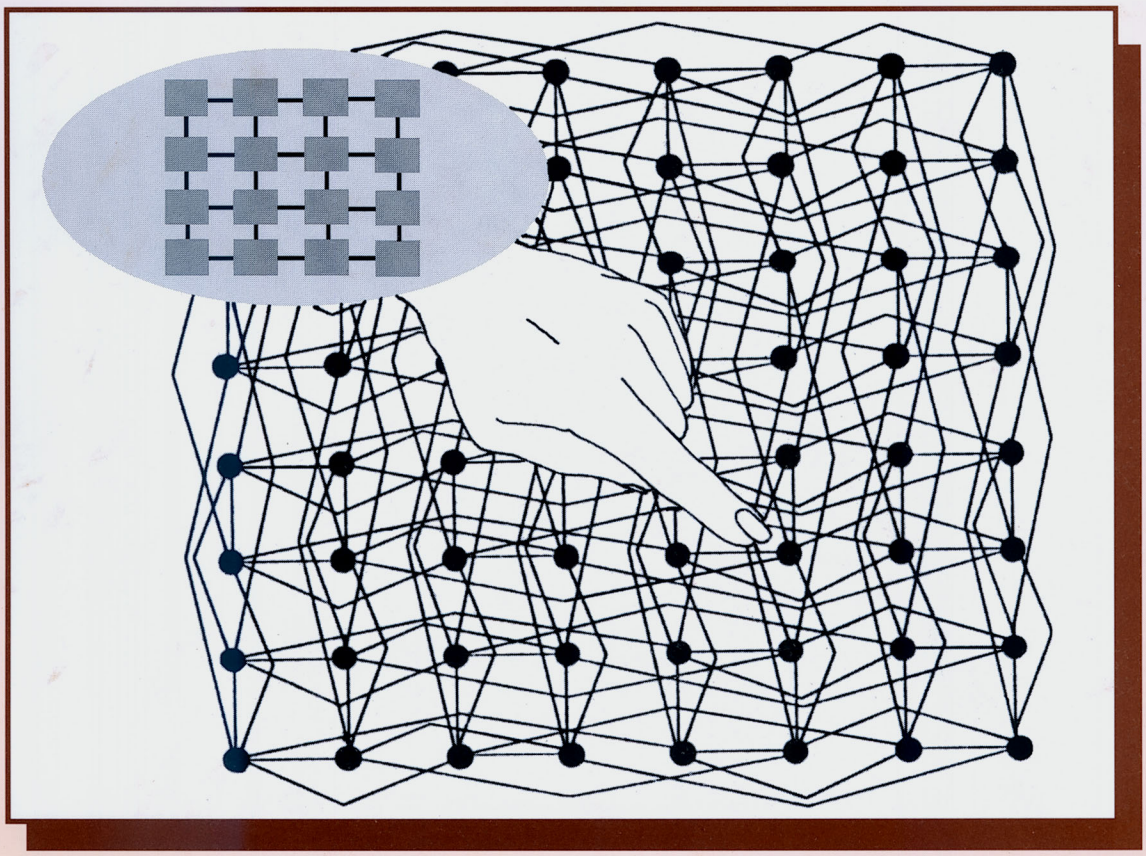
Ingeniería

Revista de la Universidad de Costa Rica
Julio/Diciembre 1996 VOLUMEN 6 Nº 2



INGENIERIA

Revista Semestral de la Universidad de Costa Rica
Volumen 6, Julio/Diciembre 1996 Número 2

CORRESPONDENCIA Y SUSCRIPCIONES

Editorial de la Universidad de Costa Rica
Apartado Postal 75
2060 Ciudad Universitaria Rodrigo Facio
San José, Costa Rica

CANJES

Universidad de Costa Rica
Sistema de Bibliotecas, Documentación e Información
Unidad de Selección y Aquisiciones-CANJE
Ciudad Universitaria Rodrigo Facio
San José, Costa Rica

Suscripción anual:
Costa Rica: ¢ 1 000,00
Otros países: US $ 25,00

Número suelto:
Costa Rica: ¢ 750,00
Otros países: $ 15,00

Edición aprobada por la Comisión Editorial de la Universidad de Costa Rica

Ciudad Universitaria Rodrigo Facio
San José, Costa Rica.

Revisión Filológica: *Lorena Rodríguez*

Diagramación:
José R. Argüello V.

Control de Calidad:
Unidad Diseño Revistas. Oficina de Publicaciones

Impreso en la Oficina de Publicaciones
de la Universidad de Costa Rica

Revista
620.005
I-46i Ingeniería / Universidad de Costa Rica. ——
Vol. I, no. 1 (ene./jun. 1991)— . —— San José, C. R. : Editorial de la Universidad de Costa Rica, 1991— (Oficina de Publicaciones de la Universidad de Costa Rica)
v. : il

Semestral.

1. Ingeniería - Publicaciones periódicas.

CCC/BUCR—250

Ingeniería 6 (2): 19-30 1996 San José, Costa Rica

HACIA UNA ESTRATEGIA INTEGRAL DE REUTILIZACIÓN DE *SOFTWARE*.

*Alan Calderón C.**

Resumen

Desde una perspectiva integradora de algunos de los factores más relevantes para la reutilización de software, se propone dar mayor énfasis al conocimiento o experiencia acumulada, sobre el uso adecuado de componentes reutilizables. Se analizan algunos métodos de trabajo, así como la necesidad y viabilidad de entornos computacionales apropiados, basados en hipertextos, que además podrían incluir técnicas de inteligencia artificial en sus mecanismos de acceso. Se concluye planteando la necesidad de que la universidad y las organizaciones desarrolladoras de software colaboren en la experimentación sistemática tendiente a la definición de estrategias integrales de reutilización.

