37

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

ESCUELA NACIONAL DE ESTUDIOS PROFESIONALES.

CAMPUS ARAGON

"LA ORIENTACIÒN A OBJETOS COMO TÉCNICA DE INGENIERÍA DEL SOFTWARE PARA LOGRAR LA REUTILIZACIÓN".

T E S I S

QUE PARA OBTENER EL TÍTULO DE

INGENIERO EN COMPUTACIÒN

P R E S E N T A :

ISRAEL SUMANO SALAZAR.

ASESOR:

LIC. ISRAEL A. JUÁREZ ORTEGA.

MEXICO

OCTUBRE, 2000

La habilidad es lo que permite hacer ciertas cosas. La motivación determina lo que se hace. La actitud determina cuán bien se hace.

– Lou Holtz

AGRACECIMIENTOS

Agradezco a Dios por ayudarme y permitir que terminara con esta parte de la misión.

Agradezco a mi familia por su apoyo, su comprensión y su respeto a mi individualidad y proyecto de vida.

Agradezco a EMG; JJJA y EAJA por su amistad, su confianza y su valioso apoyo en mis proyectos académicos.

Agradezco a todas aquellas personas que alguna vez hayan colaborado conmigo y, en cierta forma también a aquellas que hayan tratado de obstaculizarme, ya que finalmente todas colaboraron para acrecentar mi experiencia en las relaciones humanas.

Agradezco a todo el personal del I.M.P. y en especial a la †Unidad de Informática de la STI por colaborar en mi formación profesional y personal.

Mi sincero agradecimiento al Lic. Israel A. Juárez Ortega por su disposición y apoyo para desarrollar esta tesis y llevarla a buen termino.

Una mención especial para el Instituto Mexicano del Petróleo por el apoyo económico y técnico otorgado para la realización de esta tesis.

La orientación a objetos como técnica de ingeniería del software para lograr la reutilización

INTRODUCCIÓN

A medida que los precios del hardware para computadora disminuyen debido a las nuevas tecnologías de integración de circuitos y a la reducción en los costos de producción en masa, es posible construir sistemas de cómputo a un precio moderado y con un buen estándar de calidad.

Lo anterior ha provocado, sobre todo en la última década, la propagación acelerada de computadoras en todo el mundo y, como consecuencia, la incorporación de los sistemas de computación en casi cualquier actividad del quehacer humano.

El resultado final de este fenómeno es que en la actualidad las economías personal, empresarial, nacional e internacional dependen enormemente de las computadoras y sus sistemas de software.

Desafortunadamente, la industria del hardware computacional y la industria del software no se comportan a la par. Los costos de desarrollo y producción de los sistemas software no han disminuido, de hecho puede considerarse que han aumentado en razón de su complejidad.

Los problemas para la construcción de software son múltiples; un proyecto de un sistema software es muy similar a cualquier otro proyecto de ingeniería, administración, mercadotecnia, diseño o negocios; requiere de planeación, estrategias, organización, control y de un sistema de comprobación de resultados. Por tanto, es que hizo necesaria la creación de una rama especializada de la ingeniería dedicada explícitamente a hacer frente a los problemas que se presentan en la construcción de software a todos sus niveles: la ingeniería del software.

La ingeniería del software se yergue hoy como una disciplina legítima, con sus más de treinta años de existencia, ha tenido importantes logros y una evolución constante en sus modelos de proceso para la creación de productos. Aunque su base principal es la ciencia de la computación, la ingeniería del software tiene una naturaleza multidisciplinaria; utiliza también las matemáticas, la ergonomía, la psicología, las ciencias administrativas, la economía, etc. Esta amalgama de conocimientos se orienta hacia la búsqueda de métodos y aplicación de técnicas que coadyuven al proceso de análisis, diseño e implementación de software correcto, fiable y económico.

Sin embargo, a pesar de la existencia de una disciplina como la ingeniería de software, que ha alcanzado un importante grado de madurez y difusión alrededor del mundo, aún se construyeron sistemas software deficientes, no confiables, antieconómicos, inflexibles, escasamente documentados y poco atractivos para los usuarios. Esto se debe en gran medida a que los proyectos fueron encargados a profesionales con un escaso entrenamiento en la construcción de software, y en el peor de los casos a "programadores" o "analistas" que actúan sobre bases empíricas o criterios personales.

Afortunadamente, el ambiente de competencia provocado por la aparición de múltiples empresas dedicadas a la producción de software comercial, así como a la constante demanda de soluciones específicas para actividades como el dibujo, el diseño industrial, las finanzas y la investigación científica, han obligado a la especialización de profesionales para el área del software.

En la actualidad puede considerarse que muchos problemas de la ineficacia del software se han superado gracias a la profesionalización de los encargados de su creación y al establecimiento de estándares y normas de calidad que exigen cubrir determinadas especificaciones para la aprobación de un producto.

Pero a pesar de estos avances, debe reconocerse que la tarea de construir grandes sistemas software sigue siendo difícil; los equipos de desarrollo se enfrentan todavía a presiones de tiempo, costos no calculados y cambios de última hora, dificultades que han llegado a considerarse como inherentes al proceso de creación del software, pero que sin lugar a dudas producen efectos negativos en los costos y en la calidad del producto final.

Una de las causas que originan la persistencia de estas desavenencias es la aplicación incorrecta de un método para desarrollar el software, debido principalmente que se tiene una vaga comprensión de sus conceptos y a que se aplican parcial o deficientemente sus técnicas.

Este es el motivo por el cual se decide abordar el estudio de una técnica perteneciente al dominio de la ingeniería del software, dado que es de suma importancia adoptar una buena técnica que permita afrontar los problemas del desarrollo del software con éxito.

Hoy por hoy, existe una variedad de técnicas que han demostrado ser eficaces, pero hay una que en los últimos años ha tenido un fuerte impacto en el mundo del software, así como una gran influencia sobre desarrollo de aplicaciones comerciales: la orientación a objetos o método orientado a objetos.

La literatura existente alrededor de la orientación a objetos es muy basta, por lo que un estudio sobre el tema sin un enfoque específico resultaría simplemente en un texto mas, cuya aportación sería estéril. Con el afán de buscar mayor trascendencia, este trabajo enfocará las capacidades de la orientación a objetos a lograr una

condición que es necesaria para el progreso en el desarrollo del software: la reutilización.

La reutilización requiere de la construcción de componentes, considerando como tales los módulos de software que abarquen algunos de los patrones fundamentales dentro del desarrollo de software, dichos módulos se estructuran de forma tal que alcanzan un grado de generalización y flexibilidad que les permite ser incluidos en múltiples proyectos, cuyas aplicaciones resultantes no necesariamente tienen que ser afines.

El objetivo general de este trabajo es presentar a la orientación a objetos como una alternativa potente y versátil para lograr que los desarrollos de software resulten en soluciones fiables y reutilizables, atendiendo a los factores de la calidad del software así como a las exigencias de los criterios de modularidad.

Para lograr tal objetivo se han estructurado cuatro capítulos, cada uno busca tanto brindar aportaciones a su tema particular como contribuir en la consecución del objetivo general.

El capítulo I "La ingeniería del software y factores de calidad" pretende puntualizar los aspectos más relevantes de las definiciones de ingeniería del software, así como también precisar los factores involucrados en el concepto de calidad del software. Ambos conocimientos se consideran primordiales para encausar los esfuerzos del desarrollo de software hacia metas correctas y concretas.

El capítulo II "El camino a la orientación a objetos" hace énfasis en las propiedades que debiera incluir un método, lenguaje o herramienta para que pueda considerársele como "orientado a objetos"; se estudian los requisitos más importantes que constituyen el concepto de modularidad y, finalmente, se abordan algunas cuestiones clave que hay que considerar en la búsqueda de la reutilización.

El capítulo III "Técnicas orientadas a objetos" trata el conjunto de conceptos y técnicas del método orientado a objetos que tienen una implicación directa con la búsqueda de módulos de software reutilizables: los tipos abstractos de datos, las clases, la genericidad y la herencia. Se discutirán las propiedades y características que debieran tener las clases con base en la teoría de los tipos abstractos de datos (TAD); se continuará con la genericidad como un mecanismo para lograr un importante grado de adaptabilidad y finalmente se introducirá el concepto de herencia para facilitar la extensión y la especialización de las clases. En cada sección se puntualizan los beneficios que aporta el método a las exigencias de la modularidad, la calidad del software y la reutilización.

Finalmente, el capítulo IV "Metodología orientada a objetos: aplicación del método" muestra la forma de aplicar el potente conjunto de conceptos y técnicas estudiados en los capítulos precedentes; esto se lleva a cabo mediante el esbozo de ejemplos sencillos pero ilustrativos que resaltan las contribuciones del método para con el desarrollo de aplicaciones extensibles y adaptables, apegadas a los preceptos de la modularidad y los factores de calidad del software.

El trabajo procura establecer una secuencia lógica que permita percibir claramente los vínculos existentes entre estos capítulos. La información que contienen es producto de la investigación documental realizada alrededor de los trabajos de diversos autores dedicados al estudio de la ingeniería del software, el método orientado a objetos, la reutilización y la calidad del software; en algunos casos también de aquellos que han tratado de acoplar todos estos temas como una sola unidad.

Aunque muchas de las discusiones están orientadas básicamente al desarrollo y construcción de módulos de software no tiene la intención de ser una guía detallada del proceso de implementación de los mismos, es decir, no asocia el método de orientación a objetos con un lenguaje particular de programación. Sin embargo, reconociendo que difícilmente podría avanzarse en los temas sin recurrir en algún momento a una forma de código es que se introduce una notación sencilla, que tiene la virtud de ser fácilmente asimilable debido a su semejanza con PASCAL y ADA.

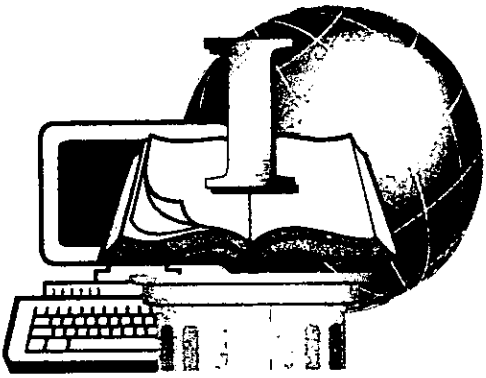
Las razones que subyacen en la decisión de omitir el uso de un lenguaje de programación específico (por no decir comercial) para el manejo de los ejemplos son:

- Evitar hacer compromisos con un lenguaje particular, el cual necesariamente tendría que defenderse frente a otras opciones, explicar sus restricciones, métodos, semántica y sintaxis. Esto obligaría a dedicar una parte importante del texto a cuestiones de programación, lo que potencialmente ocasionaría un trabajo de tamaño poco manejable.
- Continuando sobre esta misma idea, el trabajo se desviaría de su objetivo general, tendiendo peligrosamente a convertirse en un texto mas sobre programación orientada a objetos.
- Y finalmente la razón con una lógica más simple, es que el mundo de los lenguajes y las herramientas está en cambio y evolución constante, y una expresión válida hoy, es inconsistente mañana. Esto limitaría las expectativas de ampliación a futuro y la vida útil de este trabajo.

La orientación a objetos es una técnica por sí misma y, si bien es cierto que necesita de los medios que le permitan llevar su potencia al entorno de una computadora, no depende de un lenguaje o herramienta para existir.

El método orientado a objetos proyecta un amplio espectro conceptos y procedimientos para analizar de forma diferente los problemas del software. La tarea y responsabilidad como ingeniero o profesional del área del software es enfocar este espectro en un haz de soluciones verdaderamente prácticas que proporcionen respuestas satisfactorias a todas las cuestiones que surgen a lo largo del ciclo de vida del software.

La tarea de concentrar los conocimientos para que se traduzcan en opciones reales de solución implica necesariamente establecer criterios, discernir los temas útiles y formarse una idea preliminar de los alcances del estudio. Para el caso del presente, el criterio esencial fue la aplicabilidad práctica de aquellos conceptos, principios o técnicas de la orientación a objetos que contribuyen más claramente a allanar el camino hacia la reutilización del software. El conocer este criterio es importante tanto para ayudar a desvanecer falsas esperanzas acerca del contenido del trabajo como para *ajustar el rumbo* y no perderse en el mar de aspectos que abarca este o cualquier otro método si es que se desea continuar sobre esta misma línea de investigación.



La Ingeniería del Software y factores de calidad

1.1. La ingeniería del software y el arte en la construcción de software

La década de los años 90, especialmente los últimos años de la misma y por extensión el Tercer Milenio, se están caracterizando por el predominio de las *Tecnologías de Objetos* (TO) y su generalización más completa, las *Tecnologías de Componentes* (TC). Se puede ver una proliferación de libros, revistas, textos electrónicos, seminarios y congresos relativos a TO, así como también relativos a TC, tal es el caso de CORBA, COM, DCOM, ActiveX, etc. Los "objetos" así como su manifestación más practica "los componentes", están impregnando toda la industria de construcción de software.

La tecnología de objetos proporciona un marco de trabajo técnico para un modelo de proceso basado en componentes para la ingeniería del software. El método de orientación a objetos hace énfasis en la creación de módulos (clases) que encapsulan tanto los datos como los algoritmos que se utilizan para manejar dichos

datos. Si estos módulos se diseñan y estructuran correctamente, las clases resultantes son reutilizables para diferentes aplicaciones y arquitecturas de sistemas.

Sin embargo, para poder alcanzar la ambiciosa meta de construir software a partir de la combinación de componentes, es necesario asegurar que estos componentes sean correctos, fiables y prácticos; para lograrlo se debe empezar a construir sobre bases sólidas, es decir, con un sustento teórico que cimente el proceso de desarrollo para poder avanzar de manera ordenada y disciplinada. Este cimiento esta representado por el método de orientación a objetos. Dicho método se perfila como un potente recurso dentro de los actuales modelos de proceso de la ingeniería de software, acto seguido, se debe tener previo conocimiento de lo que es la ingeniería del software, su definición e implicaciones.

Definición del termino "Ingeniería del Software"

En pro de un completo y claro conocimiento del término "Ingeniería del software" se comenzará por definir primeramente que es ingeniería.

Según la Real Academia de Ciencias Exactas, Físicas y Naturales de España se define a ingeniería como: "*conjunto de conocimientos y técnicas cuya aplicación permite la utilización racional de los materiales y de los recursos naturales, mediante invenciones, construcciones u otras realizaciones provechosas para el hombre*".

Es evidente que la ingeniería de software no puede ser ajena a la definición anterior ya que reúne varias características de esta, sin embargo, para contar con una definición más precisa y enfocada a la industria del software se tiene que recurrir a definiciones expuestas por autores acreditados y estudiosos del tema, así como de organismos internacionales relacionados con la tecnología y las ciencias. De tal suerte se citan las siguientes definiciones:

1. Ingeniería del Software es la aplicación práctica del conocimiento científico en el diseño y construcción de programas de computadora y la documentación asociada requerida para desarrollar, operar (funcionar) y mantenerlos. Se conoce también como desarrollo de software o producción de software.[1]
2. Ingeniería del Software trata del establecimiento de los principios y métodos de la ingeniería a fin de obtener software de modo rentable que sea fiable y trabaje en máquinas reales.[2]
3. Ingeniería de Software es el estudio de los principios y metodologías para el desarrollo y mantenimiento de sistemas de software.[3]
4. La aplicación de un enfoque sistemático, disciplinado y cuantificable al desarrollo, operación (funcionamiento) y mantenimiento del software.[4]

---

[1] BOEHM, B. W.: *Software Engineering, IEEE Transactions on Computers*, C-25, num. 12., Diciembre, págs. 1226-1241.

[2] BAUER, F. L.: *Software Engineering, Information Processing, 71*, North Holland Publishing Co., Amsterdam, 1972.

[3] ZELKOVITZ, M. V., SHAW, A. C. y GANNON, J. D.: *Principles of Software Engineering and Design*. Prentice-Hall, Englewoods Clif, 1979.

[4] IEEE: *Standards Collection: Software Engineering*, IEEE Standard 610.12-1990, IEEE, 1993.

5. El estudio de métodos y herramientas que se pueden utilizar para producir software práctico de calidad.[5]

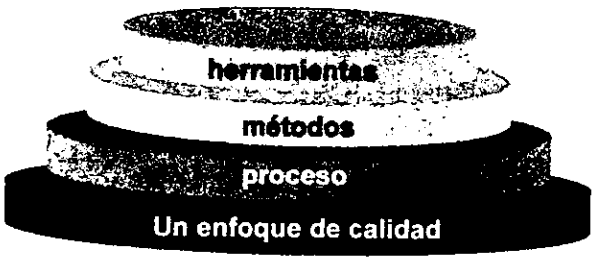
Se puede apreciar que existen propiedades en común entre estas definiciones y, que todas convergen en aspectos clave tales como:

- Aplicación del conocimiento científico (métodos, principios, disciplinas).
- Desarrollo ordenado y cuantificable.
- Creación de software y todo lo relacionado a su operación.
- Y un aspecto implícito (y necesario) en todos los procesos: **la calidad**.

El rol de la calidad y el énfasis en construir software práctico, son aspectos de orden prioritario en el desarrollo del software; sin importar que este sea comercial, científico o académico.

Se debe tener presente que en la actualidad la ingeniería de software es una disciplina multicapa[6], que tiene como base una orientación hacia la calidad, requisito necesario para que la ingeniería de software, como cualquier otra ingeniería, cuente con una cultura de continuas mejoras que conduzca a desarrollar enfoques más correctos, robustos y prácticos.

Contando con una base de calidad, se puede continuar a los siguientes estratos de la ingeniería de software: proceso, métodos y herramientas; cada uno con una importancia particular.



Capas de la ingeniería del software

La capa de proceso permite el desarrollo racional y oportuno de la ingeniería de software, es la responsable de mantener la unión de las capas de la tecnología. El proceso define un marco de trabajo para la gestión de los proyectos de software y establece el contexto en el que se aplican los métodos técnicos, se producen los resultados, se establecen los objetivos y se asegura la calidad.

---

[5] MEYER, B.: *Construcción de Software Orientado a Objetos*, Prólogo, XXXV, Prentice-Hall, Madrid, 1999

[6] PRESSMAN, R. S.: *Ingeniería del Software: Un enfoque práctico*, McGraw-Hill, 1997, pág. 18.

Los métodos de la ingeniería del software indican cómo construir técnicamente el software. Los métodos abarcan una gran gama de tareas que incluyen el análisis, diseño, construcción de programas, pruebas y mantenimiento. Los métodos de la ingeniería de software dependen de principios básicos que gobiernan cada área de la tecnología e incluyen actividades de modelado y técnicas descriptivas. Es dentro del marco de este estrato de la ingeniería del software donde se debe ubicar el presente trabajo.

Por ultimo, las herramientas de la ingeniería del software proporcionan un soporte automático o semiautomático para el proceso y para los métodos. En esta capa es posible integrar herramientas para la que información creada con una de estas herramientas pueda ser utilizada por otra, al grado de crear un sistema de soporte para el desarrollo del software asistido por computadora (*Computer-Aided Software Engineering CASE*).

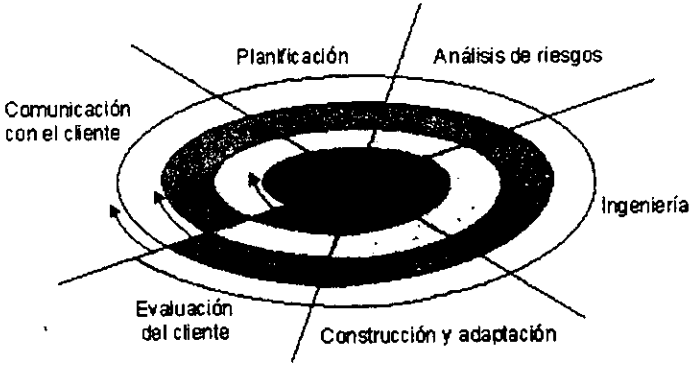
Para resolver problemas reales, en cualquier ámbito (empresarial, industrial, académico, etc.) los ingenieros deben incorporar una estrategia de desarrollo que acompañe a las capas de la ingeniería de software. Esta estrategia se conoce como modelo de proceso o paradigma de ingeniería de software. Dicho modelo de proceso se selecciona según la naturaleza de proyecto y de la aplicación, los métodos y las herramientas a utilizarse.

Dentro de los modelos de proceso más difundidos y aceptados se encuentra el modelo en espiral[7] (ver diagrama en la siguiente página), este modelo presenta un desarrollo evolutivo e interactivo del software con la construcción de prototipos que permiten hacer versiones incrementales del producto, que aumentan su complejidad conforme avanza la ejecución del modelo.

Este modelo en espiral se divide en actividades llamadas regiones de tareas. El modelo que se presenta contiene seis regiones:

- Comunicación con el cliente: involucra las tareas para establecer los canales de comunicación entre el desarrollador y el cliente.
- Planificación: tareas enfocadas a definir los recursos, el tiempo y demás información relacionada con el proyecto.
- Análisis de riesgos: tareas requeridas para evaluar riesgos técnicos y de gestión.
- Ingeniería: tareas requeridas para desarrollar una o más representaciones de la aplicación.
- Construcción y adaptación: tareas necesarias para construir, probar, instalar y proporcionar soporte al usuario.
- Evaluación del cliente: tareas requeridas para obtener las impresiones del cliente según la evaluación de las representaciones del software creadas durante la etapa de ingeniería e implementación.

[7] ibídem pág. 28.



Modelo en espiral típico

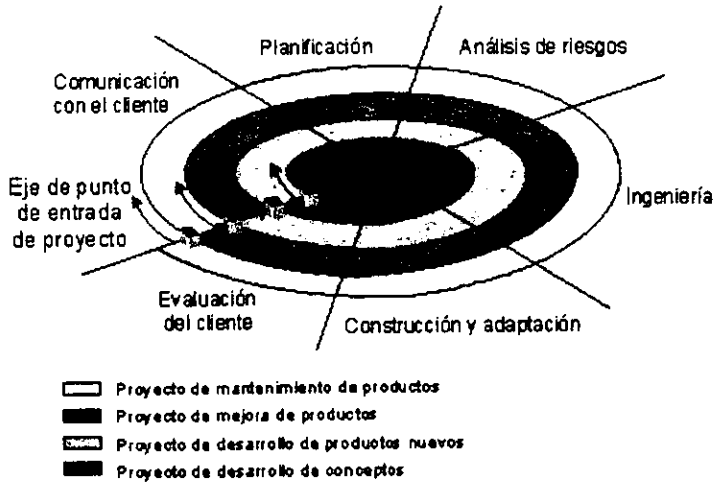
Cuando este proceso evolutivo comienza, los responsables de los proyectos arrancan a partir del centro y avanzan en sentido horario alrededor de la espiral. En el primer circuito de esta espiral se producen las especificaciones de los productos; los pasos siguientes son para desarrollar un prototipo y progresivamente crear versiones más completas del software. Cada región de la espiral tiene impacto en el producto final; cada paso por la planificación provoca ajustes en el curso del proyecto, coherentemente esta región es posterior a la comunicación con el cliente y a la evaluación del software. De esta forma se puede asegurar que las iteraciones produzcan un software cada vez más completo y acorde a las necesidades reales del usuario final.

Este modelo clásico de la espiral puede adaptarse y funcionar para todo el ciclo de vida del software. El modelo de la espiral adaptado para el ciclo de vida clásico completo[8] (ver diagrama en la siguiente página) define un eje de punto de entrada en el proyecto. Cada uno de los cubos situados a lo largo del eje representa un punto de arranque para un tipo diferente de proyecto. Así por ejemplo, para un proyecto de desarrollo de conceptos se comienza por el centro de la espiral y se avanza por todas las regiones hasta que se completa el concepto. Si el concepto se desarrolla para un producto real se procede al siguiente punto de entrada (proyecto de desarrollo de productos nuevos) el producto entonces evoluciona a través de iteraciones alrededor de la espiral hasta que alcance el siguiente punto.

Planteada de esta forma la espiral permanece operativa hasta que el software es retirado. Aun si el proceso permanece inactivo, esta propuesta nos proporciona un punto de entrada adecuado cuando se rehabilite el proceso o se requiera un cambio.

[8] ibídem pág. 29.

Modelo en espiral adaptado para el ciclo de vida clásico completo



El modelo de la espiral posee propiedades interesantes:

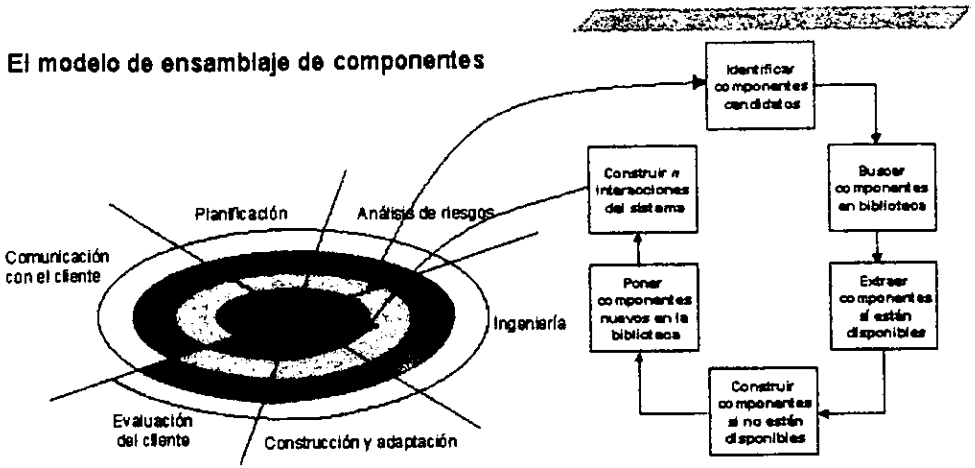
- Representa a la creación del software como un proceso evolutivo.
- Reconoce la interacción entre el cliente y el desarrollador para encontrar soluciones operativas.
- Proporciona la capacidad de reaccionar ante las necesidades y adecuaciones propuestas por los usuarios.
- El modelo en espiral propone el uso de prototipos como mecanismo de desarrollo gradual del software.
- Demanda considerar directamente los riesgos en todas las etapas del proyecto, antes de que estos evolucionen en problemas reales.
- Propone un marco de trabajo interactivo (cliente-desarrollador), de avance sistemático y acorde a un enfoque realista del desarrollo de sistemas.

Si bien este modelo es considerablemente completo y se presta a la adaptación, no es un enfoque único para encarar todos los proyectos de construcción del software. Pero resulta útil para ubicar (como en un mapa) al método de orientación a objetos dentro de la ingeniería del software, y más propiamente, dentro de los procesos de creación del software.

Un modelo que incorpora muchas de las características de la espiral y que tiene al método de orientación a objetos como piedra angular en la región de ingeniería es el modelo de ensamblaje de componentes[9] (ver diagrama en la siguiente página). Dicho modelo conserva prácticamente todas las características de la espiral clásica, su naturaleza evolutiva y enfoque interactivo para la creación del software. Sin embargo

[9] ibídem. pág.30.

para la creación de los prototipos se vale de componentes previamente creados. Para esto en la región de ingeniería se aplica el método orientado a objetos para comenzar a identificar las clases candidatas, mediante el examen de los datos a manejar y de los algoritmos que los modificarán; para con esto empaquetar ambos en una nueva clase.



Modelo de ensamblaje de componentes

Lo que hace realmente potente a este método es que las clases (componentes) creadas en proyectos anteriores son almacenadas en bibliotecas o depósitos; cuando se identifican las clases candidatas se buscan en estas bibliotecas para determinar si ya existen. En caso positivo las clases son extraídas de las bibliotecas y se vuelven a utilizar. Si una clase candidata no se halla en la biblioteca, se aplica el método de orientación a objetos para desarrollarla. Se compone así la primera interacción de la aplicación a construirse, mediante las clases extraídas de la biblioteca y las clases nuevas construidas para cumplir las necesidades únicas de la aplicación. El flujo del proceso vuelve a la espiral y volverá a introducir por último la iteración ensambladora de componentes a través de la actividad de ingeniería.

El modelo de ensamblaje de componentes por consiguiente lleva a la reutilización del software, y la reutilización proporciona beneficios a la ingeniería del software. Existen argumentos que avalan la eficiencia de la reutilización en los procesos de ingeniería del software. Como una muestra de ello se puede mencionar un estudio de reutilización realizado por QSM Associates y citado por Yourdon[10] en una de sus obras, donde se informa que el ensamblaje de componentes lleva a una reducción del 70 por ciento del tiempo de ciclo de desarrollo, un 84 por ciento del coste del proyecto y un índice de productividad del 26.2 en comparación con la norma de la industria que era del 16.9 en ese momento.

[10] YOURDON, E.: *Software Reuse, Application Development Strategies*, vol. VI, nº 12, diciembre 1994, págs.1-6.

1.2. Calidad del software

Como ya se mencionó, si en la ingeniería se busca la calidad; la ingeniería de software busca la producción de software de calidad. Por consiguiente antes estudiar las técnicas que permitan lograr mejoras significativas en los productos de software es necesario tener claro que la calidad del software es resultado de la combinación de varios factores.

Por un lado los usuarios de un producto de software determinan la calidad de un producto basados en las capacidades y facilidad de uso que les ofrezca a ellos como consumidores, es decir, en las cualidades visibles al momento de trabajar con el producto en cuestión. Por otro, la creación de un software de éxito exige adoptar una disciplina que marque las directrices que permitan el desarrollo ordenado del producto, con todo un aparato metodológico que asegure que hará lo que tiene que hacer y que además lo haga de la mejor forma posible.

1.2.1. Factores externos e internos de la calidad del software

Es deseable que los sistemas de software sean rápidos, fiables, fáciles de usar, legibles, modulares, estructurados y así sucesivamente. Sin embargo estos adjetivos describen dos tipos de cualidades diferentes.

Por un lado se consideran las cualidades tales como la velocidad y la facilidad de uso, cuya presencia o ausencia es detectada por los usuarios. Estas cualidades pueden ser denominadas como factores de calidad externos.

Otras cualidades aplicables a un producto de software, como la modularidad o legibilidad son los factores internos, perceptibles solo por los profesionales del área informática familiarizados con la estructura de un código fuente.

En ultima instancia lo que importa a los consumidores son los factores externos, ya que en la mayoría de los casos el software se adquiere por su velocidad, sus capacidades o facilidad de uso. Pero la clave para obtener los factores externos radica en la existencia de los internos, debido a que estas cualidades ocultas desarrolladas por los diseñadores e implementadores permiten que los usuarios disfruten de las cualidades visibles.

El estudio y aplicación de las técnicas modernas para lograr la calidad interna no son mas que los medios para alcanzar las cualidades externas. Por lo tanto hay que analizar factores de calidad externos para tener presente los objetivos que se desean alcanzar.

1.2.2. Revisión de los factores externos de la calidad del software

El conocimiento de los factores externos de calidad en un producto de software es determinante para saber hacia donde orientar los esfuerzos en pro de la creación de un software fiable y reutilizable. En la mayoría de las ocasiones se trabaja con un software sin percibir que cualidades reales nos ofrece y en que otras presenta deficiencias graves, esto en gran medida por el desconocimiento de las definiciones y conceptos que describen a los factores externos de la calidad o por un mal manejo de

los alcances de dichos conceptos. A saber, podemos identificar los siguientes factores externos.

- Corrección.
- Robustez.
- Extensibilidad.
- Reutilización.
- Compatibilidad.
- Eficiencia.
- Portabilidad.
- Facilidad de uso.
- Funcionalidad.
- Oportunidad.

Es importante contar con una definición clara y sencilla de cada uno a fin de saber con certeza sus implicaciones y alcances, sobre todo al momento de evaluar un producto de software o un trabajo propio.

CORRECIÓN (*Correctness*)

El más importante de los factores externos de la calidad y cuya obtención es tarea primordial en la construcción de cualquier producto de software es la corrección.

CORRECCIÓN

Corrección es la capacidad de los productos de software para realizar con exactitud sus tareas, tal y como se definen en las especificaciones.

La corrección es una cualidad principal. Si un sistema no hace lo que se supone que debe hacer, poco importan otras consideraciones que se hagan de él. El primer paso hacia la corrección es en sí difícil: se debe ser capaz de especificar los requisitos del sistema en una forma precisa, lo que implica una ardua tarea.

Los métodos que aseguran la corrección serán usualmente condicionales. Un sistema de software importante, incluso uno pequeño en los estándares actuales, implica a tantas áreas que sería imposible garantizar su corrección manejando todas las componentes y propiedades en un solo nivel. En cambio es necesario una solución multinivel, en que cada nivel confía en la corrección de los inferiores:

Capas en el desarrollo del software

En la solución condicional de la corrección, sólo hay que preocuparse en garantizar que cada nivel sea correcto bajo el supuesto de que los niveles inferiores son correctos. Esta es la única técnica realista, puesto que consigue una separación de áreas de interés y permite concentrarse en cada etapa en un conjunto limitado de problemas. No se puede comprobar de un modo provechoso que un programa escrito en un lenguaje de alto nivel X es correcto a menos que seamos capaces de asumir que el compilador que se este usando implementa correctamente a X.

Esto sin embargo, no implica que se deba confiar ciegamente en el compilador, sino que sencillamente se separan las dos componentes del problema: La corrección del compilador y la corrección del programa relativa a la semántica del lenguaje.

Para el enfoque de método orientado a objetos todavía intervienen más niveles: el desarrollo del software se basará en bibliotecas de componentes reutilizables, las cuales pueden utilizarse en muchas aplicaciones diferentes.

La solución condicional también se aplicará aquí: hay que asegurarse de que las bibliotecas sean correctas y, de forma separada, que la aplicación es correcta asumiendo que las bibliotecas lo son.

Niveles de un proceso de desarrollo que incluye reutilización

Muchos programadores cuando se les presenta el problema de la corrección del software, piensan en la prueba y depuración. Pero se puede ser más ambicioso y construir un software correcto desde el principio, aplicando en su momento las técnicas propias del método orientado a objetos tales como los tipos de datos abstractos y las aserciones, en lugar de llegar al final y depurar el producto para hacerlo correcto. Sin embargo no se trata de desvirtuar a la depuración y la prueba, solo se sugiere que ahora colaboren como medio de doble comprobación del resultado.

ROBUSTEZ (*Robustness*)

ROBUSTEZ

Robustez es la capacidad de los sistemas de software de reaccionar apropiadamente ante condiciones excepcionales.

La robustez complementa la corrección. La corrección tiene que ver con el comportamiento de un sistema en casos previstos por su especificación; la robustez lo que sucede fuera de tal especificación.

Como se refleja en la definición, la robustez es por naturaleza una noción más difusa que la corrección. Puesto que tiene que ver aquí con los casos no previstos por la especificación, no es posible decir, como con la corrección, que el sistema debería "realizar sus tareas" en tal caso; donde las tareas son conocidas, el caso excepcional formaría parte de la especificación y regresaríamos al terreno de la corrección.

Robustez versus corrección

Las nociones de normal y excepcional son siempre relativas a cierta especificación: un caso excepcional es simplemente un caso no previsto por la especificación. Si se amplía la especificación los casos excepcionales se convierten en normales – incluso si corresponden a eventos tales como la entrada de datos errónea por parte del usuario que sería preferible que no ocurriesen. "Normal" en este sentido no significa "deseable" sino simplemente "planificado durante el diseño del software". Aunque en un principio pueda considerarse paradójico que una entrada errónea sea llamado un caso normal, cualquier otro planteamiento tendría que basarse en un criterio subjetivo y por tanto sería poco útil.

Siempre habrá casos que la especificación no contemple explícitamente. El papel del requisito de robustez es asegurar que si tal caso surgiese el sistema no causará eventos catastróficos; debería producir mensajes de error apropiados, terminar su ejecución limpiamente o entrar en el llamado "*modo de degradación elegante*".

EXTENSIBILIDAD (*Extendibility*)

EXTENSIBILIDAD

Extensibilidad es la facilidad de adaptar los productos de software a los cambios de especificación.

El software se supone que es *soft* (blando), y realmente lo es en un principio; nada es más fácil de cambiar que un programa si se tiene acceso al código fuente. Solo hay que usar las herramientas adecuadas (editor de textos, compilador, etc.).

El problema de la extensibilidad es un problema de escala. Para programas pequeños realizar cambios no es normalmente una tarea difícil; pero a medida que el software crece comienza a ser cada vez más difícil de adaptar. A menudo un gran sistema de software es visto por las personas encargadas de su mantenimiento como un castillo gigante de naipes en el que sacar un elemento puede causar que todo el edificio se derrumbe.

La extensibilidad es necesaria porque en la base de todo software encontramos algún fenómeno humano y de ahí su volatilidad.

Las técnicas tradicionales de software no tienen suficientemente en cuenta el cambio y se basan en una visión ideal del ciclo de vida del software donde en la etapa inicial de análisis se congelan los requisitos y el resto del proceso se dedica al diseño y a la construcción de una solución.

Esto es comprensible: en la evolución de la disciplina la primera tarea fue desarrollar buenas técnicas para establecer y resolver problemas particulares, sin preocuparse si el problema cambia en un futuro, se comienza a resolverlo. Pero ahora que las técnicas básicas de ingeniería de software están en su verdadero sitio se ha convertido en esencial reconocer y señalar este aspecto central.

El cambio es omnipresente en el desarrollo del software: cambios de los requisitos, de nuestra compresión de los requisitos, de los algoritmos, de la representación de los datos, de las técnicas de implementación. Ofrecer soporte para los cambios es un objetivo básico de la tecnología de objetos.

Aunque muchas de las técnicas que mejoran la extensibilidad se pueden explicar con pequeños ejemplos, su relevancia sólo se ve con claridad en los grandes proyectos. Hay dos principios esenciales para mejorar la extensibilidad:

- Simplicidad del diseño: una arquitectura simple siempre será más fácil, de adaptar a los cambios que una compleja.
- Descentralización: cuanto más autónomos sean los módulos, más alta es la probabilidad de que un cambio simple afecte solo a un módulo, o a un número pequeño de módulos, en lugar de provocar una reacción en cadena de cambios en el sistema completo.

El método orientado a objetos es, antes que cualquier otra cosa, un método de arquitectura de sistemas que ayuda a los diseñadores a producir sistemas cuya estructura es a un mismo tiempo simple (incluso para grandes sistemas) y descentralizada.

REUTILIZACIÓN (*Reusability*)

REUTILIZACIÓN

Reutilización es la capacidad de los elementos de software de servir para la construcción de muchas aplicaciones diferentes.

La necesidad de la reutilización surge de la observación de que los sistemas de software a menudo siguen patrones similares; debería de ser posible explotar esta similitud y evitar reinventar soluciones a problemas que ya han sido encontradas con anterioridad. Capturando tal patrón, un elemento de software reutilizable se podrá aplicar a muchos desarrollos diferentes.

La reutilización tiene una influencia sobre todos los demás aspectos de la calidad del software, ya que al resolver el problema de la reutilización se tendrá que escribir menos software y en consecuencia se podrán dedicar mayores esfuerzos (por el mismo costo total) a mejorar otros factores tales como la corrección y la robustez.

Aquí se tiene un nuevo aspecto que la visión tradicional del ciclo de vida del software no ha reconocido adecuadamente y por la misma razón histórica: deben encontrar primero las formas de resolver un problema antes de preocuparse por aplicar la solución a otros problemas. Pero con el crecimiento del software y sus perspectivas de convertirse en una verdadera industria, la necesidad de la reutilización se ha convertido en un asunto apremiante.

COMPATIBILIDAD (*Compatibility*)

COMPATIBILIDAD

Compatibilidad es la facilidad de combinar unos elementos de software con otros.

La compatibilidad es importante debido a que los sistemas software no se desarrollan en el vacío: necesitan interactuar con otros. Pero con mucha frecuencia los sistemas tienen dificultades porque hacen suposiciones contradictorias sobre el resto del mundo. Un ejemplo es la amplia variedad de formatos de archivos soportados por muchos sistemas operativos. Un programa puede usar directamente como entrada los resultados de otros si los formatos de archivos son compatibles.

La clave de la compatibilidad recae en la homogeneidad del diseño y en acordar convenciones estándares para la comunicación entre programas. Los enfoques incluyen:

- Formatos de archivo estándares. Como en el sistema operativo UNIX, donde cualquier archivo de texto es simplemente una secuencia de caracteres.
- Estructura de datos estándares. Como en los sistemas Lisp, donde tanto los datos como los programas, se representan mediante árboles binarios (llamados listas en Lisp).
- Interfaces de usuario estándares. Como las diferentes versiones de Windows, OS/2 y MacOS donde todas las herramientas utilizan un solo paradigma para la

comunicación con el usuario, basado en componentes estándares tales como ventanas, iconos, menús, etc.

Se han definido soluciones más generales definiendo protocolos estándares de acceso a todas las entidades manipuladas por el software. Ésta es la idea que esta detrás de los tipos de datos abstractos y del enfoque orientado a objetos, así como de los denominados protocolos middleware como CORBA y Microsoft OLE-COM (ActiveX).

EFICIENCIA (*Efficiency*)

EFICIENCIA

Eficiencia es la capacidad de un sistema para exigir la menor cantidad posible de recursos hardware, tales como tiempo del procesador, espacio ocupado en memoria interna y externa o ancho de banda utilizado en los dispositivos de comunicación.

Casi sinónimo de eficiencia es la palabra "rendimiento". La comunidad inmersa en el problema del software muestra dos tipos de actitud con relación a la eficiencia:

- Algunos desarrolladores tienen una obsesión con las cuestiones de rendimiento y le dedican una gran cantidad de esfuerzos a presuntas optimizaciones.
- Por otro lado, existe la tendencia de soslayar las cuestiones de eficiencia, como se evidencia en las frases de la industria "hágalo correcto antes de hacerlo rápido" y finalmente "los modelos de computadoras del año que viene van a ser más rápidos".

No es extraño ver a la misma persona mostrando estas dos actitudes en momentos diferentes. Esta claro que los desarrolladores a menudo muestran una preocupación exagerada con la micro-optimización. Como ya se señalo, la eficiencia no se preocupa mucho si el software no es correcto, lo que sugiere una nueva actitud "*no importa cuán rápido es hasta que no esté correcto*". De manera más general, la preocupación de la eficiencia debe sopesarse con otros objetivos tales como la extensibilidad y la reutilización; optimizaciones extremas pueden hacer al software tan especializado que limite el cambio y la reutilización. Es más, la potencia creciente del hardware nos permite tener una actitud más relajada con respecto a tratar de ganar hasta el último byte o microsegundo.

Sin embargo esto no disminuye la importancia de la eficiencia. A nadie le gusta esperar para obtener respuestas de un sistema interactivo o de tener que comprar más memoria para poder ejecutar un programa. En realidad las actitudes muy despreocupadas con respecto al rendimiento incluyen muchas posturas; si un sistema al final es tan lento o pesado que impide su utilización, todos aquellos que declaran que "la velocidad no es importante" no serán precisamente los últimos en quejarse.

En la actualidad, la construcción de software es difícil precisamente porque se requiere tomar en cuenta muchos requisitos diferentes, algunos de los cuales como la corrección, son abstractos y conceptuales, mientras que otros como la eficiencia, son concretos y acotados por las propiedades del hardware.

Para algunos del lado de las ciencias, el desarrollo de software es una rama de las matemáticas; para algunos ingenieros, es una rama de la tecnología aplicada. En realidad puede considerarse que son ambas cosas. El desarrollador de software debe reconciliar los conceptos abstractos con sus implementaciones concretas, la matemática de la computación correcta con las restricciones de tiempo y espacio que se derivan de las leyes físicas y limitaciones de la tecnología actual de hardware. Esta necesidad de reconciliar ambas cuestiones, es el desafío central de la ingeniería de software.

El constante aumento de potencia en las computadoras, siendo impresionante, no es excusa para infravalorar la eficiencia, al menos por tres razones.

1. El que compra una computadora más grande y más rápida desea ver algunos beneficios de esta potencia extra – como poder tratar nuevos problemas, procesar problemas anteriores más rápidamente o procesar versiones mayores de los problemas anteriores en la misma cantidad de tiempo.
2. Uno de los factores más visibles de los avances en la potencia de las computadoras esta en incrementar la ventaja de los buenos algoritmos sobre los malos. Supongamos que una nueva máquina es dos veces más rápida que la anterior. Sea n el tamaño del problema a resolver y N el máximo n que puede manejar un cierto algoritmo en un tiempo dado. Entonces si el algoritmo es $O(n)$, es decir, se ejecuta en un tiempo proporcional a n, la nueva máquina debe ser capaz de tratar problemas de tamaño $2*n$. Para un algoritmo que se ejecute en un tiempo $O(n^2)$ la nueva máquina sólo permitirá un aumento del 41% en el tamaño de N. Un algoritmo que sea $O(2^n)$, como algunos algoritmos de búsquedas exhaustivas, sólo añadirán uno al N – lo que no es mucho aumento por el dinero invertido.
3. En algunos casos la eficiencia puede afectar a la corrección. Una especificación puede establecer que la respuesta de la computadora a un cierto evento debe ocurrir antes de cierta cantidad de tiempo especificada; por ejemplo, una computadora de vuelo debe estar preparada para detectar y procesar el mensaje de un sensor de una válvula lo suficientemente rápido para llevar a cabo una acción correctiva. Esta conexión entre eficiencia y corrección no esta restringida solamente a las aplicaciones conocidas como de "tiempo real"; pocas personas estarían interesadas en un modelo de predicción climática que tarde 24 horas en pronosticar el tiempo del día siguiente.

Debido a que este trabajo se centra en los conceptos de la ingeniería de software orientado a objetos y, no en aspectos de implementación, solamente en unos pocos apartados se tratan explícitamente el coste de rendimiento asociados. Pero el asunto de la eficiencia estará implícito a lo largo del trabajo. Se pretende que siempre que se presente una solución orientada a objetos a algún problema, se pueda estar seguro que la solución será no solo elegante sino también eficiente; siempre que se introduce un nuevo mecanismo de O-O, ya sea la *recolección de basura* (y otras técnicas de gestión de memoria en los sistemas para la computación orientada a objetos), ligadura dinámica, generacidad o herencia, estará basada en el conocimiento

de que dicho mecanismo se puede implementar a un coste razonable en tiempo y espacio; siempre que sea apropiado se mencionarán las consecuencias en el rendimiento de las técnicas estudiadas.

La eficiencia es solo uno de los factores de la calidad; no deberíamos permitirle (como hacen algunos dentro de la profesión) gobernar la vida de los ingenieros. Pero es un factor que debe tomarse en cuenta, lo mismo en la construcción de un sistema software que en el diseño de un lenguaje de programación. Si usted descarta el rendimiento, el rendimiento lo descartará a usted.

PORTABILIDAD ó TRANSPORTABILIDAD (*Portability*)

PORTABILIDAD

Portabilidad (transportabilidad) es la facilidad de transferir los productos software a diferentes entornos hardware y software.

La portabilidad tiene que ver con las variaciones no sólo del hardware físico sino más generalmente de la máquina hardware-software, la que realmente programamos y que incluye un sistema operativo, el sistema de ventanas (sí se emplea) y demás herramientas fundamentales. En este trabajo se usa la palabra "plataforma" para denotar un tipo de máquina hardware-software. Un ejemplo de plataforma es "Intel X86 con Windows NT (conocida por algunos como "Wintel").

Es importante saber que muchas de las incompatibilidades existentes entre plataformas son injustificadas, y para un simple observador a veces la única explicación parece ser una conspiración para martirizar a la humanidad en general y a los programadores en particular. Sin embargo, independientemente de las causas, esta diversidad convierte la portabilidad en un asunto primordial tanto para los que desarrollan como para los que usan el software.

FACILIDAD DE USO (*Ease of use*)

FACILIDAD DE USO

Facilidad de uso es la propiedad que permite a las personas con diferentes formaciones y aptitudes aprender a usar los productos software y aplicarlos a la resolución de problemas. También cubre la facilidad de instalación, de operación y de supervisión.

La definición insiste en los diferentes niveles de experiencia de los posibles usuarios. Este requisito plantea uno de los mayores retos para los diseñadores de software preocupados por la facilidad de uso: como proporcionar explicaciones y guías detalladas a los usuarios novatos sin fastidiar a los usuarios expertos que quieren ir directos al grano.

Una de las claves de la facilidad de uso es la simplicidad estructural. Un sistema bien diseñado, construido de acuerdo a una estructura clara y bien pensada, tiende a ser más fácil de aprender y usar que uno confuso. Por supuesto, la condición no es suficiente (lo que es simple y claro para el diseñador puede ser difícil y oscuro

para los usuarios, especialmente si se expone en términos de diseño en lugar de en términos comprensibles para el usuario), pero ayuda considerablemente.

Los diseñadores de software preocupados por la facilidad de uso consideran con recelo un precepto frecuentemente usado en la literatura sobre interfaces de usuario: *conocer al usuario*. El argumento radica en que un buen diseñador, debe de hacer un esfuerzo inicial por comprender a la comunidad de usuarios a la que se destina el sistema. Este punto de vista ignora una de las características de los sistemas que gozan de mayor éxito: que siempre sobrepasan a la audiencia inicial. Dos ejemplos útiles para ilustrar el punto anterior son los casos de Fortran, concebido inicialmente como una herramienta para resolver el problema de una pequeña comunidad de ingenieros y científicos de la programación de la IBM 704, y UNIX concebido para usos interno de los laboratorios Bell(*).

Un sistema diseñado para un grupo especifico se basará en supuestos que simplemente no se cumplen para una gran audiencia. Los buenos diseñadores de interfaces siguen una política más prudente. Hacen las menos suposiciones posibles sobre los usuarios. Cuando se diseña un sistema interactivo, se debe de esperar que los usuarios sean miembros de la raza humana y que sepan leer, mover un ratón, presionar un botón y teclear; no más allá. Aun si el software esta dirigido a un área especializada de aplicación, y se puede dar por hecho que los usuarios están familiarizados con sus conceptos básicos, trabajar bajo los preceptos de la facilidad de uso siempre es más seguro.

FUNCIONALIDAD (*Functionality*)

FUNCIONALIDAD

Funcionalidad es el conjunto de posibilidades que proporciona un sistema.

Uno de los problemas más difíciles a los que se enfrenta un jefe de proyecto es conocer cuanta funcionalidad es suficiente. La presión para ofrecer más facilidades, conocida como *featurism*(**), esta constantemente presente. Sus consecuencias son malas para los proyectos internos, donde las presiones vienen de los usuarios de la misma compañía, y son peores para los productos comerciales, ya que la parte más destacada de los análisis comparativos suele ser una tabla donde se enumera una por una las propiedades que ofrecen los distintos productos analizados.

El *featurism* es realmente la combinación de dos problemas, uno más difícil que el otro. El problema más fácil es la perdida de consistencia como consecuencia de estar añadiendo nuevas propiedades, lo que puede afectar a la facilidad de uso. Los usuarios se quejan con razón que toda la parafernalia que acompaña a una nueva versión de un producto lo hace tremendamente complejo. Pero es común que a unos usuarios les puedan resultar superfluas las nuevas facilidades y a otros increíblemente útiles y necesarias; finalmente si surgen estas nuevas propiedades, no es más que por solicitud y en algunos casos exigencia de los mismos usuarios.

(*) BISHOP, P.: *Conceptos de Informática, Ejemplos de lenguajes de alto nivel*, Anaya Multimedia, Madrid, págs. 251-282.

(**) El término en ingles es *featurism* que significa "muchos rasgos distintos".

para los usuarios, especialmente si se expone en términos de diseño en lugar de en términos comprensibles para el usuario), pero ayuda considerablemente.

Los diseñadores de software preocupados por la facilidad de uso consideran con recelo un precepto frecuentemente usado en la literatura sobre interfaces de usuario: *conocer al usuario*. El argumento radica en que un buen diseñador, debe de hacer un esfuerzo inicial por comprender a la comunidad de usuarios a la que se destina el sistema. Este punto de vista ignora una de las características de los sistemas que gozan de mayor éxito: que siempre sobrepasan a la audiencia inicial. Dos ejemplos útiles para ilustrar el punto anterior son los casos de Fortran, concebido inicialmente como una herramienta para resolver el problema de una pequeña comunidad de ingenieros y científicos de la programación de la IBM 704, y UNIX concebido para usos interno de los laboratorios Bell(*).

Un sistema diseñado para un grupo específico se basará en supuestos que simplemente no se cumplen para una gran audiencia. Los buenos diseñadores de interfaces siguen una política más prudente. Hacen las menos suposiciones posibles sobre los usuarios. Cuando se diseña un sistema interactivo, se debe de esperar que los usuarios sean miembros de la raza humana y que sepan leer, mover un ratón, presionar un botón y teclear; no más allá. Aun si el software esta dirigido a un área especializada de aplicación, y se puede dar por hecho que los usuarios están familiarizados con sus conceptos básicos, trabajar bajo los preceptos de la facilidad de uso siempre es más seguro.

FUNCIONALIDAD (*Functionality*)

FUNCIONALIDAD

Funcionalidad es el conjunto de posibilidades que proporciona un sistema.

Uno de los problemas más difíciles a los que se enfrenta un jefe de proyecto es conocer cuanta funcionalidad es suficiente. La presión para ofrecer más facilidades, conocida como *featurism*(**), esta constantemente presente. Sus consecuencias son malas para los proyectos internos, donde las presiones vienen de los usuarios de la misma compañía, y son peores para los productos comerciales, ya que la parte más destacada de los análisis comparativos suele ser una tabla donde se enumera una por una las propiedades que ofrecen los distintos productos analizados.

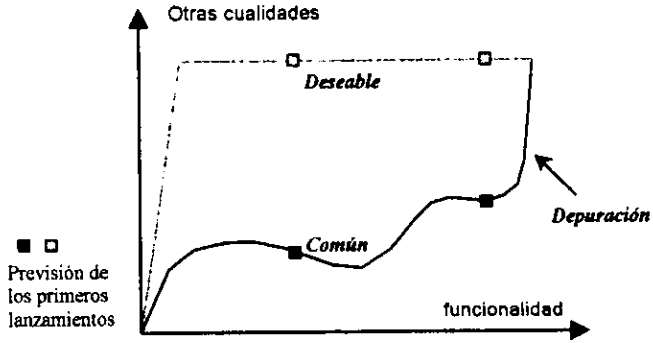
El *featurism* es realmente la combinación de dos problemas, uno más difícil que el otro. El problema más fácil es la perdida de consistencia como consecuencia de estar añadiendo nuevas propiedades, lo que puede afectar a la facilidad de uso. Los usuarios se quejan con razón que toda la parafernalia que acompaña a una nueva versión de un producto lo hace tremendamente complejo. Pero es común que a unos usuarios les puedan resultar superfluas las nuevas facilidades y a otros increíblemente útiles y necesarias; finalmente si surgen estas nuevas propiedades, no es más que por solicitud y en algunos casos exigencia de los mismos usuarios.

(*) BISHOP, P.: *Conceptos de Informática. Ejemplos de lenguajes de alto nivel*, Anaya Multimedia, Madrid, págs. 251-282.

(**) El término en ingles es *featurism* que significa "muchos rasgos distintos".

La solución para el problema descrito anteriormente es trabajar una y otra vez sobre la consistencia del producto global, tratando de que todo encaje en un molde general. Un buen producto de software esta basado en un número pequeño de potentes ideas; incluso si se tienen muchas propiedades especializadas, éstas deberían explicarse como consecuencia de los conceptos básicos.

Por otro lado, el problema más difícil al encarar la funcionalidad, es evitar centrar tanto la atención en las propiedades hasta el punto de olvidar otras cualidades. Es común que los proyectos cometan tal error. Esta situación es expresada gráficamente por Roger Osmond mediante los dos caminos posibles para completar un proyecto, la representación es conocida como Curvas de Osmond[11]:



Curvas de Osmond

La curva inferior es bastante común: en la frenética carrera por añadir más propiedades, el desarrollo se sale de la pista de la calidad global. La fase final que intenta que las cosas queden bien, puede ser larga y estresante. Si bajo la presión de los usuarios o los competidores usted está forzado a lanzar el producto prematuramente – en las etapas marcadas en la figura con cuadros en negro – el resultado puede ser perjudicial para su reputación.

Lo que Osmond sugiere (la curva superior) es, ayudado por las técnicas de desarrollo O-O que aumentan la calidad, mantener constante el nivel de calidad a lo largo del proyecto para todos los aspectos incluyendo la funcionalidad. No se debe comprometer la fiabilidad, la extensibilidad o los demás factores: hay que rechazar el pasar a considerar nuevas propiedades hasta que no se este satisfecho con las que se tienen.

Este método es difícil de aplicar en el día con día debido a las presiones mencionadas, pero nos conduce a un proceso de software más efectivo y a menudo hacia un mejor producto final. Incluso si el resultado es el mismo, como se asume en la figura, este debe alcanzarse antes (aunque la figura no muestra el tiempo). Seguir el camino sugerido significa también que la decisión de lanzar una versión inicial – en uno

[11] OSMOND, R. F.: *Essential of Successful O-O Project Management: Designing High Performance Projects*, TOOLS USA 95.

de los puntos marcados por los dos cuadros superiores de la figura – será, sino más fácil, al menos más simple: estará basado en valorar si el producto cubre un área lo suficientemente amplia del conjunto completo de características para atraer a los consumidores previstos en lugar de alejarlos. La cuestión "¿es lo suficientemente bueno?" (al igual que "el sistema no fallara") no debería ser un factor.

Pero en la práctica, es más fácil estar de acuerdo con estos conceptos que aplicarlos. Sin embargo todo proyecto debería seguir el enfoque representado por la mejor de las dos curvas de Osmond.

OPORTUNIDAD (*Timeliness*)

OPORTUNIDAD

Oportunidad es la capacidad de un sistema de software de ser lanzado cuando los usuarios lo desean, o antes.

La oportunidad es una de las mayores frustraciones de la industria del software. Un gran producto software que aparece demasiado tarde puede no alcanzar su objetivo. Esto es cierto en otras industrias también, pero pocas evolucionan tan rápidamente como el software.

La oportunidad es todavía, para grandes proyectos, un fenómeno poco común. Cuando Microsoft anunció que la última versión de su principal sistema operativo, que llevaba varios años realizando, saldría al mercado un mes antes de lo previsto, el suceso fue lo suficientemente relevante para encabezar los titulares de revistas especializadas en el ambiente computacional.

1.2.3. Otras cualidades

Además de las cualidades analizadas anteriormente, existen otras que afectan a los usuarios de sistemas software y a la gente que compra estos sistemas o encarga su desarrollo. En particular:

- **Verificabilidad.** Es la facilidad para preparar procedimientos de aceptación, especialmente datos de prueba y procedimientos para detectar fallos y localizar errores durante las fases de validación y operación.
- **Integridad.** Es la capacidad de los sistemas software de proteger sus diversos componentes (programas, datos, etc.) contra modificaciones y accesos no autorizados.
- **Reparabilidad.** Es la capacidad para facilitar la reparación de los defectos.
- **Economía**, junto con la oportunidad. Es la capacidad que un sistema tiene de completarse con el presupuesto asignado o por debajo del mismo.

1.2.4. Sobre la documentación

En una lista de los factores de calidad del software, uno podría esperar encontrar la presencia de una buena documentación como uno de los requisitos. Pero este no es un factor de calidad separado, la necesidad de documentación es una consecuencia de otros factores de calidad vistos anteriormente. Se pueden distinguir tres tipos de documentación:

- La necesidad de documentación externa, que permite a los usuarios conocer la potencia de un sistema y usarlo convenientemente, es una consecuencia de la definición de la facilidad de uso.
- La necesidad de documentación interna, que permite a los desarrolladores de software comprender la estructura e implementación de un sistema, es una consecuencia del requisito de extensibilidad.
- La necesidad de documentación de la interfaz de un módulo, que permite a los desarrolladores de software comprender las funciones proporcionadas por un módulo sin tener que comprender su implementación, es una consecuencia del requisito de reutilización. También se desprende de la extensibilidad, ya que una documentación de la interfaz de un módulo permite determinar cuando cierto cambio necesario afecta a un determinado módulo.

En lugar de tratar la documentación como un producto propio del software, es preferible producir software lo más autodocumentado posible. Esto se aplica a los tres tipos de documentación:

- Incluyendo facilidades de "ayuda" en línea y siguiendo normas para interfaces claras y consistentes. Se alivia la tarea de los autores de los manuales de usuario y de otras formas de documentación externa.
- Un buen lenguaje de implementación puede eliminar muchas de las necesidades de documentación interna si favorece la claridad y la estructura. Este debe ser uno de los requisitos principales que se busque al momento de elegir un lenguaje o cualquier tipo de notación usada para representar o programar un algoritmo.
- La notación soportará la ocultación de la información y otras técnicas (tales como las aserciones) para separar la interfaz de los módulos de su implementación. Será posible entonces utilizar herramientas para producir automáticamente documentación de la interfaz del módulo a partir del texto de los módulos.

Todas estas técnicas reducen la importancia de la documentación tradicional, aunque no debe esperarse que la eliminen completamente.

1.2.5. Compromisos (*Tradeoffs*)(*)

En esta revisión de los factorés externos de calidad del software se han descrito requisitos que pueden entran en conflicto unos con otros.

Es valido preguntar ¿cómo se puede tener integridad sin introducir protecciones de varias clases, lo cual inevitablemente perjudica a la facilidad de uso? La economía, a menudo va contra la funcionalidad. La eficiencia óptima requerirá una adaptación perfecta a un entorno particular de hardware y software, lo cual es contrario a la portabilidad. Una perfecta adaptación a una especificación puede ir en contra de la reutilización, que incita a resolver problemas más generales que el planteado inicialmente. Las presiones de la oportunidad puede tentar a los responsables de un proyecto a utilizar técnicas de Desarrollo Rápido de Aplicaciones (Rapid Application Development, RAD), cuyos resultados pueden no estar acorde con la extensibilidad.

Aunque en muchos casos es posible encontrar una solución que reconcilie factores aparentemente en conflicto, a veces es necesario tomar soluciones de compromiso. Con bastante frecuencia, los desarrolladores realizan estas decisiones implícitamente, sin tiempo para analizar las características involucradas y las diversas opciones disponibles; la eficiencia tiende a ser un factor dominante en este tipo de decisiones silenciosas. Un verdadero enfoque de ingeniería de software implica un esfuerzo por establecer los criterios con claridad y tomar las decisiones conscientemente.

Si es necesario decidirse por algún factor de calidad, uno sobresale del resto: **la corrección**. Nunca hay justificación para comprometer la corrección en aras de otras cuestiones tales como la eficiencia. Si el software no lleva a cabo su función el resto es inútil.

1.2.6. Cuestiones clave

Todas las cualidades antes expuestas son importantes. Pero en el estado actual de la industria del software sobresalen cuatro:

- Corrección y robustez: es bastante difícil producir software sin defectos (*bugs*) y muy difícil corregir los defectos una vez que están ahí. Las técnicas para mejorar la corrección y la robustez son similares: Enfoques más sistemáticos para la construcción de software; más especificaciones formales; más comprobaciones integradas a lo largo del proceso de construcción del software (no sólo la prueba y la depuración después de hecho); mejores mecanismos en los lenguajes tales como la comprobación estática de tipos, las aserciones, la gestión automática de memoria y el tratamiento disciplinado de excepciones, permiten a los desarrolladores asegurar los requisitos de corrección y de robustez y posibilitan herramientas para detectar inconsistencias antes de que se conviertan en defectos. Debido al parecido entre corrección y la robustez, es conveniente usar un término más general para cubrir ambos factores: **fiabilidad**.

(*) El término inglés "*tradeoffs*" hace referencia a aquellas situaciones en las que al solucionar un problema, entran en conflicto varios factores y es necesario optar por alguno de ellos.

- Extensibilidad y reutilización: el software debe ser fácil de cambiar; los elementos de software que se produzcan deben ser aplicables de la forma más general posible, debiendo existir un gran inventario de componentes de propósito general que se puedan reutilizar cuando se vaya a desarrollar un nuevo sistema. Aquí de nuevo son útiles ideas similares para mejorar ambas cualidades: cualquier idea que ayude a producir arquitecturas más descentralizadas, cuyos componentes sean autocontenidos y se comuniquen sólo a través de canales restringidos y claramente definidos, será de ayuda. El término **modularidad** abarcará la reutilización y la extensibilidad.

El método orientado a objetos puede mejorar significativamente estos cuatro factores de calidad – por eso es tan atractivo y versátil. También tiene contribuciones significativas que hacer con relación a los otros aspectos, en particular:

- ✓ **Compatibilidad:** el método promueve un estilo común de diseño y módulos e interfaces de sistema estándares, que ayudan a producir sistemas que pueden interactuar entre sí.
- ✓ **Portabilidad:** con el énfasis en la abstracción y la ocultación de la información, la tecnología de objetos estimula a los diseñadores a distinguir entre las propiedades de especificación y de implementación, facilitando los esfuerzos de portabilidad. Las técnicas de polimorfismo y ligadura dinámica harán posible escribir sistemas que se adapten automáticamente a distintos componentes tanto hardware como software de la máquina, por ejemplo diferentes sistemas de ventanas o diferentes sistemas de gestión de base de datos.
- ✓ **Facilidad de uso:** la contribución de las herramientas O-O a los sistemas interactivos modernos y especialmente a sus interfaces de usuario es bien conocida, hasta el punto de que a veces oscurece otros aspectos (hay gente que llama "orientado a objetos" cualquier sistema que use iconos, ventanas o entradas controladas por el ratón).
- ✓ **Eficiencia:** como se señala anteriormente, aunque a primera vista el poder extra de las técnicas orientadas a objetos parecen pasarnos factura, utilizar componentes reutilizables de alta calidad profesional, a menudo conlleva mejoras significativas en el rendimiento.
- ✓ **Oportunidad, economía y funcionalidad:** las técnicas O-O permiten a aquellos que las dominan producir software más rápidamente y a menor costo; facilitan la adición de funciones y pueden en sí mismas sugerir nuevas funciones a añadir.

A pesar de todos estos avances se debe tener presente que el método orientado a objetos no es una panacea y que muchos problemas habituales de la ingeniería de software se mantienen. Ayudar a señalar un problema no es lo mismo que resolverlo, pero si es un paso importante.

1.3. Sobre el mantenimiento del software

La lista de factores no incluye una cualidad que se menciona con frecuencia: **facilidad de mantenimiento**. Para entender por qué, se debe analizar más en detalle el concepto que subyace: el mantenimiento.

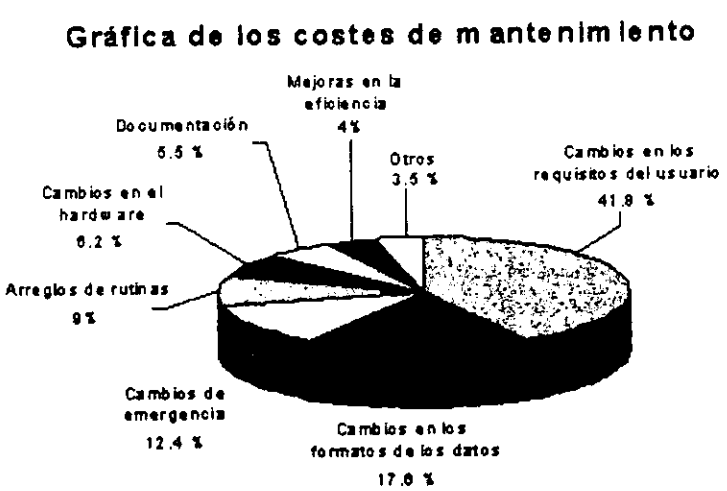
Mantenimiento es lo que sucede después que se ha distribuido un producto de software. Los tratados sobre la metodología de software tienden a centrar la atención en la fase de desarrollo: pero se estima que el 70% del coste del software se dedica al mantenimiento.

Si se reflexiona: ¿qué significa "mantenimiento" en el caso del software? Se puede considerar que el término es inapropiado, ya que un producto de software no se gasta por el uso repetido, y por tanto no necesita ser "mantenido" en la forma que lo es por ejemplo un automóvil o un aparato eléctrico. En realidad, la palabra es utilizada en el mundo del software para describir algunas actividades "nobles" y otras no tanto. La parte noble es la modificación: en la medida que las especificaciones de los sistemas informáticos cambian, reflejando los cambios en el mundo exterior, los sistemas deben cambiar también. La parte menos noble es la depuración a *posteriori:* quitar los errores que en un principio nunca deberían haber existido.

Con el fin de que se comprenda el impacto de estas actividades, se hará uso de un gráfico producto del estudio histórico desarrollado por Lientz y Swanson[12], el cual proporciona una idea general de lo que realmente cubre el término mantenimiento. Este estudio realizó una encuesta en 487 instalaciones que desarrollaban software de todo tipo; aunque puede considerarse algo antiguo, publicaciones más recientes confirman los mismos resultados en términos generales. La gráfica muestra el porcentaje de los costes de mantenimiento en cada una de las actividades identificadas por los autores.

Más de las dos quintas partes esta dedicado a extensiones y modificaciones solicitadas por los usuarios. Esta es a la que se llamó anteriormente la parte noble del mantenimiento, que es una parte inevitable. La pregunta sin respuesta es qué parte del esfuerzo total de la industria se podría ahorrar si se construyese el software desde un principio dedicando más atención a la extensibilidad.

[12] LIENTZ, B. P., SWANSON, E. B.: *Software Maintenance Management: a Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations*, Addison-Wesley, 1980.



Gráfica de los costes de mantenimiento (Lientz ySwanson)

El segundo elemento en el orden decreciente de porcentaje de coste resulta particularmente interesante: el efecto de los cambios en los formatos de los datos. Siempre que cambia la estructura física de los archivos y otros elementos de datos se deben adaptar los programas.

El ejemplo más claro e ilustrativo, es el "problema del año 2000" donde se tuvieron que corregir y en el mejor de los casos adecuar a un sinfín de programas dependientes de fechas cuyos autores ni por un momento se imaginaron que podía existir una fecha más allá del siglo veinte.

Siguiendo en una tónica similar, cuando el servicio postal de los Estados Unidos, introdujo el código postal "5+4" para las grandes compañías (usando nueve dígitos en lugar de cinco) hubo que reescribir numerosos programas que trabajaban con las direcciones y que contemplaban un código postal de cinco dígitos, un esfuerzo que la prensa estimó en varios cientos de millones de dólares.

