XIV Conferencia Latinoamericana de Informática
17avas. Jornadas Argentinas de Informática
e Investigación Operativa
Buenos Aires, Setiembre 1988.

Programación y Metodologías Orientadas a Objetos y las Metodologías avanzadas de Diseño de Soft.

Gustavo A. Pollitzer
Universidad de Belgrano
Facultad de Tecnología
Echeverría 2838
1426 Buenos Aires

Resumen:

Los términos Programación y Metodologías Orientadas a Objetos han tenido mucha difusión en los últimos años. Su promoción viene en algunos casos ligada a productos o a escuelas que promueven su aplicación desplazando otras herramientas.

El propósito del presente trabajo es:
- Analizar las llamadas Metodologías Orientadas a Objetos
- Detectar sus componentes escenciales y conceptuales.
- Rever el proceso de Diseño de Soft y detectar sus necesidades.
- Comparar las MOO con otras metodologías que se aplican a este fin.
- Ampliar las estrategias usadas para diseño incorporando los nuevos conceptos y herramientas, desarrollando así, aún más, el conjunto de Metodologías avanzadas de Diseño de Soft, tomando lo mejor de cada mundo y dejando de lado elementos no fundamentales de una escuela dada.

Metodologías Orientadas a Objetos. Conceptos fundamentales.

Las Metodologías Orientadas a Objetos toman al Objeto como la entidad básica o componente esencial para modelizar el mundo real o el problema a encarar. Ese Objeto tendrá atributos (o ítems de datos) y propiedades y estará sujeto a procesos. Su comportamiento se caracteriza por las acciones que sufre y que requiere de otros objetos. (Booch, 86)

Equivale a una estructura de datos de tipo Record. (Jacky y Kalet, 87).

Un Objeto es una instancia de una Clase a la que pertenece. Esa Clase equivale al Tipo de Datos del Objeto. Asociado con este tipo de datos, habrá un conjunto de operaciones o métodos.

La regla fundamental es que todo atributo de un Objeto sólo es accesible por sus métodos. Se implementa así algo equivalente a una cápsula de Tipo Abstracto de Datos.

De una clase puedo derivar nuevas clases o subclases, como una extensión de la anterior, su superclase. Al declarar que una clase es subclase de otra, la nueva incluye (o hereda) los atributos y los métodos de la superclase.

Podrá además ser ampliada con sus propios atributos y métodos. Si así se desea, los valores de los atributos de la superclase pueden ser redefinidos (overriden) por una nueva declaración de su valor en la subclase o en un objeto dado. En este caso deberíamos tener conciencia de las objeciones conceptuales al respecto.

Los métodos son invocados o activados por mensajes emitidos por otros métodos. El Polimorfismo implica que el programador no necesita ocuparse del tipo de los datos y los objetos, que serán interpretados por los métodos según sean las características de los objetos a que fueron destinados los mensajes.

El análisis del orden de activaciones de los métodos o procedimientos muestra que los mensajes son colocados en una estructura con la política de atención necesaria para las sucesivas invocaciones encadenadas y para la posible recursión. Un despachador se encargará a su turno de activar el método correspondiente al destinatario elegido. Así visto, debemos considerar la equivalencia con el mecanismo normal de secuencia de sucesivas llamadas a subrutinas o procedimientos.

Tipo Abstracto de Datos, Cápsula:

Defino un Tipo Abstracto de Datos por las Entidades que ha de representar, sus Atributos, y las Operaciones que podré efectuar sobre ellos.

Al usar el término Tipo Abstracto de Datos no nos referiremos a temas de especificación formal, sino a la implementación de una entidad representada por una estructura de datos y todas las funciones para acceder o hacer uso de la misma, en una entidad única o en un tipo, que permita generar repetidas instancias de la misma.

Llamamos Cápsula al módulo que, en un lenguaje adecuado implemente un TAD. Tendrá así inicialización y permanencia de los datos de la estructura y las instancias que creo al representar cada una de las entidades de ese tipo, y contendrá la implementación de los procedimientos que me permiten efectuar las operaciones correspondientes.

