XV JENUI. Barcelona, 8-10 de julio de 2009
ISBN: 978-84-692-2758-9

Utilización de patrones de diseño en la asignatura de Procesadores de Lenguaje

Francisco Ortín, Daniel Zapico, José Manuel Redondo
Departamento de Informática, Universidad de Oviedo
C/ Calvo Sotelo s/n, 33007, Oviedo
{ortin,redondojose}@uniovi.es

Resumen

La asignatura de Procesadores de Lenguaje comprende el aprendizaje de una teoría formal y el desarrollo práctico de un procesador de lenguaje. La fase de análisis semántico es comúnmente descrita mediante el uso de formalismos tales como gramáticas atribuidas. En este artículo presentamos cómo hemos utilizado patrones de diseño para la enseñanza teórica de sistemas de tipos y el desarrollo práctico del analizador semántico de un compilador. Nuestro enfoque facilita la comprensión de conceptos teóricos gracias a que la notación utilizada (UML) es conocida por los estudiantes. Adicionalmente, los diseños son aplicados en el desarrollo práctico de un procesador de lenguaje, reforzando competencias propias de Ingeniería del Software.

1. Introducción

La construcción de un compilador se fundamenta en una teoría bien definida y ayuda a obtener competencias propias de otras disciplinas relacionadas como la programación, ingeniería del software, arquitectura y estructura de los computadores y sistemas operativos. Como se ha estudiado en otros trabajos [7, 9] el aprendizaje de técnicas de compilación se ve provechosamente incrementado al realizar un ejemplo real de construcción de un compilador. Las técnicas utilizadas propias de ingeniería del software pueden ayudar a los estudiantes a comprender mejor los conceptos que vertebran el proceso de compilación.

La asignatura bajo estudio es Procesadores de Lenguaje de la Escuela Politécnica Superior de Ingeniería de Gijón de la Universidad de Oviedo. Es impartida en cuarto curso del título de Ingeniero en Informática. La asignatura se divide en dos parciales, siguiendo la estructura clásica de fases de un compilador descrita por Alfred Aho [1]. En el primer parcial se ofrece una introducción seguida de las fases de análisis: léxico, sintáctico y semántico. El segundo parcial abarca la tabla de símbolos, organización de la memoria en tiempo de ejecución, generación de código e implementación de intérpretes.

Los requisitos de la asignatura son que los estudiantes tengan experiencia en la utilización de lenguajes de programación orientados a objetos (Java o C++), tecnologías orientadas a objetos, diagramación en lenguaje unificado de modelado (UML), estructuras de datos y diseño orientado a objetos. La asignatura se imparte anualmente, con dos horas semanales de teoría y 2 horas quincenales de prácticas. Las prácticas incluyen tareas quincenales dirigidas a la construcción de un compilador como proyecto final.

En este artículo nos centramos en los sistemas de tipos dentro del análisis semántico de un compilador. El diseño y verificación de sistemas de tipos se suele realizar mediante numerosos formalismos, pero éstos no se emplean para la implementación de compiladores comerciales [13]. Un primer motivo es que los métodos formales se centran más en probar propiedades de los sistemas de tipos que en la construcción de un analizador semántico. Otra causa es que el código generado por esas herramientas es ineficiente y difícil de entender y depurar, ya que procede de la traducción directa de especificaciones matemáticas.

Inicialmente introducimos conceptos tales como expresión de tipo, promoción de tipo, equivalencia de tipos, polimorfismo y sobrecarga. Posteriormente se presenta el diseño de cada concepto utilizando patrones de diseño en UML. Los diseños son implementados en distintos lenguajes de programación orientados a objetos –Java y C++ son los lenguajes conocidos por los alumnos. Dependiendo de su centro de procedencia, el alumno se sentirá más cómodo

en un lenguaje que en otro. Para los ejemplos de este artículo hemos utilizado C++.