La cuestión no es que alguna parte del programa "conozca" la estructura física de los datos: esto es inevitable puesto que los datos deben ser accedidos finalmente para su manipulación interna, sino que con las técnicas tradicionales de diseño este conocimiento se distribuye sobre muchas partes del sistema, provocando modificaciones injustificadamente grandes en los programas en el momento en que cambia alguna de las estructuras físicas. En otras palabras, si el código postal pasa de cinco a nueve dígitos, o las fechas requieren un dígito más, es razonable suponer que un programa que manipule los códigos o las fechas necesite ser adaptado; lo que no es aceptable es que esa necesidad se extienda a todo el programa, de tal modo que un cambio en dicha longitud cause cambios en los programas en una magnitud desproporcionada con el tamaño conceptual del cambio en la especificación.

La metodología orientada a objetos proporciona una alternativa para resolver este problema, mediante la teoría de los tipos abstractos de datos, que permite a los

programas acceder a los datos por sus propiedades externas en vez de por su implementación física.

Otro elemento significativo en la distribución de actividades es el bajo porcentaje (5.5%) de los costos de documentación. Debe tenerse presente que estos son los costos de las tareas realizadas en tiempo de mantenimiento. La observación aquí es que un proyecto o presta atención a la documentación como parte del desarrollo o no la considera para nada, situación que a futuro ocasionara problemas para conservar la extensibilidad.

Siguiendo con los elementos del gráfico de Lientz y Swanson: las correcciones de emergencia ante fallos (hechas con precipitación cuando los usuarios informan de que el programa no esta produciendo los resultados esperados o se comporta de modo catastrófico) cuestan más que las correcciones rutinarias planificadas. Esto no es solo porque se tienen que realizar bajo altas presiones, sino porque rompen el proceso ordenado de distribución de nuevas versiones y pueden introducir nuevos errores. Las dos últimas actividades conllevan pequeños porcentajes:

- Una trata de las mejoras de eficiencia: esto debido a que una vez que un sistema funciona, los administradores de los proyectos y los programadores se muestran reacios a interrumpirlo para buscar mejoras de eficiencia y prefieren dejarlo tal y como esta.
- También implicada con un pequeño porcentaje se encuentra la actividad de "transferir a nuevos entornos". Una posible interpretación para ésta es que hay dos clases de programas con respecto a la portabilidad, con pocas diferencias entre ellos: algunos programas están diseñados apegados al concepto de portabilidad y cuesta relativamente poco adaptarlos a diferentes plataformas: otros están tan sujetos a la plataforma original, y serían tan difíciles de transferir, que los desarrolladores ni siquiera lo intentan.

El camino a la Orientación a Objetos

2.1. Criterios de la orientación a objetos

Como preparación para los capítulos posteriores, en los cuales se describirán las técnicas del método orientado a objetos, se considera útil y hasta cierto punto necesario contar con un previo; una visión rápida pero completa de los aspectos clave del desarrollo orientado a objetos. Este es el objetivo de esta sección.

Un beneficio que se pretende obtener es contar con un breve recordatorio de qué es lo que hace que un sistema sea orientado a objetos. Esta expresión se emplea de un modo tan indiscriminado que se hace necesaria una lista de propiedades precisas con respecto a las cuales se pueda valorar cuando un método, lenguaje o herramienta pueda considerarse como orientado a objetos[1].

[1] Se maneja el término *Objectness* en los textos originales en inglés. Desafortunadamente no existe en castellano un término equivalente, traducirlo por *objetividad* provocaría confusión por las otras acepciones de esta palabra.

Primero se examina una selección de criterios para evaluar si se es o no orientado a objetos. La lista de criterios que se presenta aquí incluye las capacidades que se consideran esenciales para la producción de software de calidad empleando el método orientado a objetos. Lo que no significa que un usuario o cualquier responsable de la elección de un método o herramienta deba rechazar un entorno que no cumpla con todas las condiciones propuestas, se debe tomar en cuenta que cada individuo se encuentra inmerso en un contexto particular. Sin embargo es necesario llegar a algunos compromisos:

- "Orientado a objetos" no es una situación booleana: un entorno A, aunque no sea 100% O-O puede ser "más" O-O que un entorno B; de modo que si sus restricciones externas lo llevan a decidir entre A y B el individuo podría decidirse por A al ser la opción orientada a objetos menos mala.
- No todo el mundo necesita todas las condiciones a la vez.
- La orientación a objetos puede ser solo uno de los factores que sirven de guía en la búsqueda de una solución de software, así que quizá sea preciso ponderar los criterios que se proponen aquí con otras consideraciones acordes con las necesidades de cada individuo.

Todo lo anterior no cambia lo que es obvio: para tomar decisiones fundamentadas, incluso si las restricciones prácticas imponen soluciones imperfectas, se necesita conocer la imagen completa, tal y como se muestra en la lista que a continuación se expone.

Categorías

El conjunto de criterios se ha dividido en tres categorías:

1. **Método y lenguaje:** estos dos aspectos casi indistinguibles abarcan los procesos de pensamiento y notaciones que se emplean para analizar y producir software. No se deje de observar que (especialmente en la tecnología orientada a objetos) el término "lenguaje" abarca no solo al lenguaje de programación en el sentido estricto, sino también a las notaciones textuales o gráficas que se usan para el análisis y el diseño.
2. **Implementación y entorno:** los criterios de esta categoría describen las propiedades básicas de las herramientas que permiten a los desarrolladores aplicar las ideas orientadas a objetos.
3. **Bibliotecas:** la tecnología orientada a objetos se basa en la reutilización de componentes de software. Los criterios en esta categoría abarcan tanto la disponibilidad de bibliotecas básicas como los mecanismos necesarios para usar las bibliotecas y para producir otras nuevas.

Esta división en categorías no es absoluta, dado que algunos criterios caen en dos o tres de estas categorías. Por ejemplo, el criterio denominado "administración de la memoria" se ha clasificado dentro del método y el lenguaje, porque un lenguaje puede dar soporte a la recolección automática de basura, o puede no ofrecerla, pero

también pertenece a la categoría de implementación y entorno; de manera similar el criterio de "aserciones" incluye requisitos para las herramientas que los soporten.

2.1.1. Método y lenguaje

El primer conjunto de criterios trata del método y de la notación que utiliza.

Ausencia de discontinuidades

El enfoque orientado a objetos es ambicioso: abarca todo el ciclo de vida de desarrollo del software. Cuando se examinan soluciones orientas a objetos se debe comprobar que el método y el lenguaje, así como las herramientas de apoyo, sean aplicables al análisis y al diseño al igual que a la implementación y al mantenimiento. En particular, el lenguaje debe de ser un vehículo para el pensamiento que ayude durante todas las etapas del trabajo.

El resultado es un proceso de desarrollo sin discontinuidades en el que la generalidad de los conceptos y notaciones ayuda a reducir la magnitud de las transiciones entre las sucesivas etapas del ciclo de vida.

Estos requisitos excluyen dos casos, que aún se encuentran con frecuencia pero que son igualmente no satisfactorios:

- El uso de los conceptos de orientación a objetos sólo en el análisis y el diseño, con un método y una notación que no se pueden usar para escribir software ejecutable.
- El uso de un lenguaje de programación orientado a objetos que no es apropiado para el análisis y el diseño.

Resumiendo:

☞ *Los lenguajes y entornos orientados a objetos, junto con el método en que se basan, deben ser aplicables a todo el ciclo de vida del software, de tal modo que minimicen las discontinuidades existentes entre las sucesivas actividades del ciclo.*

Clases

El método orientado a objetos se basa en la noción de clase. Informalmente una clase es un elemento de software que describe un tipo de dato abstracto y su implementación parcial o total. Un tipo de dato abstracto es un conjunto de objetos definido por una lista de operaciones, o *características* que se pueden aplicar a estos objetos y por las propiedades de estas operaciones.

Por lo tanto:

☞ *El método y el lenguaje deben tener la noción de clase como concepto central.*

Aserciones

Las características de un tipo de datos abstracto tienen propiedades formalmente especificadas, las cuales deben ser reflejadas por las clases correspondientes. Las aserciones – precondiciones de rutinas, postcondiciones de rutinas e invariantes de clase – desempeñan esta labor. Describen el efecto de las características sobre los objetos independientemente de la forma en que estén implementadas estas características.

Las aserciones tienen tres aplicaciones principales: ayudan a producir software fiable, proporcionan una documentación sistemática y constituyen una importante herramienta para la comprobación y depuración del software orientado a objetos.

☞ *Un lenguaje debe de hacer lo posible por dotar a las clases y a sus características con aserciones (precondiciones, postcondiciones e invariantes), basándose en herramientas que producirán la documentación a partir de estas aserciones y que, opcionalmente, permitirán monitorizarlas durante la ejecución.*

Clases y módulos

La orientación a objetos es en principio una técnica de arquitectura: su efecto principal afecta a la estructura modular de los sistemas de software.

El papel clave lo desempeñan las clases. Una clase describe no sólo un tipo de objetos sino también una unidad modular. En un enfoque orientado a objetos puro:

☞ *Las clases deben ser los únicos módulos.*

En particular no hay noción de programa principal y no existen los subprogramas como unidades modulares independientes (estas sólo aparecen formando parte de las clases).

La noción de clase es lo suficientemente potente para evitar la necesidad de ningún otro mecanismo de tipos:

☞ *Todo tipo debe estar basado en una clase.*

Incluso, a manera de ejemplo, los tipos básicos tales como `INTEGER` y `REAL` se pueden derivar de clases, normalmente tales clases estarán integradas en el lenguaje en lugar de tener que ser definida partiendo de cero por cada desarrollador.

Computación basada en características

En la computación orientada a objetos hay un único mecanismo computacional básico: dado un cierto objeto, el cual es siempre una instancia de alguna clase, invoca a una característica de dicha clase aplicándola a ese objeto. Por ejemplo, para mostrar una cierta ventana en una pantalla, se llama a una característica `mostrar` aplicada a

un objeto que represente a la ventana – una instancia de la clase *VENTANA*. Las características pueden tener argumentos. Para aumentar el salario de un empleado *e* en *n* pesos, efectivo a partir de una fecha *f*, se llama a la característica *aumenta* aplicada a *e* con *n* y *f* como argumentos.

Del mismo modo que los tipos básicos se tratan como clases predefinidas, se pueden ver las operaciones básicas (tales como la suma de números) como casos especiales predefinidos de llamadas a características, lo cual es un mecanismo muy general para describir cálculos.

☞ *El mecanismo primario de cálculo tiene que ser la llamada a características.*

Una clase que contenga una llamada a una característica de una clase *C* se dice que es un **cliente** de *C*. La invocación a una característica se conoce con el nombre de **paso de mensajes**; con esta terminología, una llamada como la del ejemplo anterior se describirá como pasarle a *e* el mensaje "aumenta tu paga" con los argumentos *n* y *f*.

Ocultación de información

Cuando se escribe una clase, hay veces en que se tiene que escribir una característica que la clase necesita sólo para propósitos internos: una característica que es una parte de la implementación de la clase pero no de su interfaz. Las otras características de la clase – que posiblemente estén disponibles para los clientes – podrían llamar a esta característica para satisfacer sus propias necesidades, pero para los clientes no es posible llamarla directamente.

El mecanismo que hace que ciertas características no estén disponibles para los clientes se llama ocultación de información. Este es un mecanismo esencial en la evolución suave de los sistemas de software.

En la práctica, para lograr el mecanismo de ocultación de información no basta que se puedan tener características exportadas (disponibles para todos los clientes) y características secretas (no disponibles para ningún cliente); los diseñadores de clases deben tener la capacidad de exportar selectivamente una característica sólo a un determinado conjunto de clientes.

☞ *Debe ser posible para el autor de una clase especificar que una característica esta disponible para todos los clientes, para ningún cliente o para determinados clientes.*

Una consecuencia inmediata de esta regla es que la comunicación entre clases debe estar estrictamente limitada. En particular, un buen lenguaje orientado a objetos no debe ofrecer ninguna noción de variable global; las clases deben intercambiar información exclusivamente a través de llamadas a características y a través del mecanismo de herencia.

Tratamiento de excepciones

Durante la ejecución de un sistema de software se pueden producir sucesos anormales. En un cómputo orientado a objetos, éstos suelen corresponder a llamadas que no se pueden ejecutar satisfactoriamente, como resultado de un mal funcionamiento del hardware, de una imposibilidad inesperada (como un desbordamiento numérico en una suma) o a un error o defecto (*bug*) en el software.

Para producir software fiable, es necesario tener la capacidad de recuperarse ante tales situaciones. Éste es el propósito de un mecanismo de excepciones.

☞ *El lenguaje debe ofrecer un mecanismo para recuperarse ante situaciones anormales inesperadas.*

Haciendo una analogía; en la sociedad de los módulos de software, lugar en donde las clases hacen de ciudades y las instrucciones (el verdadero código ejecutable) hacen las veces del poder ejecutivo del gobierno, las aserciones constituyen el poder legislativo y el mecanismo de excepciones el sistema judicial.

Comprobación estática de tipos

Cuando la ejecución de un sistema de software causa una llamada a una cierta característica aplicada a un cierto objeto. ¿Cómo se puede saber que dicho objeto es capaz de admitir dicha llamada? Es decir ¿cómo se puede saber que el objeto puede procesar el mensaje?

Para brindar una garantía de ejecución correcta, el lenguaje tiene que poseer una comprobación de tipos. Esto significa que se imponen unas pocas reglas de compatibilidad; en particular:

- Toda entidad (es decir, todo nombre que se use en el texto del software para referirse a los objetos durante la ejecución) se declara explícitamente como perteneciente a un cierto tipo, que se deriva de una clase.
- Toda llamada a una característica aplicada a una cierta entidad debe ser una característica de la clase correspondiente (y la característica debe estar disponible, en el sentido de ocultación de la información, para la clase que contiene la rutina que hace la llamada).
- La asignación y el paso de argumentos están sometidos a **reglas de compatibilidad**, basadas en la herencia, que requieren que el tipo original sea compatible con el tipo de destino.

En un lenguaje que impone tal política, es posible escribir un **verificador de tipos estático** que aceptará o rechazará los sistemas de software, garantizando que los sistemas que acepte no causarán en ejecución errores del tipo "*la característica no esta disponible para el objeto*".

☞ *Un sistema de tipos bien definido debería, mediante la imposición de cierto numero de reglas de declaración de tipos y de compatibilidad, garantizar la seguridad de tipos en ejecución de los sistemas que acepte.*

Genericidad

Para que la comprobación de tipos sea práctica, debe ser posible definir clases parametrizadas mediante tipos, conocidas con el nombre de genéricas. Una clase genérica `LISTA[G]` describirá listas de elementos de un tipo arbitrario representado por `G`, el "parámetro formal genérico"; se puede entonces declarar listas especificas a partir de derivaciones tales como `LISTA[INTEGER]` y `LISTA[VENTANA]` empleando los tipos `INTEGER` y `VENTANA` como "parámetros genéricos actuales". Todas las derivaciones comparten el mismo texto de clase.

☞ *Debe ser posible escribir clases con parámetros genéricos actuales que representen tipos arbitrarios.*

Esta forma de parametrización de tipos se llama genericidad **no restringida**.

Herencia simple

El desarrollo de software involucra un gran número de clases; muchas son variantes de otras. Para controlar la complejidad potencial resultante se necesita un mecanismo de clasificación conocido con el nombre de herencia.

Una clase será heredada de otra si incorpora todas las características de la otra además de las propias. (Un *descendiente* es un heredero directo o indirecto; la noción inversa es un *ascendiente*.)

☞ *Debe ser posible que una clase herede a otra.*

La herencia es uno de los conceptos centrales de los métodos orientados a objetos y tiene profundas consecuencias en el proceso de desarrollo del software.

Herencia múltiple

A menudo existe la necesidad de combinar varias abstracciones. Por ejemplo, una clase podría modelar la noción de "niño" que se puede ver como una "persona" con las características asociadas a ésta, y, más prosaicamente, como un "elemento deducible de impuestos", que permite algunas deducciones a la hora de pagar impuestos. La herencia está justificada en ambos casos. La herencia múltiple es la garantía de que una clase puede heredar no sólo de una clase sino de tantas como esté conceptualmente justificado.

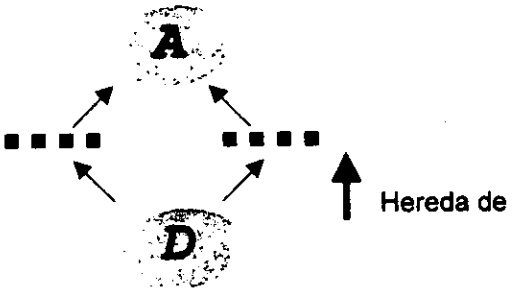
La herencia múltiple plantea algunos problemas técnicos, en particular la resolución de los conflictos de nombres (casos en que características diferentes,

heredadas de clases diferentes, tienen el mismo nombre). Una notación que ofrezca herencia múltiple debe proveer una solución adecuada para estos problemas.

☞ *Debiera ser posible que una clase pueda heredar de tantas clases como sea necesario, con un mecanismo adecuado para eliminar la ambigüedad de los conflictos de nombres.*

Herencia repetida

La herencia múltiple provoca la posibilidad de tener herencia repetida, el caso en que una clase hereda de otra a través de más de un camino, como se muestra a continuación.



Herencia repetida

En este caso el lenguaje debe brindar reglas precisas para definir qué es lo que pasa con aquellas características heredadas repetidamente de un ascendiente común, *A* en la figura. Pudiera ser deseable en algunos casos que una característica de *A* esté sólo una vez en *D* (*compartir*), pero en otros casos debiera estar dos veces (*duplicar*). Los desarrolladores deben disponer de la flexibilidad de poder indicar una política independiente para cada característica.

☞ *Tiene que haber reglas precisas que gobiernen el destino de las características en el caso de herencia repetida, permitiendo a los desarrolladores escoger separadamente, para cada característica heredada repetidamente, si se comparte o se duplica.*

Genericidad restringida

La combinación de la genericidad con la herencia nos brinda una técnica importante, la genericidad restringida, mediante la cual se puede especificar una clase con un parámetro genérico, que represente no a un tipo arbitrario como en la genericidad anterior (no restringida) sino un tipo que tiene que ser descendiente de una clase dada.

Una clase genérica *LISTA_ORDENABLE*, que describe las listas que tienen la característica *ordena* que puede reordenar la secuencia según alguna relación de orden, necesita un parámetro genérico que represente el tipo de elementos de la lista. Este tipo no puede ser arbitrario: debe tener una relación de orden. Para establecer que cualquier parámetro genérico real debe de ser un descendiente de la clase de biblioteca *COMPARABLE*, que describe a los objetos equipados con una relación de orden, se usa la genericidad restringida para declarar la clase en la forma *LISTA_ORDENABLE[G->COMPARABLE]*.

☞ *El mecanismo de genericidad debe admitir la forma restringida de genericidad.*

Redefinición

Cuando una clase hereda a otra, quizá necesite cambiar la implementación u otras propiedades de alguna de las características heredadas. Por ejemplo: una clase *SESION* que describe las sesiones de los usuarios en un sistema operativo puede tener una característica llamada *terminar* para llevar acabo las operaciones de limpieza al finalizar cada sesión; una clase heredada podría ser *SESION_REMOTA*, que maneja sesiones iniciadas en distintas computadoras en una red. Si la terminación de una sesión remota requiere acciones suplementarias (tales como notificar a la computadora remota), la clase *SESION_REMOTA* redefinirá la característica *terminar*.

La redefinición puede afectar la implementación de una característica, su signatura (el tipo de los argumentos y del resultado) y su especificación.

☞ *Debe ser posible redefinir la especificación, la signatura y la implementación de una característica heredada.*

Polimorfismo

Con la herencia en juego, la comprobación estática de tipos que se mencionó anteriormente sería demasiado restrictiva si esto significase que una entidad declarada como de tipo *C* puede referirse a objetos cuyo tipo sea exactamente *C*. Esto significa por ejemplo que una entidad de tipo *C* (en un sistema de control de navegación) no podría usarse para referirse a un tipo *BARCO_MERCANTE* o *YATE_DEPORTIVO* suponiendo que ambas clases heredaran de una clase *EMBARCACION*.

El polimorfismo es la capacidad de una entidad consistente en poder conectarse a objetos de varios tipos. En un entorno de comprobación estática de tipos, el polimorfismo no puede ser arbitrario, sino que tiene que estar controlado por la herencia; por ejemplo, no se puede permitir que una entidad de tipo *EMBARCACION* llegue a estar conectada a un objeto que represente a un tipo *BOYA*, que es una clase que no hereda de *EMBARCACION*.

☞ *Durante la ejecución debiera ser posible conectar entidades (nombres en los textos de software que representan objetos durante la ejecución) a objetos de distintos tipos posibles, bajo el control del sistema de tipos basado en la herencia.*

Ligadura dinámica

La combinación de los dos últimos mecanismos mencionados, redefinición y polimorfismo, sugiere inmediatamente el siguiente. Considérese una llamada cuyo destino es una entidad polimorfa, por ejemplo una llamada a una característica `girar` aplicada a una entidad declarada como de tipo *EMBARCACION*. Los diferentes descendientes de *EMBARCACION* pueden haber redefinido esta característica de diferentes formas. Está claro que debe existir un mecanismo automático para garantizar que la versión de `girar` será siempre la que se deduzca del tipo real de objeto, independientemente de cómo haya sido declarada la entidad. Esta propiedad recibe el nombre de ligadura dinámica.

☞ *La invocación a una característica sobre una entidad debe desencadenar siempre la característica correspondiente al tipo de objeto que durante la ejecución haya sido asociado a dicha entidad, y que no será necesariamente el mismo en diferentes ejecuciones de la llamada.*

La ligadura dinámica ejerce una influencia importante sobre la estructura de aplicaciones orientadas a objetos, ya que permite a los desarrolladores escribir simples llamadas (por ejemplo, la "llamada a la característica `girar` aplicada a la entidad `mi_embarcacion`") para denotar lo que realmente son varias posibles llamadas dependiendo de la situación correspondiente en ejecución. Esto evita la necesidad de muchas preguntas repetidas del estilo "¿Es un barco mercante? ¿Es un yate?" que se dan abundantemente en el software escrito con enfoques más convencionales.

Clases y características diferidas

En algunos casos para los cuales la ligadura dinámica provee una solución elegante, al obviar la necesidad de preguntas explícitas, puede que no exista una versión inicial de la característica a redefinir. Por ejemplo la clase *EMBARCACION* puede ser demasiado general para poder brindar una implementación por defecto de `girar`. Pero se podrá llamar a la característica `girar` aplicada a una entidad de tipo *EMBARCACION* si se puede estar seguro de que en ejecución estará realmente conectada a objetos de tipos completamente definidos como *BARCO_MERCANTE* y *YATE_DEPORTIVO*.

En tales casos *EMBARCACION* debe declararse como una clase diferida (`deferred`) (una que no esta completamente implementada) y con una característica diferida llamada `girar`. Las características y clases diferidas pueden incluir aserciones para describir sus propiedades abstractas, pero su implementación se pospone a clases descendientes. Una clase no diferida se dice que es *efectiva*.

☞ *Debe ser posible escribir una clase o una característica como diferida, es decir, especificada pero no totalmente implementada.*

Las clases diferidas (que también se denominan clases abstractas) son particularmente importantes en el análisis orientado a objetos y en el diseño de alto nivel, ya que permiten capturar los aspectos esenciales de un sistema al mismo tiempo que posponen los detalles para una etapa posterior.

Gestión de memoria y recolección de basura[2]

Los sistemas orientados a objetos, incluso más que los programas tradicionales tienden a crear muchos objetos que tienen a veces interdependencias complejas. Una política consistente en dejar a los desarrolladores a cargo de la gestión de memoria asociada, en especial cuando llegue el momento de reclamar el espacio ocupado por objetos que ya no se necesitan, podrían dañar tanto la eficiencia del proceso de desarrollo (al complicar el software y ocupar una parte considerable del tiempo de los desarrolladores) como la seguridad de los sistemas resultantes (al provocar el riesgo de un reciclaje impropio de las áreas de memoria). En un buen entorno orientado a objetos la gestión de memoria será automática, bajo el control del *recolector de basura (garbage collector)*, un componente del sistema de ejecución.

La razón de que esto sea una característica del lenguaje más que un requisito de implementación es que un lenguaje que no haya sido diseñado explícitamente para poder hacer recolección automática de memoria hace a menudo que ésta sea imposible. Éste es el caso de lenguajes en los que un puntero de un objeto de un cierto tipo puede confundirse con un puntero a otro tipo o incluso como un entero, lo cual hace difícil escribir un colector de basura seguro.

☞ *El lenguaje debe hacer posible una gestión automática de la memoria y la implementación debe ofrecer un administrador automático de memoria capaz de llevar a cabo la recolección de basura.*

2.1.2. Implementación y entorno

A continuación se verán las características esenciales de un entorno de desarrollo que preste su apoyo a la construcción de software orientado a objetos.

Actualización automática

El desarrollo de software es un proceso incremental. Los desarrolladores no suelen escribir miles de líneas de una vez; proceden por adición y modificación, empezando la mayoría de las veces a partir de un sistema que ya es de un tamaño considerable.

---

[2] Del original en inglés *Memory management and garbage collection.*

Cuando se realiza una actualización, es esencial tener la garantía de que el sistema resultante será consistente. Por ejemplo, si se cambia una característica `f` de la clase `C`, se debe tener la certeza de que todo descendiente de `C` que no redefina a `f` será actualizado para tener la nueva versión de `f`, y que toda llamada a `f` desde un cliente de `C` o desde un descendiente de `C` activará la nueva versión.

Los enfoques convencionales de este problema son manuales, y obligan a los desarrolladores a recordar todas las dependencias y a seguirle el rastro a los cambios, usando mecanismos conocidos como archivos "*make*" y los archivos "*include*". Esto es inaceptable en el desarrollo moderno de software, especialmente en el mundo orientado a objetos donde las dependencias entre las clases, que provienen de relaciones de clientes y de herencia, suelen ser complejas pero pueden deducirse de un examen sistemático del texto del software.

☞ *La actualización del sistema después de un cambio debe ser automática, el análisis de las dependencias entre clases será efectuado por herramientas y no manualmente por parte de los desarrolladores.*

Es posible cumplir con este último requisito en un entorno compilado (donde el compilador trabajará en cooperación con una herramienta de análisis de dependencias), en un entorno interpretado, o en una combinación de ambas técnicas de implementación de lenguajes.

Actualización rápida

En la práctica, el mecanismo para la actualización del sistema de los cambios debiera ser no sólo automático, sino también rápido. Más exactamente, debiera ser proporcional al tamaño de las partes que se hayan cambiado y no al tamaño del sistema como un todo. Sin está propiedad, el método y el entorno pueden ser aplicables a pequeños sistemas pero no a sistemas grandes.

☞ *El tiempo necesario para procesar un conjunto de cambios efectuados en un sistema, haciendo posible la ejecución de la versión actualizada, debe ser función del tamaño de los componentes que se hayan modificado, independientemente del tamaño del sistema total.*

Tanto los entornos que trabajan con interpretes como los que trabajan con un compilador pueden cumplir con estos requisitos, aunque en este último caso el compilador debe ser incremental. Junto con un compilador incremental, el entorno puede por supuesto incluir un compilador que realice optimizaciones globales trabajando sobre el sistema completo. Como ese compilador sólo se necesitaría para distribuir el producto final, el desarrollo se basará en el compilador incremental.

Persistencia

Muchas aplicaciones, quizás la mayoría, necesitan conservar objetos de una sesión a otra. El entorno debiera brindar un mecanismo para hacer esto de modo simple.

Los objetos suelen contener referencias a otros objetos, cada objeto puede tener un gran número de objetos dependientes, con un grafo de dependencias posiblemente complejo (que pudiera incluir ciclos). En general, no tendría sentido almacenar o recuperar un objeto sin todas sus dependencias directas o indirectas. Un mecanismo de persistencia que pueda almacenar automáticamente todas las dependencias de un objeto se dice que admite el **cierre de persistencia**.

☞ *Debe estar disponible un mecanismo de almacenamiento persistente que admita el cierre de persistencia para almacenar los objetos y todas sus dependencias en dispositivos externos y para recuperarlos en la misma sesión o en otra.*

Documentación

Los que desarrollan las clases y los sistemas deben ofrecer a los administradores, a los clientes y a los demás desarrolladores unas descripciones claras y de alto nivel del software que producen. Se necesitan herramientas que les asistan en esta labor; la mayor parte posible de la documentación debiera producirse automáticamente a partir de los textos del software. Las aserciones, como ya se ha señalado, ayudan a hacer que esos documentos extraídos del software sean precisos e informativos.

☞ *Deben estar disponibles herramientas automáticas que produzcan información relativa a las clases y a los sistemas.*

Navegación

Cuando se está examinando una clase se necesita frecuentemente obtener información sobre otras clases; en particular las características usadas en una clase pueden haber sido introducidas no en la propia clase sino en varios de sus ascendientes. Esto impone sobre el entorno la carga de ofrecer herramientas para examinar el texto de una clase, encontrar sus dependencias con otras clases y poder pasar con rapidez del texto de una clase al de otra.

Esta tarea se denomina *navegación*. Entre las capacidades típicas que brindan las buenas herramientas de navegación se cuentan las siguientes: encontrar los clientes, proveedores, descendientes y ascendientes de una clase; encontrar todas las redefiniciones de una característica; y encontrar la declaración original de una característica redefinida.

☞ *Las capacidades de navegación interactivas deben permitir a los desarrolladores de software seguir rápida y cómodamente todas las dependencias entre clases y características.*

2.1.3. Bibliotecas

Uno de los aspectos característicos del desarrollo de software por la vía de la orientación a objetos es la posibilidad de basarse en bibliotecas. Un entorno orientado a objetos debiera brindar buenas bibliotecas y buenos mecanismos para poder escribir más.

Bibliotecas básicas

Las estructuras de datos fundamentales de la ciencia de computación – conjuntos, listas, árboles, pilas... – y los algoritmos asociados – ordenación, búsqueda, recorrido, reconocimiento de patrones – aparecen en todas partes en el desarrollo del software. En los enfoques convencionales, cada desarrollador las implementa y reimplementa independientemente y continuamente; esto no significa sólo un desperdicio de esfuerzos sino que va en detrimento de la calidad del software, ya que es poco probable que un desarrollador individual que implementa una estructura de datos no como objetivo en si misma, sino como mera componente de alguna aplicación, se preocupe por alcanzar el punto óptimo de fiabilidad y eficiencia.

Un entorno de desarrollo orientado a objetos debe brindar clases reutilizables dirigidas a estas necesidades comunes de los sistemas de software.

☞ *Deben estar disponibles clases reutilizables que abarquen las estructuras de datos y de algoritmos que son necesarios con más frecuencia.*

Interfaces gráficas de usuario

Muchos de los sistemas modernos son interactivos, e interactúan con sus usuarios a través de gráficos y otras técnicas agradables de interfaz. Ésta es una de las áreas en las que el modelo orientado a objetos ha resultado más útil. Los desarrolladores deben tener la posibilidad de basarse en bibliotecas gráficas para construir aplicaciones interactivas rápida y eficientemente.

☞ *Deben estar disponibles clases reutilizables para desarrollar aplicaciones que brinden a los usuarios agradables interfaces gráficas.*

Mecanismos de evolución de bibliotecas

El desarrollo de bibliotecas de alta calidad es una tarea larga y ardua. Es imposible garantizar que el diseño de una biblioteca sea perfecto desde la primera vez. Un problema importante entonces es capacitar a los desarrolladores para actualizar y

modificar sus diseños sin tener que hacer estragos en los sistemas existentes que dependan de dichas bibliotecas. Este criterio tan importante pertenece a la categoría de bibliotecas, pero también a la categoría de métodos y lenguajes.

☞ *Deben existir mecanismos para facilitar la evolución de las bibliotecas con una perturbación mínima para los clientes del software.*

Mecanismos de indexación de bibliotecas

Otro problema que plantean las bibliotecas es la necesidad de disponer de mecanismos para identificar las clases que satisfacen una cierta necesidad. Este criterio afecta a las tres categorías: bibliotecas, lenguajes (en tanto que debe de haber una forma de insertar información sobre indexación en el texto de las clases) y herramientas (para procesar las consultas que buscan clases que satisfagan ciertas condiciones).

☞ *Las bibliotecas de clases deben estar equipadas con información indexada que permita la recuperación basada en propiedades.*

2.2. Modularidad

A partir de los objetivos de la extensibilidad y la reutilización, dos de los factores de calidad más importantes expuestos en el capítulo I, se desprende la necesidad de tener arquitecturas flexibles, hechas con componentes autónomos de software. Esta es la razón por la que aparece el término modularidad para describir la combinación de estos dos factores.

La programación modular significó en su momento la construcción de programas como ensamblajes de pequeñas piezas, normalmente subrutinas. Pero esta técnica no puede proporcionarlos beneficios de una verdadera extensibilidad y reutilización a menos que se tenga una forma mejor de garantizar que las piezas resultantes – los módulos – sean autocontenidos y estén organizados en arquitecturas estables. Cualquier definición amplia de modularidad debe asegurar estas propiedades.

Un método de construcción de software es modular, por tanto, si ayuda a los diseñadores a producir sistemas de software a partir de elementos autónomos interconectados mediante una estructura simple y coherente.

Una simple definición de modularidad sería insuficiente; al igual que sucede con la calidad del software, se debe ver la modularidad desde diferentes puntos de vista. Esta sección presenta un conjunto de propiedades complementarias: cinco criterios, cinco reglas y cinco principios de modularidad que considerados colectivamente, abarcan los requisitos más importantes de un método de diseño modular.

2.2.1. Cinco criterios

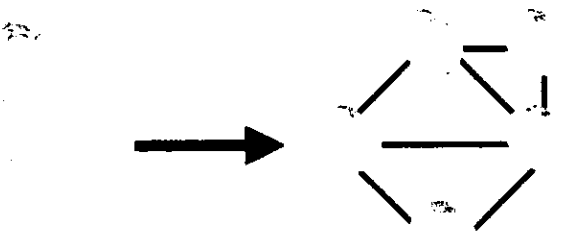
Un método de diseño que merezca ser llamado "modular" debería satisfacer cinco requisitos fundamentales:

- Descomposición.
- Composición.
- Comprensibilidad.
- Continuidad.
- Protección.

Descomposición modular (*Modular decomposability*)

> *Un método de construcción de software satisface la Descomposición Modular si ayuda en la tarea de descomponer el problema de software en un pequeño numero de subproblemas menos complejos, interconectados mediante una estructura sencilla, y suficientemente independientes para permitir que el trabajo futuro pueda proseguir por separado en cada uno de ellos.*

El proceso será frecuentemente iterativo, puesto que cada subproblema pudiera continuar siendo lo suficientemente complejo como para que requiera a su vez su propia descomposición.



Descomposición

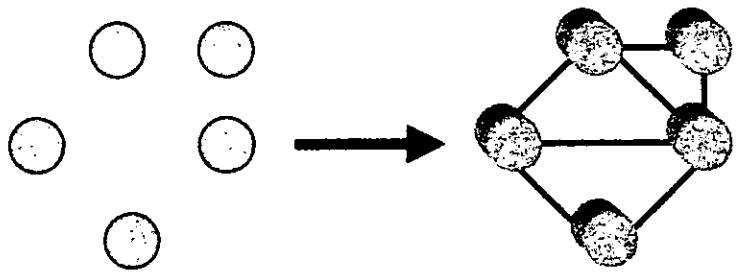
Un corolario del requisito de descomposición es la división de tareas: una vez que se ha descompuesto un sistema en subsistemas se debería poder distribuir el trabajo relativo a estos subsistemas entre diferentes personas o grupos. Éste es un objetivo difícil puesto que limita las dependencias que pueden existir entre los subsistemas:

- Se deben mantener tales dependencias en el mínimo posible; de otra manera el desarrollo de cada subsistema estaría limitado por la marcha del trabajo en otros subsistemas.
- Las dependencias deben ser conocidas: si no se enumeran todas las relaciones entre los subsistemas, entonces al final del proyecto se podría tener un conjunto de elementos que parece que funcionan independientemente pero que no se podrían unir para producir un sistema completo que satisfaga los requisitos globales del problema original.

Composición modular (*Modular composability*)

Un método satisface la Composición Modular si favorece la producción de elementos software que se puedan combinar libremente unos con otros para producir nuevos sistemas en un entorno bastante diferente de aquel en que fueron desarrollados inicialmente.

Así como la descomposición tiene que ver con la derivación de subsistemas a partir de sistemas generales, la composición aborda el proceso inverso: la extracción de elementos software existentes del contexto en que fueron desarrollados originalmente, para utilizarlos de nuevo en contextos diferentes.



Composición

Un método de diseño modular debiera facilitar este proceso generando elementos software que fuesen lo suficientemente autónomos – suficientemente independientes del objetivo inmediato que provoco su existencia – para hacer posible esta extracción.

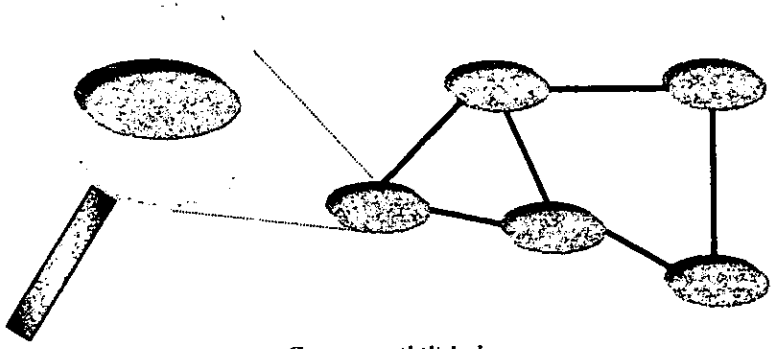
La composición esta directamente relacionada con el objetivo de la reutilización: la meta es encontrar formas de diseñar elementos de software que lleven a cabo tareas bien definidas y que sean utilizables en diferentes contextos. Este criterio refleja una utopía: transformar el proceso de diseño del software en una actividad de construcción "de cajas", de modo que se pudieran construir programas mediante la combinación de elementos prefabricados.

- Ejemplo 1: bibliotecas de subprogramas. Las bibliotecas de subprogramas se diseñan como conjuntos de elementos componibles. Una de las áreas en las que esto ha tenido éxito es el calculo numérico, que se basa en bibliotecas de subrutinas diseñadas cuidadosamente para resolver problemas de álgebra lineal, elementos finitos, ecuaciones diferenciales, etc.
- Ejemplo 2: convenciones del Shell de UNIX: Los mandatos básicos de UNIX actúan sobre una entrada que se considera como un flujo secuencial de caracteres y producen una salida con la misma estructura estándar. Esto hace posible que esas órdenes se puedan componer a través del operador | del lenguaje de órdenes ("shell"): A | B representa un programa que admitirá la entrada que le corresponda a A, hará que A la procese, enviará su salida como entrada hacia B, y hará que la procese B. Este convenio sistemático favorece la composición de herramientas de software.

Comprensibilidad modular (*Modular understandability*)

Un método favorece la comprensibilidad modular si ayuda a producir software en el cual un lector humano pueda entender cada módulo sin tener que conocer los otros, o, en el peor caso, teniendo que examinar sólo unos pocos de los restantes módulos.

La importancia de este criterio se desprende de su influencia en el proceso de mantenimiento. La mayoría de las actividades de mantenimiento, independientemente de si están o no en la categoría noble[3], implican escarbar en los elementos de software existentes. Difícilmente se le puede llamar a un método modular si el lector del software no es capaz de comprender sus elementos por separado.



Comprensibilidad

Este criterio, como los otros, se aplica a los módulos de la descripción de un sistema en cualquier nivel: análisis, diseño e implementación.

[3] vid supra, *1.3 Sobre el mantenimiento del software*, pág.23.

El criterio de comprensión modular ayudará a destacar dos cuestiones importantes: como documentar componentes reutilizables y cómo indexar los componentes reutilizables de modo que los que desarrollan software puedan recuperar éstos cómodamente a través de consultas. El criterio sugiere que la información relativa a un componente, útil para la documentación o para la recuperación, debiera en lo posible aparecer en el texto del propio componente; entonces unas herramientas para la documentación, indexación y recuperación podrán procesar el componente para extraer los elementos de información necesarios. Tener la información incluida dentro de cada componente es preferible a almacenarla en otra parte.

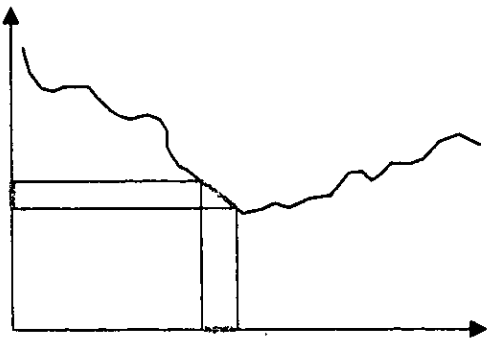
Continuidad modular (*Modular continuity*)

Un método satisface la Continuidad Modular si en la arquitectura de software que produce, un pequeño cambio en la especificación de un problema provoca sólo cambios en un solo módulo o en un pequeño número de módulos.

Este criterio está directamente ligado al objetivo general de la extensibilidad; el cambio es parte integral del proceso de construcción de software. Los requisitos cambian casi inevitablemente con el progreso del proyecto. La continuidad significa que pequeños cambios deberían afectar a módulos individuales de la estructura del sistema, en lugar de afectar a la estructura en sí.

El término "continuidad" esta tomado por analogía con la noción de función continua en el análisis matemático. Una función matemática es continua si (informalmente) un pequeño cambio en el argumento provoca un cambio proporcionalmente pequeño en el resultado. La función que se considera aquí es el método de construcción de software, que se puede ver como un mecanismo para obtener sistemas a partir de especificaciones:

```
metodo_de_construccion_de_software: Especificación - > Sistema
```



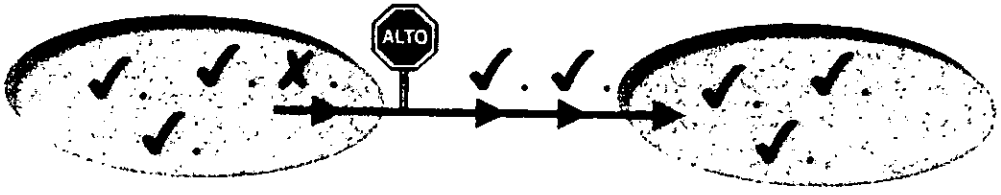
Continuidad

El término matemático sólo nos da una analogía, puesto que carecemos de una noción formal del tamaño del software.

Protección modular (*Modular protection*)

> *Un método satisface la Protección Modular si produce arquitecturas en las cuales el efecto de una situación anormal que se produzca dentro de un módulo durante la ejecución queda confinado a dicho módulo o en el peor caso sólo a unos pocos módulos vecinos.*

El problema subyacente aquí es que los fallos y los errores son una parte fundamental en la ingeniería del software. Los errores que se consideran aquí son errores de ejecución, resultado de fallos de hardware, entradas erróneas o el agotamiento de algunos recursos necesarios (como el espacio en memoria). El criterio no aborda la forma de evitar o corregir los errores, sino el aspecto que es directamente relevante para la modularidad: su propagación.



Protección modular

Ejemplo: Verificar la entrada en su origen. Un método que requiere que cada módulo que introduzca datos sea también responsable de verificar su validez será bueno para la protección modular.

2.2.2. Cinco reglas

De los criterios anteriores se derivan cinco reglas que se deben seguir para asegurar la modularidad:

- Correspondencia directa.
- Pocas interfaces.
- Interfaces pequeñas (acoplamiento débil).
- Interfaces explícitas.
- Ocultación de información.

La primera regla destaca la conexión entre un sistema software y los sistemas externos con los que está conectado; las cuatro restantes abordan todas el mismo problema –cómo se comunican los módulos. La obtención de buenas arquitecturas modulares exige que la comunicación entre módulos ocurra de forma controlada y disciplinada.

Correspondencia directa

Todo sistema software intenta abordar las necesidades de algún dominio del problema. Si se tiene un buen modelo para describir ese dominio, sería deseable mantener una correspondencia clara entre la estructura de la solución, provista por el software, y la estructura del problema, descrita por el modelo. Ésta es la primera regla:

> *La estructura modular obtenida en el proceso de construcción de un sistema software debe seguir siendo compatible con cualquier otra estructura modular obtenida en el proceso de modelado del dominio del problema.*

Esto se deriva en particular de dos criterios de modularidad:

- Continuidad: mantener un rasgo de la estructura modular del problema en la estructura de la solución hará más sencillo estimar y limitar el impacto de los cambios.
- Descomposición: si ya se ha hecho algún trabajo para analizar la estructura modular del dominio del problema, éste puede ser un buen punto de partida para la descomposición modular del software.

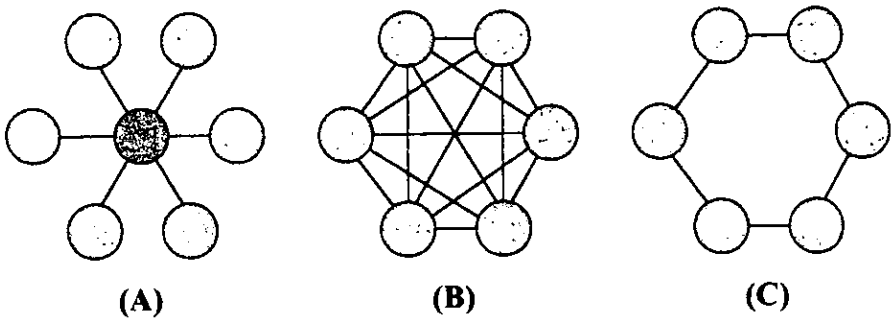
Pocas interfaces

La regla de Pocas Interfaces restringe el número total de canales de comunicación entre módulos en una arquitectura de software:

> *Cada módulo debe comunicarse con el menor número de módulos posible.*

La comunicación entre módulos puede producirse de muchas maneras. Los módulos se pueden llamar unos a otros (si son procedimientos), compartir estructuras de datos, etc. La regla de las pocas interfaces restringe el número de tales conexiones.

Más concretamente, si un sistema se descompone en n módulos, entonces el número de conexiones debería estar lo más cerca posible del mínimo, $n-1$, como muestra (**A**) en la figura, que del máximo, $n(n-1)/2$, como se muestra en (**B**).



Tipos de estructuras de interconexión entre módulos

La regla se deriva en particular de los criterios de continuidad y protección. Si hay muchas relaciones entre los módulos, entonces el efecto de un cambio o de un error podría propagarse a un gran número de módulos. También está ligado a la composición (si se desea que un módulo sea utilizable por sí mismo en un nuevo entorno, entonces éste no debiera depender de muchos otros), la comprensibilidad y la descomponibilidad.

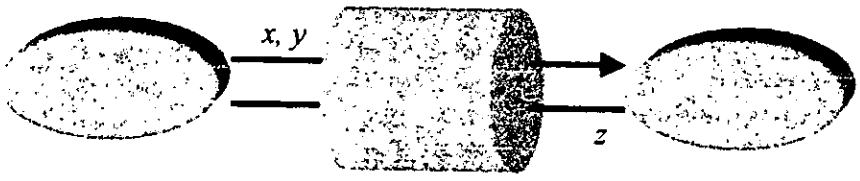
El caso (**A**) de la figura muestra la forma de alcanzar el número mínimo de enlaces, $n-1$, a través de una estructura extremadamente centralizada: un módulo maestro; todos los demás hablan con éste y sólo con éste. Pero hay también estructuras más "igualitarias", como en el caso de (**C**), en la cual hay casi el mismo número de enlaces. En este esquema cada módulo habla únicamente con sus dos vecinos inmediatos, pero no hay una autoridad central. Tal estilo de diseño puede parecer sorprendente a primera vista puesto que no se ajusta al modelo tradicional de diseño funcional descendente. Pero puede producir arquitecturas robustas y extensibles; esta es la clase de estructuras que las técnicas orientada a objetos, aplicadas adecuadamente, tenderán a producir.

Pequeñas interfaces

La regla de Pequeñas interfaces o del "Acoplamiento Débil" se refiere al tamaño de las conexiones entre los módulos más que a la cantidad de conexiones.

> ***Si dos módulos se comunican deben intercambiar la menor información posible.***

En otras palabras "los canales de comunicación entre los módulos deben tener un ancho de banda limitado".



Ancho de banda de las comunicaciones entre módulos

El requisito de las pequeñas interfaces es una consecuencia de los criterios de continuidad y protección.

Interfaces explícitas

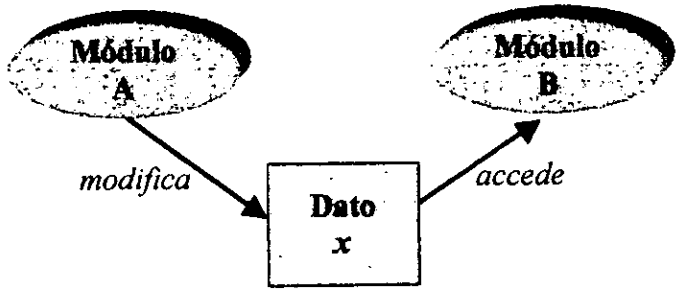
Con la cuarta regla se da un paso adelante para fortalecer un régimen totalitario: Se exige que *toda conversación entre módulos este limitada a unos pocos participantes y que conste de pocas palabras; también se requiere que tales conversaciones deban ser ¡en público y en voz alta!*

> *Siempre que dos módulos A y B se comuniquen, esto debe ser obvio a partir del texto de A, de B o de ambos.*

Detrás de esta regla están los criterios de descomposición y de composición (si se necesita descomponer un módulo en varios módulos o componer uno a partir de otros módulos, cualquier conexión externa debe ser claramente visible), el de continuidad (debe ser fácil de detectar a qué elementos puede afectar un cambio potencial) y la comprensibilidad (¿cómo se puede entender A por sí mismo si B puede influir en su comportamiento de alguna forma intrincada?)

Uno de los problemas para aplicar la regla de las Interfaces Explícitas es que las llamadas a procedimientos no son la única forma de acoplamiento entre módulos; en particular los datos compartidos son una forma de acoplamiento indirecto:

Se supone que un módulo A modifica a un dato x y que otro módulo B tiene acceso a ese dato x. Entonces A y B están de hecho fuertemente acoplados a través de x aun cuando pueda no haber entre ellos ninguna conexión aparente como una llamada a procedimiento.



Compartir datos

Ocultación de información

La regla de ocultación de información se puede enunciar de la siguiente forma:

> *El diseñador de cada módulo debe seleccionar un subconjunto de propiedades del módulo como información oficial sobre el módulo para ponerla a disposición de los autores de módulos clientes.*

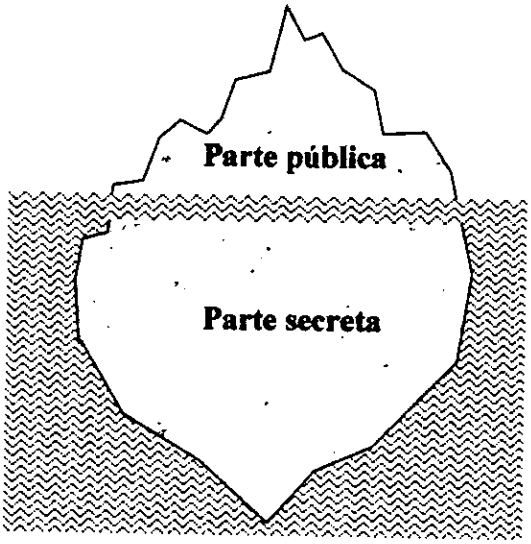
La aplicación de esta regla supone que todo módulo es conocido por el resto del mundo (esto es, por otros diseñadores de otros módulos) a través de alguna descripción oficial o propiedades públicas.

Por supuesto, el texto completo del módulo en sí (el texto del programa, el texto del diseño) podría servir como descripción: esto da una visión correcta del módulo ya que es el propio módulo.

La regla de ocultación de información establece que este no debería ser el caso general: la descripción debería incluir sólo algunas propiedades del módulo. El resto debería permanecer no público o secreto. En lugar de propiedades públicas o secretas, se puede hablar de propiedades exportadas o privadas. Las propiedades públicas del módulo también se conocen como la *interfaz del módulo*.

La razón fundamental que esta detrás de la regla de ocultación de información es el criterio de continuidad. Por ejemplo: suponga que se hacen cambios en un módulo pero que estos se aplican sólo a los elementos secretos del módulo, dejando a los públicos inalterados; entonces los otros módulos que usan a éste, y que se denominan sus clientes, no se verían afectados. Cuanto más pequeña sea la parte pública es mayor la posibilidad de que los cambios a un módulo ocurran en su parte secreta.

Es posible imaginar un módulo que soporta la ocultación de información como un iceberg; sólo la punta – la interfaz – es visible a los clientes.



Un módulo bajo la Ocultación de información

Como otro ejemplo, considérese un procedimiento para recuperar los atributos asociados a una cierta clave en una cierta tabla, tal como un archivo de personal o la tabla de símbolos de un compilador. El procedimiento será internamente muy diferente dependiendo de cómo esté almacenada la tabla (secuencialmente en un array o archivo, en una tabla hash, en un árbol binario, etc.). La ocultación de información implica que el uso de este procedimiento debe ser independiente de la implementación concreta que se haya escogido. De esta forma los módulos clientes no sufrirán frente a ningún cambio de la implementación.

La ocultación de la información hace énfasis en separar la funcionalidad de la implementación. Junto con la continuidad, esta regla también es relativa a los criterios de descomposición, composición y comprensibilidad. No se pueden desarrollar los módulos por separado, combinar varios módulos existentes o entender los módulos individuales, a menos que se sepa lo que cada uno puede o no esperar de los otros.

Como norma general, la parte pública debe incluir la especificación de la funcionalidad del módulo; todo aquello que tenga que ver con la implementación de dicha funcionalidad debe permanecer secreto, con objeto de preservar a los otros módulos contra futuros cambios en las decisiones de implementación.

Para entender qué es ocultación de información y aplicar esta regla apropiadamente es importante evitar una confusión que es bastante común. Independientemente del nombre, la ocultación de información no implica protección en el sentido de restricciones de seguridad – prohibir físicamente a los autores de los módulos clientes tener acceso al texto interno de un módulo proveedor. A los autores clientes bien se les podría permitir leer todos los detalles que quisieran: impedir esto

pudiera ser razonable en algunas circunstancias pero es una decisión de la dirección del proyecto que no es necesariamente consecuencia de la regla de ocultación de información.

Como requisito técnico la ocultación de información significa que los módulos clientes (independientemente de que a sus autores se les permita leer o no las propiedades secretas de los proveedores) deberían basarse solo en las propiedades públicas de los proveedores. Más precisamente debería ser imposible escribir módulos clientes cuya correcta funcionalidad dependa de información secreta.

Considérese de nuevo el ejemplo de un módulo que proporciona un mecanismo para buscar en una tabla. Cierto módulo cliente que pudiera pertenecer a un programa de una hoja de cálculo, usa una tabla y se basa en el módulo `tabla` para buscar cierto elemento en la tabla. Supóngase además que el algoritmo usa una implementación mediante un árbol binario de búsqueda, pero que ésta es una propiedad secreta – no forma parte de la interfaz. Se podría permitir o no al autor del módulo de búsqueda en la tabla decir al autor de la hoja de cálculo qué implementación ha utilizado para las tablas.

Ésta es una decisión de la dirección del proyecto, o quizás una decisión de mercadotecnia (*marketing*); ambos casos son irrelevantes para el asunto de ocultación de información. La ocultación de información significa otra cosa: independientemente de que el autor de la hoja de cálculo conozca que la implementación utiliza un árbol binario de búsqueda, este autor no debe tener la capacidad de escribir un módulo cliente que sólo funcione correctamente para esta implementación – y que ya no funcione si la implementación de la tabla pasa a usar el código para una tabla hash.

La clave para la ocultación de información no es una política de mercado o un asunto de administración en lo tocante a quienes pueden o no tener acceso al texto fuente del módulo, sino unas estrictas reglas de lenguaje que definen los derechos de acceso que tiene un módulo con respecto a las propiedades de sus proveedores.

2.2.3. Cinco principios

A partir de las reglas precedentes e indirectamente de los criterios, se derivan cinco principios relativos a la construcción de software:

- El principio de Unidades Modulares Lingüísticas.
- El principio de Auto-documentación.
- El principio de Acceso Uniforme.
- El principio de Abierto-Cerrado.
- El principio de Elección Única.

Unidades modulares lingüísticas

El principio de unidad modular lingüística expresa que el formalismo utilizado para describir el software en varios niveles (especificación, diseño, implementación) debe admitir la visión de modularidad mantenida:

Principio de Unidades Modulares Lingüísticas

Los módulos deben corresponderse con las unidades sintácticas en el lenguaje utilizado.

El lenguaje mencionado pudiera ser un lenguaje de programación, un lenguaje de diseño, un lenguaje de especificación, etc. En el caso de un lenguaje de programación los módulos debieran poderse compilar por separado.

Lo que este principio excluye a cualquier nivel – análisis, diseño, implementación – es combinar un método que sugiere un cierto concepto de módulo con un lenguaje que no ofrezca la estructura modular correspondiente, forzando a los desarrolladores de software a llevar a cabo traducciones y reestructuraciones manuales. No es extraño ver a compañías que quieren aplicar ciertos conceptos metodológicos (tales como los principios de la orientación a objetos) para luego implementarlos en lenguajes que no los soportan. Semejante enfoque afecta a varios de los criterios de modularidad:

- Continuidad: si los límites del módulo en el texto final no se corresponden con la descomposición lógica de la especificación o del diseño, será difícil o imposible mantener la consistencia entre los distintos niveles cuando el sistema evolucione. Un cambio de la especificación puede considerarse pequeño si afecta sólo a un pequeño número de módulos de especificación; para asegurar la continuidad debe haber una correspondencia directa entre los módulos de especificación, diseño e implementación.
- Correspondencia directa: para mantener una correspondencia clara entre la estructura del modelo y la estructura de la solución se debe tener una clara identificación sintáctica de las unidades conceptuales de ambas partes, que refleje la división sugerida por el método de desarrollo.
- Composición: para dividir el desarrollo del sistema en tareas separadas se necesita estar seguro que cada tarea dará lugar a una unidad sintáctica bien delimitada; en la etapa de implementación estas unidades deben ser compilables separadamente.
- Protección: sólo se puede aspirar a controlar el alcance de los errores si los módulos están sintácticamente delimitados.

Auto-documentación

Al igual que la regla de ocultación de información, el principio de Auto-documentación rige la forma en que se deben documentar los módulos.

Principio de Auto-Documentación.

El diseñador de un módulo debiera esforzarse por lograr que toda la información relativa al módulo forme parte del propio módulo.

Lo que esto excluye es la situación común en que la información relativa al módulo se mantiene en otros documentos del proyecto separados.

La justificación más obvia del principio de auto-documentación es el criterio de la comprensibilidad modular. Sin embargo, quizás sea más importante el papel de este principio para ayudar a satisfacer el criterio de continuidad. Si el software y su documentación se tratan como entidades separadas, es difícil garantizar que éstas vayan a permanecer compatibles – "sincronizadas" – cuando las cosas comiencen a cambiar. Mantenerlo todo en el mismo lugar, aunque no sea una garantía, es una buena manera de ayudar a mantener esta compatibilidad.

Aunque a simple vista puede parecer inocuo, este principio va en contra de mucho de lo que la literatura sobre ingeniería de software suele definir como la correcta práctica del desarrollo de software. El punto de vista dominante es que los desarrolladores de software deben producir "*un kilogramo de papel por cada gramo de producto real*". Estimular a llevar un registro del proceso de construcción del software es un buen consejo – pero no la conclusión de que el software y su documentación sean productos diferentes.

Tal enfoque ignora la propiedad específica del software, sobre la cual se vuelve una y otra vez en esta discusión: su modificabilidad. Si se tratan los dos productos por separado se corre el riesgo de encontrarse rápidamente en una situación en que la documentación dice una cosa y el software hace otra. Si hay una situación peor que la de no tener ninguna documentación ésta debiera ser la de tener una documentación incorrecta.

Acceso uniforme

Aunque en primera instancia puede parecer que sólo trata una cuestión de notación, el principio de Acceso uniforme es de hecho una regla de diseño que influye en muchos aspectos del diseño orientado a objetos y en la notación que lo soporta. Se deriva del criterio de continuidad y también puede verse como un caso especial de ocultación de información.

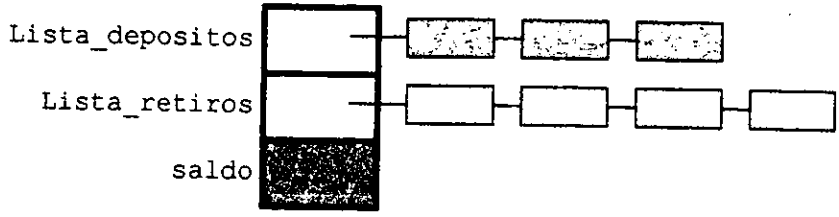
Sea x un nombre utilizado para acceder a cierto elemento de datos y f el nombre de una característica aplicable a x; pudiendo ser esta una variable que represente una cuenta bancaria y f la característica que nos da el saldo actual de la cuenta. El acceso uniforme aborda la pregunta de cómo expresar el resultado de aplicar f a x usando una notación que no haga ningún compromiso prematuro sobre cómo está implementada f.

En la mayoría de los lenguajes de diseño e implementación, la expresión que denota la aplicación de f a x depende de la implementación que el autor del software original haya escogido para la característica f: ¿es un valor almacenado en x o debe ser calculado cada vez que se solicita? Ambas técnicas son posibles en el ejemplo de las cuentas y sus saldos:

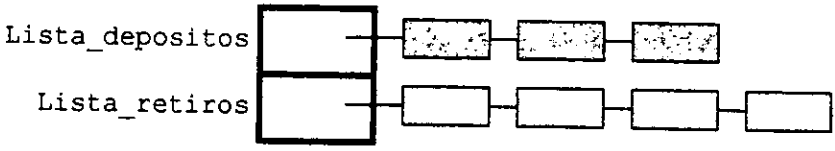
- (E1): Se puede representar el saldo como uno de los campos del registro que describe cada cuenta, como se muestra en la figura. Con esta técnica cada operación que cambie el saldo debe encargarse de actualizar el campo saldo.

- (E2): Se puede definir una función que calcule el saldo usando otros campos del registro, por ejemplo campos que representen listas de depósitos y retiros. Con esta técnica el saldo de una cuenta no está almacenado (no existe el campo saldo) sino que éste se calcula cuando se solicita.

(E1)



(E2)



Una notación común en lenguajes como Pascal, Ada, C, C++ y Java utiliza $x.f$ en el caso de E1 y $f(x)$ en el caso E2.

Decidir entre las representaciones E1 y E2 es fruto de un compromiso entre espacio y tiempo: una economiza el cálculo y la otra el espacio. La decisión de escoger una u otra es una de las decisiones típicas sobre las que los desarrolladores cambian a menudo de opinión al menos una vez durante el tiempo de vida del proyecto. En aras de la continuidad es deseable tener una notación de acceso a las características que no distinga entre los dos casos; entonces si se está a cargo de la implementación de x y se cambia de idea en alguna etapa, no será necesario cambiar los módulos que hagan uso de f. Esto es un ejemplo del principio de acceso uniforme.

En una forma general este principio puede expresarse de la siguiente forma:

Principio de Acceso Uniforme

Todos los servicios ofrecidos por un módulo deben estar disponibles a través de una notación uniforme sin importar si están implementados a través de almacenamiento o de un cálculo.

Principio de Abierto-Cerrado

Otro requisito que cualquier técnica de descomposición modular debe satisfacer es el principio de abierto-cerrado.

La contradicción entre estos dos términos es sólo aparente en tanto que corresponden a dos objetivos de naturaleza diferente:

- Un módulo se llama abierto si está disponible para ser extendido. Por ejemplo, debería ser posible expandir su conjunto de operaciones o añadir campos a sus estructuras de datos.
- Un módulo se llama cerrado si está disponible a ser usado por otros módulos. Esto supone que el módulo tiene una descripción (su interfaz en el sentido de ocultación de información) estable y bien definida. En el nivel de implementación el cierre de un módulo implica también que se puede compilar, tal vez para almacenarlo en una biblioteca y ponerlo a disposición de otros (sus clientes) para que lo usen. En el caso de un módulo de diseño o especificación, cerrar un módulo significa simplemente que haya sido aprobado por la dirección del proyecto, habiendo sido añadido al depósito oficial de elementos software aceptados (lo cual suele denominarse línea base del proyecto) y habiéndose publicado su interfaz para beneficio de los autores de otros módulos.

Principio de Abierto-Cerrado.

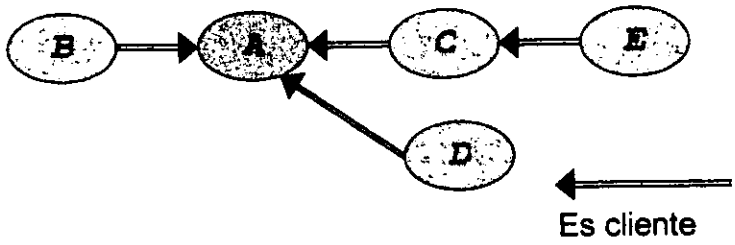
Los módulos deben ser a la vez abiertos y cerrados.

La necesidad de que estos módulos sean cerrados y que a la vez permanezcan abiertos surge de razones diferentes. La apertura es un concepto natural para que los desarrolladores de software, ya que es casi imposible prever todos los elementos – datos, operaciones – que un módulo necesitará durante su tiempo de vida; por tanto éstos desearán retener tanta flexibilidad como sea posible para futuros cambios y extensiones. Pero también es necesario cerrar los módulos, especialmente desde el punto de vista del jefe de proyecto: en un sistema que conste de muchos módulos, la mayoría dependerá de algunos otros; un módulo para interfaces de usuario pudiera depender de un módulo de análisis sintáctico, (para hacer el análisis de los textos de los mandatos) y de un módulo gráfico, el módulo de análisis como tal puede depender de un módulo de análisis léxico, y así sucesivamente. Si nunca se cierra un módulo hasta que no estemos seguros de que incluye todas las características necesarias, ningún software que tenga múltiples módulos alcanzará su terminación. Cada desarrollador podría estar siempre esperando a que otro terminase su tarea.

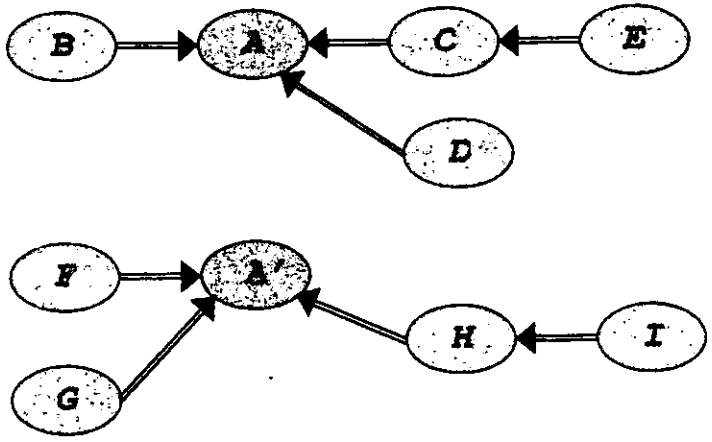
Con las técnicas tradicionales los dos objetivos son incompatibles. O se mantiene abierto el módulo y los demás no pueden usarlo todavía; o se cierra y entonces cualquier cambio o extensión provocará una dolorosa reacción en cadena de cambios en muchos otros módulos que se basan, directa o indirectamente, en el módulo original.

Las siguientes figuras ilustran una situación típica en que las necesidades de módulos abiertos y cerrados son difíciles de reconciliar. En la primera figura, el módulo

A es utilizado por los módulos clientes B, C y D, que pueden a su vez tener sus propios clientes (E, F, ...).



Sin embargo, más adelante la situación se ve alterada por la llegada de nuevos clientes, que necesitan una versión extendida o adaptada de A, que podemos llamar A′:



Con métodos no O-O hay sólo dos soluciones, igualmente insatisfactorias:

- (S1). Se podría adaptar el módulo A de modo que ofreciera la funcionalidad extendida o modificada de (A′) que requieren los nuevos clientes.
- (S2). Se podría dejar a A tal y como está, hacer una copia, cambiar el nombre del módulo por A′ en la copia y llevar a cabo todas las adaptaciones necesarias en el nuevo módulo. Con esta técnica A′ no mantiene relación alguna con A.

El problema potencial con S1 es obvio. A puede existir desde hace tiempo y tener muchos clientes como B, C y D. Las adaptaciones necesarias para satisfacer los requisitos de los nuevos clientes pueden hacer incorrectas las suposiciones en que se basaran los viejos clientes para utilizar A; si es así el cambio de A puede dar comienzo a una dramática serie de cambios en los clientes, cliente de clientes y así sucesivamente.

Superficialmente, la solución S2 parece mejor puesto que no requiere de modificar ningún software existente. Pero de hecho esta solución pudiera ser incluso más catastrófica: una explosión de variantes de módulos originales, muchos de ellos muy similares a los otros aunque nunca sean del todo idénticos.

Para este problema, el método orientado a objetos ofrece una solución especialmente elegante: la herencia.

El concepto completo se manejará más adelante, por el momento sólo se proporciona una visión preliminar de la idea básica de herencia.

Para superar el dilema de cambiar o rehacer, la herencia nos permitirá definir a un nuevo módulo A′ en términos de un módulo ya existente A indicando solamente las diferencias. Se escribirá A′ como:

```
class A′ inherit
    A
    redefine f, g, ... end
feature
f is ...
g is ...
...
u is ...
...
end
```

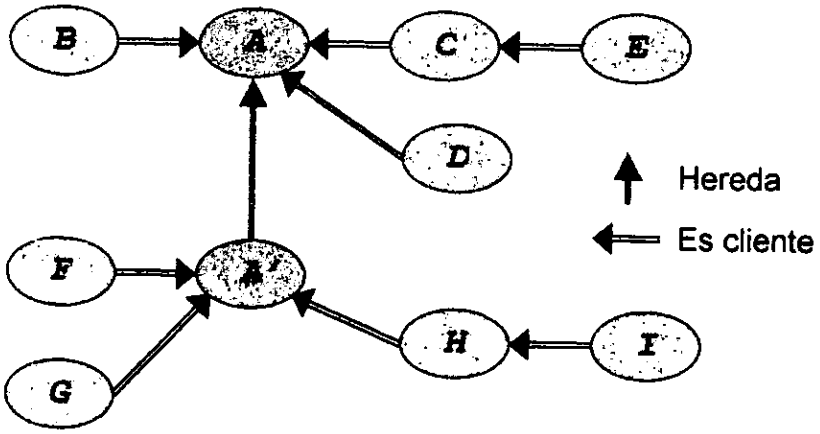
donde la cláusula **feature** contiene la definición de las nuevas características pertenecientes a A', tales como u, y la redefinición de aquellas características (tales como f, g,...) cuya forma en A' es diferente de la que tienen en A.