Proveerá protección contra accesos externos y ocultamiento de información respecto de la estructura y su forma, y sólo comunicará y exportará al exterior los detalles respecto a su forma de uso e invocación, y su comportamiento.

Los lenguajes que apoyan estos conceptos permiten separar los componentes de estos módulos en dos partes: una de ellas con los datos descriptivos que hacen a su uso y vinculación con el resto del sistema, que llamaremos Definición, y otra con la construcción real de los procedimientos y estructuras que los implementan, a la que nos referiremos como Implementación.

En la Definición especificamos las interfases y en ella declararemos el nombre del tipo, que será así un Tipo de Datos Abstracto, opaco, con su contenido y operaciones en la Implementación. Manejamos y damos así la visibilidad deseada a cada item.

Aplicación de metodologías de Diseño:

(Booch, 86) detalla los pasos para aplicar la metodología de diseño por objetos, inicialmente propuestos por Abbott:
- Identificar los Objetos y sus atributos,
- Identificar las operaciones que sufre y que requiere cada Objeto,
- Establecer la visibilidad de cada objeto en relación con los otros,
- Establecer la interfase de cada objeto,
- Implementar cada Objeto.

Esta lista es muy semejante a los pasos a dar cuando uno selecciona y define una cápsula, que implementa un tipo abstracto de datos en un sistema.

Analicemos ahora el uso de las Metodologías derivadas del Diseño de estructura por Descomposición Funcional.

El comienzo es distinto cuando se parte de las funciones que se espera que cumpla el sistema. El primer refinamiento de esas funciones analiza las subfunciones principales. Este es el primer paso fundamental de Diseño. Pero inmediatamente, cuando se analizan funciones, debe considerarse sobre qué entidades se aplican éstas: datos o atributos de entrada o de estado del sistema, de sub-entidades permanentes o de entidades repetitivas o transitorias en el sistema.

Estas entidades pueden tener existencia en el mundo real o ser una creación del sistema de procesamiento.

Se pueden elegir a-priori, para modelizar, los objetos más significativos del mundo real y sus funciones características. Pero para representar éstos, probablemente debamos recurrir a algunos sub-componentes de los mismos.

No querremos representar todos los detalles de estos subcomponentes, sino sólo aquellos elementos y funciones necesarios para implementar las funciones que deseamos que el sistema lleve a cabo. Volvemos así al punto de partida de funciones y sub-funciones del sistema.

Algunas entidades son de puro origen computacional. La organización de entidades múltiples en estructuras como árboles, pilas, heaps, etc, hace a la eficiencia de los procesos. La selección de la estructura más eficiente depende fundamentalmente de las funciones que requeriremos de ella, que sólo podremos detectar en nuestro proceso de diseño.

Otras, como las ventanas o sistemas de menú, son necesidades de la interacción hombre máquina. Y sus atributos dependerán de las estrategias de uso del sistema.

Curiosamente, existen entidades computacionales o de cálculo, no ligadas a las computadoras, que casi parecerían tener existencia en el mundo real, como una "Cuenta Corriente" o un "consumo promedio". Podemos así pensar en dar vida a nuevas entidades de este tipo, si las requiere nuestro diseño.

Una vez definido un objeto o estructura por sus atributos y funciones, podremos comenzar a considerar su implementación, pero deberemos tener en cuenta los requerimientos de uso y respuesta de estas funciones. Para ello, la información esencial es la de frecuencia de uso de cada función, así como otros datos de volumen y eficiencia requerida.

Por supuesto, en su implementación, usaremos todos los recursos disponibles para ocultar y proteger la entidad, cumpliendo con los lineamientos de Parnas y construyendo Cápsulas.

En los objetos simples, cuya estructura fundamental es de tipo RECORD, este análisis no es significativo. Sí puede serlo el de las funciones, si implican procesos más complejos que observación, modificación o ingreso de un valor.