2. Expresiones de tipo

La comprobación de tipos es la parte del análisis que detecta inconsistencias y anomalías semánticas, garantizando que las entidades se corresponden con su declaración, estableciendo este análisis conforme a un sistema de tipos dado; el algoritmo que realiza esta comprobación se denomina comprobador de tipos (*typechecker*) [3]. Un procesador de lenguaje implementa la comprobación de tipos normalmente en la fase de análisis semántico [1]. Los beneficios de los comprobadores de tipos son detección de errores, abstracción, rendimiento, seguridad y documentación [12].

En la implementación de un comprobador de tipos es necesario representar los tipos del lenguaje internamente. Una expresión de tipo es una representación interna del tipo de una construcción sintáctica [1]. Por lo tanto, cualquier procesador del lenguaje debe modelar las expresiones de tipo del lenguaje para realizar la comprobación de tipos, además del resto de tareas de traducción.

Las expresiones de tipo se basan en la definición constructiva de tipos: un tipo es o bien un conjunto de tipos básicos (primitivos) o un constructor de tipos, compuesto de otros tipos. Un tipo básico es un tipo atómico: su estructura interna no puede ser modificada por el programador (*integer*, *float*, o *boolean*). Un tipo construido es creado por el programador aplicando constructores de tipos (*record*, *array*, *set*, *class*…) a otros tipos, ya sean estos básicos o construidos.

2.1. Representación de expresiones de tipo

La representación de las expresiones de tipo puede variar dependiendo de las características del lenguaje a procesar y del lenguaje usado para implementar el compilador. Por ejemplo, la representación de las expresiones de tipo de un lenguaje que sólo tenga tipos básicos, sería tan fácil como usar variables, enteras, carácter o enumeradas. Pero esta representación no sería válida si el lenguaje a procesar admite constructores de tipo. Por ejemplo, el constructor de tipo "*puntero de tipo T*" podría crear un número infinito de expresiones de tipo.

Cuando hay constructores de tipo se necesita un esquema para modelar infinitas expresiones de tipo. Una solución factible podría ser el uso de cadenas de caracteres para representar expresiones de tipo por medio de una especie de lenguaje de expresiones de tipo (un ejemplo de este enfoque es detallado en [1]). La mayor desventaja de esta solución es el tener que procesar la cadena para extraer los distintos tipos de un tipo construido, a costa de un uso intensivo de los recursos de computación.

Una solución más adecuada sería el utilizar estructuras de datos recursivas por medio de objetos. Estas estructuras son más fáciles de manejar que las cadenas de caracteres y requieren menos recursos computacionales a la hora de procesar una expresión de tipo compuesta. Hay operaciones comunes en la mayoría de las expresiones de tipo, y otras que son específicas de cada expresión de tipo. Mediante polimorfismo, tanto las rutinas generales como las específicas pueden ser asignadas a cada expresión de tipo de un modo mantenible.

2.2. El patrón de diseño *Composite*

El problema de representar recursivamente estructuras de datos de manera jerárquica aparece en muchos contextos. El patrón orientado a objetos *Composite* [6] se usa para modelar y resolver este tipo de problemas. Este patrón modela tanto estructuras simples como compuestas recursivamente, ofreciendo un modo uniforme para manejar esas estructuras heterogéneas (Figura 1). Los elementos del patrón de diseño *Composite* son los siguientes:

1. *Component* (`TypeExpression`). Una clase, normalmente abstracta, que declara la interfaz de todos los elementos, simples y compuestos, definiendo su procesamiento de manera uniforme. Cada método puede ofrecer una implementación por defecto (`asterisk`, `squareBrackets`, `brackets` y `dot`) o ser declarada abstracta para ser implementada por las subclases *Component* (`getBytes` y `typeExpression`).
2. *Leaf* (`Character`, `Integer`, `Void`, `Boolean` y `Error`). Estos elementos representan nodos hoja en la estructura jerárquica. Los métodos