La representación gráfica de la herencia hará uso de una flecha que va desde la clase hija (la nueva clase, en este caso A′) hacia el padre (en este caso A).

Gracias a la herencia, los desarrolladores O-O pueden adoptar un enfoque mucho más incremental en el desarrollo de software del que era posible con los métodos anteriores.

Sin embargo, hay que tener presente que ni el principio abierto-cerrado ni la redefinición por herencia son formas de abordar los defectos en el diseño, y mucho menos los errores. Si hay algo equivocado en un módulo, hay que corregirlo – no hay que dejar el módulo como está y tratar de arreglar el problema en algún módulo derivado. La única excepción potencial de esta regla es el caso del software con defectos pero que no se nos permite modificar. El principio de abierto-cerrado y sus técnicas asociadas están dirigidos a la adaptación de módulos sanos: módulos que

aunque pueden no ser suficientes para algunos nuevos usos, cumplen con sus propios y bien definidos requisitos, para la satisfacción de sus clientes.



Cómo adaptar un módulo a nuevos clientes

Elección única

El último de los cinco principios de modularidad puede verse como una consecuencia de las reglas de abierto-cerrado y ocultación de la información. Con el fin de que el concepto se maneje con claridad se hará uso del siguiente ejemplo:

Suponga que está construyendo un sistema para gestionar una biblioteca. El sistema manipulará estructuras de datos que representan publicaciones. Se puede declarar el tipo correspondiente según indica a continuación en una sintaxis Pascal.

```
type PUBLICACION =
     record
          autor, titulo: STRING;
          a&o_de_publicacion: INTEGER;
     case tipo_pub: (libro, revista, actas) of
          libro: (editorial: STRING);
          revista: (volumen, numero: STRING);
          acta: (procedencia, numero: STRING)
     end
```

Esta forma hace uso de la noción Pascal de "registro variante" para describir los conjuntos de estructuras de datos con algunos campos (en este caso, `autor, titulo, a&o_de_publicacion`) comunes para todos los ejemplares y otros específicos de las variantes individuales.

Sea *A* un módulo que contiene la declaración anterior. En la medida en que *A* se considere abierto, se pueden añadir campos o introducir nuevas variantes. Sin embargo, para hacer posible que *A* tenga clientes se debe "cerrar" el módulo; esto significa que implícitamente se está considerando que se han enumerado todos los campos y variantes relevantes. Sea *B* un cliente típico de *A*, *B* manipulará las publicaciones a través de una variable como:

```
p: PUBLICACION
```

y, para poder hacer algo útil con *p*, se necesita discriminar explícitamente entre los diferentes casos (variantes), tal como en:

```
case p of
     libro:   …instrucciones que accederían al campo
              p.editorial …
     revista: …instrucciones que accederían a los campos
              p.volumen y p.numero …
     actas:   …instrucciones que accederían al campo
              p.procedencia y p.numero …
end
```

El propósito de la instrucción **case** es precisamente que su sintaxis refleje la forma de la declaración tipo *record* con sus variantes. Pero no importando la sintaxis o mecanismo que se use para representar esta estructura, la observación principal aquí es que para realizar esa discriminación cada cliente debe conocer la lista exacta de la noción de publicación admitida por *A*. La consecuencia es fácil de prever. Tarde o temprano, surgirá la necesidad de una nueva variante, como manuales técnicos.

Entonces habría que extender la definición del tipo *PUBLICACION* en el módulo *A* para considerar el nuevo caso. Se ha modificado la noción conceptual de publicación, de modo que se tiene que actualizar la declaración de tipo correspondiente. Este cambio es lógico e inevitable. Sin embargo, es difícil de justificar la otra consecuencia. Cualquier cliente de *A*, como es el caso de *B*, también requeriría una actualización si usa una estructura como la anterior que se basa en listar explícitamente todos los casos de *p*.

Lo que se ha visto aquí es una situación desastrosa para el cambio y evolución del software: una adición simple y natural puede desencadenar cambios a lo largo de muchos módulos clientes. Esta cuestión surgirá siempre que una cierta noción admita un número fijo de variantes. La noción en el ejemplo fue la de "publicación" y sus variantes iniciales eran libro, revista y actas. Lo que se tiene que aceptar es la posibilidad de que la lista de variantes, aunque fija y conocida en algún punto de la evolución del software, pudiera cambiarse posteriormente al añadir o quitar alguna variante.

Con lo anterior, ya es posible enunciar el principio de elección única y comprender su contexto y objetivo.

Principio de Elección Única

Siempre que un sistema de software deba admitir un conjunto de alternativas, habrá un módulo (y sólo uno) que conozca su lista completa.

Exigiendo que la lista de opciones esté confinada a un solo módulo, se prepara el escenario para cambios posteriores: si se añaden variantes sólo habrá que actualizar el módulo que contiene la información – el punto de elección única. Todos los demás, y en particular los clientes, podrán continuar trabajando en la forma usual.

Como se indico inicialmente, este principio obedece a las reglas de abierto-cerrado y ocultación de información en los siguientes sentidos:

- Considere el ejemplo de las publicaciones a la luz de la figura que ilustra la necesidad de los módulos abiertos y cerrados: A es el módulo que contiene la declaración original del tipo `PUBLICACION`, los clientes B, C, ... son los módulos que se basan en su lista original de variantes; A' sería la versión actualizada de A que ofrece la variante adicional (los manuales técnicos).
- También se puede entender este principio como una forma fuerte de ocultación de información. El diseñador de los módulos A y A' trata de ocultar la información (relativa a la lista de variantes disponibles para un cierto concepto) a sus clientes.

2.3. Enfoques para la reutilización

No es posible negar que existe una cierta reutilización en el desarrollo del software. Se ha dado un paulatino ascenso de los componentes reutilizables, a menudo individualmente modestos pero ganando terreno con regularidad. Estos componentes van desde pequeños módulos para trabajar con Microsoft Visual Basic (VBX) y OLE 2 (OCX, ahora ActiveX) hasta bibliotecas completas, también conocidas como "*framewoks*" para entornos orientados a objetos. A pesar de esto todavía se está lejos de la meta de convertir al desarrollo de software en una industria basada en componentes. Las técnicas de construcción de software orientado a objetos permiten aclarar el ambiente para que esta visión se transforme en realidad, para beneficio no solo de los desarrolladores de software sino, lo que es más importante aún, para quienes necesitan los productos –rápidamente y con alto grado de calidad.

En esta sección se expondrán algunas cuestiones que hay que considerar para que la reutilización tenga éxito en tan gran escala.

2.3.1. Las metas de la reutilización

Primero habría que entender por qué es tan importante la reutilización del software. Los beneficios que se mencionan con más frecuencia no son necesariamente los más significativos; la búsqueda de argumentos en pro de la reutilización debe ser una búsqueda de metas correctas, evitando espejismos, y produciendo la mayor rentabilidad para las inversiones de la empresa.

Beneficios esperados[4]

De un software más reutilizable se pueden esperar mejoras en los siguientes frentes:

- **Oportunidad** (en el sentido expuesto en los factores de calidad de software: velocidad para llevar a los proyectos hasta su culminación y los productos hasta el mercado). Al basarse en componentes ya existentes tenemos menos software que desarrollar y por tanto se puede construir con más rapidez.
- **Disminución de los esfuerzos de mantenimiento.** Si alguien es responsable del software, ese alguien es responsable de su evolución futura. Esto evita la paradoja del desarrollador competente: cuanto más trabaja uno, más trabajo crea para sí mismo, porque los usuarios de sus productos comienzan a pedir nuevas funcionalidades, adaptaciones a nuevas plataformas, etc.
- **Fiabilidad.** Al basarse en componentes de fuentes bien consideradas, se tiene la garantía, o al menos la expectativa, de que sus autores hayan aplicado todo el cuidado necesario incluyendo unas cuidadosas comprobaciones y otras técnicas de verificación; por no mencionar la esperanza de que otros desarrolladores hayan tenido la oportunidad de tratar esas componentes anteriormente subsanando los fallos restantes.
- **Eficiencia.** Los mismos factores que favorecen la reutilización incitan a los desarrolladores de componentes a usar los mejores algoritmos y estructuras de datos que conozcan en su campo de especialización, mientras que en un gran proyecto de aplicación difícilmente se puede esperar tener a mano un experto para cada campo que se trate en el desarrollo. (La mayoría de las personas, al pensar en la relación entre reutilización y eficiencia, tienden a ver el efecto inverso: la pérdida de optimizaciones bien refinadas como consecuencia de estar usando soluciones generales. Pero ésta es una visión estrecha de la eficiencia: en un gran proyecto, no se pueden realmente llevar a cabo esas optimizaciones en cada elemento de desarrollo. Sin embargo, se puede aspirar a las mejores soluciones posibles en las áreas de excelencia de su grupo y para el resto apoyarse en la mayor experiencia de otras personas.)
- **Consistencia.** No hay una buena biblioteca sin un énfasis estricto en un diseño regular y coherente. Si se comienza por utilizar alguna biblioteca – en particular algunas de las mejores bibliotecas orientadas a objetos – su estilo comenzará a influir, a través de un proceso natural de ósmosis, en el estilo del software que se desarrollará. Esto es un gran estímulo para la calidad del software producido por un grupo de aplicación.
- **Inversión.** Hacer el software reutilizable es una forma de preservar las invenciones de los mejores desarrolladores; transformando un recurso frágil en un valor permanente.

Mucha gente, cuando admite la reutilización como algo deseable, piensa solo en el primer argumento de esta lista para mejorar la productividad. Pero ésta no es

[4] MEYER, B.: *Object Success: A Manager's Guide to Object Technology, its Impact on the Corporation, and its Use for Reengineering the Software Process*. Prentice Hall Object Oriented Series, 1995.

necesariamente la contribución más importante a un proceso de software basado en la reutilización. Por ejemplo, el beneficio de la fiabilidad es igual de significativo. Es muy difícil construir software que garantice que será reutilizable si cada nuevo desarrollo debe validar independientemente todos y cada uno de los elementos de una estructura que quizá sea enorme. Al basarse en componentes producidos en cada área, por los mejores expertos, podemos tener la esperanza de construir sistemas en los que se pueda confiar, por que en lugar de estar rehaciendo lo que miles de personas han hecho antes – y caer posiblemente en los mismos errores en que ellos cayeron – podemos concentrar los esfuerzos en la fiabilidad de nuestras verdaderas contribuciones.

Este argumento no sólo se aplica a la fiabilidad. El comentario acerca de la eficiencia estaba basado en el mismo razonamiento. En este sentido se puede ver a la reutilización como algo aparte de los demás factores de la calidad: al mejorarla se tiene el potencial de mejorar casi todas las demás cualidades. La razón es económica: si en lugar de ser desarrollado sólo para un proyecto, un elemento de software tiene la potencialidad de servir una y otra vez para muchos proyectos, comienza a ser económicamente atractivo someterlo a las mejores técnicas posibles de mejoramiento de la calidad – tales como una verificación formal, que normalmente se considera demasiado costosa para la mayoría de los proyectos, u optimizaciones generales, lo cual en circunstancias ordinarias puede rechazarse como un perfeccionismo excesivo. Para los componentes reutilizables, el razonamiento cambia drásticamente, si se mejora un solo elemento, podrían beneficiarse miles de desarrolladores.

Este razonamiento, por supuesto, no es completamente nuevo; forma parte de la traducción al software de ideas que han afectado fundamentalmente a otras disciplinas cuando estas han pasado de la producción artesanal a la producción industrial en masa. Una pastilla VLSI es más cara de construir que un circuito de propósito especial, pero si está bien hecha servirá para incontables sistemas e incrementará su calidad debido a todo el trabajo de diseño que se le ha aplicado de una vez por todas.

2.3.2. ¿Qué es lo que se debería reutilizar?

Convencerse de que la reutilización es buena es la parte fácil. El verdadero reto es: ¿Cómo se obtiene?. Lo primero que hay que preguntar es que es lo que se espera exactamente de la reutilización entre los varios niveles que han sido propuestos y aplicados: reutilización de personal, de especificaciones, de diseños, de "patrones", de código fuente, de componentes especificadas, de módulos abstractos.

Reutilización de personal

La fuente más común de reutilización son los propios desarrolladores. Esta forma de reutilización se utiliza mucho en la industria: al transferir ingenieros de software de un proyecto a otro, las compañías aseguran que la experiencia previa beneficia a los nuevos desarrollos. Este enfoque no técnico de la reutilización es obviamente de alcance limitado.

Reutilización de diseño y especificaciones

En algunas ocasiones se encuentra el argumento de que se debería estar reutilizando diseños en lugar de software real. La idea es que una organización debe acumular un depósito de proyectos que describen las estructuras de diseño aceptadas para las aplicaciones más comunes que ésta desarrolla. Por ejemplo, una compañía que produce sistemas de navegación para aviones tendrá un conjunto de modelos de diseño que resumen su experiencia en esta área, tales documentos describen plantillas de módulos más que módulos reales.

Como ya se expuso en la discusión sobre documentación, la propia noción de diseño como un producto de software independiente, que tiene su vida propia separada de la implementación correspondiente, parece dudosa ya que es difícil garantizar que el diseño y la implementación se mantengan compatibles a lo largo de la evolución de un sistema de software. De modo que si solo se reutiliza el diseño se corre el riesgo de reutilizar elementos incorrectos y obsoletos.

Estos comentarios también son aplicables a una forma relacionada de reutilización: la reutilización de las especificaciones.

Hasta cierto punto se puede ver el progreso de la reutilización en los últimos años, ayudado por un progreso existente en la difusión de la tecnología de objetos y sirviéndole de ayuda a su vez. Anteriormente se pensaba que la única clase de reutilización que realmente tiene interés es la reutilización de diseños y especificaciones. Una visión estrecha de esta idea fue el mayor obstáculo para el progreso, puesto que significaba que todo intento por construir componentes reales se podía descartar por abordar sólo necesidades triviales y no tocar los aspectos verdaderamente difíciles. Este era un punto de vista predominante, y entonces una combinación de argumentos teóricos (los argumentos de la tecnología de objetos) y de logros prácticos (la aparición de componentes reutilizables que tenían éxito) bastó esencialmente para derrotarla.

La idea de reutilizar diseños se vuelve mucho más interesante con un enfoque (como el de la visión de la tecnología de objetos) que elimina gran parte de la lejanía existente entre el diseño y la implementación. La diferencia entre un módulo y el diseño de un módulo será ahora de grado, y no de naturaleza: un módulo de diseño es simplemente un módulo en que algunas partes no están implementadas completamente; y un módulo que esté completamente implementado también puede servir, gracias a las herramientas de abstracción, como un módulo de diseño. Con este enfoque la distinción entre reutilizar módulos y reutilizar diseños tiende a desaparecer.

Patrones de diseño[5]

A mediados de los noventa la idea de patrones de diseño (*design patterns*) comenzó a atraer considerablemente la atención en los círculos de orientación a objetos. Los patrones de diseño son ideas arquitectónicas aplicables a toda una gama

[5] GAMMA, E.; HELM, R.; JOHNSON, R. y VLISSIDES, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading (Mass.), 1995.

de dominios de aplicación; cada patrón hace posible construir una solución para un cierto problema de diseño.

Una de las razones del éxito de la idea de los patrones de diseño es que era más que una idea: el libro que introdujo el concepto y otros que lo han seguido vienen con un catalogo de patrones directamente aplicables que los lectores pueden aprender y aplicar.

Los patrones de diseño han hecho una contribución importante para el desarrollo de la tecnología de objetos y a medida que se sigan publicando otros nuevos ayudarán a los desarrolladores a beneficiarse de la experiencia de sus mayores y de sus iguales. Pero ¿cómo puede contribuir la idea general a la reutilización? Los patrones de diseño no deben estimular un retroceso a la actitud de "lo único que cuenta es la reutilización del diseño" mencionada anteriormente. Un patrón que sea sólo un patrón de libro, aunque sea elegante y general, es una herramienta pedagógica, no una herramienta de reutilización; después de todo, los estudiantes de computación han estado aprendiendo durante tres décadas en sus libros de texto la optimización de consultas relacionales, árboles, el método de ordenación Quicksort y el algoritmo del camino más corto de Dijkstra sin que nadie estuviese manifestando que estas técnicas fuesen avances decisivos en la reutilización. En cierto sentido los patrones desarrollados en los últimos años han sido sólo adiciones incrementales al arsenal habitual de herramientas del software profesional. Desde este punto de vista la nueva contribución son los patrones como tales, no la idea de patrón.

Como casi cualquier persona que haya examinado cuidadosamente el trabajo de los patrones puede haber observado que este enfoque es demasiado limitado. Parece que en el propio concepto de patrón hay una contribución realmente nueva, aun cuando todavía no se haya comprendido en su totalidad. Para ir más allá de su mero valor pedagógico, los patrones tienen que progresar más. Un patrón que tenga éxito no puede ser tan sólo una descripción de libro. Debe ser un componente de software o un conjunto de componentes. Esta meta puede parecer remota a primera vista debido a que muchos patrones son tan generales y abstractos que parece imposible plasmarlos en módulos concretos de software; pero en este sentido el método orientado a objetos brinda una contribución radical. A diferencia de los enfoques anteriores, nos capacita para construir módulos reutilizables que tienen elementos reemplazables: los módulos sirven como esquemas generales (patrones es la palabra apropiada) y se puede adaptar a distintas situaciones especificas. Ésta es la noción de clase de comportamiento (un término más pintoresco es el de *programa con huecos*[*]); se basa en técnicas orientas a objetos, en particular con la noción de clase diferida. Al combinar esto con la idea de grupos de componentes concebidos para trabajar en conjunto – lo cual suele conocerse con el nombre de *framework* o simplemente como bibliotecas – se obtiene una forma notable de reconciliar la reutilización con la adaptabilidad. Más allá de la primera impresión, estas técnicas están llamadas a ejercer, para el movimiento de los patrones, una profunda influencia sobre las prácticas de reutilización.

---

[*] En analogía con los juegos para niños donde los pequeños tienen que emparejar figuras de bloques con figuras de huecos – para comprender que un bloque cuadrado va en un hueco cuadrado y que un bloque circular va en un hueco circular.

Reutilización a través del código fuente

Las formas de reutilización de diseños y de especificaciones, a pesar de su utilidad, ignoran un objetivo clave de la reutilización. Si se ha de llegar al equivalente software de las piezas reutilizables de otras disciplinas de la ingeniería, lo que se necesita reutilizar es la sustancia real de la cual están hechos los productos: el software ejecutable. Ninguno de los objetivos de la reutilización vistos hasta el momento – las personas, los diseños, las especificaciones – se pueden admitir como componentes normales preparados para su inclusión en un producto nuevo de software que se esté desarrollando.

Si lo que se necesita reutilizar es el software, ¿en qué forma debería reutilizarse? La respuesta más natural es utilizar el software en su forma más original: el texto fuente. Este enfoque ha funcionado bien en muchos casos. Por ejemplo, gran parte de la cultura UNIX, extendida originalmente en universidades y laboratorios, gracias a la disponibilidad del código fuente, ha capacitado a los usuarios para estudiar, imitar y extender el sistema.

Los impedimentos económicos y psicológicos de la diseminación del código fuente limitan el efecto que esta forma de reutilización puede tener en entornos industriales más tradicionales. Pero la limitación más seria proviene de dos obstáculos técnicos.

- La identificación de software reutilizable con código fuente reutilizable elimina la ocultación de información. No es posible una reutilización a gran escala sin un esfuerzo sistemático para proteger a los reutilizadores de modo que no tengan que conocer el sinfín de detalles de los elementos reutilizados.
- Los desarrolladores de software distribuido como código fuente pueden sentir la tentación de violar las reglas de modularidad. Algunas partes pueden depender de otras de forma poco evidente, violando las cuidadosas limitaciones de comunicación entre módulos que se han impuesto en las descripciones de la modularidad. Esto suele hacer difícil reutilizar algunos elementos de un sistema complejo sin tener que reutilizar todo lo demás.

Una forma satisfactoria de reutilización debe limitar estos obstáculos admitiendo la abstracción y brindando una reutilización de grano más fino.

Reutilización de los módulos abstractos

Todos los enfoques anteriores, aunque tienen una aplicabilidad limitada, destacan aspectos importantes del problema de la reutilización.

- La reutilización de personal es necesaria aunque no suficiente. Los mejores componentes reutilizables son inútiles sin desarrolladores bien entrenados, que ya hayan adquirido suficiente experiencia para reconocer una situación en la cual los componentes existentes puedan servir de ayuda.
- La reutilización del diseño hace hincapié en la necesidad de componentes reutilizables que sean de una generalidad y de un nivel conceptual suficientemente altos – no sólo prefabricados para problemas especiales. Las clases propuestas por

la tecnología de objetos se pueden ver como módulos de diseño a la vez que como módulos de implementación.

- La reutilización del código fuente sirve como recordatorio de que el software está definido, en última instancia, por los textos de los programas. Una política de reutilización que tenga éxito debe producir elementos de programa reutilizables.

La discusión sobre reutilización del código fuente también ayuda a centrar la búsqueda de las unidades correctas de reutilización. Un componente reutilizable básico debería ser un elemento de software. (Partiendo de aquí se puede llegar por supuesto a colecciones de elementos de software.) Tal elemento debería ser un módulo de tamaño razonable, que satisfaga los requisitos de modularidad. En particular, sus relaciones con el resto del software, si lo hay, deberían estar fuertemente limitadas para facilitar una reutilización independiente. La información que describe las capacidades del módulo y que sirve de documentación primaria para una reutilización real o potencial, debería ser abstracta: en lugar de describir todos los detalles del módulo (como el código fuente), se deberían destacar las propiedades relevantes para los clientes, en consonancia con el principio de ocultación de información.

El término módulo abstracto servirá como nombre para tales unidades de reutilización, formadas por software directamente reutilizable, y disponibles para el mundo exterior a través de una descripción que contiene sólo un subconjunto de las propiedades de cada unidad.

2.3.3. Obstáculos no técnicos para la reutilización

La mayoría de los impedimentos serios de la reutilización son técnicos; su eliminación depende de la asimilación de nuevas técnicas que permitan la creación de un software más flexible y adaptable. Pero por supuesto que también hay algunos obstáculos organizativos, económicos y políticos.

El síndrome NIH

Un obstáculo psicológico que se señala a menudo para la reutilización es el famoso síndrome de No Inventado Aquí *Not invented here* (NIH). Se dice que los desarrolladores de software son individuos que prefieren rehacerlo todo por ellos mismos en lugar de basarse en el trabajo de otros.

Este argumento (que se escucha frecuentemente en círculos directivos) no se ve respaldado por la experiencia. A los desarrolladores de software les gusta tan poco como al que menos el trabajo inútil. Cuando está disponible una solución reutilizable bien publicada y de fácil acceso, esa solución se reutiliza.

Considérese el caso típico del análisis sintáctico y léxico. Mediante el uso de un generador de analizadores como la combinación Lex-Yacc, es mucho más fácil producir un analizador para un lenguaje de órdenes o para un simple lenguaje de programación que si tiene uno que programarlo partiendo de cero. El resultado es claro: donde estén disponibles tales herramientas, los desarrolladores competentes de software las reutilizarán de forma habitual. Escribir un analizador a la medida puede seguir teniendo algún sentido en algunos casos, puesto que las herramientas

mencionadas tienen sus limitaciones. Pero por defecto los desarrolladores acuden a estas herramientas; cuando se quiere una solución que no está basada en mecanismos reutilizables es cuando hay que discutir. De hecho esto puede ser la causa de un nuevo síndrome, el inverso de NIH, que se puede llamar HIN (*Habit Inhibiting Novelty*) hábito de inhibir la innovación: una solución reutilizable útil pero limitada, tan fuertemente arraigada que limita la visión de los desarrolladores y ahoga la innovación, pasa a ser contraproducente. Intente convencer a algún desarrollador de UNIX de que utilice un generador de analizadores que no sea el Yacc y se estrellará frontalmente con un HIN.

Existe algo que puede parecer externamente un NIH y suele ser la reacción cautelosa de los desarrolladores frente a componentes nuevos y desconocidos. Temen que los errores u otros problemas sean más difíciles de corregir que con una solución sobre la que tengan un control total. Con frecuencia, tales temores están justificados por los desafortunados intentos iniciales de reutilizar componentes, especialmente si esto se debe a una orden del jefe de reutilizar componentes a toda costa, sin estar acompañado por unas verificaciones de calidad adecuadas. Si los nuevos componentes son de buena calidad y brindan un servicio real, los temores desaparecerán pronto.

Lo que esto significa para el productor de componentes reutilizables es que la calidad es aún más importante aquí que en las formas más ordinarias de software. Si el costo de una solución no reutilizable específica es *N*, el costo *R* de una solución que se basa en componentes reutilizables nunca será cero: hay un costo de aprendizaje, al menos la primera vez, los desarrolladores tienen que adaptar el software para acomodar los componentes; y tienen que escribir algún software de interfaz, aunque sea pequeño, para poder invocarlos. Por tanto, aun cuando el ahorro por reutilización sea

$$r = \frac{R}{N}$$

y haya otros beneficios de la reutilización potencialmente grandes, hay que convencer también a los candidatos a la reutilización de que la solución reutilizable tiene la calidad suficiente como para justificar su renuncia al control.

Economía de las adquisiciones

Un obstáculo potencial a la reutilización viene de la política de compras de muchas grandes corporaciones y organizaciones gubernamentales, que tienden a impedir los esfuerzos de reutilización al centrar la atención en los costos a corto plazo. Por ejemplo, la legislación de los Estados Unidos hace que sea difícil para una agencia pagarle a un contratista por un trabajo que no esté explícitamente encargado (normalmente como parte de una Solicitud de Propuestas(*)). Esas reglas provienen de la intención legítima de proteger a los que pagan impuestos o a los accionistas, pero también puede desalentar a los constructores de software a la hora de realizar un esfuerzo crucial de generalización para transformar el buen software en componentes reutilizables.

(*)Conocido en inglés por las siglas RFP (*Request for Proposal*)

En un examen más detallado, este obstáculo no parece insuperable. En la medida en que el asunto de la reutilización se vaya extendiendo, puede preverse que las agencias encargadas de las comisiones incluyan en el RFP como tal el requisito de que la solución debe ser de propósito general y reutilizable, y la descripción de la forma en que se van a evaluar las soluciones candidatas con respecto a este criterio. Entonces los que desarrollan software pueden dedicar la debida atención a la tarea de generalización y ser remunerados por ello.

Compañías de software y sus estrategias

Aun cuando los clientes desempeñen su papel para eliminar los obstáculos para la reutilización, aún queda un problema potencial del lado de quienes hacen los contratos. Para una compañía de software hay una tentación constante de brindar soluciones que sean deliberadamente no reutilizables, por miedo a que no se les solicite el próximo trabajo por parte del cliente. Muchas de las compañías de software tienen un importante ingreso a partir de "alquilar mano de obra" – proveyendo de analistas y de programadores a los clientes – y enfocan sus esfuerzos a lograr un alto grado de dependencia de dichos clientes. Esta visión de la ingeniería del software no muestra gran entusiasmo por el proyecto de bibliotecas ampliamente disponibles de componentes reutilizables. Sin embargo si es posible construir de algún modo componentes reutilizables que reemplacen algunos de los costosos servicios que proporcionan las compañías de asesoría de software, tarde o temprano alguien las construirá. En ese momento una compañía que se haya negado a seguir ese camino, y que quedará sin nada que vender salvo los servicios de sus asesores, se lamentará por haber tenido metida la cabeza en la arena.

Es difícil no pensar aquí en las muchas disciplinas de ingeniería que antes necesitaban una mano de obra abundante, y que han pasado a estar industrializadas, es decir a estar basadas en herramientas – con dolorosas consecuencias para las compañías que no entendieron a tiempo lo que estaba sucediendo. Hasta cierto punto, la tecnología de objetos esta provocando un cambio similar en le mercado del software.

Sin embargo, la elección entre personas o herramientas no tiene por qué ser excluyente. La parte de *ingeniería* de la ingeniería del software no es idéntica a la de la industria de producción en masa; los seres humanos seguirán desempeñando el papel clave en el proceso de construcción del software. La meta de la reutilización no es reemplazar a los seres humanos por herramientas (que suele ser, pese a todas las afirmaciones efectuadas, lo que ha ocurrido en otras disciplinas) sino cambiar la distribución de qué es lo que se confía a los humanos y lo que se les encarga a las herramientas. Por tanto, las noticias no son totalmente adversas para una compañía de software que se haya hecho de un nombre gracias a los servicios de asesoría. En particular:

- En muchos casos los desarrolladores que utilizan componentes reutilizables pueden seguir beneficiándose de la ayuda de los expertos, que podrán asesorarlos sobre la mejor forma de utilizar los componentes. Esto deja un papel significativo a las casas de software y a sus asesores.

- Como se discutirá a continuación, la reutilización es inseparable de la extensibilidad: los componentes reutilizables quedarán abiertos para su adaptación a casos específicos. Los asesores de una compañía que desarrolla una biblioteca son los que están en una posición ideal para realizar estas adaptaciones especiales para clientes individuales. De modo que vender componentes y vender servicios no son necesariamente actividades exclusivas; un negocio de componentes puede servir de base para un negocio de servicios.
- De manera más general, una buena biblioteca reutilizable puede desempeñar un papel estratégico en la política de una compañía de software que tenga éxito, incluso si la compañía vende soluciones específicas en lugar de una biblioteca como tal, y usa la biblioteca sólo para su trabajo interno. Si la biblioteca cubre las necesidades más comunes y proporciona bases extensibles para los casos más avanzados, esa biblioteca puede capacitar a la compañía para ganar competitividad en ciertas áreas de aplicación, desarrollando soluciones a la medida de las necesidades de los clientes, más rápidamente y con más bajo coste que los competidores, que no pueden basarse en estas bases prefabricadas.

Acceso a componentes

Otro argumento que se usa para justificar el escepticismo sobre la reutilización es la dificultad de la tarea de administrar los componentes: el progreso en la producción de software reutilizable, según se dice, podría dar lugar a que los desarrolladores estuvieran abrumados por tantos componentes que podría ser que hicieran su vida peor que si los componentes no estuviesen disponibles.

Interpretado en un sentido más positivo, este comentario podría entenderse como una advertencia para los desarrolladores de software reutilizable, en el sentido de que los mejores componentes reutilizables del mundo son inútiles si nadie sabe que existen, o si se necesita mucho tiempo y esfuerzo para obtenerlos. El éxito práctico de las técnicas de reutilización requiere el desarrollo de unas bases de datos de componentes adecuadas, en las que los desarrolladores interesados puedan buscar por palabras claves apropiadas para encontrar rápidamente si existe o no un componente que satisfaga una necesidad en particular. También deben estar disponibles servicios en red, que permitan la solicitud y descarga inmediata de los componentes solicitados.

Estos objetivos plantean problemas técnicos y operativos. Pero se debe poner cada cosa en su lugar. Indexar, recuperar y distribuir componentes reutilizables son cuestiones de ingeniería, a las que se les pueden aplicar herramientas conocidas, en particular la tecnología de bases de datos; no hay ninguna razón por la que los componentes de software tengan que ser más difíciles de gestionar que los registros de clientes, la información de vuelos o los libros de una biblioteca.

Las discusiones de reutilización solían ahondar siempre en la importante cuestión: "¿qué hay que hacer para que los componentes estén disponibles para los desarrolladores de todo el mundo?". Tras los avances de las redes en los últimos años, esos debates ya no parecen tan trascendentales. En particular con el World Wide Web, han aparecido potentes herramientas (AltaVista, Yahoo...) que hacen más fácil localizar información útil, bien en Internet o bien en una Intranet de la compañía. Todo

esto evidencia que la parte verdaderamente difícil del avance en la reutilización radica no en la organización de componentes reutilizables, sino en construir esos componentes en primer lugar.

2.3.4. El problema técnico

¿Qué aspecto debe tener un módulo reutilizable?

Cambio y constancia

Como ya se ha dicho, el desarrollo de software conlleva muchas repeticiones. Para entender las dificultades técnicas de la reutilización se debe entender la naturaleza de la repetición.

Este análisis revela que aunque los programadores tienden a hacer la misma clase de cosas una y otra vez, esas cosas no son exactamente las mismas. Si así fuese la solución sería fácil, al menos sobre el papel; pero en la práctica cambian tantos detalles que hace inviable cualquier intento poco sofisticado de capturar los aspectos comunes.

Considérese el ejemplo de la búsqueda en una tabla. La forma general de un algoritmo de búsqueda en una tabla es similar en muchas ocasiones: empezar en alguna posición de la tabla t y explorar entonces la tabla a partir de esa posición, comprobando cada vez si el elemento que se encuentra en la posición actual es o no el que se está buscando, y si no lo es entonces pasar a la próxima posición. El proceso termina cuando se ha encontrado el elemento o bien se han comprobado infructuosamente todas las posiciones posibles. Un patrón general similar es aplicable a muchos casos posibles de representación de datos y algoritmos de búsqueda en una tabla, incluyendo arrays (ordenados o no), listas enlazadas (ordenadas o no), archivos secuenciales, árboles binarios, B-árboles y tablas hash de distintas clases.

No es difícil convertir esta descripción informal en una rutina que no se encuentra totalmente refinada:

```
tiene (t: TABLA, x: ELEMENTO): BOOLEAN is
      -- ¿Hay alguna ocurrencia de x en t?
      local
            pos: POSICION
      do
            from
                  pos:= POSICION_INICIAL (x, t)
            until
                  AGOTADA (pos, t) or else ENCONTRADA (pos, x, t)
            loop
                  pos:= SIGUIENTE (pos, x, t)
            end
            Result:= not AGOTADA (pos, t)
      end
```

(Algunas aclaraciones sobre la notación: **from ... until ... loop ... end** describe un bucle que se inicializa en la cláusula **from**, ejecuta la cláusula **loop** cero o más veces y termina tan pronto como se satisfaga la condición de la cláusula **until**. *Result* representa el valor que proporciona la función.)

Aunque el texto de código anterior describe (a través de sus elementos en minúsculas) un patrón general de comportamiento del algoritmo, no es directamente una rutina ejecutable puesto que contiene (en letras mayúsculas) algunas partes incompletas, que corresponden a los aspectos del problema de búsqueda en tabla que dependen de la implementación escogida: el tipo de los elementos de la tabla (*ELEMENTO*), qué posición se examina primero (*POSICION_INICIAL*), cómo ir de una posición inicial a la siguiente (*SIGUIENTE*), cómo verificar la presencia de un elemento en una cierta posición (*ENCONTRADO*) y cómo determinar que todas las posiciones interesantes ya han sido examinadas (*AGOTADA*).

Más que una rutina, el texto de anterior es un patrón de rutina, que sólo se puede convertir en una rutina concreta si se refinan las partes que están en mayúsculas.

El dilema de reutilizar o rehacer

Todas estas variaciones destacan los problemas surgidos por cualquier intento de introducir módulos de propósito general en un área de aplicación: ¿cómo se va a sacar partido de un patrón común a la vez que se acomoda la necesidad de más variaciones? Esto no es solamente un problema de implementación: es casi igual de difícil especificar el módulo de modo que los módulos clientes puedan basarse en él sin conocer su implementación.

Estas observaciones señalan el problema central de la reutilización del software, que condena los enfoques simplistas. Debido a la versatilidad del software – precisamente por su naturaleza *soft* (blanda) – los módulos que son candidatos a ser reutilizables no bastarán si son inflexibles.

Un módulo congelado (*frozen*) nos pone ante el dilema de reutilizar o rehacer: reutilizar el módulo exactamente como es, o rehacer la tarea completamente. Esto suele ser demasiado restrictivo. En una situación típica, uno descubre un módulo que puede proporcionarnos una solución para alguna parte de la tarea que tenemos entre manos, pero que no es necesariamente la solución exacta. Las necesidades específicas pueden requerir alguna adaptación del comportamiento original del módulo. De modo que lo que será necesario en estos casos es reutilizar y rehacer: reutilizar una parte, rehacer otra, ó reutilizar una gran parte y rehacer sólo una pequeña parte. Sin esta capacidad para combinar la reutilización y la adaptación, las técnicas de reutilización no pueden proporcionar una solución que satisfaga las realidades prácticas del desarrollo del software.

Esta dualidad entre reutilizar y adaptar se presentó también en la exposición del principio abierto-cerrado, el cual nos plantea que un componente software que tenga éxito debe ser reutilizable tal y como está (cerrado) y además debe seguir siendo adaptable (abierto).

2.3.5. Cinco requisitos relativos a las estructuras de los módulos

¿Cómo encontrar las estructuras de módulos que producirán componentes directamente reutilizables, manteniendo las posibilidades de adaptación?

El asunto de la búsqueda en una tabla y el patrón de rutina `tiene` que se expuso anteriormente ilustra los rigurosos requisitos que tiene que satisfacer cualquier solución. Se puede usar este ejemplo para analizar lo que se necesita para ir de un reconocimiento relativamente vago de los aspectos comunes entre variantes de un software a un conjunto real de módulos reutilizables. Este estudio revela cinco cuestiones generales:

- Variación de tipos.
- Agrupación de rutinas.
- Variación de la implementación.
- Independencia de la representación.
- Factorización del comportamiento común.

Variación de tipos

El patrón de rutina `tiene` considera una tabla que contiene objetos de tipo *ELEMENTO*. Un posible refinamiento pudiera usar un tipo especifico como *INTEGER* o *CUENTA_BANCARIA*, para aplicar el patrón a una tabla de enteros o a una tabla de cuentas bancarias.

Pero esto no es satisfactorio. Un módulo reutilizable de búsqueda debería ser aplicable a muchos tipos diferentes de elementos, sin requerir que los usuarios lleven a cabo cambios manuales en el texto del software. En otras palabras, se necesita la capacidad de describir módulos con tipos parametrizados, también conocidos de manera más concisa como módulos **genéricos**. La genericidad (la capacidad de los módulos para ser genéricos) se va a convertir en una parte importante del método orientado a objetos.

Agrupación de rutinas

Aun cuando haya sido refinada completamente, y a pesar de estar parametrizada con tipos, el patrón de rutina `tiene` no sería completamente satisfactorio como componente reutilizable. La forma en que se busca en la tabla depende de la forma en que se haya creado ésta, de la forma en que se insertan en ella los elementos y de la forma en que estos se eliminan. De modo que una rutina de búsqueda no es suficiente por sí misma como una unidad para la reutilización. Un módulo reutilizable autosuficiente tendría que contener un conjunto de rutinas, una para cada una de las operaciones mencionadas – creación, inserción, eliminación y búsqueda.

Esta idea constituye la base de una forma de módulo, el "paquete" (*package*) que se puede encontrar en lo que podríamos llamar lenguajes de encapsulamiento.

Variación de la implementación

El patrón `tiene` es muy general; en la práctica hay, como se ha visto, una amplia variedad de estructuras de datos y algoritmos que son aplicables. Tal variedad indica que no puede esperarse que un único módulo tenga en cuenta todas las posibilidades; sería enorme. Lo que se necesita es una familia de módulos que abarque todas las implementaciones diferentes.

Una técnica general para producir y usar módulos reutilizables tendrá que admitir la noción de familia de módulos.

Independencia de la representación

Una forma general de módulo reutilizable debería permitir a los clientes especificar una operación sin saber cómo está implementada. Este requisito se llama Independencia de la Representación.

Supóngase que un módulo cliente C de un cierto sistema de aplicación – un programa de gestión, un compilador, un sistema de información geográfica... – necesita determinar si un cierto elemento x aparece o no en una cierta tabla (de inversiones, de palabras clave del lenguaje, de ciudades). La independencia de la representación quiere decir aquí la capacidad de C para obtener esta información a través de una llamada como

```
presente: = tiene (t, x)
```

Sin tener que conocer qué clase de tabla es t en el momento de la llamada. El autor de C sólo necesitaría saber que t es una tabla de elementos de un cierto tipo y que x denota a un objeto de ese tipo. El que t sea un árbol binario de búsqueda, una tabla hash o una lista enlazada es irrelevante para él; el autor debería ser capaz de limitar su conocimiento al problema de gestión, a la compilación o a la geografía. Seleccionar el algoritmo de búsqueda apropiado basado en la implementación de t es asunto del módulo de gestión de la tabla y de nadie más.

Este requisito no impide que los clientes puedan escoger una implementación especifica cuando construyan la estructura de datos. Pero sólo un cliente tendrá que hacer esta selección inicial; después de esto, ninguno de los clientes que llevan a cabo búsquedas en t deberían tener que preguntar qué tipo exacto de tabla es. En particular, el cliente C que contenga la llamada anterior, puede haber recibido t desde uno de sus propios clientes (como argumento de la llamada a alguna rutina); por lo que para C el nombre t es sólo un manejador abstracto de una estructura de datos a cuyos detalles no debiera tener acceso.

Se podría ver la independencia de la representación como una extensión de la regla de ocultación de información, esencial para un desarrollo sin sobresaltos de grandes sistemas: las decisiones de implementación cambian a menudo y los clientes deben quedar protegidos. Pero la independencia de la representación va más allá de esto. Considerada hasta sus últimas consecuencias, la regla significa proteger un módulo contra cambios ocurridos no sólo durante el *ciclo de vida del proyecto* sino

también *durante la ejecución*. En el ejemplo se quiere que `tiene` se adapte automáticamente a la forma de ejecución de la tabla `t`, aún cuando dicha forma haya cambiado desde la última llamada.

Satisfacer la independencia de la representación será una ayuda más para alcanzar un principio relacionado que se vio en la discusión sobre modularidad: la elección única, qué esta encaminado a evitar las estructuras de control con varias ramas que discriminan entre muchas variantes, tal como sería

```
if      "t es un array tratado mediante hash abierto" then
        "Aplicar un algoritmo de búsqueda para hash abierto"
elseif  "t es un árbol de búsqueda" then
        "Aplicar un recorrido de búsqueda binaria en un árbol"
elseif
        (etc.)
end
```

Tener esta estructura de decisión en el propio módulo sería tan desagradable (no podemos esperar razonable que un módulo de gestión de una tabla conozca todas las variantes presentes y futuras) como replicarla en cada cliente. La solución es ocultar completamente la selección con varias alternativas de los desarrolladores de software y hacer que ésta sea llevada a cabo automáticamente por el sistema de ejecución subyacente. Éste será el papel de la ligadura dinámica (*dynamic binding*) un componente clave del enfoque orientado a objetos que se abordará cuando se describa la herencia.

Factorizar los comportamientos comunes

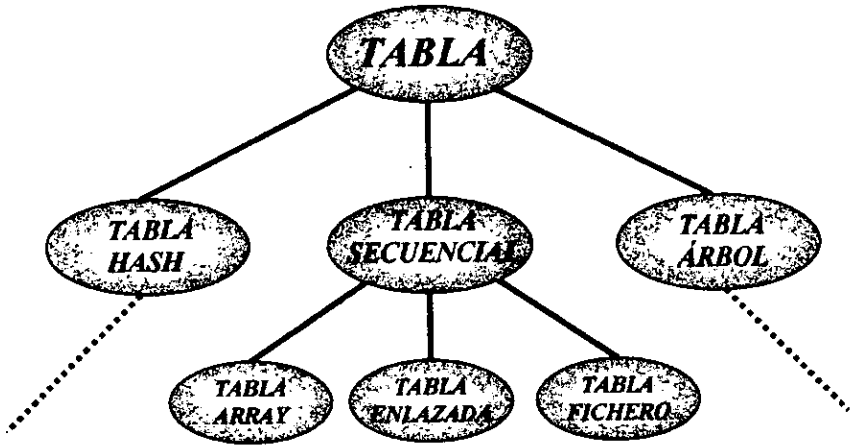
Si la independencia de la representación refleja el punto de vista del cliente respecto a la reutilización – la capacidad de ignorar los detalles y variantes internos de la implementación – entonces este último requisito, factorizar los comportamientos comunes, refleja el punto de vista del proveedor y, más generalmente, el punto de vista de los desarrolladores de clases reutilizables. Su objetivo es aprovechar cualquier posible aspecto común que pueda existir en una familia o subfamilia de implementaciones.

La variedad de implementaciones disponibles en ciertas áreas de problemas suele exigir, como ya se ha visto, una solución basaba en una familia de módulos. A menudo la familia es tan grande que es natural buscar subfamilias. En el caso de las tablas de búsqueda un primer intento de clasificación nos conduce a tres grandes subfamilias:

- Tablas manipuladas por alguna forma de esquemas de codificación hash.
- Tablas organizadas como árboles de alguna clase.
- Tablas manejadas secuencialmente.

Cada una de estas categorías abarca muchas variantes, pero es posible encontrar elementos comunes entre esas variantes. Considérese por ejemplo la familia

de las implementaciones secuenciales – aquellas en que los elementos se guardan y se buscan en el orden de su inserción original.



Algunas posibles implementaciones de una tabla

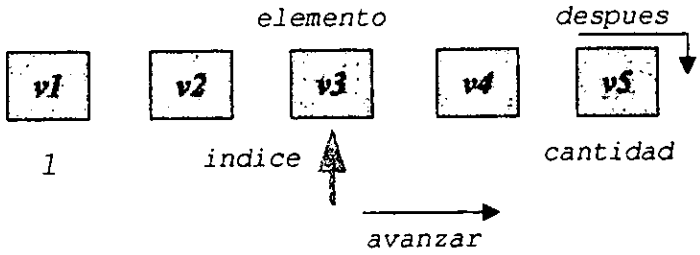
Entre las posibles representaciones de una tabla secuencial están un array, una lista enlazada, un archivo. Pero independientemente de estas diferencias, los clientes deben ser capaces, para cualquier tabla que se maneje secuencialmente, de examinar los elementos uno tras otro mediante el movimiento de un cursor (ficticio) que indica la posición del elemento que se está examinando. En este enfoque pudiera reescribirse la rutina de búsqueda para tablas secuenciales como:

```
tiene (t: TABLA_SECUENCIAL; x: ELEMENTO): BOOLEAN  is
     -- ¿Hay alguna aparición de x en t?
do
          from  inicio until
               despues  or else encontrado (x)
          loop
               avanza
          end
          Result: = not despues
end
```

Esta forma se basa en cuatro rutinas que cualquier implementación de tabla secuencial tiene que ser capaz de suministrar:

- *inicio*, un mandato que lleva el cursor al primer elemento (sí es que lo hay).

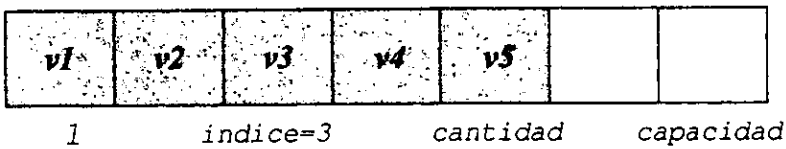
- *avanzar*, un mandato que hace avanzar al cursor una posición. (La implementación de *avanzar* es por supuesto una de las características básicas de una tabla secuencial.)
- *despues*, una consulta booleana que determina si el cursor ha avanzado más allá del último elemento; éste será verdadero después de una llamada a *inicio* si la tabla está vacía.
- *encontrado(x)*, un valor de consulta booleano para determinar si el elemento que se encuentra en la posición del cursor tiene valor *x*.



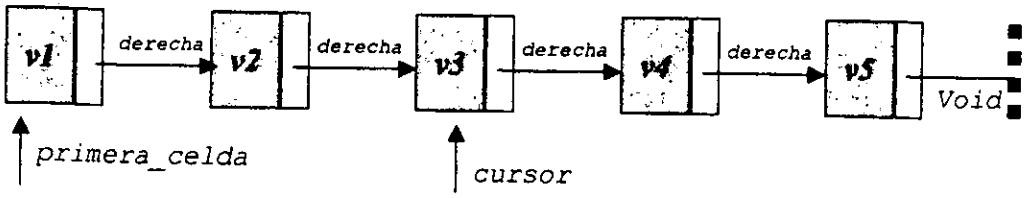
A simple vista, el texto de la rutina *tiene*, que se escribió anteriormente, se asemeja al patrón general de rutina utilizado al comienzo de este análisis, que trataba la búsqueda en cualquier tabla (no sólo secuencial). Pero la nueva forma ya no es un patrón de rutina; es una verdadera rutina, expresada directamente en una notación ejecutable. Dadas unas implementaciones apropiadas para las cuatro operaciones *inicio*, *avanzar*, *despues* y *encontrado* que son invocadas por esta rutina, se puede compilar y ejecutar esta última forma de *tiene*.

Para cada representación posible de una tabla secuencial se necesitará una representación para el cursor. Tres ejemplos de representación son los proporcionados mediante un array, una lista enlazada y un archivo.

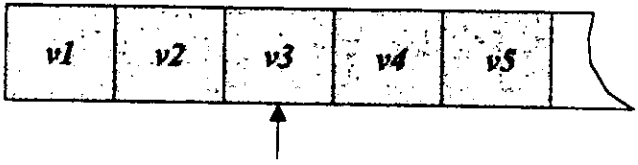
La primera de éstas usa un array de *capacidad* elementos, en donde la tabla ocupará las posiciones desde *1* a *cantidad*. En este caso se puede representar el cursor simplemente mediante un entero *indice* que varía entre 1 y *cantidad + 1*. (Se necesita este último valor para representar el cursor que se ha avanzado hasta "*despues*" del último elemento.)



La segunda representación utiliza una lista enlazada, en la que se accede a la primera celda a través de una referencia llamada *primera_celda* y en donde cada celda está enlazada con la siguiente mediante una referencia llamada *derecha*. El cursor se puede representar mediante una referencia denominada *cursor*.



La tercera representación emplea un archivo secuencial, en el cual el cursor representa simplemente la posición actual de lectura.



Posición de lectura en el archivo

La implementación de las cuatro operaciones de bajo nivel *inicio*, *avanzar*, *despues* y *encontrado* será diferente en cada una de las variantes. La tabla siguiente muestra la implementación en cada caso. (La notación *t@i* denota al *i*-ésimo elemento del array *t*, que podría escribirse en la forma *t[i]* en Pascal o C; *Void* denota una referencia vacía; la notación Pascal *f^* para un archivo *f* representa el elemento situado en la posición actual de lectura del archivo.)

	inicio	avanzar	despues	encontrado
Array	*i:=1*	*i:=i+1*	*i>cantidad*	*t @ i = x*
Lista enlazada	*c:=primera_celda*	*c:= c.derecha*	*c:=Void*	*c.elemento=x*
Archivo	*Rewind*	*read*	*end_of_file*	*f^=x*

En esta tabla *i* significa *indice* y *c cursor*.

En este sentido, el reto de la reutilización es evitar duplicaciones innecesarias de software, aprovechando los aspectos comunes de las diferentes variantes. Si aparecen fragmentos idénticos o casi idénticos en los diferentes módulos, será difícil garantizar su integridad y asegurar que los cambios o correcciones se propaguen a todos los lugares necesarios; una vez más, pueden producirse problemas de gestión de configuraciones.

Todas las variantes de la tabla secuencial comparten la función `tiene` y difieren sólo en la implementación de las cuatro funciones de bajo nivel. Una solución satisfactoria del problema de la reutilización debería contener el texto de `tiene` en un único lugar, asociado de alguna forma a la noción general de tabla secuencial independientemente de toda opción de representación. Para describir una nueva variante ya no hay que volver a preocuparse de `tiene`; lo único que hay que hacer es proporcionar las versiones adecuadas de `inicio`, `avanzar`, `despues` y `encontrado`.

2.3.6 Estructuras modulares tradicionales

Junto a los requisitos de modularidad, los cinco requisitos de variación de tipo, agrupación de rutinas, variación de implementación, independencia de representación y factorizar los comportamientos comunes definen lo que se puede esperar de los componentes reutilizables – los módulos abstractos.

Finalmente se expondrán brevemente las soluciones previas a la O-O para entender por qué éstas no son suficientes – pero también para entender qué se debe aprender y conservar de ellas en un contexto orientado a objetos.

Rutinas

El enfoque clásico de la reutilización es construir bibliotecas de rutinas. Aquí el término *rutina* denota una unidad de software que puede llamar otras unidades para ejecutar un determinado algoritmo; usando ciertas entradas, produciendo ciertas salidas y posiblemente modificando algunos otros elementos de datos. La unidad que hace la llamada pasará sus entradas (y algunas veces las salidas y elementos que haya que modificar) en forma de *argumentos actuales*. Las rutinas también pueden proporcionar una salida en forma de resultado, en este caso se conocen con el nombre de *funciones*.

Las bibliotecas de rutinas han tenido éxito en varios dominios de aplicación, en particular en el cálculo numérico, en el que unas bibliotecas excelentes han originado algunos de los primeros éxitos de la reutilización. La descomposición de sistemas en subrutinas también es lo que se obtiene mediante el método descendente de descomposición funcional. El enfoque de bibliotecas de rutinas parece ciertamente funcionar bien cuando se puede identificar un conjunto (posiblemente grande) de problemas individuales, sometido a las siguientes limitaciones:

- Cada problema admite una especificación sencilla. Más exactamente, es posible caracterizar cada instancia del problema por un pequeño número de argumentos de entrada y de salida.
- Los problemas son claramente distintos entre sí, ya que el enfoque de rutinas no permite sacarle provecho a cualquier característica común significativa que pudiera existir – salvo si se reutiliza algo del diseño.
- No está involucrada ninguna estructura de datos compleja: habría que distribuir éstas entre las rutinas que la usan, perdiéndose la autonomía conceptual de cada módulo.

El problema de búsqueda en la tabla proporciona un buen ejemplo de las limitaciones de las subrutinas. Ya se vio anteriormente que una rutina de búsqueda en sí misma no tiene el contexto suficiente para servir como módulo reutilizable por su cuenta. Aun si no se tiene en presente esta objeción, habría que enfrentarse con dos soluciones igualmente desagradables:

- Una rutina de búsqueda única, que trataría de abarcar tantos casos diferentes que requeriría una lista de argumentos muy grande y sería muy compleja internamente.
- Un gran número de rutinas de búsqueda, cada una de las cuales trataría un caso específico y que diferiría de las demás sólo en algunos detalles, lo cual sería una violación del requisito de factorización de comportamientos comunes. Los candidatos a la reutilización podrían perderse fácilmente en semejante laberinto.

Más generalmente, las rutinas no son suficientemente flexibles como para satisfacer las necesidades de reutilización. Se ha apreciado ya la conexión íntima entre reutilización y extensibilidad. Un módulo reutilizable debería estar abierto a la adaptación, pero con una rutina la única forma de adaptación es pasarle diferentes argumentos.

Paquetes

En los años setenta, con el progreso de las ideas de ocultación de información y de abstracción de datos, surgió la necesidad de una forma de módulo más avanzada que la de rutina. El resultado de esto puede encontrarse en varios lenguajes de diseño y programación de esta época; los más conocidos son CLU, Modula-2 y Ada. Todos ellos ofrecían una forma similar de módulo, conocida como paquete (*package*) en Ada.

Los paquetes son unidades de descomposición del software con las propiedades siguientes:

- P1. De acuerdo con el principio de unidad lingüística modular, un "paquete" es una estructura del lenguaje, de modo que todo paquete tiene un nombre y un ámbito sintáctico claro.
- P2. La definición de cada paquete contiene un cierto número de declaraciones de elementos relacionados, tales como rutinas y variables, que en lo sucesivo se denominarán **características** del paquete.
- P3. Cada paquete puede especificar unos derechos de acceso precisos que gobiernan la utilización de sus características por parte de otros paquetes. En otras palabras, el mecanismo de paquetes admite la ocultación de información.
- P4. En un lenguaje compilable (uno que pueda usarse para la implementación y no sólo para especificación y diseño) es posible compilar paquetes por separado.

Gracias a P3, los paquetes se pueden ver como módulos abstractos. Su contribución más importante es P2, que responde al requisito de agrupación de rutinas. Un paquete puede contener cualquier número de operaciones que estén relacionadas, tales como la creación, inserción, búsqueda y eliminación en tablas.

Como ejemplo para este punto se escribirá el esquema de paquete para un *MANEJADOR_DE_TABLA_DE_ENTEROS* que describe una implementación particular de una tabla de enteros mediante árboles binarios:

```
package MANEJADOR_DE_TABLA_DE ENTEROS  feature
      type ARBOL_BIN_ENT  is
            record
                  --Descripción de la representación de un árbol
                  binario, por ejemplo:
                  info: INTEGER
                  izq, der; ARBOL_BIN_ENT
            end
      nuevo: ARBOL_BIN_ENT is
            --proporciona un nuevo ARBOL_BIN_ENT adecuadamente
            inicializado
            do … end
      tiene ( t: ARBOL_BIN_ENT;  x: INTEGER):  BOOLEAN  is
            --¿Aparece x en t?
            do … implementación de la operación de búsqueda … end
      insertar (t: ARBOL_BIN_ENT;  x: INTEGER):  BOOLEAN  is
            --inserta x en t.
            do … end
      eliminar (t: ARBOL_BIN_ENT;  x: INTEGER)  is
            --Quita x de t.
            do … end
end  -- paquete MANEJADOR_DE_TABLA_DE_ENTEROS
```

Este paquete incluye la declaración de un tipo (*ARBOL_BIN_ENT*) y un cierto número de rutinas que representan las operaciones relativas a objetos de ese tipo. En este caso no hay necesidad de declarar variables en el paquete (aunque las rutinas pueden tener variables locales).

Los paquetes clientes podrán ahora manipular tablas empleando las diferentes características de *MANEJADOR_DE_TABLA_DE_ENTEROS*. Esto supone un convenio sintáctico que permite a un cliente usar una característica *f* de un paquete *P*. Unos extractos típicos de clientes de *MANEJADOR_DE_TABLA_DE_ENTEROS* pueden ser de la forma:

```
--declaraciones auxiliares
x: INTEGER; b: BOOLEAN
      --declaración de t usando un tipo definido en
        MANEJADOR_DE_TABLA_DE_ENTEROS:
t: MANEJADOR_DE_TABLA_DE_ENTEROS$ARB_BIN_ENT
      --inicializar t como una nueva tabla, creada por la función
      nuevo del paquete:
t: MANEJADOR_DE_TABLA_DE_ENTEROS$nuevo
      --insertar el valor x en la tabla, usando el procedimiento
      insertar del paquete:
```

```
MANEJADOR_DE_TABLA_DE_ENTEROS$insertar(t, x)
        --Asigna verdadero o falso a b, dependiendo de si x aparece
        o no en t. Para la búsqueda usa la función tiene el
        paquete:
b:= MANEJADOR_DE_TABLA_DE_ENTEROS$tiene(t, x)
```

Obsérvese la necesidad de inventar dos nombres relacionados: uno para el módulo, en este caso *MANEJADOR_DE_TABLA_DE_ENTEROS* y otro para su tipo principal de datos, en este caso *ARBOL_BIN_ENT*. Uno de los pasos clave hacia la orientación a objetos será la combinación de las dos nociones.

Otra de las limitaciones obvias de los paquetes de la forma anterior es su fracaso para tratar el problema de la variación de tipos: el módulo tal y como se da aquí sólo es útil para tabla de enteros.

Los mecanismos de paquetes proporcionan la ocultación de información al limitar los derechos de los clientes respecto a las características. El cliente que se mostró anteriormente estaba capacitado para declarar una de sus propias variables usando el tipo *ARBOL_BIN_ENT* procedente de su proveedor y también podía invocar rutinas declaradas en el proveedor; pero no tiene acceso ni a los detalles internos de la declaración de tipo (la estructura **record** que define la implementación de las tablas) ni a los cuerpos de las rutinas (sus cláusulas **do**). Además, se pueden ocultar a los clientes algunas características del paquete (variables, tipos, rutinas), haciendo que sólo sean utilizables desde dentro del texto del paquete.

A menudo, para imponer la ocultación de información, los lenguajes con encapsulamiento permiten declarar un paquete en dos partes, interfaz e implementación, relegando los elementos secretos tales como los detalles de la declaración de un tipo o el cuerpo de una rutina a la parte de implementación. Este tipo de política da lugar a un trabajo adicional para los autores de módulos proveedores, obligándoles a duplicar las declaraciones del encabezamiento de las características.

Una valoración de los paquetes

En comparación con las rutinas, el mecanismo de los paquetes supone una mejora significativa en la modularización de los sistemas software en módulos abstractos. La posibilidad de reunir un cierto número de características bajo un mismo techo es útil tanto para los proveedores como para los clientes:

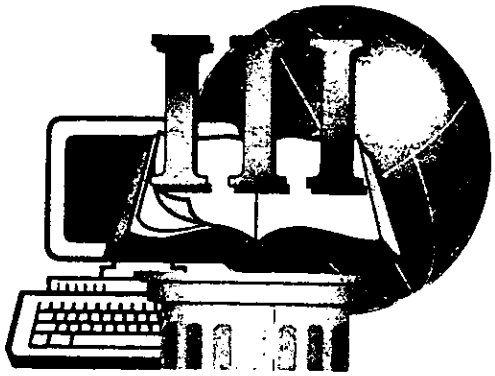
- El autor de un módulo proveedor puede tener en un solo lugar y compilar simultáneamente todos los elementos de software relativos a un concepto dado. Esto facilita la depuración y el cambio. En contraste, con las rutinas separadas siempre existe el riesgo de olvidarse de actualizar alguna de las rutinas cuando se hace un cambio de diseño o de implementación. Por ejemplo, se podría actualizar *nuevo*, *insertar* y *tiene* ... y olvidarse de actualizar *eliminar*.
- Para los autores de clientes, evidentemente es más fácil encontrar y usar un conjunto de facilidades relacionadas si estás se encuentran en un mismo lugar.

La ventaja de los paquetes con respecto a las rutinas es particularmente clara en los casos como los del ejemplo de la tabla, en donde los paquetes agrupan todas las operaciones que se aplican a una cierta estructura de datos.

Pero los paquetes siguen sin proporcionar una solución completa para las cuestiones de reutilización. Según se ha indicado, los paquetes abordan el requisito de agrupación de rutinas, pero dejan los demás sin respuesta. En particular no ofrecen nada para factorizar los aspectos comunes. Se habrá observado que el esquema de *MANEJADOR_DE_TABLA_DE_ENTEROS*, se basa en la selección de una implementación en concreto, los árboles binarios de búsqueda. Pero una biblioteca de componentes reutilizables necesitará proporcionar módulos para muchas implementaciones diferentes. La situación resultante es fácil de pronosticar: una biblioteca típica de paquetes va a ofrecer docenas de módulos similares pero no idénticos en cada una de las áreas (como es el caso de la manipulación de tablas), pero sin forma alguna de aprovechar lo que tengan en común. Para ofrecer la reutilización a los clientes, esta técnica sacrifica la reutilización por parte del proveedor. E incluso del lado del cliente, la situación no es del todo satisfactoria. Todo uso de una tabla por parte de un cliente requiere una declaración como la que sigue:

```
t: MANEJADOR_DE_TABLA_DE_ENTEROS$ARBOL_BIN_ENT
```

lo cual fuerza al cliente a escoger una implementación específica. Esto frustra el requisito de independencia de la representación: los autores de clientes tendrán que saber más de lo conceptualmente necesario respecto a las implementaciones de nociones del proveedor.



TÉCNICAS ORIENTADAS A OBJETOS

La orientación a objetos, como otras metodologías, busca crear una representación de un problema del mundo real y transformarla en una solución que es el software. La diferencia y ventaja de este método es que logra unir dos elementos que habían permanecido divorciados para otros enfoques: los datos y las funciones necesarias para su procesamiento; es decir, *modulariza* los datos con los procesos que actúan sobre ellos en vez de sólo a las funciones.

Las cualidades del método orientado a objetos, no son fortuitas, se deben a la capacidad del mismo para combinar tres conceptos muy importantes dentro del diseño del software: la abstracción, ocultación de información y la modularidad. Si bien es cierto que existen otros métodos que instrumentan mecanismos (en muchos casos complejos) para crear software que se apegue a estas características fundamentales; el método orientado a objetos lo hace por *naturaleza*.

Este capítulo abordará un conjunto pequeño pero potente de técnicas para producir módulos de software extensibles, fiables y reutilizables: los tipos abstractos de datos, las clases, la genericidad y la herencia.

Primeramente se explicarán los tipos abstractos de datos, teoría fundamental para el desarrollo posterior de las clases.

3.1. Tipos Abstractos de Datos

Para lograr que los objetos desempeñen el papel principal en las arquitecturas de software, es necesario describirlos adecuadamente, este es el papel de los tipos abstractos de datos (TAD). Puesto que los tipos abstractos de datos establecen las bases teóricas de todo el método orientado a objetos, su estudio es necesario y hasta cierto punto indispensable.

El método orientado a objetos va más allá de una técnica mas para el desarrollo del software ó una moda de la última década, cuenta con bases teóricas sólidas que le permiten lograr un alto grado de generalización y flexibilidad en las estructuras que resultan de su aplicación. Estas propiedades permiten posteriormente que estas estructuras sean adaptadas o reutilizadas directamente en distintos dominios de aplicación.

3.1.1. Criterios

Para obtener descripciones apropiadas de los objetos se necesita un método que satisfaga tres condiciones:

- Las descripciones deben ser precisas y no ambiguas.
- Deberían ser completas – o al menos tan completas como sea necesario en cada caso (quizá se decida omitir algunos detalles).
- No deberían **especificar más de lo necesario**.

El último punto es el que hace que la respuesta no sea trivial. Después de todo es fácil ser preciso, no ambiguo y completo si se dejan ver todos los detalles de la representación de los objetos. Pero normalmente esto es demasiada información para los creadores de elementos de software que necesitan acceder a los objetos.

Esta observación está próxima a los comentarios que se expusieron en la noción de ocultación de información. La cuestión allí era que al proporcionar un módulo de código fuente (o, más generalmente, elementos relacionados con la implementación) como fuente primaria de información para los creadores de elementos software que se apoyan en dicho módulo, los estamos sumergiendo en un mar de detalles, impidiéndoles concentrarse en su propio trabajo y poniendo obstáculos para nuestras esperanzas de una evolución suave del software. Aquí el peligro es el mismo si se permite que los módulos usen una cierta estructura de datos tomando como base una información que atañe más a la representación de la estructura que a sus propiedades esenciales.

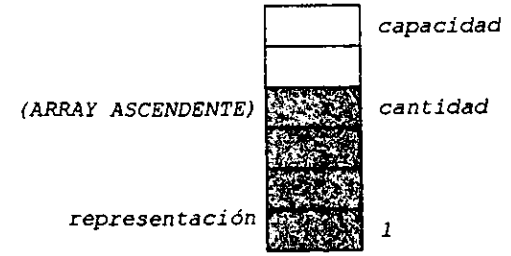
3.1.2. Variaciones de implementación

Para entender por qué es tan crucial la necesidad de descripciones abstractas de los datos se exploraran con detalle las consecuencias potenciales de utilizar las representaciones físicas como base para la descripción de objetos.

Un ejemplo cómodo y bien conocido es la descripción de una pila de objetos. Una pila de objetos sirve para acumular y recuperar otros objetos de tal forma que el

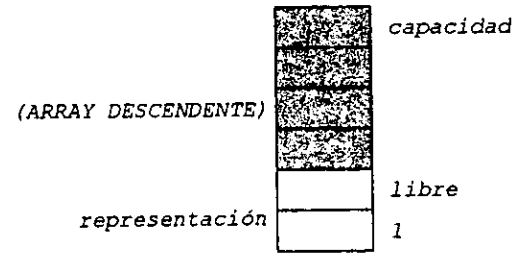
último que entra es el primero que sale, conocido en inglés por las siglas LIFO (*Last In First Out*), es decir, que el último elemento que se inserta en la pila es el primero que se recupera. La pila es una estructura común en informática y ciencias de la computación y, en muchos sistemas software; por ejemplo un típico compilador o intérprete hace uso exhaustivo de pilas de muchas clases.

Representación de pilas



Operación Insertar:

```
cantidad := cantidad + 1
representación [cantidad] := X
```

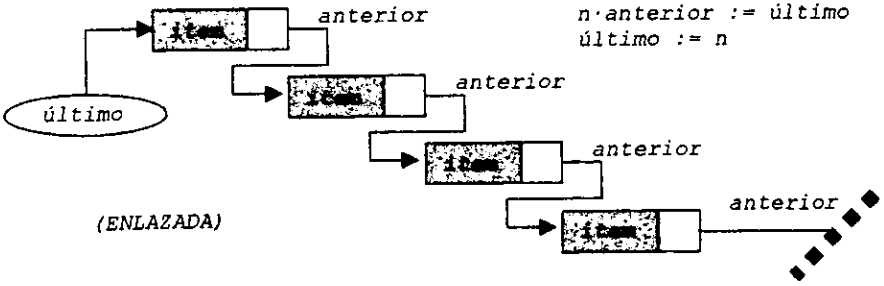


Operación Insertar:

```
representación [libre] := x
libre := libre - 1
```

Operación Insertar:

```
nuevo (n)
n·elem := x
n·anterior := último
último := n
```



Existen varias posibles representaciones físicas de pilas:

Tres posibles representaciones de una pila

La figura ilustra tres de las representaciones más comunes. Se le ha dado un nombre a cada una para facilitar la referencia.

- *ARRAY_ASCENDENTE*: representa una pila mediante un array *representación* y un entero *cantidad* cuyo valor varía entre 0 (para el caso de pila vacía) y *capacidad* que es el tamaño del array *representación*; los elementos de la pila se almacenan en la parte del array que va desde 1 hasta *cantidad*.
- *ARRAY_DESCENDENTE*: es igual que *ARRAY_ASCENDENTE* pero con los elementos almacenados desde el final del array en lugar de desde el principio. Aquí el entero se llama *libre* (es el índice de la posición libre más alta del array, vale 0 si todas las posiciones están ocupadas) y varía desde *capacidad* para una pila que está vacía hasta 0. Los elementos de la pila se guardan en el array desde los índices *capacidad* hacia abajo hasta *libre + 1*.
- *ENLAZADA*: una representación enlazada que almacena cada elemento de la pila en una celda que tiene dos campos: un campo *elem* que representa el elemento y un campo *siguiente* que contiene un puntero a la celda que contiene el elemento que se ha insertado previamente en la pila. Esta representación también necesita de un *último* que es un puntero a la celda en que está el elemento de la cima.