Las entidades computacionales, por otro lado, tienen una estructura básicamente compleja, lo que motiva su creación como entidad (u objeto) aparte. La elección de estrategias organizativas y de su utilización son tema central de diseño y a ellas dedicaremos nuestro esfuerzo.

Continuando con las Estrategias de Diseño, o elementos de metodologías que emplearemos, conviene anotar aquí las Máquinas de Estado, conceptualización que tendremos en cuenta para aquellas entidades de existencia individual y permanente, donde por supuesto, su comportamiento dependerá del estado que alcanzó en su trayectoria previa.

Tendremos que considerar, dentro de esta línea, los Módulos Incrementales y la estrategia de diseño, para los más complejos, de Inversión de Programa (Jackson, 75).

Por supuesto, entre las entidades de existencia individual y permanente debemos considerar al propio sistema y, por lo tanto, tener en cuenta este paradigma para su organización.

También siguiendo a Jackson, podemos dar estructura a los módulos de tratamiento e interpretación de comandos o mensajes con una gramática implícita.

En los refinamientos de arquitectura general, podemos considerar separarnos de la organización puramente jerárquica. Es evidente la racionalidad de esta alternativa para las situaciones de tipo Productor-Consumidor y así deberemos hacerlo si tenemos medios para implementar módulos a igual nivel en forma simple y elegante.

El manejo de Corrutinas permite estas soluciones al igual que da pie a la creación de objetos de tipo actor y su existencia y comportamiento simultáneos.

A este juego de estrategias y herramientas podemos agregar ahora, como derivado de la corriente orientada a objetos, el mecanismo de herencia, implementable como Tipos de Datos Derivados que permite separar entidades y estructuras en una forma no lograble y de dificil conceptualización sin este paradigma.

En el caso de objetos soportados por una estructura, esta separación se logra considerando a los objetos como herderos (es-un) de nodos de la estructura, que por su parte maneja a éstos en forma totalmente abstracta, sin conocer su significado o contenido, y con total separación de código.

Un enfoque distinto en el desarrollo de sistemas:

La metodología JSD (Jackson, 83) cuestiona todo el ciclo de vida tradicional y propone un proceso distinto, que comienza por la modelización del mundo real.

Pero inmediatamente, y en la misma justificación de la metodología, vincula modelo y función del sistema. Plantea como primera funcionalidad la de supervisión del sistema real, lo que permite realizar informes de actividad y comportamiento y comportamiento y detectar situaciones anormales.

Plantea, además, la modelización como forma de caracterizar la abstracción que encararemos del mundo real, y de corroborar su coherencia.

Los pasos que plantea la metodología son los siguientes:
- Caracterización de Entidades y Acciones del sistema
- Descripción de las acciones que ejecuta o sufre cada entidad en su organización temporal.
- Conexión entre los procesos del mundo real y los del modelo.
- Especificación de las funciones del sistema y las condiciones de ocurrencia e interrelación entre eventos.
- Consideraciones de tiempos en el sistema.
- Implementación del sistema.

Estos son pasos que cubren el espectro de la especificación del sistema y su implementación. Es interesante el análisis del paralelismo entre estas etapas y las especificadas por Abbott para el Diseño Orientado a Objetos.

En esta comparación debemos tener presente que la metodología JSD supone cubrir con las primeras etapas de modelización la especificación del sistema y que en las posteriores encara el diseño y la implementación funcional del mismo.

Lenguajes vinculados a Metodologías Orientadas a Objetos e incorporación de los conceptos de éstas en los lenguajes más tradicionales.

Los primitivos lenguajes procedurales disponían, como principal y casi único elemento vinculado a la éstructura, de la subrutina como paradigma de la abstracción algorítmica y la organización jerárquica (además de la localidad de variables y el paso de parámetros por interfase). Se soportaba, así, la descomposición funcional elemental. (Booch, 86)

El primer lenguaje en implementar Objetos fue Simula. Vinculado con el nacimiento de los lenguajes de Simulación (Dahl y Nygaard, 66), reafirma en esto nuestra idea de la vinculación de su uso e implementaciones en:

- La representación de Entidades del Mundo Real y
- La existencia y accionar de cada uno de ellos en forma simultánea e independiente, que no parece ser normalmente indispensable en un proceso definido en forma procedural.