Summary

Taking into account some relevant issues about software reuse, gathering and organization of knowledge about how to appropriately reuse components is emphasized. Some procedures and knowledge-based hypertext systems arquitectures are discussed, as possible tools aimed at supporting knowledge gathering and organization. Our universities should look for strategic alliances with local software industry in order to empirically derive appropriate reuse strategies.

Palabras clave: reutilización de *software*, *software* orientado a objetos, desarrollo de sistemas, desarrollo de programas, ingeniería de *software*.

1. INTRODUCCIÓN.

El gran potencial de los ambientes de desarrollo de *software* orientados a objetos (ADSOOs) radica en las posibilidades de reutilización de componentes. Las promesas de este paradigma de programación dependen de que se reutilice *software* en el desarrollo de cada aplicación nueva. Sin embargo, la reutilización no puede plantearse unidimensionalmente como un asunto meramente técnico, sino que deben integrarse consideraciones metodológicas, y de organización interna La reutilización no solo consiste en el uso de piezas de *software* preelaboradas para la construcción de nuevos programas, se trata más bien de sistematizar el conocimiento generado por una organización desarrolladora de *software*, a efecto de que sus miembros puedan accesarlo y usarlo en el momento preciso. Desde esta perspectiva integradora, es pertinente preguntar:

- ¿cuál es la estructura o marco intra-organizacional adecuado para la reutilización de *software*?,
- hasta ahora se ha dado mucha importancia a la producción de componentes reutilizables de alta calidad, pero ¿qué pasa con el conocimiento acumulado por una organización para usar un conjunto dado de componentes?,
- las organizaciones generan conocimiento valioso a distintos niveles: análisis de sistemas, diseño de sistemas, programación y diseño de la interfaz humano-sistema, sin embargo, los ADSOO solo abarcan las clases, que se encuentran al nivel de la programación,
- ¿cómo se puede sistematizar y facilitar el acceso al conocimiento generado por una organización que desarrolla de *software*?,

* Profesor de la Escuela de Computación e Informática de la Universidad de Costa Rica.

- los *browsers*[1] actuales se centran en la estructura de los componentes, cabe preguntarse ¿qué importancia tiene su función para que puedan ser efectivamente reutilizados?,
- ¿cuáles son los métodos que una organización puede aplicar para generar conocimiento en un dominio de aplicación específico?.

A continuación se exploran algunas respuestas a estas preguntas.

2. REENFOCANDO EL TEMA.

2.1 El marco intra-organizacional de la reutilización.

Los aportes de Martin L. Griss al tema de la reutilización de *software* son abundantes y de gran interés[2]. Particularmente, en [Griss,95], pg. 76, utiliza el siguiente gráfico para representar los procesos que subyacen en la reutilización, así como sus interrelaciones: "La experiencia con esfuerzos de reutilización exitosos muestra que ésta es más efectiva cuando el proceso de creación de activos—*componentes de reutilización*— es administrado y realizado en forma separada de la utilización de activos." [Griss,95], pg. 76.

En consonancia con esto, otros autores plantean contextos similares. Por ejemplo Beck distingue entre "abstractors" y "elaborators": "Yo divido el mundo del desarrollo de *software* en dos partes: el *abstractor*, que crea piezas reutilizables; y el *elaborador* que adecua estas piezas para resolver las necesidades del usuario. Ultimamente, *Microsoft* ha estado promulgando una visión similar, en la cual se divide el desarrollo de *software* en dos categorías: los constructores de componentes (por ejemplo los programadores que escriben DLLs o librerías de clases en C o C++), y los constructores de soluciones (los que usan partes de alto nivel como *Parts*, *Visual Basic*, *Power Builder*, o un *framework* de aplicación en conjunto con componentes DLL de bajo nivel, para construir aplicaciones orientadas a usuarios finales).", [Beck, 94], pg. 18.

En adelante denomino como programador de herramientas (PH) a una persona o grupo de personas que **crea** *software* reutilizable y probablemente colabora en el **soporte**, y como programador de aplicaciones (PA) a una persona o grupo que crea *software* para usuarios finales, **utilizando y adecuando** componentes previamente desarrollados. Desde mi perspectiva de lo que se trata es de sistematizar el conocimiento generado en este proceso de reutilización. Tanto el PH como el PA generan conocimiento que debe estructurarse y sistematizarse. No se trata, simplemente, de reutilizar código, sino más bien conceptos de diseño y soluciones a problemas.

---

[1] Un "browser" es un *software* que permite al diseñador o programador buscar los componentes apropiados para construir algún programa específico.

[2] Algunas de sus publicaciones son: 1) ML Griss, J Favaro & P Walton. Managerial ang organizational issues—Starting and running a software reuse program, Software Reusability, Ellis Horwood, Chichester, UK, 1993, pp. 51-78. 2) ML Griss & M Wosser. Making reuse work at Hewlett-Packard, IEEE Software 12(1):105-107, 1995. 3) ML Griss. Software reuse: From library to factory, IBM System Journal 32(4):548-566,1993. 4) ML Griss & KD Wentzel. Hybrid domain-specific kits, Journal of Systems and Software, September 1995. Además de sus interesantes columnas en la revista OBJECT Magazine ("Reuse").