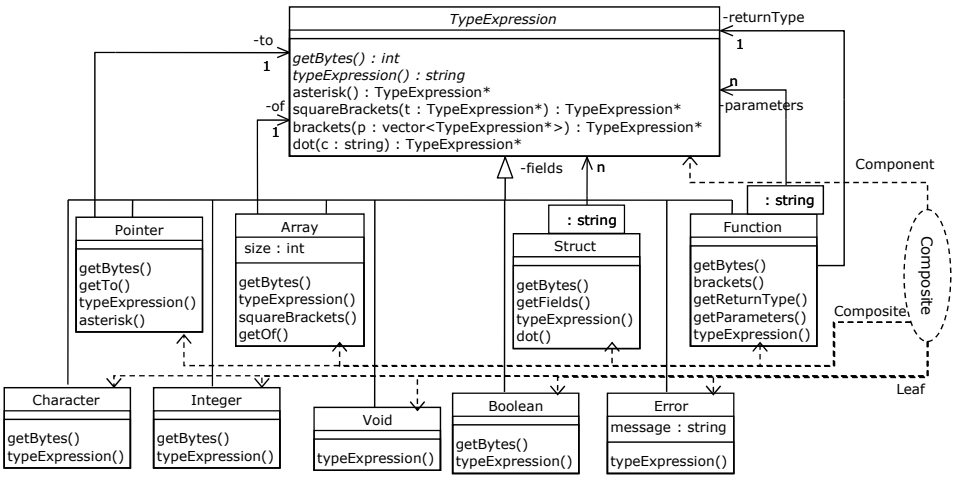

Figura 1. Diagrama de clases que modela parte de las expresiones de tipo del lenguaje C.

de la clase hoja definen operaciones concretas para ese nodo específico.

3. *Composite* (`Pointer`, `Array`, `Struct`, y `Function`). Este elemento modela estructuras construidas por composición de otras, manejando referencias a los tipos a los que se ha aplicado el constructor de tipo. El *Composite* implementa sus métodos teniendo en cuenta los tipos referenciados, obteniendo a veces resultados parciales de las operaciones implementadas por cada uno de los nodos hijo.

2.3. Implementación de expresiones de tipo

Cuando se aplica el patrón de diseño *Composite* para implementar expresiones de tipo, cada tipo básico será un nodo hoja, y cada tipo compuesto un *Composite*. La clase `TypeExpression`, ofrecerá operaciones de la fase de análisis semántico; esto es, operaciones comunes a todos los tipos (equivalencia, promoción, inferencia o unificación). `TypeExpression` también ofrecerá operaciones para la fase de generación de código (tamaño en bytes o representación a bajo nivel). Manteniendo estos métodos en la clase base de la jerarquía, se logra un tratamiento uniforme de todos los tipos de acuerdo al interfaz de la clase *Component*, a pesar de su estructura interna.

Cuando varios tipos tengan el mismo comportamiento para una operación específica, éste se implementa como un método en la clase base `TypeExpression`. Entonces, si fuese necesario, cada subclase podría sobrescribir la operación general definida en la clase `TypeExpression`; de otra manera el comportamiento por defecto será el heredado. Además, cada subclase puede definir métodos específicos que se correspondan con la expresión de tipo que la clase representa, sin declararlos en la clase base. Estos métodos específicos definen mensajes de una única expresión de tipo y no son aplicables al resto de tipos.

