UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

U.A.C.P.yP. DEL C.C.H.

I.I.M.A.S.

"CROO: SISTEMA DE BUSQUEDA Y CLASIFICACION DE COMPONENTES REUSABLES ORIENTADOS A OBJETOS"

T E S I S

QUE PARA OBTENER EL GRADO DE:

MAESTRO EN CIENCIAS DE LA COMPUTACION

P R E S E N T A :

MIGUEL VALDERRAMA WEINMANN

ASESOR: DRA. HANNA OKTABA

MEXICO, D. F. 1997

A mi hijo
Alejandro Tonatiuh

CONTENIDO

ILUSTRACIONES

INTRODUCCION

La motivación inicial de esta investigación nació de la inquietud de explorar la relación que existe entre el modelo de objetos y el reuso de software. Como objetivo particular se abordó el problema de clasificación de componentes reutilizables OO.

La necesidad de contextualizar y delimitar el ámbito del estudio nos condujo a precisar conceptos de reuso de software. En ese momento nos percatamos de la falta de organización del conocimiento sobre el tema y nos dimos a la tarea de estructurarlo. Es así como surge el tercer objetivo de la investigación que finalmente genera uno de los mayores aportes de este trabajo.

De esta forma, el texto consta de tres capítulos, cada uno abordando uno de los objetivos mencionados. Comenzamos con la estructuración del conocimiento acerca del reuso de software. Definimos una gran cantidad de conceptos y estructuramos múltiples categorías para organizarlos dentro del contexto del proceso global de desarrollo de software.

En el capítulo 2 abordamos la relación que tiene el modelo orientado a objetos con el reuso a través del estudio de los conceptos y mecanismos de programación que abarca. El marco de referencia para este desarrollo tiene su centro en la interconección de componentes y está plasmado en un modelo de composición que unifica la exposición. Concretamente, se plantean una serie de problemas para un modelo de composición en relación al reuso y analizamos la influencia que tienen los conceptos y mecanismos del paradigma OO en la solución de estos problemas. Se ha tomado el paradigma procedural como punto de partida para establecer comparaciones, con la finalidad de ilustrar desde una perspectiva evolutiva los aportes que hace el modelo de objetos a la reusabilidad.

Finalmente, estudiamos los principios de organización de componentes en depósitos, revisamos diferentes métodos de clasificación y seleccionamos al esquema de facetas para clasificar colecciones de clases. Hicimos una adaptación de este esquema basados en las características generales de este tipo de componentes y definimos relaciones y medidas de ponderación que permiten definir estrategias de clasificación y búsqueda. De particular importancia, se presenta la posibilidad de encontrar componentes similares al buscado y determinar una medida de parecido entre ambos componentes.

Como resultado lateral de la investigación se presenta un esquema de facetas para el dominio de estructura de datos. El esquema contiene y organiza la mayoría de las características que puede presentar una estructura contenedora, por lo que creemos que se puede utilizar para propósitos de enseñanza.

AGRADECIMIENTOS

Quiero agradecer a mi tutora y amiga, la Dra. Hanna Oktaba, el apoyo académico y humano que ofrece a todos los que están a su alrededor y realizan un esfuerzo de aprendizaje. Desde mi llegada a esta maestría asistí a varios de sus cursos y dirigió mi trabajo de tesis. Sin embargo, su guía en mi formación académica no es lo que más le agradezco, es realmente admirable la dedicación y cariño con que realiza su trabajo y eso tiene una fuerte influencia en la formación personal de todos los que apreciamos su esfuerzo. Las escuelas de invierno, las escuelas de programación, las relaciones académicas internacionales, la inclusión como posgrado de excelencia ante el CONACYT, la obtención y administración de recursos, la promoción de proyectos de investigación y muchas otras cosas se logran gracias al amor que le imprime a este posgrado.

Un factor determinante para la conclusión de este trabajo fue el apoyo incondicional de dos grandes amigos: Sergio Araya y Andrés Vásquez, que me impulsaron en todo momento y propiciaron las condiciones para que llegara a buen término. Este es el tipo de gestos que sellan amistades.

No puedo dejar pasar la oportunidad para agradecer a Juanita, Lulú, Violeta, Gaby y Alfredo por el apoyo constante, día a día que me proporcionaron en la maestría. Muchos dirán que es su trabajo apoyar a los alumnos, pero la disposición y cariño con que nos tratan va más allá de una mera relación laboral.

El personal de la biblioteca del IIMAS me facilitó enormente la recopilación de información. Debo reconocer que les proporcioné muchos dolores de cabeza por peticiones de material que no tenían, préstamos interbibliotecarios y devolución tardía de materiales. En todo momento encontré la mejor disposición y les estoy agradecido por ello.

Quiero hacer una mención especial a la institución venezolana que me otorgó una beca-crédito para poder realizar estos estudios: FUNDAYACUCHO. Sin su apoyo económico nada habría sido posible, les estoy realmente agradecido.

El mayor de mis agradecimientos es para los amores de mi vida. A mi esposa (Mary) por haber soportado pacientemente mis estados de tensión mientras terminaba la tesis y los escasos momentos de diversión que hemos disfrutado últimamente. A mi querido hijo Alejandro que por ser tan chiquito no se dio cuenta de lo que pasaba y que todos los días me otorga generosamente alegría y ganas de vivir. Esta deuda es la primera que pienso saldar.

Gracias a todos.

Miguel Valderrama W.

Reuso de Software

La concepción del reuso de software ha cambiado considerablemente en los últimos años. Lo más notable es que ha evolucionado a una concepción integral que abarca todas las actividades involucradas en el ciclo de vida de los sistemas.

A pesar de estos avances, frecuentemente se asocia y se confunde el reuso de software con el reuso de código, indicio inequívoco de la poca difusión de los logros alcanzados. Creemos que esta situación se debe en gran parte a la poca estructuración y organización del conocimiento que se ha generado. Pretendemos aquí estructurar en áreas y jerarquías los aspectos más relevantes surgidos en esta evolución y conformar una terminología mínima que nos permita reducir la ambigüedad al referirnos a este tema.

1. Introducción

Uno de los factores que contribuyen a incrementar la complejidad del problema del reuso de software es la falta de documentación y estructuración de la información existente. Este aspecto de la complejidad lo podemos reducir si logramos tener una visión global del problema del reuso y de sus actividades específicas.

Disponemos de tres elementos básicos para manejar la complejidad de un problema [Booch 94:16-21]: ***abstracción***, para ignorar los detalles no esenciales; ***descomposición***, para dividir en partes menos complejas; ***jerarquización***, para organizar el conocimiento. La jerarquización es el resultado de la abstracción, descomposición y organización del dominio de interés y nos proporciona una visión ordenada del mismo. Este ordenamiento en niveles lo podemos construir estableciendo (descubriendo) relaciones de clasificación y/o relaciones de composición estructural para los elementos del problema.

Con estos elementos intentaremos organizar el universo de discurso y con ello obtener una visión general del problema de reuso de software delimitando sus aspectos más relevantes. Esta estructuración del conocimiento nos proporcionará un marco de referencia y una terminología básica para referirnos al tema. No está demás aclarar que las jerarquías presentadas no son absolutas, podrían aceptar nuevos elementos e inclusive desechar algunos existentes. Nuestra intención más que definir jerarquías, es organizar el conocimiento para lograr una mejor comprensión.

Haremos un acercamiento de arriba hacia abajo. Presentamos una estructuración global de cómo entendemos el dominio, identificando cuatro grandes dimensiones o áreas que conforman el amplio espectro del reuso. Luego describimos cada área de manera general, para después adentrarnos en algunas ramificaciones. En este proceso de estructuración proporcionamos definiciones, descripciones e implicaciones, ramificando progresivamente las áreas de las que partimos para lograr la visión global que perseguimos.

Finalmente, describimos el proceso general de uso e incorporación de componentes a un depósito con la finalidad de introducir los principios fundamentales de su funcionamiento. De esta forma, completamos el panorama general para que en los siguientes capítulos exploremos una propuesta de estructuración de un depósito de componentes reutilizables OO.

1.1 Terminología básica

En general existe un acuerdo tácito, no estandarizado en la semántica de la terminología básica de reuso que se utiliza, sin embargo, existen diferentes matices que a nuestro juicio pueden causar confusión. Por ello, consideramos conveniente unificar un vocabulario mínimo para el desarrollo de esta exposición.

componente de software reutilizable (CSR)	cualquier producto en cualquier fase del ciclo de vida de un sistema que pueda reutilizarse, con la particularidad fundamental de que fue diseñado, construido, documentado y almacenado *con el fin expreso de su reuso posterior.*
reuso	uso de un componente de software reutilizable en el mismo o diferente contexto para el que fue diseñado.
reusabilidad	medida que indica el nivel potencial de reuso de un componente.
portabilidad	medida que indica el nivel potencial de reuso de un componente en más de una plataforma. La portabilidad incrementa la reusabilidad de un componente.
biblioteca de componentes	colección organizada de componentes reutilizables más la infraestructura funcional requerida para estructurar la colección, buscar y suministrar los componentes a los usuarios.
dominio	categoría de sistemas de software relacionados. Los ***dominios verticales*** se enfocan a un tipo particular de aplicación y los ***dominios horizontales*** se enfocan a un tipo particular de procesamiento de software común a varias aplicaciones, independientemente de su tipo.
oportunidades de reuso	situaciones en las que se puede incorporar el reuso en el proceso de desarrollo: utilizar componentes existentes, construir componentes reutilizables, utilizar o construir herramientas automatizadas, modificar metodologías de trabajo, etcétera.

2. Dimensiones del reuso de software

Identificamos cuatro grandes categorías o dimensiones para estructurar el dominio del reuso de software, guiándonos más por el sentido común que por alguna definición estricta: ***general, técnica, no técnica y organizacional***. La clasificación no es absoluta, podríamos definir otras categorías y un elemento podría encajar en más de una de ellas.

Esta estructuración nos servirá de guía en adelante y la abordaremos de manera incremental. Inicialmente presentamos una breve descripción de cada categoría y definimos cada uno de los elementos que se mencionan. Con este marco general en mente, abordamos con cierto detalle las dimensiones técnica y no técnica, pues nos proporcionarán un panorama amplio de las implicaciones del reuso. Las particularidades de la dimensión organizacional escapa a nuestro objetivo, pues más que implicaciones y consideraciones generales, son indicaciones concretas de cómo incorporar el reuso de software en una organización. Note que las ramificaciones de esta estructuración se van extendiendo a medida que avanzamos en la exposición.

Fig. 1-1 Dimensiones del reuso de software

GENERAL

Incide sobre cualquier otro aspecto y conforma un pilar fundamental para la consecución de los objetivos perseguidos. Sus objetivos centrales son la estandarización, la difusión y capacitación, así como la evaluación de productos, procesos y resultados.

Terminología

unidad estructurada (preferiblemente codificada) de definición de términos y conceptos. La finalidad principal es evitar ambigüedades haciendo referencias a un documento base. También puede ser muy útil para propósitos educativos.

Estándares — unidades estructuradas y codificadas que reglamenten y orienten el programa de reuso dentro del proceso global de desarrollo de software. Los estándares dependen de características particulares de cada organización, de las metodologías de desarrollo y del programa de reuso definido.

Educación — planes de capacitación que potencian el éxito de un programa de reuso, diferenciados según los tipos de personal involucrado: gerenciales, administradores de proyecto, líderes de proyectos, líderes de equipos de desarrollo, desarrolladores, soporte técnico, etcétera.

Evaluación — aspectos que miden y evalúan los productos y los resultados de un programa de reuso. Los elementos más relevantes son la ***certificación*** de los productos para garantizar cierta seguridad sobre su uso, las ***métricas*** que permiten medir esfuerzos y resultados, y los reportes de ***experiencias*** que documentan los logros prácticos alcanzados.

TECNICA — Engloba los aspectos técnicos del desarrollo de sistemas de software que utilizan y propician el reuso. Trata los elementos concretos del desarrollo que se relacionan directamente con la construcción, implantación, evolución y mantenimiento de la aplicación. Su enfoque principal está en los productos que se utilizan en la actividad de reuso, su construcción y organización, las metodologías y los tipos de reuso, la automatización y los lenguajes y paradigmas que favorecen el reuso.

Metodologías — orientaciones y adaptaciones para incorporar el reuso de software a los procesos de desarrollo de software existentes. Creación de nuevas metodologías adecuadas para lograr altos niveles de reuso.

Tipos de reuso — es importante determinar los tipos de reuso que existen y sus implicaciones para decidir cuáles serán integrados en una tecnología particular de desarrollo. Distinguimos el tipo de reuso según el tipo de componente que se utiliza, la manera cómo se incorpora y el nivel de reuso que se quiere lograr.

Ambientes y herramientas — que contemplen el reuso de software como un elemento integral en el proceso de desarrollo. Deben estar integrados a un depósito de componentes y estar basados en alguna metodología particular.

Bibliotecas — organización estructurada de software reutilizable, herramientas, procedimientos y personal encargado de su funcionamiento.

Componentes — aspectos que giran alrededor de la noción central de CSR: modelos de componentes y de composición, certificación y evaluación, especificación y documentación, construcción y evolución, etcétera.

Lenguajes y paradigmas — que propicien el reuso. De especial importancia son los niveles de abstracción, la expresividad y flexibilidad de las interfaces y los métodos de interconexión.

NO TECNICA

Los elementos no técnicos afectan la incorporación y promoción del reuso en la organización. A diferencia de los elementos técnicos, no tienen una relación directa con los productos involucrados en la construcción de la aplicación. Es notorio que a excepción de lo económico, se presenta una tendencia a menospreciar el papel que desempeñan los elementos no técnicos. Esto puede disminuir considerablemente las posibilidades de éxito de un programa de reuso.

Beneficios — para promover y evaluar un programa de reuso hay que determinar los beneficios que se pueden obtener.

Riesgos — que se deben considerar en la planificación y puesta en práctica del programa de reuso

Motivadores — elementos que impulsan ***a nivel personal*** el uso de las tecnologías de reuso.

Inhibidores — obstáculos relacionados con carencias y fallas del proceso de desarrollo y de la infraestructura de reuso.

Económicos — costos involucrados, métricas para cuantificar los beneficios obtenidos y modelos económicos de costo beneficio *ad hoc*.

Legales — implicaciones legales del reuso de software, derechos de propiedad y responsabilidades legales.

ORGANIZACIONAL

Los elementos organizacionales se ocupan de diseñar, coordinar, implementar y supervisar la implantación de un programa de reuso global en una organización. Buscan acoplar las técnicas de desarrollo de software basadas en

reuso a la estructura organizativa que desea adoptar el programa. Para ello, incorpora en un proceso integral los elementos generales, técnicos y no técnicos. Los elementos organizacionales podríamos calificarlos como no técnicos, sin embargo, creemos que su dimensión global e integradora amerita un tratamiento particular.

Cuando el reuso de software no está enmarcado en una política global de institucionalización, se tiene un conjunto de elementos y técnicas aisladas, sin coordinación, que no permiten alcanzar los logros que se esperan de esta tecnología. La mayoría de las veces, las organizaciones incorporan elementos técnicos sin abordar los aspectos no técnicos y organizacionales. Esto puede inclusive empeorar la productividad existente, porque inicialmente se requiere una inversión adicional considerable en la construcción de la infraestructura de reuso y sin las consideraciones organizacionales se corre el riesgo de no cosechar más adelante los resultados de esta costosa inversión.

Programa de Reuso

Define los elementos técnicos y no técnicos que se van a utilizar en la organización, los estándares que deben seguirse o construirse, los planes de capacitación para diferentes niveles de personal, las políticas de promoción, la incorporación del programa a la dinámica operativa de la organización y la evaluación del proceso de institucionalización y de los resultados del programa.

Proceso de Reuso

Es la especificación de los pasos a seguir para la institucionalización y uso del programa de reuso en una organización:

- identificar las oportunidades de reuso y hacer un análisis de factibilidad
- definir el programa de reuso y estrategias de implantación
- desarrollar un plan de implantación y ejecutarlo
- definir e implantar un programa de evaluación, monitoreo y ajuste que cubra todas las fases del proceso de reuso

3. Elementos técnicos

Los elementos técnicos abarcan los aspectos concretos del desarrollo que se relacionan directamente con la construcción, implantación evolución y mantenimiento de aplicaciones. En la estructuración general expuesta *mencionamos* los elementos técnicos de un mayor a un menor nivel de abstracción o granularidad. A fin de conformar una idea global de la tecnología de reuso abordaremos a continuación ***los tipos de reuso***. Al final de este capítulo abordamos el aspecto de las bibliotecas y en el capítulo 2 examinamos la relación que existe entre el paradigma orientado a objetos y el reuso de software.

3.1 Tipos de reuso de software

En la organización del conocimiento sobre el reuso de software destacan los tipos de reuso en los que puede ubicarse una determinada propuesta, pues cada categoría tiene implicaciones concretas en el logro de los objetivos que se plantea un programa de reuso.

Hemos utilizado varios criterios para ubicar una propuesta de reuso. Identificamos la ***técnica básica de reuso***, según esté basada o no en técnicas de compilación. Reconocemos el ***tipo de componente*** de acuerdo con la fase de desarrollo en la que se reutiliza y el ***tipo de incorporación*** que se realiza con o sin modificación. También determinamos el ***dominio de aplicación*** que puede cubrir una propuesta, los ***niveles de planificación*** que se pueden utilizar y los ***niveles de reuso*** que se pueden lograr.

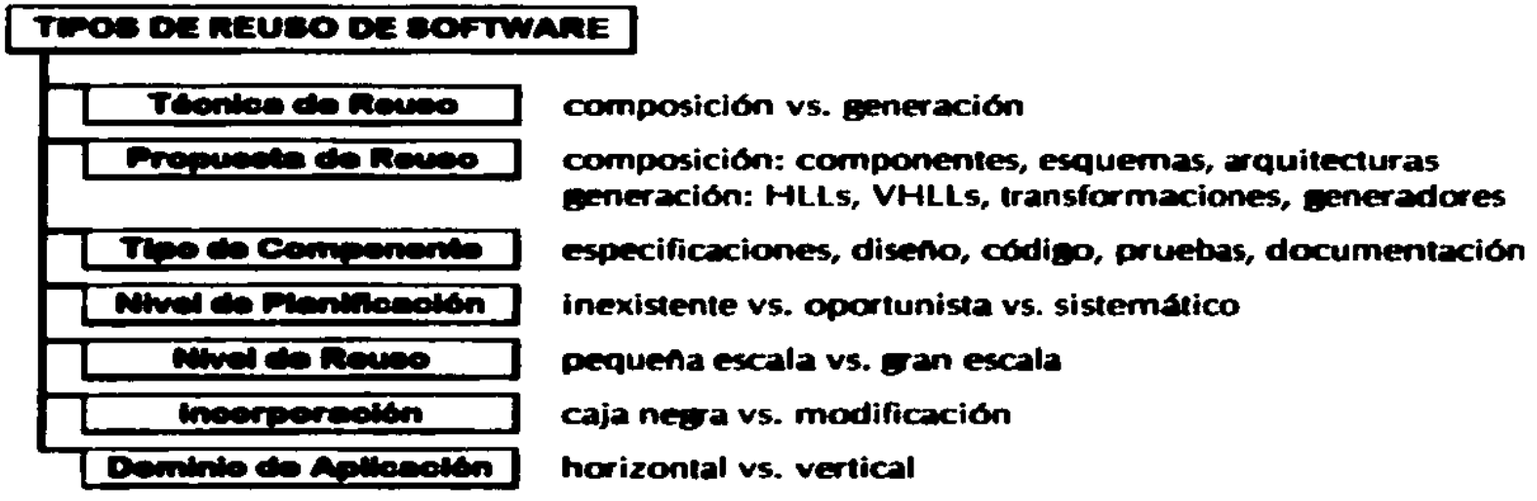
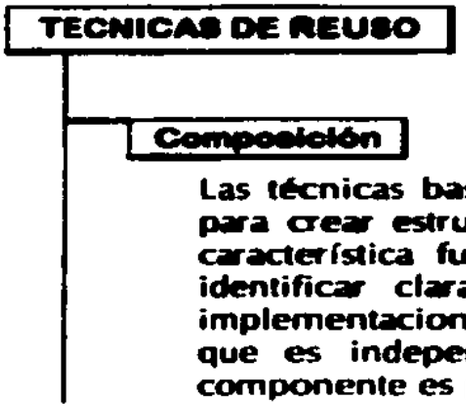


Fig. 1-2 *Tipos de reuso*

3.1.1 Técnicas de reuso: composición vs. generación



Las técnicas basadas en composición utilizan artefactos de software existentes para crear estructuras de mayor nivel, integrándolos con otros elementos. La característica fundamental de esta técnica es que un componente se puede identificar claramente, con una interfaz o especificación y una o varias implementaciones de la misma. Cada componente es atómico en el sentido de que es independiente y autocontenido (en mayor o menor grado). Cada componente es pasivo en el sentido que existe un ente externo que lo combina.

Además, "mantiene su forma e identidad antes y después de usarlo" [Biggerstaff 89:3], aunque sea necesario modificarlo para adecuarlo a un nuevo contexto.

Para obtener el éxito con esta técnica es requisito indispensable disponer de modelos de componentes y de composición bien definidos que faciliten la construcción e incorporación de componentes reutilizables. Los componentes deben estar organizados en un depósito integrado a un sistema de búsqueda, recuperación, clasificación e integración. Es una técnica flexible que se puede utilizar en propuestas de construcción de sistemas 'buttom-up' para componer estructuras de mayor nivel y en propuestas 'top-down' para descomponer la función y desempeño de un sistema.

Generación

En las técnicas basadas en generación está presente alguna técnica de compilación. No se puede identificar claramente el componente que es reutilizado, sus límites no son claros y es por tanto difuso. En lugar de utilizar especificaciones de componentes, existe un conjunto de especificaciones para el sistema y en lugar de implementaciones de componentes se tienen patrones de generación. De esta forma, en lugar de combinar varios componentes para satisfacer el conjunto de especificaciones del sistema, el programa generador construye las implementaciones adecuadas que satisfacen un conjunto de requerimientos.

Los componentes (patrones) no son atómicos porque dependen de todo el contexto del programa de generación y por tanto no son autocontenidos. Son componentes activos, porque son los actores principales en la creación de la implementación que se desea, el programador no tiene ingerencia en esta generación, salvo ciertas preferencias. Una consecuencia inmediata de estas características es que esta técnica de reuso sólo es compatible con una propuesta de construcción top-down, pues se debe construir todas las especificaciones del sistema antes de generar su implementación

Composición	Generación
• componente preciso (interfaz e implementación)	• componente difuso (patrones de generación)
• comp. atómico (independiente y autocontenido)	• comp. no atómico (depende del generador)
• componente pasivo (combinación externa)	• componente activo (combinación interna)
• construcción 'top-down' y 'buttom-up'	• construcción 'top-down'

Fig. 1-3 *Composición vs. Generación*

3.1.2 Propuestas de reuso

Las propuestas descritas están basadas fundamentalmente en los trabajos clásicos de [Krueger 92] y [Biggerstaff 89]. La orientación básica de estas propuestas es la construcción del código de aplicación, aunque algunas de ellas implican el reuso de cantidad considerable de análisis y diseño. No contemplan indicaciones ni proposiciones específicas en cuanto al proceso y metodologías de desarrollo.

A nuestro entender, el elemento fundamental de estas propuestas es el nivel de abstracción. En este contexto, una ***propuesta de reuso*** es aquella que utiliza abstracciones que encapsulan constructores de un nivel de abstracción inferior. De esta forma, al elevar el nivel de abstracción se consigue reutilizar patrones de construcción del nivel inferior y será menor el esfuerzo invertido en la construcción del sistema. Es interesante notar que el nivel de abstracción tiene una relación inversamente proporcional con las posibilidades de su aplicación (*generalidad*), a medida que crece el nivel de abstracción se reduce el número de casos en que puede aplicarse.

Este enfoque de análisis es muy ilustrativo, sin embargo, creemos que no es del todo adecuado. Ciertamente al utilizar un mayor nivel de abstracción se están reutilizando conjuntos de constructores de niveles inferiores. Sin embargo, la elevación del nivel de abstracción es un tema que excede las fronteras del reuso de software y tiene implicaciones más amplias. A nuestro parecer, además del nivel de abstracción, una propuesta de reuso debe considerarse como tal cuando involucre de manera consciente e intencional a los desarrolladores en la actividad del reuso. De manera particular debe estar presente la posibilidad de construir artefactos con el fin expreso de su posterior reuso y/o ambientes que contribuyan a su integración. En general, las propuestas descritas surgen como propuestas generales de construcción de software más que como propuestas específicas de reuso, pero es innegable que tienen implicaciones importantes. En particular, nos han permitido conformar un conjunto de criterios técnicos útiles para evaluar algunos aspectos de una propuesta de reuso.

1. Propuestas de reuso

PROPUESTAS DE REUSO

Componentes de código

Consideraremos como ***componente de código*** aquella abstracción que está encapsulada en un constructor de agrupación (módulo, clase, procedimiento, etc.) de un lenguaje de programación. La construcción de componentes ha impulsado el desarrollo de técnicas sistemáticas para su almacenamiento y recuperación en bibliotecas o catálogos de componentes. De esta manera, surgen múltiples esquemas de clasificación y recuperación, modelos de componentes y de composición. Es particularmente importante la representación de la abstracción, por lo general un componente sólo tiene una especificación sintáctica y carece de una especificación semántica. Esto dificulta en sumo grado la identificación de las oportunidades de reuso para los componentes construidos, sobre todo la identificación automatizada.

Existen propuestas que utilizan especificaciones formales para denotar el significado de la abstracción, pero la complejidad de la especificación crece con el nivel de abstracción y no es muy claro que una propuesta formal de reuso pueda ser exitosa masivamente en la construcción de aplicaciones. Para la solución de este problema pueden contribuir mucho las herramientas

automatizadas para el manejo de la biblioteca, proporcionando una estructuración clara de la información y maneras "intuitivas" de localizar componentes según las características que se demanden.

Esquemas de software

El fundamento de los componentes de código es reutilizar código que pueda aglutinarse alrededor de una abstracción. Los esquemas de software proponen una extensión a este enfoque de manera que el fundamento pase a ser la abstracción propiamente dicha, algoritmos y estructuras de datos abstractas en lugar del código. Algunos de los elementos que diferencian a los esquemas de software de los componentes de código son:

- *alto nivel de parametrización*. Los parámetros son entidades abstractas que pueden reemplazarse por datos, por otros esquemas e inclusive por indicaciones de desempeño
- *especificación formal*. Incluye una descripción semántica formal. Contiene aserciones para la sustitución de los parámetros y precondiciones, invariantes y poscondiciones para la ejecución de una instanciación del esquema
- *prueba de correctez*, para la especificación
- *ambiente integrado para su uso*. Existe un ambiente que orienta la selección e integración de los esquemas, una herramienta automatizada puede encontrar esquemas que satisfagan las precondiciones y poscondiciones del problema

Arquitecturas de software

Una arquitectura de software contiene la estructura del diseño de un sistema o subsistema de software. Por lo general, una arquitectura está acompañada de la implementación de las estructuras básicas del sistema. El nivel de abstracción involucrado es muy alto, "las arquitecturas de software se enfocan sobre los subsistemas y su interacción más que las estructuras de datos y los algoritmos" [Krueger 92:174]. Son componentes de granularidad gruesa y pueden combinarse entre sí para formar arquitecturas de mayor tamaño.
Para poder capturar la esencia de la estructura de un sistema o un subsistema se requiere tener un gran conocimiento de él. Este conocimiento se obtiene, en general, de la experiencia en la construcción de muchos sistemas similares bajo diferentes condiciones de requerimientos. El ***análisis de dominio*** se encarga de la organización sistemática del conocimiento del dominio para construir la arquitectura.

