los patrones. El conocimiento de los patrones se enriquece por el del lenguaje y el del lenguaje por los patrones, en un mismo movimiento productor de conocimientos. Un ejemplo de esto es el efecto de los *comportamientos emergentes* [BEE00]: patrones que surgen de una espontánea interacción local muy densa entre entidades, lo cual da como resultado sistemas autoorganizativos que son adaptativos, abiertos y capaces de efectos multiescala. En otras palabras, los lenguajes de patrones ofrecen un proceso dinámico para la resolución de problemas dentro de su dominio que indirectamente conduce a la resolución de un problema mucho más amplio.

4. El Lenguaje de Patrones *De Los Problemas a Los Programas*

El lenguaje de patrones *De los Problemas a los Programas* se basa en una taxonomía de problemas [BG00b]. Esta taxonomía está establecida a partir de las similitudes entre los problemas, mediante observación de las especificaciones, lo cual permite un alto nivel de abstracción de los patrones. El lenguaje ofrece una guía para ir de la especificación del problema a un al programa pasando por un esquema de solución que pauta este proceso.

Cada patrón de diseño intenta encapsular un *retazo de conocimiento* ("chunk of knowledge" [SOL86]), una plasmación de la pericia del experto. En definitiva, el lenguaje está enfocado sobre el nivel *pragmático* de un lenguaje de programación, es decir, en los "aspectos prácticos" de la actividad de programar.

Los patrones están estructurados en tres grupos:

- Especificación: Un conjunto de patrones que que sirven de ayuda para analizar el problema. Son los siguientes:
  1. *Descripción De los Datos*: Cómo describir los datos implicados.
  2. *Descripción Del Problema*: Cómo describir la funcionalidad del problema.
  3. *Problema Categorizado*: Cómo clasificar el problema en la taxonomía.
- Diseño: Un conjunto de patrones para diseñar soluciones a los problemas previamente especificados. Permiten la descripción de la solución en una notación algorítmica más abstracta que el lenguaje de programación aunque similar a él que permite omitir detalles que no son relevantes.
  4. *Entrada-Proceso-Salida*: Cómo describir una solución teniendo en cuenta los tres componentes básicos de un cómputo.
  5. *Análisis de Casos*: Cómo escribir una solución para un problema de análisis de casos.
  6. *Acumulación*: Cómo escribir una solución para un problema de acumulación.
  7. *Búsqueda*: Cómo escribir una solución para un problema de búsqueda.
  8. *Extremos*: Cómo escribir una solución para un problema de máximos o mínimos.
  9. *Construcción*: Cómo escribir una solución para un problema de construcción de una colección de datos.
- Implementación: Un conjunto de patrones que ofrecen las pautas necesarias para implementar las soluciones. Son fuertemente dependientes del lenguaje de programación.

En [ELE] se ofrece un amplio repertorio de estos patrones. Se estructuran en cuatro grandes grupos:

10. *Selección*: Cómo escribir programas usando estructuras de control de selección [BER99].
11. *Iteración*: Cómo escribir programas usando bucles [PRO00].
12. *Recurrencia:* Cómo escribir programas usando recursividad y técnicas asociadas [WAL97].
13. *Recorridos:* Cómo escribir programas en los que hay que recorrer dominios de datos [PRO00].

En el apéndice se presenta un ejemplo de patrón: el patrón *Acumulación.*

4. Utilización de los Patrones

Cada patrón ofrece esquemas abstractos de solución mediante la sintaxis de un lenguaje de programación. En nuestro curso en la Universidad Politécnica de Madrid usamos el lenguaje de programación Ada-95, por lo que los patrones hacen uso de él. Se utiliza el formalismo de función, pero solamente para facilitar la comunicación de las ideas, pues es un mecanismo simple, potente, rigursos y suficientemente conocido de la Matemática elemental. Esto no quiere decir que a la hora de usar el patrón no se puedan aplicar sus ideas mediante un procedimiento o simplemente un fragmento de código.

Para utilizar los patrones propuestos en un curso de programación hay que adaptar mínimamente este material en función del contexto en el que se desarrolle el curso. Los patrones se pueden utilizar como apuntes de clase, ya sea para explicarlos directamente o para recapitular cada cierto tiempo sobre los contenidos explicados. También se pueden usar como material complementario a las clases magistrales. En cualquier caso, su utilización debe ser secuenciada yendo siempre de lo más concreto a lo más abstracto, es decir, empezando por los esquemas de implementación para continuar con los de diseño y recapitular con los de especificación.