La Figura 1 muestra el diseño preliminar de expresiones de tipo del lenguaje de programación C, siguiendo el patrón de diseño *Composite*. El rol *Component* es asignado a la clase abstracta `TypeExpression`, definiendo el comportamiento general de todas las expresiones de tipo. Alguno de esos mensajes aceptados por la clase `TypeExpression` tiene una implementación por defecto. Una muestra de esos mensajes es la siguiente:

1. `asterisk`, `squareBrackets`, `brackets` y `dot`. Estos métodos tienen dos responsabilidades. La primera es inferir el tipo resultante cuando un operador es aplicado a una expresión. Muchos métodos reciben otras expresiones de tipo como parámetros. Por ejemplo, en una invocación a función (`brackets`) se requiere el tipo de cada argumento para inferir el tipo de retorno. La

segunda responsabilidad es la comprobación de tipos, que necesita saber si un operador puede ser aplicado a una expresión. Si la operación no tiene sentido para una expresión de tipo específica (se ha cometido un error de tipo) se devuelve la expresión de tipo `Error`.
La implementación por defecto de todos esos métodos es retornar un objeto `Error`. Si no se redefine, el comportamiento por omisión es indicar que la operación no está definida por el tipo correspondiente. Si el operador se puede aplicar a un tipo, se deberá redefinir éste. Por ejemplo, el tipo modelado por la clase `Pointer` implementa el método `asterisk` porque el operador "*" es aceptado por los punteros. Este método devuelve una expresión de tipo que se corresponde con el tipo al que referencia el puntero.

2. `typeExpression`. Devuelve una cadena de caracteres que representa la expresión del tipo modelado, usando como notación la definida por Alfred Aho [1] (en la próxima sección se da una explicación más detallada). Este método facilita la depuración a la vez que reduce el consumo de memoria. En la Figura 2 puede verse un ejemplo de una implementación en C++ de este método para la clase `Function`. En este ejemplo, una expresión de tipo de una función que no devuelve nada y que recibe dos argumentos, un puntero a entero y un real, será traducida a la cadena "*(Pointer(Integer), Float)->Void*".

```
string Function::typeExpression() const {
   ostringstream o;
   if (parameters.size()) {
      o<<"(";
      for (unsigned i=0;i< parameters.size();i++)
         o<< parameters [i]->typeExpression()<<
            (i<parameters.size()-1 ? ',' : ')');
   }
   o<<"->"<<returnType->typeExpression();
   return o.str();
}
```

Figura 2. Ejemplo de implementación del método *typeExpression*.

3. `getBytes`. Devuelve el número de bytes necesario para almacenar una variable según su tipo. Este mensaje es un ejemplo de cómo los tipos del lenguaje pueden también representar responsabilidades de la fase de generación de código.

Los tipos compuestos tienen asociaciones con los tipos base: un puntero requiere el tipo del objeto al que apunta (`to`); un *array* requiere el tipo de los objetos que colecciona (`of`); una estructura requiere una colección de sus campos (`fields`), cualificada por el nombre de cada campo (`string`); una función requiere una asociación con el valor que devuelve (`returnType`), así como con una colección de los parámetros que admite, cualificados por nombre (`parameters`).

Como todas las asociaciones apuntan a la clase raíz de la jerarquía, las clases *Composite* pueden estar compuestas de cualquier tipo (incluso de ellos mismos) a través de polimorfismo.

Las expresiones de tipo se construyen mediante un proceso dirigido por sintaxis, por medio de los constructores de tipo. Los tipos básicos son los primeros en ser creados, y son usados más tarde para construir los tipos compuestos. Este diseño facilita la construcción de tipos usando herramientas de análisis dirigidas por sintaxis tales como *lex/yacc*, *Antlr* y *JavaCC*.

2.4. Ejemplo de utilización

Una vez explicado el concepto teórico y su diseño describimos un escenario de utilización. Los aspectos léxicos y sintácticos son implementados con *lex* y *yacc* respectivamente. El resto de las partes del procesador se implementan en ISO/ANSI C++. Cuando una variable es definida (líneas 1 a 9 en Figura 3), se asocia a ésta su expresión de tipo (un puntero a la clase `TypeExpression`) en una tabla de símbolos. Los tipos son creados invocando el constructor de tipos apropiado desencadenando un proceso recursivo mientras *yacc* realiza las reducciones.