Lenguajes de alto nivel (HLL) y de muy alto nivel (VHLL)

Ambos utilizan abstracciones de propósito general, aceptan programas de código fuente y generan programas ejecutables mediante un compilador. Las abstracciones básicas de un HLL encapsulan patrones repetitivos encontrados en la construcción de software con los lenguajes ensambladores. Las

abstracciones de un VHLL son de mayor nivel y a diferencia de un HLL no surgen del descubrimiento de patrones repetitivos, son el resultado de la elaboración de un modelo matemático construido expresamente para tener ese nivel de representación.

Sistemas basados en transformaciones

Los sistemas basados en transformaciones utilizan VHLLs para construir un prototipo del sistema y especificar la conducta semántica de un sistema de software. Posteriormente, el desarrollador indica transformaciones a las especificaciones para mejorar la eficiencia sin cambiar la conducta semántica. "Una transformación es un mapeo de programas a programas. El resultado de una transformación es un programa semánticamente equivalente, pero más eficiente." [Krueger 92:170]

Generadores de aplicación

Un generador de aplicación utiliza abstracciones de muy alto nivel específicas del dominio de la aplicación. Los desarrolladores utilizan estas abstracciones para especificar qué debe hacer el sistema y estas especificaciones son trasladadas automáticamente en programas ejecutables por el generador. El nivel de abstracción puede ser mayor que en las arquitecturas de software, porque se reutiliza el diseño de un sistema de software completo.
Se requiere tener un conocimiento muy detallado del dominio de aplicación y utilizar técnicas de compilación para la construcción del generador. El criterio básico para justificar la construcción del generador es la necesidad de construir muchas aplicaciones de un mismo tipo con diferentes requerimientos. Es interesante notar que "con los generadores de aplicación la evolución de un sistema se realiza por la modificación de las especificaciones del sistema". [Krueger 92: 162]

II. Taxonomía de propuestas y criterios de evaluación

Si bien es interesante revisar diferentes propuestas de reuso de software, lo es más aún poder identificar criterios generales que permitan evaluar el impacto de una propuesta en un programa de reuso de software. A tal efecto, identificamos categorías de elementos que se pueden identificar en cada propuesta y para cada categoría mencionamos los criterios que cobran mayor relevancia para la tecnología de reuso.

CATEGORIAS Y CRITERIOS

Técnica de reuso composición vs. generación

no es clara la ventaja de una u otra técnica. Las técnicas de generación tienen la ventaja de un alto grado de automatización, pero no siempre son aplicables. Las técnicas de composición pueden ser "naturales" a los desarrolladores y utilizarse en un mayor número de casos.

Nivel de abstracción — **a mayor nivel de abstracción es mayor el beneficio**

la granularidad de la abstracción determina el beneficio que se obtiene con su reuso.

Propósito de la abstracción — **general vs. específico**

las abstracciones de propósito general están limitadas en granularidad porque deben satisfacer todos los dominios de aplicación, en cambio las abstracciones de propósito específico pueden presentar niveles mucho mayores.

Tipo de abstracción — **formal vs. no formal**

el tipo de abstracción no necesariamente es un indicador del beneficio potencial, sin embargo creemos que es difícil que una propuesta de reuso que utiliza abstracciones formales pueda promocionarse masivamente para el desarrollo de software.

Nivel de automatización — **a mayor automatización mayor beneficio del reuso**

el nivel de automatización es mayor en las técnicas de generación porque mapea directamente un conjunto de especificaciones al programa ejecutable.

Nivel de reuso — **pequeña escala vs. gran escala**

las propuestas que más destacan desde la perspectiva del reuso a gran escala son las arquitecturas de software y los generadores de aplicación. Los generadores de aplicación pueden reportar el máximo beneficio cuando están presentes las condiciones para su construcción y/o aplicación. Las arquitecturas de software por otro lado, por utilizar el principio de composición podría acercarla más como una propuesta estándar para el reuso de software. Adicionalmente, una propuesta de reuso a menor escala como los componentes de código puede desempeñar un papel importante si se integra en un ambiente automatizado para su clasificación, búsqueda y recuperación.

Es conveniente acotar que éstos no son los únicos criterios, en última instancia cualquier propuesta de reuso debe ser evaluada con base en las condiciones específicas del proceso y metodología de desarrollo, tipo de aplicaciones que se desarrollan, personal disponible y sobre todo los recursos que se pueden destinar a su incorporación. Finalmente, ilustramos en un diagrama la relación que hay entre cada propuesta y las categorías descritas.

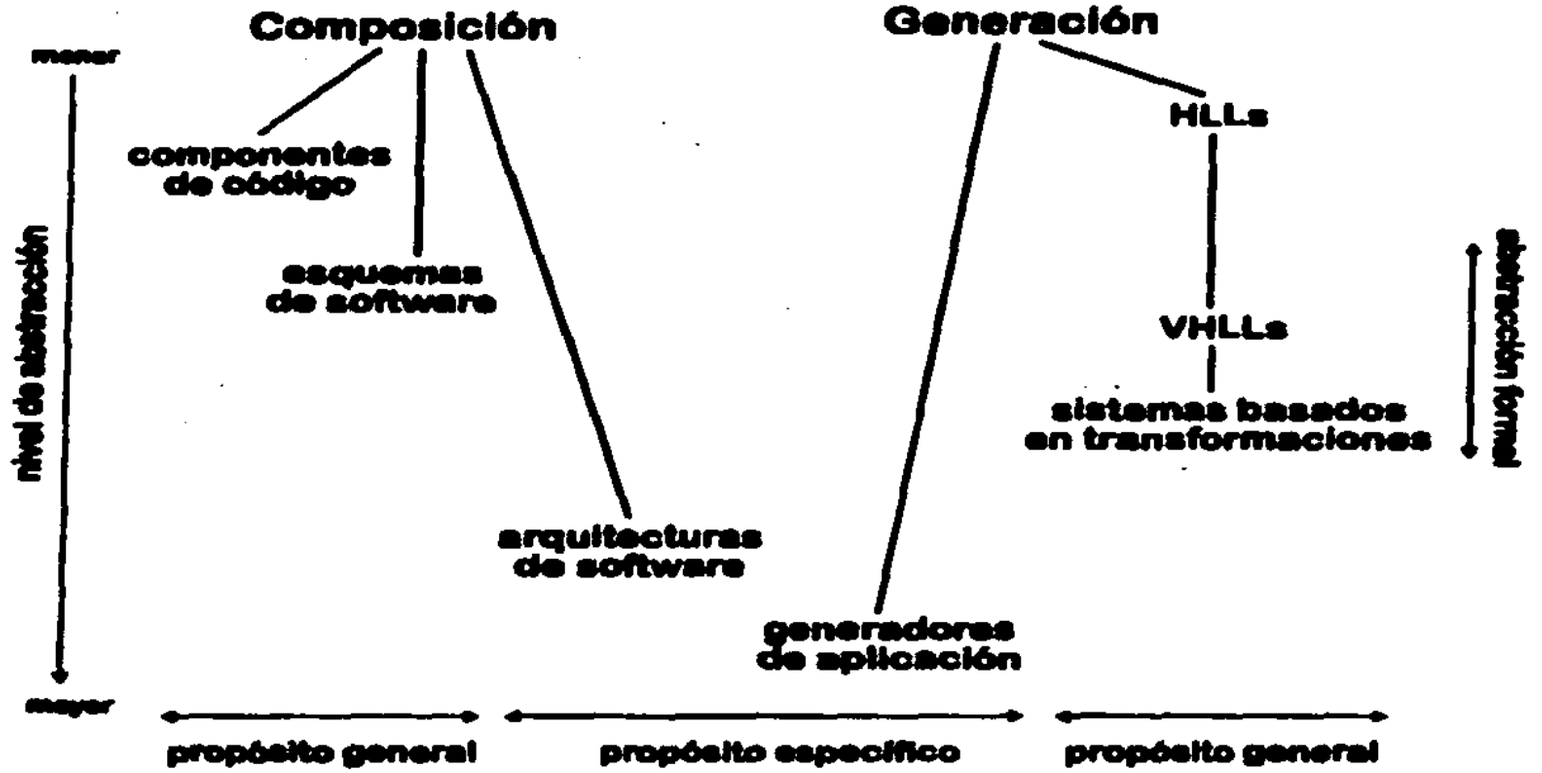


Fig. 1-4 Propuestas de Reuso

3.1.3 Tipo de componente

Podemos reutilizar componentes en cualquier fase del proceso de desarrollo. Los componentes asociados a las primeras etapas de desarrollo representan un nivel de abstracción mayor que los asociados a las últimas etapas, por lo que diremos que son componentes de mayor nivel. En este contexto, un componente en un nivel puede o no "implementar" la abstracción de un elemento de mayor nivel y puede o no tener asociados elementos de menor nivel que implementen su abstracción. Nótese que se utiliza el término *elemento* en lugar de componente para reforzar el hecho que el elemento asociado no es necesariamente un componente reutilizable. Esto se debe a que en cada nivel un componente (elemento) debe satisfacer un conjunto de requerimientos para considerarse un componente reutilizable y el hecho de que no los cumpla, no significa que no implemente la abstracción del nivel anterior.

En esta relación de interdependencia entre elementos asociados en diferentes niveles, encontramos relaciones de importancia que deben tomarse en cuenta para el diseño, construcción y uso de componentes reutilizables:

- el reuso de un componente en un nivel no determina la reutilización de sus elementos asociados en los niveles inferiores, aunque potencia su reuso en gran medida.
- el reuso de un componente en un nivel sin modificación, por lo general debe estar acompañado de sus elementos de mayor nivel.
- el no reutilizar un componente en un nivel no determina la no reutilización de los elementos asociados de diferentes niveles.
- las asociaciones entre componentes de diferentes niveles tiende a conformar una estructura de árbol. Por lo general, un componente en un nivel tiene más componentes asociados en el nivel inmediato inferior que en el inmediato superior (p.e., un diseño puede tener varias implementaciones, pero no es común que una implementación satisfaga más de un diseño).

TIPO DE COMPONENTE

Especificaciones

Un ***componente reutilizable de requerimientos*** es la especificación de una función y desempeño que se puede utilizar en más de un sistema. El hecho de reutilizar especificaciones denota un alto grado de planificación de reuso, pues se contempla el reuso desde el inicio del desarrollo. Además, reduce el tiempo de la definición de requerimientos, disminuye la posibilidad de error y estandariza las especificaciones.

Diseño

Un ***componente reutilizable de diseño*** es la especificación de implementación de un requerimiento que se puede utilizar en más de un sistema. El reuso de estos componentes reduce el esfuerzo de diseño del sistema, disminuye los errores y estandariza el diseño. El diseño de un sistema establece una arquitectura de software que constituye un marco para el reuso posterior, pues determina funciones y sobre todo interfaces de conexión. Si además de los componentes se logra tener una arquitectura reutilizable, la oportunidad de reuso será significativamente mayor.

Código

Un ***componente reutilizable de código*** es la implementación de un diseño en un lenguaje específico y muchas veces para una plataforma determinada, que se puede utilizar en más de un sistema. El reuso exclusivo de código sin conexión con los elementos asociados en niveles superiores denota un bajo grado de planificación del reuso y puede ser cuestionable la inversión de recursos. Entre los beneficios tenemos la disminución del esfuerzo de implementación, reducción de errores y estandarización de las funciones, desempeño e interfaz con el usuario.

Pruebas

Un ***componente reutilizable de pruebas*** es la especificación de un conjunto de pruebas que debe satisfacer un *tipo* de componente, más que un componente concreto. Es irrelevante si las pruebas se hacen sobre otro componente reutilizable. En particular, se podría aplicar un componente de pruebas a las diferentes implementaciones de una especificación de requerimientos o de diseño.

Documentación

Un ***componente reutilizable de documentación*** es la síntesis explicativa de un componente y/o proceso del sistema (documentación de cada componente reutilizable, manuales de usuario, manuales de procedimientos, etcétera). Un esquema adecuado para el manejo de la documentación es utilizar una mezcla de composición y generación; composición con documentos relativamente estables y generación con la información de los componentes utilizados en el sistema.

3.1.4 Nivel de planificación: inexistente, oportunista, sistemático

Podemos identificar tres categorías claramente diferenciadas que denotan el nivel de planificación con que una organización aborda la tecnología de reuso. Estos niveles de planificación se pueden utilizar como elementos de diagnóstico y pueden ayudar a precisar carencias en la organización del proceso de desarrollo.

NIVEL DE PLANIFICACION

Inexistente

No existen métodos definidos para la actividad de reuso, no existe promoción, coordinación ni responsabilidades definidas. Los artefactos que se reutilizan son de bajo nivel, en su mayoría código. No existe un depósito de componentes ni herramientas automatizadas de clasificación y recuperación. Las oportunidades de reuso se identifican porque alguien recuerda que alguna vez se construyó algo parecido y se hace un 'cut and paste' si se llega a encontrar lo requerido. Se presenta mucha edición de los artefactos que se reutilizan. Por lo general no se reutiliza un artefacto como una unidad, no existen componentes reutilizables, se reutilizan fracciones desarrolladas para un mismo o diferente propósito. Encontramos que éste es el caso más frecuente, es una práctica intuitiva y común. A pesar de ser un grado incipiente de reuso, esta situación representa un nivel planificación nulo o ***inexistente***, pues no existe anticipación sobre la actividad.

Oportunista

Aparecen los componentes reutilizables, es decir, construidos con el fin expreso de su reuso posterior. Los componentes son incluidos en una biblioteca integrada con un sistema de clasificación y búsqueda de componentes. En este nivel podemos ubicar a las tecnologías de generación, pues requieren una profundización en el conocimiento y planificación de su reuso futuro, así como la construcción de los ambientes de generación. Existen mecanismos, no sólo la memoria de individuos, que permiten asociar demandas particulares con las características de los artefactos reutilizables disponibles. Todo esto constituye una infraestructura mínima planificada que permite identificar oportunidades de reuso e integrar los artefactos a un nuevo desarrollo.

Este nivel recibe el nombre de *oportunista*, porque sólo aprovecha las oportunidades que se presentan para reutilizar artefactos existentes. No promueve la construcción de otros artefactos y la metodología de desarrollo no se ve afectada significativamente por la presencia del reuso. En la mayoría de los casos los artefactos que se reutilizan siguen siendo de bajo nivel, pero organizadamente y con clara conciencia de los beneficios que reporta. También podemos encontrar artefactos de nivel medio (documentación, pruebas, diseño), pero difícilmente nos toparemos con reuso de especificaciones. Además de reutilizar sólo algunos tipos de componentes, éstos no son de grandes dimensiones (p.e., arquitecturas o subsistemas).

Sistemático

Existen varios indicadores que permiten diferenciar un nivel de planificación sistemático de los anteriores:

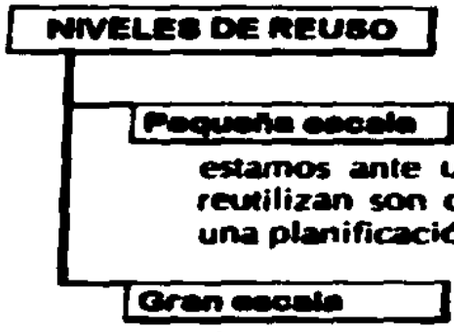
- se reutilizan artefactos de todos los niveles y de todos los tamaños
- el reuso se introduce desde la especificación del sistema
- se promueve la construcción de nuevos artefactos reutilizables
- existen métodos y métricas de evaluación
- existen métodos, coordinación y responsabilidades asignadas en la actividad de reuso

Tal vez la característica más relevante es que el proceso de desarrollo se ha modificado sustancialmente para lograr este nivel de planificación de reuso, anticipando en cada momento del desarrollo los posibles artefactos que se pueden reutilizar en las fases posteriores y especificando las características de interfaz, funciones y desempeño de los nuevos artefactos reutilizables que serán construidos. De esta forma, el reuso ya no es sólo un elemento más del proceso o metodología de desarrollo, pasa a ser una guía que orienta la construcción de los sistemas de software.

3.1.5 Nivel de reuso: pequeña escala vs. gran escala

El criterio principal con que se evalúa la escala del reuso en la literatura es el tamaño de los componentes reutilizados. Sin embargo, existe un segundo criterio de igual importancia: la existencia de un plan de reuso que abarque todo el proceso de desarrollo.

El tamaño de un componente está directamente relacionado con el beneficio que se obtiene con su reuso y el costo de construcción y mantenimiento. Por otro lado, está inversamente relacionado con la posibilidad de reuso, pues es más específico. Cuando hablamos de componentes grandes no nos referimos a componentes que sólo tienen más código e interfaces que otros, nos estamos refiriendo a componentes que son por sí solos subsistemas, arquitecturas, protocolos, etc. y están plasmados en un conjunto de requerimientos, diseños e implementaciones construidos especialmente para su reutilización.



estamos ante un reuso de pequeña escala cuando los componentes que se reutilizan son de bajo nivel (código fundamentalmente) y/o cuando no existe una planificación sistemática de la actividad de reuso.

el reuso a gran escala está estrechamente ligado a componentes de requerimientos, son componentes de alto nivel y corresponden a una planificación del reuso desde el inicio del desarrollo. Esto generalmente implica un reuso a gran escala porque potencia la reutilización de todos los componentes asociados de diseño, código, interfaces de comunicación, pruebas, documentación, etcétera.

3.1.6 Incorporación del componente: caja negra vs. modificación

Esta categoría está orientada a las tecnologías de composición. ¿Qué significa incorporar un componente sin modificación o ***reuso de caja negra***?. Básicamente no modificar la interfaz del componente. Esto se puede lograr de varias maneras:

- construir un entorno que permita incorporarlo sin modificar su interfaz
- construir una nueva interfaz basada en la original (sin modificarla) que se adapte al entorno disponible. La nueva interfaz probablemente tendrá asociada una nueva implementación
- una combinación de las anteriores

Consideraremos que una modificación a la implementación de un componente sin afectar su interfaz es sólo una nueva alternativa de desempeño, pero no una modificación del componente.

Ambas formas requieren cuidar factores de importancia. Para el reuso de caja negra, los componentes deben tener un alto grado de generalidad y flexibilidad, lo que puede disminuir las características de desempeño del componente y por tanto, del sistema resultante. Para el reuso con modificación debe verificarse que el esfuerzo de modificar e incorporar el componente debe ser menor que construir uno nuevo. Así, en el primer caso necesitamos orientaciones para la construcción de los componentes y en el otro, necesitamos métricas que permitan medir los esfuerzos de construcción.

Es preferible el reuso de caja negra, pero es indudable que el reuso con modificación es necesario y hay que tomar en cuenta dos factores fundamentales cuando se utiliza:

- puede invalidar la correctez del componente original, la modificación puede requerir validar, probar y depurar el componente nuevamente
- puede reducir considerablemente el beneficio de la reutilización, pues es una manipulación a un bajo nivel de abstracción

3.1.7 Dominio de aplicación: vertical vs. horizontal

"Un dominio es un área de actividad o conocimiento que contiene aplicaciones que comparten un conjunto de datos y capacidades" [CARDS 93a:25]. Existen dominios verticales y horizontales. Un ***dominio vertical*** o dominio específico abarca una clase de aplicaciones de propósito específico (p.e., sistemas de información contables, sistemas de control de tráfico, etcétera). En cambio, un ***dominio horizontal*** abarca una clase de aplicaciones que se pueden integrar a varios dominios verticales, es decir, abarca aplicaciones de propósito general (p.e., interfaces gráficas de usuario, procesamiento matemático y estadístico, procesamiento digital de imágenes, etc.).

El reuso en un dominio determinado por lo general implica un mayor grado de planificación y un reuso a mayor escala, pues requiere haber determinado con precisión las características del dominio y haber construido los componentes reutilizables que plasman el conocimiento del dominio para la construcción de sistemas. Este tipo de reuso representa el mayor beneficio posible y cualquier iniciativa de cierta envergadura debe considerar esta propuesta como primera opción. La tecnología que se usa para reutilizar el conocimiento del dominio puede ser de composición, generación o una combinación, no hay limitaciones al respecto.

4. Elementos no técnicos

Los elementos no técnicos afectan la factibilidad, promoción y evaluación de la incorporación del reuso al proceso de desarrollo. La fase previa a la elaboración de un programa de reuso es analizar la factibilidad de éxito que puede tener un programa de este tipo. El punto de partida es precisar cuáles son los ***beneficios*** que se esperan obtener y en la medida de lo posible cuantificar algunos de ellos. Por otro lado, hay que determinar los ***riesgos*** asociados al programa y determinar si éstos pueden ser controlados o eliminados y de qué manera.

Los beneficios establecen automáticamente los parámetros de evaluación, pues se evaluará en función de lo que se espera obtener y además son el argumento principal para la promoción inicial del programa. Luego, en la instauración se requiere una promoción más activa cuyo objetivo principal son las personas involucradas. Para esto se requiere determinar los factores que pueden ***motivar*** una actitud positiva y que propicien el éxito del programa. En contraposición, existen factores ***inhibidores*** y su identificación es básica para prever una estrategia adecuada de implementación.

Es indudable que el factor ***económico*** es el elemento no técnico de mayor peso y está presente en todo el proceso de reuso. Por lo tanto, es imprescindible disponer de un modelo económico adecuado que permita estimar los beneficios potenciales y evaluar los beneficios reales. Finalmente, nos encontramos con que la reutilización de componentes de software tiene implicaciones legales y una definición clara podría convertirse en un factor de promoción de las tecnologías de reuso.

A continuación describiremos cada uno de estos aspectos y proporcionaremos recomendaciones generales que pueden incluirse en un programa de reuso global para potenciar el logro de los beneficios que se esperan.

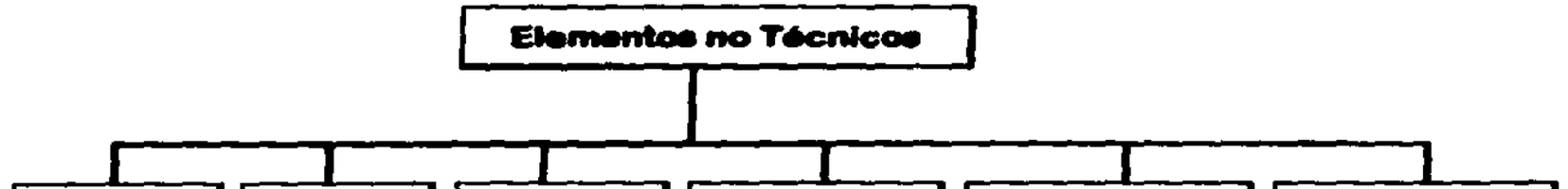


Fig. 1-5 Elementos no técnicos del reuso

4.1 Beneficios

El énfasis creciente en el reuso de software se fundamenta en la expectativa de que traiga consigo una serie de beneficios para el desarrollo de sistemas. Los beneficios son el argumento básico para justificar la necesidad del reuso de software y es por ello que es importante precisar las ventajas que promete. De esta manera, los beneficios conforman un pilar fundamental para la promoción de un programa de reuso y adicionalmente establece de manera automática los parámetros generales de evaluación con que se medirán los resultados.

Distinguimos dos categorías de beneficios, ***técnicos*** y de ***producción***. Los beneficios técnicos están relacionados con las características técnicas del producto de software (desempeño, mantenimiento, interoperabilidad, calidad, confiabilidad, documentación). Los beneficios de producción están relacionados con las ventajas económicas, de comercialización y competitividad.

BENEFICIOS TECNICOS

- **menor esfuerzo de desarrollo**
  con el uso de componentes ya construidos se ahorra esfuerzo en la construcción, documentación y pruebas del software.
- **reducción de la distancia conceptual**[1]
  al disponer de componentes de aplicación estándares, el reuso eleva el nivel de abstracción. Esto permite concentrarse más en el dominio del problema que en los problemas de implementación y reduce la complejidad del modelado del problema.
- **menor esfuerzo de mantenimiento**
  los componentes deben haber pasado por un conjunto de pruebas que garantiza una reducción en la aperición de errores y reduce la cantidad de software que debe ser mantenido.
- **mejora en la confiabilidad y calidad**
  los componentes reutilizables deben satisfacer un conjunto de requerimientos de seguridad y correctez (validación) y de calidad (calificación), además que se van probando en la práctica. Esto se traduce en una mejora de la calidad y confiabilidad total del sistema.
- **mejora en el desempeño**
  los componentes reutilizables deben satisfacer requerimientos de desempeño adecuados, esto en principio garantiza cierto nivel de desempeño para el sistema, aunque se debe asegurar que la construcción de los contextos para su inserción no provoquen un detrimento en las cualidades dinámicas esperadas.
- **mejora en la interoperabilidad**
  el concepto de componente recae fuertemente en la noción de interfaz; se espera que con la estanderización de interfaces para determinados componentes reutilizables, los componentes sean capaces de interoperar cada vez más fácilmente con otros.
- **soporte para la construcción de prototipos**
  al disponer de un conjunto de componentes reutilizables operacionales, se tiene una base para construir prototipos rápidamente.
- **mejora en la documentación**
  se persigue que exista una documentación estándar que se va estabilizando con la madurez del componente y que deriva en una reducción en la construcción de la documentación del nuevo sistema y en una mejor calidad de la misma.
- **aumento en la comprensión del sistema**
  en la medida en que se puede identificar componentes reutilizables como partes del

[1] "La distancia conceptual es el esfuerzo que debe invertirse para ir desde el concepto inicial del sistema a su implementación final" [Bell 93:3].

sistema, se está utilizando abstracciones cada vez mejor delimitadas con características, funciones e interfaces bien definidas. Así mismo, se identifican más claramente las interacciones entre los diferentes componentes.

estándar para el desarrollo y mantenimiento
por el uso de componentes y procedimientos comunes para varias aplicaciones.

BENEFICIOS DE PRODUCCION

aumento de la productividad
reduce el esfuerzo de desarrollo, la cantidad de documentación y la cantidad de pruebas.

reducción del tiempo de desarrollo
por el uso de componentes ya construidos y probados, y por el uso de arquitecturas que orientan la integración de los componentes.

desarrollo con menor cantidad de personal
por la reducción en el esfuerzo de desarrollo.

desarrollo de aplicaciones a la medida
si se dispone de una arquitectura estándar y de una biblioteca adecuada para el dominio de la aplicación, se facilita su adaptación a requerimientos particulares.

estandarización de una línea de productos
en cuanto a funciones y aspecto, si se utiliza una misma arquitectura y componentes básicos reutilizables.

reducción de costos globales a largo plazo
por la reducción del tiempo y esfuerzo de desarrollo. Los beneficios son a largo plazo, porque a corto plazo se necesita hacer una inversión en infraestructura y en el proceso de institucionalización del reuso.

reducción de costos de mantenimiento
por la reducción del esfuerzo de mantenimiento y porque el mantenimiento de un componente repercute en la evolución de todos los productos que lo utilizan.

reducción de costos de entrenamiento
siempre y cuando se establezca un modelo de diseño estándar que conforme una tecnología uniforme en lo general para los construcción de diferentes sistemas.

aumento de la competitividad
permite manejar la construcción simultánea de proyectos que tienen requerimientos similares. También permite reducir los tiempos de desarrollo y de entrega o puesta en el mercado.

mayores oportunidades para nuevos productos
al liberar recursos dedicados al desarrollo y mantenimiento, se pueden explorar y generar nuevos productos para el mercado con la misma infraestructura.

posible comercialización de componentes y herramientas
construidas en el proceso de desarrollo de aplicaciones, sobre todo a medida que se difunde la necesidad del reuso de software y aumenta la demanda de estos productos.

