Entre Ciencia e Ingeniería, ISSN 1909-8367
Año 4. No. 8 – Segundo semestre de 2010, páginas 161 - 173

Patrones Grasp y Anti-Patrones: un Enfoque Orientado a Objetos desde Lógica de Programación[1]

Grasp Patterns and Anti-Patterns: an Object Oriented Approach from Logic Programming

Ricardo Botero Tabares
Estudiante de Maestría en Ingeniería (Área Sistemas y Computación)
Especialista en Didáctica Universitaria
Ingeniero de Sistemas
Profesor Asociado Tecnológico de Antioquia – Institución Universitaria
Grupo de Investigación en Ingeniería de Software - GIISTA
rbotero@tdea.edu.co

Recibido Agosto 15 de 2010 – Aceptado Diciembre 13 de 2010

SÍNTESIS

El presente artículo plantea una estrategia didáctica para introducir el uso de los patrones GRASP (General Responsibility Assignment Software Patterns) y el desuso de los anti-patrones de diseño en un curso de "fundamentos de programación orientada a objetos", donde se solucionan problemas siguiendo cuatro fases a saber: identificación de requerimientos, diseño del diagrama de clases, especificación de responsabilidades y escritura de seudocódigo.

Descriptores: *patrones GRASP, anti-patrón de diseño, programación orientada a objetos.*

1 Documento derivado del proyecto de investigación "Sistema para el Modelamiento por Objetos - SISMOO", aprobado en convocatoria 01 de 2008, desarrollado por el Grupo de Investigación en Ingeniería de Software del Tecnológico de Antioquia – GIISTA.

ABSTRACT

This article presents a teaching strategy for introducing the use of patterns GRASP (General Responsibility Assignment Software Patterns) and the disuse of anti-design patterns in the course "fundamentals object-oriented programming", where problems are solved according to four phases: identification of requirements, design class diagram, specification of responsibilities and pseudocode writing.

Descriptors: GRASP patterns, anti-pattern design, object-oriented programming.

1. INTRODUCCIÓN

El título de este artículo parecería extraño, porque ¿cómo tratar los patrones GRASP y los anti patrones de diseño en un curso de fundamentos de programación, cuando estos temas se acoplan mejor a cursos de niveles superiores relacionados con ingeniería de software?

La respuesta es evidente: los cursos relacionados con fundamentos de programación para educación superior deben responder a los avances tecnológicos y científicos del sector del software. Así como algunos cursos universitarios sobre lenguajes de programación históricamente han pasado de Fortran a Pascal, de Pascal a C, de C a C++ y de éste a Java o C#, asimismo los cursos de lógica de programación han ido desde la programación libre a la programación estructurada y de ésta a la programación orientada a objetos. Es cierto que los cambios en un curso o laboratorio de lenguajes de programación se vislumbran mejor que aquellos en un curso de lógica, dado que una renovación de lenguaje conlleva menos traumatismos cognitivos que un cambio de paradigma.

A continuación, se exponen argumentos que justifican el estudio de los patrones y anti patrones de diseño, sobre todo los primeros, en un curso de iniciación a las ciencias de la computación.

2. DESCRIPCIÓN DEL TRABAJO

El estudio de los patrones de diseño en un curso de fundamentos de programación surge de la experiencia con los proyectos MIPSOO (Botero R., Castro C. & Parra E., 2008) y SISMOO (Botero R., Castro C. & Taborda, G., 2009), desarrollados en la Facultad de Informática del Tecnológico de Antioquia, por investigadores del Grupo de Investigación en Ingeniería de Software - GIISTA. Coadyuvó el adelanto de un curso para docentes basado en éstos proyectos, relacionado con didácticas ajustadas a las tendencias actuales para el desarrollo de aplicaciones informáticas bajo el paradigma de programación orientado a objetos, propiciado en el contexto del proyecto Alianza Futuro Digital Medellín (Alianza Futuro Digital Medellín, 2010), donde participaron profesionales docentes vinculados a la educación media técnica y superior.