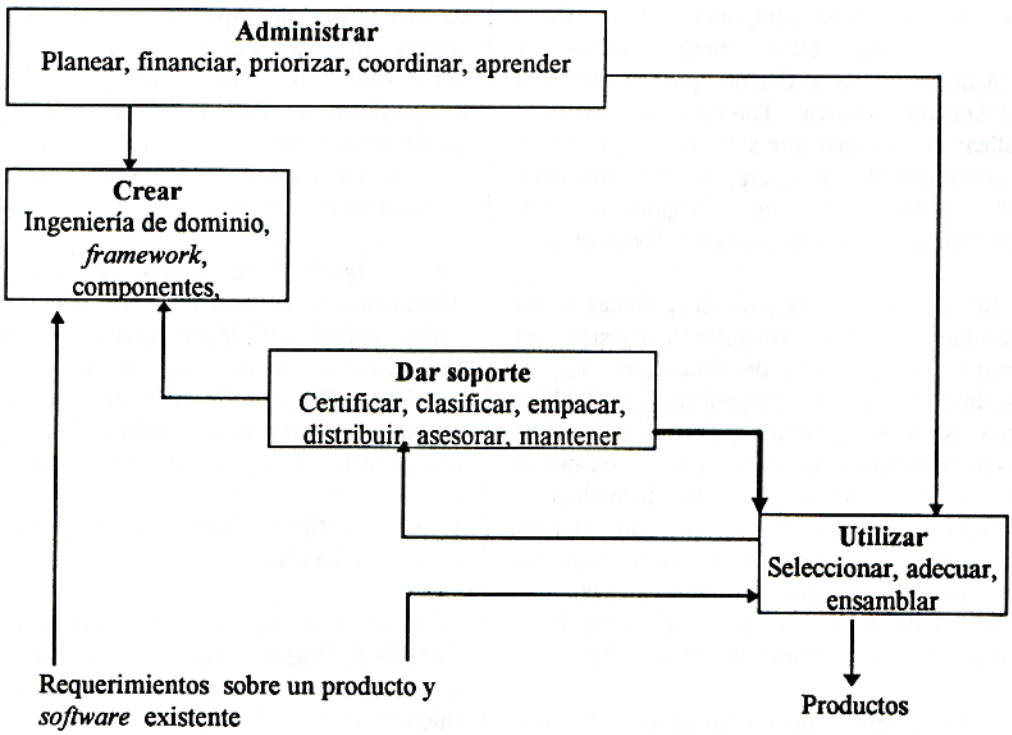


Figura N° 1. Proceso de Reutilización de *software* (tomado de [Griss,95], pg. 76).

2.2 De los componentes al conocimiento sobre los componentes.

La creación de una clase no solo da como resultado un componente que podrá ser reutilizado en el futuro. Al crear una clase, se formaliza y operacionaliza un concepto que será útil para pensar la solución de problemas futuros. La pila no es solo una pieza de *software* que, compuesta por otras, nos permite construir otros programas. La pila define una estrategia, útil en muy diversos contextos, para guardar y recuperar objetos. Al permitir guardar objetos y accesar tan solo el último insertado, la pila operacionaliza y formaliza una estrategia genérica para resolver cierto tipo de problemas de almacenamiento que se presentan en muchos contextos distintos.

Por lo tanto, reutilizar componentes de *software* implica también reutilizar conocimientos. Cada pieza de *software* concreta conocimientos prácticos generados a través del proceso de diseño y programación de aplicaciones. Detrás de cada componente de *software* hay un concepto. Pero esde la perspectiva de la reutilización, ¿qué es lo relevante de un concepto?. Perkins (1987) plantea una respuesta que debe considerarse:

"...entender cualquier pieza de conocimiento o cualquier producto del intelecto humano implica visualizarlo como un diseño, una estructura formada para cierto propósito. En particular, el entendimiento implica ser capaz de responder cuatro preguntas de diseño acerca del objeto en cuestión:

¿cuál es su propósito? (o propósitos—puede haber más de uno),

¿cuál es su estructura?,

¿cuáles son los casos modelo?,

¿cuáles son las explicaciones o argumentaciones asociadas?." [Perkins, 87], pg. 64.

Construir un nuevo componente y ponerlo a disposición de otros programadores y diseñadores no es suficiente para lograr una reutilización efectiva. También es necesario explicar el concepto que subyace: su propósito, lo relevante de su estructura, sus casos modelo o usos típicos y las argumentaciones, explicaciones y consecuencias del diseño en sí.

He diseñado esquemas para documentar clases basándome en los principios expuestos por Perkins. La intención de estos esquemas es describir el concepto que subyace a la clase, como estrategia genérica para resolver cierto tipo de problemas, así como la forma en que se usa el componente que lo formaliza y operacionaliza. En el trabajo con algunos estudiantes de programación, estos esquemas han sido útiles para aprender a usar una biblioteca de clases contenedoras[3] como la de Borland C++ 3.1 (en adelante BC++ 3.1).

2.2.1 Un ejemplo: documentación de la clase concreta CLICOLA.

La clase concreta *CLICOLA* formaliza y operacionaliza un concepto propio del dominio de las simulaciones basadas en colas. En el recuadro de la figura 2 solo aparece el documento en forma parcial para facilitar la exposición. Estos esquemas han sido incorporados a un hipertexto a fin de establecer los caminos de acceso a documentos con información complementaria.

Cada documento es un punto de entrada para otros. A través de conexiones hipertextuales, este documento de *CLICOLA* se relaciona con la documentación de *OBJECT*, *CONTAINER*, *DEQUEUE*, *SERVIDOR* y *CLIENTE*. Además, en el punto 4. de la sección **¿Cómo se usa?** hay una referencia hipertextual a la documentación de una aplicación específica (un programa de simulación de colas de impresión). Esta sección corresponde con el tópico de "casos modelo", contemplado por Perkins, que en este contexto puede verse como casos de uso. En el punto 1. de esta misma sección se hace referencia a archivos de código fuente.

En la figura 3 se ilustra el resto de la documentación de *CLICOLA*. De nuevo existen varias conexiones hipertextuales potenciales. Por ejemplo los nombres de datos-miembro (*ultEve*) y de funciones-miembro (*get*, *put*, etc) constituyen puntos de conexión con archivos de código fuente. De igual forma los identificadores de atributos globales de la clase, funciones globales y tipos. Todo esto es parte de la estructura del diseño.