4.2 Riesgos

No debemos perder de vista que en la consecución de los beneficios esperados se presentan riesgos asociados y que es necesario conocerlos para elaborar políticas adecuadas que minimicen su aparición. La precisión de los riesgos es un elemento clave de la planificación y puesta en práctica del programa de reuso. Al igual que para los beneficios, podemos identificar riesgos asociados con elementos técnicos y riesgos asociados con elementos de producción. Para controlar las situaciones de riesgo se debe disponer de una estrategia de evaluación y manejo de riesgos acorde a la metodología de desarrollo a la que se le ha incorporado el reuso.

RIESGOS TECNICOS

- **aumento en la complejidad del sistema**
  si los componentes proporcionan funciones en exceso a las necesitada y si requiere la construcción de contextos complejos para su inserción.
- **detrimento en la calidad y desempeño**
  si se reutiliza componentes que se ajustan a los requerimientos funcionales, pero no satisfagen los requerimientos de calidad y desempeño del nuevo sistema.
- **impacto negativo en el mantenimiento**
  si se reutilizan componentes no diseñados para el reuso, pues su modificación se dificulta por la falta de documentación, planes de prueba e inflexibilidad de diseño.
- **impacto negativo en la evolución**
  si no hay uniformidad de interfaces y las nuevas versiones de los componentes son excesivamente diferentes del componente usado en la construcción, se requerirá un mayor esfuerzo en la evolución del sistema.
- **expansión de errores y dificultades de adaptación entre sistemas**
  si no se dispone de mecanismos de realimentación que comuniquen las adaptaciones y errores de los componentes a sus productores para que realicen las modificaciones necesarias.
- **soporte no garantizado**
  sobre todo cuando se trata de componentes de terceros y/o propios construidos bajo políticas poco estrictas de documentación y evolución.

RIESGOS DE PRODUCCION

- **aumento de los costos y tiempos de desarrollo**
  por la construcción de componentes reutilizables, pues se le imponen requerimientos adicionales sobre su documentación, calidad, confiabilidad, pruebas, mantenimiento, portabilidad, etcétera.
- **estimación de costos incompleta del reuso del componente**
  el desarrollador debe considerar los costos de ubicar, adquirir, adaptar e integrar el componente reutilizable en relación a los costos que implica desarrollar uno nuevo. Si no se hace este análisis, puede que no haya ahorro de esfuerzo con el reuso.

estimación equivocada de la necesidad de los componentes
el costo mayor de la construcción de un componente reutilizable sólo se justifica cuando se reutiliza más de una vez. Si se hace una estimación equivocada de su reuso futuro, se estará perjudicando la inversión.

4.3 Motivadores

Podemos establecer una asociación inmediata al decir que los beneficios esperados de un programa de reuso son en sí mismos motivadores del programa. Esta asociación es correcta, pero incompleta. Los beneficios esperados conforman objetivos generales del programa de reuso, sin embargo, lo que realmente nos interesa es cómo lograr que las personas concretas en los diferentes niveles se sientan motivadas a alcanzar los beneficios que se esperan. Un motivador o incentivo es un elemento que impulsa a nivel personal el uso de la tecnología, potenciando los beneficios esperados.

¿Por qué es necesario un programa de incentivos?. Para impulsar la realización del reuso es necesario establecer un programa adecuado de incentivos coherente con las metas de la organización. Para ello, debemos identificar a los diferentes participantes del proceso. Generalmente podemos considerar al personal técnico y personal administrativo, distribuidos en diferentes niveles jerárquicos. El programa de incentivos debe elaborar un sistema de recompensas (no necesariamente monetarias) para cada uno de ellos.

Además de personal y niveles, el programa debe garantizar consistencia con las metas globales de reuso propuestas. Esto es importante, porque perder la visión global puede traer consecuencias indeseables. Por ejemplo:

- si se ofrece una recompensa por cada componente que se fabrique, puede generar una biblioteca que aunque tenga componentes reutilizables, estos no sean muy utilizados o no signifiquen mucho ahorro de esfuerzo, obteniendo una biblioteca de poco valor. Si se quiere recompensar al fabricante del componente, se debe tomar en cuenta factores adicionales, como son cuánto se reutiliza su componente y cuánto ahorro de esfuerzo significa para quien lo reutiliza. Con esto se estimula el desarrollo de componentes valiosos.
- si se ofrece una recompensa por el reuso de componentes, se corre el riesgo de que los reutilizadores incorporen componentes del mayor tamaño posible sólo para utilizar una parte de él.

Para evitar este tipo de distorsiones y disponer de una base adecuada de evaluación, es importante recordar que el reuso es un medio para lograr un objetivo, por lo que "...el sistema de incentivos debe estar diseñado para motivar la producción y utilización de componentes reutilizables donde efectivamente soporte los objetivos de la organización" [Malan 93:3].

Un elemento subyacente en un sistema de recompensas es la evaluación. Si hablamos de una evaluación cuantitativa, deberemos disponer de métricas que permitan cuantificar los desempeños personales y del proyecto. Si hablamos de una evaluación cualitativa, deberemos disponer de criterios más que medidas, p.e., novedad de una implementación. La evaluación cualitativa está relacionada con un uso espontáneo y

voluntario de la tecnología, mientras que la evaluación cuantitativa tiene un carácter obligatorio. Algunas experiencias [Wasmund 93:4] reportan que un programa de incentivos de carácter obligatorio con una evaluación cuantitativa tiene una relación de mejora en el reuso de 4:1 respecto a un programa voluntario con evaluación cualitativa, aunque clarifica que las opiniones no son uniformes a este respecto.

Evaluación	Ventajas	Desventajas
Cualitativa	- evita métricas y costos asociados - el uso de criterios reduce fricciones	- no es estímulo fuerte para el reuso
Cuantitativa	- mayor presión para el uso de la tecnología	- mayor propensión a la distorsión

El nivel administrativo juega un papel fundamental en la institucionalización del reuso, sobre todo por la reestructuración del proceso de desarrollo. "El cambio organizacional típicamente encuentra resistencia y un liderazgo de alto nivel juega un rol importante en propagar una visión que inspire y motive a aquellos afectados por el cambio" [Malan 93:4]. Es decir, los administradores deben ser los motivadores de este proceso. Por otro lado, son los que deben mantener la visión global a largo plazo del reuso para evaluar los beneficios en multiproyectos y presentar los resultados como un fuerte incentivo.

4.4 Inhibidores

Es conveniente recordar que los beneficios que se esperan del reuso son potenciales, es decir, no se obtienen de manera automática. En el camino surgen obstáculos y barreras a las que denominamos de manera genérica ***inhibidores***. Existe la tendencia a presentar los motivadores e inhibidores como conceptos opuestos y complementarios, es decir, la ausencia de un inhibidor es un motivador y la ausencia de un motivador es un inhibidor. Creemos que no es el enfoque adecuado, pues los motivadores tiene un carácter personal, mientras que los inhibidores indican carencias en los procesos de desarrollo y en la infraestructura de reuso. Están relacionados pero en diferentes niveles, primero es necesario neutralizar los inhibidores para disponer de una base adecuada y luego implementar un programa de incentivos que potencien los beneficios.

INHIBIDORES TECNICOS

- **estado inmaduro de la tecnología**
  las metodologías propuestas son muy generales, los modelos de estimación de costo - beneficio están en sus inicios, los modelos estándares de componente y composición recién están apareciendo.
- **falta de metodologías de desarrollo bien definidas para el reuso**
  existen lineamientos generales para la incorporación del reuso al proceso de desarrollo y reportes de experiencias, pero falta el salto a la sistematización de la tecnología de reuso en las metodologías.

- **conflicto de las actividades de reuso con la metodología estándar de la organización**
  los cambios en la metodología de trabajo pueden causar oposición.
- **falta de estándares**
  la carencia de estándares generales (terminología, construcción y manejo de componentes, administración de proyectos con y para reuso, metodologías de desarrollo, etc.) y estándares específicos para diferentes dominios dificulta la puesta en práctica de la actividad del reuso.
- **bibliotecas deficientes**
  información incompleta del catálogo, esquemas de clasificación inadecuados, falta de automatización, componentes deficientes, carencia de los componentes requeridos, falta de compatibilidad entre componentes.
- **falta de actualización de la biblioteca**
  en relación a los avances de la tecnología dentro del dominio.
- **falta de métricas estándar para el reuso**
  que permitan realizar una evaluación precisa de los beneficios económicos reales, cuantificar la actividad de reuso y determinar niveles de las características de los componentes como reusabilidad, calidad y mantenibilidad.
- **reuso sólo de bajo nivel**
  para obtener beneficios altamente significativos no podemos limitar el reuso a componentes de código.
- **integración inapropiada de los componentes**
  para evitar esto, ayuda mucho la existencia de una arquitectura genérica que identifique las interfaces de los componentes y sus formas y lugares de integración.
- **falta de herramientas que soporten el proceso de reuso**
  se carece de estándares y orientaciones sobre qué servicios debe proporcionar una herramienta automatizada de reuso, cómo deben estar estructurados y cómo debe comunicarse con otro tipo de herramientas.

INHIBIDORES ORGANIZACIONALES

- **falta de un programa de incentivos**
  si no se proporciona incentivos concretos para la actividad de reuso, no se estará potenciando los beneficios esperados.
- **visión a corto plazo**
  los administradores están orientados a objetivos a corto plazo, como son minimización de costos, riesgos y de tiempos de desarrollo. Esto es contradictorio con la implantación de un programa de reuso, pues inicialmente requiere recursos, riesgos y tiempos adicionales.
- **falta de capacitación y entrenamiento**
  en estrategias y técnicas de reuso para la incorporación e identificación de las oportunidades de reuso en el desarrollo de software.
- **débil percepción de las ventajas del reuso**
  el reuso más que proporcionar beneficios es un elemento "crítico para asegurar la viabilidad comercial a largo plazo de las empresas de software". [Bell 93:3]

- **falta de políticas de adquisición de componentes**
  para la evaluación de los factores legales, económicos y técnicos.
- **aumento de costos**
  por la adquisición de componentes con costos de licencia y contratos de mantenimiento.
- **falta de modelos financieros para el reuso**
  que muestren un análisis de costo - beneficio y de retorno de inversión que soporte inversión de multiproyectos, pues el beneficio económico sólo se percibe en los múltiples reusos de los componentes.
- **falta de modelos de precios de transferencia**
  cuando se transfiere un componente de un proyecto a otro (reuso), el costo en ese momento debe ser determinado con base en su historia y en su uso futuro previsto. Tampoco existen modelos que representen la depreciación de los componentes.
- **falta de modelos de estimación de costos y tiempo**
  para la planificación de la producción incorporando la tecnología de reuso.

INHIBIDORES CULTURALES

- **falta de confianza en los componentes**
  que fueron creados por otros. A largo plazo, el mayor conocimiento y experiencia ayudará a superar este prejuicio.
- **falta de confianza en los métodos de reuso**
  esto se puede abordar con el desarrollo de estándares, bibliotecas y herramientas adecuadas. También es muy importante la evaluación de los productos resultantes y de los beneficios globales.
- **resistencia del personal técnico ante nuevos conceptos y metodologías**
  en general están más interesados en ver un producto trabajando que en construir la infraestructura adecuada para el proceso de reuso. Hace falta un desplazamiento de enfoque y debe ser conducido por la capacitación en el tema.

4.5 Aspectos Económicos

Muchos son los beneficios que se esperan de la actividad del reuso de software. Desde el punto de vista técnico es relativamente fácil internalizar la conveniencia y la necesidad de su incorporación al proceso de desarrollo de software. Sin embargo, el personal técnico no controla el manejo de recursos y en general no tiene la decisión final sobre la inversión económica necesaria para instaurar un programa de reuso.

Es necesario traducir los beneficios esperados a términos financieros. Esta estimación es crucial para conseguir el apoyo de los diferentes niveles administrativos y gerenciales involucrados en la toma de decisiones para el desarrollo de sistemas. Pero esto es sólo el comienzo, más importante aún es disponer de un modelo de análisis económico que cuantifique los costos y beneficios del reuso durante el proceso y ayude a evaluar y justificar la permanencia de su incorporación.

Es obvio que un "... programa de reuso es económicamente factible si sus beneficios exceden sus costos" [Lim 92:1]. Sin embargo, no es tan obvio cuáles son todos los costos y beneficios involucrados en el proceso y mucho menos, cuáles son las métricas adecuadas para cuantificarlos. Además, se necesita un modelo económico que integre estos elementos y sus relaciones para determinar la conveniencia de la inversión y los beneficios netos a obtener. Un aspecto que incrementa la dificultad del modelo es que los beneficios del reuso se obtienen precisamente en la medida que se reutilizan los componentes y estos abarcan diferentes desarrollos, es decir, se necesita un modelo multiproyecto para una estimación adecuada de los beneficios reales y que permita distribuir el costo de los componentes reutilizables entre los diferentes proyectos.

De esta manera, tenemos tres actores principales en el aspecto económico del reuso: identificación de costos y beneficios, métricas para cuantificarlos y un modelo que integre estas métricas y establezca relaciones entre ellas. El tema da para mucho, pero aquí nos enfocaremos a la identificación de los costos involucrados en el proceso de reuso. Este es el punto de partida tanto para establecer como para evaluar modelos económicos de reuso.

4.5.1 Costos involucrados en el proceso de reuso

i. Costos de institucionalización del reuso

Nos referimos a toda la inversión inicial necesaria para implementar un programa de reuso. Es fundamental la ***capacitación*** inicial intensiva en todos los aspectos del reuso de software, esta capacitación debe abarcar dos enfoques dependiendo del tipo de personal, uno técnico y otro administrativo o gerencial. Otro costo básico es la ***infraestructura*** para la actividad del reuso, formada por: los componentes reutilizables, un depósito (biblioteca) para almacenarlos, mecanismos adecuados de clasificación, búsqueda y recuperación, herramientas automatizadas para la actualización y acceso del depósito, representación adecuada del componente en sus diferentes niveles (análisis, diseño, código, pruebas, versiones) y finalmente, políticas de administración y uso del depósito. Por último, tenemos los costos de ***implementación*** que abarca la definición de políticas y procesos de reuso a nivel técnico y administrativo, la definición o adopción de estándares de reuso, la definición de roles y perfiles y la definición de modelos de evaluación.

La mayor parte de esta inversión inicial se hace sólo una vez, aunque todos los aspectos evolucionan y maduran con el tiempo, por lo que requerirán de costos adicionales. El elemento que más debería sufrir modificaciones con el tiempo es la infraestructura de reuso por la continua renovación, construcción y adquisición de componentes y herramientas. Los costos de capacitación y mantenimiento del programa global de reuso deben considerarse como costos adicionales de un proyecto cada vez que incorpore el reuso.

ii. Costos operacionales

Son los costos que implica mantener funcionando la infraestructura de reuso. En primer lugar tenemos los costos de *incorporación* de componentes al depósito, pues

requerirá de una clasificación y probablemente de documentación adicional, así como adaptación al formato de presentación. Luego se presentan los costos de *mantenimiento* de los componentes, bien sea corrigiendo errores (correctivo) o mejorando su desempeño y agregando funciones (evolutivo).

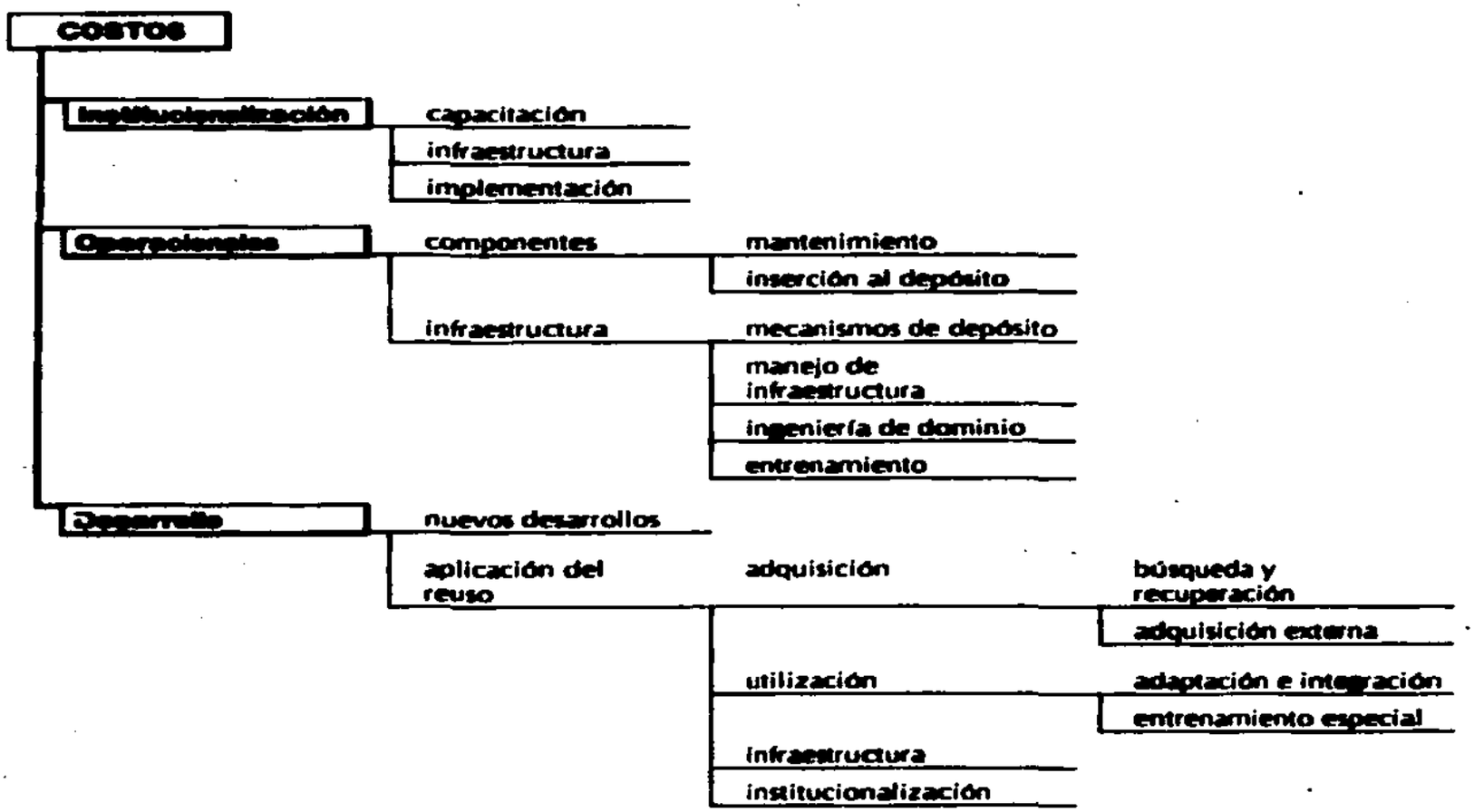


Fig. 1-6 *Costos involucrados en el proceso de reuso*

Menos frecuente serán los costos invertidos en la infraestructura del reuso, como son: a) *mecanismos para el depósito:* costos de hardware para almacenar una biblioteca de componentes, el costo de los equipos para accesar fuentes externas, el costo de herramientas para buscar, recuperar, manipular, comprender e integrar los componentes; b) *manejo de la infraestructura de reuso:* costos de mantenimiento y soporte de herramientas, costos de monitoreo y realimentación del uso de los componentes; c) *ingeniería de dominio:* siempre y cuando no se asocie como costo de un proyecto particular; d) *entrenamiento:* en nuevos componentes y herramientas.

iii. Costos de desarrollo de nuevos sistemas

En el desarrollo de sistemas que incorporan el reuso, podemos identificar costos de nuevos desarrollos y costos de aplicar el reuso. Cuando se utiliza el reuso se incurre en costos de:

- *adquisición:* búsqueda y recuperación, adquisición externa (costo, licencia, mantenimiento).
- *utilización:* adaptación e integración (abarca los costos de comprensión), entrenamiento especial por la complejidad del componente (generadores de reportes, sistemas de ventanas, hojas de cálculo, etc.).
- *infraestructura e institucionalización:* cada reutilización debe asumir una cuota de estos costos.

Si se construyen componentes reutilizables disminuyen los costos específicos del proyecto y aumentan los costos totales. Si no se construyen o adquieren componentes adicionales y se reutilizan los existentes, disminuyen los costos totales, pero no se potencia el reuso futuro, limitando el ahorro a largo plazo.

iv. Costos Comparativos

Finalmente, un modelo económico debe disponer de costos comparativos con los cuales cotejar las ventajas económicas reales, es decir, medidas de productividad en términos de tiempos de desarrollos, capacitación, cantidad y calidad de productos construidos, esfuerzo de mantenimiento, etcétera.

En algunas experiencias [REW 93:115] se ha reportado que el costo de reutilizar un componente sin modificación es de 20% del costo de desarrollar uno nuevo, mientras que reutilizar componentes con modificaciones implica un costo de 85%. Además, menciona que construir un componente reutilizable cuesta el triple que desarrollar un componente no reutilizable.

4.5.2 Consideraciones adicionales

Es notoria la cantidad de variables que deben contemplarse en un modelo de análisis económico de reuso, lo que en parte explica la carencia de modelos adecuados y su estandarización. Vale decir que la necesidad de medidas tan específicas se debe a la exigencia de justificar financieramente la adopción de un programa de reuso. Sin embargo, a medida que el reuso de software se adopte como una práctica normal en el desarrollo de sistemas se relajará este requisito y será más sencilla su evaluación

Por otro lado, esta rigurosidad de análisis se justifica cuando se intenta tener niveles de reuso altamente significativos y hay una inversión considerable de por medio. Adicionalmente, aquí hemos supuesto que el costo de cuantificar los costos es insignificante en relación a la actividad de reuso. Es decir, estamos hablando de componentes de granularidad mayor que funciones de biblioteca y de una infraestructura relativamente grande. Si éstas no son las condiciones de implantación del programa de reuso, las consideraciones para el modelo económico exceden las necesidades y no deben aplicarse en su totalidad.

Además de carencia de modelos económicos también hay carencia de métricas estándares para el reuso. A la hora de contemplar alguna métrica debemos tomar en cuenta que debe ser una medida repetible, debe reflejar precisamente algún aspecto, debe ser significativa y sobre todo fácil de obtener y de implementar de una manera uniforme. Por último, debe ser lo más general posible para ir construyendo un método estándar para cuantificar el reuso.

4.6 Aspectos Legales

Ciertamente existen normativas en relación a la protección de los derechos sobre el software, pero las nuevas actividades y procesos específicos del reuso de software requieren ampliar su alcance. Podemos visualizar las implicaciones legales desde varios puntos de vista, pero sin duda los más relevantes son los que tienen que ver con los derechos de propiedad y con las responsabilidades legales.

Los derechos de propiedad es un tema difícil porque pareciera que las cosas se reinventan o redescubren constantemente. Otro elemento escabroso es la construcción de componentes reutilizables utilizando otros componentes reutilizables, ¿cuál es el grado de propiedad por el sólo hecho de ensamblar, o de ensamblar y modificar?. Las protecciones de propiedad intelectual deben diseñar mecanismos que permitan recompensar a quienes modifiquen y agreguen valor a los componentes existentes, para poder avanzar en el desarrollo del reuso de software.

En cuanto a la responsabilidad legal, debe existir una normatividad que contemple la responsabilidad, garantías e indemnizaciones del fabricante cuando el componente presente un mal funcionamiento o deficiencias en el desempeño y debe cuantificarse esta responsabilidad en relación a cuánto daño ha causado en el sistema que lo utiliza.

Los aspectos legales cobran mucha mayor importancia en la medida en que los componentes presentan una granularidad mayor, pues implica una mayor inversión para su fabricación, un mayor ahorro/costo con su reuso y un mayor beneficio con su venta y distribución. En cualquier caso, es necesario cuidar que la normatividad no sea demasiado compleja, pues se convertiría en un inhibidor significativo.

5. Bibliotecas de componentes reutilizables

El reuso por composición está basado en la existencia de una colección de componentes organizada en una biblioteca. En este contexto, hablaremos de qué debe hacer una biblioteca, en qué consiste el proceso de incorporación de componentes al depósito y cuáles son las funciones básicas que deben realizar las herramientas de la biblioteca.

En el nivel más general, una ***biblioteca*** o ***depósito*** es una organización estructurada de software reutilizable, herramientas, procedimientos y personal encargado de su funcionamiento. El objetivo fundamental de esta organización es propiciar y facilitar la

construcción y uso de artefactos reutilizables para satisfacer metas concretas de productividad, costos y propiedades técnicas de la construcción de sistemas.

Son muchos los aspectos relacionados con una biblioteca: modelos de representación, tipos de componentes, esquemas de clasificación, mecanismos de búsqueda, interoperabilidad, costos, métricas, etcétera. En este momento nos interesa proporcionar una visión de los elementos operativos fundamentales que deben tenerse en cuenta en la construcción de un depósito de componentes.

El punto de partida es la definición de los objetivos y la delimitación de los alcances que se desea cubrir. Los dos procesos operativos principales de una biblioteca, son la incorporación de componentes y la clasificación, búsqueda y recuperación. Finalmente, mencionamos algunas demandas funcionales que deben satisfacer las herramientas de la biblioteca desde la perspectiva del usuario final y de los administradores del depósito.

5.1 Objetivos de una biblioteca

Una biblioteca de software reutilizable forma parte de la infraestructura necesaria para poder implementar un programa de reuso. Por lo tanto, es un auxiliar en la consecusión de sus beneficios. En este contexto, los objetivos de la biblioteca deben contribuir a la realización de los objetivos generales del programa de reuso. Para lograr este propósito, el proyecto de la biblioteca debe contener la especificación de objetivos específicos realizables con actividades concretas y medibles en la práctica. Algunos de los objetivos específicos que podrían ser comunes a cualquier biblioteca son:

OBJETIVOS DE UNA BIBLIOTECA

Asegurar la relevancia del acervo

- los productos reutilizables deben:
  - corresponder a los requerimientos de desarrollo de software de la institución
  - implicar un ahorro significativo en el esfuerzo de desarrollo
  - tener una calidad conocida y medida
  - ser fáciles de encontrar y obtener
- la colección de componentes debe ser lo suficientemente amplia para cubrir las demandas e inclusive proporcionar alternativas para un mismo requerimiento

Garantizar la operación de la biblioteca

- minimizar el costo de operación
- proporcionar asistencia al usuario para el uso de los componentes, de la biblioteca y para identificar las oportunidades de reuso
- medir la utilidad de la biblioteca en cuanto a procedimientos, herramientas, componentes y soporte

Reportar resultados y justificar su existencia

- medir el costo-beneficio de cada componente
- recoger y reportar los beneficios logrados por proyecto

Probablemente no se aborden todos los objetivos desde los comienzos operativos de la biblioteca, si es así, debe quedar claro cuáles se abordan desde un principio y en que momento se abordarán los restantes. En cualquier caso, antes de poner en funcionamiento la biblioteca, el personal encargado debe definir:

- objetivos generales y específicos
- actividades concretas para realizar los objetivos específicos
- criterios y orientaciones para el desarrollo de componentes reutilizables
- criterios y métricas para evaluar los componentes
- criterios y métricas para medir y evaluar el reuso y la utilidad de la biblioteca
- características de las herramientas de soporte automatizado y construirlas o adquirirlas
- procedimientos para el acceso a la biblioteca (solicitud, incorporación, búsqueda y recuperación de componentes)
- procedimientos para la administración de la biblioteca
- estrategias para la promoción del reuso
- estrategia incremental operativa de la biblioteca
- perfil del personal encargado

Si los recursos son limitados, lo más conveniente será iniciar con una estrategia incremental operativa, donde se prestará una funcionalidad restringida inicial que irá incrementando con la evolución de la biblioteca. Es fundamental considerar la preparación del personal encargado, pues para potenciar el reuso debe ayudar a identificar las oportunidades de reuso. Para esto, requiere tener conocimiento de las metodologías de diseño, de los lenguajes de programación utilizados en el desarrollo, de técnicas de ingeniería de software en general y por supuesto, de la biblioteca misma.