El proyecto MIPSOO propone un método para el aprendizaje – e inherente enseñanza – de la programación orientada a objetos que conlleva cuatro pasos para la solución de problemas:

a) Identificación de requerimientos: ¿Qué necesidades plantea el usuario?

b) Diseño del diagrama de clases: ¿Cómo puedo representar gráficamente las abstracciones de la realidad del problema que se pueden convertir en software?

c) Especificación de las responsabilidades de las clases: ¿De qué se responsabiliza o qué servicios presta cada clase definida en el diagrama de clases?

d) Escritura de seudocódigo para clases de uso no común: ¿Cómo transmitir la solución a la computadora sin los formalismos de un lenguaje de programación?

En el proyecto SISMOO se desarrolla un software para editar, compilar y ejecutar programas escritos con el seudocódigo planteado en MIPSOO, con Java tras bambalinas.

3. RESULTADOS

Durante el año 2010, el método se aplicó a un número aproximado de 250 estudiantes matriculados en el módulo Desarrollar Pensamiento Analítico Sistémico I del programa Tecnología en Sistemas del Tecnológico de Antioquia, distribuidos en tres grupos de la jornada diurna y dos de la mixta (mañana y noche), cada grupo con un tamaño cercano a 25 jóvenes en formación. Finalizados los semestres académicos, se encontró una respuesta positiva del alumnado en relación con el alcance de las competencias académicas visibles en los indicadores de logro, por cuanto se disminuyeron los índices de deserción y la cantidad de repitentes con respecto a los años anteriores, cuando se utilizaban métodos convencionales para la enseñanza de la programación.

Además, en el segundo semestre de 2010 se observó una actitud madura de los estudiantes al enfrentar los primeros retos de programación con un lenguaje de producción orientado a objetos como Java, dentro del módulo Construir Elementos de Software I, cuyo prerrequisito es Desarrollar Pensamiento Analítico Sistémico I. Éste último módulo aporta las competencias específicas, básicas y transversales necesarias para que el estudiante incursione con éxito en la solución de problemas, mediante la aplicación de los conceptos inherentes al paradigma de programación orientado a objetos, enfrentando problemas como el expuesto a continuación.

3.1 Un problema típico: la ecuación de Einstein

“La famosa ecuación de Einstein para conversión de una masa m en energía, viene dada por la fórmula $E = mc^2$, donde c es la velocidad de la luz, c = 2.997925 x 1010 m/s.

Leer la masa de un objeto en gramos y obtener la cantidad de energía producida en ergios.”

La solución a éste problema se presenta siguiendo las cuatro

etapas planteadas en MIPSOO:

a) Identificación de requerimientos

Se identifican dos requerimientos, descritos en la tabla 1.

Tabla 1: Requerimientos para el problema de la ecuación de Einstein

Identificación del requerimiento	Descripción	Entradas	Salidas
R1	Conocer la masa del objeto en gramos.	Un número real digitado por el usuario.	La masa del objeto está almacenada en memoria.
R2	Calcular la cantidad de energía producida por un objeto.	La masa del objeto (en gramos).	La energía del objeto (en ergios).

b) Diseño del diagrama de clases

El diagrama de clases de la figura 1 conlleva la definición de las abstracciones *Energía y Proyecto,* y a la reutilización de las clases de uso común Flujo y Mat.

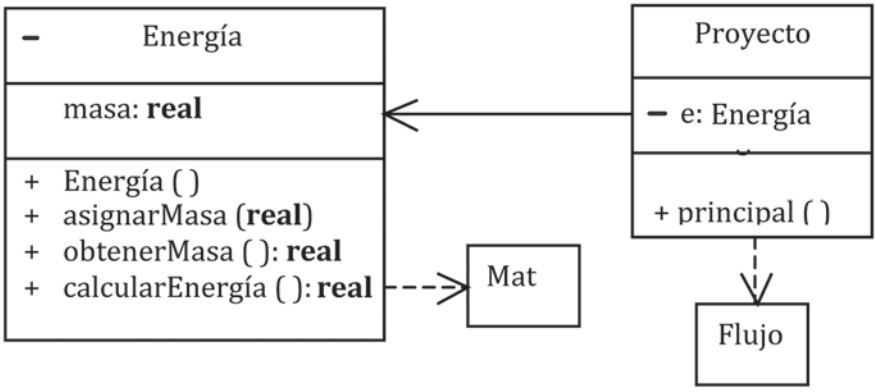

Figura 1: Diagrama de clases para el problema de la ecuación de Einstein

c) Especificación de las responsabilidades de las clases

Las responsabilidades de las clases se expresan mediante los contratos de cada uno de sus métodos. En términos generales, la clase Energía es responsable de almacenar la masa del objeto y calcular su energía; la clase Proyecto es responsable de establecer comunicación con el usuario para la captura de

la masa del objeto, crearlo, asignarle un estado y visualizar resultados para cumplir con los requerimientos. Las tablas 2 y 3 presentan los contratos de las clases de uso no común Energía y Proyecto.