Al lado de cada representación, la figura muestra un extracto de programa (en notación tipo Pascal) que muestra la implementación que corresponde a una operación básica de la pila: insertar un elemento *x* en la cima de la pila.

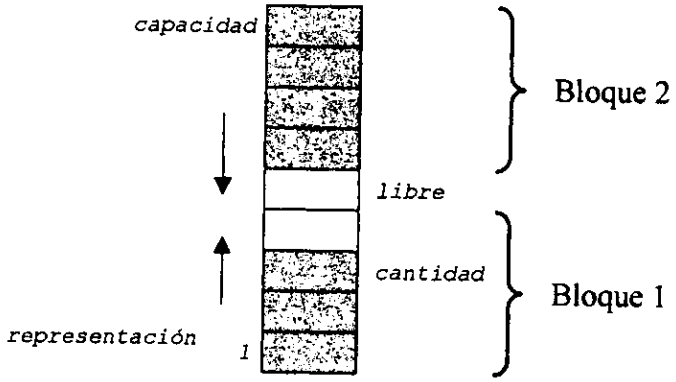
Para las representaciones con array, *ARRAY_ASCENDENTE* y *ARRAY_DESCENDENTE*, las instrucciones incrementan o decrementan el indicador de la cima (*cantidad* o *libre*) y asignan *x* al elemento correspondiente del array. Puesto que estas representaciones son de pilas con un máximo de *capacidad* elementos, para que dichas representaciones sean robustas habría que incluir respectivamente controles de la forma

```
if cantidad < capacidad then ...
if libre > 0 then ...
```

que se omiten en la figura para mayor sencillez.

Para el caso *ENLAZADA*, la representación enlazada, insertar en la pila un elemento requiere cuatro operaciones. Crear una nueva celda *n* (lo que se hace aquí con el procedimiento *nuevo* o *new* en Pascal, el cual asigna espacio para un nuevo objeto); asignar *x* al campo *elem* de la nueva celda; encadenar la nueva celda con la cima anterior de la pila asignándole al campo *siguiente* el valor actual de *último* y actualizar *último* de modo que esté ahora conectado a la nueva celda creada.

Aunque éstas son las representaciones de pila de uso más frecuente, existen muchas otras. Por ejemplo si se necesitan dos pilas de elementos que son del mismo tipo y se tiene un espacio limitado, se puede utilizar un solo array con dos marcadores enteros para las cimas, *cantidad* como en el caso de *ARRAY_ASCENDENTE* y *libre* como *ARRAY_DESCENDENTE*. Una de las pilas crecerá hacia arriba y la otra hacia abajo. La representación completa estará llena si y sólo si *cantidad = libre*.



Representación doble con cabeza para dos pilas

Por supuesto la ventaja aquí es disminuir el riesgo de quedarse sin espacio: con dos arrays de capacidad `n` representando las pilas como `ARRAY_ASCENDENTE` y `ARRAY_DESCENDENTE` se agota el espacio disponible siempre que *alguna* de las pilas alcance los *n* elementos. Con un solo array de tamaño `2n` que contenga las dos pilas cabeza con cabeza, sólo nos quedaremos sin espacio cuando el tamaño *combinado* de ambas alcance `2n`, que es menos probable ya que los dos arrays crecen independientemente. (Para valores variables cualesquiera `p` y `q`, `max(p+q) <=max(p) +max(q).`)

Cada una de estas posibles representaciones, o alguna otra, es útil en determinados casos. Seleccionar una de ellas como "la definición" de pila sería un caso típico de especificar más de lo necesario.

El peligro de la especificación excesiva

¿Por qué es tan malo utilizar una representación particular como especificación?

En los resultados del estudio sobre mantenimiento de Lientz y Swanson; más del 17% de los costes del software provienen de la necesidad de tener en cuenta cambios en los formatos de los datos[1]. Como se indicaba en aquella discusión, hay demasiados programas que están muy estrechamente ligados a la estructura física de los datos que manipulan. Si un método se basa en la representación física de las estructuras de datos para guiar el análisis y el diseño, es probable que no sea adecuado para producir software flexible.

De modo que si se van a utilizar objetos o tipos de objetos como base para las arquitecturas de los sistemas, se debe buscar un criterio de descripción mejor que la representación física.

[1] vid supra, pág. 28.

3.1.3. Hacia una visión abstracta de los objetos

Si se desea tener éxito en describir adecuadamente los objetos, es necesario buscar la completitud, la precisión y la no ambigüedad en dicha descripción, esto sin perder de vista que una especificación excesiva puede ser costosa.

Utilizar las operaciones

En el ejemplo de la pila, lo que une a las diferentes representaciones a pesar de todas sus diferencias es que todas describen una estructura de tipo "contenedor" (una estructura que se usa para almacenar otros objetos) que goza de ciertas propiedades y a la que se le pueden aplicar ciertas operaciones. Si se centra la atención no en una forma particular de representación sino en estas operaciones y propiedades, se puede obtener una caracterización abstracta y útil de la noción de pila.

Las operaciones que típicamente están disponibles en una pila son las siguientes:

- Una orden que pone un elemento en la cima de la pila. Llamaremos a esta operación `put` (*poner*).
- Un mandato que quita el elemento de la cima de la pila, si ésta no está vacía. Se le llamará `remove` (*quitar*).
- Una consulta para ver cuál es el elemento que está en la cima de la pila, si ésta no está vacía. Se le llamará `item` (*elemento*).
- Una consulta para determinar si la pila está vacía. (Esto permite a los clientes determinar de antemano si pueden hacer `remove` o `item`.)

Adicionalmente se puede necesitar una operación de creación que nos dé una pila inicialmente vacía. Se le llamará `make` (*crea*).

En una visión tradicional de las estructuras de datos se podría considerar que la noción de pila está dada por alguna declaración de datos que se corresponde con una de las representaciones anteriores, por ejemplo (para la representación *ARRAY_ASCENDENTE*, en sintaxis tipo Pascal):

```
cantidad: INTEGER
representación: array [1 . . capacidad] of TIPO_ELEMENTOS_PILA
```

donde *capacidad* es una constante entera que denota el máximo número de elementos que puede haber en la pila. Entonces `put`, `remove`, `item`, `empty` y `make` serían rutinas (subprogramas) que operan sobre las estructuras de objetos definidas por estas declaraciones.

El paso clave hacia la abstracción de los datos es invertir este punto de vista: olvidar por el momento la representación y considerar que las operaciones en sí mismas definen la estructura de datos. En otras palabras, una pila **es** *una estructura a la cual los clientes pueden aplicar las operaciones* enumeradas arriba.

Consistencia de los nombres

Si se tiene experiencia en el manejo de estructuras de datos como las pilas, los nombres seleccionados aquí para manejar el ejemplo deben parecer extraños. Cualquier programador o profesional de la informática conocerá las operaciones de pila con otros nombres.

Nombre global de la operación	Nombre usado aquí	Otro posible nombre
push	put	poner	empilar
pop	remove	quitar	desempilar
top	item	elemento	cima
new	make	crear	vacía

¿Por qué entonces utilizar algo distinto de la terminología tradicional? La razón es tener una visión de alto nivel de las estructuras de datos – especialmente de los "contenedores", aquellas estructuras de datos que se utilizan para guardar objetos.

Las pilas son sólo una rama de los contenedores; más exactamente pertenecen a la categoría de los contenedores que se llaman *dispensadores*. Un dispensador le ofrece a los clientes un mecanismo de almacenamiento (`poner`), uno de recuperación (`elemento`) y uno para remover (quitar) objetos pero sin darles ningún control sobre la elección del objeto que se inserta, se recupera o se quita. Por ejemplo la política LIFO de las pilas implica que se puede recuperar o quitar sólo el último elemento en ser almacenado. Otra rama de los dispensadores son las colas que tienen una política donde el primero que entra es el primero que sale, FIFO (*First-In First-Out*): se almacena por un extremo y se recupera y quita por el otro; el elemento que se recupera o se quita es el que lleva más tiempo almacenado en la cola sin haber sido eliminado. Un ejemplo de contenedor que no es un dispensador es un array, en el cual se determina, a través de índices enteros, la posición en la que se quiere almacenar o recuperar.

Debido a que las similitudes entre las distintas clases de contenedores (dispensadores, arrays y otros) son más importantes que las diferencias entre sus propiedades individuales de almacenamiento, recuperación o eliminación, es más prudente respetar una terminología estandarizada que reste importancia a las diferencias entre las variantes de estructuras de datos y en lugar de esto haga hincapié en lo que tienen en común. De modo que la operación para recuperar un elemento siempre se llamará `item`, la operación básica de quitar un elemento siempre se llamará `remove` y así sucesivamente.

Las cuestiones relacionadas con nombres pueden parecer superficiales a primera vista (la "cosmética" de la programación). Pero no se debe olvidar que uno de los objetivos del método orientado a objetos es finalmente proporcionar las bases para bibliotecas potentes y profesionales de componentes de software reutilizables. Tales bibliotecas contendrán decenas de miles de operaciones disponibles. Sin una nomenclatura sistemática y clara, tanto los que desarrollan las bibliotecas como los usuarios de las mismas se verán rápidamente inundados por un torrente de nombres específicos e incompatibles, lo que significa un difícil (e injustificable) obstáculo para la reutilización a gran escala.

La elección de nombres, por consiguiente, no es sólo cosmética. Un buen software reutilizable es un software que proporciona una funcionalidad correcta y que la proporciona con nombres correctos.

3.1.4. Formalizar la especificación

Hasta el momento no se ha logrado una abstracción de los datos suficientemente formal para ser duradera. Considerando el ejemplo de la pila; una pila está definida en término de las operaciones que le son aplicables, luego entonces es necesario definir estas operaciones.

Las descripciones informales como las anteriores (*put* coloca un elemento "en la cima" de la pila, *remove* quita el "último elemento que se colocó en la pila" y así sucesivamente) no son suficientes. Es necesario conocer con precisión cómo los clientes pueden utilizar estas operaciones y lo que ellas realizarán para estos clientes.

Una especificación de tipo abstracto de dato proporcionará esta información. Consta de cuatro párrafos, que se explicarán a lo largo de esta sección:

- TIPOS
- FUNCIONES
- AXIOMAS
- PRECONDICIONES

Estos párrafos se basan en una notación matemática simple para especificar las propiedades de un tipo abstracto de dato, se usará TAD(*) para abreviar.

Especificación de tipos

El párrafo TIPOS indica los tipos que se estén especificando. En general podría ser cómodo especificar varios TAD juntos, aunque para fines demostrativos este ejemplo tendrá sólo uno, *PILA*.

Un tipo es una colección de objetos caracterizados por funciones, axiomas y precondiciones. Es posible que haya confusiones por ver a los tipos como un conjunto o colección de objetos, esto tiene un sentido matemático – el tipo *PILA* es el conjunto de todas las pilas posibles, el tipo *INTEGER* es el conjunto de todos los posibles valores enteros y así sucesivamente. Sin embargo, hay un punto en el que no debe haber ninguna confusión: un tipo abstracto de dato como *PILA* no es un objeto (una pila particular) sino una colección de objetos (el conjunto de todas las pilas en este caso). No hay que olvidar cual es el objetivo real: encontrar una buena base para los módulos de los sistemas de software; basar un módulo en un objeto particular –una pila, un avión, una cuenta bancaria – no tendría sentido. El diseño O-O proporciona la capacidad de construir módulos que abarcan las propiedades de todas las pilas, todos los aviones o todas las cuentas bancarias – o en el peor de los casos de algunas pilas, aviones o cuentas.

(*) Del original en inglés ADT (*Abstrat Data Type*).

Un objeto perteneciente al conjunto de objetos descrito por la especificación de un TAD se denomina una instancia del TAD. Por ejemplo, una pila específica que satisfaga las propiedades del tipo abstracto de dato *PILA* será una instancia de *PILA*.

El párrafo TIPOS se limita a enumerar los tipos que se introducen en la especificación. En este ejemplo:

TIPOS

- *PILA [G]*

La especificación será la de un simple tipo abstracto de dato *PILA*, que describe las pilas de objetos de un tipo arbitrario *G*.

Genericidad

En *PILA[G]*, *G* denota un tipo arbitrario no especificado. *G* se denomina **parámetro genérico formal** del tipo abstracto de dato *PILA* y se dice que *PILA* es un TAD genérico. El mecanismo que permite este tipo de especificación parametrizada se conoce con el nombre de **genericidad**[2].

Es posible escribir especificaciones de TAD sin genericidad, pero al precio de repeticiones injustificadas. ¿De que sirve tener especificaciones separadas para los tipos "pila de cuentas bancarias", "pila de enteros" y demás? Estas especificaciones serían idénticas excepto allá donde se refieran explícitamente al tipo de los elementos de la pila –cuentas bancarias o enteros. Escribir estas especificaciones y llevar a cabo después la sustitución de los tipos manualmente resultaría tedioso. La reutilización es deseable no sólo para los programas sino también para las especificaciones. Gracias a la genericidad se puede hacer explícita la parametrización de tipos escogiendo algún nombre arbitrario, en este caso *G*, para representar el tipo variable de los elementos de las pilas.

Como resultado, un TAD como *PILA* no es ya exactamente un tipo, sino un patrón de tipos; para obtener un tipo de pila directamente utilizable, se debe conseguir un cierto tipo de elementos, por ejemplo *CUENTA* y hay que proporcionarlo como **parámetro genérico actual** correspondiente al parámetro formal *G*. De modo que aunque *PILA* es en sí un patrón de tipo, la notación

```
PILA [CUENTA]
```

es un tipo totalmente definido. Tal tipo, obtenido al proporcionar parámetros genéricos actuales a un tipo genérico se dice que es **derivado genéricamente**.

Las nociones que se acaban de ver se pueden aplicar recursivamente: cada tipo debiera, al menos en un principio, tener una especificación de TAD, de modo que se puede ver a *CUENTA* como si fuera en sí un tipo abstracto de dato; además, un tipo que se utiliza como parámetro genérico real de *PILA* (para producir un tipo derivado genéricamente) puede ser a su vez derivado genéricamente, de modo que es perfectamente correcto utilizar

[2] vid infra, pág. 133.

```
PILA [ PILA [CUENTA] ]
```

para especificar un cierto tipo abstracto de dato: las instancias de este tipo son pilas, cuyos elementos son a su vez pilas de cuentas bancarias; los elementos de estas últimas pilas son cuentas bancarias.

Como muestra este ejemplo, la definición precedente de "instancia" necesita de alguna calificación. Estrictamente hablando, una pila particular es una instancia no de *PILA* (el cual como se señalo es más un *patrón de tipo* que un tipo) sino de algún tipo derivado genéricamente de *PILA*, por ejemplo *PILA* [*CUENTA*]. Sin embargo, es conveniente continuar hablando sobre las instancias de *PILA* y de otros patrones de tipos similares, comprendiendo que realmente se está hablando de sus derivaciones genéricas.

De forma similar, no es totalmente preciso decir que *PILA* es un TAD: el término correcto es "patrón de TAD".

Enumerar las funciones

Después del párrafo TIPOS viene el párrafo FUNCIONES, que enumera la operaciones aplicables a las instancias del TAD. Estas operaciones serán las componentes esenciales de la definición de tipo – describen sus instancias no por lo que son sino por lo que tienen que ofrecer.

A continuación se muestra el párrafo FUNCIONES para el tipo abstracto de dato *PILA*. Para los desarrolladores de software puede resultar familiar: las líneas de este párrafo evocan las **declaraciones** en lenguajes de programación con tipos como Pascal o Ada. La línea de *new* se parece a la declaración de una variable; las otras parecen encabezamientos de rutinas.

FUNCIONES

- *put*: *PILA* [G] x G → *PILA* [G]
- *remove*: *PILA* [G] /→ *PILA* [G]
- *item PILA* [G] /→ G
- *empty*: *PILA* [G] → BOOLEAN
- *new*: *PILA* [G]

Cada línea presenta una función matemática que modela una de las operaciones sobre pilas. Por ejemplo, la función *put* representa la operación que pone un elemento en la cima de una pila.

La mayoría de los creadores de software piensan de forma natural en una operación como *put* considerándola como una función. Cuando la ejecución de un sistema de software aplica la operación *put* a una pila, esa función modificará normalmente la pila añadiéndole un elemento. En este caso *put* es una orden; una operación que puede modificar objetos.

Sin embargo, una especificación TAD es un modelo matemático y debe confiar en técnicas matemáticas bien comprendidas. En matemáticas el concepto de "cambiar algo" no existe como tal; calcular la raíz cuadrada del número 2 no modifica el valor de dicho número. Una expresión matemática se limita a definir ciertos objetos

matemáticos en términos de otros objetos matemáticos; a diferencia de la ejecución del software en una computadora, una expresión nunca cambia a un objeto matemático.

Se necesita un concepto matemático para modelar las operaciones de computadoras y es la noción de función lo que más se aproxima a esto. Una función es un mecanismo para obtener un resultado que pertenece a un cierto conjunto destino a partir de alguna posible entrada que pertenece a un cierto conjunto origen. Por ejemplo, si R denota el conjunto de los números reales, la definición de función

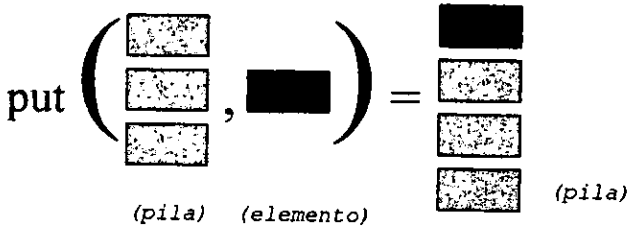
```
cuadrado_mas_uno: R → R
cuadrado_mas_uno(x)= x²+1 (para cualquier x perteneciente a R)
```

presenta una función `cuadrado_mas_uno` que tiene a R como conjunto de origen y como conjunto de destino y que produce como resultado, para cualquier entrada, el cuadrado de la entrada más uno.

La especificación de los tipos abstractos de datos usa exactamente la misma noción. La operación *put*, por ejemplo, se especifica como

```
put: PILA [G] x G → PILA [G]
```

que significa que *put* tiene dos argumentos, una *PILA* de instancias G y una instancia G y produce como resultado una nueva *PILA* [G]. (Más formalmente, el conjunto origen de una función *put* es el conjunto *PILA* [G] x G, conocido como el producto cartesiano de *PILA* [G] y G; éste es el conjunto de pares <*p*, *x*> cuyo primer elemento *p* está en *PILA* [G] y cuyo segundo elemento *x* está en G.) Ésta es una ilustración informal:



Aplicación de la función put

Con los tipos abstractos de datos sólo tenemos funciones en el sentido matemático del término; éstas no producen ni efectos laterales ni de hecho cambios de ninguna clase. Ésta es la condición que se debe satisfacer para aprovechar los beneficios del razonamiento matemático.

El papel de las operaciones modeladas por cada una de las funciones en la especificación de *PILA* se describen de la siguiente manera:

- La función *put* produce una nueva pila con un elemento adicional en la cima de la pila. La figura anterior ilustra el caso de *put* (*p*, *x*) para una pila *p* y un elemento *x*.
- La función *remove* produce una nueva pila a la que se le ha quitado el elemento que estaba en la cima, si es que había alguno; al igual que *put*, en el momento del diseño e implementación esta función debe generar un mandato (una operación que cambia al objeto y que típicamente se implementará con un procedimiento).
- La función *item* proporciona el elemento que está en la cima de la pila, si lo hay.
- La función *empty* indica si la pila está o no vacía; su resultado es un valor booleano (verdadero o falso). Se supone que el TAD *BOOLEAN* se ha definido por separado.
- La función *new* produce una pila vacía.

El párrafo FUNCIONES no define completamente estas funciones; sólo presenta sus **signaturas** –la lista de tipos de sus argumentos y resultados. La signatura de *put* es

PILA [G] x G → PILA [G]

lo cual indica que *put* admite como argumentos pares de la forma <*p*, *q*> donde *p* es una instancia de *PILA* [G]. En principio el conjunto resultado de una función (el tipo que aparece a la derecha de la flecha en la signatura, en este caso *PILA* [G]) puede a su vez ser un producto cartesiano; esto se puede utilizar para describir operaciones que proporcionen dos o más resultados.

La signatura de las funciones *remove* e *item* incluyen una flecha precedida de una diagonal (/→) en lugar de la flecha simple que se emplea en *put* y *empty*. Esta notación expresa que las funciones no son aplicables a todos los miembros del conjunto fuente.

La declaración de la función *make* aparece únicamente como

new : *PILA*

Sin flecha en la signatura. Esto es de hecho una abreviatura de

new: → *PILA*

que presenta una función sin argumentos. No hay necesidad de argumentos puesto que *new* debe proporcionar siempre el mismo resultado, una pila vacía. De modo que sólo se ha quitado la flecha para simplificar. El resultado de aplicar esta función (es decir, la pila vacía) también se escribirá en la forma *new* como abreviatura de *new*( ), que denota el resultado de aplicar *new* a una lista de argumentos vacía.

Categorías de funciones(*)

Con la especificación de un TAD para un nuevo tipo T, como *PILA* [G] en el ejemplo, se puede definir una clasificación relativa a las operaciones aplicadas a un tipo. Esta clasificación examina simplemente dónde aparece *T*, con respecto a la flecha, en la signatura de cada función:

- Una función como *new* para la cual G aparece a la derecha de la flecha es una **función de creación**. Modela una operación que produce instancias de G a partir de instancias de otros tipos – o, como en el caso de una función de creación constante como *new*, sin partir de argumento alguno. (Recuérdese que se considera que la signatura de *new* tiene una flecha implícita.)
- Una función como *item* y *empty* en las cuales G sólo aparece a la izquierda es una **función de consulta**. Modela una operación que da como resultado propiedades de instancias de G, expresadas en términos de instancias de otros tipos (*BOOLEAN* y el parámetro genérico G en los ejemplos).
- Una función como *put* y *remove* en la que G aparece en ambos lados de la flecha es una **función de orden**. Modela una operación que produce nuevas instancias de G a partir de instancias existentes de G (y posiblemente de instancias de otros tipos).

El párrafo AXIOMAS

Ya se ha visto como se describe un tipo de datos como *PILA* a través de la lista de funciones aplicables a sus instancias. Las funciones sólo se conocen por sus signaturas.

Para indicar que se tiene una pila, y no otra estructura de datos, no es suficiente la especificación del TAD tal como se ha dado hasta ahora. Cualquier estructura que sea un "dispensador", como es el caso de una cola, serviría igualmente. La selección de los nombres para las operaciones hace que esto quede bien claro: ni siquiera se tienen nombres específicos de pila tales como *push*, *pop* o *top* que nos engañen haciéndonos pensar que hemos definido pilas y nada más que pilas.

Por supuesto que esto no debe sorprendernos ya que el párrafo FUNCIONES declara las funciones (en la misma forma en que una unidad de programa puede declarar una variable) pero no las define completamente. En una definición matemática como la del ejemplo anterior

```
cuadrado_mas_uno: R → R
cuadrado_mas_uno (x) = x² + 1      (para cualquier x en R)
```

(*) Una terminología alternativa llama a estas categorías: "constructor", "función de acceso" y "modificador". Los términos que se han utilizado aquí están más directamente relacionados con la interpretación de las funciones de los TAD como modelos de operaciones aplicables a objetos software y llegarán hasta las características de una clase, los equivalentes software de las funciones matemáticas.

La primera línea hace las veces de la declaración de la signatura, pero hay una segunda línea que define el valor de la función. ¿Cómo hacer esto mismo para las funciones de un TAD?

Aquí no se debería emplear una definición explícita del estilo de la segunda línea de la definición de `cuadrado_mas_uno`, porque esto nos obligaría a escoger una representación – y lo que se pretende es protegerse contra las opciones de representación.

Para estar seguros de que se entiende el aspecto que tiene una definición explícita, se escribirá una para la representación de la pila *ARRAY_ASCENDENTE* que se esbozó anteriormente. En términos matemáticos escoger *ARRAY_ASCENDENTE* significa que se considera una instancia de *PILA* como un par *<cantidad, representación>*, en donde *representación* es el array y *cantidad* es el número de elementos que se han guardado en la pila. En este caso una definición explícita de *put* es (para cualquier instancia *x* de G):

put (<cantidad, representación>, x) = < cantidad + 1, representación [cantidad+1: x]>

donde la notación *a[n: v]* denota el array obtenido a partir de *a* al cambiar el valor del elemento de índice *n* para que pase a ser *v*, y dejando los demás elementos como están (si los hay).

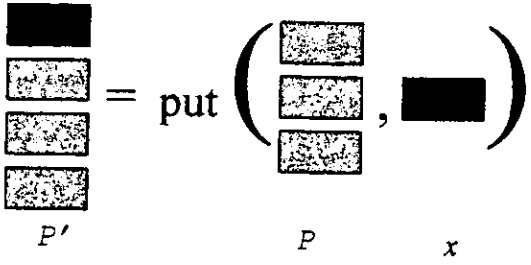
Esta definición de la función *put* es tan sólo una versión matemática de la implementación de la operación *put* que se ha esbozado empleando la notación de Pascal, junto a la representación *ARRAY_ASCENDENTE*, en la figura de las posibles representaciones de las pilas. Pero no se desea en ningún momento hacer compromisos con ninguna representación física.

Dado que una definición explícita nos obligaría a escoger una representación, hay que volver a las definiciones **Implícitas**. Es necesario abstenerse de dar los valores de las funciones de una especificación TAD; en su lugar se enunciarán las propiedades de esos valores – todas las propiedades que sean importantes, pero únicamente esas propiedades. El párrafo AXIOMAS enuncia estas propiedades. Para *PILA* serán:

AXIOMAS

- Para cualquier `x:G, p: PILA [G]`
- A1. *item(put (p, x))* = x
- A2. *remove(put (p, x))* = p
- A3. *empty(new)*
- A4. **not** *empty(put (p, x))*

Los dos primeros axiomas expresan la propiedad básica de LIFO (el último que entra es el primero que sale) de las pilas. Para comprenderlo mejor, supóngase que tenemos una pila *p* y una instancia *x* y se define *p'* como el resultado de insertar *x* en *p*. Adaptando una de las figuras anteriores:



Aquí el axioma A1 dice que la cima de p' es x, el último elemento que se insertó en la pila; el axioma A2 dice que si se quita el elemento que está en la cima de p' se volverá a la pila p que se tenía antes de poner x. Estos dos axiomas proporcionan una descripción concisa de la propiedad fundamental de las pilas en términos puramente matemáticos, sin recurrir a ningún razonamiento imperativo ni a propiedades de la representación.

Los axiomas A3 y A4 nos dicen cuando está o no vacía una pila: una pila resultante de la función de creación *new* está vacía; toda pila resultado de la operación de insertar un elemento en alguna pila existente (vacía o no) no está vacía.

Estos axiomas, al igual que los otros, son predicados (en el sentido de la lógica) que expresan que una cierta propiedad es siempre verdadera para cada valor posible de p y x. Ciertas personas prefieren leer los axiomas A3 y A4 en la forma equivalente

```
Para cualquier x:G, p: PILA [ G ]
A3'. empty (new)=true
A4'. empty (put (p, x,))=false
```

bajo la cual se pueden ver también dichos axiomas, al menos informalmente, como definición la función *empty* por inducción sobre el tamaño de las pilas.

Las especificaciones de los TAD son implícitas. Pero hay dos formas de ser implícito:

- El método TAD define un conjunto de objetos implícitamente, a través de las funciones aplicables. Esto se describía antes como definir los objetos por lo que tienen, no por lo que son. Más precisamente, la definición nunca implica que las operaciones enumeradas sean las únicas; cuando se llega a la representación es frecuente que se añadan otras operaciones.
- Las funciones en sí mismas también se definen implícitamente: en lugar de definiciones explícitas (como la que se usó para el `cuadrado_mas_uno` y para un intento inicial de definir *put* mediante una referencia a una representación matemática) se utilizan axiomas que describen las propiedades de las funciones. Tampoco aquí se dice que esto sea exhaustivo: cuando finalmente se implementen las funciones, no cabe duda de que se adquirirán más propiedades.

Este aspecto implícito es una pieza clave de los tipos abstractos de datos y, por implicación de sus futuros homólogos en la construcción de software orientado a objetos – *las clases*. Cuando se define un tipo abstracto de dato o una clase, siempre se habla *acerca* del tipo o de la clase: nos limitamos a enumerar las propiedades que se conocen y las tomamos como definición. Nunca suponemos que éstas sean las únicas propiedades aplicables.

Lo implícito expresa una naturaleza abierta: siempre debe ser posible añadir nuevas propiedades a una clase o a un TAD. El mecanismo básico para llevar a cabo tales extensiones sin perjudicar los usos ya existentes de la forma original es la herencia.

Funciones parciales

La especificación de cualquier ejemplo realista, incluso tan básico como las pilas, debe encontrarse con el problema de las operaciones indefinidas: algunas operaciones no son aplicables a todos los elementos de sus conjuntos origen. Éste es el caso de *put* y de *item*: no se puede quitar un elemento de una pila vacía; y una pila vacía no tiene cima.

La solución utilizada en la especificación precedente consiste en describir estas funciones como parciales. Una función de un conjunto origen X a un conjunto resultado Y es parcial si no está definida para todos los miembros de X. Una función que no es parcial se denomina **total**. Un ejemplo simple de función parcial en la aritmética habitual es *inv*, la función inversa de números reales, cuyo valor para cualquier número real apropiado es

$$inv(x) = 1 / x$$

Dado que *inv* no está definida para x=0 se puede decir que es una función parcial sobre R, el conjunto de los números reales: *inv*: **R** /→ **R**.

Para indicar que una función puede ser parcial, la notación hace uso de una flecha precedida de una diagonal /→; la flecha normal → se reserva para las funciones de las que se tiene garantía de que son totales.

La especificación del TAD *PILA* aplica estas ideas a las pilas declarando *put* e *item* como funciones parciales en el párrafo FUNCIONES, tal y como se indica mediante la flecha con diagonal que aparece en sus signaturas.

En algunos casos pudiera ser deseable describir también *put* como una función parcial; esto es necesario para modelar implementaciones tales como *ARRAY_ASCENDENTE* y *ARRAY_DESCENDENTE*, que sólo admiten un número finito de operaciones consecutivas de *put* para una pila dada. Esta aplicación de las funciones parciales reflejan las restricciones de implementación.

Precondiciones

Las funciones parciales son un hecho del que uno no puede evadirse en la vida del desarrollo del software, ya que reflejan simplemente la observación de que no toda operación es aplicable a cualquier objeto. Pero son también una fuente potencial de errores: si f es una función parcial de X en Y, no se puede estar seguro de si la expresión $f(e)$ tiene o no sentido aunque el valor de e esté en X; también tenemos que ser capaces de garantizar que el valor pertenece al dominio de f.

Para que esto sea posible, cualquier especificación de un TAD que incluya funciones parciales debe especificar el dominio de cada una de ellas. Éste es el papel del párrafo de PRECONDICIONES.

Para el caso de *PILA* el párrafo sería:

PRECONDICIONES

- *remove* (*p*: *PILA*[G]) **require not** *empty*(*p*)
- *item* (*p*: *PILA*[G]) **require not** *empty*(*p*)

en donde, para cada función, la cláusula **require** indica qué condiciones tienen que satisfacer los argumentos de las funciones para pertenecer al dominio de las funciones.

La expresión booleana que define el dominio se llama **precondición** de la función parcial correspondiente. Aquí la precondición tanto de *remove* como de *item* expresa que la pila que se dé como argumento no debe estar vacía. Antes de la cláusula **require** viene el nombre de la función con nombres para los argumentos (*p* para el argumento pila en el ejemplo), de modo que la precondición se puede referir a éstos.

La especificación completa

Con el párrafo PRECONDICIONES se concluye esta sencilla especificación del tipo abstracto de dato *PILA*. Para facilitar su estudio es útil agrupar aquí los componentes de la especificación, que se han visto en esta sección por separado. Ésta es la especificación completa:

ESPECIFICACIÓN DE PILAS MEDIANTE TAD

TIPOS

- *PILA*[G]

FUNCIONES

- *put*: *PILA* [G] x G → *PILA*·[G]
- *remove*: *PILA* [G] /→ *PILA* [G]
- *item PILA* [G] /→ G
- *empty*: *PILA* [G] → *BOOLEAN*
- *new*: *PILA* [G]

AXIOMAS

Para cualquier *x*:G , p: *PILA* [G]

- A1. *item*(*put* (*p*, *x*)) = x
- A2. *remove*(*put* (*p*, *x*)) = p
- A3. *empty*(*new*)
- A4. **not** *empty*(*put* (*p*, *x*))

PRECONDICIONES

- *remove* (*p*: *PILA*[G]) **require not** *empty*(*p*)
- *item* (*p*: *PILA*[G]) **require not** *empty*(*p*)

La potencia de las especificaciones de los tipos abstractos de datos proviene de su capacidad de captar las propiedades esenciales de las estructuras de datos sin incurrir en una especificación excesiva. La especificación de una pila resumida anteriormente expresa todo lo que hay que saber acerca de la noción de pila en general, excluyendo todo lo que pueda ser aplicable a alguna representación concreta de las pilas.

3.1.5. De los tipos abstractos de datos a las clases

Los tipos abstractos de datos (TAD) proporcionan una elegante teoría matemática para el modelado de estructuras de datos y de hecho también de programas en general. En la búsqueda de una buena estructura modular basada en los tipos de objetos, los tipos abstractos de datos proporcionan un mecanismo de descripción de alto nivel, libre de cuestiones de implementación. Esto nos lleva a las estructuras fundamentales de la tecnología de objetos: **las clases**.

Clases

Los TAD servirán como base directa de los módulos, cuyas propiedades de analizaron en el capítulo II. Lo que se pretende es que un sistema orientado a objetos se construya (en el nivel de análisis, diseño o implementación) como una colección de TAD que interactúan entre sí, parcial o totalmente implementados. La noción básica en este caso es la de **clase**:

CLASE

Una clase es un tipo abstracto de dato equipado con una implementación posiblemente parcial.

De modo que para obtener una clase se debe proporcionar un TAD y hay que decidir una implementación para el mismo. El TAD es un concepto matemático; la implementación es la versión orientada a computadoras. Sin embargo, la definición establece que la implementación puede ser parcial; la terminología siguiente separa este caso de las clases que están completamente implementadas:

CLASE DIFERIDA Y CLASE EFECTIVA

Una clase que está completamente implementada se denomina efectiva. Una clase que está implementada sólo parcialmente, o que no está implementada, se dice que es diferida. Una clase puede ser o bien efectiva o bien diferida.

Para obtener una clase efectiva, se deben dar todos los detalles de implementación. Para una clase diferida, se puede escoger un cierto estilo de implementación pero dejar abiertos algunos aspectos de la implementación. En el caso más extremo de implementación "parcial" puede uno abstenerse de tomar ninguna decisión de implementación; la clase resultante será completamente diferida y equivalente a un TAD.

Cómo producir clases efectivas

Considérese primero el caso de las clases efectivas. ¿Qué se necesita para implementar un TAD? La clase efectiva resultante constará de tres elementos:

- **E1**. La especificación de un TAD (un conjunto de funciones con sus axiomas y precondiciones asociadas, que describen las propiedades de las funciones).
- **E2**. La selección de una representación.
- **E3**. Una correspondencia de las funciones (E1) con la representación (E2) en la forma de un conjunto de mecanismos, o **características** (*features*), cada una de las cuales implementa una de las funciones en términos de la representación, de modo que se satisfagan los axiomas y las precondiciones. Muchas de esas características serán rutinas (subprogramas) en el sentido usual, aunque también pueden aparecer algunas que sean campos de datos o "atributos".

Por ejemplo, si el TAD es *PILA*, se puede escoger como representación (paso E2) la solución que se denominaba *ARRAY_ASCENDENTE*, y que implementa una pila mediante el par

<representación, cantidad>

donde *representación* es un array y *cantidad* es un entero. Para las implementaciones de las funciones (E3) se tendrán características que correspondan a *put*, *remove*, *item*, *empty* y *new*, que logran los efectos correspondientes; por ejemplo se puede implementar *put* mediante una rutina de la forma

```
put (x: G) is
        --Inserta x en la cima de la pila.
        --(no comprueba un desbordamiento de la pila)
    do
        cantidad:= cantidad + 1
        representación [cantidad]:= x
    end
```

La combinación de los elementos obtenidos dentro de E1, E2 y E3 producirá una clase, la estructura modular de la tecnología de objetos.

El papel de las clases diferidas

Para que una clase sea efectiva tiene que estar presente toda la información de implementación (E1 y E2 anteriores). Si falta algo, la clase es diferida.

Cuanto más diferida es una clase, más cerca se está de un TAD, pero *aderezada* con una sintaxis que seduce a los que desarrollan software en lugar de a los matemáticos. Las clases diferidas son particularmente útiles para el análisis y el diseño:

- En el análisis orientado a objetos ni se necesitan ni se desean detalles de implementación: el método usa las clases sólo por su poder descriptivo.

- En el diseño orientado a objetos pueden quedar fuera muchos aspectos de la implementación; el diseño debe concentrarse en las propiedades de alto nivel de la arquitectura del sistema – qué funcionalidades proporciona cada módulo y no cómo las proporciona.
- A medida que el diseño se va acercando gradualmente a la implementación completa, se irán añadiendo más y más propiedades de implementación hasta tener una clase efectiva.

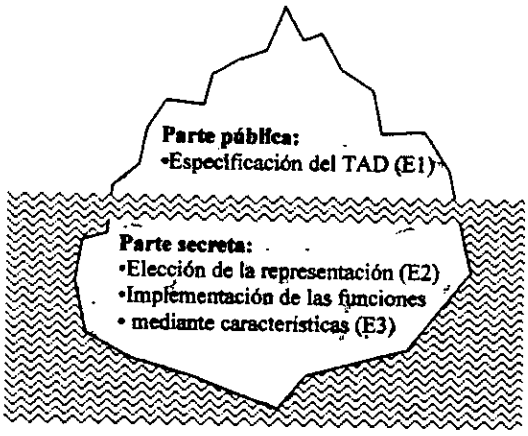
Pero el papel de las clases diferidas no se detiene aquí, pues incluso en un sistema completamente implementado se pueden encontrar muchas de ellas. Algunos de estos papeles provienen de sus aplicaciones previas: si se ha partido de clases diferidas para obtener las efectivas, se hará bien el conservar las primeras en la forma de clases antecesoras (en el sentido de la herencia) de modo que se puedan servir más tarde como memoria viviente del proceso de análisis y diseño.

Muy a menudo, en el software producido con enfoques no orientados a objetos, la forma final de un sistema no contiene ningún registro del esfuerzo considerable que se ha desarrollado para llegar a él. Si se le pide a alguien que lleve a cabo el mantenimiento del sistema – extenderlo, portarlo a otras plataformas, depurarlo – tratar de entender el sistema sin ningún registro es tan difícil como pudiera ser para un medico encontrar un tratamiento sin contar con estudios clínicos. Conservar las clases diferidas en el sistema final es una de las mejores maneras de mantener los registros necesarios.

Las clases diferidas también tienen aplicaciones puramente relacionadas con la implementación. Sirven para clasificar grupos de tipos de objetos relacionados, proporcionan algunos de los más importantes módulos reutilizables de alto nivel, capturan comportamientos comunes dentro de un conjunto de variantes y desempeñan un papel clave (en relación con el polimorfismo y la ligadura dinámica) para garantizar que las arquitecturas software sigan siendo descentralizadas y flexibles.

Tipos abstractos de datos y ocultación de información

Una consecuencia particularmente interesante de la política orientada a objetos consistente en basar todos los módulos en implementaciones de TAD (clases) es que esto proporciona una respuesta clara a la pregunta que quedó pendiente en la exposición sobre ocultación de la información: ¿Cómo seleccionar las características públicas y privadas de un módulo? – las partes visibles e invisibles del iceberg.



La visión TAD de un módulo con ocultación de información

Si el módulo es una clase que proviene de un TAD como se indicaba anteriormente, la respuesta está clara: de las tres partes implicadas en la transición, E1, la especificación del TAD, es la parte pública; E2 y E3, la representación escogida y la implementación de las funciones del TAD en términos de esta representación, deben ser secretas.

De modo que el uso de los tipos abstractos de datos como fuente de los módulos nos ofrece una guía práctica y sin ambigüedades para aplicar el principio de ocultación de información en nuestros diseños.

Introducción a un punto de vista más imperativo

La transición de los tipos abstractos de datos a las clases involucra una diferencia estilística importante: la introducción del concepto de cambio y de un razonamiento imperativo.

Como ya se expuso, la especificación de los tipos abstractos de datos está libre de cambios, o bien, para utilizar un término de la teoría de las Ciencias de la Computación, es *aplicativa*. Todas las características de un TAD se modelan mediante funciones matemáticas; esto incluye las funciones de creación, las de consulta y las órdenes. Por ejemplo, la operación de poner en el caso de las pilas se modela mediante la función orden

```
put: STACK [G] x G → PILA[G]
```

que especifica una operación que proporciona una nueva pila, en lugar de cambiar la pila ya existente.

Las clases, que están más cerca del mundo del diseño y de la implementación, abandonan esta visión solamente aplicativa y vuelven a presentar las órdenes como operaciones que pueden modificar los objetos.

Por ejemplo, *put* aparecerá como una rutina que admite un argumento de tipo G (el parámetro formal genérico) y modifica una pila insertando un nuevo elemento en la cima – en lugar de producir una nueva pila.

Este cambio de estilo refleja el estilo imperativo que prevalece en la construcción de software. (La palabra "operacional" también se utiliza como sinónimo de "imperativo".) Esto requerirá los cambios correspondientes en los axiomas de los TAD. Los axiomas A1 y A4 de las pilas, que anteriormente aparecían en la forma

- A1. *item* (*put*(*p*, *x*)) = *x*
- A2. **not** *empty* (*put*(*p*, *x*))

producirán, en la forma imperativa, una cláusula conocida como **postcondición de la rutina** que se presenta mediante la palabra clave **ensure** (*asegura*)

```
put (x: G) is
            --inserta x en la cima de la pila
      require
            … la precondición, si hay alguna …
do
            …la implementación apropiada, si es conocida …
      ensure
            item = x
            not empty
      end
```

Aquí la postcondición expresa que al retornar de una llamada a la rutina *put*, el valor de *item* será *x* (el elemento que se ha insertado) y que el valor de *empty* será falso.

3.1.6. Precondiciones y postcondiciones

La utilidad que demostraron los axiomas y precondiciones en la especificación de los TAD, hace necesario conocer la forma en que ésta valiosa información puede ser incluida dentro del texto de las clases. Esto con el fin de lograr la corrección de los elementos software, entendiendo por corrección del software la consistencia entre su implementación y su especificación. Para la mayoría de la comunidad del software ésta es aún una idea extraña: estamos acostumbrados a programas que definen las operaciones que le ordenan a nuestra máquina hardware-software ejecutar lo que para nosotros es *el cómo*; y es menos común tratar de describir los propósitos del software (*el qué*) como parte del software en sí.

Para expresar la especificación confiaremos en las aserciones. Una aserción es una expresión que involucra algunas entidades del software y que establece una propiedad que dichas entidades deben satisfacer en ciertas etapas de la ejecución del software. Una típica aserción puede ser la que expresa que un entero es positivo o la que expresa que una referencia no es vacía.

El primer uso de las aserciones es la especificación semántica de las rutinas. Una rutina no es solamente un trozo de código; como implementación de alguna función de la especificación de un tipo abstracto de dato, debería realizar alguna tarea

útil. Es necesario expresar esta tarea con precisión, como ayuda para el diseño (no se puede aspirar a asegurar que una rutina sea correcta a menos que se haya especificado lo que se supone que hace) y como ayuda para la comprensión posterior del texto.

Se puede especificar la tarea que lleva a cabo una rutina mediante dos aserciones asociadas a la rutina: una precondición y una postcondición. La precondición establece las propiedades que se tienen que cumplir cada vez que se llame a la rutina; la postcondición establece las propiedades que debe garantizar la rutina cuando retorne.

La clase pila

Un ejemplo permitirá familiarizarse con el uso práctico de las aserciones. En este capítulo ya se han introducido las bases para una clase *PILA*, se puede proponer una clase bajo siguiente forma:

```
class STACK[G] feature
      ... Declaración de características:
      count, empty, full, item, put, remove
end
```

A continuación aparece una implementación. Antes de considerar cuestiones de implementación es importante observar que las rutinas están caracterizadas por fuertes propiedades semánticas que son independientes de una representación específica. Por ejemplo:

- Las rutinas *remove* e *item* sólo son aplicables si el número de elementos no es cero.
- *put* aumenta el número de elementos en uno; *remove* lo decrementa en uno.

Tales propiedades son parte de la especificación de un tipo abstracto de dato e incluso la gente que no use ningún enfoque remotamente tan formal como los TAD las contempla implícitamente; pero en los enfoques más comunes de construcción de software los textos software no revelan rastro de ellas. Mediante las precondiciones y postcondiciones de las rutinas se puede convertirlas en elementos explícitos del software.

Las precondiciones y postcondiciones se expresarán como cláusulas de las declaraciones de las rutinas introducidas por las palabras clave **require** (*requiere*) y **ensure** (*asegura*) respectivamente. Para la clase pila, dejando en blanco la implementación de las rutinas, está nos da:

```
indexing
      description: "Pilas estructura dispensadora con política LIFO"
class STACK[G] feature --Acceso
      count: INTEGER
            --Número de elementos que hay en la pila
      item: G is
            --Elemento en la cima
        require
```

```
            not empty
          do
            ...
          end
feature           --Informar sobre el estado
      full: BOOLEAN is
            --¿Está llena la representación de la pila?
          do ... end
      empty: BOOLEAN is
            --¿No hay elementos en la pila?
          do ... end
feature           --Cambios en los elementos
      put (x:G) is
            --Añade un elemento a la cima
         require
            not full
         do
           ...
         ensure
            not empty
            item=x
            count = old count + 1
          end
      remove is
            --Quita el elemento que está en la cima
         require
            not empty
         do
            ...
         ensure
            not full
            count = old count - 1
         end
end         --clase PILA
```

Precondiciones

Una precondición expresa las restricciones bajo las que una rutina funcionará correctamente. En este ejemplo:

- No se puede llamar a *put* si la representación de la pila está llena.
- No se pueden aplicar ni *remove* ni *item* a una pila que esté vacía.

La precondición se aplica a todas las llamadas a la rutina, tanto dentro de la clase como desde los clientes. Un sistema correcto nunca ejecutará una llamada en un estado en que no se satisfaga la precondición de la rutina a la que se llama.

Postcondiciones

Una postcondición expresa propiedades del estado resultante de la ejecución de una rutina. Aquí:

- Después de *put* la pila no puede estar vacía, el elemento que se encuentra en la cima es precisamente el que se acaba de añadir y la cantidad de elementos ha quedado incrementada en uno.
- Después de un *remove* la pila no puede estar llena y el número de elementos ha disminuido en uno.

La presencia de una cláusula de postcondición en una rutina expresa una garantía por parte de quién implementa la rutina de que ésta producirá un estado en el que se satisfagan ciertas propiedades si se supone que ésta ha sido invocada satisfaciéndose la precondición.

En las postcondiciones se puede utilizar una notación especial, **old**; *put* y *remove* la utilizan para expresar los cambios de *count*. La notación **old** *e*, en donde *e* es una expresión (en la mayoría de los casos prácticos será un atributo), denota el valor de *e* antes de entrar a la rutina. En una postcondición toda aparición de *e* que no esté precedida por **old** denota el valor de esta expresión al salir de la rutina. La postcondición de *put* incluye la cláusula

```
count = old count +1
```

para establecer que *put*, cuando se aplica a algún objeto, debe incrementar en uno el valor del campo *count* de dicho objeto.

3.1.7. Invariantes de clase

Las precondiciones y las postcondiciones describen las propiedades de las rutinas individuales. También hay necesidad de expresar las propiedades globales de las instancias de una clase, que deben ser preservadas por todas las rutinas. Tales propiedades constituyen el invariante de la clase y capturan las propiedades semánticas más profundas y restricciones de integridad que caracterizan a una clase.

Definición y ejemplo

Considérese de nuevo el ejemplo inicial de pilas por arrays (*STACK*):

```
class STACK(G) creation
      make
feature
      ... make, empty, full, item, put, remove ...
      capacidad: INTEGER
      count: INTEGER
      representacion: ARRAY[G]      --Implementación
end
```

Los atributos de la clase – el array *representación* y los enteros *capacidad* y *count* – constituyen la representación de la pila. Aunque las precondiciones y postcondiciones de las rutinas, dadas anteriormente, expresan algunas de las propiedades semánticas de las pilas, éstas fallan al expresar otras propiedades de

consistencia importantes que enlazan a los atributos. Por ejemplo, *count* debiera estar siempre entre 0 y *capacidad*:

```
0 <= count; count <= capacidad
```

(lo que implica también que *capacidad* >= 0), y *capacidad* debe ser el tamaño del array:

```
capacidad = representacion.capacidad
```

Un invariante de clase es una aserción, que expresa restricciones de consistencia generales que se aplican a cada instancia de la clase como un todo, es diferente de las precondiciones y las postcondiciones, que caracterizan a rutinas individuales.

Las aserciones anteriores involucran sólo atributos. Los invariantes pueden expresar también las relaciones semánticas entre funciones o entre funciones y atributos. Por ejemplo el invariante para *STACK* pudiera incluir la propiedad siguiente que describe la conexión entre *empty* y *count*:

```
empty = (count = 0)
```

En este ejemplo, la aserción invariante liga un atributo con una función; no es particularmente interesante ya que simplemente repite una aserción que aparece en la postcondición de la función (*empty* en este caso). Las aserciones más útiles son las que involucran o bien sólo a atributos, como las anteriores, o bien a más de una función.

Veamos otro ejemplo típico. Supóngase – en línea con los ejemplos previos que tenían que ver con la noción de cuenta bancaria – que se tiene una *CUENTA_BANCARIA* con las características *lista_ingresos*, *lista_retiros* y *saldo*. El invariante de esa clase podría incluir una cláusula de la forma:

```
saldo_consistente: lista_ingresos.total - lista_retiros.total = saldo
```

donde la función *total* da el valor acumulado de la lista de operaciones (ingresos o retiros). Esto enuncia la condición de consistencia básica entre los valores a los que se puede acceder a través de las características *lista_ingresos*, *lista_retiros* y *saldo*.

3.1.8. Condensando el conocimiento

Si se sigue el desarrollo de este trabajo, desde la sección dedicada a la modularidad y se llega hasta este punto, se puede observar que se ha comenzado con el objetivo de obtener estructuras lo más modulares posibles; distintos argumentos nos llevaron a la conclusión de que los objetos, o para ser más precisos, los tipos de objetos, proporcionarían una base mejor que sus competidores tradicionales – la funciones. Esto planteó la siguiente pregunta: cómo se describen los tipos de objetos. La respuesta vino en forma de tipos abstractos de datos (y sus sustitutos prácticos, las

clases), esto significa que debemos basar la descripción de los datos en *las funciones que se les pueden aplicar*. Esto podría tomarse como una contradicción, pero no es así:

- Los tipos de objetos, representados por los TAD y las clases, siguen siendo las bases indiscutibles de la modularización.
- No es sorprendente que tanto los aspectos de objeto como los de funciones aparezcan en la arquitectura final del sistema: ninguna descripción de software estará completa si le falta uno de estos dos componentes.
- Lo que distingue fundamentalmente los métodos orientados a objetos de los enfoques más antiguos es la distribución de los papeles: los tipos de objetos son los ganadores indiscutibles cuando llega el momento de seleccionar el criterio para construir módulos. Las funciones quedan como sus servidores.
- En la descomposición orientada a objetos, ninguna función existe por sí sola: toda función está ligada a algún tipo de objeto. Esto se lleva a los niveles de diseño e implementación: *ninguna característica existe por sí sola; toda característica esta asociada a alguna clase.*

Construcción de software orientado a objetos

El estudio de los tipos abstractos de datos ha dado la respuesta a la pregunta: cómo describir los tipos de objetos que servirán de columna vertebral para una arquitectura de software correcta y flexible.

Luego entonces es posible dar una definición de lo que puede considerase el proceso de construcción de software orientado a objetos.

CONSTRUCCIÓN DE SOFTWARE ORIENTADO A OBJETOS

La construcción de software orientado a objetos es la construcción de sistemas software como colecciones estructuradas de implementaciones, posiblemente parciales, de tipos abstractos de datos.

Ésta definición cuenta con elementos esenciales de la teoría que fundamenta el método de orientación a objetos:

- La base es el concepto de ***tipo abstracto de dato***.
- Para el software no se necesitan los TAD como tales, que son un concepto matemático, sino la ***implementación*** de los TAD, que es un concepto de software.
- Sin embargo, estas implementaciones no necesitan estar completas; la indicación "***posiblemente parciales***" alude a las clases diferidas – incluyendo el caso extremo de una clase completamente diferida, que no tenga implementada ninguna característica.
- Un sistema es una ***colección de clases***, en la que ninguna está particularmente a cargo del sistema –no hay una cima o programa principal.
- La colección es ***estructurada*** gracias a dos relaciones entre clases: cliente y herencia.

3.2. El papel de las clases

Al momento ya se conocen las limitaciones de los enfoques tradicionales: arquitecturas centralizadas que limitan la flexibilidad. Por otro lado, la reutilización y la extensibilidad son las principales razones que motivan la exigencia de un mejor enfoque para el diseño modular. Ahora, con el conocimiento de los tipos abstractos de datos se sientan las bases para encontrar los bloques básicos para la construcción de software: **las clases**.

Los tipos abstractos de datos son un concepto matemático, deseable para la etapa de especificación (que también se denomina análisis). Debido a que introduce implementaciones, parciales o totales, el concepto de clase establece la conexión necesaria con la construcción de software – diseño e implementación. Recuérdese que una clase se dice efectiva si la implementación es total, de lo contrario es diferida.

Las clases son la noción básica de la que deriva todo en la tecnología de objetos, a pesar de que aún en la actualidad se piense que los objetos son el principio y fin de la dicha tecnología. Para aclarar este punto es necesario recapitular en algunos aspectos:

Al igual que un TAD, una clase es un tipo: describe un conjunto de posibles estructuras de datos llamadas instancias de la clase. Los tipos de datos abstractos también tienen instancias; la diferencia es que una instancia de un TAD es un elemento puramente matemático, mientras que una instancia de una clase es una estructura de datos que puede ser representada en la memoria de una computadora y manipulada por un sistema software.

Por ejemplo si se ha definido una clase `PILA` tomando la especificación de TAD de la sección anterior y añadiéndole la representación adecuada de la información, las instancias de esa clase serán estructuras de datos que representan pilas individuales.

Otro ejemplo podría ser una clase `PUNTO` que modele la noción de un punto en el espacio bidimensional bajo una representación apropiada; una instancia de esta clase sería una estructura de datos que represente a un punto. Dicha representación podría construirse con un registro de dos campos que almacene las coordenadas horizontal y vertical, `x` e `y`, de un punto.

La definición de clase genera como subproducto la definición de "**objeto**". Un objeto es simplemente una instancia de alguna clase. Por ejemplo una instancia de la clase `PILA` – una estructura de datos que representa una pila particular – es un objeto; de modo que una instancia de la clase `PUNTO`, representa a un punto particular del espacio bidimensional.

Por lo tanto, los textos de software que sirven para producir los sistemas son clases. *Los objetos son sólo un concepto de tiempo de ejecución: son creados y manipulados por el software durante su ejecución.*

Esta sección se centrará en las clases, para analizar su estructura, caracteristicas e importancia para la construcción de sistemas software, que aprovechan su potencia y generalización para lograr sistemas extensibles y reutilizables.

Primeramente para entender el enfoque orientado a objetos es esencial comprender que las clases desempeñan dos papeles que los enfoques anteriores a la O-O siempre han tratado pór separado: *módulo y tipo.*

Módulos y tipos

Los lenguajes de programación y demás notaciones que se emplean en el desarrollo de software (los lenguajes de diseño, los lenguajes de especificación, las notaciones gráficas para el análisis) incluyen siempre alguna facilidad modular y algún sistema de tipos.

Un módulo es una unidad de descomposición de software. En la exposición de modularidad se estudiaron formas de módulos, como las rutinas y los paquetes. Independientemente de la estructura exacta escogida, se puede decir que la noción de módulo es un concepto **sintáctico**, puesto que la descomposición en módulos sólo afecta a la forma de los textos software, no a lo que el software pueda hacer; de hecho es posible escribir programa, por ejemplo en Ada, como un único paquete, o en Pascal como un único programa principal. Un enfoque como este no es recomendable, por supuesto, y cualquier persona competente en el desarrollo de software usará las facilidades modulares del lenguaje para descomponer su software en partes manejables. Pero si se toma un programa existente, por ejemplo en Pascal, siempre se pueden fusionar todos los módulos en uno sólo y se seguiría teniendo un sistema con la misma semántica. De modo que la práctica de descomponer en módulos está dictada por principios de ingeniería y de gestión de proyectos más que por una necesidad intrínseca.

A primera vista, los tipos son un concepto bastante diferente. Un tipo es una descripción estática de ciertos objetos dinámicos: los diferentes elementos de datos que serán procesados durante la ejecución del sistema software. El conjunto de tipos incluye tipos predefinidos como `INTEGER` y `CHARACTER` así como tipos definidos por el desarrollador: tipos registro (también conocidos como tipos estructura), tipos puntero, tipos conjunto (como en Pascal), tipos array y otros. La noción de tipo es un concepto **semántico**, ya que todo tipo influye directamente en la ejecución de un sistema software al definir la forma de los objetos que creará el sistema y manejará durante la ejecución.

La clase como módulo y como tipo

En los enfoques no O-O el concepto de módulo y el de tipo permanecen separados. La propiedad más destacada de la noción de clase es que subsume estos dos conceptos, uniéndolos en una única contracción lingüística. Una clase es un módulo o una unidad de descomposición de software; pero también es un tipo (o, en los casos que involucran genericidad, un patrón de tipo).

Gran parte de la potencia de los métodos orientados a objetos proviene de esta identificación. En particular la *herencia* sólo podrá entenderse por completo si se observa que proporciona a la vez una extensión de módulos y una especialización de tipos.

Lo que no está claro aún es *la forma en que* es posible en la práctica unificar estos conceptos que en principio parecen tan distantes. A lo largo de las siguientes secciones se proporcionarán la información necesaria para entender como el método orientado a objetos logra unificar estos conceptos.

3.2.1. Un sistema de tipos uniforme

Un aspecto importante que propone el enfoque O-O es la sencillez y la uniformidad del sistema de tipos; que se deriva de una propiedad fundamental:

REGLA DE LOS OBJETOS

Todo objeto es una instancia de alguna clase.

La regla de los objetos no sólo se aplicará a los objetos compuestos definidos por el desarrollador (tales como estructuras de datos con varios campos) sino también a los objetos básicos tales como los enteros, números reales, valores booleanos y caracteres, los cuales serán considerados todos ellos instancias de clases de bibliotecas predefinidas (`INTEGER`, `REAL`, `DOUBLE`, `BOOLEAN`, `CHARACTER`).

Este deseo por hacer de todo posible valor, aunque sea simple, una instancia de alguna clase puede parecer al principio exagerado o incluso extravagante. Después de todo, los matemáticos y los ingenieros han usado con éxito los enteros y los reales durante mucho tiempo, sin saber que estaban manejando instancias de clases. Pero la insistencia en la uniformidad tiene su compensación por varias razones:

- Siempre resulta deseable tener un marco simple y uniforme en lugar de muchos casos especiales. Aquí el sistema de tipos estará basado enteramente en la noción de clase.
- Describir los tipos básicos como TAD y por tanto clases es simple y natural. No es difícil, por ejemplo, ver la forma de definir la clase `INTEGER` con características que abarquen operaciones aritméticas tales como "+", operaciones de comparación tales como " <=" y las propiedades asociadas, derivadas de los axiomas matemáticos correspondientes.
- Al definir a los tipos básicos como clases, se les permite participar en todos los aspectos O-O, especialmente en la herencia y la genericidad. Si no se tratasen los tipos básicos como clases habría que introducir estrictas limitaciones y muchos casos especiales.

Con una buena implementación, no hay que temer las consecuencias negativas de la decisión de definir a todos los tipos a partir de clases. Nada le impide a un compilador tener un conocimiento especial sobre las clases básicas de modo que el código que genere para las operaciones relativas a valores de tipos como `INTEGER` y `BOOLEAN` puedan ser tan eficientes como si éstos fueran tipos integrados en el lenguaje.

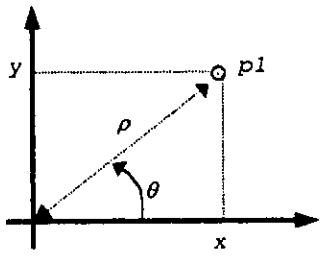
3.2.2. Una clase sencilla

Ahora se verá el aspecto que tiene una clase, mediante el estudio de un ejemplo sencillo pero típico, que muestra muchas de las propiedades fundamentales aplicables a casi todas las clases.

Al momento de implementar una clase en algún lenguaje de programación particular deberá tomarse en cuenta la capacidad de la herramienta o entorno de desarrollo para representar dichas propiedades, así como las facilidades adicionales que ofrezca para el manejo de clases a nivel de implementación y entorno[3].

Las características

El ejemplo es la noción de punto, tal como éste podría aparecer en un sistema de gráficos bidimensional.



Un punto y sus coordenadas

Para caracterizar al tipo *PUNTO* como un tipo abstracto de dato se necesitarán las cuatro funciones de consulta x, y, ρ y θ. (Los nombres de las últimas dos se escribirán como *rho* y *theta* en los textos de software). La función x da la abscisa de un punto (la coordenada horizontal) e y da la ordenada (la coordenada vertical), ρ es la distancia al origen y θ es el ángulo con el eje horizontal. Los valores x e y de un punto se denominan coordenadas cartesianas, y los valores ρ y θ se denominan coordenadas polares. Otra función de consulta útil es *distancia* que proporciona la distancia entre dos puntos.

La especificación del TAD enumeraría órdenes tales como *trasladar* (para trasladar un punto cierto desplazamiento horizontal y vertical dado), *rotar* (para rotar el punto alrededor del origen un cierto ángulo dado) y *escalar* (para poner el punto más cerca o más lejos del origen cierto factor dado).

No es difícil escribir la especificación completa del TAD incluyendo estas funciones y algunos de los axiomas asociados. Por ejemplo, las signaturas de dos de las funciones serán

```
x: PUNTO → REAL
trasladar: PUNTO x REAL x REAL → PUNTO
```

y uno de los axiomas será (para todo punto p y cualesquiera números reales a, b):

```
x(trasladar(p1, a ,b )) = x(p1)+a
```

que expresa que trasladar un punto por valor de <a, b> incrementa su abscisa por valor de a.

[3] vid supra, pág. 32.

Atributos y rutinas

Todo tipo abstracto de dato como *PUNTO* está caracterizado por un conjunto de funciones, que describen las operaciones aplicables a las instancias del TAD. En las clases (las implementaciones de los TAD) las funciones se traducen en características – las operaciones aplicables a las instancias de la clase.

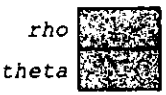
Se ha visto que las funciones de los TAD son de tres clases: *consultas, órdenes y creadores*. Para las características es necesaria una clasificación complementaria, basada en la forma en que se implementa la característica: en el espacio o en el tiempo.

El ejemplo de las coordenadas del punto muestra las diferencias claramente. Hay disponibles dos representaciones comunes para los puntos: cartesianos y polares. Si se escoge la representación cartesiana, cada instancia de la clase contendrá dos campos que representan a la x y la y respectivas del punto correspondiente:

(*punto_cartesiano*)

Si *p1* es el punto que se muestra, para obtener su x o su y se requiere simplemente mirar el campo correspondiente de la estructura. Sin embargo para obtener ρ y θ se requieren algunos cálculos: para calcular ρ hay que hacer $\sqrt{x^2 + y^2}$ y para θ se debe calcular arctg(y/x) para x diferente de cero.

Si se usa la representación en polares la situación se invierte: se podría acceder a ρ y θ sin más que examinar un campo; pero x e y requerirían pequeños cálculos (de ρ cosθ y ρ senθ).



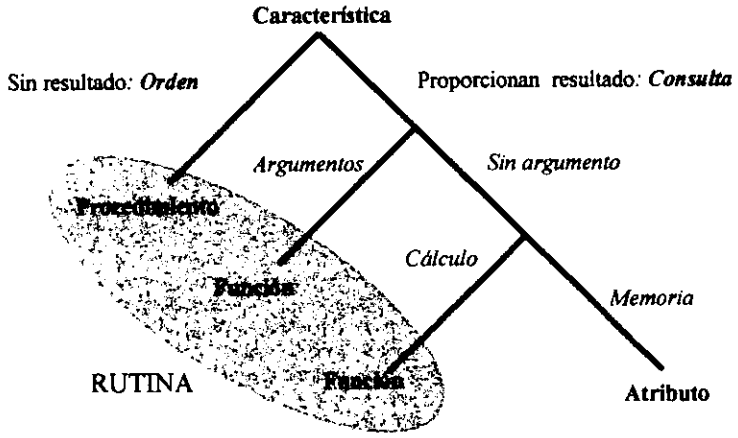
(*punto_polar*)

Este ejemplo muestra la necesidad de dos clases de características:

- Algunas características se representarán mediante espacio, es decir asociando un cierto elemento de información a cada instancia de la clase. Se denominarán **atributos**. Para los puntos en representación cartesiana x e y son atributos; *rho* y *theta* son atributos en la representación en polares.
- Algunas características se representarán en el tiempo, es decir definiendo un cierto cálculo (algoritmo) aplicable a todas las instancias de la clase. Éstas se llamarán **rutinas**. Para los puntos con representación en coordenadas cartesianas *rho* y *theta* son rutinas; para la representación en polares x e y son rutinas.

Hay una discusión adicional que afecta a las rutinas (la segunda de estas categorías). Algunas rutinas proporcionarán un resultado; se llamarán **funciones**. Aquí `x` e `y` en la representación en polares y `rho` y `theta` en la representación en cartesianas son funciones puesto que proporcionan un resultado de tipo `REAL`. Las rutinas que no proporcionan un resultado corresponden a las órdenes de la especificación de un TAD y se llaman **procedimientos**. Por ejemplo la clase `PUNTO` incluirá los procedimientos `trasladar`, `rotar` y `escalar`.

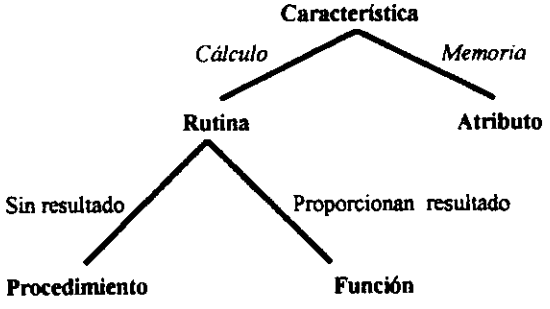
El árbol siguiente muestra esta clasificación de las características:



Clasificación de las características según su papel

Ésta es una clasificación externa, en la cual la cuestión principal es el aspecto que tendrá una rutina ante los clientes (sus usuarios).

También se puede considerar una visión más interna, usando como criterio principal la forma en que se implementa la característica en la clase, lo cual nos lleva a una clasificación diferente:



Clasificación de las características por la implementación

Acceso uniforme

En muchos casos, debería ser posible manipular objetos, por ejemplo un punto *p1*, sin tener que preocuparse sobre si la representación interna del punto *p1* es cartesiana, polar u otra. ¿Es apropiado entonces distinguir explícitamente entre atributos y funciones?

La respuesta depende de a quién pertenezca el punto de vista considerado: el punto de vista del proveedor (tal como ve las cosas el creador de la clase en sí, en este caso *PUNTO*) o el punto de vista del cliente (tal como ve las cosas el creador de la clase que usa a *PUNTO*). Para el proveedor, la distinción entre atributos y funciones es significativa y necesaria, puesto que en algunos casos se puede querer implementar una característica por almacenamiento y en otros por cálculo y esta decisión se debe reflejar en alguna parte. Lo que sería erróneo, sin embargo, es forzar a los clientes a tener en cuenta esta diferencia. Si se está accediendo a *p1*, es deseable poder obtener su x o su ρ sin tener que saber la forma en que están implementadas estas consultas.

El principio de acceso uniforme introducido en la discusión de modularidad, responde a este problema. El principio establece que el cliente debe ser capaz de acceder a una propiedad de un objeto usando una única notación, sin que importe si la propiedad está implementada por memoria o por cálculo (espacio o tiempo, atributo o rutina). Este principio tan importante se respetará al proponer más adelante que la notación para invocar a una característica *x* para *p1* será siempre

p1.x

tanto si su efecto es acceder a un campo de un objeto como si es ejecutar una rutina.

El principio de acceso uniforme es esencial para garantizar la autonomía de los componentes de un sistema. El principio preserva la libertad del diseñador de la clase para experimentar con diferentes técnicas de implementación sin causar molestias a los clientes.

La clase

A continuación se da una versión del código de la clase *PUNTO*. (Toda aparición de dos guiones consecutivos – introduce un comentario, que continúa hasta el final de la línea; los comentarios son explicaciones dirigidas al lector del código de la clase y no afecta la semántica de la misma.)

```
indexing
        description: "Puntos bidimensionales"
class  PUNTO feature
        x, y: REAL
                --Abscisa y ordenada
        rho: REAL is
                --Distancia al origen (0,0)
           do
              Result:= sqrt(x^2 + y^2)
           end
        theta: REAL is
                --Angulo con la horizontal
           do
              Result:= arctg (y / x)
```

```
        end
    distancia (p: PUNTO):REAL is
        --Distancia a p
        do
            Result: =sqrt((x -p.x)^2 + (y-p.y)^2)
        end
    trasladar (a, b: REAL) is
        --Desplazar horizontalmete a y verticalmente b.
        do
            x:=x+a
            x:=y+b
        end
    escalar  (factor:REAL) is
        --Escalar en factor.
        do
            x:=factor * x
            y:=factor * y
        end
    rotar (p: PUNTO; angulo: REAL) is
        --Rotar un angulo alrededor de p.
        do
            ...
        end
end
```

La clase consta principalmente de una cláusula que enumera las diferentes características y que se presenta mediante la palabra clave **feature**. También hay una cláusula **indexing** que da una información *descriptiva*, útil para los lectores de la clase pero que no tiene efecto en la semántica de su ejecución.