En general, todas las actividades operativas de la biblioteca implicará la elaboración de informes o reportes, donde al menos, la información cuantitativa debe ser generada automáticamente. Este manejo implica a su vez el registro (automático donde sea posible) de toda la información posible, resultados de búsquedas, uso de cada componente, usuarios involucrados, reportes de problemas, evolución y versiones de los componentes, costos involucrados, etcétera. La información a ser registrada depende en gran medida de las dimensiones del programa de reuso y de los objetivos planteados.

5.2 Proceso de incorporación de componentes

El ingreso de componentes al depósito debe seguir un proceso que garantice un control mínimo de sus cualidades. En particular, los componentes deben tener un conjunto de información mínimo y deben pasar por un proceso de evaluación que disminuya la posibilidad de errores y promueva la confianza en el uso de los mismos. En el siguiente cuadro presentamos de manera sinóptica las fases y tareas del proceso de incorporación de componentes, para luego describirlas con detalle.

PROCESO DE INCORPORACION

Proponer un nuevo componente

- verificar que el componente no existe en la biblioteca
- verificar que el componente no ha sido propuesto, rechazado o aprobado
- identificar el tipo de solicitud: elaborado por el proponente, para desarrollar, para adquirir
- proporcionar la información mínima requerida por la biblioteca
- clasificar el componente propuesto

Hacer una evaluación inicial

- determinar los requerimientos para reutilizar el componente
- determinar la conveniencia de modificar las características del componente propuesto para maximizar su reusabilidad o para acelerar su construcción. Si hay modificaciones, hacerlas junto con el proponente
- hacer una estimación de costo / beneficio
- determinar la demanda probable
- rechazar el componente propuesto o determinar si puede continuar el proceso de incorporación
- registrar los resultados de la evaluación inicial
- si el componente no está construido, desarrollarlo o adquirirlo

Hacer una evaluación detallada

- evaluar la calidad del componente y verificar que satisfaga el criterio mínimo de aceptación
- identificar las deficiencias y fallas y definir orientaciones para la evolución del componente
- determinar las diferencias entre el componente y los estándares para la construcción de componentes de la biblioteca
- determinar la completez del material reutilizable
- asignar una puntuación de calidad y reusabilidad
- registrar los resultados de la evaluación detallada

Incorporar el componente

- clasificar el componente
- ensamblar toda la información recabada bajo el formato demandado por la herramienta de la biblioteca
- almacenar el componente
- dar de alta el componente para que esté disponible a todos los usuarios
- notificar al usuario proponente y a todos los que mostraron interés en este componente o en alguno relacionado

5.2.1 Propuestas de nuevos componentes

Se espera que los usuarios de la biblioteca propongan nuevos componentes para su incorporación al depósito, en algunos casos construidos por ellos mismos y en otros solicitudes para adquirir o desarrollar. En cualquier caso, la propuesta debe ir acompañada de un conjunto de información en algún formato preestablecido que permita realizar una evaluación preliminar. La información más importante es una documentación completa y una justificación que destaque la cantidad de esfuerzo que se ahorra con su reuso y las posibilidades de reuso del componente en proyectos actuales y futuros.

Cada una de las propuestas de componentes se debe registrar para que cada nueva propuesta se pueda cotejar con las existentes, pues se puede haber rechazado una propuesta similar o estar en vías de aceptación, compra o desarrollo. Para poder realizar esto, es necesario clasificar cada componente propuesto aún cuando finalmente no sea aceptado, ya que la comparación se hará buscando el componente entre las propuestas. En cada registro se debe almacenar la información relevante del componente (descripción funcional, fuentes de origen o disponibilidad, justificación, nombre, palabras claves, etc.) y del proponente (nombre, cargo, proyecto para su uso inmediato, proyectos para su uso futuro, etc.).

5.2.2 Evaluación y aceptación inicial

El personal de la biblioteca debe hacer una evaluación cualitativa y cuantitativa del componente propuesto, esto es, analizar sus características y oportunidades de reuso. Esta evaluación incluye determinar los requerimientos para su reutilización, determinar la conveniencia de modificar las características del componente propuesto para cubrir las necesidades de otros o futuros proyectos, hacer una estimación de costos y beneficios y determinar la demanda probable.

Los resultados de esta evaluación pueden conducir al personal de la biblioteca a un rechazo de la propuesta o a una aceptación inicial. Es importante que los resultados de estas evaluaciones sean registrados junto con la información del componente para futuros accesos. Por otro lado, los resultados se deben revisar con el usuario proponente para refinar los requerimientos de la propuesta y en algunos casos limitar los requerimientos a las posibilidades de su realización concreta. En el caso de que el componente propuesto no esté construido, el personal de la biblioteca utilizará la información provista para establecer prioridades de demanda del producto para su adquisición o desarrollo.

5.2.3 Evaluación detallada

Todos los componentes del depósito deben tener una calidad conocida, pero no siempre es conveniente demandar el máximo de calidad para todos porque podría significar un obstáculo para la ampliación de la cantidad de componentes. En realidad, lo deseable es que exista un criterio mínimo de aceptación basado principalmente en:

- el costo de reuso debe ser menor que el costo de desarrollar uno nuevo
- los requerimientos deben ser mayores que los normales, pues las deficiencias y fallas se propagan con los múltiples reusos
- la documentación debe ser suficiente (descripción, instrucciones, etc.)

Por otra parte, las características óptimas de calidad en general se obtendrán con la evolución del depósito y sus componentes a través de la realimentación de las experiencias prácticas. Hay que tomar en cuenta que iniciar una biblioteca con artefactos certificados de muy alta calidad incide en costos elevados.

Así, tenemos que el papel del personal de la biblioteca no es limitar la aceptación a componentes de máxima calidad, más bien, tiene la responsabilidad de identificar las deficiencias e inclusive fallas. Esto servirá para hacer recomendaciones específicas sobre el trabajo adicional que se requiere y así, se definen las orientaciones para la evolución de los artefactos. Lo importante de esto es que:

- un defecto o una característica débil de desempeño, no son razón suficiente para rechazar la incorporación de un componente
- los usuarios de la biblioteca deben tener información exacta del estado en que se encuentra el componente a reutilizar

Por otro lado, la biblioteca debe tener un conjunto de criterios y métricas de evaluación y orientaciones generales para el desarrollo de componentes reutilizables. Estos requisitos no siempre se pueden satisfacer, en algunos casos porque los componentes son adquiridos de terceros o el componente fue desarrollado internamente sin seguir estos lineamientos (p.e., porque fue construido antes de que existieran los lineamientos) y se requiere reutilizarlo. Si un componente no satisface estos requerimientos, no necesariamente debe ser rechazado, y si es aceptado, estas desviaciones deben ser registradas en su información global.

Adicionalmente, debe examinarse la completez del material reutilizable para asegurar que el reutilizador dispone de todos los elementos para encontrar, comprender, instalar y probar el componente. En general un componente debe contener: a) resumen; b) descripción detallada; c) instrucciones de instalación, prueba y operación; d) material reutilizable (el componente mismo); e) criterios y escenarios de prueba, objetivos y resultados.

Pero eso no es todo, es conveniente que el personal de la biblioteca asigne una puntuación acerca de la calidad y la reusabilidad general del componente, en una escala simple con pocos valores. Ciertamente esta es una actividad sumamente subjetiva, pero es muy ilustrativo para el usuario. Los argumentos para esta puntuación deben ser expuestos en una justificación que resalte las ventajas y deficiencias del artefacto.

Finalmente, todos los resultados de la evaluación deben ser registrados y esto debe ser hecho utilizando un formato consistente. Esta información debe ser actualizada cada vez que el componente se modifique y las distintas actualizaciones deben indicar la fecha y responsable como mínimo.

5.2.4 Incorporación

Una vez que el componente se ha evaluado y se ha completado su documentación, se procede a su almacenamiento en la biblioteca. En primer lugar, se debe clasificar el artefacto para posibilitar su búsqueda. El proponente debe hacer una clasificación inicial asistido por un administrador del repositorio. La asistencia se convertirá en revisión cuando el esquema de clasificación sea familiar a los usuarios.

Posteriormente se procede al almacenamiento físico (o por referencia) para posibilitar su recuperación. La primera tarea a realizar aquí es empacar toda la información acerca del componente (documentación, evaluación, recomendaciones) y todas las partes y formas del componente mismo (requerimientos, diseño, código, pruebas, etc.). En principio, todo este paquete debe serle entregado al usuario cuando solicite la recuperación del componente. Cuando se trate componentes externos es posible que el personal de la biblioteca tenga que acomodar la información disponible al formato preestablecido para uniformizar la búsqueda, acceso y presentación.

Se introduce toda la información adicional disponible que el personal de la biblioteca recabó y/o generó. Debe existir un mínimo de información obligatoria para cada componente y la herramienta debe garantizar su existencia, así como puede presentar una medida que indique de cuánta información se dispone en relación al total posible. Finalmente, debe ser agregado al catálogo de componentes y notificar a todos los usuarios que han mostrado interés en componentes relacionados.

De esta manera, una vez que se tenga ensamblada toda la información del componente, éste se puede poner a disposición de todos los usuarios del depósito. Es importante asegurar que todo el paquete esté disponible, probablemente un componente de análisis o diseño no estará disponible en forma electrónica y la biblioteca debe asegurar una rápida entrega del material, por ejemplo, teniendo ya copias impresas.

5.2.5 Información de un componente

Durante el proceso de incorporación se genera y ensambla un conjunto de información para cada componente. Adicionalmente, existen otras características que deben formar parte de la información detallada y que son el resultado del uso y evolución del componente en el depósito.

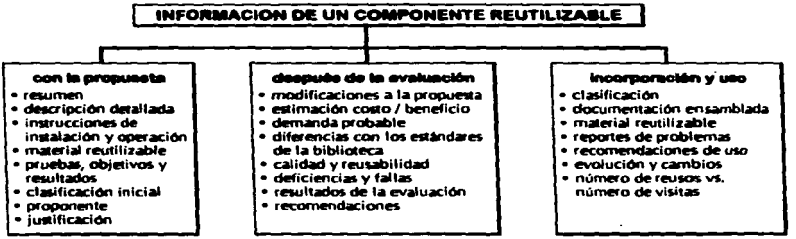


Fig. 1-7 Información de un componente reutilizable

5.3 Herramientas de la biblioteca

Desde el inicio operativo de la biblioteca es imprescindible el uso de herramientas automatizadas. Las herramientas se diferencian principalmente por la tecnología subyacente en su construcción y/o por el tipo y cantidad de funciones que realizan. En cualquier caso, las herramientas deben satisfacer un mínimo de requisitos.

En el nivel más general, los procedimientos deben ser simples de utilizar, debe proporcionar ayuda general y detallada en línea para cualquier operación y debe tener un manual de usuario detallado. Las funciones mínimas que debe realizar son la clasificación, búsqueda y recuperación de componentes, pero es muy deseable que realice otras funciones, como registrar y difundir la información relevante y apoyar la promoción del reuso.

Podemos distinguir dos usuarios diferentes de las herramientas: administradores y usuarios finales. En este contexto, identificaremos algunas funciones importantes que la herramienta debe soportar para cada tipo de usuario.

5.3.1 Funciones para los administradores

FUNCIONES PARA LOS ADMINISTRADORES

Clasificación de componentes

El método de clasificación soportado por la herramienta debe ser fácil de utilizar, es decir, debe ser sencillo asociar una clasificación a un componente y debe permitir una búsqueda rápida basada en la clasificación. Es deseable que utilice un manejador de bases de datos para almacenar la información y permitir búsquedas flexibles. Es importante que soporte el mantenimiento y actualización del mecanismo de clasificación.

Registro de problemas y modificaciones

La biblioteca debe registrar en una bitácora los problemas, errores y deficiencias reportados para difundir a todos los usuarios esta información y para definir la evolución de los componentes. También debe registrar en una bitácora las modificaciones hechas a los artefactos y establecer el vínculo entre ambas bitácoras, de manera que se pueda saber si un problema ya fue solucionado y a que modificación corresponde y viceversa. Puede ser importante registrar las dependencias entre componentes, de manera que cuando se encuentra un error en un componente, se pueda notificar a todos los usuarios que utilizaron y/o van a utilizar componentes que dependen de él.

Registro de usuarios y proyectos

Para que un usuario pueda utilizar la biblioteca debe darse de alta. En este registro debe proporcionar sus datos personales, datos de ubicación y proyecto asociado.

Esta información se utilizará para notificar nuevas versiones del componente, para reportes de problemas y para obtener realimentación acerca de los artefactos y los servicios de la biblioteca.

Generación de reportes de acceso y estado de los componentes

Para poder generar reportes, primero tiene que soportar actividades de registro sobre el uso de la biblioteca. En particular debe poder informar acerca de:

- *información de uso de un componente*. Determina qué usuario extrajo el componente, cuándo, para qué uso y en qué proyecto. También es importante que se registre el número de veces que se ha visitado el componente *versus* el número de veces que ha sido reutilizado.
- *información de usuario*. Nombre, número de teléfono, dirección de correo postal, e-mail, cargo y proyecto asociado para cada usuario de la biblioteca.
- *información de la evolución del componente*. Fechas y autores de cambios, relación entre los cambios y los problemas reportados, información de realimentación, dependencias con otros componentes.
- *información de problemas reportados*.

Control de versiones

Debe mantener un registro de cada versión del componente y de los usuarios de cada una de las versiones, controlar la solución de los problemas reportados y llevar un registro histórico de la evolución.

5.3.2 Funciones para los usuarios finales

FUNCIONES PARA LOS USUARIOS FINALES

La función fundamental que requiere el usuario es una búsqueda y recuperación efectiva. Esto reducirá el tiempo, costo y la dificultad de identificar y obtener los artefactos reutilizables.

Especificación de búsqueda sencilla

El reutilizador debe ser capaz de describir los requerimientos del componente buscado sin aprender un nuevo lenguaje.

Refinamiento iterativo de la búsqueda

El sistema debe permitir y asistir al reutilizador para cambiar la especificación de la búsqueda basada en resultados anteriores.

Información detallada de los candidatos

Resúmenes, evaluaciones de calidad, instrucciones de operación, problemas, recomendaciones, etcétera.

Recuperación del componente

Para la recuperación se debe entregar todo el paquete que conforma el componente, si es posible todo en forma electrónica. Un componente puede tener dependencias con otros componentes y la herramienta debe ayudar a extraer todos los componentes necesarios para su funcionamiento de una manera automática, o por lo menos notificar al usuario cuáles son esas dependencias.

6. Incorporación del reuso

A nuestro parecer, esta estructuración del conocimiento es adecuada para entender las implicaciones del reuso de software y puede servir como base para la definición de un programa de reuso. Existen dos criterios básicos con los que se debe filtrar esta información para su incorporación en una organización:

- **el peso específico de cada elemento en el contexto de su incorporación.** No todos los elementos descritos tienen la misma importancia para todas las organizaciones, por ejemplo:
  - si la infraestructura de reuso es únicamente para uso interno, los aspectos legales revisten poca importancia.
  - los elementos técnicos dependen en mucho de la infraestructura de desarrollo que ya exista, las herramientas de desarrollo y programación, el personal existente, etcétera.
  - las actividades de promoción del reuso para su incorporación pueden ser menores si ya existe una disposición positiva a nivel técnico y organizacional.
- **el tamaño de la organización.** La información presentada, por lo general asume organizaciones e infraestructuras de desarrollo grandes. Sin embargo, las organizaciones de menor tamaño también pueden alcanzar beneficios significativos si disciernen los elementos pertinentes a su estructura, por ejemplo:
  - no requieren utilizar complejos modelos económicos, porque las inversiones son mucho menores y el costo de llevar un análisis costo - beneficio detallado puede ser innecesario.
  - las implicaciones legales por lo general son menores.
  - los riesgos son menores por tener una estructura organizativa de menor tamaño.
  - deben centrarse en la infraestructura técnica de reuso (componentes, bibliotecas, etcétera).
  - los programas de incentivos pueden ser simples.

Lo invariable es que para cada organización la incorporación del reuso será diferente y en la medida que así se entienda mayores serán los beneficios que puede alcanzar.

Página intencionalmente en blanco

El paradigma OO y el reuso de software

Es común encontrar en la literatura frases como: "Entre las muchas razones porque la gente adopta los métodos orientados a objetos está la reputación que tiene para mejorar el reuso de software." [Lea+ 92:2]. Esta afirmación es prácticamente un lugar común entre los beneficios que se promueven para el paradigma orientado a objetos. Extrañamente, no existe mucha documentación que explique y justifique tal reputación. Creemos que existen criterios técnicos con los que se puede justificar esta relación del modelo OO con el reuso de software. En particular, nos interesa explorar vinculación del paradigma con el reuso de código. Sobre esto trata este capítulo.

Para abordar el problema, ilustramos la relación que tiene el modelo orientado a objetos con el reuso de software a través del estudio de los conceptos y mecanismos de programación que abarca. El marco de referencia para este desarrollo tiene su centro en la interconección de componentes y está plasmado en un modelo de composición que unifica la exposición. Concretamente, se plantean una serie de problemas para un modelo de composición en relación al reuso de software y analizamos la influencia que tienen los conceptos y mecanismos del paradigma[1] OO en la solución de estos problemas. Se ha tomado el paradigma procedural como punto de partida para establecer comparaciones, con la finalidad de ilustrar desde una perspectiva evolutiva los aportes que hace el modelo de objetos a la reusabilidad.

1. Infraestructura de Reuso

La actividad del reuso está enmarcada en una metodología global de desarrollo de software y está sustentada por varios elementos que determinan una ***infraestructura de reuso***. Entre estos elementos, destacan los relacionados con el uso de un depósito de componentes reutilizables: una colección de componentes, un depósito para almacenarlos y diferentes subsistemas que soportan el uso del depósito (documentación, certificación, clasificación, almacenamiento, búsqueda y recuperación). Estos elementos están relacionados en función de proporcionar la entrada y uso de componentes del depósito a través del proceso ilustrado a continuación.

---

[1] "Un paradigma en el sentido de los paradigmas científicos que analiza Thomas Kuhn, es una manera de visualizar la realidad que sirve como base para toda una corriente de pensamiento científico. Respecto a la computación, un paradigma, entonces, es un modelo mental que involucra una conceptualización especial de qué es un problema y cómo representar el proceso de su solución por medio de la computadora." [Greiff 93b:10]

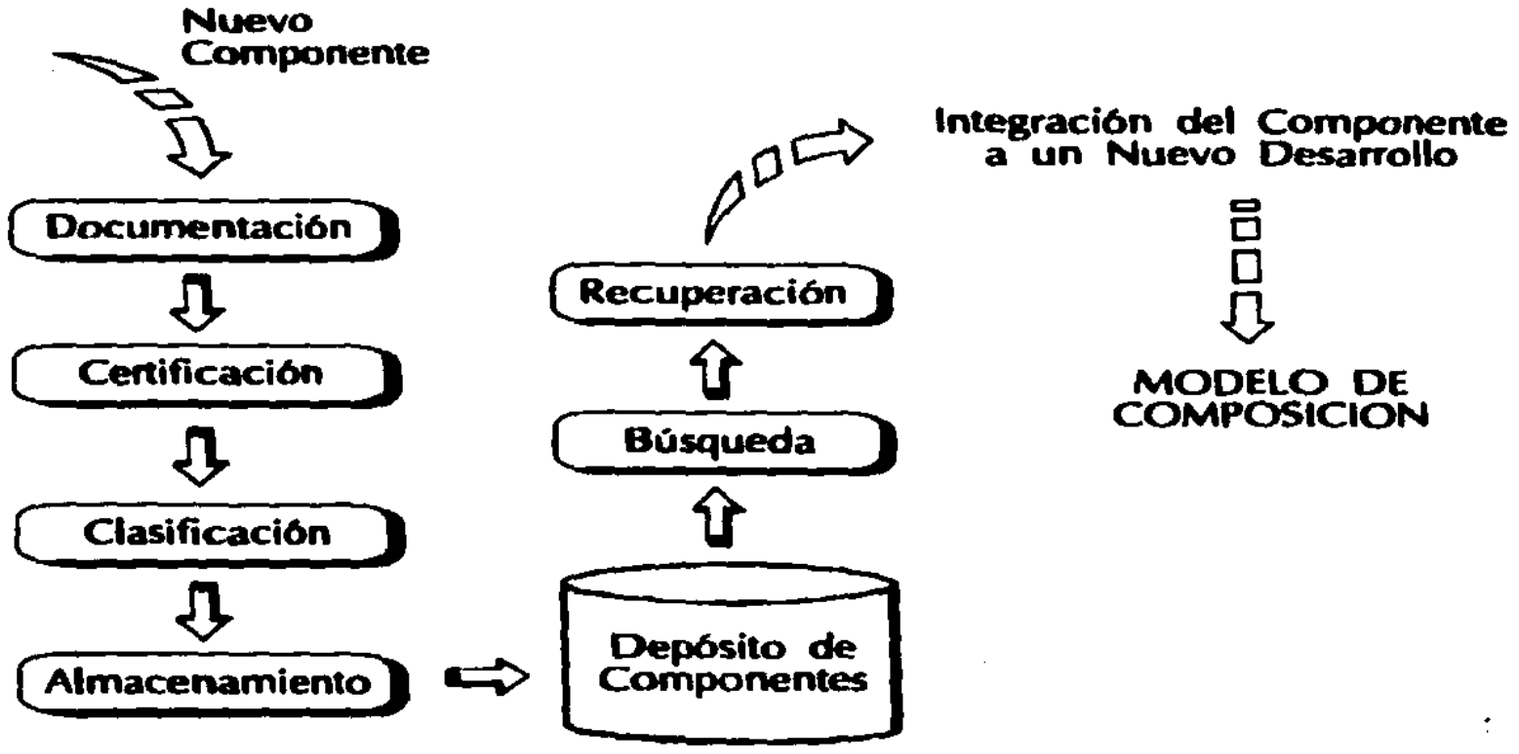


Fig. 2-1 *Infraestructura de reuso*

Si ya disponemos de esta infraestructura para almacenar y recuperar los componentes, el paso siguiente en la actividad de reuso es la incorporación de un componente reutilizable a un nuevo desarrollo. Para ello, necesitamos de ciertos mecanismos que nos permitan interconectar el componente. Estos mecanismos determinan las formas como se pueden integrar los componentes y conforman un ***modelo de composición***, que a nuestro entender tiene una gran influencia en los beneficios que se pueden obtener del reuso.

En el contexto que nos interesa, si queremos estudiar la relación que existe entre el modelo OO y el reuso de software, consideramos que un marco de análisis adecuado debe estar ligado a la noción de modelo de composición. Específicamente, nuestro marco de análisis va estar formado por una *generalización del modelo de composición* y por un conjunto de *problemas relativos a la reusabilidad* que debe abordar el modelo para soportar apropiadamente la interconexión de componentes.

Con este marco en mente, estudiaremos el ***modelo de composición OO*** en relación a los problemas planteados en el marco de análisis y evaluaremos las aportaciones que ofrece la programación OO a la resolución de estos problemas. También se analizará el *modelo de composición procedural* como un punto importante de comparación.

2. Modelo de Composición

La integración de componentes reusables a un nuevo desarrollo implica la existencia de un ***modelo de composición***, esto es, un conjunto de mecanismos que determinan las formas de interconexión de los componentes. A su vez, un modelo de composición implica la existencia de un ***modelo de componente***, que en esencia es un conjunto de características que especifican los servicios que proporciona el componente y el contexto en el que puede ser reutilizado.

Es difícil concebir un modelo de composición uniforme para cualquier tipo de componente reutilizable, dado que implicaría tener un *modelo general de componente* independiente de la granularidad, de la fase de desarrollo en la que se reutiliza y del paradigma bajo el cual fue concebido. Aún contando con este modelo general, sería necesario diseñar mecanismos de composición adecuados que también fueran independientes de cualquier tipo de representación y del tamaño de la abstracción. Por último, habría que modificar los elementos que sustentan el reuso de componentes, ya que tienen una fuerte dependencia del modelo de componente.

El problema del modelo de composición general podríamos abordarlo de dos maneras:

- ***Construcción incremental***. El primer paso sería abstraer un modelo de componente para cada producto reutilizable del proceso de desarrollo (componentes de requerimientos, diseño, arquitectura, documentación, código, pruebas, etc.), para luego construir un modelo de mayor nivel (un meta-modelo) que reúna y defina características comunes sobre los primeros modelos. Después, podríamos descubrir mecanismos de composición que se adapten a tal modelo de componente.
- ***Construcción impositiva***. Comienza por definir los mecanismos de composición que se desean, para, en una etapa posterior, construir un modelo de componente que se adapte a tal modelo de composición. Obviamente los mecanismos de composición que se definan deben ser realizables para cualquier tipo de componente.

En cualquier caso, esta tarea no es nada sencilla y se ve magnificada porque puede requerir la modificación y/o creación de nuevas metodologías de desarrollo que sustenten el reuso por composición utilizando tal modelo.

En lugar de abordar el problema de modelo de composición desde alguna de estas perspectivas, pensamos que es más conveniente relajar la generalidad del modelo y comenzar a estudiarlo desde un nivel menos complejo. En este sentido, creemos que un buen punto de partida sería la conceptualización de un modelo de composición en el nivel más básico: la interconexión de componentes de código.

En este nivel, un modelo de composición de código está constituido por conceptos y mecanismos de programación. Por ***concepto*** de programación entenderemos una construcción abstracta que permite modelar un problema y que genera un modelo que se puede mapear a un lenguaje de programación concreto a través de uno o varios

mecanismos de programación específicos. Por ejemplo, como concepto tenemos la encapsulación y podemos implementarla a través de la interfaz de un procedimiento, de una clase o de un módulo. En algunos casos el concepto se designa igual que el mecanismo, por ejemplo, herencia es un concepto y un mecanismo.

¿Cuáles son los mecanismos y conceptos que debe tener un modelo de composición de código?. Existe una amplia gama de estos mecanismos y conceptos, algunos exclusivos de un determinado paradigma de programación y otros comunes[2] a más de uno. Además, dentro de un mismo paradigma, los lenguajes de programación que lo soportan pueden implementar los conceptos de maneras distintas. En teoría, con estos elementos podríamos tener muchas combinaciones de posibilidades de composición. Sin embargo, debemos escoger una combinación adecuada para el modelo y para ello contamos con ciertos elementos que pueden orientarnos:

- el modelo de composición debe ser lo más general posible.
- un modelo de composición está ligado a un modelo de componente.
- el modelo de composición / componente debe poder utilizarse en el modelado del problema y poder mapearse a un lenguaje de programación concreto.