Tabla 2: Contratos de la clase Energía

Método	Requerimiento asociado	Precondición	Postcondición
Energía	R1	No existe un objeto para calcularle su energía.	Existe un objeto en memoria listo para ser procesado.
asignarMasa	R1	El objeto tiene una masa igual a 0 (cero)	El objeto tiene una masa (número real positivo) igual a la especificada por el usuario.
obtenerMasa	R2	El objeto tiene una masa distinta de cero, desconocida para el mundo exterior al objeto.	El mundo exterior al objeto conoce la masa del objeto.
calcularEnergía	R2	Se desconoce la energía producida por el objeto.	Se conoce la energía producida por el objeto.

Tabla 3: Contratos de la clase Proyecto

Método	Requerimiento asociado	Precondición	Postcondición
principal	R1 y R2	No hay entradas de datos y no se ha efectuado proceso alguno.	Se tiene un objeto en memoria con una masa definida y una energía conocida.

Las responsabilidades de las clases de utilidad Flujo y Mat, conocidas también como clases de uso común, no se definen de forma explícita porque se asumen comprendidas por el analista: Flujo se responsabiliza de las operaciones de entrada y salida estándar y Mat de las operaciones con funciones matemáticas.

d) Escritura de seudocódigo

El seudocódigo guarda similitud con la sintaxis de lenguajes de producción como Java y Visual Basic.Net; es un buen preámbulo a la etapa de codificación en alguno de éstos u otros lenguajes. La figura 2 expone el seudocódigo para este problema, donde las palabras reservadas del seudolenguaje se escriben en negrita, en contraposición a los demás identificadores para nombres de clase – Energía y Proyecto –, atributos – masa y e –, métodos – Energía ( ), asignarMasa ( ), obtenerMasa ( ) y calcularEnergía( ) – , variables locales – ener – y objetos –e.

```
clase Energía
   privado real masa
   público Energía( )
   fin_metodo
   //------------------------------
   asignarMasa(real m)
      masa = m
   fin_metodo
   //------------------------------
   real obtenerMasa( )
      retornar masa
   fin_metodo
   //------------------------------
   real calcularEnergía( )
      real ener
      const real C = 2.997925 * Mat.elevar (10, 10)
      ener = masa * Mat.elevar(C, 2)
      retornar ener
   fin_metodo
fin_clase
//-----------------------------------------------------------------------------------
clase Proyecto
   Energía e
   estatico principal( )
      e = nuevo Energía( )
      Flujo.imprimir ("Ingrese la masa del objeto en gramos:")
      real ms = Flujo.leerReal( )
      e.asignarMasa(ms)
      Flujo.imprimir("Energía producida = " + e.calcularEnergía( ) + " ergios")
   fin_metodo
fin_clase
```

Figura 2: Seudocódigo para el problema de la ecuación de Einstein

3.2. Patrones GRASP y anti-patrones en lógica de programación

En este artículo se presta particular atención a la manera como se pueden introducir los patrones GRASP (Patrones Generales de Software para Asignación de Responsabilidades) y los anti-patrones en un curso de lógica de programación. Los patrones GoF (Banda de los Cuatro) o patrones Gamma (Gamma E., Helm R., Johnson R. & Vlissides J., 1994), clasificados en creacionales, estructurales y de comportamiento, se pueden tratar en cursos de niveles superiores relacionados con ingeniería de software (Bennett S., McRobb S. & Farmer R., 2007).