3.2.3. Convenciones básicas

La clase *PUNTO* muestra un cierto número de técnicas que son básicas dentro del método de orientación a objetos.

Reconocer los tipos de características

Las características x e y están declaradas simplemente como de tipo REAL, sin ningún algoritmo asociado; de modo que sólo pueden ser atributos. Todas las demás características tienen una cláusula de la forma

```
is
    do
      ... instrucciones ...
end
```

que define un algoritmo; esto indica que la característica es una rutina. Las rutinas *rho*, *theta* y *distancia* están de modo que proporcionan un resultado, de tipo REAL en todos los casos, lo cual se indica mediante unas declaraciones de la forma

```
rho: REAL is …
```

Esto las define como funciones. Las otras dos, *trasladar* y *escalar* no devuelven un resultado (ya que no tienen una declaración de resultado de la forma: `T` para algún tipo `T`) y son por tanto procedimientos.

Puesto que `x` e `y` son atributos mientras `rho` y `theta` son funciones, la representación escogida en esta clase particular para los puntos es la cartesiana.

Cuerpos de las rutinas y comentarios de encabezamiento

El cuerpo de la rutina (la cláusula **do**) es una sucesión de instrucciones. Se pueden utilizar puntos y comas, en la tradición Pascal-Algol, para separar las instrucciones y declaraciones sucesivas, pero los puntos y comas son opcionales. Por sencillez, se omitirán en las instrucciones que se encuentren en líneas distintas, pero siempre se incluirán para delimitar las instrucciones y declaraciones que aparezcan en una misma línea.

Todas las instrucciones de las rutinas de la clase *PUNTO* son asignaciones; para la asignación la notación que se emplea es el símbolo `:=` (tomado una vez más de las convenciones de Pascal). Por supuesto, este símbolo no debe confundirse con el símbolo de igualdad = que se utiliza, como en matemáticas, como operador de comparación.

Otra convención de la notación es el uso de comentarios de encabezamiento. Como ya se ha señalado los comentarios empiezan por dos guiones consecutivos `--`. Éstos pueden aparecer en cualquier lugar del texto de una clase en el que su creador crea que los lectores pueden beneficiarse con una explicación. El comentario de encabezamiento desempeña un papel especial, el cual, como regla general de estilo, debiera aparecer al principio de cada rutina, después de la palabra **is**, y con una sangría tal como se muestra en los ejemplos de la clase *PUNTO*. Ese comentario de encabezamiento debería expresar concisamente el propósito de la rutina.

Los atributos también deberían tener un comentario de encabezamiento inmediatamente después de su declaración, alineado con los comentarios de encabezamiento de las rutinas, tal como se ilustra en el texto de la clase con `x` y con `y`.

La cláusula indexing

Al comienzo de la clase hay una cláusula que comienza con la palabra **indexing**. Ésta contiene una sola entrada, rotulada como `description`. Esta cláusula de indexación no tiene efecto sobre la ejecución del software pero sirve para asociar información a la clase. En su forma general contiene cero o más entradas de la forma

```
palabra_indice: valor_indice, valor_indice, …
```

Donde `palabra_indice` es un identificador cualquiera, y cada `valor_indice` es un elemento arbitrario del lenguaje (identificador, entero, cadena...).

El beneficio es doble:

- Los lectores de la clase obtienen un resumen de sus propiedades, sin tener que ver los detalles.
- En un entorno de desarrollo de software que admita la reutilización, las herramientas de consulta (que suelen recibir el nombre de navegadores *browsers*) pueden utilizar la información de indexación para ayudar a los usuarios a encontrar las clases disponibles, las herramientas podrían permitir que los usuarios insertasen varias palabras de búsqueda y las harían concordar con las palabras y valores de indexación.

El ejemplo tiene una única entrada de índice, con `description` como palabra índice, y como valor índice tiene una cadena que describe el propósito de la clase. Es recomendable adoptar un estilo de escritura para las clases tal como se maneja en el ejemplo, comenzando con una cláusula **`indexing`** que proporcione una panorámica concisa de la clase, del mismo modo comenzar una rutina con un comentario de encabezamiento.

Tanto las cláusulas de indexación como los comentarios de encabezamiento son aplicaciones fieles del principio de autodocumentación: *siempre que sea posible la documentación de un módulo debe aparecer en el propio texto del módulo.*

Definir el resultado de una función

Necesitamos otra convención para entender los códigos de las funciones presentes en la clase *PUNTO*: *rho*, *theta* y *distancia*.

Cualquier lenguaje que admita funciones (rutinas que devuelven valores) debe ofrecer una notación que permita al cuerpo de la función indicar el valor que será devuelto por una llamada en particular. La convención que se utiliza aquí es sencilla. Se basa en un nombre de entidad predefinido, `Result`, que denota el valor que proporcionará la llamada. Por ejemplo, el cuerpo de *rho* contiene una asignación a `Result`:

```
Result:= sqrt (x^2 + y^2)
```

`Result` es una palabra reservada y sólo puede aparecer en funciones. En una función declarada del tal modo que tiene un resultado de tipo `T`, `Result` será tratado de la misma manera que otras entidades de tipo `T` y se le pueden asignar valores a través de asignaciones como la anterior.

Cualquier llamada a la función devolverá, como resultado, el valor final asignado a `Result` durante la ejecución de la llamada. Este valor siempre existe puesto que las reglas del lenguaje requieren que toda ejecución de la rutina, cuando comience, asigne a `Result` un valor por defecto. Para un `REAL` (que es éste el caso de *rho*) el valor de inicialización por defecto es cero; de modo que una función de la forma

```
valor_no_negativo (x: REAL): REAL is
          --El valor de x si éste es positivo; cero en caso contrario.
      do
         if x > 0.0 then
            Result:= x
         end
      end
```

devolverá siempre un valor bien definido (tal y como se describe en el comentario de encabezamiento) aun cuando la instrucción condicional no tenga cláusula **else**.

Heredar facilidades de propósito general

Otro aspecto de la clase *PUNTO* que requiere una aclaración es la presencia de llamadas a la función `sqrt` (en *rho* y en *distancia*). Esta función proporciona la raíz cuadrada de un número real, pero ¿de dónde sale esta función?

Puesto que no parece apropiado cargar a un lenguaje de propósito general con operaciones aritméticas especializadas, la mejor técnica es definir tales operaciones como características de alguna clase especializada – digamos *ARITMETICA* – y después bastará con requerir que cualquier clase que necesite de estas facilidades herede de esta clase especializada. Es suficiente entonces escribir *PUNTO* en la forma

```
clase PUNTO inherit
      ARITMETICA
feature
      ... El resto igual que en el código original ...
end
```

3.2.4. El estilo orientado a objetos

A continuación se revisarán las propiedades fundamentales de la clase *PUNTO* para entender el cuerpo de una rutina típica y sus instrucciones, para posteriormente ver la forma en que la clase y sus características pueden ser utilizadas por otras clases – los **clientes**.

La instancia actual

Aquí esta de nuevo el código de una de las rutinas dadas como ejemplo, el procedimiento *trasladar*:

```
trasladar (a, b:REAL) is
          --Desplaza horizontalmente a y verticalmente b.
      do
        x:= x+a
        y:= y+b
      end
```

El código de una clase describe las propiedades y el comportamiento de los objetos de un cierto tipo, puntos en este ejemplo. Esto se hace describiendo las propiedades y el comportamiento de una instancia típica del tipo – o más formalmente una **instancia actual** de la clase.

De vez en cuando es necesario referirse a la instancia actual explícitamente. La palabra reservada `Current` servirá para este propósito.

En el texto de una clase, `Current` denota la instancia actual de la clase que lo engloba. Como ejemplo de cuando es necesario `Current`, supóngase que se reescribe distancia de modo que ésta compruebe primero si el argumento `p` es el mismo punto que la instancia actual, en cuyo caso el resultado es 0 y no se necesita ningún cálculo adicional. Entonces distancia quedará de la forma

```
distancia (p: PUNTO):REAL is
        --Distancia a p
do
   if p/= Current then
      Result:= sqrt ((x - p.x)^2+(y - p.y)^2)
   end
end
```

(/= es el operador de desigualdad. En la instrucción condicional no es necesaria la cláusula **else**: si `p= Current` el resultado es cero).

Sin embargo, en la mayoría de las circunstancias, la instancia actual es implícita y no es necesario hacer referencias a `Current` por su nombre. Por ejemplo, las referencias a `x` en el cuerpo de *trasladar* y en las demás rutinas significan simplemente, si no hay más calificación: "la `x` de la instancia actual".

Clientes y proveedores

Ahora que ya se conoce la forma de definir clases simples. Es necesario saber cómo usar estas definiciones. Tales usos estarán englobados como parte de otras clases – puesto que en un enfoque orientado a objetos puro todo elemento software es parte del código de alguna clase.

Hay sólo dos formas de utilizar una clase como *PUNTO*. Una es **heredar** de ella. La otra es ser un **cliente** de *PUNTO*.

La forma más simple de ser cliente de una clase es declarar una entidad del tipo correspondiente:

CLIENTE, PROVEEDOR

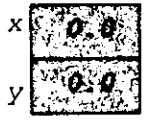
Sea `P` *una clase. Una clase* `C` *que contenga una declaración de la forma* `a : P` *se dice que es un cliente de* `P`. *Se dice entonces que* `P` *es un proveedor de* `C`.

En esta definición, `a` puede ser un atributo o función de `C` o una entidad local o un argumento de una rutina de `C`.

Por ejemplo, las declaraciones anteriores de *x*, *rho*, *theta* y *distancia* hacen a la clase *PUNTO* cliente de `REAL`. Otras clases podrían a su vez ser clientes de *PUNTO*. Por ejemplo:

```
class GRAFICOS feature
      p1: PUNTO
      ...
      una_rutina is
            --Realiza algunas acciones con p1.
         do
            ... Crear una instancia de PUNTO y conectarla a p1 ...
            p1.trasladar (4.0, -1.5)      --**
            ...
         end
      ...
end
```

Antes de que se ejecute la instrucción marcada con `--**`, el atributo *p1* tendrá un valor que denota una cierta instancia de la clase *PUNTO*. Supóngase que esta instancia representa el origen de coordenadas `x = 0, y = 0`:



se dice que la entidad *p1* está conectada a este objeto.

Llamada a una característica

La instrucción que tiene el doble asterisco

```
p1.trasladar (4.0, -1.5)
```

merece un examen detallado puesto que es el primer ejemplo completo de lo que puede llamarse el **mecanismo básico de la computación orientada a objetos**: la invocación o llamada a una característica. En la ejecución de un sistema software orientado a objetos, todos los cómputos se llevan a cabo invocando a ciertas características sobre ciertos objetos.

Esta llamada a característica en particular significa: aplicar a *p1* la característica *trasladar* de la clase *PUNTO*, con los argumentos 4.0 y –1.5 que se corresponderán con `a` y `b` en la declaración de *trasladar* tal como ésta aparece en la clase. Más generalmente, una llamada a una característica aparece en una de las siguientes formas básicas

```
x.f
x.f (u, v,...)
```

En las llamadas de este tipo, `x`, denominado receptor de la llamada, es una entidad o expresión (que en tiempo de ejecución estará conectada a un cierto objeto). Como cualquier otra entidad o expresión, `x` tiene un cierto tipo, dado por una clase `C`, entonces `f` debe ser una de las características de `C`. Más exactamente, en la primera

forma, `f` debe ser un atributo o una rutina sin argumentos; en la segunda forma, `f` debe ser una rutina con argumentos, y `u, v, ...`, que se denominan argumentos actuales de la llamada, deben ser expresiones que sean compatibles en tipo y en número con los argumentos formales declarados para `f` en `c`.

Además, `f` debe estar disponible (tiene que haber sido exportada) para el cliente que contenga esta llamada. El efecto en tiempo de ejecución de llamada anterior se define como sigue:

Efecto de invocar a una característica f sobre un receptor x.

Aplicar la característica `f` al objeto conectado a `x` después de haber dado como valor inicial a cada argumento formal de `f` (si lo hay) el valor del argumento actual correspondiente.

El principio del Receptor Único

Podría pensarse que la llamada a una característica no tiene nada de especial, después de todo, cualquier desarrollador de software sabe escribir un procedimiento `trasladar` que traslade un punto un cierto desplazamiento y que se invoque en la forma tradicional (disponible, con pequeñas modificaciones, en todos los lenguajes de programación):

```
trasladar (p1, 4.0, -1.5)
```

Sin embargo, a diferencia del estilo orientado a objetos de llamada a una característica, esta invocación trata a todos los argumentos por igual. La forma O-O no tiene tal simetría: se escoge un cierto objeto (en este caso, el punto `p1`) como receptor, relegando a los demás argumentos, en este caso los números reales 4.0 y –1.5 al papel de actores secundarios. Esta forma de hacer que todas las llamadas sean relativas a un único objeto receptor es un aspecto fundamental del estilo orientado a objetos:

PRINCIPIO DEL RECEPTOR ÚNICO

Toda operación orientada a objetos es relativa a un cierto objeto, la instancia actual en el momento de ejecutar esa operación.

En la construcción de software orientado a objetos nunca se pide realmente: "aplica esta operación a estos objetos". En lugar de esto lo que se dice es "*Aplica esta operación a este objeto*" ó "*Aplica esta operación a este objeto utilizando estos valores como argumentos*".

La identificación módulo-tipo

El principio del receptor único es una consecuencia directa de la combinación módulo-tipo, que se presentó anteriormente como punto de partida de la descomposición orientada a objetos: si todo módulo es un tipo, entonces toda operación aplicada al módulo es relativa a cierta instancia de dicho tipo (la instancia actual). Luego entonces para lograr entender como se puede reconciliar la noción *sintáctica* de **módulo** (un grupo de facilidades relacionadas que forman parte de un sistema de software) con la noción *semántica* de **tipo** (una descripción estática de ciertos posibles objetos en tiempo de ejecución) se puede hacer uso del ejemplo de la clase *PUNTO*:

Cómo funciona la fusión módulo-tipo

Las facilidades proporcionadas por la clase PUNTO, vista como un módulo, son precisamente las operaciones disponibles para ser aplicadas a instancias de la clase PUNTO, vista como tipo.

Esta identificación de las operaciones sobre las instancias de un tipo y los servicios ofrecidos por un módulo constituye el núcleo de la disciplina de estructuración que impone el método orientado a objetos.

El papel de Current

La forma de las llamadas indica por qué el código de una rutina (como *trasladar* en *PUNTO*) no necesita especificar "quién" es `Current`: dado que toda llamada a la rutina será relativa a un cierto receptor, que se especifica explícitamente en la llamada, la ejecución tratará a todo nombre de característica que aparezca en el código de la rutina (por ejemplo `x` en el código de *trasladar*) aplicándolo a ese receptor en particular. Por tanto, para la ejecución de la llamada

```
p1.trasladar (4.0, -1.5)
```

toda aparición de `x` en el cuerpo de *trasladar*, tal como las de la instrucción

```
x:= x+a
```

significa "el `x` de `p1`".

El significado exacto de `Current` se deriva de estas observaciones. `Current` significa: "el receptor de la llamada actual". Por ejemplo, a lo largo de toda la llamada anterior, `Current` denota el objeto conectado a *p1*. En una llamada subsiguiente, `Current` representará el receptor de esa nueva llamada. Que todo esto tenga sentido es una consecuencia de la extrema sencillez del modelo orientado a objetos, basado en llamadas a características y en el principio del receptor único.

PRINCIPIO DE LLAMADA A UNA CARACTERÍSTICA

- F1. *Ningún elemento software se ejecuta excepto como parte de una llamada a una rutina.*
- F2. *Toda llamada tiene un receptor.*

Llamadas calificadas y no calificadas

Se dijo anteriormente que todo en el método orientado a objetos se basa en llamadas a características. Una consecuencia de esta regla es que en realidad el código software contiene más llamadas que las que pueden apreciarse a primera vista. Las llamadas que se han visto hasta el momento eran de una de las dos formas siguientes:

```
x.f
x.f (u, v, ...)
```

Estas llamadas emplean lo que se denomina notación punto (con el símbolo ".") y se denominan **calificadas** porque el receptor de la llamada se identifica explícitamente: es la entidad o la expresión (x en los dos casos anteriores) que aparecen antes del punto.

Sin embargo, otras llamadas no están calificadas porque su receptor es implícito. Como ejemplo, suponga que se desea añadir a la clase *PUNTO* un procedimiento *transformar* que traslada y escala un punto. El código del procedimiento se basa en *trasladar* y en *escalar*.

```
transformar (a, b, factor: REAL) is
            --Desplaza a horizontalmente y b verticalmente y después
            --escala factor.
      do
         trasladar (a, b)
         escalar (factor)
      end
```

El cuerpo de la rutina contiene llamadas a *trasladar* y a *escalar*. A diferencia de los ejemplos anteriores, estas llamadas no muestran un receptor explícito y no usan la notación punto. Tales llamadas se denominan **no calificadas**.

Las llamadas no calificadas no violan la propiedad denominada F2 en el principio de llamada a una característica: al igual que las llamadas calificadas, tienen un receptor, la instancia actual. Cuando el procedimiento *transformar* se aplica a un cierto receptor, su cuerpo llama a *trasladar* y *escalar* aplicándolos al mismo receptor. De hecho esto podría haberse escrito en la forma

```
do
      Current.traslasdar (a, b)
      Current.escalar (factor)
```

Más generalmente, se puede reescribir cualquier llamada no calificada como una llamada calificada que tenga a `Current` como receptor. La forma no calificada, por supuesto es más sencilla e igualmente clara.

Las llamadas no calificadas que se acaban de examinar son llamadas a rutinas. Lo mismo se aplica a los atributos, aunque la presencia de llamadas es quizás menos obvia en este caso. Se observó anteriormente que en el cuerpo de *trasladar* la aparición de `x` en la expresión `x+a` representa al campo `x` de la instancia actual. Otra forma de expresar esta propiedad es que `x` es realmente una llamada a característica, de modo que la expresión como un todo se podría haber escrito en la forma `Current.x+a`.

De forma más general, cualquier instrucción o expresión de una de las formas

```
f
f(u, v, ...)
```

es de hecho una llamada no calificada y se puede también escribir esta en forma calificada mediante

```
Current.f
Current.f (u,v, ...)
```

aunque las formas no calificadas son más convenientes. Si se usa esa notación como instrucción, `f` debe ser un procedimiento (sin argumentos en el primer caso y con el número y tipo apropiado de argumentos en el segundo). Si es una expresión, `f` pudiera ser un atributo (sólo en la primera forma, puesto que los atributos no tienen argumentos) o una función.

Es importante señalar que esta equivalencia sintáctica sólo se aplica a una característica que se utilice como instrucción o a una expresión. Por tanto en la asignación siguiente del procedimiento *trasladar*

```
x:= x+a
```

sólo la aparición de `x` en la parte derecha es una llamada no calificada: `a` es un argumento formal, no una característica y la aparición de `x` en la parte izquierda no es una expresión (no se le puede asignar un valor a una expresión) de modo que no tiene sentido reemplazar ésta por `Current.x`.

Exportación selectiva y ocultación de información

En los ejemplos vistos hasta el momento todas las características de las clases se exportan a todos los posibles clientes. Por supuesto que esto no siempre es aceptable, hay que recordar lo importante que es la ocultación de información para el diseño de arquitecturas flexibles y coherentes.

La forma de restringir las características a ningún cliente, o a algunos clientes solamente, requiere de una cierta notación. Cada lenguaje de programación puede tener una notación o incluso un método para manejar estas restricciones. La notación propuesta aquí intenta ejemplificar los casos que pueden presentarse si se desea exportar u ocultar información a los clientes de una clase.

Apertura total

Por defecto, según se ha indicado, las características declaradas sin ninguna precaución particular están disponibles para todos los clientes. En una clase de la forma.

```
class S1 feature
   f ...
   g ...
end
```

las características `f`, `g`, ... están disponibles para todos los clientes de *S1*. Esto significa que en una clase `C`, para una entidad `x` declarada del tipo *S1*, una llamada de la forma

```
x.f
```

será válida, suponiendo que la llamada satisfaga las demás condiciones de validez para las llamadas a `f` en lo relativo al número y tipo de los argumentos.

Restringir el acceso de los clientes

Para restringir el conjunto de clientes que puede invocar a una cierta característica `h` se usará la posibilidad de que una clase pueda tener dos o más cláusulas **`feature`**. La clase sería entonces de la forma

```
class S2 feature
   f ...
   g ...
feature {A, B}
   h ...
   ...
end
```

Las características `f` y `g` tienen el mismo rango que antes: disponibles a todos los clientes. La característica `h` está disponible sólo para `A` y `B` y para sus descendientes (las clases que heredan directa o indirectamente de `A` o `B`). Esto significa que con `x` declarada del tipo *S2* una llamada de la forma

```
x.h
```

no será válida a menos que aparezca en el código de `A` o `B` o de alguno de sus descendientes.

Como caso especial, si se quiere ocultar una característica `i` a todos los clientes se puede declarar ésta como exportada a una lista vacía de clientes:

```
class S3 feature {}
   i ...
end
```

en este caso una llamada de la forma `x.i (...)` nunca es válida. Las únicas llamadas permitidas a `i` son las no calificadas, es decir de la forma

```
i (...)
```

que aparezca en el texto de una rutina de *S3* propiamente o en alguno de sus descendientes. Este mecanismo asegura una forma total de ocultación de información.

La posibilidad de ocultar una característica para todos los clientes, según se ilustra con `i`, está presente en muchos lenguajes O-O. Pero muchos otros no ofrecen un mecanismo selectivo como el que se ilustra con `h`: exportar una característica a ciertos clientes y a sus descendientes. Esto es desafortunado ya que conforme avanza la especialización del software las aplicaciones necesitan de este grado de control tan refinado.

Estilo para declarar características secretas

Una pequeña indicación respecto al estilo. Una característica que se declara en la forma que se ha utilizado anteriormente para `i` es secreta, pero quizás esta propiedad no se perciba con suficiente claridad a partir de la sintaxis. En particular, la diferencia con respecto a una característica pública pudiera no ser suficientemente visible, como en el caso

```
class S4 feature
      exportada
feature { }
      secreta...
end
```

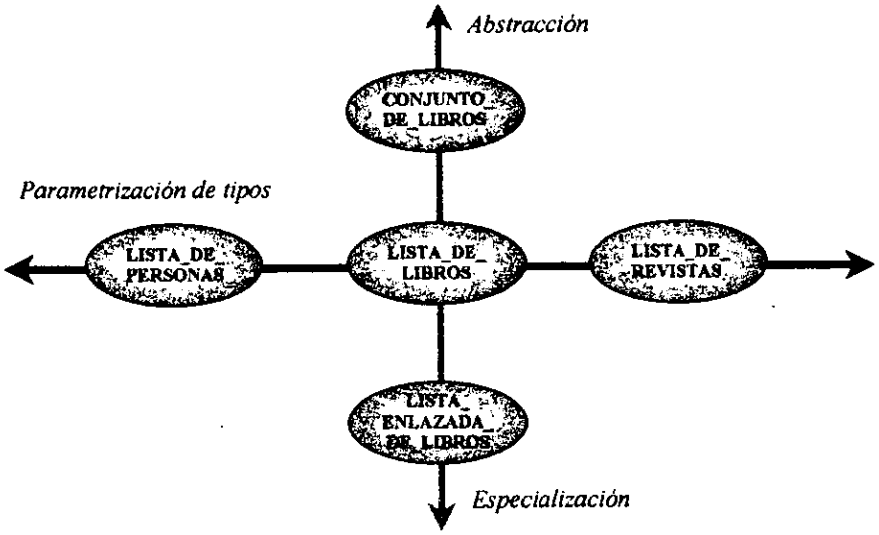
donde la característica *exportada* está disponible para todos los clientes mientras que *secreta* no está disponible para ningún cliente. La diferencia entre **feature** { }, con una lista vacía entre llaves, y **feature** sin llaves es más bien escasa. Por esta razón, el estilo recomendado no hace uso de una lista vacía sino de la lista que consta de una clase NONE, tal como en

```
class S5 feature
      exportada...
feature {NONE}
      secreta...
end
```

La clase NONE es una clase especial, definida de tal modo que no tiene instancias ni descendientes. De modo que exportar una característica a NONE es lo mismo, a efectos prácticos, que mantenerla en secreto. Como resultado, no hay ninguna diferencia significativa entre las formas ilustradas por *S4* y *S5*; sin embargo, por razones de claridad y legibilidad es preferible la segunda.

3.3. Genericidad

Fundir los conceptos de módulo y tipo ha llevado a desarrollar el potente concepto de clase, que sirve como base para el método orientado a objetos; tal como está, ya nos permite hacer mucho. Pero para alcanzar las metas de extensibilidad, reutilización y fiabilidad es preciso hacer que la estructura de clase sea más flexible, este esfuerzo estará dirigido en dos direcciones. Una, vertical en la figura de abajo, que representa abstracción y especialización; dará lugar al estudio de la herencia. La presente sección estudia la otra dimensión, horizontal en la figura: la parametrización de tipos, conocida también como **genericidad**.



3.3.1. Generalización de tipos horizontal y vertical

Con los mecanismos analizados hasta el momento todo lo que se ha necesitado es escribir la clase que esta en el centro de la figura, *LISTA_DE_LIBROS*, cada una de cuyas instancias representa una lista de objetos libro. Conocemos los tipos de características de que dispondría: `put` para añadir un elemento, `remove` para eliminar un elemento, `count` para saber cuántos elementos están presentes y así. Pero es fácil observar que puede haber dos formas de generalizar la noción de *LISTA_DE_LIBROS*:

- Las listas son casos especiales de estructuras "contenedores" de las cuales, entre otros muchos ejemplos, se puede mencionar también a los árboles, las pilas y los arrays. Se podría describir una variante más abstracta mediante una clase *CONJUNTO_DE_LIBROS*. Una variante más especializada, que abarca una forma particular de representación de listas, podría describirse mediante una clase *LISTA_ENLAZADA_DE_LIBROS*. Ésta es la dimensión vertical de la figura – la dimensión de la herencia.

- Las listas de libros son casos especiales de listas de objetos de alguna clase particular, de la cual otros ejemplos (entre muchos) son las listas de revistas, las listas de personas, las listas de enteros. Ésta es la dimensión horizontal de la figura – la dimensión de la genericidad. Al darle a las clases parámetros que representan tipos cualesquiera, se evitará la necesidad de escribir muchas clases que son casi idénticas – tales como *LISTA_DE_LIBROS* y *LISTA_DE_PERSONAS* – sin sacrificar la seguridad que se logra con los tipos estáticos.

3.3.2. Necesidad de la parametrización de tipos

La genericidad no es un concepto nuevo, aunque no se ha visto aplicado a las clases. Esta idea ya se introdujo en el análisis de los tipos abstractos de datos, donde se vio la necesidad de definir un TAD como algo parametrizable por tipos.

Tipos abstractos de datos genéricos

El ejemplo de TAD con el que se trabajó, *PILA*, se declaraba como *PILA*[G], lo cual significa que cualquier uso real requiere que se especifique un "parámetro genérico actual" que represente el tipo de objetos almacenados en una pila en particular. El nombre G se usa en la especificación de un TAD para indicar cualquier tipo posible que puedan tener los elementos de la pila; es lo que se denomina **parámetro genérico formal** de la clase. Con este enfoque se puede utilizar una única especificación para todas las pilas posibles; la alternativa sería difícil de aceptar pues habría que tener una clase *PILA_DE_ENTEROS*, una clase *PILA_DE_REALES* y así sucesivamente una clase para cada uno de los tipos posibles de los cuales se quiera tener pilas.

Cualquier TAD que describa estructuras "contenedor" será similarmente genérico – estructuras de datos como los conjuntos, listas, árboles, matrices, arrays y muchas otras que sirven para almacenar objetos de varios tipos posibles.

Esta misma discusión, aplicada a las clases contenedoras de nuestros sistemas de software en lugar de los TADs contenedores de los modelos matemáticos, producirán una situación similar.

La cuestión

Tomemos el mismo ejemplo de pila, ya no como un TAD matemático sino como una clase software. Conocemos la forma de escribir una clase *PILA_DE_ENTEROS* que describa la noción de pila de enteros. Las características incluirán *count* (número de elementos en la pila), *put* (poner en la pila un nuevo elemento), item (valor del elemento de la parte superior), *remove* (quitar el elemento que esta en la parte superior), *empty* (¿está vacía la pila?).

El tipo INTEGER se utilizará con frecuencia en esta clase. Por ejemplo es el tipo de argumento de *put* y el resultado de *item*:

```
put (elemento: INTEGER) is
            --Pone elemento en la cima
      do ... end
```

```
item: INTEGER is
            --Devuelve el elemento que está en la cima
      do ... end
```

Estas apariciones del tipo `INTEGER` provienen de la regla de la declaración explícita que se ha usado en el desarrollo de esta notación: cada vez que se introduce una entidad, que puede denotar a posibles objetos durante la ejecución, hay que escribir una declaración explícita de tipo para ella, como es el caso de *elemento*: INTEGER. En esta clase esto significa que hay que especificar un tipo para la consulta *item*, para el argumento elemento del procedimiento *put* y para otras entidades que denoten a posibles elementos de la pila.

Pero como consecuencia hay que escribir una clase diferente para cada tipo de pila: *PILA_DE_ENTEROS*, *PILA_DE_REALES*, *PILA_DE_PUNTOS*, etc. Todas estas clases de pilas son idénticas excepto en el tipo de las declaraciones de *item*, *elemento* y unas pocas entidades más: dado que las operaciones básicas sobre una pila son las mismas independientemente del tipo de los elementos de la pila, nada hay en los cuerpos de las diferentes rutinas que dependa de que se haya escogido *INTEGER*, *REAL*, *PUNTO* o *LIBRO* como tipo de los elementos de la pila. Para los que estén preocupados por la reutilización esto no es muy atractivo.

La cuestión está entonces en la contradicción que las clases contenedoras parecen provocar entre dos objetivos fundamentales de la calidad que se introdujeron en el primer capitulo:

- Fiabilidad: conservar los beneficios de la seguridad de tipos a través de declaraciones explícitas de tipo.
- Reutilización: ser capaz de escribir un único elemento software que abarque variantes de una misma noción.

El papel de los tipos

Existen dos razones básicas por las cuales una notación O-O debe poseer una comprobación estática de tipos:

- La razón de la *legibilidad*: las declaraciones explicitas le dicen al lector, sin lugar a dudas, cuál es el uso al que está destinado cada elemento. Esto es valiosísimo para cualquiera – el autor original o cualquier otra persona – que necesite entender el elemento, por ejemplo para depurarlo o extenderlo.
- La razón de la *fiabilidad*: gracias a las declaraciones explícitas de tipos, un compilador podrá ser capaz de detectar operaciones erróneas antes de que tengan oportunidad de producir daños. En las operaciones fundamentales de la computación orientada a objetos, llamadas a características de la forma general `x.f (a, ...)`, donde `x` es de un cierto tipo *TX*, las oportunidades de error son múltiples y variadas: la clase correspondiente a *TX* podría no tener una característica llamada `f`; la característica podría existir pero ser secreta; el número de argumentos podría no coincidir con los que se han declarado para `f` en la clase, el tipo de `a` o de otro argumento podría no ser compatible con el que `f` espera. En todos estos casos, dejar que el texto software pase sin oposición –como ocurriría

en un lenguaje sin control estático de tipos – significaría normalmente consecuencias desagradables durante la ejecución, como puede ser la caída del sistema con un diagnóstico del tipo "*mensaje no comprendido*" ó "*tipo no válido*". Con los tipos explícitos, el compilador no dejará pasar estas construcciones erróneas.

La clave para la fiabilidad del software, según se indicaba en la discusión de dicha noción, es la prevención más que la cura. Hay estudios que han detectado que el costo de corregir un error crece astronómicamente cuando se retrasa el momento de la detección. La comprobación estática de tipos, que permite la detección temprana de errores de tipo, es una herramienta fundamental en la conquista de la fiabilidad.

3.3.3. Clases genéricas

Como se ilustra con el ejemplo de la pila, reconciliar la comprobación estática de tipos con los requisitos de reutilización para las clases que describen estructuras contenedores significa que se desea:

- Declarar un tipo para cada entidad que aparezca en el texto de la clase pila, incluyendo las entidades que representen a elementos de la pila.
- Escribir la clase de modo que no tenga ninguna dependencia con el tipo de elementos y que entonces se pueda utilizar para construir pilas de elementos arbitrarios.

A primera vista estos requisitos parecen irreconciliables pero no lo son. El primero nos exige declarar un tipo; esto no supone que la declaración sea exacta. Tan pronto como se haya dado un nombre de tipo se habrá apaciguado al mecanismo de comprobación de tipos. He aquí la idea de la genericidad: obtener una clase parametrizada por un tipo, equipada con el nombre de un tipo ficticio, que se denomina parámetro genérico formal.

Declaración de una clase genérica

Por convenio, el parámetro genérico tendrá el nombre G; esto es una recomendación de estilo, no una regla formal. Si se necesitan más parametros genéricos, se les darán los nombres *H*, *I* y así sucesivamente.

La sintaxis incluirá los parámetros genéricos formales entre corchetes, después del nombre de la clase, tal como se hacía con los TAD genéricos. Véase un ejemplo:

```
indexing
        description: "Pilas de elementos de un tipo arbitrario G"
class STACK[G] feature
        count: INTEGER
                --Número de elementos que hay en la pila
        empty: BOOLEAN is
                --¿No hay elementos en la pila?
           do ... end
        full: BOOLEAN is
```

```
            --¿Está llena la pila?
         do ... end
      item: G is
            --El elemento de la cima
         do ... end
      put (x: G) is
            --Añade un elemento en la cima
         do ... end
      remove is
            --Quita el elemento que está en la cima
         do ... end
end   --clase STACK
```

En la clase se puede utilizar un parámetro genérico formal como `G` en las declaraciones: no solamente para resultados de funciones (como es el caso de *item*) y para argumentos formales de rutinas (como es el caso de *put*), sino también para atributos y para entidades locales.

Utilización de una clase genérica

Un cliente puede utilizar una clase genérica para declarar sus propias entidades, tales como una entidad que represente a una pila de elementos. En tal caso la declaración debe proporcionar los tipos, denominados **parámetros genéricos actuales** (o parámetros genéricos concretos) – tantos como parámetros genéricos formales tenga la clase. En este ejemplo es sólo uno:

```
pp: STACK[PUNTO]
```

El dar un parámetro genérico actual a una clase genérica para producir un tipo, como en este caso, se denomina **derivación genérica**. El tipo resultante, como *STACK*[*PUNTO*], se dice que está derivado genéricamente.

Una derivación genérica produce y requiere un tipo:

- El resultado de la derivación, *STACK*[*PUNTO*] en este ejemplo, es un tipo.
- Para producir este resultado, se necesita un tipo existente para servir como parámetro genérico actual, *PUNTO* en este ejemplo.

El parámetro genérico actual es un tipo cualquiera. En particular nada impide que se escoja a su vez un tipo derivado genéricamente. Si se supone la existencia de otra clase genérica `LIST[G]` se podría definir entonces una pila de listas de puntos:

```
plp: STACK[ LIST [PUNTO] ]
```

o incluso, usando la propia *STACK*[*PUNTO*] como parámetro genérico actual, se podría definir una pila de pilas de puntos:

```
ppp: STACK[ STACK [PUNTO] ]
```

No hay límites – salvo el que sugiere la indicación habitual de que el código del software debe ser tan sencillo como sea posible – para la profundidad de este anidamiento.

Terminología

Para discutir la genericidad, es necesario precisar los términos que se utilizan:

- Producir un tipo tal como *STACK [PUNTO]*, proporcionando un tipo, *PUNTO* en este caso, en calidad de parámetro genérico actual para una clase genérica, *STACK* en este caso, es realizar una derivación genérica. Podría pensarse en el término "instanciación genérica" para este proceso, pero esto puede ser confuso porque "instanciación" suele denotar un suceso en tiempo de ejecución, la producción de un objeto – una instancia – a partir de su molde (una clase). Una derivación genérica es un mecanismo estático, que afecta al texto del software, no a su ejecución. De modo que es preferible utilizar términos que sean completamente diferentes.
- Para fines de claridad y consistencia, el término "parámetro" es usado exclusivamente para denotar los tipos de las clases parametrizadas, nunca para denotar los valores que se pasan en la llamada a una rutina, que reciben el nombre de argumentos. En el software tradicional los términos "parámetro" y "argumento" son sinónimos. Aunque la decisión de qué término se utilizará para las rutinas y cuál se utilizará para las clases genéricas es una cuestión de convención, lo que sí es deseable es adoptar una regla consistente para evitar cualquier confusión.

Comprobación de tipos

Mediante el uso de la genericidad se puede garantizar que una estructura de datos contenga sólo elementos de un único tipo. Suponiendo que una clase contuviera las declaraciones

```
pcir: STACK[CIRCULO]; pcta: STACK[CUENTA]; cir: CIRCULO; cta: CUENTA
```

entonces lo que sigue serían instrucciones válidas en las rutinas de esa clase:

```
pcir.put (circ)          --Añade un circulo a la pila de circulos
pcta.put (cta)           --Añade una cuenta a la pila de cuentas
cir:= pcir.item          --Asigna a una entidad círculo a la cima de una
                         --pila de círculos
```

pero las instrucciones siguientes no son válidas y serán rechazadas:

```
pcir.put (cta)           --intenta añadir una cuenta a una pila de
                         --círculos
pcta.put (cir)           --intenta añadir un círculo a una pila de
                         --cuentas
cir:= pcta.item          --intenta acceder como círculo a la cima de
                         --una pila de cuentas
```

Esto descartará las operaciones erróneas de la categoría que se describía anteriormente, tal como el intento de retirar dinero de un círculo.

La regla de tipos

La regla de tipos que hace que el primer conjunto de ejemplos sea válido y segundo inválido es intuitivamente clara pero es conveniente precisarla.

Primero la regla básica no genérica. Si se considera una característica declarada en la forma siguiente, que no usa ningún parámetro genérico formal, en una clase no genérica C

```
f(a: T): U is
```

Entonces una llamada de la forma `x.f(d)` que aparece en una clase cualquiera `B` donde `x` es de tipo `C` será correcta a efectos de tipo si y sólo si: `f` está disponible para `B` – es decir, si se exporta de manera general, o se exporta selectivamente a un conjunto de clases que incluye a `B`; y `d` es de tipo `T`. El resultado de la llamada – se supone en este caso que `f` es una función – es de tipo `U`.

Supongamos ahora que `C` es genérico, con `G` como parámetro genérico formal y que tiene una característica

```
h(a:G): G is
```

una llamada a `h` será de la forma `y.h(e)` para una entidad `y` que haya sido declarada, para algún tipo `V`, como

```
y: C[V]
```

El equivalente de la regla no genérica es que ahora `e` debe ser de tipo `V`, ya que el argumento formal correspondiente `a` ha sido declarado como de tipo `G`, el parámetro genérico formal, y en el caso de `y` podemos considerar que cada aparición de `G` en la clase `C` es ahora `V`. Similarmente, el resultado de la llamada será de tipo `V`. Los ejemplos anteriores siguen todos este modelo: una llamada de la forma *p.put(z)* requiere un argumento *z* de tipo *PUNTO* si `p` es de tipo *STACK*`[`*PUNTO*`]:INTEGER` si *p* es de tipo *STACK*`[INTEGER]`; y *p.item* proporciona un resultado de tipo *PUNTO* en el primer caso y de tipo `INTEGER` en el segundo.

Estos ejemplos involucran características sin argumentos o con un solo argumento, pero la regla se puede extender inmediatamente a un número cualquiera de argumentos.

Operaciones sobre entidades de tipos genéricos

En una clase genérica `C[G, H, ...]` considérese una entidad cuyo tipo es uno de los parámetros genéricos formales, por ejemplo `x` de tipo `G`. Cuando la clase sea utilizada por parte de un cliente para declarar entidades, `G` puede representar finalmente cualquier tipo. De modo que cualquier operación que las rutinas de `C`

realicen sobre x debe ser aplicable a todos los tipos. Esto nos deja sólo con cuatro clases de operaciones:

Usos de entidades de tipo genérico formal

Los usos válidos para una entidad x cuyo tipo G *es un parámetro genérico formal son los siguientes:*

- G1 • *Utilizar x en la parte izquierda de una asignación* x:=y, *donde la expresión y de la parte derecha es también de tipo* G.
- G2 • *Utilizar x en la parte derecha de una asignación* y:=x, *donde la parte izquierda y de la entidad es también de tipo* G.
- G3 • *Utilizar x en una expresión booleana de la forma* x=y *o* x/=y, *donde y es también de tipo* G.
- G4 • *Utilizar x como un argumento actual en una llamada a una rutina cuyo argumento formal correspondiente haya sido declarado de tipo* G.

En particular, una instrucción de creación de la forma !!x no es admisible, puesto que no se sabe nada sobre los procedimientos de creación que puedan tener los posibles tipos que sean los parámetros genéricos actuales que se puedan corresponder con G.

Tipos y clases

Se ha aprendido a ver las clases, la noción central de la tecnología de objetos, como el producto de una mezcla entre el concepto de módulo y el concepto de tipo. Mientras no se tenía genericidad se podía decir que toda clase es un módulo y también un tipo.

Con la genericidad, la segunda de las afirmaciones no es literalmente verdadera, aunque la diferencia sea pequeña. Una clase genérica declarada como C[G] es, en lugar de un tipo, un patrón de tipo que abarca un conjunto infinito de tipos posibles; se puede obtener uno de estos tipos al proporcionar un parámetro genérico real – que es a su vez un tipo – que corresponde a G.

Esto nos lleva a una noción más general y más flexible. Pero para ganar en potencia hay que pagar un pequeño precio en sencillez: sólo mediante un pequeño abuso del lenguaje se puede seguir hablando, si *x* se ha declarado de tipo T, de las "características de T" o de "los clientes de T"; más que una clase, T puede ser ahora un tipo C[U] derivado genéricamente de alguna clase genérica C y algún tipo U. Por supuesto que sigue estando involucrada una clase – la clase C – por lo que este abuso del leguaje es aceptable.

Cuando se necesite ser riguroso la terminología debe ser la siguiente. Todo tipo T está asociado con una clase, la **clase base** de T, de modo que siempre es correcto hablar de las características o los clientes de la base T. Si T es una clase no genérica, entonces es su propia clase base. Si T es una derivación genérica de la forma C[U,...], entonces la clase base de T es C.

3.3.4. Arrays

Como conclusión de esta discusión es útil dar un vistazo a un ejemplo muy útil de clase contenedora: *ARRAY*, que representa a arrays de una dimensión.

Los arrays como objetos

La noción de array suele formar parte de la definición de los lenguajes de programación. Pero con la tecnología de objetos no es necesario cargar la notación con una estructura especial predefinida: un array será exactamente un objeto contenedor, una instancia de una clase que pudiera llamarse *ARRAY*.

ARRAY es un buen ejemplo de clase genérica. Véase a continuación una primera presentación:

```
indexing
      description:"Sucesiones de valores, todos del mismo tipo o%
                  %de tipos compatibles a los que se accede mediante%
                  %indices enteros en un intervalo contiguo."
class ARRAY[G] creation
      make
feature
      make (minindex, maxindex: INTEGER) is
            --Asignar espacio a un array con límites minindex y
            --maxindiex  (vacío si minindex > maxindex)
         do ... end
      lower, upper, count: INTEGER
            --Indices mínimo y máximos permitidos y tamaño del
            --array.
      put (v: G; i: INTEGER) is
            --Asignar v a la entrada de índice i.
         do ... end
      infix "@", item (i: INTEGER): G is
            --Elemento cuyo índice es i.
         do ... end
end   --class ARRAY
```

Para crear un array con límites *m* y *n*, con a declarado de tipo *ARRAY*[T] para algún tipo T se ejecutará la instrucción de creación

```
!! a.make (m, n)
```

Para establecer el valor de un elemento del array se usa el procedimiento *put*: la llamada *a.put (x, i)* establece *x* como valor del i-ésimo elemento. Para acceder al valor de un elemento se usará la función *item* (que tiene el sinónimo **infix** "@"), tal como en

```
x:= a.item (i)
```

A continuación se da un esbozo de la forma en que un cliente podría utilizar esta clase:

```
ap: ARRAY[PUNTO]; p1: PUNTO; i, j: INTEGER
...
! ap.make (-32, 101)  --Asigna espacio a un array con estos límites
ap.put (p1, i)        --Asigna p1 a la entrada de índice i.
...
p1:=ap.item (j)    --Asigna a p1 el valor de la entrada de índice j.
```

En una notación convencional (como Pascal), se escribiría

```
ap[i]:= p1    para    ap.put (p1,i)
p1:=ap[i]     para    p1:=ap.item (i)
```

Propiedades de los arrays

La idea de describir los arrays como objetos y *ARRAY* como una clase es un buen ejemplo de la potencia de unificación y simplificación de la tecnología de objetos, que ayuda a reducir la notación (el diseño o lenguaje de programación) a lo esencial y a reducir la cantidad de estructuras de propósito especial. En este ejemplo un array se ve simplemente como un ejemplo de una estructura contenedora, con sus propios métodos de acceso representados por las características *put* e *item*.

Puesto que *ARRAY* es una clase normal puede participar de lleno en los desarrollos orientados a objetos; en particular las otras clases podrán heredar de ésta. Una clase *LISTA_POR_ARRAY* que describa la implementación de la noción abstracta de lista mediante arrays puede ser un descendiente de *LIST* y de *ARRAY*.

3.4. Herencia

Inventar clases nuevas y únicas es posible, pero los sistemas rara vez surgen de la nada.

Casi siempre, el nuevo software se construye sobre desarrollos previos; la mejor manera de crearlo es por imitación, refinamiento y combinación. Los métodos de diseño tradicionales han ignorado durante mucho tiempo este aspecto del desarrollo de sistemas, que en la tecnología de objetos es una cuestión esencial.

Las técnicas de se han expuesto hasta ahora no son suficientes. Las clases proporcionan una buena técnica de descomposición modular y poseen muchas de las cualidades esperadas de los componentes reutilizables: son módulos homogéneos, coherentes; se puede separar claramente sus interfaces de sus implementaciones de acuerdo con el principio de ocultación de información; la genericidad les da cierta flexibilidad y se puede especificar sus semánticas de forma precisa gracias a las aserciones. Pero se necesita más para alcanzar en su totalidad los objetivos de reutilización y extensibilidad.

Para la *reutilización*, cualquier enfoque global debe enfrentarse al problema de la repetición y la variación. Para evitar reescribir nuevamente el mismo código, perder tiempo, introducir inconsistencias y errores peligrosos, se necesitan técnicas para capturar las evidentes semejanzas que existen dentro de grupos de estructuras similares – todos los editores de texto, todas las tablas, todos los manipuladores de archivos – aunque considerando las muchas diferencias que caracterizan a los casos individuales.

Para la *extensibilidad*, un sistema del tipo descrito hasta aquí tiene la ventaja de garantizar la consistencia de tipos en tiempo de compilación, pero prohibe la combinación de elementos de diversas formas aun en casos legítimos. Por ejemplo, aún no se puede definir un array que contenga objetos geométricos de tipos compatibles pero diferentes, tales como *PUNTO* y *SEGMENTO*.

Progresar en la reutilización o en la extensibilidad exige que se aprovechen las fuertes relaciones conceptuales que guardan las clases entre sí: una clase puede ser una extensión, una especialización o la combinación de otras clases. Se necesita soporte por parte del método y del lenguaje para registrar y utilizar estas relaciones. La **herencia** proporciona este soporte.

3.4.1. Polígonos y rectángulos

Para dominar los conceptos básicos de la herencia se usará un ejemplo simple. El ejemplo es un esbozo más que un ejemplo completo, pero muestra bien las ideas esenciales.

Polígonos

Suponga que se quiere construir una biblioteca gráfica. Las clases de esta biblioteca describirán abstracciones geométricas: puntos, segmentos, vectores, círculos, elipses, polígonos generales, triángulos, rectángulos, cuadrados y otros.

Considere primero la clase que describe los polígonos generales. Entre las operaciones se incluirán él cálculo del perímetro, la traslación y la rotación. La clase puede tener la siguiente forma:

```
indexing
      description: "Polígonos con número arbitrario de vértices"
class POLIGONO    creation
...
feature  --Acceso
      cantidad: INTEGER
            --número de vértices
      perimetro: REAL is
            --longitud del perímetro
         do ... end
feature -- transformación
      visualizar is
            --Mostrar el polígono en pantalla
         do ... end
      rotar (centro: PUNTO; angulo: REAL) is
            --Rotar el ángulo alrededor del centro
```

```
        do ... end
      trasladar (a, b: REAL) is
            --Desplaza a horizontalmente y b verticalmente.
        do ... end
      ...otras declaraciones de características ...
feature {NONE} --Implementación
      ... vertices: LINKED_LIST[PUNTO]
            --Puntos sucesivos formando el polígono
invariant
      igual_cantidad_que_implementacion:cantidad = vertices.cantidad
      al_menos_tres: cantidad >=3
            --Un polígono tiene al menos tres vértices
end
```

El atributo *vertices* proporciona la lista de los vértices; la elección de una lista enlazada es sólo una posible implementación. (Un array podría ser mejor).

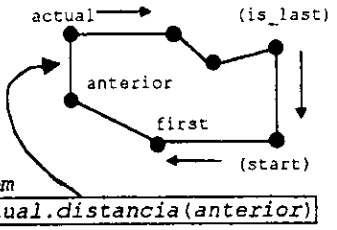
He aquí una posible implementación para un procedimiento típico, *rotar*. El procedimiento realiza una rotación con cierto ángulo alrededor de un centro de rotación dado. Para rotar un polígono, es suficiente rotar cada vértice sucesivamente:

```
rotar (centro : PUNTO; angulo: REAL) is
            --Rotar cierto ángulo alrededor del centro
      do
         from
            vertices.comenzar
         until
            vertices.fin
         loop
            vertices.item.rotar (centro, angulo)
            vertices.avanzar
      end
end
```

Para entender este procedimiento, observe que la característica *elem* de *LINKED_LIST* proporciona el valor del elemento actual de la lista (allí donde está el cursor). Dado que *vertices* es del tipo *LINKED_LIST [PUNTO]*, *vertices.item* denota un punto, para el cual puede aplicarse el procedimiento *rotar* definido para la clase *PUNTO* discutida en la sección que abordo el tema de las clases. Como el receptor de cualquier característica siempre tiene un tipo claramente definido, es válido dar el mismo nombre, aquí *rotar*, a características de diferentes clases.

Otra rutina, más importante para nuestros propósitos inmediatos, es la función para calcular el perímetro de un polígono. Debido a que los polígonos propuestos aquí no tienen propiedades especiales, la única forma para calcular sus perímetros es aplicar una iteración que recorra sus vértices y sumar las longitudes de los lados. He aquí la implementación de *perimetro*:

```
perimetro: REAL is
            --Suma de las longitudes de los lados
      local
            actual, anterior: PUNTO
      do
            from
                  vertices.start; actual:=vertices.item
                  check not vertices.after end   --una consecuencia
                                                 --de al_menos_tres
            until
                  vertices.is_last
            loop
                  anterior:= actual
                  vertices.forth
                  actual:= vertices.item
                  Result:= Result + actual.distancia(anterior)

            end
            Result:=Result + actual.distancia(vertices.first)
      end
```



Rectángulos

Se supone que se necesita una nueva clase para representar rectángulos. Se podría empezar partiendo de cero. Pero los rectángulos son un tipo especial de polígonos y muchas de las características son las mismas: un rectángulo probablemente será trasladado, rotado o visualizado de la misma forma que un polígono general. Por otra parte, los rectángulos tienen también características especiales (tales como la diagonal), propiedades especiales (el número de vértices es cuatro, los ángulos son ángulos rectos) y versiones especiales de algunas operaciones (para calcular el perímetro de un rectángulos, se puede hacer un algoritmo mejor que el anterior algoritmo para polígonos generales).

Se puede aprovechar esta mezcla de semejanzas y particularidades definiendo la clase *RECTANGULO* como una clase que **hereda** de la clase *POLIGONO*. Esto hace que todas las características de *POLIGONO* – que se denomina **padre** de *RECTANGULO* – también se apliquen por defecto a la clase heredera. Es suficiente con dar a *RECTANGULO* una **cláusula de herencia**.

```
class RECTANGULO inherit
      POLIGONO
feature
      ...características específicas para rectángulos
end
```

La cláusula **feature** de la clase que hereda no repite las características del padre: las características están disponibles automáticamente debido a la cláusula de herencia. Solamente se enumeran las características que son específicas de la clase

heredera. Éstas pueden ser características nuevas, tales como *diagonal*; pero pueden ser también redefiniciones de características heredadas.

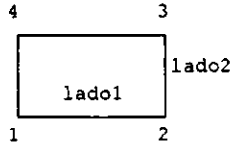
La segunda posibilidad es útil para una característica que ya tenía sentido en el padre pero requiere una forma diferente en la clase heredera. Considérese *perimetro*. Se tiene una implementación mejor para rectángulos: no se necesita calcular las cuatro distancias de vértice a vértice; el resultado es simplemente el doble de la suma de las longitudes de los dos lados. Una clase heredera que redefine una característica del padre debe declararlo en la cláusula de herencia a través de la subcláusula **redefine**:

```
class RECTANGULO inherit
      POLIGONO
         redefine perimetro
feature
      ...
end
```

Esto permite a la cláusula **feature** de *RECTANGULO* contener una nueva versión de *perimetro*, la cual reemplazará la versión de *perimetro* de la clase *POLIGONO*. Si la subcláusula **redefine** no estuviese presente, una nueva declaración de *perimetro* entre las características de *RECTANGULO* sería un error, debido a que *RECTANGULO* ya tiene una característica *perimetro* heredada de *POLIGONO*, y esto equivaldría a declarar una característica dos veces.

La clase *RECTANGULO* se parece a la siguiente:

```
indexing
      description: "Rectángulos, vistos como un caso especial de%
                  %polígonos generales"
class RECTANGULO inherit
      POLIGONO
         redefine perimetro
creation
      crear
feature --Inicialización
      crear (centro: PUNTO; l1, l2, angulo: REAL) is
            --Crear con centro en PUNTO, orientación angulo
            --y lados de longitud l1,l2
          do ... end
feature --Acceso
      lado1, lado2: REAL
            --La longitud de los dos lados
      diagonal:REAL
            --La longitud de la diagonal
      perimetro: REAL is
            --Suma de las longitudes de los lados
            --(Redefinición de la versión de POLIGONO)
         ...do
            Result:= 2* (lado1 + lado2)
         ...end
invariant
    cuatro_lados: cantidad=4
```



```
    primer_lado: vertices.i_th(1)).distancia(vertices.i_th(2))=lado1
    segundo_lado:(vertices.i_th(2)).distancia(vertices.i_th(3))=lado2
    tercer_lado: (vertices.i_th(3)).distancia(vertices.i_th(4))=lado1
    cuarto_lado:(vertices.i_th(4)).distancia(verti  s.i_th(1))=lado2
end
```

Debido a que *RECTANGULO* hereda de *POLIGONO*, todas las características de la clase padre se aplican a la nueva clase: `vertices`, `rotar`, `trasladar`, `perimetro` (en la forma redefinida) y otros. No es necesario que sean repetidos en la nueva clase.

Este proceso es transitivo: cualquier clase que herede de *RECTANGULO*, digamos *CUADRADO*, también tiene las características de *POLIGONO*.

Terminología y convenciones básicas

Los siguientes términos serán útiles además de "clase heredera" y "padre".

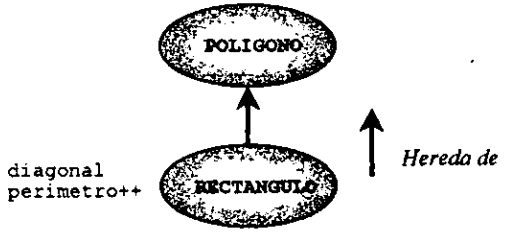
TERMINOLOGÍA DE HERENCIA

Un descendiente de una clase C *es cualquier clase que hereda directamente o indirectamente de* C, *incluyendo el propio* C. (*Formalmente:* C *o, recursivamente, un descendiente de un descendiente de* C.)
Un descendiente propio de C *es otro descendiente de* C *que no sea el propio* C.
Un antecesor de C *es una clase* A *tal que* C *es un descendiente de* A. *Un antecesor propio de* C *es una clase* A *tal que* C *es un descendiente propio de* A.

En la literatura también se encuentran los términos "subclase" y "superclase", pero se prescindirá de ellos porque son ambiguos; a veces "subclase" significa descendiente inmediato, a veces se usa en el sentido más general de descendiente propio, y no siempre está claro cuál es.

La terminología asociada emplea las características de una clase: una característica es *heredada* (proviene de un antecesor propio) o *inmediata* (introducida en la propia clase).

En las representaciones gráficas de las estructuras de software orientado a objetos, donde las clases se representan mediante elipses ("burbujas"), las relaciones de herencia aparecerán como flechas simples



Un enlace de herencia

La flecha apunta hacia arriba, de la clase heredera al padre; la convención, fácil de recordar, es que representa la relación *"hereda de"*. En cierta literatura se encontrará la práctica contraria; aunque en general tales elecciones de convenciones gráficas son en cierto modo una cuestión de gusto, en este caso una convención parece definitivamente mejor que la otra – en el sentido que una sugiere la relación natural y la otra puede llevar a confusión. Una flecha no es un dibujo arbitrario sino que indica un enlace unidireccional, entre los dos finales de la flecha. Aquí:

- Una instancia de la clase heredera puede verse como una instancia del padre, pero no inversamente.
- El texto de la clase heredera siempre mencionará el padre (según lo establece la cláusula `inherit`), pero no inversamente; esto es en realidad una propiedad importante del método, resultado entre otros del principio de abierto-cerrado, el que una clase no "conoce" la lista de sus clases herederas y otros descendientes propios.

Aunque no existe una regla absoluta para la colocación de las clases en los diagramas de herencia, se debería intentar siempre colocar las clases padre por encima de sus clases herederas.

La herencia de los invariantes

Se han enunciado los invariantes de la clase *RECTANGULO*, los cuales expresan que el número de lados es cuatro y que las longitudes sucesivas son *lado1*, *lado2*, *lado1* y *lado2*.

La clase POLIGONO también tenía un invariante, el cual continúa aplicándose a su clase hija:

REGLA DE HERENCIA DEL INVARIANTE

La propiedad de invariante de una clase es el **and** *booleano de las aserciones que aparecen en su cláusula* **invariant** *y de las propiedades de invariante de sus padres si los hubiere.*

Debido a que los padres pueden tener sus propios padres, esta regla es recursiva: al final el invariante completo de una clase se obtiene aplicando un "and" entre las cláusulas invariantes de todos sus antecesores.

La regla refleja una de las características básicas de la herencia: decir que B hereda de A es afirmar que puede verse una instancia de B también como una instancia de A. Como resultado, cualquier restricción de consistencia que se aplique a las instancias de A, según se expresa en el invariante, se aplica también a las instancias de B.

En el ejemplo, el segundo invariante de la cláusula (*al_menos_tres*) de *POLIGONO* afirma que el número de lados debe ser al menos tres; esto es subsumido por la subcláusula *cuatro_lados* en la cláusula invariante de *RECTANGULO*, la cual requiere que sea exactamente cuatro.

La herencia y la creación

Aunque no se muestra, un procedimiento de creación para POLIGONO sería de la forma

```
crear_poligono (lv:LINKED_LIST[PUNTO]) is
            --Crear el polígono con los vértices tomados de lv
      require
         lv.cantidad>=3
      do
         ...Inicializar la representación del polígono a partir de
            los elementos de lv.
      ensure
            --vertices y lv tienen los mismos elementos
            --puede expresarse formalmente
      end
```

Este procedimiento toma una lista de puntos, que contiene al menos tres elementos, y los usa para crear el polígono.

El procedimiento de creación de la clase *RECTANGULO*, mostrado anteriormente, tiene cuatro argumentos: un punto que sirve de centro, la longitud de dos lados y una orientación. Observe que la característica *vertices* sigue siendo aplicable a los rectángulos; como consecuencia, el procedimiento de creación de *RECTANGULO* debería crear una lista de *vertices* con los valores de los puntos apropiados (las cuatro esquinas se calculan a partir del centro, las longitudes de los lados y la orientación, dados como argumentos).

El procedimiento de creación para polígonos generales es inadecuado para rectángulos, debido a que sólo serían aceptables listas de cuatro elementos que satisfagan el invariante de la clase *RECTANGULO*. Inversamente, el procedimiento de creación para rectángulos no es apropiado para polígonos arbitrarios. Esto es un caso común: un procedimiento de creación del padre no es necesariamente correcto como procedimiento de la clase heredera. La razón precisa es fácil de ver; a partir de la observación de que el papel formal de un procedimiento de creación es establecer el invariante de la clase. El procedimiento de creación del padre era necesario para establecer el invariante del padre; pero como se ha visto, el invariante de la clase hija puede ser más fuerte (y normalmente lo es); no puede esperarse que el procedimiento original garantice el nuevo invariante.

En el caso de que una clase hija añada nuevos atributos, los procedimientos de creación necesitarían inicializar estos atributos y además requerirían argumentos adicionales. He aquí la regla general:

REGLA DE CREACIÓN EN LA HERENCIA

El status de creación de una característica heredada en la clase padre (es decir, que sea o no un procedimiento de creación) no tiene que ver con su status de creación en la clase heredera.

Un procedimiento de creación heredado aún está disponible en la clase heredera como una característica normal de la clase; pero por defecto no mantiene su

status como un procedimiento de creación. Sólo los procedimientos enumerados en la propia cláusula **creation** de la clase heredera tienen ese status.

En algunos casos, por supuesto, un procedimiento de creación del padre puede aplicarse todavía como procedimiento de creación; en esos casos simplemente se indicará en la cláusula de creación:

```
class B inherit
   A
creation
   crear
feature
```

donde `crear` se hereda –sin modificación – de `A`, el cuál también se incluía en su propia cláusula **creation**.

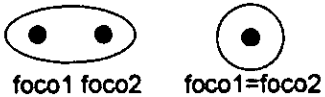
Un ejemplo de jerarquía

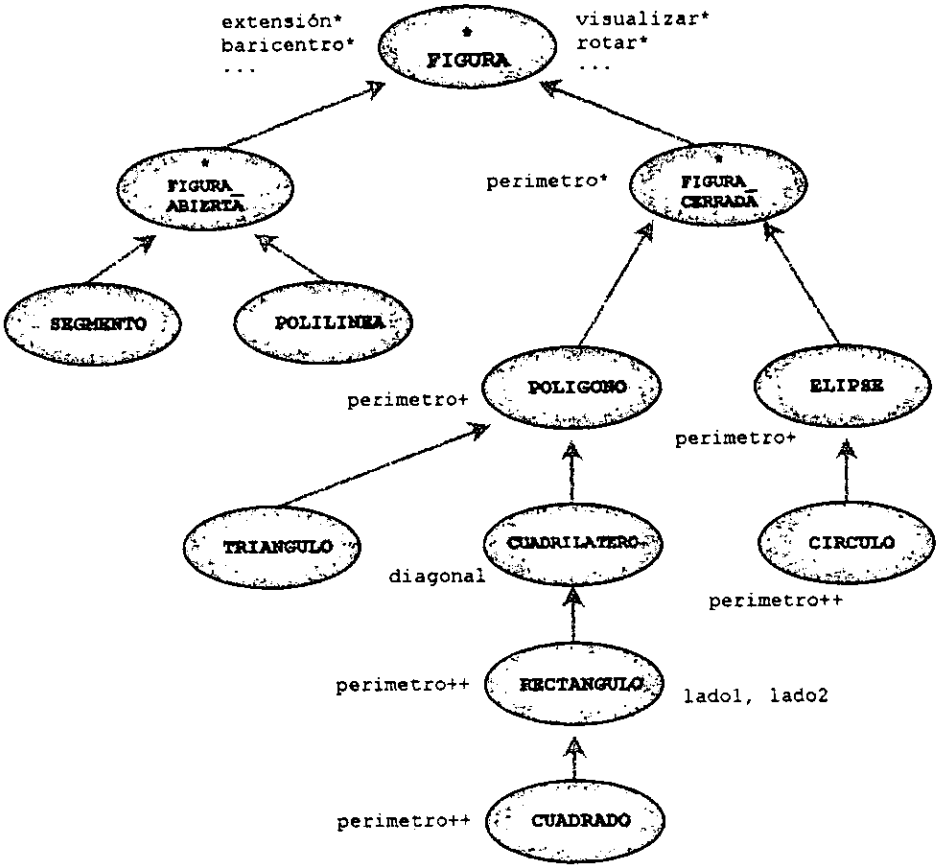
Para el resto del capítulo será útil considerar el ejemplo *POLIGONO-RECTANGULO* en el contexto de una jerarquía de herencia más general de tipos de figuras geométricas, tal como la que se muestra en la próxima página.

Las figuras se han clasificado en variantes abiertas y cerradas. Junto con los polígonos, un ejemplo de figura cerrada es la elipse; un caso especial de elipse es el círculo.

En el ejemplo original, por simplicidad, *RECTANGULO* heredaba directamente de *POLIGONO*. Debido a que la clasificación de polígonos que se esboza aquí se basa en el número de vértices, es preferible introducir una clase intermedia *CUADRILATERO*, en el mismo nivel de *TRIANGULO*, *PENTAGONO* y clases similares. La característica `diagonal` puede trasladarse al nivel de *CUADRILATERO*.

Observe la presencia de `CUADRADO` como clase hija de *RECTANGULO*, caracterizado por el invariante *lado1=lado2*. Similarmente, una elipse tiene dos focos, que para un círculo son el mismo punto, dando a círculo una propiedad invariante de la forma `equal (foco1=foco2)`.





Jerarquía de tipos de figuras(*)

3.4.2. Polimorfismo

Las jerarquías de herencia nos dan una flexibilidad considerable para la manipulación de objetos, manteniendo la seguridad de la comprobación estática de tipos. El polimorfismo tiene mucho que ver con la forma en que pueden manejarse los tipos, proporciona un mecanismo de conexión entre entidades de tipos diferentes pero compatibles.

(*) La figura hace uso de la notación siguiente: * para señalar una característica o una clase diferida; + para una clase efectiva o para una característica que se hace efectiva y ++ para señalar la redefinición (informalmente sugieren hacer doblemente efectivo). De acuerdo con las convenciones de Business Object Notation (B.O.N).

Conexión polimorfa

"Polimorfismo" significa la capacidad de adoptar varias formas. En el desarrollo orientado a objetos lo que puede tomar varias formas es una entidad variable o un elemento de estructura de datos, los cuales tendrán la capacidad, en tiempo de ejecución, de conectarse a objetos de tipos diferentes, todos ellos controlados por la declaración estática.

Dada la estructura de herencia mostrada en la figura, supongamos las siguientes declaraciones usando nombres de entidades cortos pero mnemotécnicos:

```
p: POLIGONO; r: RECTANGULO; t: TRIANGULO
```

Entonces las siguientes asignaciones son válidas:

```
p:=r
p:=t
```

Estas instrucciones asignan a una entidad que denota un polígono el valor de una entidad que denota a un rectángulo en el primer caso, y un triángulo en el segundo.

Tales asignaciones, en las cuales el tipo del origen (el lado derecho) es diferente al tipo del destino (el lado izquierdo) se llaman **asignaciones polimorfas**. Una entidad tal como p la cual aparece en alguna asignación polimorfa es una **entidad polimorfa**.

Antes de la introducción de la herencia, todas las asignaciones eran monomorfas (no polimorfas): se podía asignar un punto a un punto, un libro a un libro, una cuenta a una cuenta. Con el polimorfismo, se comienza a ver más acción en la escena de las conexiones.

Las asignaciones polimorfas tomadas como ejemplo son legítimas: la estructura de la herencia permite ver una instancia de *RECTANGULO* o de *TRIANGULO* como una instancia de *POLIGONO*. Se dice que el tipo del origen es compatible con el tipo del destino. En la dirección inversa, como con r:=p, la asignación no sería válida.

En lugar de una asignación, se puede llevar a cabo el polimorfismo a través del paso de parámetros, como puede ser una llamada de la forma `f ( r)` o `f ( t)` y la declaración de una característica de la forma:

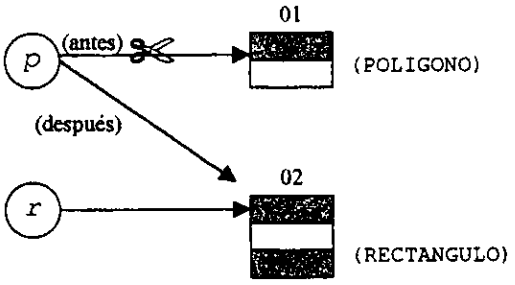
```
f (p: POLIGONO) is do ... end
```

Como se recordará, la asignación y el paso de parámetros tienen la misma semántica, y se llaman en conjunto *conexión*; se puede hablar de conexión *polimorfa* cuando el origen y el destino tienen diferentes tipos.

¿qué sucede exactamente durante una conexión polimorfa?

Todas las entidades que aparecieron en los casos anteriores de conexiones polimorfas son de referencia: los valores posibles para p, r y t no son objetos sino referencias a

objetos. Además el resultado de una asignación tal como `p:=r` es simplemente reconectar una referencia.



Reconexión de la referencia polimorfa

Por tanto a pesar del nombre, no debiera imaginarse, cuando se piensa en el polimorfismo, en la transmutación de objetos en tiempo de ejecución. Una vez creado, un objeto nunca cambia su tipo. Esto sólo lo hacen las referencias cuando se reconectan a objetos de tipos diferentes. Esto también significa que el polimorfismo no perjudica la eficiencia; una reconexión de referencias (una operación muy rápida) cuesta lo mismo independientemente de los objetos involucrados.

Debido a que una clase descendiente puede introducir nuevos atributos, las instancias correspondientes pueden tener más campos; la última figura sugería esto mostrando el objeto *RECTANGULO* mayor que el objeto *POLIGONO*. Tales diferencias en el tamaño de los objetos no causan ningún problema si todo lo que se esta reconectando es una referencia.