La generalidad nos pide ir más allá de un determinado lenguaje concreto, es decir, quisiéramos un modelo tan general que nos permita representar e interconectar componentes sin importar el lenguaje / paradigma bajo el que fueron construidos. Sin embargo, la necesidad de mapeo nos limita a nivel de paradigma, esto es, el modelo de composición de código debe sustentarse en los conceptos y mecanismos abstractos de un paradigma de programación. Particularizar este modelo a un lenguaje específico restringiría la generalidad del mismo.

¿Por qué no escoger los mejores conceptos y mecanismos de programación y combinarlos de manera que obtengamos el mejor modelo de composición?. En el desarrollo de una solución para un problema se selecciona un paradigma de programación adecuado para el problema en particular y esto automáticamente *determina* un modelo de composición y un modelo de componente. Los criterios para seleccionar un paradigma se basan en los beneficios que presenta y éstos varían básicamente en relación al tipo de problema. Por lo tanto, no se trata de seleccionar un modelo de composición que favorezca el reuso de código, sino que dicho modelo queda determinado por el paradigma seleccionado para la solución de un problema particular.

Fig. 2-2 *Un paradigma determina un modelo de composición y de componente.*

[2] Si consideramos los cuatro paradigmas básicos de programación [Graiff 93:37]: procedural, orientado a objetos, funcional y lógico, podemos mencionar a la encapsulación como un concepto común a varios paradigmas y a la herencia como exclusivo al paradigma orientado a objetos.

No es nuestra intención hacer una revisión detallada de los mecanismos y conceptos subyacentes en cada paradigma de programación, más bien nos interesa mostrar el modelo de composición / componente determinado por el modelo OO y su relación con el reuso de software desde el punto de vista de la composición de componentes de código, teniendo como marco de comparación el paradigma procedural. El por qué de este objetivo es lo que sigue.

2.1 ¿Por qué un modelo de composición OO?

2.1.1 ¿Por qué analizar la relación entre el paradigma OO y el reuso de software?

Desde hace más de una década aparece con fueza el modelo de objetos[3] y provoca un cambio en la manera como se desarrollan los sistemas de software, mostrando ventajas significativas sobre el paradigma procedural. La difusión ha sido tal, que existe un amplio acuerdo en torno a que el paradigma orientado a objetos es adecuado para una extensa gama de problemas debido a las múltiples ventajas que presenta. Así, cada vez más, el paradigma OO se selecciona para el desarrollo de sistemas y, en lo que nos concierne, recordemos que implícitamente se está seleccionando un modelo de composición y de componente de código que determina las posibilidades de interconexión de los artefactos reusables.

Por otro lado, entre las ventajas mejor documentadas del paradigma OO, que se presentan en la literatura, encontramos criterios de ingeniería de software tales como: simplificación en el manejo de la complejidad, generación de arquitecturas robustas, reducción de costos de mantenimiento, extensibilidad, etcétera. Adicionalmente, muchos autores también divulgan entre las ventajas la afinidad que tiene el paradigma OO con el reuso de software, pero extrañamente no presentan mayores justificaciones de tal afirmación[4], lo que dificulta la posibilidad de establecer y sistematizar metodologías y modelos que mejoren el reuso.

2.1.2 ¿Por qué el paradigma procedural como referencia?

La programación OO es el resultado de una evolución que tiene como uno de sus ancestros al paradigma procedural. Compartimos esta perspectiva evolutiva planteada por

---

[3] Tiene sus inicios a finales de los 60's con el lenguaje Simula 67, pero no es sino hasta mediados de los 80's que irrumpe con tal fuerza, tanto en el mercado como en la academia, que prácticamente ya no se concibe el desarrollo de sistemas sin agregar el calificativo *orientado a objetos*. Aunque el modelo presenta muchos beneficios, su auge se ha visto envuelto por un aura publicitaria y una sensación de moda, que lo ha hecho aparecer como la panacea para el desarrollo de sistemas complejos, lo que a nuestro juicio parece exagerado. Esta crítica no es nueva y podemos verla reflejada tempranamente en [King 89:24]: "Tengo un gato llamado *Trash*. Actualmente, parecería que si estuviera tratando de venderlo (al menos a un científico de computación), no debería hacer hincapié en que es amable con los humanos, autosuficiente y que vive principalmente de ratones de campo. Mas bien, tendría que argumentar que es orientado a objetos".

[4] Entre las primeras excepciones que documentaron esta afinidad podemos destacar a [Meyer 87] y a [Johnson+ 88] desde el punto de vista teórico, y a [Lewis+ 91, 92] desde el punto de vista práctico.

[Oktaba 93:40], según la cual los conceptos de los lenguajes de programación han evolucionado en niveles de abstracción, permitiendo "expresar el modelo computacional utilizando conceptos que reflejen las abstracciones del mundo real". En esta evolución los conceptos básicos de la programación OO (objeto, clase, herencia) se encuentran en los estadios superiores y se construyen a partir de las abstracciones logradas en los niveles inmediatos anteriores, entre éstos se hayan los conceptos de programación del paradigma procedural (subrutina, módulo). Esta evolución de conceptos se ve reflejada en una mayor riqueza del modelo de composición OO y una manera de visualizarlo es estableciendo una comparación de las posibilidades de interconexión con el paradigma procedural.

Por otro lado, al momento de propagarse el paradigma OO como una metología de desarrollo, se enfrenta al paradigma procedural que era el más utilizado. En la literatura se exponen los beneficios del primero en contraposición a las deficiencias del segundo, para justificar el cambio a la "nueva" tecnología. Adicionalmente, queremos mostrar que el incremento de beneficios se manifiesta también en las posibilidades que ofrece para el reuso de componentes.

2.2 Una generalización del modelo de composición

Independientemente del paradigma de programación empleado podemos visualizar un modelo de composición general:

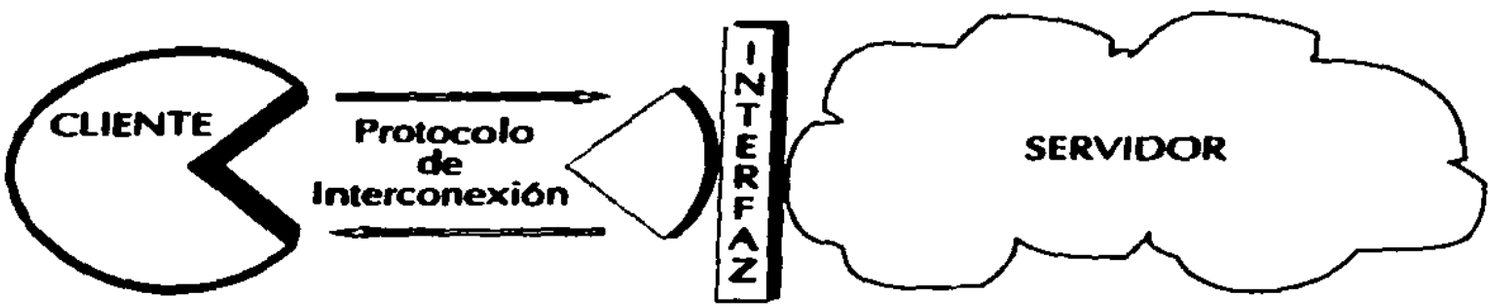


Fig. 2-3 Modelo de composición general

Servidor	proveedor de servicios (es el componente reusable).
Interfaz	información acerca de los servicios que proporciona el servidor y del contexto en que pueden ser solicitados.
Cliente	consumidor de servicios.

El elemento central en este modelo es la interfaz, pues implica una intermediación con una implementación (servidor) y su uso por otra entidad (cliente). De esta manera, tenemos una triple relación *Servidor-Interfaz-Cliente* que conforma un modelo de composición donde el cliente le solicita un servicio al servidor a través de una interfaz. El servidor contiene la implementación de los servicios que proporciona y la interfaz describe la abstracción representada por el componente (aislando la implementación) y especifica la forma como debe utilizarse. De esta manera, la interfaz especifica un *protocolo de conexión* para el componente y el cliente debe ajustarse a este protocolo para poder utilizar los servicios disponibles.

Con base en este modelo general, podemos establecer principios generales que facilitan y propician el reuso de componentes:

Independencia debe mantenerse desacoplados lo más posible a los clientes de los servidores. Básicamente debemos asegurar: independizar a los clientes de la representación de datos en los servidores, minimizar el efecto que tengan sobre ellos los cambios de requerimiento en los servidores e independizar a los servidores de las características especiales de futuros clientes.

Genericidad debe capturar la máxima similitud de una abstracción cuando se utiliza en diferentes contextos. Básicamente debemos asegurar: evitar la repetición de código que se diferencie sólo en los tipos de datos que utiliza y abstraer las similitudes entre servidores para proveer un acceso uniforme.

Refinamiento es la posibilidad de adaptar (modificar o extender) el componente a cambios en los requerimientos, sin perturbar a los clientes de dicho componente. Este también es un aspecto de independencia que tiene mucha influencia en la integración entre servidores y clientes.

Los principios descritos son importantes porque definen requerimientos generales para la construcción y uso de componentes reutilizables. Asimismo, constituyen un marco de referencia para la comparación de diferentes modelos de composición, cotejando la manera como los promueven y se ajustan a ellos.

2.3 Problemas que debe abordar un modelo de composición

Un modelo de composición debe abordar varios problemas relacionados con la construcción de programas en general, cuya solución tiene implicaciones importantes para el reuso de software. Los problemas planteados aquí[5] constituyen un marco general de

[5] Hemos tomado de [Meyer 88:31-34] los problemas de variación de tipos, independencia de representación de servicios (que lo llama sólo independencia de representación) y similitud entre implementaciones. Estos problemas originalmente fueron planteados como problemas técnicos de la reusabilidad en general, sin embargo, creemos que son problemas técnicos relativos a la interconexión de componentes y los hemos adaptado a esta perspectiva.

referencia para la comparación del paradigma procedural y del paradigma OO, en relación a los beneficios que ofrecen uno y otro para el reuso.

2.3.1 El problema de la variación de tipos

Se presenta cuando hay estructuras de datos contenedoras iguales y/o procedimientos que tienen el mismo comportamiento algorítmico y difieren sólo en los tipos de datos que utilizan. Es un problema para el cliente que debe solicitar la construcción de un nuevo componente para un nuevo tipo de dato y es un obstáculo para los implementadores del componente que deben repetir el mismo código. También es un problema para el mantenimiento, pues los componentes que tienen el mismo código para diferentes tipos de datos mantienen una relación de integridad conceptual a través de su código común y si se decide modificar una implementación, se deberá modificar la implementación de todos los componentes relacionados. Por último, al repetir muchas veces la misma implementación de un componente aumenta la complejidad del uso del depósito.

2.3.2 El problema de la independencia de representación de datos y de la evolución funcional del sistema

Los sistemas evolucionan a medida que se les imponen nuevas demandas. Por lo general, estas demandas se pueden clasificar en cambios de representación de la información y en cambios en la funcionalidad del sistema.

¿Y esto qué tiene que ver con la reusabilidad?. El modelo de componente debe reducir la incidencia de estos cambios lo más que pueda, pues de lo contrario sus beneficios se limitarían al desarrollo inicial del sistema, perdiéndose en la evolución del mismo. En cuanto a los cambios de representación de la información, el modelo de composición debe aislar al cliente de los cambios en los formatos de almacenamiento. Para lograr esto, debe evitar que los clientes accedan directamente a las representaciones de los datos, en lugar de ello, debe proporcionar servicios que lo hagan.

En cuanto a los cambios en la funcionalidad del sistema, se puede reducir la incidencia de los cambios haciendo que el centro del reuso no se halle en las funcionalidades, porque es precisamente lo que cambia y afecta a los clientes. Veremos más adelante que el modelo OO, propicia un reuso centrado en las abstracciones de las entidades del dominio del problema. Esto lleva a un modelo de componente más estable para su reuso, pues las entidades son más estables que las funcionalidades en las que se ven involucradas.

2.3.3 El problema de la independencia de representación de servicios

Este problema surge cuando se tiene diferentes implementaciones para un mismo servicio. Afecta fundamentalmente al cliente que en lugar de hacer la invocación de un sólo servicio, tiene que hacer una invocación diferente para cada implementación; por esta razón también lo llamamos *invocación no uniforme*. Es un problema de independencia de representación porque el cliente tiene que saber a que implementación

concreta se está refiriendo en la invocación. Podemos desglosar este problema en dos, uno de naturaleza estática y el otro de naturaleza dinámica.

i. Invocación estática no uniforme

El problema surge cuando los mecanismos de programación disponibles obligan a que cada implementación diferente de un mismo servicio tenga un nombre diferente para poder distinguirlas. El cliente tiene que hacer invocaciones con diferentes nombres para solicitar un mismo servicio y el servidor tiene que proporcionar diferentes interfaces para un sólo servicio, haciendo más complejo el modelo de composición. Debido a que la invocación se hace de manera explícita en el código la denominamos invocación estática no uniforme.

ii. Invocación dinámica no uniforme

El problema surge cuando se tiene diferentes implementaciones de una abstracción de datos, a cada una se le asocia una implementación distinta de un servicio que proporciona la misma funcionalidad abstracta y dinámicamente hay que detectar el tipo de implementación concreta de la abstracción e invocar el servicio correcto asociado. Veamos un ejemplo:

Imaginemos una sucursal bancaria que atiende a sus usuarios mediante un sistema de ventanilla única y una formación de multifila. En la última semana, el gobierno decide otorgar una bonificación extra a todos los jubilados a través del banco. Como es un trámite fuera de lo común, ocupa tiempo adicional en las ventanillas y se produce congestionamiento en las filas. En medio del problema, el banco decide asignar una ventanilla únicamente para el trámite de jubilados y busca en cada fila a los usuarios que solicitan el servicio de cobro de bono de jubilación para pasarlo a una nueva fila.

Las filas se modelan con la abstracción cola y en aras de ilustrar el problema se utilizará una implementación diferente de cola para representar cada fila. Almacenaremos todas las filas en una sola colección (p.e., un arreglo de *n* colas) y mover a los jubilados a la nueva fila con algo tan general como:

```
MueveUsuariosServicioFilas(jubilados, Fila[12]);

donde,

PROCEDURE MueveUsuariosServicioFilas(s: Servicio, nueva: Cola)
   VAR i: integer;
BEGIN
   FOR (i = 1 TO n) DO MueveUsuariosServicioFila(Filas[i], s, nueva);
END;

PROCEDURE MueveUsuariosServicioFila(c: Cola, s: Servicio, nueva: Cola);
```

La semántica de *MueveUsuariosServicioFila* es remover a todos los usuarios de la cola *c* que solicitan el servicio *s* e insertarlos en la cola *nueva*. La definición de este servicio para todos los tipos de colas requiere poder representar una interfaz común para todas las implementaciones (el manejo de un tipo *Cola*). La invocación de este servicio para cada implementación de cola requiere poder manejar compatibilidad de tipos entre la asbtracción y sus implementaciones (p.e., si se tiene las implementaciones *cL: ColaLista* y *cA: ColaArreglo* se invocará *MueveUsuariosServicioFila(cL, s, nueva)* y *MueveUsuariosServicioFila(cA, s, nueva)*).

Para remover a un usuario, primero hay que buscarlo en la cola para ver si se encuentra en ella. Lo interesante del asunto es que hay una operación de búsqueda para cada implementación de cola y que en la invocación del procedimiento *MueveUsuariosServicioFila* no está determinada estáticamente la implementación concreta de la cola que se almacena en el arreglo, por lo tanto, ¿cómo saber que función de búsqueda invocar?. Una manera de manejar esto es disponer de un identificador de tipo para cada implementación de cola y determinar el tipo del argumento para invocar la función correcta, por ejemplo:

```
PROCEDURE MueveUsuariosServicioFila(s: Servicio, c: Cola, nueva: Cola)
   VAR posicion: integer;
BEGIN
   REPEAT
        IF (c es de tipo ColaLista) THEN
             posicion := BuscaColaLista(s, c);
        ELSIF (c es de tipo ColaArreglo) THEN
             posicion := BuscaColaArreglo(s, c);
        ELSIF . . .
        (etc.)

        IF (posicion > 0) THEN (mueve al usuario de la cola c a la cola nueva)
   UNTIL (posicion = 0)
END
```

Bajo esta estrategia de discriminación dinámica, el código se verá poblado de estructuras condicionales para determinar la invocación del servicio correcto, y lo que es peor una nueva implementación de una abstracción provocará la modificación y recompilación de gran parte del código. Desde el punto de vista de la reusabilidad, si las estructuras condicionales se dejan en el código cliente, se dificultará el reuso de componentes porque la interconexión requerirá una cantidad de código adicional considerable para utilizar un servicio. Por otra parte, si las estructuras condicionales se dejan en el código servidor, el código de todos los procedimientos relacionados con una estructura de datos deberá modificarse cada vez que se cree una nueva implementación de la estructura y deberá recompilarse el código cliente. Ello nos

conduce a un depósito de componentes reusables poco estable y a una estrategia de mantenimiento del depósito sumamente costosa y compleja.

Lo que realmente se necesita es un mecanismo de programación que determine automáticamente el servicio correcto que debe invocarse en función de la implementación concreta de la abstracción que se esté utilizando. De esta manera, deberíamos poder escribir un código cliente del servicio de búsqueda en una cola, de la siguiente manera:

```
PROCEDURE MueveUsuariosServicioCola(s: Servicio, c: Cola, nueva: Cola)
   VAR posicion: integer;
BEGIN
   REPEAT
        posicion := Busca(s, c);
        IF (posicion > 0) THEN (mueve al usuario de la cola c a la cola nueva)
   UNTIL (posicion = 0)
END
```

Con este ejemplo hemos mostrado que bajo una situación dinámica en la que se utilizan diferentes implementaciones de una misma abstracción y en la que se carece de un mecanismo automático de discriminación, se deben conocer todas las implementaciones para invocar el servicio requerido. Esto constituye un problema de independencia de representación. Cuando no se logra esta independencia debemos introducir estructuras condicionales que nos permitan determinar dinámicamente la invocación adecuada, por lo que nos referiremos a este problema como *invocación dinámica no uniforme*. Veremos también cómo el modelo de composición OO aborda eficientemente este problema con el uso de la herencia y el polimorfismo.

2.3.4 El problema de la similitud entre implementaciones

Si existen diferentes implementaciones de un componente que tengan similitudes significativas, existe una relación de integridad conceptual a través del código que debe mantenerse. El modelo de componente debe poder factorizar estas similitudes, o de lo contrario, si se decide modificar una estrategia de implementación, se tendrá que modificar las implementaciones de todos los componentes que siguen tal estrategia, dificultando la evolución y mantenimiento del depósito de componentes.

Por ejemplo, para las operaciones de búsqueda en una *ColaArreglo* y en una *ColaLista*, se debería poder abstraer lo común de las dos implementaciones, esto es:

```
PROCEDURE Busca(x: Elemento, c: Cola): Boolean;
   pos: Posicion;
BEGIN
   pos := PosicionInicial(c);
   WHILE NOT ( HaTerminado(c, pos) AND NOT
```

```
                    EsIgual(c, pos, x) ) DO
        pos := SiguientePosicion(c);
    END;
    RETURN NOT HeTerminado(c, pos)
END
```

Las diferencias entre las dos implementaciones de búsqueda son cómo determinar: la posición inicial (pos:=1, pos:=head) la siguiente posición (pos:=pos+1, pos:=pos.next), el fin del recorrido (pos>n, pos=null), y si el elemento en la posición es igual al buscado (c[pos]=x, pos.dato=x). A pesar de las diferencias, la estructura algorítmica del proceso de búsqueda es común a ambas implementaciones.

2.3.5 El problema del refinamiento

Lo ideal es reutilizar un componente sin modificarlo, sin embargo, no siempre el componente reusable satisface los requerimientos solicitados en su totalidad, puede ser frecuente que sólo una parte de él sea útil a las demandas impuestas. En este contexto, es muy importante que el modelo de composición proporcione un mecanismo que permita modificar y/o extender los servicios del componente, de manera que se pueda adaptar a los nuevos requerimientos.

Esta adaptación del componente a un nuevo contexto, debe asegurar que las modificaciones no afecten a sus clientes, en términos más concretos, no puede modificar la interfaz del componente, ni sintáctica ni semánticamente. Pero no sólo eso, modificar la implementación de los servicios también puede afectar a los clientes en sus suposiciones acerca del desempeño, por lo que si se quiere hacer una modificación deberá construirse un componente nuevo.

El asunto se complica aún más, porque al refinar un componente se está creando una nueva implementación del mismo, y sería muy conveniente que este nuevo componente lo puedan utilizar los clientes del anterior sin modificar su código anfitrión. Para lograr esto, se requiere construir un nuevo componente con base en el otro y proporcionar mecanismos para intercambiarlos según las necesidades, a esto lo llamaremos *interoperabilidad*.

2.3.6 El problema de la compatibilidad de interfaces

Un componente modela una abstracción y está asociado a una implementación determinada. El depósito de componentes reusables puede tener diferentes implementaciones para una misma abstracción, por ejemplo, *Pila* implementada con una *Lista* o un *Arreglo*. De igual forma, para la *PilaLista* podemos implementar el servicio de búsqueda con un método secuencial o binario.

El cliente debe poder accesar de manera uniforme a cualquiera de las operaciones de búsqueda o a cualquiera de las implementaciones de la estructura. Dicha estructura le permite intercambiar los servidores sin modificar su código anfitrión, es decir, tener

acceso uniforme a diferentes implementaciones. Para ello, es necesario que las implementaciones presenten una interfaz compatible, en el sentido de que en ambas se puedan solicitar los mismos servicios de la misma manera.

2.3.7 Otros problemas

i. El problema de la granularidad

Aunque el concepto de clase/objeto logra un mayor nivel de granularidad de la abstracción que se reutiliza en relación al procedimiento, es todavía insuficiente para lograr grandes beneficios. En la construcción de sistemas grandes se presenta repetición de estructuras complejas que representan arquitecturas completas de subsistemas y no encontramos mecanismos de programación que abarquen la granularidad de estos componentes y las relaciones que tienen con otros componentes parecidos. Creemos que una propuesta que pretenda atacar este problema debe proveer mecanismos concretos en lenguajes o ambientes de programación que mapeen estas abstracciones a programas ejecutables.

ii. El problema de la interconexión multiparadigma

Cuando hablamos de un depósito de componentes reutilizables, podemos tener componentes concebidos bajo distintos paradigmas: bibliotecas de procedimientos, bibliotecas de clases, conjuntos de reglas lógicas, componentes de representación de conocimiento, componentes de manejo de procesos concurrentes, paralelos y distribuidos, etcétera. Es muy probable que para un desarrollo necesitemos interconectar estos componentes, sin embargo, no existen mecanismos estándares de programación que permitan interconectar estos componentes en un nivel en el que se abstraigan los detalles de implementación y la sintaxis particular de cada uno de ellos.

iii. El problema de la integridad semántica

La interconexión de componentes debe asegurar que no se modifiquen las propiedades individuales de los componentes y que se satisfagan las propiedades requeridas en la interconexión. En este campo, ha habido avances importantes con el uso de especificaciones formales, pero éstas no forman parte de los paradigmas procedural y OO, aunque existen extensiones que pueden utilizarse.

En el caso del modelo de composición OO, en el lenguaje *Eiffel* encontramos el uso de *aserciones*. Las aserciones describen precondiciones y poscondiciones de los métodos e invariantes[6] de clase. Lamentablemente, este no es un elemento fundamental del modelo OO.

---

[6] "Una *invariante* es alguna condición booleana (verdadero o falso) cuya certeza debe ser preservada. Para cada operación asociada con un objeto, podemos definir *precondiciones* (invariantes asumidas por la operación) así como *poscondiciones* (invariantes satisfechas por la operación)" [Booch 94:43]. Adicionalmente podemos tener *invariantes de clase* que "...son restricciones semánticas adicionales, que proporcionan información crucial para la semántica de las clases" [Meyer 92b:301].

3. El modelo de composición del paradigma procedural

El elemento de construcción básico en este paradigma es el *procedimiento*[7] y es por tanto el modelo de componente subyacente. La interfaz de un procedimiento describe una abstracción funcional y está compuesta por un nombre y un conjunto de parámetros formales, cada uno de algún tipo de dato.

Un cliente realiza la conexión a un servidor proporcionando un nombre y un conjunto de parámetros actuales (valores para los parámetros formales) que se ajusten al protocolo especificado por la interfaz. De esta forma se realiza la composición de procedimientos y con ella la construcción de programas.

Desde la perspectiva de la reusabilidad encontramos dos limitaciones inmediatas:

- la interfaz encapsula un nivel de granularidad muy pequeño (instrucciones y procedimientos), limitando a este nivel la granularidad del reuso.
- las posibilidades de interconexión están limitadas a un nombre y un conjunto de parámetros.

Pasemos a revisar los conceptos del paradigma y sus contribuciones y limitaciones a la reutilización de código.

3.1 Conceptos y mecanismos del paradigma

Los conceptos básicos del paradigma procedural son el *procedimiento* y el *módulo*, adicionalmente hemos contemplado el mecanismo de genericidad por estar disponible en algunos lenguajes (Ada). Aunque el mecanismo de sobrecarga ya está presente en los lenguajes procedurales para algunas operaciones primitivas (operadores aritméticos), no permite su uso en la definición de nuevas operaciones, por lo que su análisis lo pospondremos para el modelo de composición OO.

3.1.1 Procedimiento

Como vimos, el procedimiento es el modelo de componente y determina el modelo de composición del paradigma. Este mecanismo representa un avance en la construcción y estructuración de programas y específicamente para reusabilidad abre la posibilidad de reutilizar conjuntos de procedimientos agrupados en una funcionalidad de mayor nivel. Sin embargo, el procedimiento por sí solo tiene severas limitaciones para la reusabilidad:

- es difícil representar abstracciones funcionales complejas. Sólo se dispone de un nombre y un conjunto de parámetros. Esta es una limitación para la construcción de componentes reutilizables.
- es problemático representar estructuras de datos complejas. Cada estructura está relacionada con un conjunto de componentes (procedimientos) que la accesan y esto establece un acople entre los componentes. Si se decide cambiar la

[7] También llamado rutina, subrutina, subprograma, función

representación de una estructura de datos, tendremos que modificar la implementación de los procedimientos relacionados y posiblemente el código cliente que hace uso de tales componentes. Esta es una limitación tanto en la construcción como en la utilización de componentes reutilizables.

- No fortalece los principios que deben observarse entre servidores y clientes, ni aborda los problemas planteados para el modelo de composición.

3.1.2 Módulo

El módulo es un mecanismo que "... encapsula en una sola unidad sintáctica, estructuras de datos conjuntamente con las operaciones que las manipulan" [Oktaba 93:40]. En relación al modelo de composición general planteado, le otorga mayor importancia al concepto de interfaz, separándola físicamente de la implementación y permitiendo la compilación separada.

En primer lugar, vemos una mejora en la granularidad de la abstracción que se puede representar, pues agrupa una mayor cantidad de información y toda la funcionalidad relacionada con ella. El beneficio para los implementadores es que pueden construir componentes reusables de mayor tamaño y que pueden tener un mantenimiento mejor controlado del depósito de componentes, pues la evolución de los componentes por lo general se puede limitar al interior de los módulos. Por el lado de los usuarios, posibilita una búsqueda mejor estructurada basada en conjuntos de características relacionadas y le da mayor versatilidad al modelo de composición, ya que proporciona un constructor de mayor nivel para la interconexión de componentes: el módulo.