También surgía de este enfoque la caracterización de objetos en Clases.

Por una línea aparte, se desarrollaron los lenguajes para implementar los conceptos de Tipos Abstractos de Datos como Alphard, CLU (Liskov, 81), Modula-2 (Wirth, 71) y como más inclusivo, Ada.

El concepto se hizo central en el enfoque del proyecto de Xerox PARC y la implementación de Smalltalk. (Goldberg y Robson, 83) (Goldstein y Bobrow, 80)

Se profundizó en la elaboración y aplicación de los conceptos de Herencia.

Sin embargo, dentro de esta escuela se promovieron otros conceptos que, por este hecho, quedaron adheridos al de Objetos. En especial, los de activación y paso de información a funciones y procedimientos.

A las características de métodos (procedimientos) ligados a Objetos y Clases (tipos abstractos de datos) y los mecanismos de Herencia, se unieron la activación y transferencia de información por Mensajes y el mecanismo de despacho para ejecución de cada uno de ellos.

Esto permitió dar un aspecto de no procedural al lenguaje, por el simple hecho de no indicarse cuando o en qué orden deben ejecutarse la funciones. Esto tiene mucha similitud con la característica de no determinismo de los mecanismos de iteración de E. Dijsktra y sus comandos con guardia, no tomados en sí como no procedurales.

Los lenguajes más avanzados de la actualidad poseen y soportan mecanismos de estructuración mucho más avanzados y flexibles. Para su uso adecuado deben considerarse otros tipos de abstracciones como la de datos, y la conceptualización como objetos es una forma directa de encararla. Los packages en Ada o módulos en Modula-2 permiten implementar estos enfoques.

En los principales lenguajes de programación, muchos de ellos de tipo procedural, se han incorporado mecanismos ad-hoc para implementar los conceptos de las metodologías de Objetos.

Así, en versiones de los lenguajes como Lisp, Pascal, C, se pueden definir y trabajar con Objetos, Clases, y aprovechar los mecanismos de Herencia con distintos niveles de generalidad.

Modula-2 (Wirth, 83) incorpora los principales mecanismos de localidad y permanencia de datos, ocultamiento de información por separación de la definición y la implementación, compilación por separado y manejo de la visibilidad de todos los componentes por importación y exportación.

Permite así implementar Tipos Abstractos de Datos en Cápsulas. También facilita las estructuras no jerárquicas por un directo manejo de corrutinas, que para nuestro estudio debemos vincular con la implementación de objetos actores.

Sin embargo, no podemos decir que permita en forma directa e integral representar algunos conceptos centrales de la metodología de Objetos.

Por supuesto, Ada incluye un conjunto de mecanismos que permiten implementar los conceptos derivados de las Metodologías Orientadas a Objetos. Esto lo podemos ver especialmente en el caso de los Tipos de Datos derivados.

El análisis conceptual realizado por (N. Wirth, 88) sobre extensión de tipos de datos busca definir nuevos tipos de datos, relacionados con uno existente tomado como base, cumpliendo reglas de compatibilidad.

La importancia de estas construcciones será significativa si el tipo base puede estar implementado por separado en un módulo compilado independientemente, que oculte sus detalles.

Su declaración es:

T´ = RECORD (T) ... declaraciones adicionales ... END

El tipo T´ es extensión del T . T es tipo base de T´.

El tipo T´tiene como componentes los declarados en la declaración extensiva más los que contiene T.

Podemos desconocer cuales son estos últimos, porque sólo se accede a ellos en el módulo que los declara o podemos conocerlos y tener acceso a los mismos.

Se pueden extender tipos ya extendidos previamente. También admite la Herencia múltiple.

Queremos usar este mecanismo para reproducir la relación Clase-Subclase: El Tipo Extendido (subclase) incorpora los atributos del Tipo Base (su clase).