Aunque este es un ejemplo de un compilador de una pasada, este diseño también es válido para compiladores de múltiples pasadas. En dicho caso las expresiones de tipo serían construidas mediante llamadas a los métodos `visit` del patrón de diseño *Visitor* [6, 14].

```
1:  int vector[10];
2:  int *pointer;
3:  int **doublePointer;
4:  char **v[10];
5:  char *w[10];
6:  struct Date { int day, month, year; };
7:  struct Date date;
8:  int *f(int,char*);
9:  void p(int*);
10: int main() {
11:   date;                 Struct((day x Integer)x
                                  (month x int)x(year x Integer))
12:   v;                    Array(10,Pointer(Pointer(Character)))
13:   vector[*pointer];     Integer
14:   **v[**doublePointer]; Character
15:   w[*f(3,w[1])];        Pointer(Character)
16:   p(f(date.day,w[2]));
17: }
```

Figura 3. Traza de inferencia de tipos en un programa de ejemplo.

A cada expresión del lenguaje se debe asignar una expresión de tipo (un puntero a `TypeExpression`). En cada reducción realizada por *yacc*, el tipo de la subexpresión es inferido aplicando el operador adecuado. Esta inferencia se realiza pasando el mensaje apropiado al tipo de la subexpresión. La Figura 3 muestra una traza de todas las expresiones de tipos inferidas en cada sentencia del programa principal. El código fuente es mostrado en la parte izquierda, y la cadena devuelta al pasar el mensaje `typeExpression` a cada tipo inferido aparece en la parte derecha. Por ejemplo, el tipo `Pointer` se obtiene a partir de la primera subexpresión en la línea 13 (variable pointer); `asterisk` es invocado en la siguiente reducción infiriendo el tipo apuntado (`Integer`); el siguiente paso es obtener el tipo de `vector` de la tabla de símbolos. Entonces, se pasa el mensaje `squareBrackets` a `Array` (el tipo de vector) usando la expresión de tipo `Integer` (previamente inferida) como argumento; finalmente, el tipo inferido es `Integer`.

Los procesos de inferencia y comprobación de tipos se ofrecen de manera uniforme (no dependen de los tipos particulares inferidos), obteniendo comportamiento heterogéneo por medio del polimorfismo. La Figura 4 ilustra esta característica de uniformidad usando *yacc* para implementar un compilador de una pasada. Los mensajes `asterisk`, `squareBrackets` y `dot` representan esta funcionalidad heterogénea uniforme.

```
exp: '(' exp ')'     { $$=$2; }
   | '*' exp         { $$=$2->asterisk(); }
   | exp '[' exp ']' { $$=$1->squareBrackets($3); }
   | exp '.' ID      { $$=$1->dot($3); }
   | INT_LITERAL     { $$=new Integer($1); }
   | CHAR_LITERAL    { $$=new Character($2); }
   | ID              { $$=symbolTable.find($1); }
```

Figura 4. Construcción e inferencia uniforme de tipos con yacc.

3. Otros patrones de diseño empleados

El resto de explicaciones para llevar a cabo un comprobador de tipos se asocian a patrones de diseño. Lo siguiente es una enumeración de los patrones empleados para cada uno de los conceptos. El detalle de cómo utilizamos éstos se puede obtener de [10]:

- La equivalencia de tipos se obtiene, de nuevo, mediante la utilización del *Composite*.
- El acceso a todos los tipos se diseña mediante un el patrón *Flyweight*.
- En la construcción de tipos compuestos se utiliza el *Builder* y *Singleton* para los tipos simples.
- La coerción de tipos y el polimorfismo paramétrico vuelven a hacer uso de patrón de diseño *Composite*.