3.2.1 Patrones GRASP

Los patrones GRASP, más que patrones propiamente dichos, son una serie de "buenas prácticas" de aplicación recomendable en el diseño de software. Entre ellos, los patrones *Experto, Creador, Bajo acoplamiento y Alta cohesión* pueden ser tratados en un curso de lógica de programación, porque guardan directa relación con la creación y asignación de responsabilidades a los objetos. Sin embargo, el patrón *Controlador,* que también conforma los patrones GRASP, se puede estudiar en un curso superior de ingeniería de software o en un laboratorio de programación, dado que sirve como intermediario entre la interfaz de usuario y las clases que encapsulan los datos.

Todo patrón tiene un nombre, plantea un problema y aporta una solución. Así, el patrón *Experto* plantea el problema "¿Cuál es el principio fundamental mediante el cual se asignan responsabilidades a los objetos?" y aporta la solución "Asignar la responsabilidad a la clase que tiene la información necesaria para cumplirla" (Larman, C., 1999). La tabla 4 expone cuatro patrones GRASP que se aplican al caso de la ecuación de Einstein, con el problema que resuelve y su respectiva solución.

Tabla 4: Patrones GRASP comprensibles en un curso de lógica de programación orientada a objetos

Nombre del patrón	Problema que resuelve	Solución
Experto	¿Cuál es el principio fundamental mediante el cual se asignan responsabilidades a los objetos?	Asignar la responsabilidad a la clase que tiene la información necesaria para cumplirla.
Creador	¿Quién debe ser responsable de crear una instancia de alguna clase?	Asignarle a la clase C2 la responsabilidad de crear instancias de la clase C1 en uno de los siguientes casos: I. C2 agrega los objetos de C1. II. C2 contiene los objetos de C1. III. C2 registra las instancias de los objetos de C1. IV. C2 utiliza específicamente los objetos de C1. V. C2 tiene los datos de inicialización que serán transmitidos a C1 cuando este objeto sea creado. De iure, C2 es un creador de los objetos de C1.
Bajo acoplamiento	¿Cómo dar soporte a una dependencia escasa y a un aumento de la reutilización?	Asignar una responsabilidad para mantener bajo acoplamiento.
Alta cohesión	¿Cómo mantener la complejidad dentro de límites manejables?	Asignar una responsabilidad de modo que la cohesión siga siendo alta.

El patrón *Experto* posibilita una adecuada asignación de responsabilidades facilitando la comprensión del sistema, su mantenimiento y adaptación a los cambios con reutilización de componentes. El patrón Creador aporta un principio general para la creación de objetos, una de las actividades más frecuentes en programación. El patrón *Bajo Acoplamiento* es una medida de la fuerza con que una clase se relaciona con otras, porque las conoce y recurre a ellas; una clase con bajo acoplamiento no depende de muchas otras, mientras que otra con alto acoplamiento presenta varios inconvenientes: es difícil entender cuando está aislada, es ardua de reutilizar porque requiere la presencia de otras clases con las que esté conectada y es cambiante a nivel local cuando se modifican las clases afines. El patrón *Alta Cohesión* es una medida que determina cuán relacionadas y adecuadas están las responsabilidades de una clase, de manera que no realice un trabajo colosal; una clase con baja cohesión realiza un trabajo excesivo, haciéndola difícil de comprender, reutilizar y conservar.

Los patrones de la tabla 4 encuentran clara aplicación en el ejemplo de la fórmula de Einstein para el cálculo de la energía producida por un objeto material. La clase Energía es experta en calcular la energía producida por un objeto de masa conocida, es decir, tiene la responsabilidad de calcular un valor en ergios equivalente a la energía de un objeto con peso en gramos, a través de su método calcularEnergía( ). La clase Proyecto contiene el algoritmo principal ( ), único método en una aplicación orientada a objetos que no admite sobrecarga; esta clase es responsable de crear objetos de la clase Energía. De hecho la clase que contiene el método principal ( ) siempre será un patrón Creador. La clase Energía tiene bajo acoplamiento porque se relaciona con pocas clases, para el caso sólo con la clase de utilidad Mat, cuya interfaz pública la componen métodos estáticos que sirven para el cálculo de funciones aritméticas. Las clases de uso común Mat y Flujo, cuya vista parcial se presenta en la figura 3, son responsables de realizar cálculos con funciones aritméticas y de posibilitar la entrada/salida estándar, respectivamente. Así, el bajo acoplamiento de las clases Energía, Mat y Flujo incentivan su reutilización en otras situaciones, contextos o problemas.