Estructuras de datos polimorfas

poli_arr:*ARRAY*[*POLIGONO*]

Cuando se asigna un valor *x* a un elemento del array, como en

poli_arr.put (x, un_indice)

(para algún valor válido del índice *un_indice*), la especificación de la clase *ARRAY* indica que el tipo del valor asignado debe ser compatible con el parámetro genérico real:

```
class ARRAY[G] creation
    ...
feature      --Cambiar de elementos
    put(v:G; i: INTEGER) is
      --Asigna v a la entrada de índice i
    ...
end   --class ARRAY
```

Debido a que *v*, el argumento formal que corresponde a *x*, se declaró de tipo G en la clase, y el parámetro genérico real correspondiente a G es *POLIGONO* en el caso

de *poli_arr*, el tipo de *x* debe ser compatible con *POLIGONO*. Como se ha visto, esto no requiere que *x* sea de tipo *POLIGONO*, se acepta cualquier descendiente de *POLIGONO*.

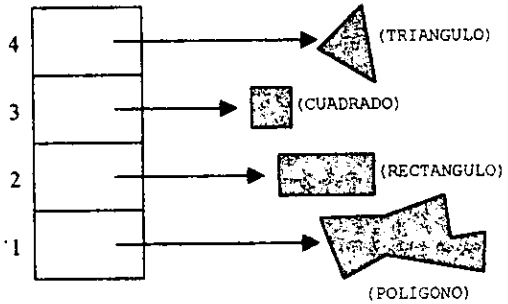
De modo que suponiendo que el array tiene límites 1 y 4, que se han declarado algunas entidades como

```
p: POLIGONO; r: RECTANGULO; c: CUADRADO; t: TRIANGULO
```

y que se han creado los objetos correspondientes, se puede ejecutar entonces

```
poli_arr.put(p,1)
poli_arr.put(r,2)
poli_arr.put(c,3)
poli_arr.put(t,4)
```

Obteniéndose un array de referencias a objetos de tipos diferentes:



Un array polimorfo

Tales estructuras de datos, que contienen objetos de tipos diferentes (todos ellos descendientes de un tipo común) se llaman **estructuras de datos polimorfas**. El uso de arrays es sólo una posibilidad; cualquier otra estructura contenedora, tales como una lista o una pila, pueden ser de igual forma polimorfa.

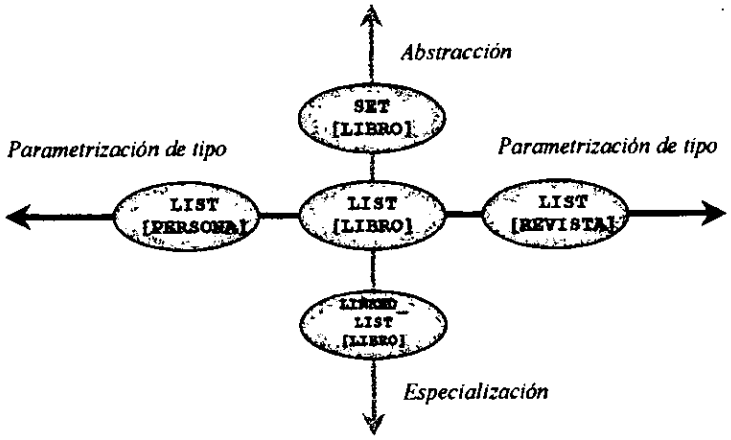
La introducción de estructuras de datos polimorfas alcanza el objetivo de combinar la genericidad y la herencia para la máxima flexibilidad y seguridad. Merece la pena recordar la figura que ilustraba la idea (siguiente página).

Los tipos que se denominaban informalmente *CONJUNTO_DE_LIBROS* y similares se han sustituido en la figura por tipos derivados genéricamente, como *SET*[*LIBRO*].

La combinación de generacidad y herencia es potente. Permite describir estructuras de objetos que son tan generales como se desee, pero no más. Por ejemplo:

- *LIST*[*RECTANGULO*]: puede contener cuadrados pero no triángulos.
- *LIST*[*POLIGONO*]: puede contener cuadrados, rectángulos, triángulos pero no círculos.

- *LIST[FIGURA]*: puede contener instancias de cualquiera de las clases en la jerarquía de *FIGURA*; pero no libros o cuentas bancarias.



Dimensiones de la generalización

Eligiendo como parámetro genérico real una clase en un lugar variable en la jerarquía, pueden fijarse los límites de qué aceptará un contenedor.

3.4.3. Tipos y herencia

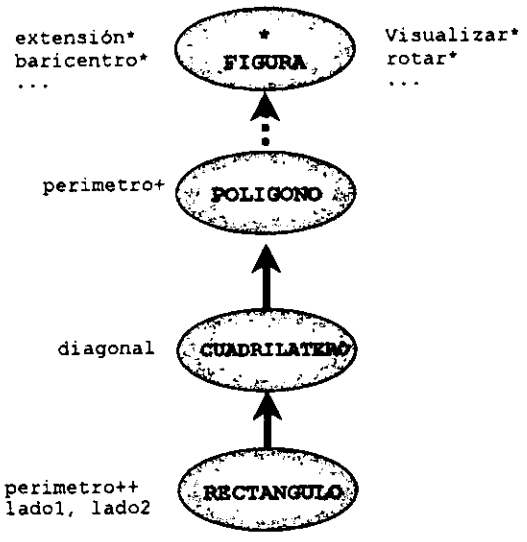
Que la flexibilidad extraordinaria proporcionada por la herencia no se obtenga a costa de la fiabilidad se debe al uso de un enfoque con *comprobación estática de tipos*, en el cual se garantiza en tiempo de compilación que no puedan ocurrir combinaciones de tipos incorrectas en tiempo de ejecución.

Consistencia de tipos

La herencia es consistente con el sistema de tipos. Las reglas básicas son fáciles de explicar según el ejemplo anteriormente expuesto. Considere las siguientes declaraciones:

```
p: POLIGONO
r: RECTANGULO
```

refiriéndose a la jerarquía de herencia anterior, cuya parte relevante es la que se ilustra en la siguiente figura:



Entonces son válidos:

- *p.perimetro*, sin problema, debido a que *perimetro* se definió para polígonos.
- *p.vertices*, *p.trasladar* (..), *p.rotar*(..). con argumentos válidos.
- *r.diagonal*, *r.lado1*, *r.lado2*: las tres características se consideraron declaradas en el nivel *RECTANGULO* o *CUADRILATERO*.
- *r.vertices*, *r.trasladar* (..), *r.rotar*(..): las características se consideraron declaradas en el nivel *POLIGONO* o en un nivel superior, por lo tanto son aplicables a rectángulos, los cuales heredan todas las características de polígonos.
- *r.perimetro*, el mismo caso anterior. La versión original de la función que se llama aquí es la redefinición dada en *RECTANGULO* y no la original de *POLIGONO*.

Sin embargo, las siguientes llamadas a características son ilegales debido a que las características no se consideraron disponibles en el nivel polígono:

```
p.lado1
p.lado2
p.diagonal
```

estos casos son todos resultados de la primera regla fundamental de tipos:

REGLA DE LLAMADA A UNA CARACTERÍSTICA

En una llamada a una característica `x.f`, *donde el tipo de* `x` *se basa en una clase* `C`, *la característica* `f` *debe estar definida en uno de los antecesores de* `C`.

Recuérdese que los antecesores de C incluyen el propio C. La frase "donde el tipo x se basa en una clase C" es una advertencia de que un tipo puede involucrar más de un nombre de clase si la clase es genérica: *LINKED_LIST*[INTEGER] es un tipo de clase "basado en" el nombre de clase *LINKED_LIST*; los parámetros genéricos no forman parte en esta regla.

La regla de llamada a características es estática, esto significa que puede verificarse basándose exclusivamente en el texto del sistema más que a través de controles en ejecución. El compilador (que es típicamente la herramienta que realiza tal verificación) rechazará las clases que contengan llamadas a características que no sean válidas. Si se logra definir un conjunto de reglas de tipos bien ajustados, entonces no habrá riesgo; una vez que un sistema haya sido compilado, su ejecución nunca aplicará una característica a un objeto que no este equipado para tratarlo.

La comprobación estática de tipos es una de las características principales de la tecnología de objetos para alcanzar la meta de la reutilización del software.

Límites al polimorfismo

El polimorfismo no restringido sería incompatible con la noción estática de tipo. La herencia regula qué conexiones polimorfas están permitidas.

Las conexiones polimorfas usadas como ejemplos, tales como *p*:=*r* y *p*:=*t*, tuvieron todas como tipo del origen un descendiente de la clase del destino. Se dice que el tipo es compatible con la clase del destino; por ejemplo *CUADRADO* es compatible con *RECTANGULO* y *POLIGONO*; pero no con *TRIANGULO*. Esta noción ya se ha usado informalmente pero debe darse una definición precisa:

COMPATIBILIDAD O CONFORMIDAD DE TIPOS

Un tipo U *es compatible o conforme con un tipo* T *sólo si la clase base de* U *es un descendiente de la clase base de* T; *además, para los tipos derivados genéricamente, todo parámetro real de* U *debe (recursivamente) ser compatible con el correspondiente parámetro formal en* T.

Con la genericidad se ha tenido que hacer una distinción técnica entre tipos y clases. Cada tipo tiene una clase base, la cual en ausencia de la genericidad es el propio tipo (por ejemplo *POLIGONO* es su propia clase base), pero para un tipo derivado genéricamente es la clase a partir de la cual se construye el tipo, por ejemplo la clase base de *LIST*[*POLIGONO*] es *LIST*. La segunda parte de la definición indica que B[Y] será compatible con A[X] si B es un descendiente de A e Y es un descendiente de X.

Obsérvese como toda clase es un descendiente de ella misma, de ese modo todo tipo es compatible consigo mismo.

Con esta generalización de la noción de descendiente se obtiene la segunda regla fundamental de tipos:

REGLA DE COMPATIBILIDAD DE TIPOS

Una conexión con origen x *y destino* y (*esto es, una asignación* x := y, *o el uso de* y *como un argumento real de la llamada a una rutina donde el correspondiente argumento formal es* x) *es válido solamente si el tipo de* y es compatible con el tipo x.

La regla de compatibilidad de tipos expresa que se puede asignar un tipo más específico a uno más general, pero no inversamente. Por tanto `p:=r` es válido pero no `r:=p`.

Instancias

Con la introducción del polimorfismo se necesita una terminología más específica para hablar sobre las instancias. Informalmente, las instancias de una clase son objetos creados en tiempo de ejecución de acuerdo con la definición de una clase. Pero también ahora se deben considerar los objetos construidos a partir de la definición de sus descendientes propios. Una definición más exacta es:

INSTANCIA DIRECTA, INSTANCIA

Una instancia directa de una clase C *es un objeto que se obtienen de acuerdo con la definición exacta de* C, *a través de una instrucción de creación* `!!x...` *donde el receptor* x *es de tipo* C (*o, recursivamente, por duplicación de una instancia directa de* C). *Una instancia de* C *es una instancia directa de un descendiente de* C.

La última parte de la definición implica, debido a que los descendientes de una clase incluyen la propia clase, que una instancia directa de C es también una instancia de C.

Por tanto la ejecución de

```
p1, p2: POLIGONO; r: RECTANGULO
...
!!p1...; !!r...; p2:=r
```

creará dos instancias de *POLIGONO* pero sólo una instancia directa (la instancia de *p1*). El otro objeto es una instancia directa de *RECTANGULO* y por tanto una instancia de *POLIGONO* y *RECTANGULO*.

Aunque las nociones de instancia e instancia directa se han definido para una clase, se extienden inmediatamente para cualquier tipo (con una clase base y los posibles parámetros genéricos).

El polimorfismo significa que una entidad de un cierto tipo podría conectarse no sólo a instancias directas de ese tipo, sino a instancias arbitrarias. Se podría verdaderamente considerar que el papel de la regla de compatibilidad de tipos es asegurar la siguiente propiedad:

CONSISTENCIA DE TIPOS ESTÁTICOS Y DINÁMICOS

Una entidad declarada de tipo T *podría conectarse en tiempo de ejecución a instancias de* T.

El tipo estático, el tipo dinámico

El nombre de la última propiedad sugiere el concepto de "tipo estático" y "tipo dinámico". El tipo usado para declarar una entidad es el *tipo estático* de la referencia correspondiente. Si, en ejecución, la referencia se conecta a un objeto de un cierto tipo, este tipo se convierte en el *tipo dinámico* de la referencia.

De este modo con la declaración `p: POLIGONO`, el tipo estático de la referencia que `p` denota es `POLIGONO`; después de la ejecución de `!!p`, el tipo dinámico de esa referencia es también `POLIGONO`; después de la asignación `p:=r`, con `r` de tipo `RECTANGULO` y no vacío, el tipo dinámico es `RECTANGULO`.

La regla de compatibilidad de tipos establece que el tipo dinámico debe siempre ser compatible con el tipo estático.

Para evitar cualquier confusión se debe recordar que se está tratando con tres niveles: una entidad es un identificador en el texto de la clase; en ejecución su valor es una *referencia*; la referencia puede conectarse a un *objeto*. Entonces:

- Un objeto sólo tiene un tipo dinámico, el tipo con el cual ha sido creado el objeto. Así el tipo nunca cambiará durante el tiempo de vida del objeto.
- En cualquier momento de la ejecución, una referencia tiene un tipo dinámico, el tipo del objeto con el cual se conecta la referencia. El tipo dinámico puede cambiar como resultado de operaciones de reconexión.
- Sólo una entidad es la que tiene ambos tipos, estático y dinámico. Su tipo estático es el tipo con el cual la entidad se declaró: `T` si la declaración fue `x:T`. Su tipo dinámico en un instante dado de la ejecución es el tipo del valor de la referencia, que significa el tipo de objeto que tiene conectado.

3.4.4. Ligadura dinámica

Las operaciones definidas para todos los polígonos no necesitan implementarse idénticamente para todas las variantes. Por ejemplo, `perimetro` tiene diferentes versiones para los polígonos generales y para los rectángulos, se les llamará entonces $perimetro_{POL}$ y $perimetro_{RECT}$. La clase `CUADRADO` también tendrá su propia variante (multiplicando la longitud del lado por cuatro). Se pueden imaginar variantes adicionales para otros tipos especiales de polígonos. Esto plantea inmediatamente una pregunta fundamental: ¿qué sucede cuando se aplica a una entidad polimorfa una rutina con más de una versión?

En un fragmento como:

```
!!p.crear (...); x:=p.perimetro
```

está claro que se aplicará $perimetro_{POL}$. Como está tan claro que en

```
!!r.crear (...).; x:=r.perimetro
```

se aplicará $perimetro_{RECT}$. Pero qué sucede si la entidad polimorfa p, declarada estáticamente como un polígono, se refiere dinámicamente a un rectángulo. Suponga que se ha ejecutado

```
!!r.crear (...).
p:=r
x:=p.perimetro
```

La regla conocida como **ligadura dinámica** establece que **la forma dinámica del objeto** determina la versión de la operación que se aplicará. Aquí será $perimetro_{RECT}$.

Por supuesto, como puede observarse, el caso más interesante sucede cuando no se puede deducir a partir de una simple lectura de texto del software qué tipo dinámico exacto tendrá p en tiempo de ejecución, como en

```
      --Calcula el perímetro de una figura construida de acuerdo
      --con la selección del usuario
p:POLIGONO
...
if icono_elegido=icono_rectangulo then
      !RECTANGULO!p.crear (..).
elseif icono_elegido=icono_triangulo then
      !TRIANGULO!p.crear (..).
elseif
      ...
end
...
x:=p.perimetro
```

o después de una asignación polimorfa condicional **if** ... **then** *p* := *r* **elseif** ... **then** *p* := *t*...; o si *p* es un elemento de un array polimorfo de polígonos; o simplemente si *p* es un argumento formal, declarado de tipo *POLIGONO*, de la rutina que incluye la llamada – a la cual los que la invocan pueden pasarle argumentos reales de cualquier tipo que sea compatible con *POLIGONO*.

Entonces dependiendo de lo que suceda en una ejecución particular, el tipo dinámico de *p* será *RECTANGULO*, o *TRIANGULO* y así sucesivamente. No se tiene manera de saber cuál de estos casos se tendrá. Pero gracias a la ligadura dinámica no se necesita saber: sea lo que fuere *p*, la llamada ejecutará la variante apropiada de *perimetro*.

Esta capacidad de las operaciones para adaptarse automáticamente a los objetos a los cuales se aplican es una de las propiedades más importantes de los sistemas orientados a objetos.

Puede temerse que la ligadura dinámica sea un mecanismo costoso, que requiera una búsqueda en tiempo de ejecución en el grafo de la herencia, lo cual implicaría un coste adicional que aumenta con la profundidad de ese grafo. Afortunadamente éste no es el caso con un lenguaje O-O diseñado correctamente (y con comprobación estática de tipos). La eficiencia de la ligadura dinámica no es un problema cuando se trabaja en un buen entorno de desarrollo.

3.4.5. Clases y características diferidas

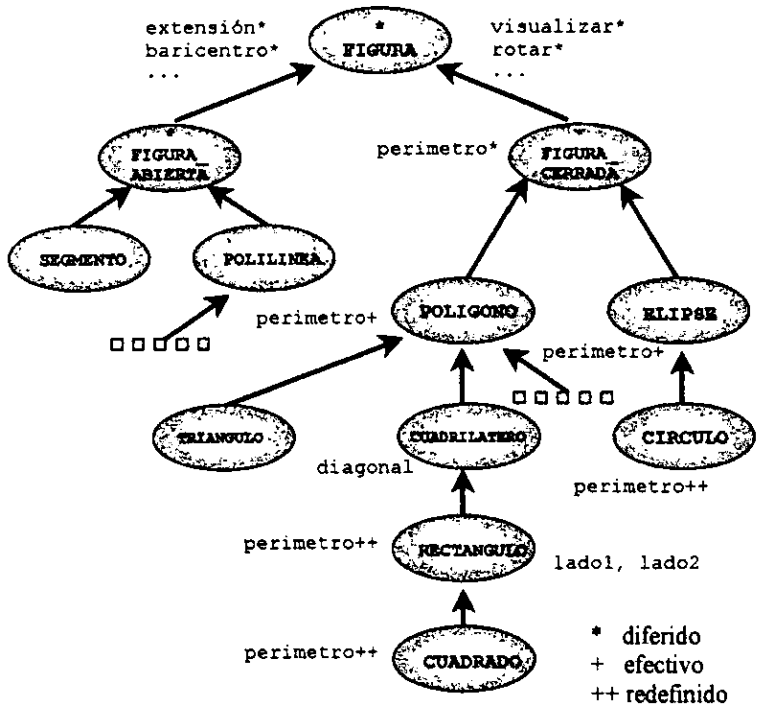
El polimorfismo y la ligadura dinámica nos permiten confiar en las abstracciones durante el diseño del software, y estar seguros de que la ejecución elegirá las implementaciones apropiadas. Pero no siempre se necesita que las cosas estén totalmente implementadas. Los elementos abstractos de software, parcialmente implementados o no implementados en absoluto, ayudan en muchas tareas: analizar el problema y diseñar la arquitectura (en cuyo caso se pueden mantener en el producto final para funcionar como documentación del análisis y diseño); capturar las semejanzas entre las implementaciones; describir los nodos intermedios en una clasificación.

Las características y clases diferidas proporcionan el mecanismo de abstracción necesario.

Moviendo figuras arbitrarias

Para entender la necesidad de rutinas y clases diferidas, considérese nuevamente la jerarquía *FIGURA*, reproducida por comodidad a continuación.

El concepto más general es *FIGURA*. Contando con los mecanismos del polimorfismo y la ligadura dinámica, se puede querer aplicar el esquema descrito anteriormente, como en:



```
transformar (f: FIGURA) is
      --Aplica una transformación especifica a f
   ... do
       f.rotar (..).
       f.trasladar(..).
       end
```

con los valores apropiados para los argumentos que faltan. Entonces todas las llamadas siguientes son válidas:

```
transformar ( r)                    --con r: RECTANGULO
transformar ( c)                    --con c: CIRCULO
transformar (figarray.item (i))     --con figarray: ARRAY[POLIGONO]
```

en otras palabras se quiere aplicar *rotar* y *trasladar* a la figura *f*, y dejar que sea el mecanismo de ligadura dinámica subyacente el que elija la versión apropiada (diferente para las clases *RECTANGULO* y *CIRCULO*) dependiendo de la forma actual de *f*, que sólo es conocida en tiempo de ejecución.

Esto debiera funcionar, y es un ejemplo típico de la forma en que el polimorfismo y la ligadura dinámica hacen posible un estilo elegante, aplicando el principio de Elección Única. Simplemente, lo que habría que hacer es redefinir *rotar* y *trasladar* para las clases implicadas.

Pero no hay nada que redefinir: *FIGURA* es un concepto muy general, abarca todos los tipos de figuras de dos dimensiones. No se tiene forma de escribir una versión de propósito general de *rotar* y *trasladar* sin más información sobre las figuras implicadas.

Por tanto he aquí una situación donde la rutina *transformar* se ejecutaría correctamente gracias a la ligadura dinámica, pero es estáticamente ilegal debido a que *rotar* y *trasladar* no son características válidas de *FIGURA*. La verificación de tipo tomará *f.rotar* y *f.trasladar* como operaciones no válidas.

Se podría por supuesto, introducir en el nivel de *FIGURA* un procedimiento *rotar* que no haga nada. Pero esto es un camino peligroso a seguir; *rotar(centro, angulo)* tiene una semántica bien definida y "no hacer nada" no es una implementación apropiada para el mismo.

Diferir una característica

Lo que se necesita es una manera de especificar *rotar* y *trasladar* a nivel de *FIGURA* que exija que los descendientes tengan que proporcionar implementaciones concretas. Esto se logra declarando las características como "diferidas" (*deferred*). Se reemplaza la parte de instrucciones del cuerpo de la característica (**do** instrucciones) por la palabra clave **deferred**. La clase *FIGURA* declararía:

```
rotar (centro: PUNTO; angulo: REAL) is
          --Rotar cierto ángulo alrededor del centro.
      deferred
      end
```

y lo mismo para *trasladar*. Esto significa que la característica es conocida en la clase en que aparece esta declaración, pero sólo está implementada en los descendientes propios. Entonces una llamada *f.rotar* en el procedimiento *transformar* será válida.

Con tal declaración, *rotar* es lo se denomina una *característica diferida*. Una característica no diferida – una característica que tiene una implementación – se dice que es **efectiva** (*effective*).

Hacer efectiva una característica

En algunos descendientes propios de *FIGURA* se reemplazará la versión diferida por una efectiva. Por ejemplo:

```
class POLIGONO inherit
      FIGURA_CERRADA
feature
      rotar (centro: PUNTO; angulo: REAL) is
            --Rotar cierto ángulo alrededor del centro.
        do
            ... Instrucciones para rotar todos los vértices.
        end
      ...
end   --clase POLIGONO
```

Observe que *POLIGONO* hereda las características de *FIGURA* no directamente sino a través de *FIGURA_CERRADA*; el procedimiento *rotar* sigue siendo diferido en *FIGURA_CERRADA*.

Este proceso de proporcionar una versión efectiva de una característica que es diferida en el padre se llama **"hacer efectiva"**.

Una clase que haga efectiva una o más características heredadas no necesita enumerarlas en su subcláusula **redefine**, debido a que no hubo una verdadera redefinición (en el sentido de una implementación) en el lugar original. La clase proporciona simplemente una declaración efectiva de las características, que deben ser de tipos compatibles con el original, como en el ejemplo de *rotar*.

Hacer efectiva una característica es por supuesto parecido a la redefinición, e independientemente de que las características que se hacen efectivas no se enlistan en la subcláusula **redefine** por lo demás estarán regidas por las mismas reglas. He aquí la necesidad de un término común:

REDECLARACIÓN

Redeclarar una característica es redefinirla o hacerla efectiva.

Los ejemplos usados para introducir la redefinición y hacer efectiva una característica ilustran las diferencias entre estas dos formas de redeclaración:

- Cuando se pasa de *POLIGONO* a *RECTANGULO*, ya se ha hecho una implementación de `perimetro` en el padre; se quiere ofrecer una nueva implementación en *RECTANGULO*. Esto es una redefinición. Observe que la característica se encuentra nuevamente redefinida en *CUADRADO*.
- Cuando se pasa de *FIGURA* a *POLIGONO*, no tenemos implementación de `rotar` en el padre; se quiere ofrecer una implementación en *POLIGONO*. Esto es hacer efectiva una característica. Los descendientes propios de *POLIGONO* pueden por supuesto redefinir la versión efectiva.

Puede existir la necesidad de cambiar algunas propiedades de una característica heredada y seguir dejándola deferida. Estas propiedades no pueden incluir la implementación de la característica (debido a que no tiene implementación), pero pueden incluir la signatura de la característica – el tipo de sus argumentos y del resultado – y sus aserciones. A diferencia de una redefinición de característica diferida a una efectiva, tal redefinición de diferida a diferida se considera una redefinición que requiere de una cláusula **`redefine`**. El siguiente cuadro resume los cuatro posibles casos de redefinición.

Redeclaración de → a↓	Diferido	Efectivo
Diferido	Redefinir	Indefinir
Efectivo	Hacer efectiva	Redefinir

Esto muestra un caso que es poco común: la indefinición, o redefinición de una característica efectiva a una diferida – olvidarse de su implementación original para comenzar de nuevo.

Clases diferidas

Una característica, como se ha visto, es diferida o efectiva. Esta situación se extiende a las clases:

CLASE EFECTIVA, CLASE DIFERIDA

Una clase es diferida si tiene una característica diferida. Una clase es efectiva si no tiene características diferidas.

Por tanto, para que una clase sea efectiva, todas sus características deben ser efectivas. Una o más características diferidas hacen la clase diferida. En el segundo caso se debe marcar la clase:

REGLA DE DECLARACIÓN DE CLASES DIFERIDAS

La declaración de una clase debe utilizar la palabra clave yuxtapuesta **deferred class** *(a diferencia de sólo* **class** *para una clase efectiva).*

Así la clase figura se declarará como:

```
deferred class FIGURA feature
      rotar (..). is
            ... Declaración de una característica diferida como antes
            ... otras declaraciones de características ...
end   --clase FIGURA
```

A la inversa, si una clase se señala como **deferred** debe tener al menos una característica diferida. Pero una clase puede ser diferida aunque no declare ninguna característica diferida propia: en el ejemplo, la clase *FIGURA_ABIERTA* muy probablemente no hace efectivas a *visualizar*, *rotar* y otras características diferidas que hereda de *FIGURA*, debido a que la noción de figura no es todavía lo suficientemente concreta para incluir implementaciones por defecto de estas operaciones. Por lo que es diferida y se declarará como:

```
deferred class FIGURA_ABIERTA inherit
      FIGURA
...
```

aun si la clase no introduce una característica diferida.

Un descendiente de una clase diferida es una clase efectiva si proporciona definiciones efectivas para todas las características diferidas presentes en sus padres, y no introduce ninguna característica diferida propia. Las clases efectivas tales como *POLIGONO* y *ELIPSE* deben proporcionar implementaciones de *visualizar*, *rotar* y otras rutinas que hayan heredado diferidas.

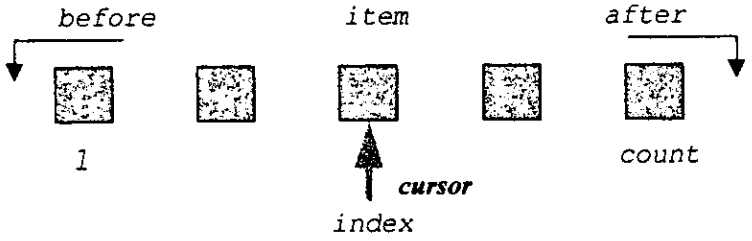
Por comodidad, se dirá que un tipo es diferido si su clase base es diferida. Por lo que *FIGURA*, vista como tipo, es diferida; y si la clase genérica *LIST* es diferida – como debiera ser si ella representa listas generales independientemente de la implementación – el tipo *LIST*[INTEGER] es diferido. Solamente la clase base cuenta aquí: C[X] es efectiva si la clase C es efectiva y diferida si C es diferida, independientemente del status de X.

Especificar la semántica de las características y clases diferidas

Aunque una característica diferida no tiene implementación, y una clase diferida tampoco tiene implementación o tiene una implementación parcial, será necesario con frecuencia expresar sus propiedades semánticas abstractas. Se pueden utilizar las aserciones para este propósito.

Como cualquier otra clase, una clase diferida puede tener un invariante de clase; y una característica diferida puede tener una precondición, una postcondición o ambas.

Considere el ejemplo de las listas secuenciales, descritas independientemente de una implementación particular. Como con muchas otras estructuras, es conveniente asociar a cada lista un cursor, que indica la posición activa actual:

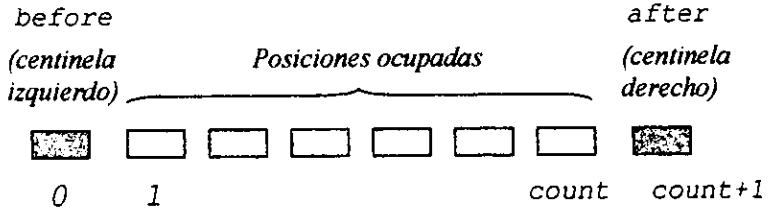


La clase es diferida:

```
indexing
      description: "listas que se recorren secuencialmente"
deferred class
      LIST[T]
feature  --Acceso
     count: INTEGER is
       deferred
       end
     index: INTEGER is
        --Posición del cursor
       deferred
       end
     item: G is
        --Elemento en la posición del cursor
       deferred
       end
feature  --Informar sobre el estado
      after: BOOLEAN is
         --¿Está el cursor después del último elemento?
       deferred
       end
      before: BOOLEAN is
         --¿Está el cursor antes del primer elemento?
       deferred
       end
feature   --Mover el cursor
      forth is
         --Avanza el cursor una posición
       require
         not after
       deferred
       ensure
         index= old index + 1
       end
       ... otras características ...
invariant
       cantidad_no_negativa: count>=0
       queda_al_menos_uno: index>=0
```

```
        queda_al_menos_un_hueco: index <= count + 1
        definicion_de_siguiente: after = (index = count + 1)
        definicion_de_anterior: before = (index = 0)
end     --clase LIST
```

El invariante expresa las relaciones entre diferentes consultas. Las dos primeras cláusulas expresan que el cursor puede sólo salirse del conjunto de elementos una posición a la izquierda o a la derecha:



Posiciones del cursor

Las aserciones de *forth* expresan precisamente qué debe hacer este procedimiento: avanzar el cursor una posición. Debido a que se quiere mantener el cursor dentro del rango de los elementos de la lista, más dos posiciones "centinelas" según se indicaba en la última figura, la aplicación de *forth* requiere **not** *after*; el resultado, como expresa la postcondición, es incrementar *index* en uno.

Otro ejemplo puede ser la pila. Una biblioteca necesitaría una clase general *STACK*[G], la cual será diferida debido a que debe cubrir todas las posibles implementaciones; los descendientes propios tales como *FIXED_STACK* o *LINKED_STACK* describirán implementaciones específicas. Uno de los procedimientos diferidos de *STACK* es *put*:

```
put (x: G) is
            --Añade x sobre la cima.
        require
            not full
        deferred
        ensure
            no_vacio: not empty
            en_la_cima: item = x
            un_elemento_mas: count = old count + 1
        end
```

Las funciones booleanas *empty* y *full* (también diferidas a nivel de *STACK*) expresan si la pila está vacía y si su representación está llena.

Sólo las **aserciones** hacen que las clases diferidas muestren toda su potencia. Según puede observarse, las **precondiciones y postcondiciones** se aplican a todas las redeclaraciones de una rutina. Esto es especialmente significativo en el caso diferido: estas aserciones, si se presentan, fijarán los límites para todas las formas efectivas posibles de la rutina. Así las especificaciones anteriores para *put* restringen todas las variantes de esta característica en los descendientes de *STACK*.

Gracias a estas técnicas de aserciones se puede hacer a las clases diferidas informativas y ricas en semántica, aun cuando no prescriban ninguna implementación.

3.4.6. Una técnica de redeclaración

La posibilidad de redeclarar una característica – redefiniéndola o haciéndola efectiva – proporciona un estilo de desarrollo flexible e incremental. Existe una forma que le añade potencia:

- La capacidad de redeclarar una función como atributo.

Las técnicas de redeclaración ofrecen una aplicación avanzada de uno de los principios centrales la modularidad que nos conduce al método orientado a objetos: el acceso uniforme.

El principio de acceso uniforme expresa que no debería existir ninguna diferencia fundamental, desde la perspectiva del cliente, entre un atributo y una función sin argumentos. En ambos casos la característica es una consulta; que sólo se diferencian en su representación interna.

El primer ejemplo era una clase que describía cuentas bancarias, donde la característica *saldo* podía implementarse como una función, que suma todos los ingresos y sustrae todos los retiros, o como un atributo, actualizado siempre que sea necesario reflejar el saldo actual. Para el cliente, esto no tiene diferencia excepto posiblemente por su rendimiento.

Con la herencia, se puede ir más lejos, y permitir que una clase que hereda una rutina pueda *redefinirla como atributo*.

El ejemplo inicial es aplicable directamente. Considere una clase original *CUENTA1*:

```
class CUENTA1 feature
    saldo: INTEGER is
            --Saldo actual
      do
        Result:= lista_de_depositos.total - lista_de_retiros.total
      end
    ...
end   --clase CUENTA1
```

Entonces un descendiente puede elegir la segunda implementación del ejemplo original, redefiniendo *saldo* como un atributo:

```
class CUENTA2 inherit
      CUENTA1
         redefine saldo end
feature
      saldo: INTEGER
            --Saldo actual
      ...
end   --clase CUENTA2
```

CUENTA2 tendrá posiblemente que redefinir ciertos procedimientos, tales como extraer e ingresar, por lo que además de sus otras obligaciones deben actualizar saldo, para mantener invariante la propiedad

```
saldo:= lista_de_depositos.total - lista_de_retiros.total
```

En este ejemplo la redeclaración es una *redefinición*. Hacer efectiva una característica puede también convertir una característica diferida en un atributo. Por ejemplo una clase *LIST* diferida podría tener una característica

```
count: INTEGER is
            --Número de elementos insertados
        deferred
        end
```

entonces una implementación de un array podría hacer efectiva esta característica como un atributo:

```
count: INTEGER
```

Combinadas con el polimorfismo y la ligadura dinámica, tales redeclaraciones de rutinas en atributos llevan el principio de acceso uniforme a su extremo. No solamente puede implementarse una solicitud del cliente de la forma *a.servicio* a través del almacenamiento en memoria (como atributo) o a través de un cómputo (como función) sin requerir que el cliente sea consciente de la selección (la idea básica del acceso uniforme) sino que se tiene ahora una situación donde la misma llamada podría, en sucesivas ejecuciones de la solicitud durante una única sesión, provocar un acceso a un campo en algunos casos y una llamada a una rutina en otros. Esto podría por ejemplo suceder con ejecuciones sucesivas de la misma llamada a *a.saldo*, si mientras tanto *a* se reconecta polimórficamente a diferentes objetos.

Finalmente, podría pensarse que es posible redefinir un atributo como una función sin argumentos. Pero no; la asignación, una operación aplicable a atributos, no tiene sentido para funciones. Supóngase que *a* es un atributo de una clase C, y una rutina de C contiene la instrucción

```
a:= una_expresión
```

si un descendiente de C redefine *a*, entonces la rutina – suponiendo que ella no se redefina también – sería inaplicable, debido a que no puede hacer una asignación a una función. Esta falta de simetría (la redeclaración permitida de funciones a atributos pero no inversamente) es desafortunada pero inevitable.

Esto hace que el uso de un atributo sea como una implementación final, no reversible, mientras que a la inversa, utilizar una función deja aún la posibilidad de que luego se dé una implementación basada en almacenamiento (en lugar de estar basada en cómputo).

3.4.7. El significado de la herencia

Una vez expuestas las técnicas básicas de la herencia, es necesario entender sus conceptos fundamentales, su papel dentro de la búsqueda de la calidad del software y de un proceso efectivo de desarrollo de software.

Es posible que en el estudio de la herencia aparezca con mayor claridad que en ninguna otra parte el papel dual de las clases como módulos y tipos. Desde el punto de vista modular, un descendiente describe una extensión del módulo padre; desde el punto de vista de tipos, un descendiente describe un subtipo del tipo del padre.

Aunque algunos aspectos de la herencia se corresponden más con el enfoque de tipos, la mayoría son útiles para ambos puntos de vista.

El punto de vista modular

Desde el punto de vista modular, la herencia es particularmente efectiva como una técnica de reutilización.

Un módulo es un conjunto de servicios ofrecidos al exterior. Sin la herencia, cada nuevo módulo debe definir todos los servicios que ofrece. Por supuesto, la implementación de estos servicios podría confiar en los servicios proporcionados por los otros módulos: éste es el propósito de la relación cliente. Pero no hay forma de definir un nuevo módulo añadiendo simplemente los nuevos servicios de los módulos definidos previamente.

La herencia ofrece esa posibilidad. Si `B` hereda de `A`, todos los servicios (características) de `A` están automáticamente disponibles en `B`, sin la necesidad de definirlos nuevamente. `B` es libre de añadir nuevas características para sus propios propósitos específicos. La redefinición proporciona un grado adicional de flexibilidad, que permite a `B` tomar lo mejor de las implementaciones que ofrece, conservando algunas tal y como son, mientras rechaza otras sustituyéndolas por versiones más apropiadas.

Esto conduce a un estilo de desarrollo de software que, en lugar de tratar de resolver cada nuevo problema desde el principio, estimula la construcción basada en logros anteriores y la extensión de sus resultados.

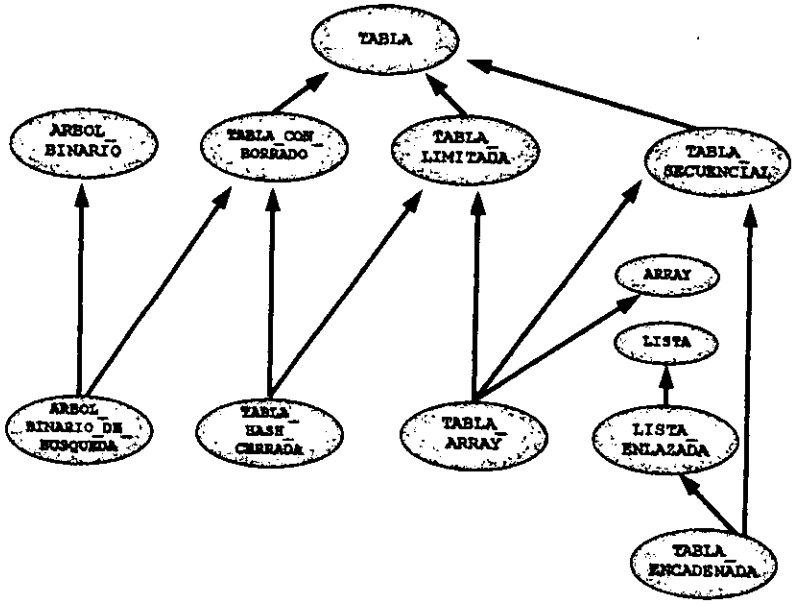
El beneficio exacto de este enfoque se puede entender mejor en los términos del principio Abierto-Cerrado. El principio establece que una buena estructura modular debiera ser a la vez cerrada y abierta:

- Cerrada, porque los clientes necesitan los servicios de los módulos para seguir adelante con sus propios desarrollos, y una vez han optado por una versión del módulo no deberían verse afectados por la introducción de nuevos servicios que ellos no necesiten.
- Abierta, porque no hay garantía de que se incluirán exactamente desde el inicio todos los servicios potencialmente útiles para los clientes.

Este doble requisito parece un dilema, y la estructura modular clásica no ofrece solución. Pero la herencia lo resuelve. Una clase es cerrada, debido a que puede ser compilada, almacenada en una biblioteca, y usada por las clases clientes. Pero es también abierta, debido a que una nueva clase puede usarla como padre, añadiendo

nuevas características y redeclarando las características heredadas. En este proceso no hay necesidad de cambiar el original ni afectar a sus clientes. Esta propiedad es fundamental en la aplicación de la herencia para la construcción de software reutilizable y extensible.

Entre los problemas más difíciles en el diseño de estructuras modulares reutilizables se contaba la necesidad de aprovechar las semejanzas que pueden existir entre los grupos de abstracciones de datos afines[4] – todas las tablas hash, todas las tablas secuenciales, etc. Usando estructuras de clases conectadas por la herencia, se pueden lograr beneficios de las relaciones lógicas que existen entre estas implementaciones. El diagrama siguiente es un esbozo parcial y a grandes rasgos de una posible estructura para una biblioteca para el manejo de tablas.



Esbozo de una estructura para una biblioteca de tablas

Con este punto de vista se puede expresar el requisito de reutilización bastante concretamente: la idea es situar en el diagrama la definición de cada característica lo más arriba que sea posible a fin de que pueda ser compartida por el mayor número posible de clases descendientes. Este proceso implica mover las características a lo más alto que sea posible dentro la jerarquía de clases, esto como resultado de descubrir las abstracciones de más alto nivel y de paso fusionar la mayor cantidad de estas características como resultado de descubrir semejanzas.

[4] vid supra. pág. 78.

El punto de vista de tipo

Desde la perspectiva de los tipos, la herencia trata la reutilización y la extensibilidad, que se abarcaron en algún momento dentro del término modularidad. La clave es la ligadura dinámica.

Un tipo es un conjunto de objetos caracterizado (según se establece en la teoría de tipos abstractos de datos) por ciertas operaciones. El tipo *INTEGER* describe un conjunto de números con operaciones aritméticas, el tipo *POLIGONO*, un conjunto de objetos con las operaciones *vertices*, *perimetro* y otras.

Para los tipos, la herencia se representa mediante la relación *es*, también conocida como *es-un*, como "todo perro es un mamífero", "todo mamífero es un animal". Similarmente, todo rectángulo es un polígono.

Esta relación significa:

- Si se consideran los valores en cada tipo, la relación es simplemente la inclusión de conjuntos: los perros constituyen un subconjunto del conjunto animales; de igual manera, las instancias de *RECTANGULO* constituyen un subconjunto de las instancias de *POLIGONO*.
- Si se consideran las operaciones aplicables a cada tipo, decir que todo B *es un* A significa que cada operación aplicable a las instancias de A es aplicable también a las instancias de B. (con la redefinición, sin embargo, B puede poseer su propia implementación, la cual para instancias de B reemplaza la implementación dada en A.)

Usando esta relación, se puede describir redes *es-un* que representan muchas variantes posibles de un tipo, tales como todas las variantes de *FIGURA*. Cada nueva versión de rutinas tales como *rotar* y *visualizar* se define en la clase que describe la correspondiente variante del tipo. En el ejemplo de las tablas, cada clase del gráfico proporciona su propia implementación de *buscar*, *insertar*, *eliminar*, excepto por supuesto cuando la versión del padre siga siendo apropiada.

Una advertencia sobre el uso de "*es*" y "*es-un*". Algunas veces se emplea mal la herencia para modelar la relación instancia-a-molde, tal como con una clase *SAN_FRANCISCO* que hereda de *CIUDAD*. Esto tiene muchas probabilidades de ser un error: *CIUDAD* es una clase, que puede tener una instancia que represente a San Francisco. Para evitar tales errores, es suficiente recordar que el término es-un no se aplica para "*x* es un A" (como en "*SAN_FRANCISCO* es una *CIUDAD*"), una relación entre una instancia y una categoría, sino para "todo B es un A" (como en "Toda *CIUDAD* es una *UNIDAD_GEOGRAFICA*"). Esto es una relación entre dos categorías – dos clases en los términos del software. Algunos autores prefieren llamar a esta relación "*es un tipo de*" o "*puede actuar como un*", esto es en cierto modo cuestión de gusto.

La herencia y la descentralización

Con la ligadura dinámica se pueden producir arquitecturas de software descentralizadas necesarias para lograr las metas de la reutilización y la extensibilidad.

Simplemente compárese el enfoque O-O con los enfoques clásicos. En Pascal, se puede utilizar un tipo registro con variantes:

```
type FIGURA =
   record
      "Campos comunes"
   case figtipo: (poligono, rectangulo, triangulo, circulo, ...) of
      poligono: (vertices: LISTA_DE_PUNTOS; cantidad:INTEGER);
      ...
   end
```

para definir las diferentes formas de figuras. Pero esto significa que cada rutina que maneja figuras (rotar y semejantes) debe distinguir entre las posibilidades:

```
case f.figura_tipo of
    poligono: ...
    circulo: ...
    ...
end
```

Las rutinas *buscar* y otras en el caso de la tabla usarían la misma estructura. El problema es que todas estas rutinas poseen demasiado **conocimiento** acerca del sistema global: cada una debe conocer exactamente qué tipo de figuras se permiten en el sistema. Cualquier adición de un nuevo tipo, o cambio en un tipo existente, afectará a cada rutina. Una rutina de rotación no tiene obligación de conocer la lista exhaustiva de tipos de figuras. Debiera serle suficiente la información necesaria para hacer su trabajo: rotar cierto tipo de figuras.

Esta distribución del conocimiento entre muchas rutinas es el origen principal de la inflexibilidad en los enfoques clásicos del diseño del software. Muchas de las dificultades de la modificación del software pueden encontrase en este problema. Además, esto explica en parte por qué los proyectos de software son tan difíciles de mantener bajo control, y cómo unos cambios aparentemente pequeños tienen consecuencias de gran repercusión, forzando a los desarrolladores de software a reabrir módulos que se consideraron buenos para tenerlos definitivamente cerrados.

Las técnicas orientadas a objetos tratan el problema inteligentemente. Un cambio en una implementación de una operación particular sólo afectará la clase que incluye la implementación. La adición de una nueva variante de tipo en muchos casos no afectará en nada a las otras. La descentralización es la clave: las clases manejan sus propias implementaciones y no se entrometen en asuntos de otras.

Independencia de la representación

La ligadura dinámica también trata uno de los problemas principales de la reutilización: la independencia de la representación[5] – la capacidad para solicitar una operación con más de una variante, sin conocer qué variante se aplicará. La discusión de esta noción en el capítulo II empleaba el ejemplo de una llamada

```
presente:= tiene (x, t)
```

[5] vid supra, pág. 77.

que debería utilizar el algoritmo de búsqueda apropiado dependiendo en ejecución de la forma de `t`. Con la ligadura dinámica, se tiene exactamente que: si `t` se declara como una tabla, pero puede ser instanciada como un árbol de búsqueda binaria, una tabla hash cerrada, etc. (suponiendo que estén disponibles todas las clases necesarias) entonces la llamada

```
presente:= t.tiene (x)
```

encontrará, en ejecución, la versión apropiada de `tiene`. La ligadura dinámica logra entonces que un cliente pueda solicitar una operación, y dejar que el sistema subyacente al lenguaje sea el que encuentre automáticamente la implementación apropiada.

De este modo la combinación de clases, la herencia, la redefinición, el polimorfismo y la ligadura dinámica proporcionan un conjunto extraordinario de respuestas a las cuestiones planteadas en los capítulos anteriores: los requisitos para la reutilización; criterios, principios y reglas para la modularidad.

La paradoja de la extensión-especialización

La herencia se ve algunas veces como extensión y otras como especialización. Aunque estas dos interpretaciones perecen contradictorias, hay verosimilitud en ambas – pero no a partir de la misma perspectiva.

Todo depende, nuevamente, si se ve a una clase como un tipo o como un módulo. En el primer caso, la herencia, o el término *es*, es claramente especialización; "perro" es una noción más especializada que "animal", y "rectángulo" que "polígono". Esto se corresponde, como se indicaba, con la inclusión de subconjuntos: si `B` hereda de `A`, el conjunto de objetos en tiempo de ejecución representados por `B` es un subconjunto del conjunto correspondiente para `A`.

Pero a partir de la perspectiva modular, donde una clase se ve como un proveedor de servicios, `B` implementa los servicios (características) de `A` más los suyos propios. *Menos* objetos a menudo permiten *más* características, debido a que implican un valor de información mayor; pasando de los animales arbitrarios a los perros se puede añadir la propiedad específica de *ladrar*, y a partir de polígonos arbitrarios a rectángulos se puede añadir la característica `diagonal`. De este modo con respecto a las características implementadas, los subconjuntos pueden verse de otra forma: las características aplicables a las instancias de `A` son un subconjunto de aquellas aplicables a instancias de `B`.

La herencia, entonces, es una especialización desde el punto de vista de los tipos y una extensión desde el punto de vista modular. Ésta es la paradoja de la extensión-especialización: más características a aplicar, por lo que menos objetos a qué aplicarlas.

La paradoja de la extensión-especialización es una de las razones para evitar el término "subclase", que sugiere "subconjunto". Otra, que se ha observado, es la confusión que a veces se encuentra en la literatura en el uso de "subclase" para indicar tanto herencia directa como herencia indirecta. Ese problema no se presenta para los términos definidos precisamente *clase heredera, descendiente* y *descendiente propio* y sus inversas *padre, antecesor* y *antecesor propio*.

METODOLOGÍA ORIENTADA A OBJETOS: APLICACIÓN DEL MÉTODO

Este último capítulo se enfocará a demostrar las capacidades que posee el método orientado a objetos para enfrentar los problemas comunes de los sistemas computacionales. Las siguientes secciones abarcarán una forma práctica de identificar y estructurar las clases; usar la herencia; como adoptar un enfoque orientado a objetos para el análisis y finalmente explorar un modelo de proceso alternativo para el ciclo de vida del software. Todo esto matizado con el consabido interés por la reutilización.

4.1. Patrón de diseño: sistemas interactivos Multi-panel

Se iniciara la aplicación del método orientado a objetos con un ejemplo para un patrón de diseño que, además de ilustrar algunas propiedades típicas del método, brinda una oportunidad excelente para compararlo con otros enfoques, en particular con la descomposición funcional.

Dado que este ejemplo captura a pequeña escala algunas propiedades importantes de la construcción de software orientado a objetos, además de mostrar concretamente la forma en que se puede pasar de la descomposición clásica a una visión orientada a objetos y los beneficios que se obtienen de esta transformación, puede considerársele como un instrumento de gran valor pedagógico.

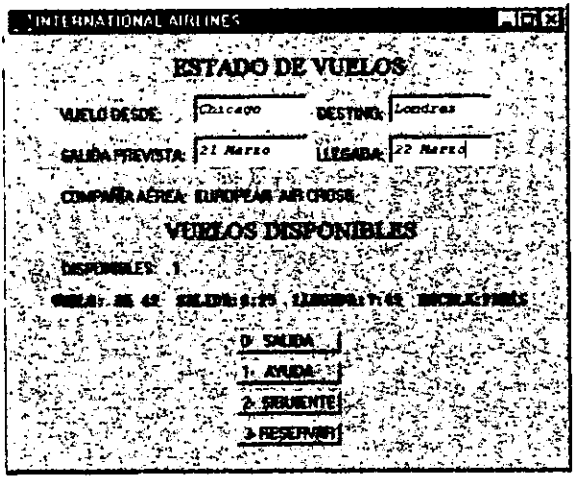
4.1.1 Sistemas Multi-panel

El problema es escribir un sistema que abarque a un tipo general de sistema interactivo, común en el procesamiento de datos comerciales, en el que los usuarios van siendo guiados en cada paso de una sesión por un panel de pantalla completa, con transiciones predefinidas entre los paneles disponibles.

El patrón general es sencillo y bien definido. Cada sesión pasa por un cierto número de *estados*. En cada estado se despliega un cierto panel que muestra las preguntas al usuario. El usuario debe llenar la respuesta solicitada; a esta respuesta se le aplicará una comprobación de consistencia (y se repetirán las preguntas hasta que se obtenga una respuesta aceptable); después, la respuesta se procesará de alguna forma; por ejemplo el sistema actualizará una base de datos. Una parte de la respuesta del usuario será la selección del próximo paso a realizar, que el sistema interpretará como una transición a otro estado, en el que de nuevo se volverá a aplicar el mismo proceso.

Se demostrará un ejemplo típico, el sistema de reservación para una línea aérea, en el que los estados podrían representar pasos de procesamiento tales como la identificación del usuario, consultas sobre los vuelos (para un cierto itinerario en una cierta fecha), consultas sobre plazas (para un cierto vuelo) y reservación.

Un panel típico, para consultar el estado de los vuelos, podría ser como el que se muestra a continuación (cuya única intención es ilustrar estas ideas y no pretende ser realista en cuanto a un diseño ergonómico). La pantalla se muestra en la fase final de un paso; los elementos que están en *cursiva* son las respuestas del usuario, los elementos que están en la línea "**Vuelo: AA...**" denotan la respuesta mostrada por el sistema.



La sesión comienza en un estado inicial y termina cada vez que se alcance un estado final. Se puede representar la estructura general mediante un grafo de transiciones que muestra los estados posibles y las transiciones entre ellos. Los arcos del grafo están etiquetados mediante números enteros que corresponden a las

posibles opciones que puede seleccionar el usuario al final del estado. La figura siguiente muestra un grafo de un sistema simple de reservación de una línea aérea.

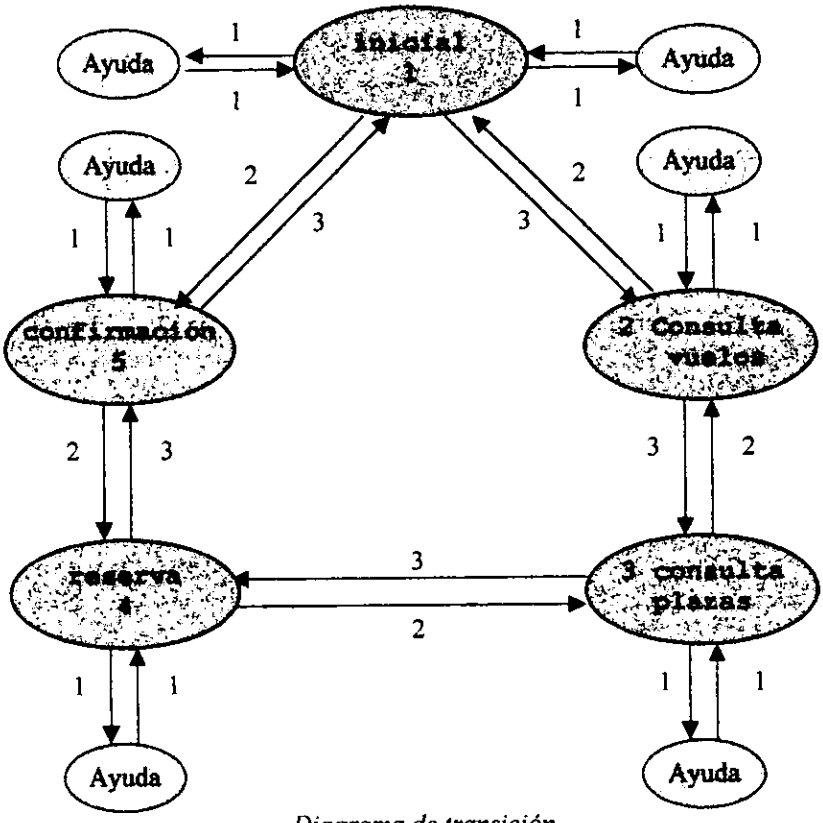


Diagrama de transición

El problema es abordar el diseño e implementación de tales aplicaciones alcanzando tanta generalidad y flexibilidad como sea posible. En particular:

1. El grafo puede ser grande. No es infrecuente ver aplicaciones con varios cientos de estados y sus transiciones correspondientes.
2. La estructura está sometida a cambios. Los diseñadores no pueden prever todos los estados y transiciones posibles. A medida que los usuarios empiecen a utilizar el sistema, surgirán peticiones de cambios y de adiciones.
3. Nada en el esquema dado es específico de la selección de aplicación hecha: el sistema de reservación de una línea aérea es solo un ejemplo de trabajo. Si una compañía necesita una cierta cantidad de sistemas como éste, bien sea para sus propios propósitos o (en una empresa de software) para diferentes usuarios, será muy beneficioso definir un diseño general o, mejor aún, un conjunto de módulos que se puedan reutilizar entre aplicaciones.

4.1.2 Un intento simple

Se comenzará por un esquema de programa elemental y nada sofisticado. Esta versión está formada por un cierto numero de bloques, uno para cada caso del sistema: $B_{consulta}$, $B_{reserva}$, $B_{cancelación}$, etc. Un bloque típico (expresado en una notación improvisada muy semejante a Pascal) tendría el aspecto:

```
Bconsulta:
      "Muestra el panel de consultas de vuelos"
      repeat
            "Leer la respuesta del usuario y elegir C como
            el siguiente paso"
            if "Hay error en la respuesta" then "Mostrar el
            mensaje de error apropiado"  end
      until not "error en la respuesta" end
"Procesar la respuesta"
      case C in
            C0: goto Salida,
            C1: goto BAyuda,
            C2: goto BReserva,
            . . .
end
```

y así para cada estado.

Hay algo que decir a favor de esta estructura: no es difícil de elaborar y funcionará bien. Pero desde el punto de vista de la ingeniería del software deja mucho que desear.

La crítica más evidente es la presencia de instrucciones `goto` (que implementan saltos incondicionales similares al `switch` de C) que le da a la estructura de control la apariencia de un "*Juego de Serpientes y Escaleras*".

Pero los `gotos` son el síntoma, no el verdadero error. Se ha tomado la estructura superficial del problema – la forma en sí del diagrama de transiciones – y se ha "*cableado*" en el algoritmo; la estructura de control del programa es un reflejo exacto de la estructura del diagrama de transición. Esto hace que el diseño de software sea vulnerable a cualquiera de los cambios comunes y simples que se han mencionado a lo largo de este trabajo, cada vez que alguien pida añadir un estado o cambiar una transición, habrá que cambiar la estructura central de control del sistema. Y se puede olvidar, por supuesto, toda esperanza de reutilización entre aplicaciones (el objetivo 3 de la lista anterior) ya que la estructura de control tendría que abarcar todas las aplicaciones.

4.1.3 Una solución funcional descendente

Repitiéndose en este ejemplo particular la evolución de las especies de programación en su totalidad, se pasará de una estructura de bajo nivel basada en `goto` a una solución descendente, organizada jerárquicamente, se analizarán sus propias limitaciones y sólo entonces se pasará a una versión orientada a objetos. La

solución jerárquica pertenece al estilo general conocido también con el nombre de "*estructurado*" aunque este término debe usarse con cuidado.

La función de transición

El primer paso para mejorar la solución es librarse del papel central que posee el algoritmo de recorrido dentro de la propia estructura del software. El diagrama de transición no es más que una de las propiedades del sistema y no hay razón para que se imponga por encima de todo lo demás. Al separarla del resto del algoritmo nos librará como mínimo, de las instrucciones `goto`. Por otra parte también deberíamos ganar en generalidad, ya que el diagrama de transición depende de la aplicación concreta, tal como las reservaciones de una línea aérea, mientras que su recorrido se puede describir de manera general.

¿Qué es un diagrama de transición? Desde un punto de vista abstracto, es una función *transición* que tiene dos argumentos, un estado y una opción escogida por el usuario, de tal modo que *transición(e, op)* es el estado que se obtiene cuando partiendo del estado *e* el usuario escoge *op*. Aquí la palabra "función" se usa en un sentido matemático; si hablamos de software se puede decir implementar *transicion* bien mediante una función en el sentido del software (una rutina que proporciona un valor) o mediante una estructura de datos por ejemplo como un array. Por el momento se pospondrá la decisión de seleccionar una de estas dos soluciones y se seguirá considerando *transicion* como un concepto abstracto.

Además de la función *transición* es necesario designar a uno de los estados, digamos el estado *inicial*, como el lugar en que comienzan todas las sesiones y habrá que designar a uno o más estados como finales a través de una función booleana *es_final*. Una vez más, se trata de una función en el sentido matemático, independientemente de su eventual implementación.

Se puede representar la función *transición* en forma de tabla(*), en que las filas representan los estados y las columnas representan las opciones, según se muestra a continuación:

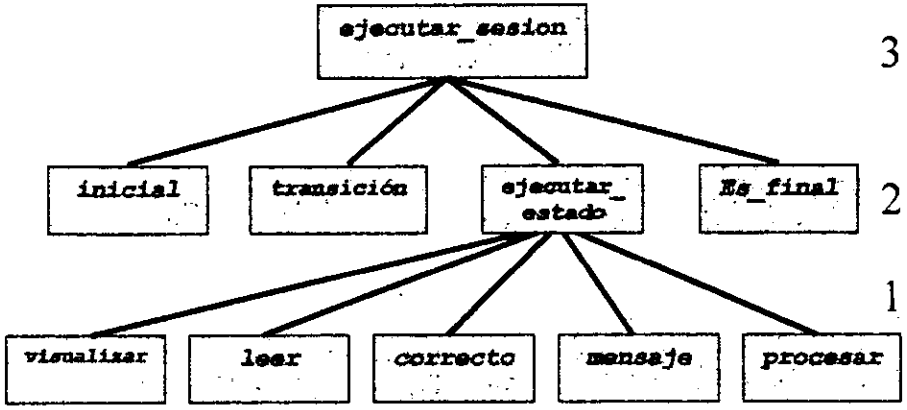
Opción / Estado	0	1	2	3
1 *(inicial)*	-1	0	5	2
2 *(vuelos)*		0	1	3
3 *(plazas)*		0	2	4
4 *(reserv.)*		0	3	5
5 *(Confir.)*		0	4	1
0 *(Ayuda)*		volver		
-1 *(Final)*				

(*) Los convenios que se usan en esta tabla son como sigue: hay un estado de ayuda, el 0, con una transición especial `volver` que vuelve al estado en que se estaba cuando se llegó a `Ayuda`, y un solo estado final, -1. Estos convenios no serán necesarios para el resto del ejemplo pero sirven de ayuda para mantener la sencillez de la tabla.

La arquitectura de la rutina

Siguiendo los preceptos tradicionales de la descomposición descendente (desde arriba hacia abajo), se escoge una "cima" (el programa principal) para el sistema. Debería ser claramente la rutina *ejecutar_sesion* que describe la forma de ejecutar una sesión interactiva completa.

Inmediatamente debajo de éste (nivel 2) se encuentran las operaciones relativas a estados: la definición de los estados iniciales y finales, la estructura de transición, y *ejecutar_estado* que preescribe las acciones que habrá que ejecutar en cada estado. Y después, en el nivel más bajo (1) se encuentran las operaciones constitutivas de *ejecutar_estado*: visualizar una ventana y cosas por el estilo.



Descomposición funcional descendente

Obsérvese que se puede decir que una solución como ésta, al igual que cualquier cosa orientada a objetos que se verá posteriormente, "refleja el mundo real": la estructura del software refleja perfectamente la estructura de una aplicación, que contiene estados, que a su vez contiene operaciones elementales. Los detalles del mundo real no constituyen, en este ejemplo y en muchos otros, una diferencia significativa entre O-O y otros enfoques; lo que cuenta es como se modela el mundo.

Al escribir *ejecutar_sesion* se tratará que sea lo más independiente posible de la aplicación.

```
ejecutar_sesion  is
     --Ejecuta una sesión completa del sistema interactivo
local
     estado, siguiente: INTEGER
do
     estado:=  inicial
     repeat
          ejecutar_estado (estado, → siguiente)
          --la rutina ejecutar_estado actualiza el valor
            de siguiente.
```

```
            estado:=  transicion (estado, siguiente)
        until es_final (estado) end
end
```

Éste es un algoritmo típico de recorrido de un diagrama de transición. En cada etapa estamos en un estado *estado*, fijado originalmente en *inicial*; el proceso termina cuando *estado* satisface *es_final*. Para todo estado que no sea final se ejecuta *ejecutar_estado*, que toma el estado actual y proporciona la opción de transición del usuario a través de su segundo argumento *siguiente*, que será utilizado por la función de transición junto con *estado*, para determinar el próximo estado.

Esta técnica de utilizar el procedimiento *ejecutar_estado* que cambia el valor de uno de sus argumentos nunca sería apropiada en un buen diseño O-O, pero aquí es la más conveniente. Para señalizarla claramente, la notación utilizada marca los argumentos de "salida" como *siguiente* con una flecha →. En lugar de un procedimiento que modifica a un argumento, quienes programan en C harían que *ejecutar_estado* fuera una función con efecto lateral que se invocaría en la forma *siguiente:=ejecutar_estado(estado)*; se verá que esta práctica también es criticable.

Dado que *ejecutar_estado* no muestra ninguna información sobre ninguna aplicación interactiva en particular, hay que rellenar las propiedades específicas de la aplicación que aparecen en el nivel 2 de la figura: la función *transicion*, el estado *inicial* y el predicado *es_final*.

Para completar el diseño, se debe refinar la rutina *ejecutar_estado* que describe las acciones que habrá de realizar en cada estado. Su cuerpo es esencialmente una forma abstracta del contenido de los bloques sucesivos de la versión inicial basada en **goto**:

```
ejecutar_estado  (in e: INTEGER; out op: INTEGER)  is
      --Ejecuta las acciones asociadas al estado e,
      --devolviendo en op la opción escogida por el usuario
      --para el siguiente estado.
local
      r: RESPUESTA;  ok: BOOLEAN
do
      repeat
            visualizar (e)
            leer (e, →r)
            ok:= correcto (e, r)
            if not ok then mensaje (e, r) end
      until ok end
      procesar (e, r)
      op:= siguiente_opcion ( r)
end
```

Esto supone que las rutinas del nivel 1 tienen las siguientes funcionalidades:

- *visualizar (e)* muestra el panel asociado con el estado *e*.
- *Leer (e, →r)* lee en *r* la respuesta del usuario a la visualización del panel que se ha mostrado del estado *e*.
- *Correcto (e, r)* proporciona verdadero si y sólo si *r* es una respuesta aceptable a la pregunta que se ha mostrado en el estado *e*; si lo es, *procesar (e, r)* procesará la respuesta *r*, por ejemplo actualizando una base de datos o mostrando más información; si no lo es, *mensaje (e,r)* mostrará el mensaje de error correspondiente.

El tipo *RESPUESTA* del objeto que representa la respuesta del usuario no se ha refinado más. Un valor *r* de dicho tipo representa globalmente la respuesta introducida por el usuario en un estado dado. Se supone que esto incluye la selección del usuario para el siguiente paso, que se escribe *siguiente_opcion(r)*. (*RESPUESTA* ya es, de hecho, muy parecida a una clase, aunque el resto de la arquitectura no sea orientada a objetos en modo alguno).

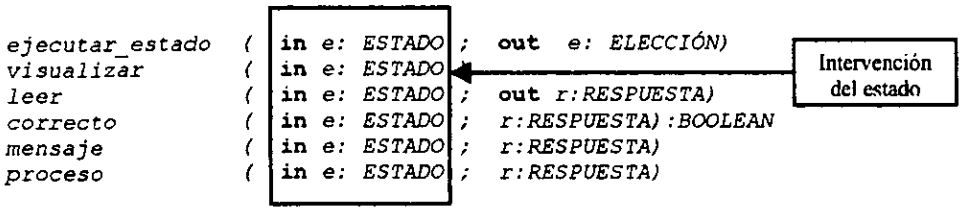
Para obtener una aplicación viable, es necesario completar varias características de nivel 1: *visualizar*, *leer*, *correcto*, *mensaje* y *procesar*.

4.1.4 Una crítica de la solución

¿Se tiene ya una solución satisfactoria? No, aún no. Es mejor que la primera versión, pero se queda corta a efectos de extensibilidad y reutilización.

Estadismo

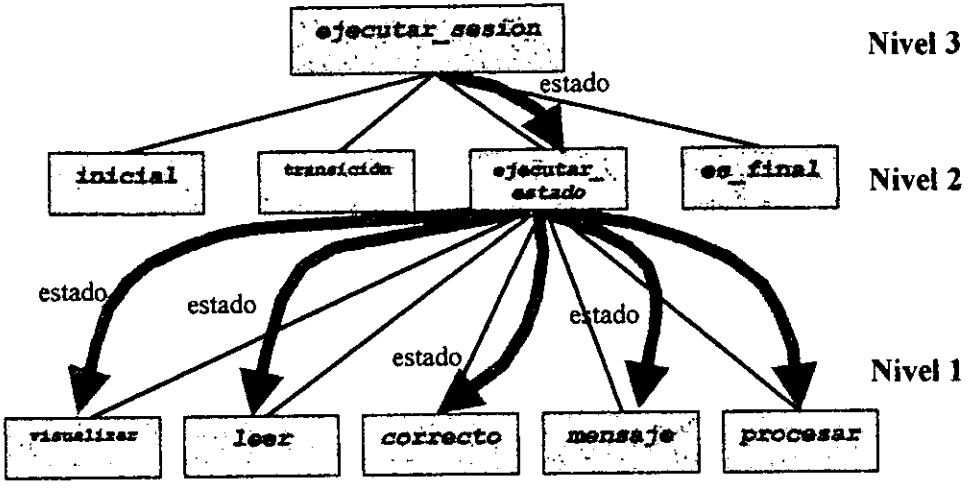
Aunque a nivel superficial parece que se ha logrado separar lo genérico de lo específico de la aplicación, en realidad los diferentes módulos siguen estando fuertemente acoplados entre sí y con la aplicación seleccionada. El problema es la estructura de transición de datos del sistema. Considérese las signaturas (tipos de argumentos y del resultado) de las rutinas:



La observación es que el papel del estado es demasiado omnipresente. El estado en curso aparece como argumento *e* en todas las rutinas, empezando por el módulo cima *ejecutar_sesion*, donde se le conoce como *estado*. De modo que la estructura jerárquica que se muestra en la última figura, aparentemente simple y manejable, es una mentira, o más exactamente una fachada. Tras la elegancia formal de la descomposición funcional se esconde un revoltijo de transmisión de datos. El verdadero cuadro, que se muestra en la siguiente figura, debe involucrar a los datos.

El trasfondo de la tecnología de objetos, es la batalla entre los aspectos *función* y *datos* (objetos) de los sistemas software para controlar la arquitectura. En los enfoques tradicionales el papel de las funciones se impone sin oposición sobre los datos.

La venganza viene en forma de sabotaje. Atacando las propias bases de la arquitectura: los datos hacen al sistema insensible al cambio.