En cuanto a las explicaciones asociadas al objeto *CLICOLA*, éstas se pueden presentar subsumidas entre la descripción de los atributos (o datos miembro) las funciones miembro y las relaciones públicas de la clase. Estas explicaciones deben contribuir a que el programador de aplicaciones tenga una imagen coherente del componente que le permita usarlo, y no revelar los detalles de su estructura interna. El principio de abstracción debe seguirse con cuidado al escribir el conocimiento necesario sobre los componentes.

Evidentemente, este esquema está orientado a sistematizar el conocimiento generado por un grupo de desarrolladores en relación con el uso de una clase, lo cual es útil al nivel de la programación. Lo que aquí se ha planteado es tan solo un punto de partida que podría ser útil al tratar de sistematizar conocimientos sobre reutilización de distintos tipos. Gamma y otros autores introdujeron en [Gamma,95] un concepto fundamental en el tema de la reutilización: el patrón de diseño.

[3] Clases como la pila cuyo propósito es guardar objetos, en la memoria principal, para su posterior acceso, durante la misma ejecución de un programa.

/*----------*/
/* CLICOLA.H */
/* Escuela de Computación e Informática */
/* Todos los Derechos Reservados, 1995. */
/*----------*/

class_CLASSTYPE **CliCola** : public Queue

¿Para qué sirve?

Los objetos de la clase "CliCola" modelan colas de clientes que esperan por un servicio. Estos clientes pueden ser personas que esperan un servicio en un banco, automóviles que esperan su turno en un lava-carros, o pueden ser los trabajos pendientes en una cola de impresión en una instalación multiusuario, y hasta programas que esperan su turno de CPU en un ambiente de programación concurrente. Para simular la llegada de un cliente, permiten almacenar objetos de la clase "CLIENTE" (o sus especializaciones) a través de la función "PUT", y para simular la atención permite recuperarlos, en el orden en el que fueron insertados (orden de llegada), a través de la función "GET". De esta manera, CliCola implementa un contenedor de tipo "FIFO" (First In, First Out) de propósito específico.

¿Cómo se usa?

1. Archivos Asociados:
..\CLICOLA.H;
..\CLICOLA.CPP;

2. ¿Requiere Especialización ? : No

3. ¿Requiere que el usuario genere o incluya clases auxiliares ? : Si, esta clase se usa en conjunto con la clase CLIENTE y SERVIDOR.

4. Casos: simulación de colas de impresión.

Estructura de la clase:

1. Relaciones

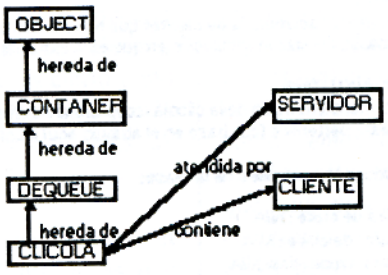
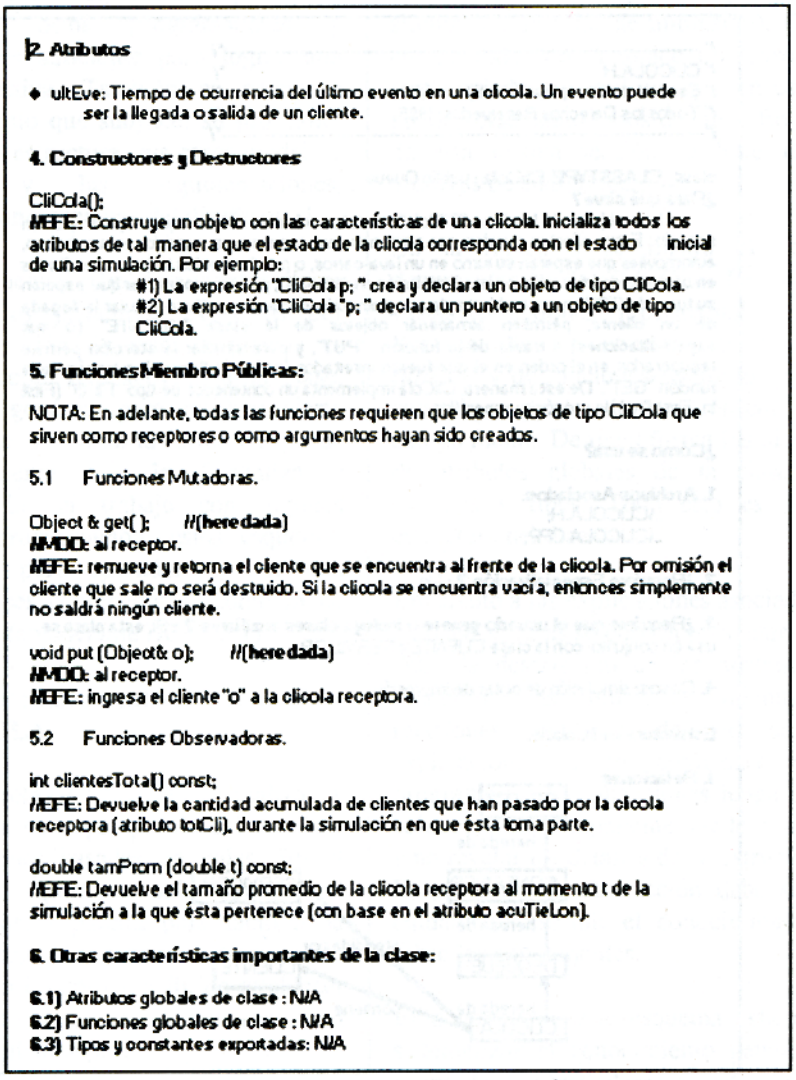


Figura N° 2: *Documentación de CLICOLA, parte #1*

Vista como diseño, *CLICOLA* responde a ciertos objetivos, los cuales se exponen en la sección **¿Para qué sirve?**. La estructura debe abarcar todas aquellas relaciones que el componente mantenga con otros componentes, así como aspectos de estructura interna y mecanismos de uso, que permitan aclarar al programador de aplicaciones la forma cómo usar este componente. Esto se incluye en las secciones **¿Cómo se usa?** y **Estructura de la Clase**.