Podemos utilizar los patrones propuestos siguiendo un modelo de aprendizaje como el expuesto en [AKB79], según el cual, empezaremos por mostrar los mecanismos de implementación que permitan comprender los patrones de implementación y así posteriormente presentar los patrones de diseño (etapa cognitiva). Gracias a las pautas dadas por los patrones de diseño, el estudiante adquiere habilidades para combinar las construcciones del lenguaje y resolver problemas.

Una vez mostrados los patrones de la etapa de diseño, se aplican a una amplia variedad de problemas razonablemente complejos y se definen estrategias de combinación. El estudiante adquiere las habilidades necesarias para saber qué esquema aplicar ante un determinado problema (etapa asociativa). Aquí se desarrollan las habilidades para detectar cómo, cuándo y por qué aplicar los patrones, cómo combinarlos y cómo considerarlos al nivel de abstracción necesario para usarlos en diversos ámbitos de aplicación. Estas habilidades son las que marcan la diferencia de conocimiento entre el experto y el que no lo es. Sin ellas no se puede hacer un mínimo plan o diseño del programa y por tanto, resulta casi imposible abordar problemas de cierta complejidad.

En las etapas más avanzada del aprendizaje es cuando se construyen nuevos esquemas mentales de conocimiento, creando soluciones a problemas nuevos, se empieza a poder considerar varias soluciones alternativas a un mismo problema y se aprende a elaborar las soluciones de manera autónoma.

6. Conclusiones

En este trabajo hemos presentado una propuesta para dotar de estructura el conocimiento de un primer curso de programación y así mejorar la efectividad de la enseñanza y el aprendizaje. La propuesta se basa en lo que se sabe acerca de cómo las personas aprenden, e intenta ilustrar el conocimiento que se vierte en un primer curso de programación, haciéndolo explícito al formularlo con un lenguaje de patrones.

Por supuesto, se asume que el repertorio de patrones propuesto no es ni único, ni canónico, ni

completo, pero como dice Soloway en [SOL85]: "encontrar excepciones y desarrollar nuevas estructuras para acomodar las inconsistencias es una poderosa técnica de aprendizaje" y "reconocer que las abstracciones admiten excepciones es precisamente una clase de habilidad de resolución de problemas que uno quiere fomentar". La propuesta no está cerrada ni acabada. En futuros trabajos se identificarán y elaborarán otros patrones que complementen el material presentado en éste.

Referencias

[AKB79] J. R. Anderson, P. J. Kline y C. M. Beasley. A General Learning Theory and Its Application to Schema Abstraction. En G. H. Bower (ed.). *The Psychology of Learning and Motivation,* vol. 13, New York Academic Pres (1979), pp. 277-318.

[ALE79] C. Alexander: *The Timeless Way of Building*, Oxford University Press, 1979.

[BEE00] M. Beedle. SCRUM: A Pattern Language for Hyperproductive Software Development. Pattern Languages of Program Design 4. Addison-Wesley 2000.

[BER99] Bergin, J. Patterns for Selection. Proceedings of the 4th European Conference on Patterns Languages of Programming and Computing, 1999 (EuroPLoP´99).

[BGG00a] J.M. Burgos, J.Galve y J. García. From Problems to Programs: A Pattern Language to Go from Problem Requirements to Solution Schemas in Elementary Programming. *EuroPLoP'2000.*

[BGG00b] J.M. Burgos, J.Galve y J. García. Una Taxonomía de Problemas para la Enseñanza de la Programación, *Actas del VII Congreso Internacional de Informática en la Educación INFOREDU'2000*, Mayo 2000, La Habana (Cuba).

[CL99] M.J Clancy y M.C. Linn. Patterns and Pedagogy. Proceedings of the 30th SIGCSE Technical Symposium on Computer Science Education, 1999, 37-42.

[ELE] Página de Internet de los Patrones Elementales: http://www.cs.uni.edu/%7Ewallingf/patterns/elementary/

[GHJV95] E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[MOR90] E.Morín. Introducción al Pensamiento Complejo. Gedisa Eds. 1990.