Finalmente, la clase Energía posee alta cohesión porque el trabajo que realiza, valga decir su responsabilidad, está bien definida. Toda clase de uso común, como Mat y Flujo, tiene bajo acoplamiento y alta cohesión.

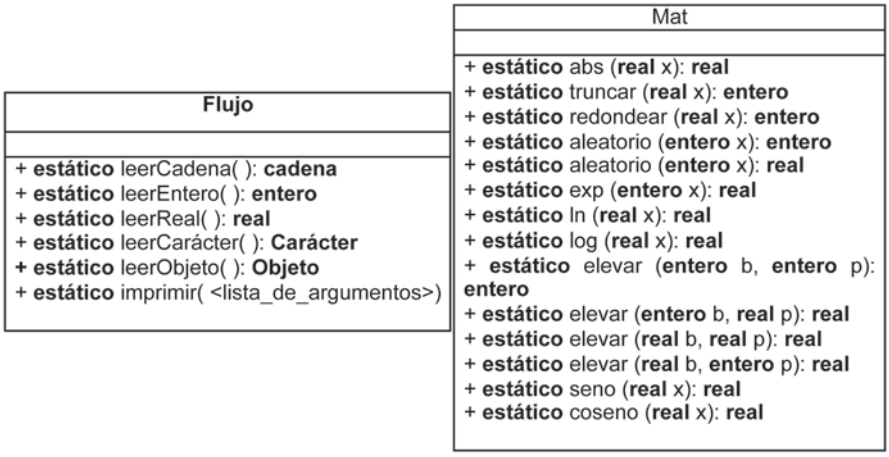

Figura 3: Vista parcial de las clases de uso común Flujo y Mat.

3.2.2 Anti-patrones de diseño

Un anti-patrón de diseño es un patrón que conduce a una mala solución de un problema; la documentación de un anti patrón aporta argumentos a los diseñadores de sistemas para no elegir malos hábitos. Las publicaciones (Brown, W., Malveau, R., McCormick, H. & Mowbray, T., 1998) y (Brown, W., McCormick, H., Scott, W. & Thomas, S., 2000), describen los anti patrones como la contrapartida natural al estudio de los patrones de diseño, dado que la identificación formal de errores frecuentes permite reajustar los elementos involucrados hacia una mejor solución.

La tabla 5 refiere algunos anti patrones que se deben reconocer desde las primeras etapas del proceso formativo de un ingeniero para evitar hábitos inapropiados de diseño de software, diseño orientado a objetos y programación.

Tabla 5: Algunos anti-patrones de diseño tratables en lógica

Nombre del anti-patrón	Significado
Clase gorda	Dotar a una clase con demasiados atributos y/o métodos, haciéndola responsable de gran parte de la lógica del negocio.
Entrada chapuza	No controlar el manejo de entradas inválidas.
Fábrica de combustible	Generar diseños innecesariamente complejos.
Gran bola de lodo	Construir un sistema sin estructura definida.
Acoplamiento secuencial	Construir una clase condicionada a que sus métodos se invoquen en un orden determinado.
Grano base	Heredar de una clase utilidad en lugar de delegar en ella.
Problema del yoyó	Construir estructuras, digamos de herencia, difíciles de comprender por su excesiva fragmentación.
Nomenclatura heroica	Identificar los artefactos de un programa (clases, propiedades, métodos, objetos, interfaces, entre otros) con nombres que aparentan estandarización con la ingeniería de software, pero que en realidad ocultan una implementación anárquica.
Complejidad no indispensable	Dotar de complejidad innecesaria a una solución.
Código ravioli	Construir sistemas con multitud de objetos débilmente conectados.
Número mágico	Incluir en los métodos cifras concretas sin explicación aparente.
Ocultación de errores	Capturar un error antes de que se muestre al usuario, reemplazándolo por un mensaje sin importancia o ningún mensaje en absoluto.
Programación de copiar y pegar	Generar programas copiando y modificando código existente, en lugar de crear soluciones genéricas.
Programación por permutación	Tratar de aproximarse a una solución modificando el código una y otra vez para ver en definitiva funciona.
Reinvención de la rueda	Buscar soluciones desde cero sin tener en cuenta otras que puedan existir para afrontar el mismo problema.
Reinventar la rueda cuadrada	Crear una solución pobre cuando ya existe la adecuada.