4. Comparación con enfoque tradicional

En una asignatura típica de procesadores de lenguaje, el principal formalismo utilizado para describir comprobadores de tipos son las gramáticas atribuidas [1, 5]. Posteriormente a la

fase de análisis sintáctico se presentan las gramáticas atribuidas. Una gramática atribuida es un sistema formal que permite al usuario añadir atributos a una gramática libre de contexto, así como las relaciones entre éstos [8].

Antes de usar las gramáticas atribuidas, el profesor ha de explicar algunos temas tales como notación, atributos heredados y sintetizados, gramáticas atribuidas bien definidas (no circulares), gramáticas S-atribuidas y L-atribuidas, y traducción a lenguajes imperativos [11]. La explicación de esos conceptos no es una tarea trivial porque las gramáticas atribuidas son lenguajes declarativos, y los estudiantes están acostumbrados a programar en lenguajes imperativos.

El problema del enfoque tradicional es doble: primero, se deben realizar muchos ejercicios porque los estudiantes no están acostumbrados a programar en lenguajes declarativos; segundo, no hay herramientas disponibles para traducir una gramática atribuida, con todas sus características, a lenguajes imperativos orientados a objetos como Java o C++ [13].

Las gramáticas atribuidas son un mecanismo de alto nivel para describir características de manera dirigida por la sintaxis. No obstante, no ofrecen una traducción directa a implementaciones de comprobadores de tipos, lo que supone un importante inconveniente para las competencias prácticas de desarrollo de un compilador. Como ejemplo, si se va a desarrollar un compilador de una pasada usando *yacc*, una gramática no circular debería ser traducida a su equivalente gramática S-atribuida: *yacc* es un generador de analizadores sintácticos LALR dirigido por sintaxis e imperativo, que sólo soporta atributos sintetizados. Posteriormente, las reglas declarativas de las gramáticas atribuidas deben ser traducidas a rutinas imperativas en *yacc*.

4.1. El enfoque de los patrones de diseño

El enfoque presentado en este artículo utiliza patrones de diseño expresados en UML para especificar sistemas de tipos en lugar del formalismo dirigido por sintaxis de las gramáticas atribuidas. Ambos son notaciones de alto nivel, pero, como UML no está dirigido por sintaxis, se hace más fácil separar la sintaxis del lenguaje de las cuestiones relativas a la comprobación de tipos. Por lo tanto, los profesores pueden separar sus explicaciones de comprobación de tipos de las características sintácticas del lenguaje. Así, la comprobación de tipos puede reutilizase tanto para compiladores de una pasada (Figura 4) como de múltiples pasadas (por medio del patrón de diseño *Visitor* [6, 14]).

En el alto nivel de abstracción, los patrones de diseño facilitan la comprensión, abstracción, encapsulación, reutilización, y modularización de los algoritmos de comprobación de tipos, siendo directamente aplicables a la construcción práctica de compiladores. En el bajo nivel de abstracción, las implementaciones de los métodos clarifican la inferencia y comprobación de tipos, y las responsabilidades de la fase de análisis semántico, y muestra cómo están interconectados los diferentes elementos de los sistemas de tipos.

5. Evaluación

La efectividad de nuestra aproximación ha sido evaluada analizando la retroalimentación de los estudiantes, su rendimiento y las características del compilador desarrollado como proyecto final.

5.1. Retroalimentación de los alumnos

En una encuesta anónima, 33 estudiantes completaron el cuestionario de diez puntos (Tabla 1) usando una escala de cinco puntos (donde 1 significaba muy en desacuerdo, y 5 muy de acuerdo). Con la encuesta se pretendió evaluar si, en opinión de los estudiantes, los diseños presentados en este artículo facilitan la comprensión de los conceptos de los sistemas de tipos y la implementación de un compilador, y si ello ha fortalecido sus habilidades en ingeniería del software. La encuesta también preguntaba si pensaban que el desarrollo práctico de un compilador aplicando técnicas orientadas a objetos fortalecería sus competencias propias de otras asignaturas.