2. Atributos

- ultEve: Tiempo de ocurrencia del último evento en una clicola. Un evento puede ser la llegada o salida de un cliente.

4. Constructores y Destructores

CliCola():
//EFE: Construye un objeto con las características de una clicola. Inicializa todos los atributos de tal manera que el estado de la clicola corresponda con el estado inicial de una simulación. Por ejemplo:
#1) La expresión "CliCola p; " crea y declara un objeto de tipo CliCola.
#2) La expresión "CliCola *p; " declara un puntero a un objeto de tipo CliCola.

5. Funciones Miembro Públicas:

NOTA: En adelante, todas las funciones requieren que los objetos de tipo CliCola que sirven como receptores o como argumentos hayan sido creados.

5.1 Funciones Mutadoras.

Object & get( ); //(heredada)
//MODI: al receptor.
//EFE: remueve y retorna el cliente que se encuentra al frente de la clicola. Por omisión el cliente que sale no será destruido. Si la clicola se encuentra vacía, entonces simplemente no saldrá ningún cliente.

void put (Object& o); //(heredada)
//MODI: al receptor.
//EFE: ingresa el cliente "o" a la clicola receptora.

5.2 Funciones Observadoras.

int clientesTotal() const;
//EFE: Devuelve la cantidad acumulada de clientes que han pasado por la clicola receptora (atributo totCli), durante la simulación en que ésta toma parte.

double tamProm (double t) const;
//EFE: Devuelve el tamaño promedio de la clicola receptora al momento t de la simulación a la que ésta pertenece (con base en el atributo acuTieLon).

6. Otras características importantes de la clase:

6.1) Atributos globales de clase : N/A
6.2) Funciones globales de clase : N/A
6.3) Tipos y constantes exportadas: N/A

Figura N° 3: *Documentación de CLICOLA, parte #2.*

La plantilla que se ha utilizado para documentar cada una de las funciones miembro es la recomendada por Liskov y Guttag[1]. Este esquema ha demostrado ser de mucha utilidad para la documentación de funciones, así como para su especificación, durante el proceso mismo de diseño del componente.

2.3 Browsers, estructura y función.

Si no se trata solamente de producir componentes de calidad, sino también de sistematizar el conocimiento sobre estos, entonces cabe preguntarse ¿qué tipo de conocimiento construye un programador de aplicaciones sobre un componente, conforme lo usa?.

Evidentemente algunos aspectos de la estructura interna del componente pueden ser útiles para usarlo en forma apropiada, de ahí que este sea parte del conocimiento construido. El PA también llega a conocer los servicios que ofrece el componente. En el caso de una clase, los servicios están representados por medio del conjunto de atributos públicos de la clase, sobre todo las funciones (o métodos). Además, el PA llega a conocer algunas relaciones estructurales que tiene el componente con otros de la base. Es difícil llegar a conocer todas las relaciones, dado que pueden ser muchas, y que pueden estar asociadas con detalles irrelevantes de la implantación. En el caso de las clases, se trata básicamente de las relaciones de herencia y composición. Todo esto, permite al PA conectar un componente con otros, a efecto de construir alguna aplicación específica. Esto es cierto tanto en funciones y clases, como en componentes más complejos, como los *frameworks* o marcos de trabajo[4] y los susbsistemas.

Sin embargo, también es evidente que el conocimiento construido no se limita a las tres categorías descritas. El PA aprende que un componente dado puede ser parte de una estrategia para resolver cierto tipo de problemas. Por ejemplo, un programador aprende que la clase pila es útil cuando se trata de eliminar la recursividad en algoritmos que la requieren, y que esto puede reducir el tiempo de ejecución de los programas. También aprende que puede ser útil para implantar procesos de búsqueda, cuando el árbol recorrido no se representa exhaustivamente. Es decir, aprende a identificar situaciones, problemas o contextos en que un componente puede ser útil; a la vez que aprende estrategias para usarlo, junto con otros, en tales situaciones, problemas o contextos específicos. Parte del conocimiento generado sobre un componente consiste en distinguir en qué situaciones y por qué es que conviene usar una pila en vez de una cola, una lista o un diccionario. Esto es lo que en psicología cognoscitiva se ha denominado ***metaconocimiento***. Aunque el caso de los contenedores es muy familiar y pertenece al orden de los conceptos básicos de programación, es claro que la misma situación se presenta con bibliotecas de componentes más especializados.

[4] Un *framework* o marco de trabajo es un conjunto de clases y sus relaciones estructurales.

2.3.1 ¿Qué es metaconocimiento?.

Sternberg hace la siguiente propuesta sobre la forma en que se organiza el conocimiento y las unidades en que se estructura:

"Un componente es un proceso de información elemental que opera sobre representaciones internas de objetos o símbolos... Un componente puede transformar un estímulo sensorial en una representación conceptual, o una representación conceptual en otra, o transformar una de éstas en una respuesta motora. Qué se puede considerar como *elemental* es visto como una propiedad del nivel teorético con que se aborda un estudio, más que una propiedad de la mente humana. Un componente dado puede o no ser elemental dependiendo del contexto teórico en que es presentado." [Sternberg, 88].

Sternberg identifica tres grandes categorías de componentes básicos:

"(1) los metacomponentes, que son procesos ejecutivos usados para planear, monitorear y evaluar las estrategias que uno adopta para resolver problemas; (2) componentes de desempeño, que son procesos no-ejecutivos usados para realizar las instrucciones de los metacomponentes para la resolución de problemas; y (3) los componentes de asimilación

cognoscitiva, que son procesos no-ejecutivos usados primordialmente para aprender cómo resolver problemas.