El estudio de estos anti patrones puede abrir el panorama para evitar, en cursos de lógica de programación, el planteamiento inapropiado de requerimientos, el diseño de clases con alto acoplamiento y baja cohesión, la asignación inapropiada de responsabilidades y un seudocódigo inextricable. Requerimientos mal definidos pueden generar los anti-patrones *Gran bola de lodo, Complejidad no indispensable y Reinventar la rueda cuadrada*; la baja cohesión conlleva una *Clase gorda*; responsabilidades mal asignadas conducen a Fábrica de combustible o a Gran bola de lodo; los Números mágicos, la Complejidad no indispensable o la Programación por permutación generan un seudocódigo confuso. Como se observa un anti-patrón puede ser inmanente a varias fases de la solución.

4. CONCLUSIONES

El tránsito de la enseñanza de la programación ha pasado por paradigmas basados en el proceso como la estructurada, pasa por un paradigma enfocado al diseño como el orientado a objetos y tiende a otras aproximaciones basadas en arquitecturas web como la programación orientada a recursos o servicios. Por ello, los cursos de introducción a las ciencias de la computación impartidos en los primeros semestres de programas profesionales en ingeniería de sistemas y afines deben autoevaluarse de manera continua para responder a los avances del área específica que le atañe, en consonancia con las tendencias pedagógicas contemporáneas y las exigencias del sector productivo del software.

Un curso renovado de lógica de programación debe introducir, en particular, los fundamentos del paradigma orientado a objetos esencial para la apropiación significativa de otros temas relacionados con bases de datos, interfaces humano-computador, computación gráfica, entretenimiento, ingeniería de software e inteligencia artificial. Los conceptos de clase y objeto, método, visibilidad o alcance, sentencias de control, sobrecarga, clases de utilidad, clases de uso común, herencia y polimorfismo, posibilitan el tratamiento de los patrones GRASP y los anti-patrones de diseño. De esta manera, los conceptos importantes se imparten temprana y frecuentemente, en consonancia con el patrón de diseño educativo "Early Bird" (Bergin, J., 2005), utilizado en diferentes áreas disciplinares para capturar y comunicar conocimiento experto.

BIBLIOGRAFÍA

- Botero R., Castro C. & Parra E. (2008). ***Método Integrado de Programación Secuencial y Programación Orientada a Objetos para el Análisis, Diseño y Elaboración de Algoritmos – MIPSOO***. Proyecto de investigación, Facultad de Informática, CODEI, Tecnológico de Antioquia. Medellín.

- Botero R., Castro C. & Taborda, G. (2009). ***Sistema para el Modelamiento por Objetos – SISMOO.*** Proyecto de investigación, Facultad de Informática, CODEI, Tecnológico de Antioquia. Medellín.
- ***Alianza Futuro Digital Medellín.*** Recuperado el 21 de agosto de 2010 de http://www.futurodigital.org/
- Gamma E., Helm R., Johnson R. & Vlissides J. (1994). ***Design Patterns: Elements of Reusable Object-Oriented Software.*** USA: Addison - Wesley.
- Bennett S., McRobb S. & Farmer R. (2007). ***Análisis y Diseño Orientado a Objetos de Sistemas Usando UML.*** (3 ed.). España: McGraw-Hill / Interamericana.
- Larman, C. (1999). ***UML y patrones. Introducción al Análisis y Diseño Orientado a Objetos.*** México: Prentice Hall.
- Brown, W., Malveau, R., McCormick, H. & Mowbray, T. (1998). ***AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis.*** USA: Wiley and Sons.
- Brown, W., McCormick, H., Scott, W. & Thomas, S. (2000). ***AntiPatterns in Project Management.*** USA: Wiley Computer Publishing.
- Bergin, J. (2005). ***Fourteen Pedagogical Patterns.*** Recuperado el 18 de Agosto de 2010 de

  http://csis.pace.edu/~bergin/PedPat1.3.html