Para ello debemos analizar los mecanismos que realizan el control de tipos y definir nuevas condiciones para los mismos. Nos interesa extender éstos para poder considerar a un elemento de un Tipo Extendido también como un elemento del Tipo Base.

Consideremos la asignación. Un tipo RECORD representa el producto cartesiano de sus tipos componentes. Un Tipo Base tendrá una dimensionalidad menor que la del Tipo Extendido. Para encontrar significación a la asignación de instancias de Tipo Extendido a entidades de Tipo Base debemos considerar la Proyección del primero sobre este último.

Resulta evidente que la asignación inversa no podría realizarse (quedarían componentes indeterminados), lo que concuerda con la asimetría de la relación Clase-Subclase.

(Wirth 88) extiende la definición ampliada a punteros a los Tipos Base y Extendido pero creo interesante analizar si esta extensión es necesaria.

Si los punteros del Tipo Base son sólo manejados dentro del módulo que los define, la única asignación de punteros del Tipo Extendido a los del Tipo Base se realiza en la invocación a funciones de este módulo (cápsula) para operaciones vinculadas con este tipo.

Wirth justifica ese relajamiento en las asignaciones de punteros por la muy importante posibilidad de construir estructuras con datos heterogéneos, por supuesto extensiones del Tipo Base. Pero si a esta estructura la consideramos como perteneciente a (las instancias de) el Tipo Base y protegida por la implementación del mismo, sería una estructura homogénea, que soporta integrantes heterogéneos en su diferenciación del Tipo Base.

Esta interpretación es paradójicamente apoyada por el mismo autor, en el punto en que considera la necesidad de la discriminación del tipo. Esta necesidad surge por el tratamiento de un puntero en el módulo del Tipo Base y la necesidad de interpretación del mismo en su devolución al ambiente más general, que reconoce las diferencias de los tipos extendidos.

Esto implica que, en el dominio de los punteros, cuando aceptamos uno de ellos como perteneciente a una Clase (Tipo Base), pasa a integrar un conjunto y no pierde su individualidad, que podremos reconocer al identificarlo para ser tratado en el entorno de los tipos extendidos.

Otra conceptualización posible es la de que los punteros no poseen tipo propio sino representan al tipo de lo apuntado. Y es esto último lo que se consultará para la discriminación.

Plantea Wirth para esto una operación de prueba de tipo (v IS T : BOOLEAN) y una muy interesante guardia de tipo, que implicará que la aplicación de la operación está condicionada a que los componentes correspondan al tipo especificado o a una extensión de éste.

```
v : T
v := x (T)     condiciona la ejecución a que x sea del tipo T o
               una extensión de éste.

Type
     T = RECORD
                    c : ...        x (T). c  condiciona el acceder
                    .....     al campo  c  a que  x  sea de tipo  T
          END
```

Polimorfismo en procedimientos: Esta propiedad implica poder aceptar para su tratamiento, no sólo parámetros de un único tipo, sino una variedad de éstos.

Se relajan así las reglas de control de tipo. La definición de este relajamiento creada para los tipos extendidos cubre esta necesidad.

Es en esta asignación de valores a los parámetros formales de la interfase, que proponemos la consideración (y relajamiento) de los tipos de datos correspondientes a punteros.

Beneficios de las Metodologías Orientadas a Objetos:

Una característica significativa de estas metodologías es que fuerzan a trabajar de forma tal que se promueven buenas prácticas de Diseño:

- Una correspondencia más directa con los entes reales a simular o representar
- Modularidad
- Encapsulamiento, con bajo acoplamiento y ocultamiento
- Reusabilidad
- Facilidad de extensión.

Los seguidores de estas estrategias enfatizan los siguientes aspectos (Jacky y Kalet, 87):

- Más fácil modelización del mundo real o del comportamiento de los items a ser modelizados y mejor (más directo) diseño del sistema.
- Menos errores, por modularidad y encapsulamiento.
- Más fácil de extender, por facilidad de incorporar nuevos objetos (clases). Cada nuevo objeto (clase) se incorpora con todos sus métodos.