El flujo de datos

En este ejemplo la subversión de la estructura proviene en particular de la necesidad de discriminación basada en los estados. Todas las rutinas del nivel 1 deben realizar acciones diferentes dependiendo del estado *e*: mostrar el panel para un cierto estado; leer e interpretar una respuesta del usuario (que constará de un cierto número de campos de entrada, diferentes para cada estado); determinar si la respuesta es o no correcta; mostrar el mensaje de error adecuado; procesar una respuesta correcta – hay que conocer el estado. Las rutinas efectuarán una discriminación de la forma

```
inspect
      e
      when inicial then
            ...
      when consulta_sobre_vuelos then
      ...
...
end
```

Esto significa largas y complejas estructuras de control y, lo que es peor aún, un sistema frágil: añadir un estado requerirá cambios que afectarán a toda la estructura. Se trata de un caso típico de distribución desenfrenada del conocimiento: hay

demasiados módulos del sistema basados en un mismo elemento de información – la lista de todos los posibles estados – que está sometido a cambios.

La situación es de hecho peor de lo que parece si se aspira a tener soluciones generales reutilizables. Hay un argumento adicional implícito en todas las rutinas consideradas hasta ahora: la *aplicación* – una reservación de líneas aéreas o cualquier otra cosa que se esté construyendo. De modo que para hacer que rutinas tales como `visualizar` fueran verdaderamente generales habría que hacerles conocer todos los estados de todas las aplicaciones posibles en un entorno determinado de computación. De igual modo, la función `transicion` contendría el grafo de transiciones para todas las aplicaciones. Esto, por supuesto, no es realista.

4.1.5 Una arquitectura Orientada a Objetos

Las propias deficiencias de la descomposición funcional descendente señalan lo que se debe hacer para obtener una buena versión orientada a objetos.

Demasiada transmisión de datos en una arquitectura de software indica por lo general una deficiencia en el diseño. El remedio, que nos lleva directamente al diseño orientado a objetos, puede expresarse en la siguiente regla de diseño:

LEY DE INVERSIÓN

Si las rutinas intercambian demasiados datos, poner las rutinas en los datos.

En lugar de construir los módulos alrededor de las operaciones (tal como `ejecutar_sesion` y `ejecutar_estado`) y distribuir las estructuras de datos entre las rutinas resultantes, con todas las consecuencias desagradables que se han visto, el diseño orientado a objetos hace lo contrario: usa los tipos de datos más importantes como bases de la modularización y asocia cada rutina al tipo de dato con el que esté más estrechamente relacionada. *Cuando los objetos se imponen, sus viejos amos, las funciones, pasan a ser vasallos.*

La ley de inversión es la clave para obtener un diseño orientado a objetos a partir de una descomposición funcional (de procesos) clásica. Esa necesidad surge en los casos en que se quiere hacer *ingeniería inversa* de un sistema existente que no sea O-O para poder darle un mejor mantenimiento y prepararlo para su evolución; esto también es frecuente en equipos de trabajo poco experimentados en el diseño orientado a objetos que tienen que pensar primero de manera "funcional".

Por supuesto que lo mejor es diseñar de modo orientado a objetos desde el principio; entonces no es necesario hacer esta inversión. Pero la ley es útil más allá de los casos de ingeniería inversa y de desarrolladores poco experimentados. Incluso alguien que haya sido expuesto a los principios de la construcción de software orientado a objetos puede llegar a un diseño inicial que tenga "*nubes*" de descomposición funcional en un "*paisaje*" de objetos. Analizar la transmisión de datos es una buena forma de detectar y corregir los defectos de diseño.

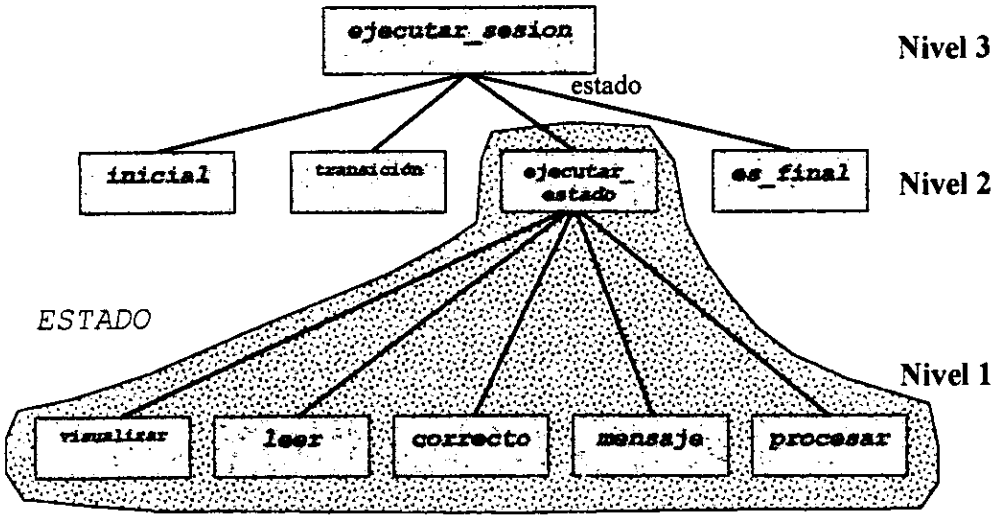
El estado visto como clase

El ejemplo "estado" es típico. Este tipo de datos, omnipresente en la transmisión de datos entre rutinas, es un perfecto candidato para servir como uno de los componentes modulares de una arquitectura orientada a objetos, que debe basarse en clases (tipos de datos descritos de forma abstracta).

La noción de estado era importante en el planteamiento original del problema, pero en la arquitectura funcional esta importancia se ha perdido: el estado pasó a estar representado por una variable, que se pasaba de rutina a rutina. Ahora es necesario darle la categoría que merece. El *ESTADO* debería ser una clase, y una de las más importantes en la estructura de nuestro nuevo sistema orientado a objetos.

En dicha clase se encontrarán todas las operaciones que caracterizan a un estado: mostrar la pantalla correspondiente (*visualizar*), analizar la respuesta del usuario (*leer*), verificar la respuesta (*correcto*), dar un mensaje de error a una respuesta incorrecta (*mensaje*) y procesar una respuesta correcta (*procesar*). También se debe incluir *ejecutar_estado* que expresa la secuencia de acciones que hay que llevar a cabo si la sesión alcanza un estado dado; puesto que este nombre puede ser redundante dentro de una clase que a su vez se llama *ESTADO* puede simplemente reemplazarse por *ejecutar*.

Partiendo del diagrama de la descomposición funcional descendente se puede destacar el conjunto de rutinas que deberían pasársele a *ESTADO*:



Características de ESTADO

La clase tendrá la forma siguiente:

```
class ESTADO feature
      entrada: RESPUESTA
      opcion: INTEGER
      ejecutar  is do ... end
      visualizar   is ...
      leer   is ...
      correcto:  BOOLEAN is ...
      mensaje  is ...
      procesar  is ...
end
```

Las características *entrada* y *opcion* son atributos; las demás son rutinas. Comparadas con sus contrapartidas de la descomposición funcional, las rutinas han perdido su argumento explícito de estado, aun cuando el estado reaparecerá en las llamadas hechas por los clientes, como por ejemplo, *e.ejecutar*.

En el enfoque anterior, *ejecutar* (llamado anteriormente *ejecutar_estado*) proporcionaba la opción escogida por el usuario para el siguiente paso. Pero ese estilo viola los principios de un buen diseño. Es preferible tratar a *ejecutar* como una orden, cuya ejecución determina el resultado de la consulta "¿qué selección realizó el usuario en el último estado?" y que está disponible a través del atributo *opcion*. Análogamente, el argumento *RESPUESTA* de las rutinas del nivel 1 es sustituido ahora por el atributo secreto *entrada*. La razón es la ocultación de información: el código del cliente no necesita ver las respuestas excepto a través de la interfaz que proporcionen las características exportadas.

Herencia y clases diferidas

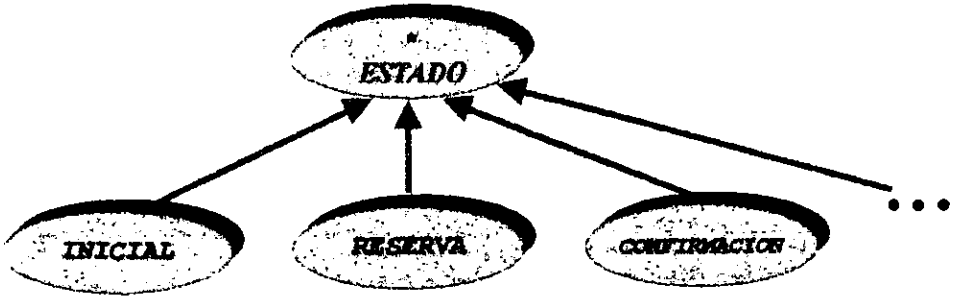
La clase *ESTADO* no describe ningún estado particular, sino la noción general de estado. El procedimiento *ejecutar* es el mismo para todos los estados, pero las otras rutinas son específicas de cada estado.

La herencia y las clases diferidas son técnicas efectivas para abordar tales situaciones. En el nivel de *ESTADO* se conoce el procedimiento *ejecutar* con todo detalle, y los atributos. También se conoce la existencia de rutinas de nivel 1 (*visualizar*, etc.) pero no sus implementaciones. Estas rutinas deberían ser diferidas; la clase *ESTADO*, que describe un conjunto de variantes, más que una abstracción totalmente definida, es a su vez una clase diferida. Esto da lugar a:

```
indexing
      description: "Estado de una aplicación controlada por
                    paneles interactivos"
deferred class
      ESTADO
feature      --Acceso
      opcion: INTEGER
            --La opción escogida por el usuario para el próximo paso
      entrada: RESPUESTA
            --Respuesta del usuario para las preguntas hechas en este
```

```
            --estado
feature     --Informe del estado
      correcta:  BOOLEAN is
            -- ¿Es entrada una respuesta correcta?
            deferred
            end
feature      --operaciones básicas
      visualizar is
            --Despliega el panel asociado con el estado en curso
            deferred
            end
      ejecutar is
            --Ejecuta acciones asociadas con el estado en curso
            --y actualiza opcion para denotar la elección del
            --usuario para el siguiente paso.
            local
                  ok:  BOOLEAN
            do
                  from ok:=false until ok loop
                        visualizar; leer; ok:=correcto
                        if not ok then mensaje end
                  end
                        procesar
                  ensure
                        ok
                  end
            mensaje is
                  --Da el mensaje de error correspondiente a entrada
                  require
                        not correcto
                  deferred
                  end
            leer is
                  --Obtiene la respuesta del usuario a través de
                  --entrada y opción
                  deferred
                  end
            procesar is
                  --procesa entrada
                  require
                        correcto
                  deferred
                  end
end   --clase ESTADO
```

Para describir un estado específico se introducirán descendientes de *ESTADO* que proporcionen las implementaciones efectivas de las rutinas diferidas:



Jerarquía de clases de estado

Un ejemplo tendría el siguiente aspecto:

```
class CONSULTA_VUELOS inherit
      ESTADO
feature
      visualizar is
          do
            ... Procedimiento especifico para mostrar consultas
                sobre vuelos ...
          end
      ... De manera similar para leer, correcto, mensaje y procesar
end - class CONSULTA_VUELOS
```

Esta arquitectura separa, con el nivel exacto de granularidad, los elementos comunes a todos los estados y los elementos específicos de estados individuales. Los elementos comunes, tales como el procedimiento `ejecutar` se concentran en *ESTADO* y no necesitan ser redeclarados en descendientes tales como *CONSULTA_SOBRE_VUELOS*. Se satisface el principio abierto-cerrado: *ESTADO* es cerrado en tanto que es una unidad compilable y bien definida; pero también es abierto, puesto que se puede añadir cualquier número de descendientes en cualquier momento.

ESTADO es un caso típico de **clase de comportamiento** – una clase diferida que captura el comportamiento común de un gran número de posibles objetos, y que implementa lo que es totalmente conocido al nivel más general (`ejecutar`) en términos de lo que depende de cada variante. La herencia y el mecanismo de clases diferidas son esenciales para capturar tal comportamiento en un componente reutilizable autocontenido.

Descripción de un sistema completo

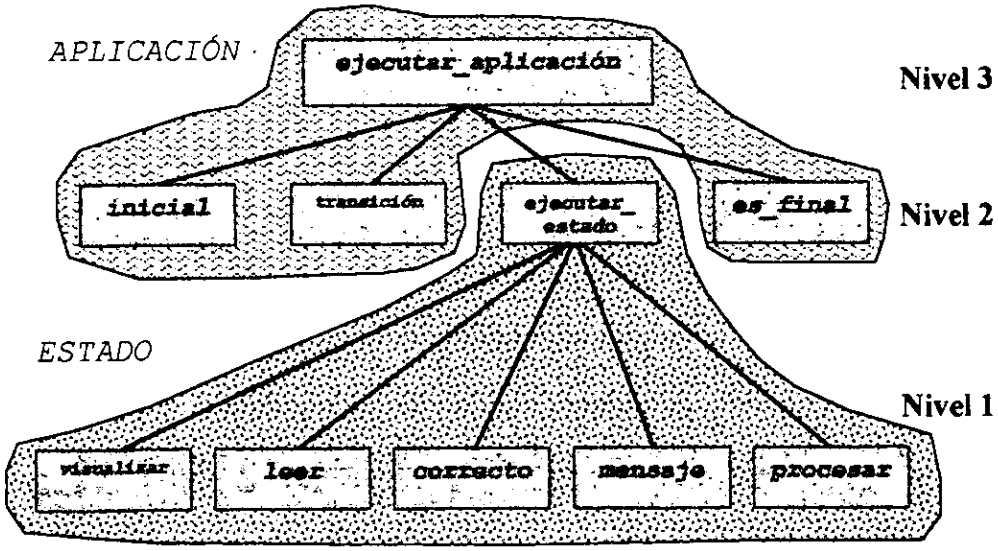
Para completar el diseño sigue siendo necesario encargarse de gestionar una sesión. En la descomposición funcional, ésta era la tarea del procedimiento `ejecutar_sesion`, el programa principal. Pero un sistema de software lleva a cabo muchas funciones las cuales no pueden jerarquizarse de forma tan simple, cada función es igualmente importante. Para esta situación, el enfoque de tipos abstractos

de dato es el más adecuado; considera al sistema como un todo, como un conjunto de objetos abstractos capaces de ofrecer un cierto número de servicios.

Ya se ha captado una abstracción clave: *ESTADO* (junto con *RESPUESTA*). Pero aún falta una abstracción en este diseño. Para la comprensión del problema es básico el concepto de *APLICACIÓN*, que describe sistemas interactivos específicos tal como el de reservaciones de una línea aérea. Esto producirá una nueva clase.

Los componentes restantes de la descomposición funcional, que se muestran en la figura, son todos ellos características de una aplicación y hallarán su verdadera vocación como características de la clase *APLICACIÓN*.

- `ejecutar_aplicacion`, que describe cómo ejecutar una aplicación. Aquí el nombre se reducirá al de `ejecutar` puesto que la clase que incluye la característica ya da información suficiente (y no hay posibilidad de confusión con el `ejecutar` de *ESTADO*).
- `inicial` y `es_final`, que indican los estados que tienen una categoría especial en una aplicación. Observe que es más apropiado tener estas características en *APLICACION* que en *ESTADO* puesto que describen propiedades de las aplicaciones en lugar de propiedades de los estados; un estado no es inicial ni final por sí mismo, sino sólo con respecto a una aplicación. (Si se reutilizan estados entre aplicaciones diferentes, un estado puede muy bien ser final en una cierta aplicación pero no en otra.)
- `transicion` para describir la transición entre los estados de la aplicación.



Características de ESTADO y APLICACIÓN

Los componentes de la descomposición funcional han encontrado todos un lugar como características de las clases de la descomposición O-O – algunos en *ESTADO*, otros en *APLICACIÓN*. Esto es natural; la tecnología de objetos es sobre todo un mecanismo de arquitectura, que afecta principalmente a la forma en que se organizan los elementos de software en estructuras coherentes. Los elementos en si mismos pueden ser, en el nivel más bajo, los mismos que se pueden encontrar en una solución no O-O, o al menos similares (la abstracción de datos, la ocultación de información, las aserciones, la herencia, el polimorfismo y la ligadura dinámica ayudan a hacerlos más simples, generales y potentes).

Un sistema controlado por paneles, del tipo del estudiado en esta sección, siempre necesitará operaciones para recorrer el grafo de la aplicación (antes `ejecutar_sesion`, ahora `ejecutar`), leer entradas del usuario (`leer`), detectar estados finales (`es_final`). En lo más profundo de la estructura, se encontrarán algunos de los mismos bloques de construcción, independientemente del método. Lo que cambia es la forma en que se agrupan para producir una arquitectura modular.

Por supuesto que no es necesario limitarse a las características que provengan de la solución anterior. Lo que para la descomposición funcional fuera el final del proceso – la construcción de `ejecutar` para las aplicaciones y todos los demás mecanismos que necesita – es ahora sólo el comienzo. Hay muchas cosas más que se pueden querer hacer en una aplicación:

- Añadir un nuevo estado.
- Añadir una nueva transición.
- Construir una aplicación (por aplicación repetida de las dos operaciones precedentes).
- Eliminar un estado o una transición.
- Almacenar la aplicación completa, sus estados y transiciones, en una base de datos.
- Simular la aplicación (por ejemplo en un monitor sin capacidades gráficas, o con segmentos vacíos en lugar de las rutinas de la clase *ESTADO*, para verificar solamente las transiciones).
- Monitorizar el uso de la aplicación.

Todas estas operaciones, y otras, producirán características de la clase *APLICACIÓN*. No son ni más ni menos importantes que el "programa principal" anterior, el procedimiento `ejecutar` que ahora es simplemente una característica de la clase, que no gobierna sobre las demás. Al renunciar a la noción de "cima del sistema" se deja espacio para la evolución y la reutilización.

La clase aplicación

Para finalizar la clase *APLICACIÓN* se dan a continuación algunas posibles decisiones de implementación:

- Numerar los estados 1 a *n* para la aplicación. Obsérvese que estos números no son propiedades absolutas de los estados, sino que son relativos a cierta aplicación; por lo que no hay un atributo "número de estado" en la clase *ESTADO*.

En lugar de esto lo que habrá es un array unidimensional *estado_asociado* que es un atributo de *APLICACIÓN* y que proporcionará el estado asociado a cada número.

- Representar la función transición mediante otro atributo, un array bidimensional de tamaño n x m, donde m es el número posible de opciones de salida para cada estado.
- El número del estado inicial se almacena en el atributo *inicial* y se fija mediante la rutina *escoger_inicial*. Para los estados finales se puede utilizar la convención de que una transición a un pseudoestado 0 denota el fin de la sesión.
- El procedimiento de creación de *APLICACIÓN* usa los procedimientos de creación de las clases de biblioteca *ARRAY* y *ARRAY2*. Esta última describe arrays de dos dimensiones y sigue los mismos pasos que *ARRAY*, su procedimiento de creación *make* admite cuatro argumentos, como en el caso de *!!a.make (1, 25, 1, 10)* y sus rutinas *item* y *put* emplean dos índices, como en el caso de *a.put (x, 1, 2)*. Los límites de un array bidimensional *a* se denotan *a.lower1* etc.

Véase a continuación la clase resultante de estas decisiones:

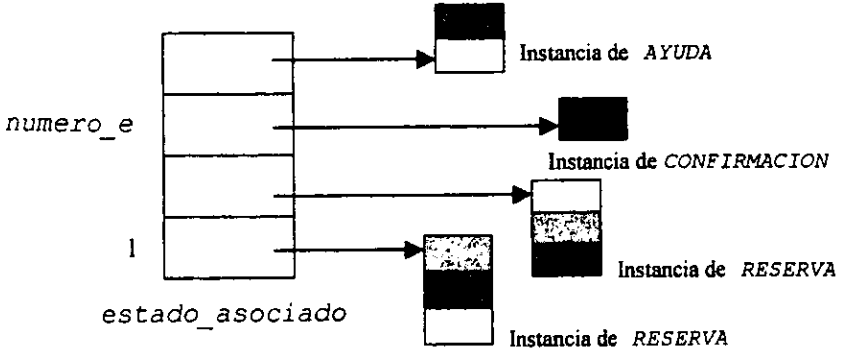
```
indexing
      description: "Aplicaciones interactivas dirigidas por paneles"
class APLICACION creation
      make
feature --Inicialización
      make (n, m: INTEGER) is
            --Reserva espacio para una aplicación con n estados
            --y m opciones posibles.
           do
            !! transicion.make (1, n, 1, m)
            !! estados_asociados.make (1, n)
           end
feature  --Acceso
      inicial: INTEGER
            --Número del estado inicial
feature  --Operaciones básicas
      ejecutar  is
            --Ejecuta una sesión con el usuario
           local
              e: ESTADO; numero_e: INTEGER
           do
            from
                numero_e:= inicial
            invariant
                0 <= numero_e; numero_e <= n
            until  numero_e= 0 loop
                e:= estado_asociado.item (numero_e)

             e.ejecutar

                --Esto se refiere por supuesto al procedimiento
                --ejecutar de ESTADO
                --(véanse más adelante comentarios sobre esta
                --instrucción clave)
```

```
                numero_e:= transicion.item (numero_e, e.opcion)
            end
        end
feature --Cambiar elemento
    poner_estado (e: ESTADO; numero_e: INTEGER) is
        --Pone el estado e en la posición numero_e.
    require
        1<=numero_e; numero_e <= estado_asociado.upper
    do
        estado_asociado.put (e, numero_e)
    end
escoger_inicial (numero_e: INTEGER) is
        --Define que el estado de número numero_e es el
        --estado inicial.
    require
        1<=numero_e; numero_e <= estado_asociado.upper
    do
        inicial:= numero_e
    end
poner_transicion (origen, destino, etiqueta: INTEGER) is
        --Introduce la transición etiquetada etiqueta que va
        --del estado número origen al estado número destino.
    require
        1<=origen;   origen<=estado_asociado.upper
        0<=destino;  destino<=estado_asociado.upper
        1<=etiqueta; etiqueta<=transicion.upper
    do
        transicion.put (origen, etiqueta, destino)
    end
feature { NONE } -- Implementación
    transicion: ARRAY2 [INTEGER]
    estado_asociado: ARRAY [ESTADO]
    ... Otras características ...
invariant
    transicion.upper1 = estado_asociado.upper
end  --class APLICACIÓN
```

Obsérvese la manera simple y elegante en que la llamada `e.ejecutar`, que ha sido destacada en el listado anterior, captura la semántica esencial del problema. La característica llamada es `ejecutar` de *ESTADO*; aun cuando es efectiva, debido a que describe un comportamiento conocido general, `ejecutar` se basa en las características diferidas `leer`, `mensaje`, `correcto`, `visualizar`, `procesar`, que están al mismo nivel que *ESTADO* y se hacen efectivas sólo en los descendientes propios tales como *RESERVA*. Cuando se pone la llamada `e.ejecutar` dentro del propio `ejecutar` de *APLICACIÓN* no hay por qué saber qué tipo de estado denota `e` – aunque sabemos que es un estado (éste es el beneficio de la comprobación estática de tipos). Para entrar en acción, esta instrucción necesita de la maquinaria de la ligadura dinámica: cuando en ejecución `e` se asocie a algún objeto de un tipo particular de estado, digamos *RESERVA*, entonces las llamadas a `leer`, `mensaje` y demás consortes dispararán automáticamente la versión apropiada.

El valor de `e` se obtiene de `estado_asociado` que es una **estructura de datos polimorfa** que puede contener objetos de tipos diferentes, todos ellos

compatibles con *ESTADO*. Lo que se encuentre en el índice actual *numero_e* determinará las siguientes operaciones de estado.



Un array polimorfo de estados

Ésta es la forma en que se construye una aplicación interactiva. La aplicación estará representada mediante una entidad, digamos *reserva_aerea* que se declara de tipo *APLICACIÓN*. Hay que crear el objeto correspondiente, por ejemplo, en el procedimiento de creación de la clase que haga de raíz:

```
!! reserva_aerea:make (total_de_estados, total_de_opciones_posibles)
```

Por otra parte habrá que definir y crear los estados de la aplicación como entidades de tipos descendientes de *ESTADO*, que pueden ser o bien nuevos o reutilizados de una biblioteca de estados. A cada estado *e* se le asigna un número *i* para la aplicación:

```
reserva_aerea.poner_estado (e, i)
```

Se escoge uno de los estados, digamos el estado número i_0 como el estado inicial:

```
reserva_aerea.escoger_inicial (i0)
```

Para configurar una transición cuya etiqueta es *etiq* y que va del estado de número *num_orig* al estado número *num_dest* se usa

```
reserva_aerea.poner_transicion (num_orig, num_dest, etiq)
```

Esto incluye las transiciones de salida (*exit*), para las cuales *num_dest* es 0 (el valor por defecto). Ahora se puede ejecutar la aplicación:

```
reserva_aerea.ejecutar_sesion
```

Durante la evolución del sistema se pueden utilizar en cualquier momento las mismas rutinas para añadir un nuevo estado o una nueva transición.

Por supuesto es posible extender la clase *APLICACIÓN*, bien sea modificándola o añadiendo descendientes, para admitir más características tales como eliminación, simulación, o cualquiera de las demás características mencionadas en el curso de esta sección.

4.1.6. Consideraciones sobre el problema

Este ejemplo ofrece una imagen nítida de las diferencias entre la construcción de software orientado a objetos y los enfoques anteriores. En particular, muestra los beneficios de librarse del concepto de programa principal. Al centrar la atención en las abstracciones de datos y olvidarse, durante el mayor tiempo posible, de cuál es "la función principal del sistema", se obtiene una estructura que con mayor probabilidad se presta por sí misma de un modo elegante a cambios futuros y que se podrá reutilizar para muchas variantes diferentes.

Este efecto igualador es una de las propiedades características del método. Requiere cierta disciplina para aplicarlo consistentemente, puesto que significa resistirse a la tentación constante de preguntar: "¿Qué es lo que hace el sistema?" Ésta es una de las cualidades que distingue al verdadero profesional orientado a objetos de aquellos que (aunque estén usando técnicas O-O y lleven ya un tiempo utilizando un lenguaje O-O) aún no han asimilado el método y siguen produciendo arquitecturas funcionales detrás de una fachada de objetos.

También se ha visto en este ejemplo una heurística que suele ser útil para identificar las abstracciones clave en una aplicación orientada a objetos (para "encontrar las clases"): analizar las transmisiones de datos y vigilar los conceptos que aparecen en las comunicaciones entre numerosos componentes de un sistema. Esto suele ser un indicador de que la estructura tiene que ponerse "de cabeza", haciendo que sean las rutinas las que estén asociadas a las estructuras de datos y no al revés.

Una última lección que se rescata del desarrollo de este ejemplo es que hay que rehuir de dar mucha importancia a la idea de que los sistemas orientados a objetos se deducen directamente del "mundo real". La potencia de modelado del método es realmente impresionante y es agradable producir arquitecturas de software cuyos componentes principales reflejen directamente las abstracciones del sistema externo que se esté modelando. Pero hay muchas maneras de modelar el mundo real y no todas nos llevarán a un buen sistema. La primera versión, la de los `goto`, estaba tan cerca del mundo real como las otras dos – realmente más cerca si se quiere, puesto que refleja directamente la estructura de los diagramas de transición, mientras que las otras dos requieren de la introducción de conceptos intermedios. Pero es un desastre de ingeniería del software.

En contraste, la descomposición orientada a objetos que se ha producido finalmente es buena porque las abstracciones que emplea – *ESTADO*, *APLICACIÓN*, *RESPUESTA* – son claras, manejables, están preparadas para el cambio y son reutilizables para una amplia gama de aplicaciones. Aunque una vez que se entienden pueden parecer tan reales como cualquier otra, para un novato pueden parecer menos "naturales" (es decir, menos próximas a la percepción informal de la realidad subyacente) que los conceptos utilizados en las soluciones menos buenas que se estudiaron en primer lugar.

Para producir buen software lo que cuenta no es lo cerca que se esté de la percepción del mundo real que tenga una persona, sino lo buenas que son las

abstracciones que se escogen para modelar los sistemas externos y para estructurar el software. Ésta es realmente la verdadera definición del análisis, diseño e implementación orientados a objetos, la tarea que habrá de ejecutarse bien, día a día, para hacer que los proyectos tengan éxito, y también es la destreza que distingue a los expertos en objetos de los aficionados a los objetos: **encontrar las abstracciones correctas**.

4.2. Herencia, un caso práctico: "deshacer" en un sistema interectivo

Como segundo ejemplo de diseño, se abordará una necesidad con la que se enfrentan de manera frecuente los diseñadores de casi todos los sistemas interactivos: la manera de proporcionar órdenes de deshacer una determinada acción.

Se expondrá la forma en que la herencia y la ligadura dinámica producen una solución sencilla, regular y general para un problema aparentemente intrincado y de muchas facetas. También enseñarán unas cuantas lecciones generales sobre los problemas y principios del diseño orientado a objetos.

4.2.1. El contexto del problema

Se dice que errar es humano, y cuanto más rápidos y potentes se vuelven nuestros sistemas interactivos, más fácil es hacer que realicen acciones que no eran exactamente lo que deseábamos. Ésta es la razón por la cual se desea una forma de borrar el pasado reciente; un "botón" o mecanismo que se pueda activar para hacer que desaparezca la última acción realizada, la cual se llevó a cabo por error, juego o mala intención.

La naturaleza de "Deshacer"

En un sistema interactivo el equivalente al "botón" o mecanismo antes mencionado, es una operación de *Deshacer*, que proporciona el diseñador del sistema para beneficio de cualquier usuario que, en alguna etapa de la sesión de trabajo, desee cancelar el efecto de la última orden ejecutada.

El primer objetivo de un mecanismo de deshacer es permitir a los usuarios recuperarse de daños debidos a posibles errores de entrada. Es muy fácil teclear el carácter equivocado o hacer clic en "OK" en lugar de en "Cancelar". Pero una capacidad para deshacer debe ir más allá. Así se libera a los usuarios de tener que concentrarse nerviosamente en todas y cada una de las teclas pulsadas, y en los botones en los que se hace clic. Más allá de esto, se estimula un estilo de interacción *"¿Qué pasa si...?"* En el que los usuarios prueban distintos tipos de entradas, sabiendo que pueden volver atrás con facilidad si el resultado no es el que esperaban.

Todo buen sistema interactivo debe brindar un mecanismo como éste. Cuando está presente, tiende a ser una de las operaciones utilizadas con más frecuencia.

Deshacer y rehacer de múltiples niveles

Ofrecer un mecanismo para deshacer es mejor que no ofrecer ninguno, pero no es suficiente. La mayoría de los sistemas que proporcionan la facilidad *Deshacer* se autolimitan a un nivel: sólo se puede cancelar el efecto de la última orden. Si nunca se cometen dos errores seguidos, esto es suficiente. Pero si se va en la dirección equivocada y se desea retroceder varios pasos, entonces hay un problema.

En realidad, no hay ninguna excusa para restringir a un nivel la posibilidad de deshacer. Una vez que se ha establecido el mecanismo para deshacer, el paso de un nivel a múltiples niveles es un asunto sencillo. Y en pro de adquirir buenos hábitos como desarrollador, no hay que limitar el número de órdenes que se pueden deshacer a un valor ridículamente pequeño; si después de todo es preciso limitarlo, hay que permitir que sea el usuario quién escoja su propio límite (a través de "preferencias" que podrá actualizar para todas las sesiones futuras) y tome un valor por defecto que sea al menos 20. El coste adicional es pequeño y está bien justificado si se aplican las técnicas que se expondrán en esta sección.

Con un sistema *Deshacer* multinivel se necesita también una operación de *Rehacer* (*redo*) para aquellos usuarios que se entusiasman y deshacen demasiado. Con un Deshacer de un nivel no es necesario ningún Rehacer; el convenio aplicado universalmente en este caso es que un Deshacer que siga inmediatamente a otro cancela el efecto de éste, de modo que el Deshacer y el Rehacer son la misma operación. Pero esto no funciona si se quiere retroceder más de un paso. De modo que habrá que tratar el Rehacer como una operación distinta.

Aspectos prácticos

Aunque la posibilidad de Deshacer-Rehacer puede añadirse posteriormente al sistema, con un esfuerzo razonable en un sistema O-O bien escrito, es mejor, si se tiene la intención de dar soporte a esta facilidad, hacerlo como parte del diseño, desde el propio comienzo del sistema.

Para hacer que el mecanismo de Deshacer-Rehacer sea útil hay que tener en cuenta algunas consideraciones prácticas. En primer lugar hay que incluir esta facilidad en la interfaz de usuario. Para empezar, se debe suponer que el conjunto de operaciones disponibles para los usuarios se va a enriquecer con dos nuevos tipos de solicitudes: Deshacer (que puede obtenerse por ejemplo tecleando Control-U, o Control –Z) y Rehacer (por ejemplo con Control-R). Deshacer cancela el efecto de la última orden que aún no haya sido deshecha; Rehacer reejecuta la última orden deshecha que aún no haya sido rehecha. Habrá que definir algún convenio para tratar los intentos de deshacer más de lo que se haya hecho (o más de lo que se ha registrado), o para rehacer más de lo que se haya deshecho: ignorar la solicitud, o dar un mensaje de aviso.

En segundo lugar, no todas las órdenes se pueden deshacer. En algunos casos se trata de una imposibilidad de hecho, como la orden "disparar los misiles" o una orden más en el contexto del software como "imprime el documento". En otros casos una orden podría teóricamente deshacerse, pero el coste no vale la pena; los editores de texto por ejemplo no dejan deshacer el efecto de una orden de Guardar (Save), que escribe el documento en curso en un archivo. La implementación de deshacer necesitará tener en cuenta las órdenes que no se pueden deshacer, y debe dejar clara

esta categoría en la interfaz de usuario. Hay que estar seguro que sólo se restrinja la orden que no se puede deshacer a aquellos casos para los cuales esta propiedad sea fácilmente justificable en términos de usuario.

Por último, puede ser tentador ofrecer, junto al Deshacer y Rehacer, un esquema más general de "Deshacer, Saltar y Rehacer" (Undo, Skip, Redo) que permite a los usuarios, después de efectuar una o más operaciones de Deshacer, saltar algunas ordenes antes de desencadenar un Rehacer. Sin embrago esto plantea un problema conceptual: después de saltar algunas órdenes, el próximo *Rehacer* podría ya no tener sentido. Como un ejemplo trivial se puede considerar una sesión de un editor de textos, con un texto que contiene una sola línea y un usuario que ejecuta las dos órdenes.

```
(1)   Añadir una línea al final.
(2)   Eliminar la segunda línea.
```

El usuario deshace las dos, luego quiere saltarse `(1)` y rehacer `(2)`. Desafortunadamente en este momento `(2)` no tiene sentido: no hay tal segunda línea. Esto es menos problema en la interfaz de usuario (se podría indicar al usuario que la orden es imposible) que en la implementación: la orden `Elimina la segunda línea` era aplicable a la estructura de objeto obtenida como resultado de `(1)`, pero aplicarla a la estructura de objetos que existía antes de `(1)` pudiera ser imposible (es decir, pudiera causar la caída del sistema u otros resultados desagradables). Ciertamente, hay soluciones posibles, pero no justifican el esfuerzo.

Requisitos de la solución

El mecanismo de *Deshacer-Rehacer* que se intentará proporcionar debiera satisfacer las propiedades siguientes.

- R1. El mecanismo debe ser aplicable a una amplia gama de aplicaciones interactivas, independientemente del dominio de aplicación.
- R2. El mecanismo no tendría por qué rediseñarse para cada nueva orden.
- R3. Debe hacer uso racional del almacenamiento.
- R4. Debiera ser aplicable tanto para Deshacer de un solo nivel como para el caso de múltiples niveles.

El primero de los requisitos es consecuencia de la observación de que no hay nada específico de la aplicación en deshacer y rehacer. Para facilitar la discusión se usará como ejemplo de aplicación una herramienta que es familiar para todos: un editor de textos sencillo (como el Bloc de notas), que permite a los usuarios insertar textos y llevar a cabo órdenes como `INSERTAR_LINEA`, `ELIMINAR_LINEA`, `SUSTITUCION_GLOBAL` (de una palabra por otra) y cosas parecidas. Pero sólo será un ejemplo y ninguno de los conceptos que se discuten a continuación es específico de los editores de texto.

El segundo requisito excluye tratar a *Deshacer* y a *Rehacer* como a las demás órdenes de un sistema interactivo. Si *Deshacer* fuese una orden, sería necesaria una estructura de la forma

```
if "la última orden fue INSERTAR_LINEA" then
      "Deshacer el efecto de INSERTAR_LINEA"
elseif "la última orden fue ELIMINAR_LINEA" then
      "Deshacer el efecto de ELIMINAR_LINEA"
etc.
```

Ya se sabe lo malas que son esta clase de estructuras, lo contrario de lo que indica el principio de elección única que hay que usar, para la extensibilidad. Hay que cambiarlas cada vez que se añade una orden; además, el código de cada rama reproducirá el código de la orden correspondiente (por ejemplo, la primera rama tiene que saber mucho sobre lo que hace INSERTAR_LINEA), lo cual nos indica un diseño mal hecho.

El tercer requisito indica que hay que ser inteligentes en el uso del almacenamiento. Está claro que dar soporte a *Deshacer* obliga a almacenar algo de información para cada *Deshacer*; por ejemplo cuando se ejecuta un ELIMINAR_LINEA, no se podrá deshacer posteriormente si al menos no se pone en algún lugar, antes de ejecutar la orden, una copia de la línea que se está eliminando y un registro de su posición en el texto. Pero debería almacenarse sólo lo que sea lógicamente necesario.

El efecto inmediato de este tercer requisito es excluir una solución obvia: guardar el estado del sistema como un todo – la estructura de objeto completa – antes de la ejecución de cada orden; para que el *Deshacer* se limite a restaurar la imagen guardada. Esto funcionaría pero sería un desperdicio terrible de espacio. Es lamentable, ya que sería trivial escribir la solución: bastaría utilizar las facilidades del lenguaje para almacenar y recuperar una estructura de objetos completa de un solo golpe. Hay que buscar algo un poco más sofisticado.

El requisito final, soportar una profundidad arbitraria de *Deshacer*, se ha discutido ya. Será más fácil considerar primero un mecanismo de nivel uno y luego generalizarlo a múltiples niveles. Estos requisitos completan la presentación del problema.

4.2.2. Encontrar las abstracciones

El paso clave de toda solución orientada a objetos es la búsqueda de la abstracción correcta. Aquí el concepto fundamental salta a la vista: las órdenes son candidatas idóneas para ser tratadas bajo un enfoque de orientación a objetos.

Las órdenes vistas como clases

El problema está caracterizado por una abstracción de datos fundamental: *ORDEN*, que representa cualquier operación del editor que no sea *Deshacer* o *Rehacer*. La ejecución es sólo una de las características que se pueden aplicar a una orden: la orden puede almacenarse, verificarse – o deshacerse. De modo que se necesita una clase que provisionalmente será de la forma

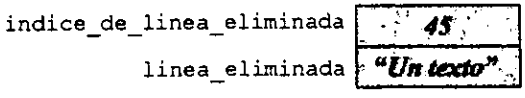
```
deferred class ORDEN feature
      ejecutar is deferred end
      deshacer is deferred end
end
```

ORDEN describe la noción abstracta de orden y por tanto debe quedar diferida. Los tipos reales de órdenes serán representados por los descendientes efectivos de esta clase, tales como.

```
class ELIMINACION_DE_LINEA inherit
      ORDEN
feature
      indice_de_linea_eliminada: INTEGER
      linea_eliminada: STRING
      actualizar_indice_linea_eliminada (n: INTEGER) is
            --Indica que el número de la línea a eliminar es n
          do
            indice_de_linea_eliminada:= n
          end
      ejecutar is
            --Elimina la línea
          do
            "Elimina la línea número indice_de_linea_a_eliminar"
            "Guarda el texto de la línea eliminada en linea_eliminada"
          end
      deshacer is
            --Restaura la última línea eliminada
          do
             "Pone de nuevo la linea_eliminada en la posición
                  indice_de_linea_eliminada"
          end
end
```

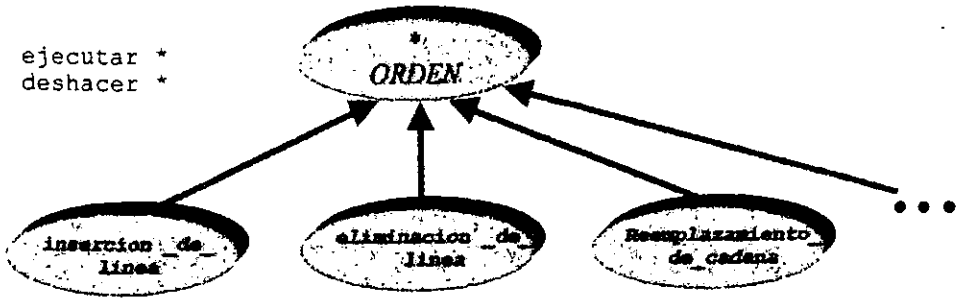
Y de manera similar para cada clase orden.

¿Qué es lo que representan esas clases? Una instancia de *ELIMINACION_DE_LINEA*, según se ilustra a continuación, es un pequeño objeto que almacena toda la información asociada a una ejecución de la orden: la línea que se elimina (*linea_eliminada*, un string) y su índice dentro del texto (*indice_de_linea_eliminada*, un entero). Ésta es la información que se necesita para deshacer la orden más adelante o para rehacerla.



Los atributos que se precisarán – tales como *indice_de_linea_eliminada* y *linea_eliminada* en este caso – serán diferentes para cada clase de orden, pero siempre deben ser suficientes para admitir las variantes locales de *ejecutar* y *deshacer*. Tales objetos, al describir conceptualmente la diferencia entre los estados que preceden y siguen a la aplicación de una orden, permiten satisfacer el requisito R3 de la pasada lista – almacenar sólo lo que sea estrictamente necesario.

La estructura de herencia de las clases de orden puede tener el aspecto siguiente:



El grafo que se muestra es plano (todos los descendientes de *ORDEN* están al mismo nivel), pero nada impide añadir más estructuras y agrupar los tipos de orden en categorías intermedias; esto se justificaría si dichas categorías tuvieran sentido como tipos de datos abstractos, es decir, si tuvieran características específicas.

Cuando se define un concepto, siempre es importante indicar lo que no abarca. Aquí el concepto de orden no incluye *Deshacer* y *Rehacer*, por ejemplo no tendría sentido deshacer un *Deshacer* (salvo en el sentido de hacer un *Rehacer*). Por esta razón, la discusión usa el término *operación* para *Deshacer* y *Rehacer*, reservando el término *orden* para las operaciones que se pueden deshacer y rehacer, tal como la inserción de una línea. No hay necesidad de una clase que abarque la noción de operación, ya que las operaciones que no son órdenes, tales como *Deshacer*, tienen una sola característica relevante, su capacidad de ejecutarse.

El paso interactivo básico

Para comenzar se verá la forma de admitir el Deshacer de un solo nivel. La generalización a un deshacer-rehacer multinivel se expondrá posteriormente.

En todo sistema interactivo tiene que haber en algún lugar, en un módulo que esté a cargo de la comunicación con los usuarios, algo como:

```
paso_interactivo_basico is
        --Decodificar y ejecutar una solicitud del usuario
    do
        "Determinar qué quiere el usuario que se haga a
        continuación "
        "Hacerlo (si es posible)"
    end
```

En un sistema estructurado tradicionalmente, como un editor, estas operaciones se ejecutarán como parte de un bucle, el "bucle básico del programa":

```
from inicio until la_salida_se_haya_solicitado_y_confirmado loop
    paso_interactivo_basico
end
```

mientras que sistemas más sofisticados pueden utilizar un esquema controlado por eventos, en el cual el bucle es externo al sistema propiamente dicho (y es gestionado por el entorno gráfico subyacente). Pero en todos los casos se necesita algo parecido a *paso_interactivo_basico*.

A la luz de las abstracciones que se acaban de identificar, se puede reformular el cuerpo del procedimiento en la forma

```
"Tomar la última solicitud del usuario y decodificarla"
if "Si la solicitud es de una orden normal (no un Deshacer)" then
   "Determinar cuál es la orden correspondiente en el sistema"
   "Ejecutar dicha orden"
else if "La solicitud es de un Deshacer" then
     if "Si hay alguna orden que deshacer" then
        "Deshacer la última orden"
     else if "Hay alguna orden que Rehacer" then
      "Rehacer la última orden"
     end
else
     "Informar que se ha introducido una solicitud errónea"
end
```

Esto implementa la convención sugerida anteriormente, consistente en que un *Deshacer* aplicado justamente después de un *Deshacer* (en el caso de un solo nivel) significa un *Rehacer*. La solicitud de Deshacer o Rehacer se ignorará si no hay nada que deshacer o rehacer. En un editor de texto sencillo con una interfaz por teclado, el "Decodificar la solicitud" analizaría la entrada del usuario, buscando códigos como control-I (para insertar una línea), control-D (para eliminar una línea) y así sucesivamente. En el caso de las interfaces gráficas hay que determinar cuál es la entrada producida por el usuario, que puede ser la selección de una opción de un menú, un botón en que se ha hecho clic, o la tecla que se haya pulsado.

Recordar la última orden

Con la noción de objeto se puede ser más específico sobre las operaciones llevadas a cabo por *paso_interactivo_basico*. Se usará un atributo

```
solicitud: ORDEN
           --Orden solicitada por el usuario interactivo
```

que representa la última orden que hay que ejecutar, deshacer, o rehacer. Esto nos permite refinar el esquema anterior de *paso_interactivo_basico* para obtener:

```
"Tomar la última solicitud del usuario y decodificarla"
if "La solicitud es de una orden normal (no un Deshacer)" then
   "Crear el objeto orden apropiado y conectarlo a solicitud"
      --Solicitud se crea como una instancia de algún
      --descendiente de ORDEN, tal como ELIMINACION_DE_LINEA
      --(esta instrucción se detalla más adelante)
      solicitud.ejecutar; modo_deshacer:= False
else if "la solicitud es Deshacer" and solicitud /= Void then
      if modo_deshacer then
         "Es entonces un Rehacer"
      else
      solicitud.deshacer; modo_deshacer:= True
      end
else
      "Solicitud errónea: dar una advertencia, o no hacer nada"
end
```

Ligadura dinámica

La información que se almacena antes de la ejecución de cada orden es una instancia de algún descendiente de *ORDEN* tal como *ELIMINACION_DE_LINEA*. Esto significa que, según se indicaba, la solución satisface la propiedad R3 de la lista de requisitos: lo que se almacena para cada orden es la diferencia entre el nuevo estado y el anterior, no el estado completo.

La clave de esta solución –y su refinamiento posterior – es el polimorfismo y la ligadura dinámica. El atributo *solicitud* es polimórfico; esta declarado como de tipo *ORDEN* por lo cual estará conectado a objetos de alguno de sus tipos descendientes efectivos, tal como *INSERCION_DE_LINEA*. Las llamadas *solicitud.ejecutar* y *solicitud.deshacer* sólo tienen sentido gracias a la ligadura dinámica: la característica que se disparará debe ser la versión redefinida para la correspondiente clase de orden, una *INSERCION_DE_LINEA* o una orden de algún otro tipo que se determina a partir del objeto al que esté conectado *solicitud* en el momento de la llamada.

Las acciones del sistema

Ninguna parte de la estructura vista hasta este momento es específica de la aplicación. Las operaciones concretas de la aplicación, basadas en sus estructuras de objetos específicas –por ejemplo las estructuras que representan al texto en curso en el caso del editor de texto – están en otra parte; ¿cómo se puede establecer la conexión?

La respuesta se basa en los procedimientos *ejecutar* y *deshacer* de las clases órdenes, que deben invocar a características específicas de la aplicación. Por ejemplo el procedimiento *ejecutar* de la clase *ELIMINACION_DE_LINEA* debe tener acceso a las clases especificas del editor para llamar a las características que tengan acceso al texto de la línea en curso, a su posición en el texto completo y a eliminarla.

Como resultado hay una separación clara entre las partes de interacción con el usuario de un sistema, que en gran medida son independientes de la aplicación, y las partes especificas de la aplicación, que están más cerca del modelo conceptual de la

aplicación – sea procesamiento de texto, para CAD-CAM o para cualquier otra cosa. El primer componente; especialmente cuando se generaliza a un mecanismo de historia como se explica a continuación, será ampliamente reutilizable en varios dominios diferentes de aplicación.

Forma de crear un objeto orden

Después de decodificar una solicitud, el sistema debe crear el correspondiente objeto orden. La instrucción aparece en forma abstracta como "`Crear el objeto orden apropiado y asociarlo a la solicitud`"; se puede expresar con más precisión, usando instrucciones de creación, como en

```
if "la solicitud es INSERCION_DE_LINEA" then
      !INSERCION_DE_LINEA!solicitud.make(texto_de_entrada,
        indice_cursor)
else if
      !ELIMINACION_DE_LINEA!solicitud.make(linea_en_curso,
        indice_de_linea)
else if
      . . .
```

Esto hace uso de una instrucción de creación de la forma ! *ALGUN_TIPO* ! *x* ..., que crea un objeto de *ALGUN_TIPO* y lo conecta a *x*; recuérdese que *ALGUN_TIPO* debe ajustarse al tipo declarado para *x*, tal como sucede aquí, por cuanto *solicitud* está declarado como de tipo *ORDEN* y todas las clases de órdenes son descendientes de *ORDEN*.

Si para cada tipo de orden se usa un entero o código carácter **unique**, existe una forma ligeramente más sencilla que se basa en un **inspect**:

```
inspect codigo_de_la_solicitud
when insercion_de_linea then
     ! INSERCION_DE_LINEA ! solicitud.make (texto_de_entrada,
       posicion_cursor)
     etc.
```

Las dos formas son selecciones multivía, pero no violan el principio de elección única: tal como se indicaba en la exposición de ese principio, si un sistema proporciona un cierto número de alternativas entonces alguna parte del sistema *debe* conocer la lista completa de alternativas. El extracto anterior, en cualquiera de sus variantes, es ese punto de elección única. Lo que el principio excluye es dispersar ese conocimiento en muchos módulos. Aquí, no hay ninguna otra parte del sistema que se necesite acceder a la lista de órdenes; cada clase *ORDEN* trata solamente un tipo de órdenes.

4.2.3. Deshacer-Rehacer de múltiples niveles

La posibilidad de una profundidad arbitraria para deshacer, con la correspondiente posibilidad de rehacer, es una extensión elemental del esquema anterior.

La lista histórica

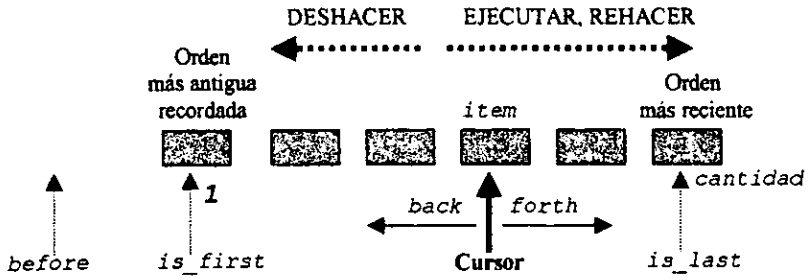
Lo que nos ha limitado a un único nivel de *Deshacer* era el utilizar un solo objeto, la última instancia creada de `ORDEN` que está disponible a través de `solicitud`, como único registro de las órdenes ejecutadas con anterioridad.

Para brindar mayor profundidad de deshacer es necesario sustituir la `solicitud` de una sola orden por una lista de órdenes ejecutadas más recientemente, la lista histórica:

```
historia: CIERTA_LISTA [ORDEN]
```

`CIERTA_LISTA` no es el nombre de una clase real; en una especie de tipo abstracto de dato; siguiendo fielmente el estilo orientado a objetos, se examinarán las características y propiedades que se necesitan en `CIERTA_LISTA` para llegar a la conclusión de cuál de las clases de *listas* se puede utilizar. Las operaciones principales que necesitamos son elementales y fáciles de comprender.

- `put` para insertar un elemento al final (el único lugar en el que se necesitarán las inserciones), `put` situará el cursor de la lista sobre el elemento que se acaba de insertar.
- `empty` para determinar si la lista está vacía.
- `is_first`, `is_last` y `before` para responder a las preguntas sobre la posición del cursor.
- `back` para mover el cursor una posición hacia atrás y `forth` para avanzar el cursor una posición.
- `item` para acceder al elemento de la posición del cursor, si lo hay; esta característica tiene la precondición (**not** `empty`) **and** (**not** `before`), que se puede expresar con una nueva consulta `on_item`.



En ausencia de un *Deshacer*, el cursor siempre estará (excepto para una lista vacía) sobre el último elemento, lo que hace que `is_last` sea verdadero. Si el usuario empieza un *Deshacer* el cursor se moverá hacia atrás en la lista; si inicia un *Rehacer*, el cursor se moverá hacia delante.

La figura muestra al cursor sobre un elemento que no es el último; esto significa que el usuario ha ejecutado uno o más *Deshacer*, posiblemente intercalado con algunos *Rehacer*, aunque el número de *Deshacer* debe ser siempre al menos tan grande como el número de *Rehacer* (en el estado capturado por la figura es mayor por valor de dos). Si en tal estado el usuario selecciona una orden normal – es decir que no es ni *Deshacer* ni *Rehacer* – el elemento correspondiente debe insertarse inmediatamente a la derecha del elemento del cursor. En este caso los restantes elementos de la derecha se pierden, puesto que no tiene ya sentido un *Rehacer* en este caso. Como consecuencia, necesitamos una característica más en `CIERTA_LISTA`: el procedimiento `remove_all_rigth`, que elimina todos los elementos situados a la derecha del cursor.

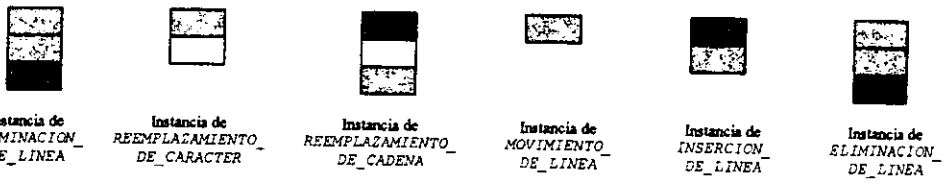
Un *Deshacer* es posible si y sólo si el cursor está sobre un elemento, según lo establece `on_item`. Un *Rehacer* es posible si y sólo si hay al menos un *Deshacer* que no haya sido anulado, es decir si (**not** *empty*) **and** (**not** `is_last`), lo cual se puede expresar a través de una consulta denominada `not_last`.

Implementación de Deshacer

Teniendo la lista histórica es fácil entonces implementar `Deshacer`.

```
if on_item then
      historia.item.deshacer
      historia.back
else
      mensaje ("No hay nada que deshacer")
end
```

Obsérvese de nuevo lo esencial que es la ligadura dinámica. La lista histórica es una estructura de datos polimorfa:



A medida que el cursor se mueve hacia la izquierda, cada valor sucesivo de `historia.item` puede estar conectado a un objeto de cualquiera de los tipos de orden disponibles; en cada caso, la ligadura dinámica asegura que `historia.item.deshacer` selecciona automáticamente la versión apropiada de `deshacer`.

Implementación de Rehacer

Rehacer es similar:

```
if not_last then
      historia.forth
      historia.item.rehacer
else
      mensaje ("No hay nada que rehacer")
end
```

Esto supone la existencia de un nuevo procedimiento, `rehacer`, en la clase `ORDEN`. Hasta ahora se ha dado por cierto que `rehacer` era lo mismo que `ejecutar`, y de hecho así es en la mayoría de los casos; pero para algunas órdenes volver a ejecutar después de un Deshacer pudiera ser algo diferente de ejecutar a partir de cero. La mejor manera de manejar tales situaciones – proporcionando suficiente flexibilidad, pero sin sacrificar la comodidad para los casos comunes – es proporcionar en la clase `ORDEN` el comportamiento por defecto para `rehacer`:

```
rehacer is
      --Vuelve a ejecutar la orden que se ha deshecho
      --por defecto esto es lo mismo que ejecutarlo.
do
      ejecutar
end
```

Esto hace de `ORDEN` una clase comportamiento: junto con sus diferidos `ejecutar` y `deshacer` tiene un procedimiento efectivo `rehacer` que define un comportamiento basado, por defecto, en los otros dos. La mayoría de los descendientes conservará probablemente este comportamiento por defecto, pero algunos de ellos pudieran redefinir `rehacer` para tener en cuenta algunos casos especiales.

Ejecutar una orden normal

Si una operación de usuario no es ni *Deshacer* ni *Rehacer*, entonces es una orden normal identificada por su referencia que pudiera seguir llamándose `solicitud`. En este caso se debe ejecutar dicha orden, pero se debe insertar en la lista histórica; además, como ya se ha hecho notar, habrá que olvidar todos los elementos que estén a la derecha del cursor. De modo que la secuencia de instrucciones queda de la forma:

```
if not is_last then remove_all_right end
   historia.put (solicitud)
      --Recordar que put inserta al final de la lista y mueve el
      --cursor hasta este nuevo elemento.
solicitud.ejecutar
```

Con esto ya se han visto todos los elementos esenciales de la solución.

4.2.4. Aspectos de implementación

A continuación se examinarán algunos detalles que ayudan a obtener la mejor implementación posible.

Precalcular objetos órdenes

Antes de ejecutar una orden se debe obtener, y en algunos casos crear, el objeto orden correspondiente. La instrucción se escribía de forma abstracta como "Crear el objeto orden apropiado y conectarlo a solicitud" y el primer borrador de implementación era

```
inspect codigo_de_solicitud
when insercion_de_linea then
     ! INSERCION_DE_LINEA ! solicitud.crear (..).
etc. (una rama por cada tipo de orden)
```

Como ya se señaló, esta instrucción no viola el principio de elección única: de hecho es el punto de la elección única – el único lugar en todo el sistema que conoce cuál es el conjunto de órdenes. Pero hasta ahora se ha desarrollado una aversión por las instrucciones `if` o `inspect` con muchas ramas, de modo que aunque ésta parezca inevitable en principio, se verá que se puede evitar.

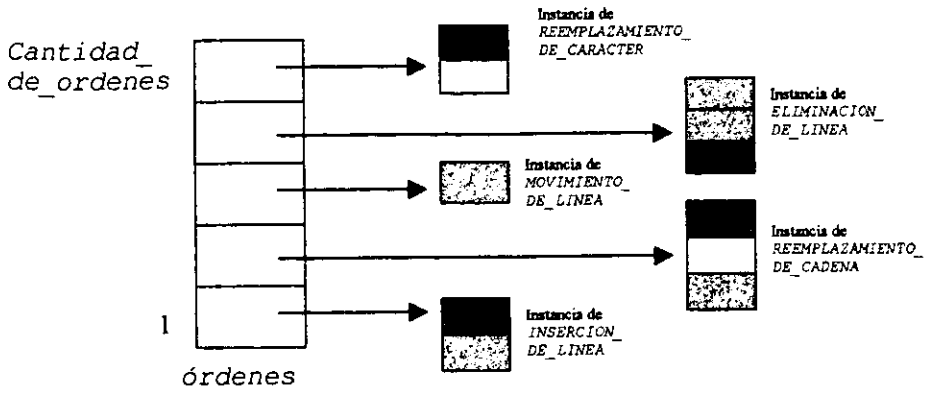
La forma de hacerlo, es mediante un patrón de diseño, que se puede denominar **precalcular un conjunto de instancias polimorfas**, que es ampliamente aplicable.

La idea es simplemente, crear de una vez por todas una estructura de datos polimorfa que contenga una instancia de cada variante; posteriormente, cuando se necesite un nuevo objeto bastará obtener la entrada correspondiente en la estructura.

Aunque serían posibles varias estructuras de datos para una lista como está, es más conveniente utilizar un *ARRAY [ORDEN]*, que permitirá identificar a cada tipo de orden mediante un entero entre 1 y *cantidad_de_órdenes* (que es el número de tipos de órdenes). De modo que se declara

```
órdenes: ARRAY [ORDEN]
```

y se inician sus elementos de manera tal que el *i-ésimo* elemento (`1 <= i <= n`) se refiera a una instancia de la clase descendiente de *ORDEN* correspondiente al código `i`; por ejemplo, se crea una instancia de *ELIMINACION_DE_LINEA* y se asocia al primer elemento del array (suponiendo que eliminar una línea tenga el código 1), y así sucesivamente.



El array de plantillas de órdenes

El array *órdenes* es otro ejemplo de la potencia de las estructuras de datos polimorfas. Su inicialización es trivial:

```
!! ordenes.make (1, cantidad_de_ordenes)
! INSERCION_DE_LINEA ! solicitud.crear; ordenes.put (solicitud, 1)
! REEMPLAZAMIENTO_DE_CADENA !solicitud.crear;ordenes.put (solicitud, 2)
... y sucesivamente para cada tipo de orden.
```

Obsérvese que con este enfoque los procedimientos de creación de varias clases de órdenes no tendrían que tener ningún argumento; si una clase orden tiene atributos, deben actualizarse separadamente más tarde a través de procedimientos específicos, como en *il.make (texto_de_entrada, posicion_cursor)* donde *il* es de tipo *INSERCION_DE_LINEA*.

Entonces ya no hay necesidad de ninguna instrucción condicional múltiple **if** o **inspect**. La inicialización anterior hace las veces del punto de elección única. Ahora se puede entonces escribir la operación de "Crear el objeto orden apropiado y conectarlo a *solicitud*" de la forma

```
solicitud:= clone (órdenes @ codigo)
```

donde *codigo* es el código de la última orden. (Puesto que cada tipo de orden tiene ahora un código, correspondiente a su índice en el array, la operación básica de interfaz de usuario que se escribía antes como "*decodificar solicitud*" analiza la solicitud del usuario y determina el código correspondiente.)

La asignación a *solicitud* hace uso de un *clone* de la orden que está en el array que actúa de molde, de modo que la lista histórica puede contener más de una instancia de la misma orden (como en el ejemplo anterior, en que se incluían en la lista histórica dos objetos *ELIMINACION_DE_LINEA*).

Sin embargo, si se usa la técnica sugerida de separar completamente los argumentos de la orden de los objetos orden (de modo que la lista histórica contenga

instancias de *INSTANCIA_ORDEN* en lugar de *ORDEN*), entonces este duplicado no es necesario y se pueden poner las referencias directas a los objetos en el array haciendo:

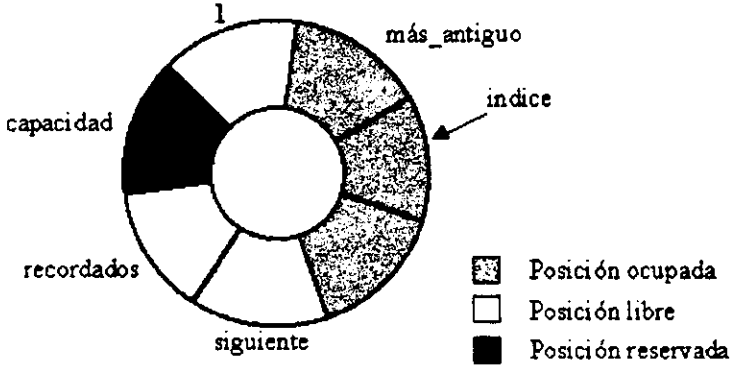
```
solicitud:= órdenes @ codigo
```

En sesiones de trabajo largas estos ahorros pueden ser significativos.

Una representación para la lista histórica

Para la lista histórica se planteaba un tipo *CIERTA_LISTA* con las características *put*, *empty*, *before*, *is_first*, *is_last*, *back*, *forth*, *item* y *remove_all_right*. (También hay un *on_item* que se expresaba en términos de *empty* y *before*, y un *not_last* que se expresaba en téminos de *empty* e *is_last*.)

Para implementar *CIERTA_LISTA* se confeccionará una clase denominada *LISTA_LIMITADA*. Ésta se basará en un array, de modo que en la historia se conserva sólo un número limitado de órdenes. Sea *recordados* el número máximo de órdenes que se recordarán; la clase *LISTA_LIMITADA* puede utilizar este array, gestionándolo de forma circular reutilizando las posiciones iniciales a medida en que el número de órdenes sobrepasa los *recordados*. Con esta técnica, que es común para representar colas limitadas se puede dibujar el array cerrado en forma de rosquilla:



El tamaño *capacidad* del array es *recordados + 1*; esta convención deja reservada una de las posiciones (la última, la de índice *capacidad*) para distinguir entre una lista vacía y una lista llena. Las dos posiciones ocupadas están marcadas por dos atributos enteros: *mas_antiguo* que es la posición que ocupa la orden que lleva más tiempo entre los registrados, y *siguiente* que es la primera posición libre (en la que se insertará la siguiente orden). El atributo entero *indice* indica la posición actual del cursor.

A continuación se verá la implementación de las distintas características. Para *put* (*c*), que inserta la orden *c* al final de la lista, se ejecuta:

```
representacion.put (x, siguiente)    --Donde representación es el
                                       nombre del array
siguiente:= (siguiente \\ recordados) + 1
indice:= siguiente
```

donde \\ es la operación que proporciona el resto entero de la división. El valor de *empty* es verdadero si y sólo si *siguiente = mas_antiguo*; el de *is_first* es verdadero si y sólo si *indice = mas_antiguo* y el de *before* si y sólo si (indice \\ recordados) + *1 = mas_antiguo*. El cuerpo de *forth* es

```
indice:= (indice \\ recordados) + 1
```

y el cuerpo de *back* es

```
indice:= ( (indice + recordados - 2) \\ recordados) + 1
```

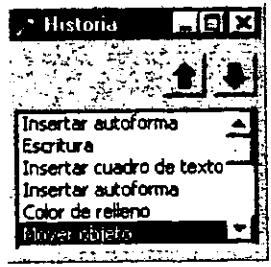
La consulta *item* que da el elemento situado en la posición del cursor devuelve *representación @ indice* que es el elemento del array cuyo índice es *indice*. Por último, el procedimiento *remove_all_right*, que elimina todos los elementos situados a la derecha de la posición del cursor, se implementa sencillamente como

```
siguiente:= (indice \\ recordados) + 1
```

4.2.5. Una interfaz de usuario para deshacer y rehacer

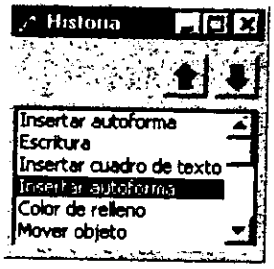
A continuación se sugiere una posible interfaz para el mecanismo de *Deshacer-Rehacer*. Tiene la apariencia del formato usado en herramientas basadas en sistema operativo Windows de Microsoft, e incluso de otras muchas que hagan uso extensivo de una Interfaz Gráfica de Usuario.

Aunque hay combinaciones de teclado para Deshacer y Rehacer, el mecanismo completo implica desplegar una ventana de historia (haciendo clic sobre un botón de la interfaz o seleccionando un elemento en el menú de herramientas (*Tools*)). La ventana de historia es el equivalente visible por el usuario de la lista de historia tal y como existe dentro de muchos paquetes de software. Una vez que se despliega, será actualizada regularmente cada vez que se ejecute una orden y otras operaciones. En ausencia de deshacer, tendrá un aspecto al siguiente:



Esto muestra la lista de las órdenes recientes. A medida en que se ejecutan nuevas órdenes, irán apareciendo al final de la lista. La orden activa en curso (sobre la cual está la posición del cursor) se destaca en vídeo inverso, como el caso de **Mover objeto**.

Para deshacer la orden activa se puede hacer clic en el botón de la flecha hacia arriba o se puede utilizar una combinación de teclado (por ejemplo ALT-U). El cursor se desplaza hacia arriba (hacia atrás) en la lista; después de unos cuantos Deshacer, la ventana tendría este aspecto:



Como ya se sabe, internamente esto significa que el software ha realizado algunas llamadas a `back`. En este momento tenemos la opción de escoger varias posibilidades:

- Se pueden realizar muchas operaciones de Deshacer haciendo clic en el botón de flecha hacia arriba, la línea destacada en vídeo inverso pasará a ser la anterior.
- Se pueden efectuar uno o más Rehacer haciendo clic en el botón de flecha hacia abajo o usando la combinación de teclado equivalente; la línea en vídeo inverso pasa a la siguiente llevando a cabo internamente llamadas a `forth`.
- Se puede ejecutar una orden normal. Como se ha visto, esto eliminará de la lista histórica cualquier orden a la que se le haya hecho deshacer pero no rehacer, efectuando internamente una llamada a `remove_all_right`; en la interfaz, desaparecerán todas las órdenes que estén por debajo de la línea marcada.

4.2.6 Consideraciones del problema

El patrón de diseño que se ha presentado en esta sección tiene un importante papel práctico, ya que permite escribir sistemas software interactivos significativamente mejores con poco esfuerzo adicional. También brinda una interesante contribución teórica al poner de manifiesto algunos aspectos de la metodología orientada a objetos que vale la pena explorar un poco más.

El papel de la implementación

Una propiedad del ejemplo de interfaz de usuario presentada anteriormente es que éste *se ha deducido directamente* de la implementación: se tomó la noción interna, relevante para el desarrollador, de lista histórica y se tradujo a una noción externa,

relevante para el usuario, que es la de ventana de historia, con su correspondiente mecanismo de interacción con el usuario.

Al establecer una relación como ésta entre la funcionalidad del sistema y su implementación se va en contra de lo que enseña la metodología tradicional de ingeniería del software. Se nos ha dicho que hay que deducir la implementación de la especificación, pero no al revés. Las técnicas de "desarrollo iterativo" y de la "espiral del ciclo de vida" cambian poco esta regla fundamental de que la implementación es esclava del concepto previo y de que los desarrolladores de software deben hacer lo que los "usuarios" (refiriéndose a los clientes, que no suelen ser usuarios técnicos) le digan.