Las tres categorías de componentes son altamente interactuantes, lo cual significa que deben ser entendidos y entrenados en forma integral." [Sternberg, 87].

El primer tipo de conocimiento descrito en esta sección está más cerca de lo que Sternberg denomina componentes de desempeño. Este conocimiento corresponde a la estructura de los componentes reutilizables[5], mientras que el metaconocimiento corresponde a la función de estos.

2.3.2 El concepto de plan.

Otra forma de plantear la diferencia entre conocimiento sobre la estructura y conocimiento sobre la función de los componentes reutilizables es introducir el concepto de plan. Los ***planes*** son piezas básicas en la organización del metaconocimiento que un programador desarrolla sobre una base de componentes reutilizables. Los planes son genéricos pues no tienen por objetivo resolver un problema concreto, su valor consiste en dar respuestas a categorías de problemas parecidos. Por otro lado las estrategias también son flexibles y adaptables a las circunstancias específicas.

La noción de plan ha sido utilizada por muchos psicólogos de la cognición en estudios empíricos sobre el comportamiento de programadores[6], principalmente en modelos orientados a la comprensión de programas. Las investigaciones referidas se han centrado en programadores que usan lenguajes procedimentales (o imperativos) como *Pascal*, *Fortran*, etc. En el contexto actual, es necesario extender el concepto de plan no solo para abarcar situaciones de reutilización más complejas (pues los componentes en sí son más complejos), sino también otros niveles de reutilización (ver sección aparte). Para los efectos de este trabajo, un ***plan*** se caracteriza como una estrategia genérica orientada a la reutilización de piezas de *software*[7] que abarca: 1) metas o propósitos, 2) conocimiento sobre un conjunto de piezas de *software* y 3) un procedimiento para adecuar, adaptar u organizar los componentes, de acuerdo con las metas y una aplicación específica[8]. En este caso las piezas son principalmente clases.

Debido a que los *browsers* actuales solo documentan la estructura de los componentes y no los planes aprendidos, es útil la diferenciación entre metaconocimiento y conocimiento operacional (componentes de desempeño) para sustentar un punto de vista crítico que permita plantear herramientas de *software* cuyo diseño esté orientado a la función. Al respecto señala Fischer ([Fischer, 87], pg. 60): "La reutilización y el rediseño exitosos enfrentan muchos problemas desafiantes a nivel básicamente cognoscitivo. Es un gran error partir del supuesto de que la reutilización y el rediseño no plantean retos a nivel de diseño. Se necesitan nuevas arquitecturas y herramientas de soporte para reducir las demandas cognoscitivas".

En conclusión, una estrategia integral de reutilización requiere *software* que facilite la sistematización del metaconocimiento (los planes) que un grupo dado de programadores genera sobre un conjunto de componentes. Para

---

[5] Se previene al lector para que no confunda el concepto de componente de Sternberg (una entidad netamente psicológica) con los componentes reutilizables, que son la base de la reutilización de *software*.

[6] Ver por ejemplo: [Letovsky,87], [Holt, 87], [Rist,87] y [Soloway,84].

[7] Ya sean simples trozos de código, funciones, clases, frameworks, o subsistemas.

[8] Esta misma noción puede modificarse levemente para que, en vez de conocimiento sobre componentes de *software*, incluya conocimiento sobre otras estructuras formalizadas como trozos de modelos entidad-relación. Esta variante sería útil la actividad de análisis de *software* de aplicación.

finalizar esta sección, no se debe perder de vista que la actividad de desarrollo de *software* no se puede reducir a la utilización de clases y funciones o marcos de trabajo, es decir, existen planes que ***no*** tienen que ver directamente con la utilización de un componente básico, este es el caso de los patrones de diseño propuestos por Gamma, en [Gamma, 95]. Los autores de este novedoso libro proponen un esquema orientado a documentar *planes del nivel de diseño*, es lo que denominan **patrones de diseño**. Para estos autores, un patrón de diseño tiene un nombre, y está compuesto por la descripción de un problema y su contexto, la solución genérica, y las consecuencias o decisiones de diseño implícitas en la aplicación del patrón ([Gamma, 95], pg. 3).

2.4 Niveles de reutilización.

Así como Soloway (1984), Rist (1987) y otros se centran en los planes desarrollados por los programadores para reutilizar funciones o procedimientos, y Gamma se centra en planes a nivel de diseño, es razonable pensar que "aislar" planes a nivel de la actividad de análisis puede ser de gran utilidad. Aunque el modelo tradicional de desarrollo de *software* planteaba una división tajante entre análisis, diseño y programación, actualmente, es casi un lugar común la aceptación de fronteras difusas entre estos tres tipos de actividades, así como la visualización de un proceso de desarrollo cíclico en vez del lineal conceptualizado en el modelo de cascada.

Sostener que es útil introducir el concepto de plan a nivel de la actividad de análisis de sistemas no sólo es una consecuencia natural del planteamiento que se ha desarrollado en este trabajo, sino que está sustentado en los supuestos que han hecho tanto los psicólogos (a nivel sobre todo de la programación) como los ingenieros de *software* que han trabajado el tema de la reutilización (en el caso de Gamma a nivel del diseño). Por otro lado, es útil visualizar los planes a nivel de la programación como estrategias de implantación de los planes a nivel de diseño, y estos a su vez como estrategias de implantación de los planes a nivel del análisis. Una organización de este tipo permite establecer asociaciones relevantes entre los objetivos finales de todo proceso de desarrollo de *software* (desarrollar aplicaciones orientadas a necesidades de una comunidad de usuarios) y las piezas básicas provistas por un ambiente de desarrollo. Este tipo de asociaciones puede contribuir a la organización de la experiencia que un grupo de desarrolladores tiene con una base de componentes reutilizables.