Estas características deseables de las Estructuras de los Sistemas de Soft ya habían sido incorporadas en metodologías y herramientas anteriores de Diseño.

Analicemos un poco más el último argumento. Se supone en él que se querrá incorporar objetos nuevos más frecuentemente que nuevas operaciones que afecten varias Clases existentes. Si este último fuera el caso, habría que tocar cada objeto para definir el método correspondiente para la nueva operación.

Objeta que en la organización "clásica procedural" cada operación tiene un CASE orientado a cada tipo de objetos, que debería ampliarse al incorporar nuevos. Esto no es así si hemos implementado cápsulas por objetos.

En resumen, opinamos que las Metodologías Orientadas a Objetos:
- Ayudan a que los que se inician incorporen buenas costumbres de diseño.
- Son una estrategia más a aprovechar.
- Parte de los conceptos y las herramientas desarrolladas para soportarlos permiten ampliar y dar más fuerza a las buenas estrategias avanzadas de Diseño de Estructura e implementar lo ya concebido conceptualmente.

Aporte significativo al Diseño de Soft: Mayor separación funcional

Separar conceptos equivale a separar código. Las nuevas herramientas y conceptualizaciones pueden ayudar en este sentido.

Si bien corresponden a funciones diferentes, si una entidad está dentro de un árbol (o cualquier otra estructura), las intrucciones para pasar al próximo elemento están en general mezcladas con las de procesar el nodo. Y los datos y punteros también, todos en la misma cápsula.

Tenemos ahora herramientas para separarlas. Puedo ahora implementar operaciones independientes sobre:
- un elemento,
- la estructura que los soporta y organiza (arbol, jerarquía, secuencia)
- los elementos auxiliares de representación: tipos elementales, contenidos o ventanas, etc.

Esta posibilidad, que no sólo promueve el desacoplamiento entre funciones sino que permite crear módulos independientes con funciones esenciales y así dar un sólido material para reusabilidad, puede llegar a ser más importante que los otros frutos de las metodologías orientadas a objetos.

Análisis de los argumentos sobre Modelización del Mundo Real por Objetos:

Las Metodologías Orientadas a Objetos se prestan mucho para expresar o modelizar el mundo real. Un mundo de complejidad baja, en que no haya todavía una significativa tradición de Sistemas de Información (o no la tenga el implementador o usuario).

En los sistemas de complejidad algo mayor o con tradición de Sistemas de Información, hay que hilar más fino e interactuar con archivos, otros sistemas de información, la estructura de los elementos de soporte y cómputo, y crear organizaciones complejas para hacer eficiente el proceso.

Todo esto implica el trato con un conjunto importante de datos, estructuras y procedimientos que podemos seguir manejando con la metodología de objetos, pero que, forzar este enfoque puede ser una imposición pesada, por aferrarse a la metodología, más que una solución práctica y con iguales propiedades de claridad y controlabilidad de la solución que su aplicación a los objetos básicos.

Un ejemplo extremo lo tendríamos si deseáramos hacer un sistema para jugar al ajedrez. Tomaríamos las piezas y los casilleros del tablero como objetos y sus movimientos posibles como métodos. Será así muy facilmente manejable la representación de una posición dada o la visualización de las jugadas. Pero rápidamente notaríamos que el meollo del problema no ha sido atacado. Sabemos que por esta via sólo se tiene una solución impracticable, por la enorme cantidad de alternativas que deben ser analizadas para evaluar una jugada. Y deberemos crear una cantidad de estrategias y conceptos, medidas y evaluaciones para las que la conceptualización como objetos puede ser una carga intelectual más.

Aún en los casos simples, el verse forzado a considerar que 3 + 4 es un mensaje cuyo receptor es el objeto 3, siendo + es el selector del método a aplicar y 4 el argumento, puede resultar muy pesado si se lo toma como algo más que un divertimento conceptual...