Más importante aún, aborda el problema de la independencia de representación de datos complejas, pues la estructura está contenida por completo en el módulo y define un protocolo de acceso a través de una interface. De esta manera, un cambio de representación, en general, produce cambios en el interior del módulo y no en el código cliente del componente. Con esto refuerza el principio de independencia.

El principio de refinamiento también se ve fortalecido porque permite agregar nuevas funcionalidades en el módulo sin alterar la interfaz utilizada por los antiguos clientes. Por otro lado, aborda el problema de evolución funcional del sistema ya que desplaza el centro del reuso del componente a una entidad de mayor nivel.

3.1.3 Genericidad

Como respuesta al problema de la variación de tipos, algunos lenguajes implementan el mecanismo de genericidad[8].Este tipo de constructor toma como base una plantilla que contiene un código común para un conjunto de *parámetros formales genéricos* y produce una instancia particular para algunos tipos específicos (*parámetros actuales genéricos*). Es un mecanismo de expansión de código que realiza un trabajo sintáctico de manera estática.

---

[8] Este mecanismo no es uno de los elementos básicos del paradigma procedural, pero ya estaba disponible en algunos lenguajes (p.e, el paquete genérico en Ada).

Este mecanismo representa un avance significativo en lo que se refiere al ahorro en la escritura de código, pues en lugar de escribir códigos repetidos para diferentes tipos de datos, se escribe un sólo código y se generan instancias para tipos de datos específicos. Como beneficio adicional, a la hora de construir el componente no se requiere saber el tipo de dato que actuará como parámetro formal en la interfaz, es decir, no tiene que asumir características especiales acerca de los futuros clientes.

Desde la perspectiva del reuso, este es un mecanismo que permite reutilizar la *definición* de un código genérico, lo que representa una ventaja considerable tanto para el implementador como para el cliente del componente, sin embargo, duplica la definición del código con cada instancia diferente (*repetición de código*).

Es importante resaltar que el mecanismo de genericidad es una forma de polimorfismo estático que limita el reuso del componente a situaciones estáticas. ¿Y esto que quiere decir?., que una vez que se genera (estáticamente) una instancia de un componente genérico, pierde toda su flexibilidad y posibilidad de adaptación, pues ya tiene sus parámetros definidos. La definición paramétrica de un componente agrega un grado más de versatilidad al modelo de composición, pues permite adaptar el componente a diferentes contextos utilizando diferentes parámetros. Sin embargo, esta adaptabilidad es estática y la ofrece el componente genérico, que no puede utilizarse dinámicamente. Veremos que en el modelo OO existe una forma de polimorfismo dinámico que permite adaptar el uso del componente en tiempo de ejecución.

4. El modelo de composición del paradigma OO

El elemento de construcción básico en este paradigma es el *objeto*. Este es un modelo de componente diferente del procedimiento y por ende está asociado a otro modelo de composición. En una primera aproximación, podemos destacar que este modelo de componente aporta un mayor beneficio al reuso porque involucra abstracciones de mayor granularidad que el procedimiento.

Al igual que el procedimiento, un objeto está formado por una interfaz y por una implementación. Entenderemos por interfaz de un objeto la especificación del conjunto de servicios que ofrece más la especificación del contexto de aplicación de estos servicios[9]. Cada servicio que ofrece el objeto tiene una especificación idéntica a la de un procedimiento, es decir, un nombre más un conjunto de parámetros formales.

[9] [Oldabe+ 93b:26] se refiere a estos dos aspectos de la interfaz como *interfaz sintáctica* e *interfaz semántica*. Aunque no todos los autores hacen énfasis en que la interfaz aborca información acerca del contexto donde deben invocarse los servicios, hemos decidido hacer este énfasis, porque el objeto tiene un estado que influencia su conducta y muchas veces determina un cierto orden en el que deben invocarse sus operaciones. Así, si tenemos un objeto que modela una máquina de refrescos, éste no permitirá invocar el servicio de expulsión del refresco hasta que se haya invocado el servicio de pago del mismo. La especificación del contexto de aplicación se puede definir con el uso de invariantes, pero no todos los lenguajes orientados a objetos incluyen esta posibilidad.

Estamos ante una interfaz diferente, y en consecuencia ante diferentes formas de interconexión de componentes. En el modelo de composición procedural, el nombre del procedimiento junto con los parámetros actuales constituye la invocación del servicio que se le solicita al componente identificado por la misma información. En el modelo de composición OO, existe una clara diferencia sintáctica[10] entre el componente y los servicios que proporciona, es decir, para invocar un servicio, se debe especificar el identificador del objeto más el identificador del servicio requerido. De tal manera, vemos que este modelo de composición agrega un elemento sintáctico a la composición de componentes, pues además del nombre del servicio y del conjunto de parámetros, se tiene el nombre del objeto.

Paradigma	Asociación del servicio
Procedural	*F* (nombre del servicio)
OO	*F* (objeto + nombre del servicio)

***Fig. 2-4** Solicitud de un servicio*

Por supuesto, este agregado sintáctico no representa *el* beneficio para el modelo de composición, sólo es una base sobre la que se estructuran mecanismos estáticos y dinámicos que amplían las posibilidades de composición. En particular, como veremos en el concepto de *polimorfismo dinámico*, el tipo de dato del objeto receptor del mensaje puede variar dinámicamente, por lo que el requerimiento del servicio podrá ser respondido por diferentes implementaciones y esto sí representa una gran flexibilidad en la interconexión de componentes. Pero, no nos adelantemos, veamos como funcionan estos mecanismos y qué aportan al modelo de composición y al reuso de software.

4.1 Conceptos y mecanismos del paradigma

Los conceptos básicos del paradigma OO son el *objeto*, la *clase* y la *herencia*. El concepto de clase está estrechamente ligado al de tipos de datos abstractos, por lo que se analizarán juntos. Las implicaciones de la herencia se han dividido en sus aspectos estáticos (herencia simplemente) y sus aspectos dinámicos que da lugar a un mecanismo que hemos llamado *polimorfismo dinámico* y que tiene aportes particulares al reuso de componentes. Abordaremos, en primer lugar, el mecanismo de sobrecarga que postergamos en el paradigma procedural.

[10] Decimos sintáctica, porque semánticamente, más que diferencia, lo que existe es la integración conceptual de los servicios y la abstracción del componente en una sola unidad.

4.1.1 Sobrecarga

"La *sobrecarga* se refiere a la posibilidad de definir varias funciones con el mismo nombre" [Quintanilla+ 93:6] y la versión adecuada de la operación invocada se selecciona en función del tipo y número de parámetros. Esta propiedad concede al programador cliente, la posibilidad de escribir el mismo código cuando reutiliza diferentes implementaciones de un mismo servicio, y otorga al implementador la posibilidad de mostrar una interfaz uniforme para las diferentes implementaciones de una operación.

En lenguajes como pascal ya existe la sobrecarga de operaciones primitivas sobre tipos básicos del lenguajes. Sin embargo, la sobrecarga que nos interesa es la que le da la posibilidad al cliente de definir nuevos nombres de funciones sobrecargadas. Esto último lo proporciona el paradigma orientado a objetos, pues permite definir varios métodos con el mismo nombre al interior de una clase.

La sobrecarga es un mecanismo estático que aborda uno de los problemas de independencia de representación de servicios, el que llamamos *invocación estática no uniforme*. De esta manera, se puede utilizar el mismo nombre de operación para diferentes implementaciones y la determinación de la operación correspondiente se hace estáticamente, en tiempo de compilación.

4.1.2 Tipos de datos abstractos (TDAs), clases y objetos

El modelado de soluciones computacionales a problemas del mundo real ha sido abordado con diferentes propuestas de elementos para su representación. Cada paradigma de programación aporta elementos diferentes para el modelado de soluciones, por ejemplo: el procedimiento en la programación procedural, la función en la programación funcional, el predicado en la programación lógica, el objeto en la programación OO, y otros como la corutina en la programación concurrente.

Uno de los problemas que se presentaron inicialmente en el modelado de soluciones fue la dificultad de representar información compleja con los tipos de datos básicos y los mecanismos de estructuración disponibles en los lenguajes de programación. Así, surge la necesidad de disponer de nuevos tipos de datos. Paralelamente, se percibe que un tipo de dato es más que sólo el conjunto de valores que puede asumir, tal como destaca [Greiff 93b:11] "...lo que realmente define a un tipo de dato es lo que podemos *hacer* con esos elementos", sin importarnos como está implementado en la máquina.

En este contexto, surge el concepto de TDA como un mecanismo de abstracción que permite extender el lenguaje con nuevos tipos de datos, así como el procedimiento permite extender el lenguaje con nuevas rutinas. Un TDA se manifiesta como un conjunto de valores más las operaciones que los manipulan, haciendo énfasis en que el usuario no tiene conocimiento acerca de la representación de la información. También aparecen lenguajes de programación que apoyan el concepto de abstracción de datos, permitiendo "...agrupar en una misma declaración los datos y las operaciones que los manipulan, ocultando los detalles de implementación." [Quintanilla+ 93:5]

De acuerdo con [Oktaba+ 93b:25], "hablar de un tipo de dato abstracto es hablar de la definición de un nuevo tipo de dato con dos propiedades fundamentales:

1. *unidad lógica de datos y operaciones*. En realidad no tiene sentido tener datos si no tenemos operaciones para manipularlos y viceversa.
2. *ocultamiento de representación*[11]. No es relevante para el usuario conocer como se representan los datos en la máquina o cómo se implementan las operaciones"

El beneficio fundamental del concepto de TDA para la construcción de programas es aportar una nueva herramienta para el modelado, estructuración y modularización. Otra visión del aporte de este concepto lo encontramos en [Harrison 92:2], "la abstracción de datos puede ser descrita como un método de controlar la interacción entre un programa y sus estructuras de datos".

i. Pero, ¿qué tiene que ver el concepto de TDA con la reusabilidad?

Mucho. En primer lugar, aporta los mismos beneficios que el concepto de módulo: presenta una granularidad de abstracción similar y el ocultamiento de representación aisla el uso del componente de su implementación, protegiendo los datos de la abstracción (protección de la integridad del componente).

En segundo lugar, encontramos aportes originales. El desarrollo basado en TDAs posibilita la construcción de componentes reusables que mapeen directamente abstracciones del dominio del problema, incrementando la posibilidad de su reuso y, más importante aún, le otorga al cliente una gran flexibilidad para la integración del componente a su sistema pues permite manejar el componente como un tipo de dato con todos los privilegios de los tipos primitivos del lenguaje. Adicionalmente, aparece la idea de compatibilidad de interfaces, esto es, podemos tener diferentes TDAs que proporcionan implementaciones diferentes para los mismos servicios, por lo que el cliente las puede intercambiar sin modificar su código anfitrión.

ii. Y, ¿qué tiene que ver el concepto de TDA con OO?

Muchísimo, podemos empezar con que la idea de TDA "...fue influenciada por las primeras experiencias basadas en el uso de clases en Simula 67" [Oktaba+ 93b:25]. Por otro lado, una clase es la implementación[12] de un TDA. La clase es una declaración de un tipo de dato[13] y los objetos de esa clase son los valores que puede asumir el tipo. Esta

---

[11] Entenderemos como sinónimos los términos: *ocultamiento de representación*, *ocultamiento de información*, *encapsulamiento*, *encapsulación* (esta nota no está incluida en la cita).

[12] En [Oktaba+ 93b:25-26] encontramos que "el modelado de un tipo de dato abstracto comprende dos facetas: su especificación y su implementación". La *especificación* describe la semántica de la abstracción, para lo cual se utilizan métodos de especificación formal como la especificación algebráica. La *implementación* describe un conjunto de valores y las operaciones asociadas.

A su vez, en la clase distinguimos una interfaz y una implementación. La interfaz puede contener aspectos sintácticos y en algunos casos semánticos de la abstracción que representa. En general, en términos de la programación OO cuando se habla de la especificación de una abstracción se está hablando de la interfaz de la clase más que de la especificación del TDA que modela la abstracción del problema. Es interesante notar que "...las clases *diferidas o abstractas*, describen un conjunto de posibles implementaciones del mismo tipo de dato abstracto" [Meyer 92:37].

[13] Clase no es lo mismo que *tipo*, pues dos objetos pueden ser instancias de diferentes clases y ser valores del mismo tipo, gracias a las relaciones de subtipo establecidas por la herencia.

relación es posible porque "un objeto es una abstracción que encapsula, ocultando su representación interna, a un estado junto con cierto comportamiento, lo que se puede interpretar en términos de un tipo de dato abstracto como un dato junto con las operaciones que lo manipulan." [Oktaba+ 93b:26]

En los lenguajes OO, cuando se declara una clase se está declarando un nuevo tipo de dato, que es tratado de la misma manera que los tipos de datos primitivos, esto es, con el nuevo tipo se puede declarar variables, declarar argumentos en los procedimientos y utilizar como valor de retorno. A tal punto existe una relación entre el TDA y el paradigma OO, que los programas se pueden visualizar como colecciones estructuradas de tipos de datos abstractos.

Así, vemos que la abstracción de datos es uno de los pilares más fuertes del paradigama OO y al abarcarla, proporciona también sus beneficios para la reusabilidad. Adicionalmente, el paradigma también contempla el mecanismo de genericidad con el uso de *clases genéricas o parametrizadas*, permitiendo adaptar el uso de una clase a diferentes tipos de objetos.

4.1.3 Herencia

En el contexto de modelado de programas basado en tipos de datos, se estructuran las abstracciones del problema en un sistema de tipos, donde un tipo determina las características estructurales y de comportamiento de los objetos que le pertenecen, regulando así su uso [Quintanilla+ 93:5]. Es notorio que diferentes tipos presenten características estructurales y de comportamiento comunes, lo que da pie a una relación de subtipo. Así, se pueden definir nuevos tipos con base en otros diciendo que mantienen una relación de subtipo y especificando cómo difieren de ellos.

Esta relación de subtipo se implementa en el paradigma OO con el concepto de herencia, donde una clase (subtipo) hereda (comparte) todas las operaciones asociadas con otro tipo (clase). Así, vemos que la forma como se manifiesta la herencia está íntimamente ligada al concepto de TDA. Una relación de subtipo define una compatibilidad entre tipos, de manera que se les puede asociar un mismo conjunto de operaciones que manipulen sus valores. Por lo tanto, el tipo de una clase puede adoptar como valores los objetos de su clase y los objetos de las subclases de ésta.

Desde el punto de vista de la reusabilidad, la herencia aborda:

1. el problema de similitud entre implementaciones, pues el código común a varias clases se factoriza en una clase superior de la cual heredan todas las demás, así en lugar de existir repetición de código, cada subclase reutiliza las definiciones e implementaciones de la superclase.
2. el problema de refinamiento, pues si se quiere extender y/o modificar los servicios de un componente, se crea una nueva subclase y se redefinen o agregan los servicios requeridos, con la particularidad de que esta extensión es intercambiable con el componente original en el código cliente.

De esta manera, tenemos que el concepto de herencia introduce un nuevo mecanismo que hace a los componentes orientados a objetos adaptables a diferentes

contextos sin modificar la implementación del componente original. Por lo tanto, agrega flexibilidad al modelo de composición OO, permitiendo adaptar la conducta del componente reusable sin afectar a sus clientes.

4.1.4 Polimorfismo Dinámico

El *polimorfismo dinámico* permite que el tipo de dato del objeto servidor pueda variar dinámicamente, por lo que el requerimiento del servicio podrá ser respondido por diferentes implementaciones. Esto representa una gran flexibilidad en la interconexión de componentes.

El *polimorfismo dinámico* contempla dos aspectos:

1. *polimorfismo para la variable*: una variable se declara estáticamente de un cierto tipo, indicando que puede adoptar solamente los valores (objetos) de ese tipo (clase), pero gracias a la herencia, también puede adoptar los valores de los subtipos de su clase (sus subclases).
2. *polimorfismo para la invocación del procedimiento*: como la variable puede dinámicamente adoptar valores de cualquiera de sus subclases, la invocación de un procedimiento sobre esta variable (objeto) no puede ser asociada estáticamente a una determinada implementación, para ello se utiliza el mecanismo de *enlace dinámico*, que determina el tipo dinámico de la variable y asocia la implementación correcta en tiempo de ejecución.

De esta manera, vemos que el *polimorfismo dinámico* se logra en los lenguajes OO por la combinación de herencia, enlace dinámico y la invocación de servicios sobre variables que dinámicamente adoptan valores de sus subtipos.

1. Pero, ¿qué tiene que ver el polimorfismo dinámico con la reusabilidad?

En primer lugar, aborda el problema de independencia de representación de servicios que llamamos *invocación dinámica no uniforme*, pues permite al reutilizador diseñar algoritmos generales que sirvan para componentes reutilizables con diferentes características de implementación y que coincidan en la especificación de un mismo comportamiento a través de una superclase común. De esta manera, si se reutilizan nuevas implementaciones de una abstracción (del depósito o de construcción propia), no será necesario modificar el código de los algoritmos que utilizan dicha abstracción.

Por otra parte, ¡permite que el componente reutilice código que no está escrito!. Hasta ahora hemos visto que el cliente es quien manipula el uso del polimorfismo dinámico utilizando los componentes reutilizables, pero si el componente hace invocaciones a sí mismo[14], puede manipular el polimorfismo dinámico utilizando al código cliente. Veamos el siguiente caso:

[14] *self* en Smalltalk, *this* en C++

COMPONENTE REUTILIZABLE (SERVIDOR)	COMPONENTE CLIENTE

Clase X {
  Metodo A() Begin ... End
  Metodo B() Begin ... Self A(); End
};
	Clase SubX : hereda de X {
	  Metodo A() Begin ... End
	};

Si declaramos una variable *x* con tipo estático ***X***, esta puede asumir valores de ***X*** y de su subtipo ***SubX***. Al invocar ***X.B()*** sólo en tiempo de ejecución se sabrá si se invoca ***X.A()*** o ***SubX.A()***, dependiendo del tipo dinámico de *x*. Realmente, lo que interesa es reutilizar componentes y no al revés, pero esto proporciona una gran flexibilidad en el reuso de componentes. [Deng-Jyi + 93:47] se refiere al método ***A()*** como un *adaptador de software* de la clase ***X***, pues permite adaptar el comportamiento del componente a nuevos contextos.

De esta manera, hemos visto que con el uso del polimorfismo dinámico es más sencillo reutilizar las funcionalidades de una abstracción, pues se puede adaptar a diferentes contextos con el uso de tipos dinámicos y con la definición de nuevas implementaciones de un servicio de autoinvocación.

Las posibilidades de reuso que proporciona el polimorfismo dinámico, dependen en gran medida de cómo se estructura la jerarquía de clases. En particular, el uso de este mecanismo se ve obstaculizado si en la jerarquía se tienen clases por conveniencia para la implementación y que no modelan las relaciones naturales de subtipo entre las entidades del problema. Dificulta el uso del mecanismo porque éste es posible a través del uso de una abstracción del problema y de sus diferentes implementaciones (especializaciones), si hay subclases que no modelan alguna entidad, puede haber conflictos en la invocación polimórfica.

Una forma de propiciar este reuso polimórfico, es diseñar el tronco del árbol jerárquico de una biblioteca con clases abstractas y las hojas con implementaciones concretas. De esta manera, se propicia que el cliente diseñe algoritmos utilizando las clases abstractas, que luego servirán para las diferentes implementaciones concretas de la biblioteca y de las nuevas que agregue el propio cliente. Con esto se logra desacoplar una abstracción particular de su implementación, incrementando el principio de independencia.

Otro criterio que beneficia al reuso polimórfico es tratar de construir la jerarquía de clases con el máximo número de niveles posibles, manteniendo la representación de las relaciones de subtipo del problema. De esta manera, se tendrá tipos abstractos más generales que proporcionan la funcionalidad suficiente en cada nivel, propiciando así, el reuso polimórfico en un mayor número de niveles de la jerarquía.

4.2 Formas de Reuso OO

Las diferentes posibilidades de interconexión de componentes que proporciona el paradigma las podemos clasificar en tres categorías de reuso: por *instanciación*, por *derivación* y por *polimorfismo*.

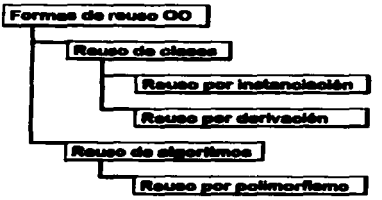


Fig. 2-5 Formas de reuso OO

4.2.1 Reuso por instanciación

Cada vez que se genera una instancia de una clase se está reutilizando la representación de una abstracción y la implementación de los servicios que proporciona, así, el programador puede invocar estos servicios a través de la interfaz del objeto, sin preocuparse de cómo están implementados. Esta es la forma de reuso más común en el paradigma y lo vemos plasmado en la construcción de un programa OO cada vez que se crean objetos:

a) para construir nuevos objetos, esto es, se crean nuevas abstracciones en base a abstracciones existentes, declarando los objetos como atributos de la nueva asbtracción (a esto lo llamamos *reuso por composición*)
b) para su uso local en la implementación de los servicios de un objeto
c) para su uso como parámetros actuales en la invocación de un servicio

Este tipo de reuso es adecuado cuando el componente se ajusta completamente a los requerimientos. Está basado en la propiedad de *encapsulación* asociada a la abstracción de datos, pues la interfaz esconde la implementación que se está reutilizando y a la vez proporciona el protocolo para su acceso.

4.2.2 Reuso por derivación o subclasificación

Cuando la especificación del componente no se ajusta por completo a los requerimientos, se puede heredar de una clase y extender su funcionalidad. Con esto se está reutilizando el componente para crear una nueva abstracción especificando nuevos comportamientos y/o implementaciones. En teoría, la derivación se puede hacer sin

conocer los detalles de implementación de la superclase, es decir, sólo en base a su interfaz. Por tanto, esta forma de reuso está basada en los conceptos de *herencia* y *encapsulación*.

Es importante notar que las extensiones a una clase deben satisfacer ciertas condiciones, tal como nos señala [Lea 93:3]: "las subclases pueden añadir nuevas propiedades (p.e., métodos, atributos), y/o añadir más restricciones a las existentes. Sin embargo, las adiciones no pueden invalidar ninguna propiedad listada en la superclase." Para satisfacer esto, es muy útil el uso de invariantes, aunque a veces sea más sencillo escribir la implementación que describir sus efectos. Un caso común de extensión que viola condiciones es la declaración de una subclase para utilizar sólo una parte de la funcionalidad de la superclase sin preservar sus propiedades, alterando su semántica. Nótese que las propiedades de una clase conforman un protocolo de interoperabilidad sobre el que dependen los clientes, por lo que su incumplimiento obstaculiza el reuso.

Relacionado con este problema, algunos autores [Taenzer 89:32] resaltan la particularidad de que la herencia aumenta la complejidad del componente al aumentar la cantidad de métodos disponibles, y que en algunas ocasiones es preferible hacer un reuso por instanciación en lugar de derivación para reducir la complejidad de la clase. No estamos de acuerdo con esta posición, porque independientemente de la complejidad de la clase, la relación de herencia modela una conexión de subclasificación entre entidades del problema, que no es posible modelar con una relación de instanciación. Por otro lado, esta propuesta conduce a una construcción de la jerarquía de clases que obstaculiza el reuso por polimorfismo, una forma muy poderosa de reuso. En general, podemos decir que si existe confusión entre reutilizar un componente por instanciación o derivación, podemos estar ante un mal diseño de la jerarquía de clases, ante un mal modelado o ante un mal uso de los conceptos del paradigma[15].

4.2.3 Reuso por polimorfismo

Podemos decir que el reuso por instanciación y el reuso por derivación son una forma de reuso de clases. El reuso polimórfico, en cambio, es una forma de reuso de algoritmos. Si un método presenta diferentes implementaciones en las subclases de una superclase, se dice que el método es polimórfico respecto a la superclase. En este contexto, un cliente que utilice las subclases puede reutilizar el algoritmo polimórfico, pasando como argumento el objeto de la clase derivada.

El uso de clases abstractas para propiciar el reuso por polimorfismo, separa la implementación de la especificación de la abstracción y, con ello, facilita el reuso por derivación porque no impone restricciones de implementación a la clase heredera. De esta manera, se pueden construir múltiples implementaciones para una misma abstracción, y además, teniendo varias opciones de implementación, aumenta la posibilidad de reuso por instanciación.

---

[15] La confusión en el uso de estos conceptos se propicia en la literatura: "...los lenguajes, manuales de usuario y libros de texto obscurecen estos problemas intentando relacionar los conceptos de subclasificación al uso de esquemas de almacenamiento por capas en tiempo de ejecución que pueden ser vistos como objetos de superclase empotrados dentro del objeto de subclase." [Lea 93:5]

4.3 Evaluación de las formas de reuso OO

Indudablemente, el reuso por instanciación o composición es el más sencillo de todos, el componente se reutiliza tal cual está definido. El reuso por derivación es más difícil, porque requiere definir o redefinir implementaciones, pero la dificultad realmente surge porque las implementaciones están estrechamente ligadas a la representación de los datos y muchas veces es necesario mirar el código de las superclases (violar la encapsulación) para extender su funcionalidad. Por último, el reuso por polimorfismo es el más difícil de todos porque implica la construcción de una jerarquía de clases con niveles desacoplados al máximo en términos de abstracción-implementación y que además modele las relaciones de subclasificación del problema.

En general, el reuso por instanciación está asociado al reuso de clases concretas, el reuso por derivación está asociado a clases abstractas[18] y el reuso por polimorfismo está asociado a una jerarquía de clases abstractas para las abstracciones y clases concretas para diferentes implementaciones.

Por otra parte, el tipo de reuso OO que se utiliza guarda una estrecha relación con el estado de madurez del depósito de componentes. Cuando se tienen jerarquías de clases reutilizables muy estables, generalmente se emplea el reuso por instanciación. "Esto contrasta con sistemas menos maduros en los cuales la nueva funcionalidad es obtenida creando nuevas subclases" [Helm 92:300], es decir reuso por derivación. El reuso por polimorfismo implica el reuso por derivación, aunque no denota falta de madurez de los componentes.

Finalmente, también el tipo de reuso está relacionado con el ciclo de vida de diseño e implementación del *software* OO. En los estados iniciales el reuso predominante es el reuso por derivación, mientras que en los estados posteriores el reuso predominante es el reuso por composición. Esto se debe a que en los estados posteriores ya se han rediseñado y refactorizado las jerarquías de clases y se logran objetos con interfaces estándares.

4.4 Evidencia experimental

Desde el punto de vista experimental, no existe suficiente investigación acerca de la incidencia que tiene el uso del paradigma OO sobre el reuso de software. No obstante, podemos mencionar algunos estudios que si lo hacen [Lewis 91, Lewis 92, Tewari 92, Poulin 95]. En particular, destaca el estudio de Lewis por ser pionero en el tema.

En su estudio, Lewis y su grupo ilustran el impacto que tiene el paradigma OO en el reuso de componentes de software y la productividad de desarrollo, en relación al paradigma procedural. Para ello, utilizaron seis grupos de programadores, distribuidos en dos categorías, tres desarrollando con un lenguaje procedural (Pascal) y tres con un lenguaje orientado a objetos (C++). Dentro de cada categoría, un grupo no podía reutilizar nada, otro sólo podía reutilizar componentes que se ajustaran completamente a

[18] "Las clases abstractas describen atributos, estados, relaciones, transiciones de estado, servicios e interacciones, sin especificar como son implementadas." [Lea 92:2]

los requerimientos, el último podía reutilizar cualquier cosa y de cualquier forma. Los resultados mostraron básicamente que:

a) el paradigma OO promueve una mayor productividad que el paradigma procedural sólo cuando se reutiliza
b) el reuso promueve mayor productividad sin considerar el paradigma del lenguaje utilizado
c) el nivel de mejora debido al reuso es mayor con el paradigma OO que con el paradigma procedural.