1	Los diseños presentados en prácticas me ayudan a comprender la teoría	4,45
2	Abordar la complejidad de implementar un compilador con diseños orientados a objetos facilita su implementación en la práctica	4,61
3	Los diseños en UML agilizan la implementación de las fases de un compilador	4,33
4	La práctica de la asignatura refuerza las capacidades y objetivos de otras asignaturas relacionadas	4,03
5	Los patrones de diseño utilizados me ayudan a producir un código más mantenible	4,67
6	Esta práctica me está sirviendo para asentar mis conocimientos previos de tecnologías orientadas a objetos	4,42
7	El desarrollo de esta práctica me está ayudando en mejorar mis destrezas de programación	4,12
8	En general, considero que el analizar y mostrar los diseños de las fases de un compilador reduce el tiempo de desarrollo de cada una de ellas	4,27
9	Es preferible explicar menos conceptos en teoría, para profundizar más en la construcción de las fases de un compilador	3,73
10	Considero más importante construir un pequeño procesador de lenguaje con todas sus fases que estudiar conceptos teóricos y notaciones de descripción de los mismos.	4,06
	Promedio	4,27

Tabla 1. Resultado de la encuesta.
Muy en desacuerdo (1), en desacuerdo (2), indiferente (3), de acuerdo (4), muy de acuerdo (5).

Con alguna variación, la mayoría de los estudiantes estuvieron de acuerdo en que los objetivos del enfoque presentado se lograron. La evaluación media (4,27) mostró que la efectividad de este trabajo fue un éxito notable: la evaluación media de casi todos los temas rondó el 4. Los estudiantes expresaron que los patrones de diseño ayudan a implementar un compilador, facilita la compresión de conceptos teóricos, y proporciona buenas prácticas para mejorar sus competencias propias de ingeniería del software.

La única pregunta que tuvo una evaluación por debajo de cuatro (casi de acuerdo) fue la cuestión 9. Probablemente este resultado provenga de las opiniones de algunos alumnos de que el nivel abstracción de diseño es suficiente para implementar un compilador.

5.2. Rendimiento de los alumnos

Originalmente el curso se impartía utilizando gramáticas atribuidas para describir los sistemas de tipos [1]. En el 2003, se introdujeron los patrones de diseño descritos en este artículo junto con gramáticas atribuidas. Ese año los estudiantes desarrollaron un compilador de una pasada usando yacc. En el 2004, las gramáticas atribuidas fueron suprimidas del curso, desarrollando un compilador de varias pasadas usando el patrón de diseño *Visitor* [6, 14]. La evolución del rendimiento de los alumnos en esos años puede verse en la Tabla 2.

El rendimiento de los alumnos se vio mejorado. En el 2002 el porcentaje de los alumnos que aprobaba la asignatura era del 70,65%. Este porcentaje (respecto a presentados, teniendo en cuenta a los alumnos repetidores) se ha incrementado gradualmente hasta el 98,95% obtenido el último año. Se ha usado una escala numérica de cuatro puntos: 4 (Matrícula de honor), 3 (Sobresaliente), 2 (Notable), 1 (Suficiente), 0 (Suspenso). La media de los estudiantes también ha pasado del 1,8 al 2. Cabe destacar que los estudiantes de 2003 obtuvieron las peores notas; ese fue el año en que se enseñaron ambas técnicas (gramáticas atribuidas y patrones de diseño).

Hemos medido el número de horas estimadas que un estudiante tarda en desarrollar el proyecto. Para evitar plagios, incrementamos gradualmente las características del compilador en cada convocatoria. De esta forma, estimamos

Año	Con gramáticas atribuidas	Con patrones de diseño	Porcentaje de Aprobados	Promedio de calificación	Estimación de horas empleadas
2002	✓		70,65%	1,82	119,22
2003	✓	✓	87,62%	1,78	117,14
2004		✓	92,65%	2,07	114,6
2005		✓	98,55%	2,05	110

Tabla 2. Evolución del rendimiento de los estudiantes.

que presentar el compilador en junio implica 90 horas, en septiembre 120 horas y 150 horas para febrero. Como muestra la Tabla 2, las horas usadas por los estudiantes en su proyecto final ha ido reduciéndose gradualmente (de 119 a 110).