Finalmente, parece altamente factible que una investigación empírica con analistas de sistemas sobre la forma en que comprenden modelos conceptuales de sistemas (ya sea modelos entidad-relación o modelos de objetos basados en los lenguajes gráficos de Rumbaugh (1996), o Booch (1996)), lleve a conclusiones similares a las obtenidas en el dominio de los programas. Por todas estas razones surge la necesidad de crear *software* de apoyo a la reutilización de conocimiento, que abarque los tres niveles propios de su actividad de desarrollo.

2.5 Acumulación y acceso al conocimiento.

Si se acepta la importancia que tiene sistematizar el metaconocimiento generado por un grupo de desarrolladores en relación con una base de componentes, entonces la siguiente pregunta es ¿qué tipo de procedimientos y mecanismos se pueden utilizar para construir una base de conocimientos[9] que permita la acumulación y el acceso eficiente?. No es fácil responder a esta pregunta, sin embargo, con base en el desarrollo tecnológico actual se pueden establecer algunos puntos de referencia:

La tecnología de hipertextos ofrece la arquitectura básica para la estructuración y

[9] No se usa el término aquí en su acepción más técnica, como se hace en el contexto de los desarrollos en inteligencia artificial.

almacenamiento computarizado de la base de conocimientos.

En cuanto al acceso, existen en la actualidad varias posibilidades:
técnicas usadas en las bases de datos documentales[10],
técnicas basadas en la representación formal del conocimiento de la base hipertextual por medio de lenguajes lógicos. Los sistemas que usan este enfoque se conocen en la actualidad como administradores de conocimiento[11],
técnicas basadas en la representación por casos del conocimiento de la base hipertextual. Aunque aplicado a otro contexto, Chandler señala: "Un sistema consejero basado en casos almacena una biblioteca de casos indizados por atributos que anticipan su utilidad, y los pone a disposición de los usuarios en forma apropiada. Funciona interactivamente con un experto humano en la solución de cierto tipo de problemas, aumentando la memoria del humano con las experiencias de otros y suministrándole casos para que use en sus razonamientos." [Chandler, 94], pg. 285.

La alimentación de la base hipertextual demanda, por un lado, la definición de procedimientos de trabajo que aseguren la producción del conocimiento por parte de las personas. Por otro lado, si se adopta una estrategia de acceso basada en técnicas de inteligencia artificial, será necesario usar intérpretes artificiales de lenguaje natural, a fin de mantener actualizada la base.

Finalmente, como ejemplo de los problemas descritos y de las soluciones esbozadas, en el campo de la manufactura de piezas mecánicas, encontramos que:

"... existen numerosas descripciones escritas que recogen diversas explicaciones de expertos en varios dominios. Estas descripciones carecen de todo tipo de esquematización: son imprecisas, abarcan diferentes niveles de generalidad, son incompletas y a veces hasta contradictorias. En otras palabras, el conocimiento del campo está muy mal documentado. Sin embargo, este conocimiento es necesario para resolver problemas tales como la manufactura de partes que rotan. Debido a la cantidad y variedad de conocimiento que forma parte de la solución de tales problemas, el mero planeamiento del proceso de manufactura es complejo." [Schmalhofer, 92], pg. 406.

El sistema basado en conocimiento que proponen, pretende facilitar el planeamiento de la manufactura de piezas mecánicas mediante la reutilización de planes previamente elaborados. Esto implica la extracción de conocimiento a partir de documentos que describen los planes, y la sistematización de los documentos por medio de un sistema hipertextual. Los autores enfatizan el papel que puede jugar el hipertexto visto como una primera fase de inserción del sistema basado en conocimiento. El rechazo natural que despierta la introducción de tecnología novedosa, puede verse mitigado mediante la implantación del sistema de hipertexto como un primer paso. Estos autores muestran cómo "...una documentación inteligente en forma de una estructura hipertextual de dominio específico es construida a manera de representación intermedia de conocimiento. Puesto que esta documentación inteligente rendirá sus frutos por sí misma, contribuirá a reducir la barrera generada por la introducción de sistemas basados

[10] Ver por ejemplo: 1) Prieto-Díaz, Rubén & Freeman, Peter. Classifying Software for Reusability. *IEEE Software*, enero-87. 2) Maarek, Yoëlle S. & Smadja, Frank A.. Full Text Indexing Based on Lexical Relations An Application: Software Libraries. *Proceedings of SIGIR'89*. pgs. 198-206, Cambridge, MA, junio-89. ACM Press. 3) Helm, Richard & Maarek, Yoëlle S. Integrating Information Retrieval and Domain Specific Approches for Browsing and Retrieval in Object-Oriented Class Libraries. *Proceedings of OOPSLA'91*, pgs. 47-61.

[11] Una búsqueda en INTERNET usando los descriptores KNOWLEDGE MANAGERS ofrecerá gran cantidad de información al lector interesado.

en conocimiento." (pg. 406 de [Schmalhofer, 92]).

2.6 El análisis de dominio: método para la generación de conocimiento

Hasta aquí se ha considerado la sistematización (estructuración, acumulación y acceso) del conocimiento sobre una base dada de componentes, pero ¿qué pasa cuando se construyen nuevos componentes reutilizables?. El análisis de dominio, tal como ha sido enfocado por diversos autores[12], apunta hacia la producción de conjuntos de piezas reutilizables de *software*, con un grado apropiado de calidad y utilidad. Estos conjuntos de componentes pueden estar orientados a un dominio de aplicación específico—análisis de dominio vertical—o a un conjunto de dominios—análisis horizontal, en todo caso, el énfasis ha estado en los componentes. Desde la perspectiva sostenida en este trabajo, la construcción de nuevos componentes puede verse como la formalización y operacionalización de nuevos conceptos para resolver problemas nuevos. El análisis de dominio es así una oportunidad muy especial para sistematizar conocimiento sobre un dominio nuevo. La utilización de esquemas como el propuesto en este trabajo para clases, o el propuesto en [Gamma, 95] para *frameworks* (aunque no ha sido este el propósito original de los autores), pueden ser útiles en el aprovechamiento del conocimiento generado durante un proceso de análisis de dominio.