5. Evaluación de los modelos de composición

Hemos mostrado la relación que tienen los conceptos y mecanismos de programación del paradigma procedural y del paradigma OO con el reuso de componentes. Para ello, en lugar de partir de un concepto abstracto acerca del reuso, planteamos que los aportes de un paradigma al reuso de software tienen una manifestación concreta en la manera como pueden integrarse los componentes a un nuevo desarrollo. Estas formas de interconexión de los componentes forman un modelo de composición, que hemos utilizado como hilo conductor y sus conceptos generales (servidor, interfaz, cliente) se han empleado como una terminología uniforme en la exposición.

Para realizar la comparación de los modelos de composición procedural y OO necesitábamos un marco de referencia y lo establecimos planteando una serie de problemas relativos a la reusabilidad que debía resolver el modelo de composición. Los problemas planteados son básicamente obstáculos para el paradigma procedural y se sabe de antemano que el paradigma OO los aborda. En este sentido, tal vez no sea la manera más justa de establecer la comparación, pues se pudiera haber planteado problemas que no pudiera resolver ninguno de los dos paradigmas. No obstante, la metodología empleada no buscaba demostrar la supremacía del paradigma OO, el objetivo básico fue ilustrar sus aportes concretos.

Después de haber visto la relación entre los problemas planteados y los dos modelos de composición bajo estudio, queremos retomar esta relación de una manera sintética. Para ello nos basaremos en una tabla comparativa que resume las vinculaciones entre los mecanismos de programación y los problemas que aborda. En la tabla aparecen los problemas en las filas y los mecanismos en las columnas. En principio, pensamos utilizar un sólo símbolo (✔) para denotar en un cruce que un mecanismo aborda un problema. Sin embargo, el asunto se complica cuando un problema lo aborda más de un mecanismo, ya que podemos encontrar diferencias significativas en los aportes de cada uno. De esta manera, podemos tener más de un símbolo en un cruce, indicando que el mecanismo realiza un mayor aporte a la solución del problema que otro mecanismo con un número inferior de símbolos. En el caso en que dos mecanismos tengan el mismo número de símbolos no significa que los dos hacen el mismo aporte, sino que son de naturaleza diferente y no se distingue claramente la preeminencia de uno sobre el otro. Por otro lado,

no puede establecerse comparaciones entre la cantidad de símbolos entre diferentes filas. Por último, queremos advertir que el número de símbolos asignado es un tanto arbitrario pues responde a criterios de apreciación.

Comencemos con la *variación de tipos*. La *genericidad* crea una instancia diferente para diferentes tipos de datos. Por su parte, el *polimorfismo dinámico* también hace su aporte, porque manipula variaciones de tipo, pero con una interfaz compatible. Por la manera como trabaja cada mecanismo, podríamos subdividir el problema en variación de tipos estática y dinámica. La genericidad se lleva el mayor reconocimiento porque puede crear un número infinito de instancias del componente reusable y porque no necesita conocer el tipo de dato involucrado, lo que reduce el acople entre el servidor y el cliente.

	Procedimiento	Módulo	Genericidad	Sobrecarga	Clase (TDA)	Herencia	Polimorfismo dinámico
Variación de tipos			✔✔				✔
Independencia de representación de datos		✔			✔		
Evolución funcional del sistema		✔			✔✔		
Granularidad	✔	✔✔			✔✔✔		
Invocación estática no uniforme				✔			
Invocación dinámica no uniforme							✔
Similitud entre implementaciones						✔	
Compatibilidad de interfaces						✔	
Refinamiento						✔	✔✔
Flexibilidad de interconexión	✔		✔✔		✔✔	✔✔✔	✔✔✔

Fig. 2-6 Problemas de reusabilidad vs. mecanismos de programación

La *independencia de representación de datos* se logra evitando que el cliente accese directamente los datos, para lo cual debe proporcionar servicios de acceso. Esto lo hacen tanto el *módulo* como la *clase*. Al mismo tiempo, estos dos mecanismos abordan la evolución funcional del sistema porque desplazan el centro de reuso de la funcionalidad al módulo y a la clase. Esta última tiene mayor calificación, porque al modelar entidades del problema propicia un mayor reuso, y con él un mayor desplazamiento.

En la *granularidad de abstracción* que puede encapsular un componente, hemos puesto al *procedimiento* con un símbolo como medida de referencia. Obviamente, el *módulo* que agrupa datos y procedimientos, tiene mayor calificación. La *clase*, aunque también agrupa datos y procedimientos, puede componer a otras clases como subpartes, en cambio un módulo sólo puede utilizar a otro módulo; así, la clase se muestra superior. Adicionalmente, la *genericidad* puede combinarse con el módulo y con la clase, aumentando en cada caso su evaluación, pues la granularidad se ve multiplicada por la cantidad de abstracciones que involucra.

Los problemas que son abordados por un sólo mecanismo no requieren mayor comentario. La *sobrecarga* aborda el problema de la *invocación estática no uniforme* al permitir definir servicios con el mismo nombre. El otro problema de la independencia de representación de servicios, la *invocación dinámica no uniforme* se resuelve con el uso del *polimorfismo dinámico*, que permite asociar en tiempo de ejecución la implementación correcta a una solicitud de servicio hecha a una variable que puede asumir dinámicamente valores de cualquiera de sus subtipos.

La herencia merece mención aparte porque resuelve varios problemas. Permite capturar el código común a varias implementaciones de una misma abstracción en una superclase y, con esto, aborda el problema de *similitud entre implementaciones*. También aborda de manera elegante la *compatibilidad de interfaces* a través de las relaciones de subtipo, pues una subclase tiene una interfaz compatible con su superclase, en el sentido de que se pueden solicitar los mismo servicios. Por último, para resolver el problema de *refinamiento*, la herencia provee una manera poderosa de redefinir y extender la funcionalidad de un componente: la subclasificación. Esta flexibilidad se ve aumentada por el hecho de que el nuevo componente tiene una interfaz compatible con el componente original y por lo tanto, son intercambiables.

Finalmente, una de la cosas que más nos interesa visualizar es la *flexibilidad de interconexión*, porque mientras haya mayor cantidad de formas de integración de componentes, mayor será la posibilidad de reuso. El *procedimiento* nuevamente se toma como referencia. Todos los mecanismos aportan elementos para facilitar la adaptación de un componente a un contexto anfitrión, sin embargo, hemos suprimido de esta visión al módulo, por considerar que su aporte está más relacionado con la independencia de representación de datos, la granularidad y la evolución funcional del sistema. De igual forma, la constribución de la sobrecarga es marginal a este respecto.

Así, queremos destacar primero a la *clase* por asociar el componente reusable con un tipo de dato, lo que permite que sea utilizado de variadas formas. La *genericidad* también es un factor muy fuerte porque permite adaptar el componente a nuevos contextos mediante parámetros. De manera especial tenemos a la *herencia* que permite adaptación

del componente por refinación y al *polimorfismo dinámico* que permite adaptar los algoritmos polimórficos a nuevas implementaciones sin afectar al cliente. Nóte, que la mayor flexibilidad de interconexión está relacionada con los conceptos del paradigma OO, salvo la genericidad, que también es contemplada por el paradigma procedural.

No cuantificamos exactamente el aporte de cada mecanismo, el símbolo utilizado debe interpretarse como una medida relativa. Sin embargo, es notorio que cuando nos desplazamos del paradigma procedural al paradigma OO encontramos mayor cantidad de símbolos. Cada mecanismo de programación hace un aporte particular a las posibilidades de integración de los componentes reusables a nuevos contextos. Sin embargo, los mayores aportes están relacionados al modelo de composición OO. No es de extrañarse que esto suceda por dos razones:

1. los conceptos del paradigma OO son el resultado de una evolución sobre el paradigma procedural
2. el paradigma OO abarca los conceptos del paradigma procedural

De esta forma, la naturaleza evolutiva y unificadora del paradigma OO lo convierte en un candidato adecuado para el reuso de código, utilizando un modelo de composición rico en mecanismos y posibilidades de interconexión de componentes. Podemos decir entonces, que existe una cierta afinidad entre el paradigma OO y el reuso de software.

Los pocos resultados experimentales muestran coherencia con esta afirmación y nos ayudan a reafirmar nuestra convicción sobre la afinidad que existe entre el paradigma OO y el reuso de software.

Queremos advertir que esto no significa que al utilizar el paradigma OO se obtengan automáticamente los beneficios del reuso de componentes. En primer lugar, hace falta un conocimiento sólido de la manera como los mecanismos del paradigma contribuyen a este objetivo y de cuándo y cómo utilizarlos. Por otro lado, es de vital importancia tener componentes reusables como capital de inversión y la infraestructura para su manejo adecuado: clasificación, búsqueda, recuperación y documentación. Adicionalmente, estos elementos técnicos son sólo el medio, no el fin, que permite estructurar metodologías para implementar el aspecto de la reusabilidad en el desarrollo de software.

6. Componentes OO de mayor nivel

El análisis realizado se ha centrado en el estudio de los mecanismos de programación que abarca el paradigma OO y su aporte al reuso de código desde la perspectiva de un modelo de composición, donde el modelo de componente es el objeto y la clase.

En este contexto, los componentes reutilizables son objetos y clases. Estos componentes representan un nivel bajo de abstracción. Sin embargo, La relación entre el paradigma OO y el reuso de software se puede explorar desde niveles mayores de análisis, donde surgen modelos de componentes OO de mayor granularidad. Estos son los niveles que realmente reportarán beneficios significativos.

6.1 Categorías de clases

Las clases por lo general están organizadas en jerarquías de herencia a las que se denomina categorías de clases. La categoría es un elemento de estructuración que permite organizar una parte de un dominio de un problema que tiene características comunes a través de la relación de herencia.

Estamos ante un nivel de abstracción bastante mayor que el de la clase, pero no podemos identificar un modelo de componente asociado para la categoría. La razón es que el modelo de componente sigue siendo la clase, cuando se quiere reutilizar una categoría de clases se instancian objetos de alguna de sus clases (reuso por composición) o se extiende la categoría mediante la herencia (reuso por derivación). El tipo de reuso más frecuente con las categorías de clases reutilizables es el reuso por composición porque se tienen interfaces estables y probadas.

El reuso de categorías de clases es probablemente la forma de reuso más frecuente en el desarrollo OO y se ubica en el nivel de reuso de código. Aunque cada categoría de clases corresponde a un diseño, por lo general representa poco ahorro de esfuerzo en este nivel, desde la perspectiva del desarrollo global de una aplicación. Aquí ubicamos a las categorías de clases más comunes: estructuras de datos, manejo de errores, procesamiento numérico, componentes de GUIs, etcétera.

6.2 Frameworks

Los frameworks OO están formados por una o varias categorías de clases que abstraen en conjunto el diseño y la implementación de un subsistema o de una parte importante de la arquitectura de un sistema o de un dominio (conjunto de sistemas). Algunos subsistemas que pueden ser adecuados para el desarrollo de un framework son: un editor gráfico de texto, un subsistema de facturación, un subsistema de instrumentación y adquisición de datos, etcétera.

Un framework define principalmente la organización de las estructuras de la arquitectura y las interfaces de interconexión. Para lograr este nivel de generalidad, los troncos de las jerarquías de clases por lo general están formado por clases concretas y las raíces por clases abstractas (no definen implementación). Además, un framework define restricciones para las posibles implementaciones de sus clases abstractas mediante clases que definen el flujo y la lógica interna de control, independientemente de la implementación concreta que se haga.

En definitiva, el framework conforma un esqueleto general para el desarrollo de un tipo de aplicación para que los usuarios la adapten a las demandas específicas de una aplicación concreta. La manera como los usuarios adaptan el framework son:

- *reuso por composición*: instanciando clases tal cual son proporcionadas.
- *reuso por derivación*: extendiendo el framework derivando nuevas clases e implementado la funcionalidad específica de la aplicación concreta.
- *reuso de la lógica de la arquitectura*: utilizando las clases de lógica y control de las maneras definidas por el framework.

Una característica peculiar que destaca de estas arquitecturas es que pareciera que se invierten los papeles entre el servidor y el cliente. En lugar de que el cliente reutilice el framework, lo que realmente sucede es que la lógica y el control del framework invoca las extensiones proporcionadas por el cliente para requerir servicios y datos específicos de la aplicación, lo que podría interpretarse como que el framework reutiliza al cliente.

Note que un framework es la construcción de un diseño de alto nivel *que tiene un código asociado* que se puede extender y modificar para instanciar una aplicación concreta. El nivel de abstracción alcanzado por un framework es muy alto y en consecuencia el nivel de reuso y beneficio que se obtiene es considerable. Por supuesto, la dificultad de construcción es también considerable, pues para construir un framework se debe tener un sólido conocimiento de toda la arquitectura del subsistema, de las interfaces y de las posibles variaciones de construcción que se presentan en aplicaciones concretas para poder abstraerlas en interfaces comunes y abstractas.

6.3 Patrones de diseño

El diseño de software reutilizable OO es difícil, en particular porque debe buscarse aislar los detalles de implementación y desacoplar al máximo los clientes de los servidores. En la práctica, ha hecho falta orientaciones que guíen la construcción para obtener este desacople. Por otro lado, las soluciones que se han encontrado en el desarrollo de sistemas OO para construir componentes reutilizables OO no han sido ampliamente divulgadas ni aprendidas. Estas soluciones están formadas por estructuras de objetos comunicándose que resuelven problemas particulares de diseño.

En este contexto aparece el concepto de ***patrón de diseño*** como un elemento que captura decisiones de diseño que se pueden reutilizar. "Patrones son una manera de registrar y codificar la experticia y experiencia de manera que otros puedan reutilizarla. Un diseñador familiar con tales patrones puede aplicarlos inmediatamente a problemas de diseño. *Un patrón describe una solución a un problema en un contexto particular de manera que otros puedan reutilizar esta solución una y otra vez*. Note que un patrón de diseño no describe un diseño particular para ningún sistema particular. En lugar de ello, se abstrae de muchos diseños y describe lo que es esencial, común e intrínseco a los problemas abordados y soluciones encontradas a través de todos los diseños" [Helm+ 95].

Un patrón de diseño se diferencia claramente de un framework, en primer lugar por el nivel de abstracción que aborda, un framework es el diseño y esqueleto de código de una arquitectura de un sistema o subsistema, mientras que un patrón de diseño es la solución a un problema particular de diseño que se puede reutilizar. Un framework es una solución de diseño acompañada de código, mientras que un patrón de diseño describe un diseño abstracto que puede tener múltiples implementaciones. Por otro lado, la implementación de un patrón de diseño usualmente involucra pocas clases, pues resuelve

un problema atómico de diseño. El patrón es un modelo de componente de mayor nivel que el objeto y la clase, que representa una solución de diseño reutilizable y que se puede utilizar para construir categorías de clases y frameworks reutilizables. Otra diferencia notable es que el framework es una solución para un dominio de aplicación particular, mientras que los patrones de diseño son soluciones que se presentan independientemente del dominio, son soluciones para hacer la construcción de componentes más flexibles, resistentes al cambio y reutilizables.

Un patrón de diseño es un componente reutilizable de diseño, que a su vez es la especificación de cómo desarrollar y empacar componentes reutilizables, son indicaciones de soluciones de diseño para problemas frecuentes en la construcción de software flexible y reutilizable. Los patrones de diseño especifican "...qué decisiones tomar, cuándo y cómo tomarlas y por qué son las decisiones correctas" [Beck 94]. Un patrón es una plantilla (template) de una solución, no es una solución, tiene las orientaciones para construir la solución.

En este sentido, podemos identificar instancias de un patrón de diseño, donde una instancia es una aplicación específica de un patrón para formar un modelo de objetos particular. El desarrollo futuro que se espera es tener bibliotecas de instancias de patrones para construir aplicaciones dentro de un dominio de aplicación [Coad 94].

El hecho de que un patrón de diseño sea un componente reutilizable de diseño implica que un objetivo básico que debe satisfacer el patrón es comunicar esas decisiones de diseño a los usuarios. En este sentido es fundamental disponer de un modelo de documentación adecuado que permita comunicar efectivamente las decisiones de diseño. Existen muchas propuestas de documentación de patrones, pero la información de un patrón debe contener como mínimo:

- *Problema*. Debe indicar el problema que resuelve, de manera que un usuario pueda decidir rápidamente si el patrón le puede servir para un problema particular.
- *Contexto*. Análisis del conjunto de restricciones que actúan sobre cualquier implementación concreta del diseño. Por lo general, existen restricciones exclusivas entre sí, p.e., eficiencia vs. flexibilidad, espacio vs. velocidad.
- *Solución*. La solución debe especificar qué hacer para resolver las restricciones planteadas en el contexto.

Existen esquemas de documentación bastante más completos, por ejemplo, en el catálogo de 23 patrones de diseño de [Gamma+ 94] utilizan la siguiente información: *nombre del patrón y clasificación, propósito, también conocido como, motivación, aplicabiblidad, estructura, participantes, colaboraciones, consecuencias, implementación, código muestra, usos conocidos y patrones relacionados*. Como puede apreciarse el formato de documentación del patrón debe ser lo más completo posible y registrar un conjunto amplio de información.

Además de las ya mencionadas, encontramos características particulares de los patrones de diseño. La primera es que el diseño que captura un patrón de diseño debe ser un diseño probado y utilizado en la práctica, es decir, ya debe haber mostrado beneficios significativos. Por otro lado, los patrones no funcionan aisladamente en el desarrollo de

una solución, colaboran con otros patrones para resolver problemas mayores. Cuando se identifica un conjunto de patrones que pueden colaborar, se habla de la existencia de una familia o de un lenguaje de patrones. Finalmente, es interesante notar que la proporción de código de infraestructura específico de los patrones es mucho menor que la cantidad de código que define la funcionalidad de la aplicación.

6.4 Evaluación

Hemos identificado tres niveles de componentes y abstracción diferentes para el modelo OO. Las categorías de clases reutilizables son estructuraciones de código que se incorporan a ese nivel de la construcción de la aplicación. Los frameworks son estructuraciones de arquitecturas de diseño y código que se incorporan a nivel de diseño y a nivel de código. Los patrones de diseño son estructuraciones de diseño que se incorporan a nivel de diseño, aunque también se pueden incorporar a nivel de código si existen las instancias de patrones que se requieren.

Cada uno representa un nivel diferente de abstracción y por lo tanto beneficios diferentes para el reuso. Desde la perspectiva de modelo de composición de código, notamos que ninguno aporta nuevos mecanismos de composición, finalmente la incorporación se hace utilizando las formas de reuso de composición, derivación y polimorfismo.

Desde la perspectiva de un modelo de composición de diseño el aporte es significativo. En el caso de los patrones de diseño, proporciona nuevas maneras de interconectar clases y objetos de manera que se minimice el acople y se obtengan construcciones más flexibles y reutilizables. En el caso de los frameworks, se conecta el diseño completo de la arquitectura de un sistema o subsistema. En ambos casos el modelo de componente es un modelo de documentación que explica el diseño. Para poder asociar mecanismos de composición específicos al modelo de componente a este nivel hace falta algo más tangible, por ejemplo la representación del diseño bajo una o varias notaciones que permita incluirla gráficamente a un diseño mayor con la ayuda de alguna herramienta automatizada.

Página intencionalmente en blanco

Clasificación de componentes reutilizables OO

Se propone un sistema de recuperación, búsqueda y clasificación de componentes reutilizables OO. Comenzamos explorando las generalidades de la organización de un depósito de componentes y revisamos algunos de los esquemas de clasificación utilizados para organizar componentes de software. Luego, proponemos una adaptación del esquema de facetas para clasificar clases. Partimos, del análisis de las características generales de las clases y estructuramos un esquema de facetas adecuado para tales propiedades. Finalmente, definimos los procesos de búsqueda y clasificación basados en la estructura propuesta. De particular importancia, se presentan medidas que permiten calcular el parecido entre dos componentes basados en sus características.

Para mostrar que el esquema propuesto es adecuado para la organización del conocimiento por dominios, se aplicó a un dominio particular: "Estructuras de Datos" (el contenido del esquema se muestra en el Anexo A).

1. Organización de un depósito de componentes

La tecnología de reuso por composición tiene como eje central una colección de componentes de software clasificada, lo que implica en primer lugar la existencia de un conjunto de componentes reutilizables y en segundo término, la existencia de una estructura de clasificación bien definida. Asumiremos que ya se dispone de una colección de componentes reutilizables y centraremos la discusión en los problemas básicos que implica el proceso de reuso alrededor de ellos.

La clasificación de la colección presenta el primer y principal problema a resolver: cómo será organizado el depósito para sustentar el reuso. Para ello, el depósito debe disponer de un ***sistema de clasificación e incorporación*** adecuado. A su vez, el soporte para el reuso demanda disponer de un proceso sistemático de asistencia para encontrar los componentes requeridos, es decir, necesita un ***sistema de búsqueda y recuperación***. Finalmente, la información de cada componente se debe presentar en un formato uniforme, coherente y de fácil acceso, para lo cual requerimos de un ***sistema de documentación***.

En este contexto, el usuario de la colección puede realizar dos funciones básicas: agregar y buscar componentes. Para agregar componentes, debe tener la asistencia del personal administrador de la biblioteca. Por otro lado, la búsqueda y recuperación de componentes es un proceso menos dependiente del personal de la biblioteca, aunque puede ser conveniente contar con su asistencia. Es de esperarse que el proceso de búsqueda sea mucho más frecuente que el de incorporación, pues la función fundamental del depósito es que utilicen sus componentes.

1.1 Reuso con modificación

¿Qué sucede si el usuario no encuentra lo que busca?. Tiene dos opciones:

- construir o adquirir lo que busca
- modificar un componente existente para que se adapte a sus demandas.

Por lo general, el reuso con modificación se refiere a una transformación del componente para satisfacer necesidades particulares, pero también podría satisfacer necesidades generales, en cuyo caso tendríamos la construcción de un componente reutilizable a partir de la modificación de otro componente reutilizable. El reuso con modificación implica un proceso de cuatro pasos:

1. buscar y no encontrar el componente requerido.
2. buscar el(los) componente(s) candidatos a modificar.
3. evaluar el esfuerzo que se requiere para hacer las modificaciones sobre el conjunto candidato.
4. modificar un componente.

La búsqueda de componentes candidatos es una búsqueda aproximada en el sentido que trata de encontrar componentes que se aproximen de alguna manera al que realmente estamos buscando. Esto tiene una demanda especial sobre el sistema de clasificación, pues la organización del depósito debe incluir la información que haga posible la búsqueda de componentes aproximados. Esta demanda se transmite al sistema de búsqueda que debe encontrar los componentes de acuerdo a la estructuración del depósito.

Existen dos formas básicas de buscar los componentes candidatos:

- *relajar las condiciones de búsqueda*: se busca sólo algunas de las características del componente deseado. La relajación la puede hacer el usuario o el sistema de búsqueda.
- *buscar componentes similares*: se trata de encontrar un componente parecido utilizando relaciones de similitud entre las características de los componentes, para ello el esquema de clasificación debe permitir establecer este tipo de relación entre componentes similares.

Buscar un componente candidato significa realmente buscar otro componente al deseado. Para ello, se transforma la búsqueda original en otra búsqueda. En el primer caso, reduciendo las características de búsqueda y en el segundo buscando características similares.

La búsqueda de componentes similares tiene un mayor impacto en la selección del sistema de clasificación y del sistema de búsqueda que el relajamiento de las condiciones de búsqueda, pero a cambio, el usuario obtiene mayor asistencia e información en el propio proceso de búsqueda.

Ciertamente, el beneficio es mayor si reutilizamos componentes sin modificación alguna, sin embargo, el reuso con modificaciones también puede representar beneficios significativos. El esfuerzo invertido en estas modificaciones sea menor que el esfuerzo requerido en la construcción o adquisición del componente nuevo.

La demanda de búsquedas aproximadas para las herramientas de la biblioteca sólo tiene sentido cuando se espera que el depósito tenga cantidades considerables de componentes, pues de otra forma los usuarios pueden identificar directamente los componentes candidatos una vez que estén familiarizados con el depósito.

1.2 Organización del depósito

La organización del depósito podría seguir un orden natural presente en el conjunto, o en caso de no existir tal orden o por razones de conveniencia, obedecer a un orden artificial impuesto. En cualquier caso, el orden implica una clasificación de los elementos involucrados.

Una ***clasificación*** establece categorías en un nivel abstracto dentro de un dominio de conocimiento con base en ciertas propiedades de interés, estableciendo un conjunto de reglas para identificar (catalogar) los objetos específicos del dominio en términos de las propiedades representadas por las categorías. El resultado de clasificar es el agrupamiento de cosas parecidas juntas, con la particularidad de que "todos los miembros de un grupo o clase comparten al menos una característica que no comparten los miembros de otra clase" [Prieto 89].

La clasificación u orden al que nos referimos es generada por una estructura bien definida a la que llamaremos ***esquema de clasificación***, la cual está compuesta por un conjunto de categorías abstractas que permiten agrupar objetos con propiedades similares. El beneficio inmediato de un esquema de clasificación es que, en el proceso de búsqueda, permite especificar ciertas propiedades que debe satisfacer un objeto más que buscar el objeto mismo, es decir, no es necesario tener una especificación completa del objeto para poder buscarlo en la colección.

Podemos visualizar el esquema de clasificación como un mapeo que toma un elemento de un conjunto de cierto dominio y le asocia un lugar específico en otro conjunto: un ***conjunto clasificado*** (*Fig. 3-1*). El esquema de clasificación no necesariamente asigna un único lugar a un componente en el conjunto ordenado. Por otro lado, es deseable que un lugar en el conjunto ordenado corresponda a un único componente del dominio, pues de lo contrario el esquema sería demasiado general para el dominio.

Los valores concretos que podemos asignar a las categorías abstractas del esquema de clasificación dependen del dominio de conocimiento a clasificar. Estos valores son palabras que denotan una semántica específica para el dominio, es decir, un esquema de clasificación está asociado a un conjunto de palabras acotado, un ***vocabulario*** que se utiliza generar un orden sistemático de una colección.

Cuando hablamos de clasificación de componentes de software reutilizables, estamos imponiendo características específicas de este dominio al esquema de clasificación:

- la organización de la colección debe hacerse utilizando los atributos específicos de los componentes de software para facilitar la especificación de la búsqueda
- como la colección está en continua expansión y los componentes no siempre se ajustan a las categorías establecidas por el esquema, debe ser posible modificar la

estructura de clasificación sin que tenga un fuerte impacto sobre la colección clasificada, es decir, sin tener que reclasificar los objetos existentes
- debe ser fácil de manejar para el usuario final y para el administrador del depósito
- debe proporcionar soporte para encontrar componentes similares
- debe ser fácilmente automatizable.