Estos resultados indicarían que, en el contexto de la asignatura aquí descrita, los patrones de diseño suponen un mecanismo adecuado para enseñar sistemas de tipos. La mejora en el porcentaje de estudiantes aprobados puede interpretarse como una mejor compresión de los conceptos teóricos. Al mismo tiempo, el enfoque práctico presentado en este artículo facilita el desarrollo del proyecto final, reduciendo el número de horas empleadas por los estudiantes. No se detectaron casos de plagio entre estudiantes, y ningún otro factor relevante se introdujo en el curso.

5.3. Beneficios cualitativos

Gracias a que se han usado diseños UML para enseñar tanto la teoría como prácticas de sistemas de tipos, se logrado aprovechar mejor el tiempo. El tiempo ahorrado al no tener que explicar gramáticas atribuidas ha sido empleado en mejorar la calidad del compilador a desarrollar. De hecho, en los años 2005 y 2006 se han incluido características nuevas tales como: promoción de tipos, equivalencia de tipos y polimorfismo de tipos básico. Además, se pasó de una arquitectura de una única pasada a una de varias pasadas [2, 4], fortaleciendo el ejercicio de ingeniería del software propio de desarrollar un procesador del lenguaje real.

6. Conclusión

El trabajo llevado a cabo ha puesto de manifiesto que la utilización de patrones de diseño en la enseñanza de Procesadores de Lenguaje de cuarto curso de Ingeniería Informática ha obtenido unos resultados positivos. Los diseños en UML son más cercanos al alumno que los formalismos comúnmente empleados para describir sistemas de tipos, facilitando así la comprensión de los conceptos teóricos. Además, los estudiantes implementan un comprobador de tipos real, fortaleciendo sus competencias en otras disciplinas como ingeniería del software y programación. Este enfoque representa una combinación satisfactoria de teoría y práctica, habiendo obtenido un incremento gradual del rendimiento y una retroalimentación muy positiva.

Referencias

[1] Aho, A.H., Ullman, J.D., Sethi, R. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1985.

[2] Appel, A.W. *Modern Compiler Implementation in Java*. Cambridge University Press, 1997.

[3] Cardelli, L. *The Computer Science and Engineering Handbook*. CRC Press, 2004.

[4] Galles, D. *Modern Compiler Design*. Addison Wesley, 2005.

[5] Fischer, C.N., LeBlanc, R.J. Crafting a Compiler. Benjamin Cummings, 1988.

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

[7] Griswold, W.G. *Teaching Software Engineering in a Compiler Project Course.* Journal on Educational Resources in Computing 2(4), 2002.

[8] Knuth, D.E. *Semantics of context-free languages.* Mathematical Systems Theory 2 (2), 1968.

[9] Liu, H. *Software engineering practice in an undergraduate compiler course*. IEEE Transactions on Education 36(1), 1993.

[10] Ortin, F., Zapico, D., Cueva, J.M. *Design Patterns for Teaching Type Checking in a Compiler Construction Course*. IEEE Transactions on Education 50 (3), 2007.

[11] Paakki, J. *Attribute grammar paradigms –a high-level methodology in language implementation*. ACM Computing Surveys 27 (2), 1995.

[12] Pierce, B.C. *Types and Programming Languages*. MIT Press, 2002.

[13] Scott, M.L. *Programming Language Pragmatics*. Morgan Kaufmann Publishers, 2000.

[14] Watt, D., Brown, D. *Programming Language Processors in Java: Compilers and Interpreters*. Prentice Hall, 2000.