3. CONCLUSIONES.

No solo se ha señalado la importancia de reenfocar los esfuerzos de reutilización para que abarque la sistematización de la experiencia de una organización que desarrolla *software*. Se han esbozado procedimientos de trabajo con base en los esquemas para documentar clases y los patrones de diseño aportados por Gamma. Se ha sustentado el enfoque propuesto con base en la reformulación del concepto de plan dentro del marco de referencia de Perkins—con su noción del conocimiento como diseño—y de Sternberg—con su noción de metaconocimiento. Aunque provenientes de campos distintos, se han dado ejemplos de sistemas de apoyo que se podrían trasladar a una organización que desarrolla *software*, para ser usados como parte de una estrategia de reutilización integral. Finalmente, se ha analizado brevemente las implicaciones del enfoque sostenido para los métodos de análisis de dominio.

La puesta en práctica de algunas de estas ideas requiere de la coalición estratégica entre la universidad y las empresas (o, en términos más generales, las organizaciones) productoras de *software*. A la universidad corresponde el papel de la apropiación tecnológica, no solo de sistemas específicos, sino también de metodologías. A las empresas la decisión de analizar la implantación de una estrategia de reutilización integral como una inversión a recuperar en el mediano y largo plazo. No se puede perder de vista que cualquier intento inicial será necesariamente experimental. Es necesario que, conjuntamente universidad y organizaciones desarrolladoras, emprendan una serie de experimentos tendientes a definir las características de las estrategias de reutilización integral más apropiadas para nuestra industria de *software*. Paralelamente es conveniente estudiar las estrategias de reutilización a nivel inter-organizacional, enfocando la industria nacional de *software* como un todo. El aprovechamiento de las ventajas comparativas de nuestro país, en el mercado internacional, probablemente dependa de este esfuerzo de investigación que no podrá darse, en forma unilateral, en el interior de las universidades.

[12] Ver pgs. 180, 181, 289, 290 y 375-80 de [Booch, 96], este autor aporta mucha bibliografía relacionada con el tema. Además pgs. 182-195 de [Berard, 93]. De Prieto-Díaz, en [Booch, 96] aparece referencia a un trabajo del 91.

4. BIBLIOGRAFÍA.

Beck, Kent. *Patterns and Software Development.* Dr. Dobb's Journal, febrero-94.

Bernard, E. Essays on Object-Oriented Software Engineering. Englewood Cliffs, NJ: Prentice-Hall, 1993.

Booch, Grady. Análisis y diseño orientado a objetos con aplicaciones. Addison-Wesley Iberoamericana, segunda edición. Delaware, E.E.U.U., 1996.

Chandler, Terrel N. *The Science Education Advisor: Applying a User Centered Design Approach to the Development of an Interactive Case-Based Advising System.* Journal of Artificial Intelligence in Education 5(3), 283-318, 1994.

Fischer, Gerhard. *Cognitive View of Reuse and Redesign.* IEEE Software, julio-87.

Gamma, E., Helm, R., Johnson, R. y Vlissides, J. Design Patterns: Elements of reusable object-oriented software. Addison-Wesley, Massachusetts, 1995.

Griss, Martin L. *Software reuse: A process of getting organized.* Object Magazine, 5(2) mayo-95.

Holt, Robert W.; Boehm-Davis, Deborah A. & Shultz, Alan C. *Mental Representations of Programs fos Student and Professional Programmers.* En Empirical Studies of Programmers: Second Workshop, Ed. por G. Olson, S. Sheppard y E. Soloway. Ablex Publishing Corp. New Jersey, 1987.

Letovsky, Stanley. *Cognitive Processes in Program Comprehension.* En Empirical Studies of Programmers, Ed. por E. Soloway y S. Iyengar. Ablex Publishing Corp. New Jersey, 1987.

Perkins D.N. Knowledge as Design. En Teaching Thinking Skills: Theory and Practice. Editado por Baron, Joan Boykoff & Sternberg, Robert J. W.H. Freeman and Company, New York, 1987.

Rist, Robert. S. *Plans in Programming: Definition, Demostration, and Development.* En Empirical Studies of Programmers. Editado por E. Soloway y S. Iyengar. Ablex Pub. Comp., New Jersey, 1987.

Rumbaugh, James; Blaha, Michael; Premerlani, William; Eddy, Frederick & Lorensen, William. Modelo y diseño orientado a objetos. Prentice Hall, España, 1996.

Schmalhofer, Franz; Reinartz, Thomas & Tschaitschian, Bidjan. *Intelligent Documentation as a Catalyst for Developing Cooperative Knowledge-Based Systems.* Lecture Notes in Artificial Intelligence 599 (Current Developments in Knowledge Acquisition)—EKAW' 92 6th European Kowledge Acquisition Workshop. Pgs. 406-424. Publicado por Springer-Verlag, mayo-1992, Alemania.

Soloway, Elliot. y Ehrlich, Kate. *Empirical Studies of Programming Knowledge.* En IEEE Transactions on Software Engineering, SE-10, 5, 1984.

Sternberg, Robert. J. *Mechanisms of Cognitive Development: A componential approach.* En Mechanisms of Cognitive Development (segunda edición) editado por Sternberg R.J. Illinois: Waveland Press, 1988.

Sternberg, Robert. J. *Teaching Intelligence: The Application of Cognitive Psychology to the Improvement of Intellectual Skills.* En TEACHING THINKING SKILLS: Theory and Practice editado por Boykoff J. y Sternberg R. New York: Freeman and Company, 1987.