[PRO00] V. K. Proulx. Programming Patterns and Design Patterns in the Introductory Computer Science Course, *Proceedings of the 31st SIGCSE Technical Symposium on Computer Science Education*,2000,80-84. http://www.ccs.neu.edu/teaching/EdGroup/

[RÜP98] A. Rüping. Writing and Reviewing Technical Documents. Proceedings of the 3rd EuroPLoP´98.

[SOL85] E. Soloway. From Problems to Programs Via Plans: The Content and Structure of Knowledge for Introductory Lisp Programming. *Journal of Educational Computing Research, 1(2),* 1985, 157-172.

[SOL86] E. Soloway. Learning to Program = Learning to Construct Mechanisms and Explanations. *Communications of the ACM*, 29, 9 (Sept. 1986), 850-858.

[WAL97] E. Wallingford. RoundAbout: A Pattern Language for Recursive Programming. *Proceedings of the 4th Patterns Languages of Programming Conference,* September 1997 (PLoP'97), Allerton Park, Illinois.

Apéndice: El Patrón *Acumulación*

Contexto

Este patrón es aplicable a problemas cuya especificación es del tipo:

"Calcular el resultado de acumular (mediante una operación de acumulación como la suma) la aplicación de un cómputo sobre los elementos de una colección que cumplen una propiedad".

Motivación

El cálculo de un sumatorio es un tipo de problema muy común de las Matemáticas. En este tipo de problema, se trata de realizar la suma de una una expresión en la que existe al menos una variable que se mueve en un rango de valores. Por ejemplo:

> *Calcular el resultado del sumatorio*
> $\sum (i^3 + 1), i = 1..N$

Este tipo de problemas también aparecen frecuentemente en programación y se les encuadra en el tipo general de problemas conocido como *problemas de acumulación.* Un ejemplo de un problema de acumulación es también:

> *Calcular las puntos que suman los oros de una mano de cartas.*

Los problemas de esta clase consisten en aplicar un cómputo a todos los elementos de una colección de datos y combinar todos esos resultados parciales mediante una operación binaria (normalmente la suma o el producto) para obtener un resultado global. El tipo del resultado global es el mismo que el tipo del resultado parcial de la función de cómputo.

Para poder obtener el resultado de la acumulación, la colección tiene que ser recorrida completamente. Este tipo de problemas no impone ninguna precondición sobre la colección de datos.

Estructura de la Solución Recursiva

La solución recursiva para este tipo de problemas se ajusta a la siguiente plantilla general:

```
function Acumulación
   (Datos : Tipo_De_Los_Datos)
   return Tipo_Del_Resultado is
begin
  if Condición_Básica (Datos) then
     return Solución_Básica (Datos);
  elsif Propiedad (Datos) then
     return
       Cómputo (Datos)
       ⊕ Acumulación
              (Siguiente (Datos)));
  else
     return Acumulación
              (Siguiente (Datos)));
  end if;
end Acumulación;
```

Podemos distinguir elementos de cómputo y de recorrido. Los elementos de cómputo son:

- `Cómputo:` El proceso a realizar con cada elemento.
- El operador ⊕: Permite combinar los resultados parciales para obtener el resultado global.
- `Solución_Básica:` Lo que devuelve la función la última vez que se la llama. Suele ser el elemento neutro de la operación de acumulación (0 para la suma y 1para el producto).

Los elementos de recorrido son:

- El predicado `Condición_Básica:` Representa la condición que se tiene que cumplir para poder decir que el recorrido de la colección se ha terminado, esto es, que la colección esté vacía.
- La función de tranformación `Siguiente:` Cómo avanzar para recorrer todos los datos de la colección.
- El predicado `Propiedad:` Dice qué elementos del dominio no hay que considerar (mediante filtrado).

Siempre es posible reconvertir el esquema de solución a uno con dos ramas en el que la llamada recursiva está en un único punto:

```
function Acumulación
   (Datos : Tipo_De_Los_Datos)
   return Tipo_Del_Resultado is
begin
  if Condición_Básica (Datos) then
     return Solución_Básica (Datos);
  else
     return
       Delta (Datos) ⊕
       Acumulación
        (Siguiente (Datos)));
  end if;
end Acumulación;
```

donde la función Delta es la encargada de tomar un elemento, comprobar la propiedad para, en caso de que se cumpla, aplicarle el `Cómputo` y, en caso contrario, devolver el neutro de la acumulación.