Todas las funciones, disponibles:

Una de las propiedades que se promueven en el uso de un lenguaje orientado a objetos es el poder disponer permanentemente de todas las funciones definidas para todos los objetos ya especificados en el sistema.

La imagen de esta característica, desde el punto de vista de un lenguaje procedural equivale a un ambiente donde todas las funciones o procedimientos definidos previamente estan accesibles para ser llamados por el sistema que estoy armando.

Esto equivaldría a un ambiente en Pascal donde no comienzo a realizar un programa desde cero, sino que incluyo inicialmente todas las funciones definidas previamente. Esta visión sería más natural si pensamos en un ambiente como el de APL, donde en un workspace estan realmente presentes todas las funciones definidas en el sistema. Como operación característica existe la consulta de la lista de operaciones disponibles en el espacio.
Algo equivalente encontramos en ambientes LISP.

Rápidamente detectamos la facilidad para disponer de esta propiedad en lenguajes interpretados. En ambientes de lenguajes compilados, la unidad con que se trabaja es el elemento a compilar, que no se sobrecarga con funciones que podrían o no ser utilizadas por el sistema en desarrollo.

Este problema es encarado por los lenguajes más modernos, donde se considera fundamental la compilación por separado, y la posibilidad de invocar funciones ubicadas y provistas por otros módulos, que serán vinculadas al actualmente en desarrollo en momento de linkedición. Tendremos así acceso a todo el conjunto de funciones, que estarán ubicadas en nuestra biblioteca de módulos. Como recordaremos, esta organización era utilizada en la década del 60...

Por otra parte, el concepto de compilación por separado se vincula con el de ocultamiento de información y el de limitación del acceso a las funciones y la descripción de su implementación. Por lo tanto, la existencia de todas las funciones en un mismo ambiente, en un lenguaje en el que quisiéramos imitar la programación orientada a objetos pecaría por falla en este sentido, salvo que el lenguaje proveyera dispositivos para este fin (funciones protegidas, etc.). Otro problema relacionado es el de nombres de las funciones, que deberán tener mecanismos de diferenciación en su incorporación o importación.

Encontramos este mismo tipo de propiedad en ambientes con sistema operativo transparente, es decir, donde el sistema operativo no se hace evidente al usuario, quien considera estar siempre expresándose en lenguaje de programación o comando, en el mismo nivel. Aquí todo está a su alcance, para invocación o uso desde cualquier sistema.

El diseño tradicional no implica el uso generalizado de un Common:

Leemos, en un trabajo donde se promueven metodologías de diseño orientadas a objetos que "Finalmente en los diseños con las metodologías de descomposición en funciones, se termina poniendo todas las variables en acceso global...". Esto es ignorar reglas fundamentales de la metodología propuesta por Constantine, Yourdon y seguidores donde, en las mediciones de bondad de diseño, se penaliza al ubicar las variables y las estructuras en un common o ambiente global con las peores notas en grado de acoplamiento, medida fundamental del diseño. Esta calificación es síntoma evidente de un diseño que debe reverse.

Por supuesto debemos transar con las limitaciones que nos plantea el lenguaje con que podemos trabajar y así, con Fortran, Cobol y aún Pascal, deberemos aceptar el uso real de esas variables y estructuras como globales, cualquiera sea la metodología de diseño que empleemos.

Paralelismo o simultaneidad de ocurrencia:

Esta propiedad se puede implementar con corrutinas, definiendo la sincronización entre ellas (Productor - Consumidor o varias corrutinas en interacción).
Puedo elegir tener algunos módulos a igual nivel y otros en relación jerárquica. Dispondré de un dispatcher diseñado por mi, con mi estrategia, que existirá sólo si lo necesito.

Podemos además hacer consideraciones sobre que la M O O presupone que:

- El sistema será sólo la suma e interconexión de sus objetos.
- Todos sus objetos son totalmente separables.