El énfasis legítimo en que hay que implicar a los usuarios – destinado a evitar las *historias de terror* de sistemas que no hacen lo que los usuarios necesitan – nos ha llevado desafortunadamente a infravalorar la contribución de los desarrolladores de software, cuya importancia se extiende tanto a los aspectos más externos como a los relacionados con la aplicación. Por ejemplo, es ingenuo creer que los usuarios van a sugerir las capacidades de interfaz correctas. Algunas veces podrán hacerlo, pero a menudo van a razonar sobre la base del sistema que conocen y no verán todos los problemas implicados. Esto es comprensible: ellos tienen su propio trabajo que hacer, y sus propias áreas de conocimiento técnico; hacerlo todo bien en un sistema software no es su responsabilidad. Algunas de las peores interfaces de usuario han sido diseñadas con demasiada influencia de los usuarios. El lugar en que los usuarios son irremplazables es en los comentarios negativos: ellos verán las debilidades prácticas de una idea que en principio pueda parecer atractiva para los desarrolladores. Esas críticas siempre deben tenerse en cuenta. Los usuarios podrían hacer también unas sugerencias positivas brillantes, pero no se debe fiar uno de ellas. Y alguna que otra vez, una sugerencia del desarrollador seducirá a los usuarios – posiblemente después de un cierto número de iteraciones considerando sus críticas – aun cuando debe su origen a una técnica de implementación tan sencilla, tal como la lista histórica.

Esta igualación de las relaciones tradicionales es una de las contribuciones distintivas de la tecnología de objetos. Al hacer que el proceso de desarrollo carezca de discontinuidades y sea reversible se hace posible que una gran idea de implementación influya sobre la especificación. En lugar de un flujo en una sola dirección, del análisis al diseño y a la "codificación", se tiene un proceso continuo con ciclos de realimentación a todo lo largo. Esto supone, por supuesto, que la implementación ya no se ve como la parte sucia, como el componente de bajo nivel de la construcción del sistema. Sus resultados, desarrollados con técnicas correctas, pueden y deben ser tan claros, elegantes y abstractos como cualquiera que se pueda producir en las formas de análisis y diseño tradicionales.

4.3. Análisis orientado a objetos

Formulado inicialmente para abordar los aspectos de implementación de la construcción del software, el método orientado a objetos se extendió rápidamente hasta abarcar la totalidad del ciclo de vida del software. Resultan especialmente interesantes las aplicaciones de las ideas O-O al modelado de sistemas software, o incluso a los sistemas y problemas que no son software. Esta utilización de la tecnología de objetos para plantear problemas más que soluciones se conoce con el nombre de **análisis orientado a objetos**.

Aquí se revisará brevemente lo que hace que el análisis orientado a objetos sea especial con respecto a otros métodos de análisis y como aprovecha las técnicas de la orientación a objetos para comenzar a modelar correctamente un sistema desde el principio.

4.3.1. Objetivos del análisis

Para comprender las cuestiones relacionadas con el análisis es preciso ser consciente de los papeles que desempeña el análisis en el desarrollo del software, para esto es imperativo definir los requisitos de un método de análisis.

Tareas

Dedicando tiempo al análisis y producción de documentos de análisis se persiguen siete objetivos:

OBJETIVOS DE LA REALIZACIÓN DE UN ANÁLISIS

- **A1** • *Comprender el problema o problemas que tendrá que resolver el sistema software, si llega a existir.*
- **A2** • *Suscitar cuestiones relevantes acerca del problema y del sistema.*
- **A3** • *Proporcionar una base para responder preguntas acerca de propiedades específicas del problema y del sistema.*
- **A4** • *Decidir lo que tiene que hacer el sistema.*
- **A5** • *Decidir lo que no tiene que hacer el sistema.*
- **A6** • *Asegurar que el sistema satisfaga las necesidades de sus usuarios, y definir los criterios de aceptación (especialmente cuando el sistema se desarrolla para un cliente externo con una relación contractual)*
- **A7** • *Proporcionar una base para el desarrollo del sistema.*

Si se está aplicando el análisis a un sistema que no sea de software, o independientemente de una decisión para construir un sistema de software, A!, A2 y A3 pueden ser los únicos objetivos relevantes.

Para un sistema software, la lista supone que el análisis sigue a una fase de *estudio de viabilidad* que haya dado lugar a la decisión de construir un sistema. Si, como sucede en ocasiones, las dos fases se fusionan en una sola (lo cual no es una

proposición absurda, por cuanto quizás se necesite un análisis en profundidad para determinar si es concebible un resultado satisfactorio), entonces la lista necesita otro elemento: A0, la decisión de construir o no el sistema.

Los objetivos A2 y A3 son los peor cubiertos por la literatura relativa al análisis, y merecen todo el énfasis que puedan recibir. Uno de los beneficios principales de un proceso de análisis, independientemente del documento que pueda producir al final, es que nos lleva a efectuar las preguntas relevantes (A2): ¿Cuál es la máxima temperatura admisible? ¿Cuáles son las categorías de empleados reconocidas? ¿En qué difiere el manejo de las acciones y el de los bonos? Al proporcionarnos un marco de trabajo, que será preciso rellenar empleando información procedente de otras personas competentes en el dominio de la aplicación, un método de análisis nos ayudará a detectar y eliminar las oscuridades y ambigüedades que pueden ser fatales para el desarrollo. En lo tocante a A3, un buen documento de análisis será el lugar al que todo mundo recurra cuando surjan preguntas delicadas o interpretaciones conflictivas durante el proceso de desarrollo.

Requisitos

Los requisitos prácticos de análisis y de las notaciones que le sirven de apoyo se derivan de la lista anterior de objetivos:

- Tiene que haber una forma de permitir que las personas que no pertenezcan a la comunidad del software aporten información para el análisis, examinen los resultados y los discutan (A1, A2).
- El análisis también debe tener una forma que sea utilizable directamente por parte de los desarrolladores de software (A7).
- El enfoque debe poder crecer a escala (A1).
- La notación de análisis debe ser capaz de expresar propiedades precisas sin ambigüedades (A3).
- Tiene que capacitar a los lectores para obtener una visión rápida de la organización general del sistema o de cualquier subsistema (A1, A7).

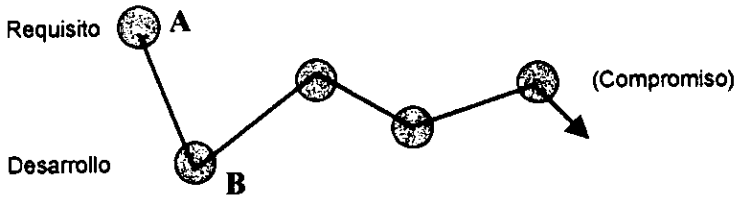
El crecer a escala (tercer punto) significa ocuparse de sistemas que sean complejos, grandes, o ambas cosas – aquellos para los cuales se necesita más el análisis. El método debería capacitarnos para describir la estructura de alto nivel del problema o del sistema, y para organizar la descripción en varios niveles de abstracción, de tal modo que sea posible en cualquier momento centrarse en una parte tan grande o tan pequeña del sistema como se desee, manteniendo al mismo tiempo la visión general. Aquí, por supuesto, las capacidades de estructuración y de abstracción propias de las tecnologías de objetos serán sumamente valiosas.

El crecer a escala significa también que los criterios de extensibilidad y reutilización, que han guiado gran parte de las secciones anteriores, siguen siendo tan aplicables al análisis como lo son al diseño e implementación de software. Los sistemas cambian, y esto requiere que sus descripciones los sigan; y los sistemas son similares a los sistemas anteriores, lo cual nos impulsa a utilizar bibliotecas de elementos de especificación para construir sus especificaciones, del mismo modo que se utilizan bibliotecas de componentes software para construir sus implementaciones.

4.3.2. La naturaleza cambiante del análisis

La contribución más significativa de la tecnología de objetos al análisis no es técnica sino más bien organizativa. La tecnología de objetos no se limita a proporcionar nuevas formas de realizar el análisis; afecta a la misma naturaleza de la tarea y a su papel en el proceso del software.

Este cambio procede del hincapié efectuado por el método en la reutilización. Si en lugar de suponer que todo proyecto nuevo tiene que comenzar partiendo de cero, considerando los requisitos del cliente como un mandamiento divino, se introduce en el escenario un repertorio de componentes de software que crece regularmente, algunos de los cuales han sido obtenidos (o se pueden obtener) y otros han sido desarrollados como resultado de proyectos internos, entonces el proyecto se vuelve distinto: no es la ejecución de una orden procedente de lo alto, sino una **negociación**.



El análisis de requisitos como negociación

La figura sugiere este proceso: el cliente comienza con un requisito **A**; nosotros contraatacamos con una propuesta **B**, que abarca quizás tan sólo una parte de los requisitos, o bien una forma ligeramente distinta de los requisitos, pero que está basado en su mayor parte en componentes reutilizables ya existentes y que por tanto se pueden alcanzar con un coste significativamente menor y además en menos tiempo. Es posible que el cliente encuentre inicialmente demasiado grande el sacrificio de funcionalidad; esto abre una fase de regateo que debería eventualmente llevarnos a un compromiso aceptable.

Este regateo siempre ha estado ahí, por supuesto. Los requisitos del cliente solamente eran un mandato divino en algunas descripciones del "proceso del software" dentro de la literatura de ingeniería de software, que presenta una visión idealizada a efectos pedagógicos, y quizás también en algunos contratos del gobierno. Pero en la mayoría de las situaciones normales, los desarrolladores tienen una cierta libertad para discutir los requisitos. Con el advenimiento de la tecnología de objetos, este fenómeno oficioso pasa a formar parte oficialmente del proceso del desarrollo del software, y gana una nueva prominencia con el desarrollo de bibliotecas reutilizables.

4.3.3. La contribución de la tecnología de objetos

La tecnología de objetos afecta también, por supuesto, a las técnicas de análisis. Sus principales aportaciones al modelado de sistemas son:

- Las **clases** nos permitirán organizar las descripciones en nuestro sistema con respecto a tipos de objetos, en el sentido amplio de la palabra "objeto" (que no abarca solamente los objetos físicos sino también conceptos importantes del dominio de la aplicación).
- El enfoque **TAD** – la idea de caracterizar los objetos por las operaciones aplicables y sus propiedades – proporciona unas especificaciones claras, abstractas y evolutivas.
- Para capturar las relaciones entre componentes, los dos mecanismos básicos de "cliente" y herencia son perfectamente adecuados. La relación **cliente**, en particular, abarca conceptos del modelado de la información tales como "parte de", la asociación y la agregación.
- La **herencia** aborda la clasificación. Incluso los mecanismos de herencia más especializados resultarán sumamente valiosos para modelar conceptos de análisis.
- Las **aserciones** son esenciales para capturar la semántica de los sistemas: son aquellas propiedades que no son estructurales.
- Las **bibliotecas** de clases reutilizables nos proporcionarán – sobre todo a través de sus clases diferidas de nivel superior – unos elementos de especificación ya prefabricados.

Esto no significa necesariamente que el enfoque orientado a objetos abarque todas las necesidades del análisis de sistemas; pero ciertamente proporciona una base correcta. A continuación se ofrecerá un ejemplo que muestra evidencias claras de cómo las técnicas del método orientado a objetos colaboran en el modelado de sistemas desde su fase de análisis.

4.3.4. Programación de una emisora de televisión

Se verá concretamente la forma de aplicar los conceptos O-O al modelado puro.

El ejemplo implica la planificación de una emisora de televisión. Dado que se ha extraído de un área de aplicación familiar, es posible comenzarlo (aunque con mucha seguridad no podría concluirse) sin el beneficio de informaciones procedentes de "expertos del dominio", futuros usuarios, etc.; por lo que para fines prácticos se tomará como base para el ejercicio de análisis la comprensión que posee cualquier persona acerca de la TV.

Aun cuando este esfuerzo puede ser el preludio para la construcción de un sistema computarizado para gestionar automáticamente la programación de la emisora, esta posibilidad no es ni cierta ni relevante aquí; el interés esta centrado únicamente en el modelado.

Horarios

Para comenzar se tomará como punto de partida un horario para un periodo de 24 horas; por supuesto, la clase (abstracción de datos) *HORARIO* se presenta a sí misma. Un horario contiene una sucesión de segmentos individuales de programa; por lo que es factible empezar con

```
class HORARIO feature
      segmentos: LIST[SEGMENTO]
end
```

Cuando se efectúa un análisis, es preciso vigilar constantemente por miedo a caer en una excesiva especificación. ¿Sería una excesiva especificación utilizar *LIST*? No: *LIST* es una clase diferida, que describe el concepto abstracto de sucesión; la programación de televisión es ciertamente secuencial, por cuanto uno no puede retransmitir dos segmentos por la misma emisora al mismo tiempo. Al utilizar *LIST* se captura una propiedad del problema, mas no la solución.

Adicionalmente, obsérvese la importancia de la reutilización: al utilizar clases como *LIST* se tiene acceso inmediatamente a todo un conjunto de características que describen las operaciones con listas: órdenes tales como *put* para añadir elementos, consultas tales como el número de elementos *count*. La reutilización es tan importante para el análisis orientado a objetos como lo es para otras tareas O-O.

Lo que sería una especificación excesiva aquí sería *igualar* el concepto de horario con el de lista de segmentos. La tecnología de objetos, según se recordará de la sección de tipos abstractos de datos, es implícita; describe las abstracciones mediante la enumeración de sus propiedades. En este caso, un horario va a ser ciertamente algo más que sus segmentos, así que se necesita una clase separada. Algunas de las demás características del horario de presentan a sí mismas de forma natural:

```
indexing
      description: "Horarios de TV para veinticuatro horas"
deferred class HORARIO feature
      segmentos: LIST[SEGMENTO] is
            --los sucesivos segmentos
          deferred
          end
      hora_emision: FECHA is
            --Periodo de veinticuatro horas para este horario.
          deferred
          end
      asigna_hora_emision (t: FECHA) is
         --Asigna este horario para ser retransmitido en el instante t.
          require
            t.es_futuro
          deferred
          ensure
            hora_emision = t
          end
      print is
```

```
            --Imprimir versión en papel del horario
        deferred
        end
end
```

Obsérvese el uso de cuerpos diferidos. Esto es adecuado ya que por naturaleza un documento de análisis es independiente de la implementación e incluso independiente del diseño; al no poseer un cuerpo, las características diferidas son la herramienta correcta. Por supuesto, podría uno olvidarse de escribir la especificación **deferred** y utilizar en su lugar otros formulismos o notaciones. Pero hay dos argumentos que justifican la utilización de la notación completa:

- Al escribir textos que se ajusten a la sintaxis de la notación del software, se puede hacer uso de todas las herramientas de un entorno de desarrollo de software. En particular, el mecanismo de compilación hará también las veces de una preciosa herramienta CASE, que aplicará las reglas de tipos y otras restricciones de validez para comprobar la consistencia de nuestras especificaciones, y también para detectar contradicciones y ambigüedades; además las capacidades de navegación y documentación de un buen entorno O-O serán tan útiles para el análisis como lo son para el diseño y la implementación.
- El uso de la notación del software significa también, que si se tomará la decisión de seguir adelante con el diseño e implementación de un sistema software, será posible seguir una ruta de transición suave, el trabajo consistiría entonces en añadir nuevas clases, versiones efectivas de las clases diferidas y nuevas características. Esto posibilita el apoyo a la *carencia de discontinuidades* del enfoque.

La clase presupone una consulta booleana *es_futuro* aplicable a objetos del tipo *FECHA*; solamente permite establecer la fecha de emisión para fechas futuras. Obsérvese esta primera utilización de una precondición y de una postcondición para expresar propiedades semánticas de un sistema durante el análisis.

Segmentos

En lugar de seguir adelante para mejorar y refinar *HORARIO*, pasemos en esta fase al concepto de *SEGMENTO*. Se puede comenzar con las siguientes características:

```
indexing
      description: "Segmentos individuales de un horario de
                    transmisiones"
deferred class SEGMENTO feature
      horario: HORARIO is deferred end
            --Horario al que pertenece el segmento
      indice: INTEGER is deferred end
            --Posición del segmento dentro de su horario
      hora_inicio, hora_final: INTEGER is deferred end
            --Principio y fin de la transmisión
      siguiente: SEGMENTO is deferred end
            --Segmento que hay que emitir a continuación, si lo hubiere
      patrocinador: COMPANÍA is deferred end
            --Patrocinador principal del segmento
```

```
        calificación: INTEGER is deferred end
              --Calificación del segmento (para menores, adultos, etc.)
        ... Órdenes tales como cambiar_siguiente, asigna_patrocinador,
        asigna_calificación son omitidas ...
        Duracion_minima: INTEGER is 30
              --Duración mínima de los segmentos, en segundos
        Intervalo_maximo: INTEGER is 2
              --Tiempo máximo entre dos segmentos sucesivos, en segundos
invariant
        lista_entrada: (1<=indice) and (indice <= horario.segmentos.count)
        horario_entrada: horario.segmentos.item (indice) = Current
        siguiente_en_lista: (siguiente /= Void) implies
            (horario.segmentos.item  (indice + 1) = siguiente)
        no_siguiente_si_i_ultimo: (siguiente = Void) = (indice =
              horario.segmentos.count)
        calificacion_no_negativa: calificacion >= 0
        tiempos_positivos: (hora_inicio > 0) and (hora_final > 0)
        duracion_suficiente: hora_final - hora_inicio >= Duracion_minima
        intervalo_correcto: (siguiente.hora_inicio) - hora final <=
              Intervalo_maximo
end
```

Cada segmento "conoce" el horario del cual forma parte, lo cual se expresa mediante la consulta *horario*, y su posición dentro de ese horario, que queda expresada mediante *indice*. Posee una *hora_inicio* y una *hora_final*; también se podría añadir una consulta llamada *duracion*, con una cláusula del invariante que exprese la relación en las dos anteriores. La redundancia es aceptable en un análisis de sistemas siempre y cuando las características redundantes expresen conceptos de interés para los usuarios o para los desarrolladores, y siempre y cuando las relaciones entre elementos redundantes se enuncien de forma clara en todo el invariante. En este caso las cláusulas *lista_entrada* y *horario_entrada* del invariante expresan la relación entre el *indice* propio de un segmento y su posición dentro de la lista de segmentos del horario.

Un segmento también conoce el segmento que irá a continuación, *siguiente*. Las cláusulas del invariante expresan una vez más los requisitos de consistencia: la cláusula *siguiente_en_lista* indica que si el segmento se encuentra en la posición *i* el *siguiente* está en la posición *i+1*; la cláusula *no_siguiente_si_i_ultimo*, indica que existe un *siguiente* si y sólo si el segmento no es el último del horario.

Las dos últimas cláusulas del invariante expresan restricciones relativas a las duraciones: *duracion_suficiente* define una duración mínima de 30 segundos para que un fragmento de programa merezca denominarse segmento, e *intervalo_correcto* denota un máximo de dos segundos como el tiempo que mediará entre dos segmentos sucesivos (tiempo en que la pantalla de la TV puede quedar oscurecida).

La especificación de clases ha utilizado dos atajos que casi con seguridad tendrían que eliminarse en la próxima iteración del proceso de análisis. En primer lugar, los tiempos y las duraciones se han expresado como enteros, medidos en segundos,: esto no es suficientemente abstracto, debieran basarse en clases de biblioteca como *DATE*, *TIME* y *DURATION*. En segundo lugar, el concepto *SEGMENTO* abarca dos conceptos separados: un fragmento de programa de TV, que se puede

definir independientemente de su instante de emisión; y la planificación de un cierto programa en un cierto segmento horario. Para separar estos dos conceptos la solución es sencilla; basta con añadir a *SEGMENTO* un atributo

```
contenido: FRAGMENTO_PROGRAMA
```

con una nueva clase *FRAGMENTO_PROGRAMA* que describa el contenido independiente de su planificación. La característica *duracion* debería de aparecer en *FRAGMENTO_PROGRAMA*, y tendría que existir una nueva cláusula del invariante de *SEGMENTO* que indicara

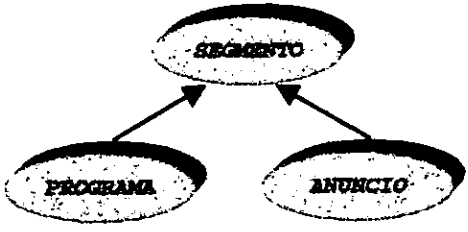
```
contenido.duracion = hora_final - hora_inicio
```

Por brevedad, el resto del ejemplo seguirá tratando el contenido como si formara parte del segmento. Estas discusiones son típicas de lo que sucede durante el proceso de análisis, que se ve favorecido por el método orientado a objetos: se examinan diferentes abstracciones, se analiza si justifican las distintas clases y se pasan características a otras clases si se piensa que no se han asignado correctamente.

Cada segmento posee un patrocinador y una calificación. Aun cuando aquí también podríamos beneficiarnos de una clase separada, *calificacion*, se ha especificado simplemente como un entero, con la convención de que una calificación superior implica más restricciones; 0 significa un segmento que será adecuado para todas las audiencias.

Programas y anuncios

Al investigar el concepto de *SEGMENTO* un poco más, se distinguen dos clases: segmentos de programa y pausas comerciales (segmentos para anuncios). Esto sugiere inmediatamente utilizar la herencia:



Este impulso para utilizar la herencia durante el análisis, siempre es sospechoso; se debe tener precaución frente a las apariciones de taxonomía, que impulsan a crear clases ilegítimas en donde bastaría con simples propiedades distintivas. El criterio que sirve de guía se daba en la descripción de herencia: ¿corresponde realmente cada una de las clases propuestas a una abstracción separada, caracterizada por propiedades y características específicas? Aquí la respuesta será sí; no es fácil pensar en características que son propias de los

programas y de los anuncios, según se enumerarán parcialmente más adelante. El uso de la herencia producirá además el beneficio de la apertura de cambios; se puede añadir un heredero como *REPORTAJE-COMERCIAL* posteriormente para describir segmentos de otra clase distinta.

Se propone comenzar *ANUNCIO* de la forma siguiente:

```
indexing
      description. "Segmento de anuncio"
deferred class ANUNCIO inherit
      SEGMENTO
         rename patrocinador as anunciante end
feature
      primario: PROGRAMA is deferred
            --Programa al cual está asociado este anuncio
      indice_primario: INTEGER is deferred
            --índice de primario
      asigna_primario (p: PROGRAMA) is
            --conecta el anuncio a p.
         require
            existe_programa: p /= Void
            mismo_horario: p.horario = horario
            antes: p.hora_inicio <= hora_inicio
         deferred
         ensure
            indice_actualizado: indice_primario = p.indice
            primario_actualizado: primario = p
         end
invariant
      indice_primario_significativo: indice_primario = primario.indice
      primario_antes: primario.hora_inicio <= hora_inicio
      patrocinador_admisible: anunciante.compatible
            (primario.patrocinador)
      calificacion_admisible: calificacion <= primario.calificacion
end
```

Obsérvese el uso del *cambio de nombres*, otro ejemplo de una capacidad de notación que a primera vista podría parecer útil sólo para las clases del nivel de implementación, pero que resulta ser igualmente necesaria para el modelado. Cuando un segmento es un anuncio, es más adecuado hacer alusión a su *patrocinador* en la forma de *anunciante*.

Todo segmento anuncio está asociado a un segmento de programa anterior (no a un anuncio), su *primario* cuyo índice dentro de ese horario es *indice_primario*. Las dos primeras cláusulas del invariante expresan condiciones de consistencia; las dos últimas expresan reglas de compatibilidad:

- Si un cierto programa tiene un patrocinador, todo anunciante que aparezca en ese programa debe de ser admisible para él; no es lógico anunciar Pepsi-Cola durante un programa patrocinado por Coca-Cola. La consulta *compatible* de la clase *COMPAÑÍA* podría aportarse a través de alguna base de datos.
- La calificación de un anuncio debe ser compatible con la de su programa primario: no es lógico anunciar Massacre III en un programa para niños.

El concepto de `primario` necesita un refinamiento. En esta fase del análisis queda claro que realmente sería preciso añadir un nivel: en lugar de hacer que un horario sea una sucesión de segmentos de programa y de anuncios, sería preciso verlo como una sucesión de espectáculos, en donde cada espectáculo (que se describirá mediante una clase *ESPECTACULO*) posee sus propias características, tal como el patrocinador del espectáculo, y una sucesión de segmentos de espectáculo y de anuncios. Esta mejora y refinamiento, que se desarrolla cuando se entiende mejor el problema y se ha aprendido a partir de los primeros intentos, es un componente normal del proceso de análisis.

Reglas de negocio

Ya se ha visto la forma en que las cláusulas del invariante y otras aserciones pueden establecer las restricciones semánticas del dominio de la aplicación, que también se conoce en el ámbito del análisis como *reglas de negocio*: en la clase *HORARIO*, la regla consiste en planificar un segmento únicamente en el futuro. Dentro de *SEGMENTO*, la regla consiste en que la interrupción entre dos segmentos no puede sobrepasar una cierta duración preestablecida; en *ANUNCIO*, la regla consiste en que la calificación del anuncio tiene que ser compatible con la del programa que lo contenga.

Ciertamente, una de las contribuciones principales del método consiste en que se pueden utilizar las aserciones (en su forma de cláusulas **`require`**, **`ensure`** e **`invariant`**) para expresar estas reglas junto con la estructura.

Sin embargo, lo siguiente es una advertencia práctica: aun cuando no se llegue a ningún compromiso de implementación, existe un riesgo de excesiva especificación. En las aserciones del texto del análisis, solamente se deben incluir aquellas reglas de negocio que tengan un alto grado de certeza y perdurabilidad. Si alguna de estas reglas está sujeta a cambios, debe utilizarse la abstracción para expresar lo que se necesite, pero dejando espacio para las modificaciones. Por ejemplo, las reglas de compatibilidad entre patrocinador y anunciante pueden cambiar; consiguientemente, el invariante de *ANUNCIO* se mantiene alejado de la excesiva especificación postulando simplemente un consulta booleana `compatible` que existiría en la clase *COMPAÑÍA*.

Una de las grandes ventajas del análisis es que uno decide lo que dice y lo que no dice. Se enuncia lo que se conoce – si no se especifica nada, la especificación no tendrá mucho interés – pero nada más. Esto mismo se buscó en el desarrollo de los tipos abstractos de datos (TAD): se desea la verdad, toda la verdad relevante, y nada más que la verdad.

4.4. El proceso de construcción del software

Fundamental entre los problemas metodológicos de la tecnología de objetos resulta la forma en que ésta afecta al desarrollo del software en toda su extensión. Como parte final de este trabajo se examinarán las consecuencias de los principios orientados a objetos sobre la organización de proyectos y su descomposición en fases.

Se presentarán ideas fundamentales para entender como la tecnología de objetos requiere de un nuevo modelo de procesos, que apoye un desarrollo de software, sin discontinuidades y reversible. Para lograr lo anterior se propone el uso de

unidades organizativas básicas denominadas **clusters**(*); de los principios de la ingeniería concurrente que dan lugar al modelo de clusters del ciclo de vida del software; los pasos y tareas de ese modelo, el papel de la **generalización** para la reutilización, y los principios de **ausencia de discontinuidades** y de **reversibilidad**.

4.4.1. Clusters

La estructura modular del método orientado a objetos es la clase. A efectos organizativos, lo normal será necesitar agrupar las clases por colecciones, denominadas clusters. Un cluster es un grupo de clases relacionadas o, recursivamente, de clusters relacionados(**).

Entre los clusters típicos se podrían incluir el cluster *analizador* para revisar la entrada de texto del usuario, un cluster *gráfico* para las manipulaciones de interfaces gráficas, y un cluster de *comunicaciones*. Un cluster básico podría tener de cinco a cuarenta clases, cuando se alcanzan aproximadamente veinte clases, se tiene que pensar en descomponerlo en subclusters. El cluster es también la unidad natural para el desarrollo por parte de un único desarrollador: cada cluster debería ser gestionado por una sola persona, y una persona debería ser capaz de comprender todo el cluster – sin embargo, en un desarrollo de gran tamaño nadie puede comprender todo un sistema completo, ni siquiera un subsistema principal.

Los clusters no son una estructura de lenguaje, aún cuando sea necesario que aparezcan en los archivos de control que se utilizan para ensamblar sistemas a partir de componentes. Son una herramienta de gestión. La responsabilidad de hallar los clusters recaerá sobre el jefe de proyecto; es una tarea menos complicada que la de hallar clases; la agrupación de clases se basa fundamentalmente en el sentido común y en la experiencia del jefe de proyecto. En realidad este extremo merece cierto hincapié, por que en algunos casos no se comprende correctamente: el trabajo realmente difícil, que puede llevar a un proyecto a una vida llena de éxitos o arruinarlo, y para el cual no se puede hablar de soluciones correctas o incorrectas, consiste en identificar las clases (las abstracciones de datos correctas); el agrupamiento de estas clases en clusters es una cuestión organizativa, para la cual hay muchas soluciones posibles, que dependen de los recursos que estén disponibles y de la experiencia de los diferentes miembros del equipo. Una decisión de agrupamiento que no sea óptima puede crear problemas y hacer más lento el desarrollo, pero no será por sí misma causa del fracaso del proyecto.

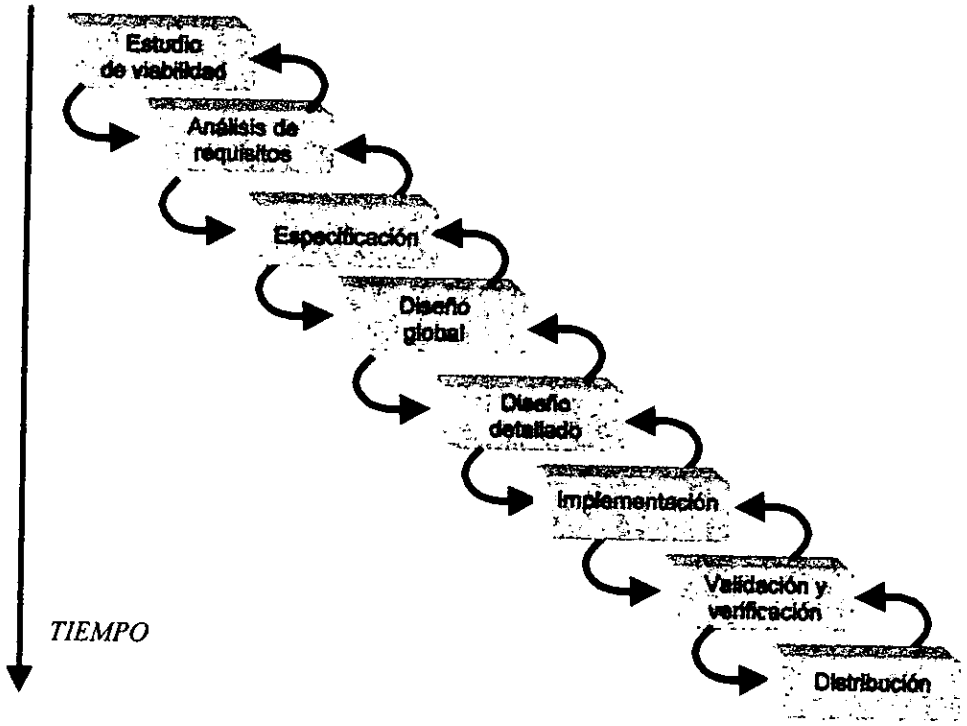
(*) El término *cluster* usado aquí no tiene que ver con el contexto de un determinado sistema operativo. El término es más bien un concepto empleado para designar una forma de organización para las clases dentro de la tecnología de objetos (intuitivamente es fácil comprenderlo si se usa la traducción literal de la palabra: *inglés*/español; *cluster*/grupo).

(**) Los dos casos son exclusivos: por sencillez y para mayor simplicidad de administración, se sugiere que un cluster que contenga subclusters no debería poseer clases propias. Por tanto un cluster será o bien un cluster básico, o bien un supercluster, formado por otros clusters.

4.4.2. Ingeniería concurrente

Una de las consecuencias de la división en clusters es que se puede evitar la desventaja asociada a la naturaleza todo-nada de los modelos tradicionales de ciclo de vida del software. El conocido enfoque de "cascada"[1], que se presentó en 1970, era una reacción contra el "codificarlo ahora y arreglarlo después", enfoque que pertenece ya a la prehistoria del software. Sin embargo tenía el mérito de separar los distintos asuntos, y de definir las tareas principales de la ingeniería del software, y también el mérito de hacer hincapié en la importancia de unas especificaciones iniciales y de unas tareas de diseño.

Pero el modelo de cascada también adolece (entre otras deficiencias) de la rigidez de su enfoque. Si se toma literalmente, significará que ningún diseño podría seguir adelante mientras la especificación no esté completa en su totalidad, o que no podrá haber implementación mientras no esté completo todo el diseño. Ésta es una receta segura para el desastre: *un grano de arena en la máquina, y el proceso completo se detiene.*



El modelo de cascada

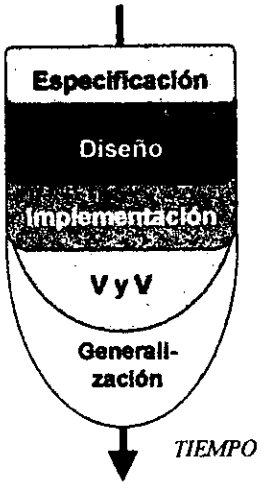
[1] PRESSMAN, R. S., op.cit., págs. 22-24.

Existen varias propuestas tales como el modelo en espiral que se manejó en el capítulo I que intentan reducir este riesgo ofreciendo un enfoque más iterativo, sin embargo todas ellas mantienen el enfoque de un solo hilo de la cascada, que escasamente reflejan la naturaleza del desarrollo de software actual, especialmente, para grandes equipos "virtuales" que pueden estar distribuidos en muchos centros, comunicándose a través de Internet y otros mecanismos de "ubicación electrónica".

El desarrollo orientado a objetos que tenga éxito tendrá la necesidad de apoyar un cierto esquema de **ingeniería concurrente**[2], que ofrezca una dosis de descentralización y de flexibilidad, sin perder los beneficios de ordenación heredados del modelo de cascada y de la espiral de ciclo de vida del software. En particular será preciso mantener un componente secuencial, con actividades bien definidas. El desarrollo orientado a objetos no significa que se puedan o se deban eliminar las buenas prácticas de programación. En todo caso, la mayor potencia del método requiere que seamos *más* organizados que antes.

Mediante una división en clusters se puede lograr el equilibrio correcto entre secuencialidad e ingeniería concurrente. Se tendrá un proceso secuencial, pero estará sometido a unos ajustes retroactivos (éste es el concepto de reversibilidad), y se aplica a los **clusters** y no al sistema completo.

El miniciclo de vida que gobierna el desarrollo de un cluster se puede representar en la forma siguiente:



Ciclo de vida individual de un cluster

La forma de las representaciones de esta actividad sugiere la naturaleza carente de discontinuidades del desarrollo. En lugar de aparecer como pasos separados tal como el modelo de cascada, se aprecia un modelo de *acreción* – imagínese que la figura representase una estalactita – en el cual cada paso abarca al anterior y le añade su propia contribución.

[2] MEYER, B.: *Sequential and Concurrent Object-Oriented Programming*. Angkor/SOL, París, 1990.

4.4.3. Pasos y tareas

Los pasos propuestos en el miniciclo de vida de cada uno de los clusters son:

- Especificación. Se identifican las clases (abstracciones de datos) del cluster y sus características y restricciones principales.
- Diseño. Se define la arquitectura de las clases y sus relaciones.
- Implementación. Se finalizan las clases, añadiendo todos los detalles.
- Verificación y validación. Se comprueba que las clases del cluster funcionen satisfactoriamente (mediante un examen estático, comprobaciones y otras técnicas).
- Generalización. Se preparan para la reutilización.

Dado el elevado nivel de abstracción del método, la distinción entre diseño e implementación no queda siempre claramente definida. Por tanto una variante del modelo fusiona estos dos pasos en uno, "diseño-implementación".

Sigue existiendo la necesidad de dos fases que abarquen a todo el sistema y que sean independientes de los clusters. En primer lugar, como con cualquier otro enfoque, será preciso realizar un **estudio de viabilidad**, que dará lugar a una decisión positiva o negativa. Entonces, el proyecto necesita descomponerse en clusters; esto es, según se ha indicado, una responsabilidad del jefe de proyecto, que por supuesto podrá basarse en los puntos de vista de otros miembros experimentados del equipo.

4.4.4. El modelo de clusters del ciclo de vida del software[3]

El esquema general de desarrollo, conocido con el nombre de Modelo de Clusters, se puede ver en la figura de la siguiente página.

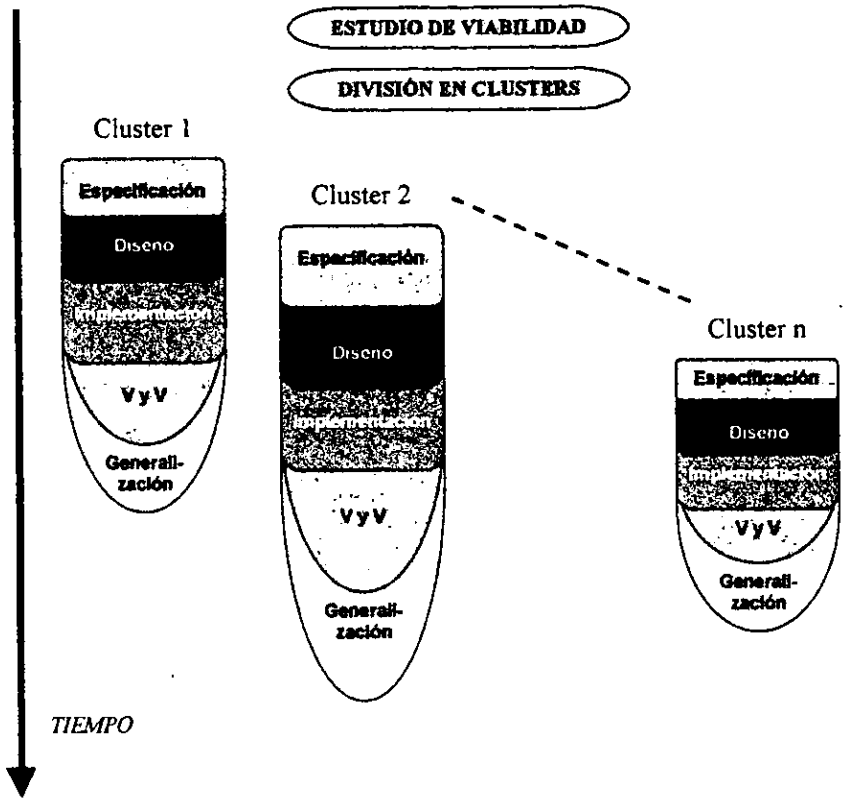
El eje vertical representa un componente secuencial del proceso: un paso que aparezca más abajo que otro se ejecutará después de aquel. La dirección horizontal refleja la ingeniería concurrente: las tareas del mismo nivel pueden desarrollarse en paralelo.

Habrá varios clusters, y varios pasos dentro de cada cluster, que vayan avanzando a su propio ritmo dependiendo de la dificultan de la tarea. El jefe de proyecto tiene la responsabilidad de decir el momento en el que debe comenzar un nuevo cluster o una nueva tarea.

El resultado es otorgar al jefe de proyecto la combinación correcta de orden y flexibilidad. Se consigue el orden porque la definición de tareas de clusters proporciona un entorno de control y unos puntos de control con respecto a los cuales se pueden estimar los progresos y los retrasos (que es uno de los aspectos más difíciles de la gestión de proyectos); además se proporciona flexibilidad porque se pueden amortiguar los retardos inesperados, o bien se pueden aprovechar los casos en los que el progreso sea inesperadamente rápido, iniciando las actividades más pronto o más tarde.

---

[3] MEYER, B.: *Object Success: A Manager's Guide to Object Technology*. Prentice Hall Object-Oriented Series. E.U.A., 1995.



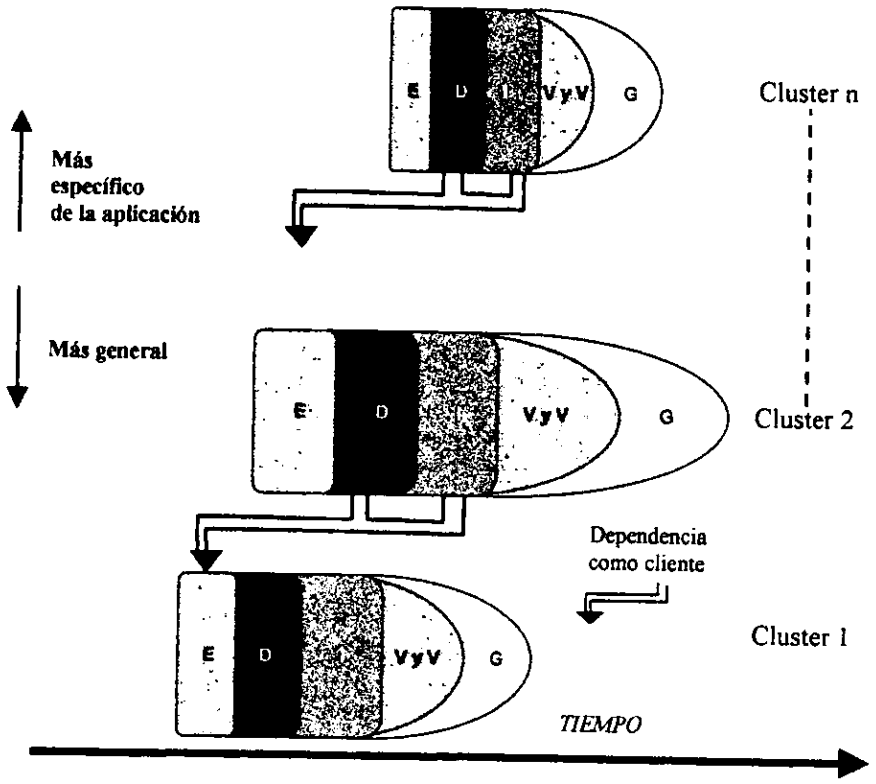
El modelo de clusters del ciclo de vida del software

El jefe de proyecto controla también el grado de concurrencia de la ingeniería: para un equipo pequeño, o bien en las fases iniciales de un proyecto difícil, es posible que exista un pequeño número de clusters paralelos, incluso uno solo; para un equipo más grande, o bien una vez que las cuestiones existenciales básicas parezcan estar controladas, se puede comenzar a tratar varios clusters a la vez.

Mejorando los enfoques tradicionales, el modelo de clusters capacita a los jefes de proyecto para realizar su tarea en toda su extensión, ejerciendo su poder de decisión para dedicar los recursos a aquellas tareas en que más se necesiten.

Para evitar la divergencia, es preciso reconciliar regularmente el estado actual del desarrollo de los diferentes clusters. Ésta es la tarea de **integración**, que conviene realizar a intervalos preestablecidos, por ejemplo una vez a la semana. Es responsabilidad del jefe de proyecto, y asegura que en todas las fases posteriores al arranque exista una **demostración actualizada**, que no estará necesariamente actualizada para todos los aspectos del sistema, pero que sí estará preparada para mostrarse a cualquier persona – clientes, administradores, etc. – que necesiten asegurarse del progreso del proyecto. Esto sirve también para eliminar cualquier

inconsistencia entre clusters antes de que tenga la oportunidad de dar lugar a daños, y asegura a los miembros del proyecto que todas las piezas encajan entre sí y que el futuro sistema va avanzando.



Los clusters de un proyecto vistos como un conjunto de capas de abstracción

Lo que hace posible la forma del modelo de clusters de ingeniería concurrente es el conjunto de propiedades de ocultación de información que aporta el método orientado a objetos. Los clusters pueden depender entre sí; por ejemplo, un cluster de interfaz gráfica quizás necesite, para la visualización remota, clases del cluster de comunicaciones. Gracias a la abstracción de datos es posible que un cluster siga adelante aun cuando los clusters de los cuales depende todavía no estén acabados; basta que la fase de *especificación* de las clases necesarias esté completa, de tal modo que uno pueda seguir adelante basándose en su interfaz oficial, que se aportará en forma sintetizada o como una versión diferida. Este aspecto del modelo es quizás más fácil de imaginar si se pone atención en la figura anterior, donde se hace hincapié en las capas de software que corresponden a los diferentes clusters, situando los clusters más generales en la parte inferior, y los más relacionados con la aplicación en la parte superior. El diseño e implementación de cada uno de los clusters depende solamente de las especificaciones de los clusters que están por debajo de él, y no de su propio diseño e implementación. La figura solamente muestra dependencias del

cluster que aparece inmediatamente por debajo, pero un cluster puede basarse en cualquier otro cluster de nivel inferior.

4.4.5. Generalización

La última tarea de los minicicios de vida de los clusters, la generalización (la G de la figura anterior) no tiene un equivalente en los enfoques tradicionales. Su objetivo es pulir las clases de tal modo que se transformen en componentes de software potencialmente reutilizables.

La inclusión de un paso de generalización sugiere inmediatamente una crítica: en lugar de una adición a posteriori, ¿No deberían ser los problemas de reutilización parte del proceso de software completo? ¿cómo puede uno hacer que un software sea reutilizable a posteriori? Pero esta crítica se hace en un lugar incorrecto.

El punto de vista a priori de reutilización del software (*"para que sea reutilizable, el software tiene que ser diseñado como reutilizable desde el principio"*) y el punto de vista a posteriori (*"el software no será reutilizable en la primera pasada"*) son complementarios y no contradictorios. El éxito de una política de reutilización requiere tanto instaurar una *cultura de reutilización* en las mentes de todos los implicados, como dedicar unos recursos suficientes para mejorar la reutilización de las versiones iniciales de las clases.

Pese a las mejores intenciones, los elementos de software que se producen como parte de un proyecto orientado a objetos no suelen ser totalmente reutilizables. En parte esto se debe a las restricciones que afectan a los proyectos – la presión de los clientes que desean la próxima versión en cuanto sea posible, la presión de la competencia que va publicando sus propios productos, la presión de los accionistas que están deseosos de ver resultados. Vivimos en un mundo apresurado, y en una industria todavía más apresurada. Pero existe una razón intrínseca para no confiar siempre en las promesas de reutilización: mientras no haya alguien que lo haya reutilizado, no se puede estar seguro de que un producto esté totalmente liberado de sus dependencias explícitas y (especialmente) implícitas, del trasfondo original de sus desarrolladores, de su afiliación corporativa, de su contexto técnico, de las prácticas de trabajo, de los recursos de hardware y del entorno de software.

La inclusión de un paso de generalización en el modelo oficial del proceso puede considerarse también como una medida política. Hay muy pocos ejecutivos corporativos que conocedores de los beneficios de la reutilización, se opongan públicamente a ella. Sin embargo para instalar en una compañía una cultura de reutilización se necesita un verdadero compromiso, es decir, que la dirección este dispuesta a reservar algunos recursos, además de tiempo y dinero, que estén destinados a cada proyecto, para la generalización. Esta es una decisión valiente, porque los beneficios pueden no ser inmediatos y quizás los proyectos urgentes tengan pequeños retrasos. Pero es la única forma de garantizar que, finalmente, se dispondrá de componentes reutilizables.

Si la empresa logra comprometerse a dedicar recursos a la generalización, ya se dio un importante paso, pero es justo saber que esto no es suficiente. El éxito en la reutilización proviene de una combinación de esfuerzos a priori y posteriori, tal y como se manifiesta en la siguiente máxima:

CULTURA DE LA REUTILIZACIÓN[4]

Desarrolle todo el software suponiendo que se va a reutilizar.
No confié en que ningún software va a ser reutilizable mientras no lo haya visto reutilizado.

La primera parte implica aplicar los problemas de reutilización a lo largo de todo el desarrollo. La segunda parte implica no dar por sentado el resultado, sino realizar un paso de generalización para eliminar cualquier posible resto de elementos específicos del contexto. La tarea de generalización puede implicar las actividades siguientes:

- **Abstracción**. Introducción de una clase diferida para describir la abstracción pura que subyace a una cierta clase.
- **Factorización**. Reconocimiento de que dos clases, que no estaban relacionadas originalmente, son de hecho variantes de un mismo concepto general, que se puede describir entonces mediante una clase ascendiente común.
- Añadir **aserciones**, especialmente postcondiciones y cláusulas del invariante que reflejen una mayor comprensión de la semántica de la clase y sus características.
- Añadir cláusulas para gestionar las excepciones cuya posibilidad pudiera haberse ignorado inicialmente.
- Añadir **documentación**.

El papel de la generalización es mejorar clases que puedan considerarse lo suficientemente buenas para los propósitos internos – siempre y cuando se utilicen sólo dentro de un sistema particular – pero ya no cuando pasan a formar parte de una biblioteca que está disponible para cualquier autor cliente que las utilice para sus propias necesidades. Unos descuidos que pueden haber sido perdonables en el primer entorno, como una especificación insuficiente o basarse en suposiciones no documentadas, se transforman en errores que detienen todo el proceso. Ésta es la razón por la cual el desarrollo para la reutilización es más difícil que el desarrollo ordinario de aplicaciones: cuando nuestro software está disponible para cualquiera, para trabajar en aplicaciones de cualquier tipo para cualquier plataforma de cualquier lugar del mundo, todo empieza a ser importante. *La reutilización engendra el perfeccionismo; no puede uno dejar como está lo que más o menos funciona.*

4.4.6. Ausencia de discontinuidades y reversibilidad

La naturaleza de tipo "estalactítico" del ciclo de vida de los clusters refleja una de las diferencias más radicales entre el desarrollo O-O y los enfoques anteriores. En lugar de erigir barreras entre los pasos sucesivos del ciclo de vida, la tecnología de objetos bien entendida define un único entorno de análisis, diseño, implementación y mantenimiento. Esto se conoce con el nombre de **desarrollo sin discontinuidades**, que manifiesta la necesidad de un proceso reversible de desarrollo del software.

[4] MEYER, B.: *Object-Oriented Software Construction*. Prentice Hall PTR. 2ª Edición, 1997, pág. 929.

Desarrollo sin discontinuidades

Por supuesto, las distintas tareas van a permanecer. Para estudiar unos ejemplos extremos, no se está haciendo lo mismo cuando se definen propiedades generales del sistema que todavía no se ha construido y cuando se están efectuando las últimas iteraciones de la depuración. Pero la idea de ausencia de discontinuidades consiste en reducir las diferencias en aquellos lugares en los que el enfoque tradicional las acentuaba; reconocer, tras las variaciones técnicas, la unidad fundamental del proceso del software. A lo largo del desarrollo surgen los mismos mecanismos de estructuración, y son aplicables las mismas fórmulas de razonamiento; además se puede utilizar la misma notación.

Los beneficios del enfoque sin discontinuidades son numerosos:

- Se evitan unas transiciones costosas y proclives a error entre pasos, que se ven magnificadas por los cambios de notación, de composición mental, y de personal (analistas, diseñadores, implementadores, etc.). Estos saltos suelen denominarse **desajustes de impedancia** (**impedance mismatches**) por analogía con un circuito formado por elementos eléctricamente incompatibles; las inconsistencias entre análisis y diseño, diseño e implementación, e implementación y evolución se cuentan entre las peores causas de problemas en el desarrollo tradicional del software.
- Al comenzar desde las clases de análisis como base para el resto del desarrollo, se asegura una correspondencia próxima entre la descripción del problema y la solución[5]. Esta **propiedad de correspondencia directa** facilita el diálogo con los clientes y usuarios, y facilita la evolución asegurando que todo el mundo piense en términos de unos mismos conceptos básicos. Esto forma parte del apoyo para la extensibilidad propio del método orientado a objetos.
- La utilización de un único entorno facilita los ajustes retroactivos que acompañarán inevitablemente al progreso normalmente unidireccional del desarrollo del software.

Reversibilidad

El último beneficio citado define una de las principales contribuciones de la tecnología de objetos al ciclo de vida del software – la reversibilidad.

La reversibilidad es la admisión oficial de una característica del desarrollo del software que, aun cuando es inevitable y universal, es uno de los secretos más celosamente guardados de la literatura del software: la influencia de las etapas finales del proceso del software sobre las decisiones que se toman en las fases iniciales.

Todos desearían, por supuesto, que los problemas estuvieran completamente definidos antes de empezar a resolverlos. Ésta es la forma normal de proceder, y en el software esto significa que se finaliza el análisis antes de comprometerse con el diseño; se finaliza el diseño antes de comenzar la implementación, y se finaliza la implementación antes de efectuar la implantación. Ahora bien, que sucedería si durante la implementación un desarrollador se da cuenta repentinamente de que el sistema podría hacer algo mejor, o que tendría que hacer algo completamente distinto.

[5] vid supra, pág. 50.

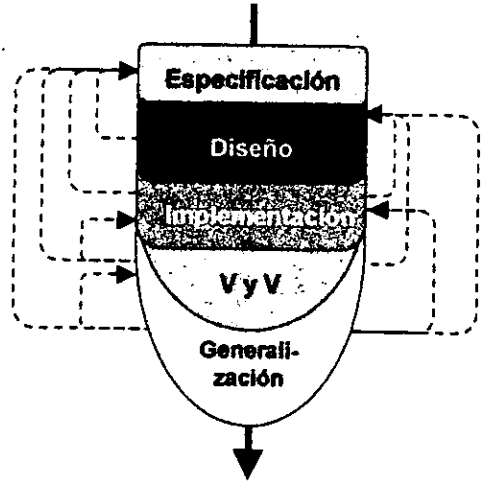
Esta situación es muy frecuente dada la naturaleza cambiante del software y, más aún si se busca crear siempre un software correcto y verdaderamente útil para el usuario final.

Una actitud positiva que pudiesen adoptar los directivos ante esta situación es la de intentar aprovechar esas tardías ideas de especificación, sin atraer la atención de quien quiera que esté a cargo de imponer el plan de calidad de software de la compañía y sus muestras del estilo en cascada en contra de modificar la especificación en el momento de la implementación.

Sin embargo, es posible que ni la mejor voluntad y disposición de un directivo pueda romper la rigidez de un modelo o estilo de trabajo. De ahí que el desarrollo orientado a objetos gane ventaja sobre otros enfoques, ya que reconoce al cambio como un fenómeno intrínseco en el desarrollo del software. Es necesario aceptar que algunas veces se comprenden muchos más aspectos del problema tan sólo en el momento de plantear la solución, y más aún, que la solución afecta al problema y sugiere mejores funcionalidades.

Lo anterior puede sustentarse con el ejemplo del patrón de *hacer y deshacer* donde una técnica de implementación, la "lista histórica" – que alguien entrenado con un enfoque tradicional descartaría como relevante para la tarea de definir la funcionalidad de un sistema – sugería en realidad una forma para proporcionar a los usuarios finales del sistema una interfaz cómoda para deshacer y rehacer órdenes.

La introducción de la reversibilidad sugiere que el impulso general progresivo de los diagramas anteriores del miniciclo de vida de clusters se ve ponderado en la realidad por la constante posibilidad de revisiones y correcciones retroactivas:



El ciclo de vida individual de un cluster reversible

4.4.7. Aportación del método orientado a objetos[6]

El hincapié sobre la ausencia de discontinuidades y sobre la reversibilidad es quizás el componente más potencialmente subversivo de la tecnología de objetos. Afecta a la organización del proyecto, y a la propia naturaleza de la profesión del software; de acuerdo con las tendencias modernas de otras industrias, tiende a eliminar las barreras entre especialidades muy definidas – analistas que solamente tratan con conceptos *etéreos*, diseñadores que solamente se preocupan por la estructura, implementadores que solamente escriben código – y favorece la aparición de una única categoría de *generalistas*: desarrolladores en un amplio sentido del término, personas que son capaces de seguir una parte del proyecto desde el principio hasta el fin.

El enfoque se aparta también del punto de vista dominante en la literatura actual de la ingeniería del software, que trata el análisis y la implementación (con el diseño en algún lugar intermedio) como unas actividades fundamentalmente distintas, susceptibles a distintos métodos, que emplean distintas notaciones y que persiguen objetivos distintos, frecuentemente con la connotación que el análisis y el diseño son lo único que importa, siendo la implementación una tarea inevitable. Este punto de vista tiene justificaciones históricas: desde su infancia en los setenta, la ingeniería del software ha sido un intento de poner un cierto orden en la aleatoria naturaleza de la construcción de programas, enseñando al personal del software pensar antes de disparar. De aquí proviene el hincapié que se hace en las fases iniciales del desarrollo del software, la insistencia acerca de la necesidad de especificar lo que se va a implementar. Esto está justificado, tanto entonces como ahora. Pero algunas de las consecuencias de este esfuerzo esencialmente beneficioso han ido demasiado lejos, creando desajustes de impedancia entre las distintas actividades, y produciendo un modelo estrictamente secuencial aun cuando la calidad del producto y del proceso exigen la ausencia de discontinuidades y la reversibilidad.

Con la tecnología de objetos se pueden eliminar las diferencias innecesarias entre el análisis, el diseño y la implementación – reconociendo sus diferencias necesarias – y se puede rehabilitar la vilipendiada tarea de la implementación. Era natural ya para los pioneros de la ingeniería de software – cuando la programación significaba resolver muchos asuntos dependientes de la máquina y explicar el resultado a la computadora en un lenguaje que pudiera comprender, normalmente de bajo nivel y a veces poco elegante – apartarse de estos aspectos mundanos y en su lugar hacer hincapié en la importancia de estudiar los conceptos abstractos del dominio del problema. Pero se pueden retener estas cualidades de **abstracción** sin perder el enlace con la solución.

El secreto consiste en hacer que los conceptos y notaciones de la programación sean de un nivel suficientemente alto para que puedan servir igualmente bien como herramientas para el modelado. Esto es lo que logra la correcta aplicación de la tecnología de objetos.

---

[6] cfr. MATTISON, R.: *The Object-Oriented Enterprise*. McGraw-Hill, 1997, págs. 198-200.

CONCLUSIONES

La exigencia de construir sistemas software en periodos más reducidos de tiempo y bajo las presiones que imponen los clientes y competidores hacen de la reutilización una opción atractiva para ser implantada dentro de la empresa. Más allá de la velocidad de respuesta en el desarrollo de un producto software, la reutilización introduce fiabilidad, eficiencia y calidad dado que el uso de un "componente" ya probado reduce el riesgo de introducir errores; su alto grado de generalización lo hacen fácilmente adaptable y extensible; su uso repetido le introduce constantes mejoras y los esfuerzos de optimización son más efectivos si se enfocan a uno o varios componentes que a toda la aplicación en general.

Pero evidentemente optar por la reutilización hace necesario contar con una estrategia que permita desarrollar componentes efectivos, este es el papel que desempeña la orientación a objetos.

El método orientado a objetos al formar parte de la ingeniería del software queda necesariamente inmerso en los principios y compromisos que rigen a ésta. De tal suerte que la búsqueda de la calidad es un aspecto primordial. En este renglón queda claro que:

- La calidad del software es una cuestión de equilibrio entre un conjunto de factores diferentes, cuya correcta coordinación da fuerza y estabilidad al producto.

- Los factores externos de la calidad del software, perceptibles para los usuarios y clientes, deben distinguirse de los factores internos, perceptibles por los diseñadores e implementadores.
- Los factores externos son los más importantes y evidentes, pero estos sólo se pueden alcanzar a través de los factores internos.
- El método orientado a objetos ofrece una visión mejorada para asegurar los factores de corrección y robustez relacionados con la seguridad, conocidos en conjunto como fiabilidad; y para aquellos factores que requieren de arquitecturas de software más descentralizadas, en conjunto conocidos como modularidad.

El punto anterior aborda un concepto que tomo especial relevancia para los fines de este trabajo: la modularidad. Del estudio de este concepto se desprenden una serie de puntos concluyentes:

- La elección de una estructura adecuada de módulo es la clave para alcanzar los objetivos de reutilización y extensibilidad.
- Los módulos sirven tanto para la descomposición del software (la visión descendente) como para la composición de software (la visión ascendente).
- Los conceptos modulares se aplican a la especificación y al diseño tanto como a la implementación.
- Una definición amplia de modularidad debe combinar varias perspectivas; los diferentes requisitos pueden parecer incompatibles unos con otros, como la descomposición (que promueve los métodos descendentes) y la composición (que favorece el enfoque ascendente) pero finalmente el desarrollador debe tener la capacidad de conciliar estos aspectos para alcanzar un equilibrio.
- El control del volumen y la forma de la comunicación entre módulos es un paso fundamental para producir una buena arquitectura modular.
- Una dirección efectiva de los proyectos requiere tener soporte para módulos que sean a la vez abiertos y cerrados. Es decir, módulos que puedan ser perfectamente válidos para ser utilizados, a través de su interfaz, por sus módulos clientes y que a su vez pueda ser sometidos a extensiones.

Dentro del rubro de la modularidad debe tenerse presente que la integridad a largo plazo de las estructuras de los sistemas modulares requiere de:

- La ocultación de información, que exige una rigurosa separación entre interfaz e implementación.
- El acceso uniforme, que libera a los clientes de las representaciones internas escogidas por los proveedores.
- El principio de elección única nos lleva a limitar la diseminación del conocimiento exhaustivo de las variantes de un cierto concepto.
- La auto-documentación, que hace que la información relativa a los módulos sea incluida en los mismos y no corra el riesgo de perderse, como sucede cuando dicha información es almacenada en documentos separados.

Todo lo anterior nos lleva a entender que la estructura modular que se busca va más allá de limitar una sección de código o un agrupar una serie de funciones, es algo

mucho más dinámico y completo. La respuesta a estas exigencias llega en la forma de los tipos abstractos de datos (TAD), que para fines del software se traducen en las estructuras modulares básicas del método orientado a objetos: las clases.

La introducción de los tipos abstractos de datos como antecedente de las clases es definitiva para que estas tomen la forma requerida por el método y sus mecanismos, de hecho los TAD sobrepasan la función de establecer las bases teóricas del método, aportando aspectos que en la práctica son determinantes, tales como:

- La teoría de los tipos abstractos de datos reconcilia la necesidad de precisión y completitud en las especificaciones con el deseo de evitar la especificación excesiva.
- La especificación de un tipo abstracto de dato es una descripción matemática formal más que un texto de software. Pero tiene la virtud de ser aplicativa, es decir, libre de cambios.
- Un tipo abstracto de dato puede ser genérico y estar definido por funciones, axiomas y precondiciones. Los axiomas y precondiciones expresan la semántica de un tipo y son esenciales para una descripción completa y no ambigua.
- Dado que un sistema orientado a objetos es una colección de clases. Cada clase está basada en un tipo abstracto de dato y proporciona una implementación parcial o total del TAD.
- Los tipos abstractos de datos son descripciones implícitas en lugar de explícitas. Lo implícito, que también significa abierto, se aplica al método orientado a objetos en su totalidad.

Partiendo del molde establecido por los tipos abstractos de datos, las clases quedan configuradas como descripciones estáticas de las propiedades comunes de un conjunto de objetos que serán creados en tiempo de ejecución. El capturar estas propiedades comunes de los objetos es un aspecto clave para la búsqueda de módulos de software reutilizables.

Para tener un verdadero enfoque hacia la reutilización debe comenzarse por reconocer que el desarrollo de software es una actividad muy repetitiva que implica el uso frecuente de patrones comunes y que además existe una variación considerable en la forma en que se utilizan y combinan esos patrones, lo cual frustra el intento simplista de trabajar con componentes prefabricados.

Las clases creadas a partir de la aplicación del método orientado a objetos proporcionan características que las nociones de módulos comúnmente aceptadas no poseen:

- La clase sirve a la vez como módulo y como tipo. La originalidad y potencia del modelo orientado a objetos proviene en parte de la combinación de estos dos conceptos.
- Las clases proporcionan una descripción suficientemente abstracta de los objetos, debido a que se respetan las especificaciones obtenidas a partir de los TAD.
- Las clases contienen tanto los datos como las funciones que operan sobre ellos, lo que permite que estas sean identificadas por sus características, incluyendo atributos (que representan campos de las instancias de la clase) y rutinas (que representan cómputos relativos a estas instancias).

- La posibilidad de contar con clases diferidas permite que la implementación tenga un desarrollo gradual, añadiendo más características y haciendo efectivas otras hasta que finalmente se consiga una clase efectiva.
- La capacidad de las clases de crecer, relacionarse con otras clases y generar otras mas, proporciona las facilidades necesarias para satisfacer los requisitos de extensibilidad y reutilización.

En este orden de ideas se llega al aspecto más importante de este trabajo: si finalmente se cuenta con una estructura modular completa, correcta y flexible; como es la clase, obtenida a partir del método de orientado a objetos y con expectativas reales de ser reutilizable, se deben proporcionar los mecanismos que permitan aprovechar y aumentar la potencia de la clase para constituirse en el elemento fundamental para la construcción de sistemas. De acuerdo con esto se obtuvieron una serie de puntos concluyentes acerca de las técnicas orientadas a objetos:

- La genericidad permite que las clases puedan tener parámetros genéricos formales que representan tipos, lo que coadyuva a la adaptabilidad.
- Con la herencia se pueden definir nuevas clases por extensión, especialización y combinación de clases previamente definidas.
- La herencia es una técnica clave tanto para la reutilización como para la extensibilidad.
- Desde el punto de vista del módulo, una clase heredera extiende los servicios de sus padres. Esto en particular sirve a la reutilización.
- Desde el punto de vista de tipos, la relación entre una clase heredera y un padre de la clase original es la relación *es*. Esto sirve tanto para la reutilización como para la extensibilidad.
- Una utilización fructífera de la herencia requiere la redefinición (la posibilidad de una clase de suplantar la implementación de algunas de las características de sus antecesores propios), de polimorfismo (la capacidad de una referencia para poder ser conectada en ejecución con instancias de diferentes clases), de ligadura dinámica (la selección dinámica de la variante apropiada de una característica redefinida) y la consistencia de tipos (el requisito de que una entidad sólo puede estar conectada a instancias de los tipos descendientes).
- Las técnicas de herencia, en especial la ligadura dinámica, permiten arquitecturas de software altamente descentralizadas en las que cada variante de una operación se declara dentro del módulo que describe la correspondiente variante de la estructura de datos.

Lo anterior pone de manifiesto la capacidad de soporte de las técnicas orientadas a objetos para con las clases lo que permite conjuntar una metodología completa, con el valor agregado de facilitar un enfoque dirigido hacia la reutilización de estas clases, que finalmente se constituirán en las únicas formas de descomposición y composición del software.

Como se describió en su momento, la reutilización en la práctica plantea problemas económicos, psicológicos y organizativos. Sin embargo, más importantes son aún los problemas técnicos subyacentes: las nociones de módulos comúnmente usadas no son adecuadas para admitir una reutilización seria.

La mayor dificultad de la reutilización es la necesidad de combinar la reutilización con adaptación. El dilema de "reutilizar o rehacer" no es admisible: una buena solución debe permitir retener algunos aspectos de la reutilización de un módulo y adaptar otros.

Los enfoques simples como los de reutilización de personal, de diseño, de código fuente y de bibliotecas de subrutinas han alcanzado un cierto éxito en contextos específicos, pero todos quedan cortos a la hora de proporcionar todo el potencial de beneficios de la reutilización.

La reutilización es un factor de calidad como también un factor de cambio. Debe ser atendido en proporción a su importancia. Este trabajo da a conocer una propuesta metodológica para encarar tanto las exigencias propias de la reutilización como la del resto de los factores de calidad del software. Siendo congruentes con las dimensiones de este reto, es necesario cambiar y en el mejor de los casos complementar las formas actuales de concebir y construir el software. Se demostró como una técnica vigente y con amplias perspectivas de consolidación en la industria del software como es la orientación a objetos cuenta con los suficientes recursos para alcanzar este fin.

Si el software es uno de los recursos más manejables y un candidato ideal para ser llevado a un alto grado de optimización, la inversión en educación y herramientas para su correcto desarrollo queda totalmente justificada. Uno de los roles del ingeniero es precisamente promover la optimización, servir de motor para impulsar la innovación y consecuentemente la evolución.

Consciente de esta misión, comienzo a asumir mi papel como ingeniero al proponer, evaluar y aplicar las técnicas de computación existentes de manera que resulten rentables y fáciles de usar.

Concluiré diciendo que hay tanto valor y aporte en una propuesta sustentada como lo hay en cualquier otro tipo de proyecto. Aplicar los conocimientos existentes, derivados incluso de temas fundamentales, como lo haría un ingeniero civil o electricista cuando aplica la física o las matemáticas, es ya en sí un aporte. Porque finalmente todo lo que se hace está orientado a la consecución de un beneficio práctico.

BIBLIOGRAFÍA

BAUER, F. L.
Software Engineering, Information Processing, 71.
North Holland Publishing Co., Amsterdam, Holanda, 1972.

BISHOP, PETER.
Conceptos de Informática.
Anaya Multimedia, 1ª Edición, Madrid, España, 1989.

BOEHM, B. W.
Software Engineering.
IEEE Transactions on Computers, C-25, num. 12, Diciembre, 1976.

GAMMA, E.; HELM, R.; JOHNSON, R. y VLISSIDES, J.
Design Patterns: Elements of Reusable Object-Oriented Software.
Addison-Wesley, Reading, Massachusetts, E.U.A., 1995.

IEEE.
Standards Collection: Software Engineering.
IEEE Standard 610.12-1990, IEEE, 1993.

LIENTZ, B. P., SWANSON, E. B.
"Software Maintenance Management: a Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations".
Addison-Wesley Publishing, Massachusetts, E.U.A., 1980.

MATTISON, R.
The Object-Oriented Enterprise.
McGraw-Hill, International Edition, Philadelphia, U.S.A., 1997.

MEYER, BERTRAND.
Construcción de Software Orientado a Objetos.
Prentice-Hall, 2ª Edición, Madrid, España, 1999.

MEYER, BERTRAND.
Object-Oriented Software Construction.
Prentice Hall PTR, 2ª Edición, 1997.

MEYER, BERTRAND.
Object Success: A Manager's Guide to Object Technology, its Impact on the Corporation, and its Use for Reengineering the Software Process.
Prentice Hall Object Oriented Series, E.U.A., 1995.

MEYER, BERTRAND.
Sequential and Concurrent Object-Oriented Programming.
Angkor/SOL, París, Francia, 1990.

OSMOND, R. F.
Essential of Successful O-O Project Management: Designing High Performance Projects.
TOOLS USA, E.U.A., 1995.

PRESSMAN, R. S.
Ingeniería del Software: Un enfoque práctico.
McGraw-Hill Interamericana, 4ª Edición, Madrid, España, 1998.

SOMMERVILLE, IAN.
Ingeniería de Software.
Addison-Wesley Iberoamericana,2ª Edición, Wilmington, Delaware, E.U.A., 1998.

YOURDON, E.
Software Reuse: "Application Development Strategies".
vol. VI, nº 12, Diciembre, 1994.

ZELKOVITZ, M. V., SHAW, A. C. y GANNON, J. D.
Principles of Software Engineering and Design.
Prentice-Hall, Englewoods Clif, E.U.A., 1979.

cuántos más obstáculos hay en el camino
más legítima es la victoria.