Conclusiones:

Las Metodologías Orientadas a Objetos
- No parecen un cambio significativo en cuanto a Diseño, respecto de los enfoques de Modularización encarados por Parnas, Constantine, Yourdon y seguidores y de Tipos Abstractos de Datos e implementados en lenguajes como Alphard, CLU y más prácticamente en Ada y Modula-2 y hasta en paquetes comerciales como Turbo Pascal 4.0
- Sí parecen ser un buen vehículo para imponer estos mismos conceptos, por el peso de difusión que han logrado los términos usados. En ésto es equivalente a lo que la terminología Programación Estructurada logró respecto a la disciplina de programación propuesta por N. Wirth.
- Establecen una disciplina a la que en otra forma no se lograba gran éxito de adhesión.

Las herramientas implementadas en los lenguajes procedurales por la corriente generada por las tendencias de adhesión a la moda de programación por objetos parecen proveernos de un herramental muy poderoso para llevar adelante los conceptos e ideas del Diseño Estructurado (o Composite Design según G. Myers).

Tomemos como recomendación el no "encandilarnos" con un "paradigma" deslumbrante. Vale más conocer a fondo los objetivos de nuestro diseño y las métricas que las metodologías disponibles nos proveen, y tomar de cada moda los elementos positivos que aporta y las herramientas que ese esfuerzo permitió implementar.

Conocer los objetivos del diseño implica, muchas veces, conocer el camino recorrido en el desarrollo de los conceptos y técnicas y las razones y méritos de su afirmación o desplazamiento en la búsqueda, a veces recurrente, de soluciones que se apoyan en las previas para construir nuestro herramental.

Bibliografía:

Atkinson M. P. y Morrison R. 1985. Procedures as persistent data objects. ACM TOPLAS 7, 4, Oct. 539-559.

Atkinson M. P. y Buneman P. O. 1987. Types and Persistence in Database programminglanguages. ACM Comp.Surv. 19,2 Jun.105-190.

Booch,G. 1986. Object-Oriented Development. IEEE Trans.Softw.Eng. SE-12, 2 (Feb.86), 211-221.

Dahl, O. y Nygaard, K. 1966. Simula, an Algol-based simulation language. Commun.ACM 9, 9 Sept. 671-678.

Goldberg, A. y Robson, D. 1983. Smalltalk-80: The language and its implementation. Addison Wesley, Reading, Mass. USA.

Goldstein, I. P. y Bobrow, D. G. 1980. Extending Object Oriented programming in Smalltalk. Proceedings of the 1980 Lisp Conference. Ago, ACM, 75-81.

Ichbiah, et al. 1979. Rationale of the design of the programming language Ada. En ACM SIGPLAN Not, 14, 6

Jacky J. P. y Kalet, I. R. 1987. An Object-Oriented Programming discipline for standard Pascal. Commun.ACM 30, 9 Sept. 772-776.

Jackson, M. A. 1975. Principles of program design. Academic Press, Londres.
Jackson, M. A. 1983. System development. Prentice-Hall. Londres.
Liskov, B., Synder, A., Atkinson, R., y Schiffert, C. 1977. Abstraction mechanisms in CLU. Commun.ACM 20, 8 Ago 564-576.
Liskov, B. 1981. CLU Reference manual, Goos and Hartmanis, Eds. Lecture notes in Computer Science, vol. 114. Springer-Verlag, Berlín.
Parnas, D. L. 1972. On the criteria to be used in decomposing systems into modules. Commun.ACM 15, 12 Dic. 1053-1058.
Soustrup, B. 1986. An overview of C++. SIGPLAN Not.21,10 Oct.7-18.
Yourdon, E.y Constantine, L. 1979..Structured Design. Fundamentals of a discipline of computer programs and systems design. Prentice-Hall,Englewood Cliffs, NJ. USA.
Wirth, N. 1971. Program development by stepwise refinement. Commun.ACM 14, 4. 221-227
Wirth, N. 1983. Programming in Modula-2. 2nda ed. Springer-Verlag, Berlín.
Wirth, N. 1988. Type extensions. ACM TOPLASS 10, 2 Abr. 204-214.