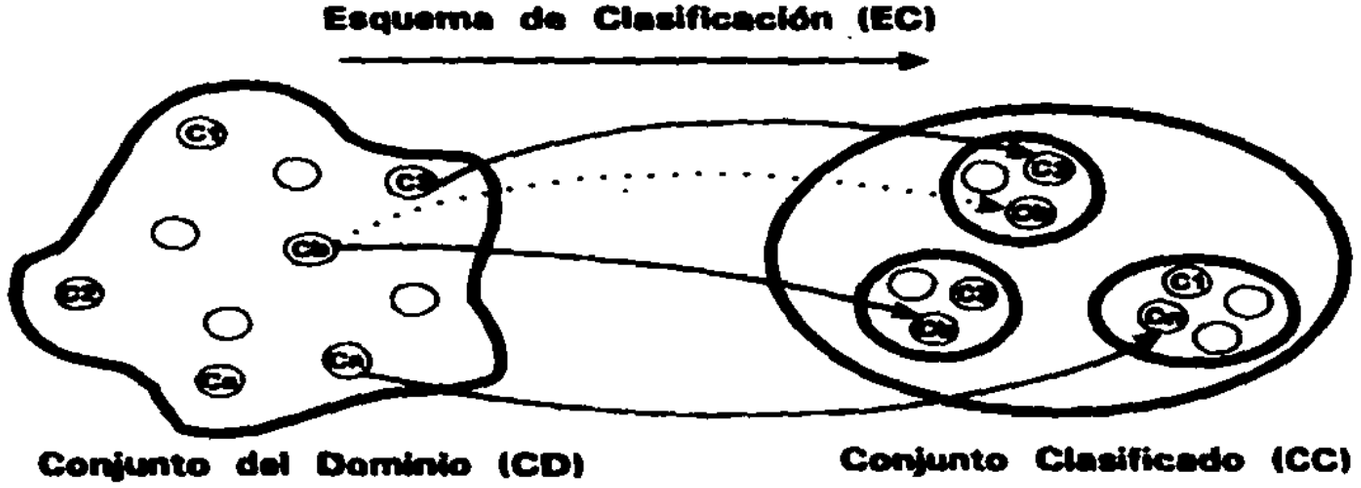


Fig. 3-1 Un esquema de clasificación es un mapeo a un conjunto ordenado

A continuación, examinaremos algunos de los esquemas de clasificación que se han utilizado para clasificar colecciones de componentes y evaluaremos si satisfacen las demandas descritas. Nos centraremos con más detalle en el esquema de clasificación de facetas porque lo adaptaremos para clasificar componentes OO reutilizables.

1.3 Esquemas de clasificación

Existen diferentes propuestas de clasificación de componentes reutilizables. Los diferentes esquemas de clasificación no son excluyentes y podría ser beneficioso utilizar más de uno en las herramientas que manipulan el depósito. Para el usuario final, puede significar mayor aprovechamiento de las oportunidades de reuso, pero para el administrador implica un mayor esfuerzo de mantenimiento. Si no es claro qué esquema de clasificación utilizar y existen diferentes tipos de usuarios potenciales del depósito, puede ser conveniente construir un prototipo inicial de la herramienta que incluya varios esquemas de clasificación y que el usuario final decida con cual se siente más a gusto.

Decidir cuál es el esquema de clasificación más adecuado para un depósito depende de muchos factores. Probablemente el más importante sea el sistema de búsqueda y recuperación asociado. Cada esquema de clasificación determina un orden del depósito y por lo tanto una manera particular de buscar componentes. Esto a su vez determina en gran medida el aprovechamiento que hará del depósito el usuario final. Por ello, es importante esta elección y debemos disponer de criterios que nos ayuden en este proceso:

Criterios para evaluar esquemas de clasificación

Representación semántica

la mayoría de las búsquedas son de carácter semántico, para buscar, especificamos lo que quiere decir el componente que deseamos. Los esquemas de clasificación que proporcionan mayor semántica están relacionados con un mayor esfuerzo en el proceso de clasificación y de mantenimiento (p.e., especificaciones formales).

Inserción de nuevos componentes

puede ser necesario modificar la estructura de clasificación cuando aparecen nuevos componentes que no se pueden clasificar con la estructura actual. La modificación de la estructura puede afectar el ordenamiento del depósito. Algunos esquemas son más sensibles que otros a los cambios de la estructura de clasificación (p.e., el esquema jerárquico es más sensible).

Automatización de la clasificación

si el depósito es grande y no hay suficientes recursos puede ser necesario automatizar el proceso de clasificación de componentes. Un esquema que se presta para esto es el de texto libre. Los procesos automáticos de clasificación aportan poca semántica.

Aprendizaje

es posible que el sistema de búsqueda "aprenda" de los resultados de búsquedas anteriores, de manera que modifique el proceso de búsqueda en función de su "experiencia". Esto es posible con esquemas que utilizan técnicas de inteligencia artificial.

Tipo de usuario

el nivel de experiencia y preparación de los usuarios del depósito pueden determinar el esquema a utilizar. También es importante considerar si los usuarios ya conocen el depósito y si la mayoría de los componentes ya han sido reutilizados.

Recursos

finalmente, los recursos de que se dispone pueden determinar el esquema de clasificación, p.e., cuando no se dispone de suficientes recursos, lo mejor es automatizar todo lo que se pueda.

Podemos distinguir diferentes tipos de esquemas de clasificación según difieran en los elementos que utilizan para clasificar. La mayoría de las propuestas utilizan un ***vocabulario índice*** compuesto por palabras que representan conceptos. La clasificación consiste en asociar (indexar) a un componente un conjunto de estas palabras.

Un ***vocabulario índice controlado*** está formado por un conjunto de palabras predefinidas que son derivadas de un análisis del dominio de clasificación para representar los conceptos relevantes. Adicionalmente, un vocabulario controlado proporciona un conjunto de relaciones predefinidas que organizan los conceptos en el esquema de clasificación, obteniendo una mayor expresividad semántica. Entre los esquemas de clasificación que utilizan un vocabulario controlado están el esquema jerárquico y el esquema de facetas.

También existen esquemas que utilizan un vocabulario no controlado como el análisis de texto libre, donde las palabras del vocabulario índice se derivan automáticamente del análisis de un texto según sus propiedades estadísticas y posicionales. De esta manera, se obtiene un proceso de indexación automática en el que a cada elemento se le asocia un conjunto de las palabras derivadas de sus textos. El análisis estadístico de textos se justifica cuando su tamaño es grande, p.e., para la documentación del software. Esta es una propuesta puramente sintáctica y no puede agregar mayor semántica que la existencia de ciertas palabras.

En el caso de los esquemas de clasificación que utilizan especificaciones formales, el proceso de clasificación es diferente. La especificación de un componente está compuesta de un conjunto de constructores de un lenguaje de especificación. En lugar de clasificar al componente se clasifica su especificación, se realiza una clasificación sintáctica y semántica. En este caso no hay un vocabulario controlado, lo que se hace es transformar (normalizar) la especificación a un formato común que permita hacer comparaciones con otras especificaciones. Las especificaciones formales permiten representar la semántica de los componentes y por lo tanto, también permiten hacer búsquedas semánticas.

1.3.1 Esquema jerárquico

Probablemente la organización más difundida para un depósito de componentes es una estructura jerárquica de categorías, generalmente definidas por dominios de aplicación. Cada componente se asocia a una o más categorías y típicamente el usuario inspecciona los componentes en el depósito guiándose por las categorías.

Como los nuevos componentes no siempre se ajustan a las categorías establecidas puede ser necesario modificar el esquema de clasificación. Esto provoca con frecuencia la reclasificación de una parte de la colección para mantener la consistencia semántica del ordenamiento. Adicionalmente, el uso de este esquema de clasificación presenta otros inconvenientes cuando se trata de colecciones grandes:

- el usuario debe conocer la categoría del componente para búsquedas eficientes.
- al modificar la estructura de clasificación el usuario debe reaprenderla, pues los componentes pueden haber cambiado de lugar.
- si la colección es muy grande puede ser poco práctico utilizar este procedimiento como mecanismo básico para la búsqueda.

Por otro lado, el esquema de clasificación podría ser adecuado para colecciones pequeñas, pues el método de búsqueda asociado (navegación libre), fomenta que los usuarios conozcan con mayor detalle el depósito y puedan identificar con facilidad las oportunidades de reuso para los componentes.

1.3.2 Esquema de palabras clave

En el esquema de análisis de texto libre las palabras del vocabulario se determinan automáticamente. Estas son palabras clave y cuando un usuario desea hacer una búsqueda especifica un conjunto de estas palabras para establecer la comparación con las clasificaciones del depósito. Las palabras clave también se pueden determinar manualmente, pero esto no cambia el hecho de que el vocabulario es no controlado.

En el mejor de los casos el total de palabras clave para un dominio debe ser un subconjunto del vocabulario controlado para el mismo dominio. Sin embargo, por lo general esto no sucede así, aunque la intersección no sea vacía. Por otro lado, estos sistemas no pueden determinar si dos palabras clave representan el mismo concepto.

Finalmente, el uso de palabras clave es atractivo por su posibilidad de automatización y la simplicidad de su uso para la clasificación y la búsqueda. Aún sin la presencia de un vocabulario controlado, este esquema se podría extender para representar relaciones entre las palabras clave, de manera que se obtenga un mayor nivel de representación semántica.

1.3.3 Esquema de hipertexto

Un sistema hipertexto básicamente es una agrupación de nodos unidos por una red de enlaces, conformando una estructura de gráfica dirigida cíclico. Un enlace denota una relación entre dos nodos. A este enlace podemos asignarle información del tipo de relación que denota e inclusive una medida que muestre la intensidad del enlace (cuando aplique). De esta forma, podemos representar diferentes relaciones entre nodos.

Un sistema de hipertexto puede ser adecuado como herramienta de documentación para cualquier esquema de clasificación, pues se puede utilizar para enlazar todos los productos que comprenden cada componente (p.e., código fuente, pruebas, documentación, notas de uso) y así, proporcionar un acceso uniforme a cada una de sus partes.

Adicionalmente también se puede utilizar como un esquema de clasificación. Para ello se utilizan nodos que denotan categorías y enlaces de clasificación de un nodo componente a un nodo categoría. Los enlaces de clasificación de un nodo componente conforman su clasificación.

En este sentido, el sistema de hipertexto es un esquema de clasificación muy flexible, no hay limitación en el número o tipo de categorías que se pueden definir ni en el número de enlaces que se puede establecer entre un nodo componente y los nodos categoría. La desventaja de ser tan flexible, es que la responsabilidad de la consistencia semántica de las clasificaciones depende totalmente del usuario del esquema.

Con el sistema de hipertexto se puede representar cualquier tipo de relación entre componentes, en principio binarias, pero se puede extender a relaciones n-arias si se agregan nodos que representen relaciones. Gracias a esta cualidad podríamos implementar diferentes esquemas de clasificación mediante hipertexto (inclusive simultáneamente) y disponer de los diferentes mecanismos de búsqueda asociados, por ejemplo:

- ***esquema de clasificación jerárquico***: se definen nodos de categorías enlazados entre sí donde una categoría puede pertenecer como máximo a otra categoría. Los nodos componentes se enlazan a las categorías raíz dentro de esta estructura.
- ***esquema de clasificación por palabras clave***: se definen nodos de palabras clave a los cuáles se enlazan los componentes.
- ***esquema de clasificación de facetas***: se definen nodos de facetas y de términos, donde un término pertenece a una faceta. Los nodos componentes se deben enlazar a un término de cada faceta.

1.3.4 Especificaciones formales

Aunque las especificaciones formales no se utilizan masivamente en el desarrollo de software, éstas son prácticamente imprescindibles en algunos sistemas grandes y complejos (p.e., sistemas de control de tráfico aéreo).

Desde la perspectiva de reuso en estos sistemas, el objetivo básico es identificar automáticamente de un depósito de componentes, aquellos que se ajustan a las especificaciones del sistema. Para ello, se requiere de tres elementos básicos: los componentes en el depósito deben tener una especificación formal, los componentes que se desean buscar se describen mediante una especificación formal, el sistema de búsqueda y recuperación debe poder comparar ambas especificaciones.

Para buscar un componente en el depósito se debe construir la especificación formal del componente y buscar una especificación equivalente en el depósito. Podemos distinguir dos partes de una especificación formal, una especificación sintáctica y una especificación semántica. Por lo tanto, también podemos hacer dos tipos de búsquedas, una búsqueda sintáctica y otra semántica.

Algunos elementos que se pueden utilizar para la comparación sintáctica son: el número y tipo de parámetros de entrada y de salida, el número de excepciones, el número de variables de estado, complejidades de algoritmos, comparación de operadores del componente. No todos los elementos a comparar están definidos de manera explícita en la especificación buscada, por lo que hay que normalizar la especificación para que se ajuste a un formato uniforme de búsqueda.

Cuando se construye la especificación semántica, existen varias formas posibles de describir el significado del componente. Esto dificulta la comparación semántica porque deben encontrarse equivalencias entre especificaciones diferentes. Para ello, se aplica un proceso de normalización semántica que consiste en transformaciones que preservan el significado de las descripciones, de manera que dos descripciones equivalentes se reduzcan a una forma común que permita hacer la comparación. Las técnicas de especificación y simplificación algebráicas tienen un avance importante en este sentido.

Como las comparaciones semánticas son complejas, es conveniente hacer primero una comparación sintáctica que particione el depósito y reduzca el número de componentes candidatos. En este contexto, el esquema de clasificación de componentes está formado por los procesos de normalización sintáctica y semántica, donde la clasificación de un componente es la normalización sintáctica y semántica de su especificación.

1.3.5 Esquema de clasificación de facetas

El esquema de clasificación de facetas fue propuesto originalmente por [Ranghanathan 67] como una alternativa al problema de clasificación de material bibliográfico, actualmente es común su uso en bibliotecas en Europa y la India.

El esquema de clasificación de facetas está compuesto por un conjunto de categorías llamadas ***facetas***, donde cada faceta denota un tipo de propiedad o aspecto de los objetos del dominio. Cada faceta puede adoptar como atributo-valor un ***término***, que es una palabra que representa el valor concreto de la propiedad denotada por la faceta. De esta manera, un objeto se identifica/clasifica con un conjunto de términos, donde cada término es un atributo-valor de la abstracción representada por cada faceta.

Las facetas del esquema de clasificación dependen de los objetos que se desea clasificar (p.e., material bibliográfico o componentes de software) y deben abarcar todos los tipos de propiedades de todos los objetos, donde cada tipo de propiedad se representa con una faceta. El uso de este esquema de clasificación presupone que todos los objetos a clasificar comparten las mismas categorías de propiedades. Por este motivo, las facetas son relativamente estáticas para el conjunto a clasificar.

Por otro lado, para cada faceta existe un conjunto delimitado de términos, denominado ***espacio de términos***. Para clasificar un objeto se selecciona uno o más términos para cada una de las facetas del esquema. Una propiedad interesante de este esquema de clasificación es que no es necesario determinar todos los términos (propiedades concretas) de todas las facetas (categorías de propiedades) para comenzar a utilizarlo, se pueden ir agregando propiedades en la medida que se vayan descubriendo. En este sentido, el espacio de términos tiene características más dinámicas que las facetas, sin embargo, es conveniente definir un espacio de términos inicial suficientemente amplio.

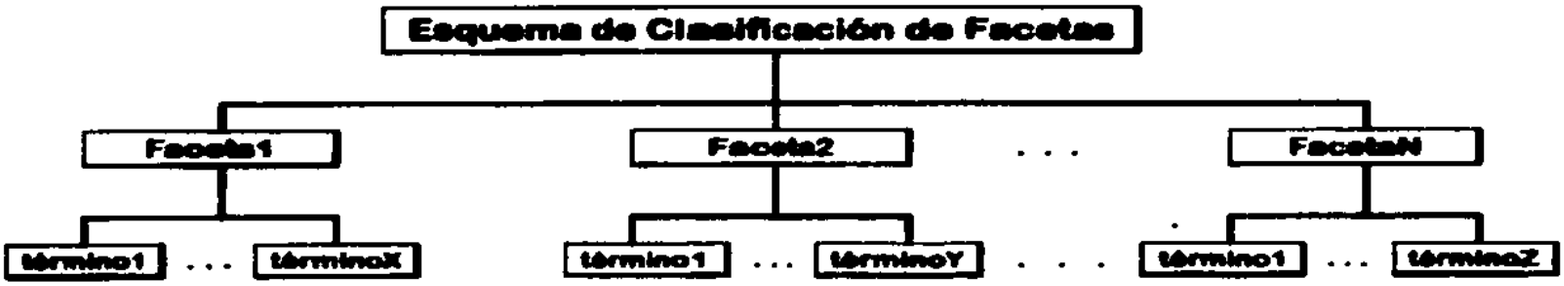


Fig. 3-2 *Las facetas agrupan los términos del esquema de clasificación*

De esta manera, el método de clasificación de facetas clasifica un objeto determinado, sintetizando sus propiedades fundamentales (facetas) a partir de un conjunto de palabras elementales (términos). Adicionalmente, el orden dado a las facetas puede reflejar el orden de importancia que se le otorga a cada propiedad del objeto.

1. Clasificación de componentes reutilizables utilizando facetas

La clasificación de componentes de software utilizando el esquema de facetas fue propuesto originalmente por Rubén Prieto Díaz en [Prieto 85], y el primer artículo al respecto fue publicado en [Prieto 87] y luego reimpreso en [Prieto 89] con algunas modificaciones. La aplicación del esquema se realizó sobre componentes funcionales (funciones, procedimientos) con cierta independencia de acción, y se implementó una medida de similitud entre componentes.

El esquema de clasificación de facetas asocia un descriptor (*Dcomp*) a cada componente de software. El descriptor está compuesto de un término (*Tij*) por cada faceta (*Fi*) seleccionado del espacio de términos correspondiente a la faceta que caracteriza. Así, se constituye una tupla de términos donde cada término es un atributo/valor de una faceta determinada. De esta manera, el esquema de clasificación de facetas construye un conjunto ordenado que contiene un descriptor asociado para cada componente.

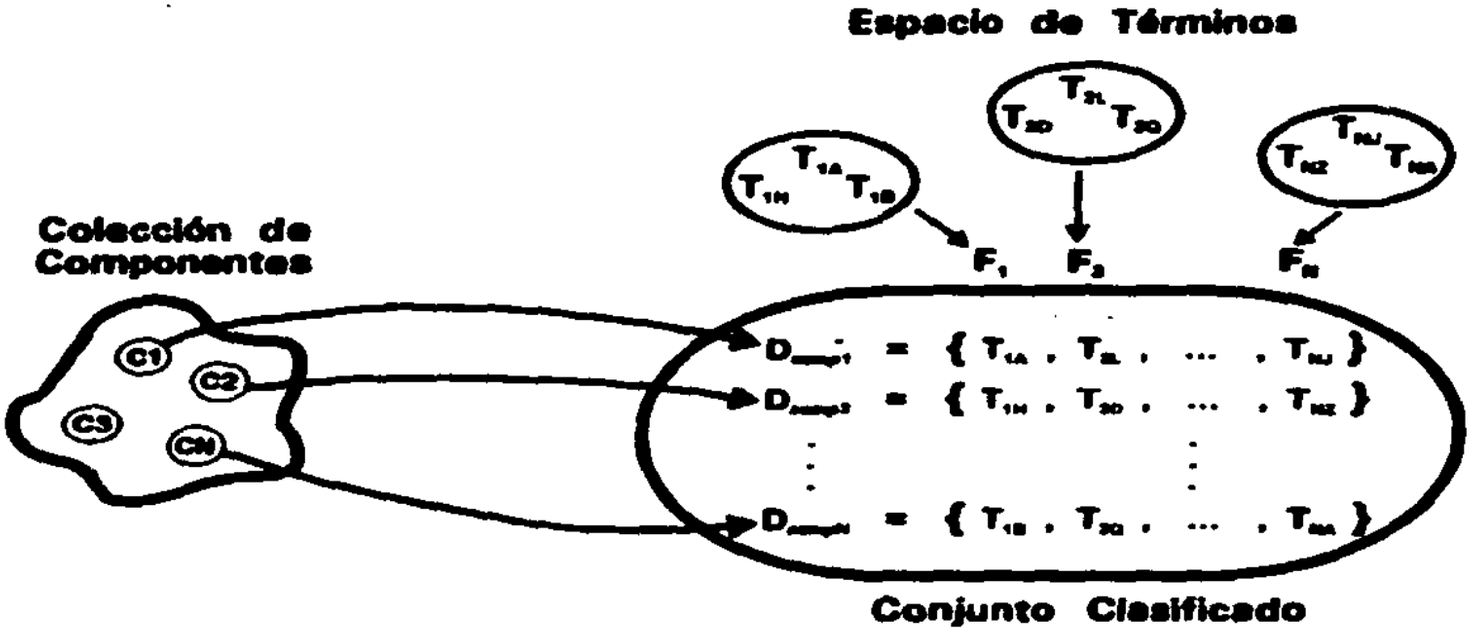


Fig. 3-3 El esquema asocia un descriptor a cada componente

Cada término se puede asociar con uno o más sinónimos, de manera que se pueda clasificar y especificar búsquedas con el término o cualquiera de sus sinónimos. Para consideraciones del esquema todos los sinónimos de un término denotan la misma semántica, sin embargo, para el usuario del esquema puede ser más significativo el uso de uno de los sinónimos en lugar del término mismo. El usuario se sentirá más cómodo, pero el esquema normaliza el descriptor reemplazando cualquier sinónimo por el término asociado. Esta es una característica de un vocabulario controlado, se agrupan todos los sinónimos (*Sij*) y el sinónimo que mejor describa el concepto se selecciona como el término representativo, es decir, pasa a ser un término (*Tij*) del esquema.

- **Relaciones entre componentes**

Se pueden establecer relaciones entre los términos dentro de cada faceta. Estas relaciones entre los términos del dominio representan de manera indirecta relaciones entre los objetos que son clasificados con ellos. Se pueden establecer diferentes relaciones de conexión entre términos, por ejemplo: *sinónimo / antónimo, similar-a, generalización / especialización*. Por otro lado, debe tomarse en cuenta que un mayor número de relaciones aumenta la expresividad de la estructura de clasificación, pero también "...la estructura será menos intuitiva para la persona que la usa y más difícil de mantener" [Karlsson 93].

En particular, nos interesan las relaciones que representen el parecido entre los componentes. En su propuesta original en [Prieto 89] y [Prieto 91], define un componente similar como aquel que contiene términos que denotan cierta cercanía a los conceptos representados por los términos del componente buscado, y se establece relaciones cuantificables de ***cercanía conceptual*** entre los términos de cada faceta.

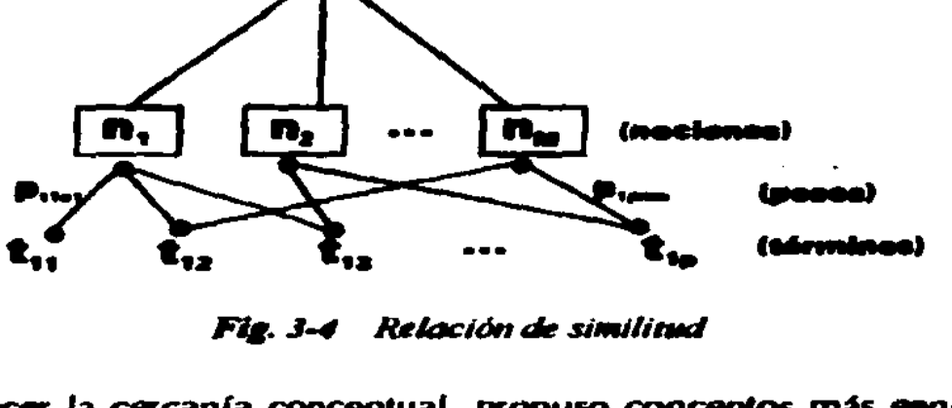


Fig. 3-4 *Relación de similitud*

Para establecer la cercanía conceptual, propuso conceptos más generales llamados ***nociones*** en el espacio de términos, para relacionar dos o más términos. Una noción es una abstracción intermedia entre la faceta y los términos, que permite establecer una medida (un peso) de cercanía conceptual entre un término y una noción. La representación global de similitudes entre términos de una faceta se logra mediante un ***grafo de distancia conceptual***. Este es un grafo dirigido ponderado acíclico en el que las hojas son los términos y los nodos son las nociones.

De esta manera, la gráfica de distancia conceptual proporciona un medio para medir la similitud entre los términos en una faceta, donde un concepto general en la forma de noción relaciona dos o más términos de una faceta a través de aristas ponderadas. La similitud puede ser calculada sumando los pesos de distancia conceptual recorridos por el camino más corto entre dos términos, a través del grafo.

- **Mecanismo de búsqueda y recuperación asociado al esquema de facetas**

Para buscar un componente se crea un descriptor de consulta seleccionando términos y sinónimos del esquema de facetas. Dado el descriptor de un componente, el sistema lo normaliza reemplazando sinónimos por términos. Seguidamente trata de encontrar el término asignado en la primera faceta, si no encuentra un término idéntico en ningún descriptor en la colección, el sistema busca los términos más cercanos para esa faceta. Luego procede de la misma manera para los términos asignados a las otras facetas. De esta manera, en caso de no encontrar todos los términos de una especificación en una descripción en la colección, recupera descripciones de componentes similares. El hacer una búsqueda de un término cercano implica construir un nuevo descriptor utilizando la gráfica de cercanía conceptual, sobre el que se va a hacer la nueva búsqueda.

- **Ventajas del esquema de clasificación de facetas para colecciones de componentes reutilizables**

Clasificar componentes reutilizables implica organizar una colección en continua expansión, pues crece el número de componentes y evolucionan sus características. Cuando aparecen nuevas características es necesario modificar el esquema de clasificación para que pueda clasificar los nuevos componentes. La ventaja del esquema de clasificación de facetas es que al modificar el esquema no se necesita reclasificar los objetos que ya están clasificados, sencillamente se agregan nuevos términos al espacio de términos que se utilizan con los nuevos componentes.

En este misma línea, existe otra ventaja considerable porque el esquema de clasificación de facetas se puede definir utilizando una muestra representativa de la colección, no es necesario analizar toda la colección de objetos existentes y posibles a futuro para definir el esquema.

El esquema de facetas es más efectivo para colecciones de dominio específico que para colecciones heterogéneas, pues un esquema de facetas para una colección diversificada, viene a ser demasiado general, perdiendo su precisión descriptiva.

Podemos resumir las principales ventajas del esquema de clasificación para clasificar colecciones de componentes en las siguientes:

- ✓ el esquema de clasificación se deriva del análisis de una muestra representativa de la colección, no se requiere analizar todo el universo.
- ✓ es sencillo expandir el esquema, se añaden nuevos términos, sin tener que reclasificar los objetos existentes.
- ✓ permite definir una medidas de similitud.
- ✓ el usuario puede especificar los requerimientos del componente en lenguaje "natural", utilizando las palabras significativas del dominio.
- ✓ proporciona un vocabulario estándar que es común al administrador de la biblioteca y al usuario final.
- ✓ las diferentes facetas permiten buscar un componente mediante diferentes propiedades según los intereses del usuario.
- ✓ es fácilmente automatizable.

2. Esquema de clasificación de facetas para componentes reutilizables OO

Las ventajas del esquema de clasificación de facetas nos motiva a utilizarlo para organizar colecciones de componentes OO reutilizables. Sin embargo, su uso no es automático. Este tipo de componentes poseen características particulares que deben ser capturadas por el esquema de clasificación. Adicionalmente, la modificación del esquema de clasificación implica modificar los procesos de clasificación y de búsqueda. El esquema propuesto se aplicó al dominio de estructuras de datos y los resultados se muestran en el *Anexo A*.

2.1 Facetas para clasificar componentes OO

La propuesta original en [Prieto 87] se concentró en componentes procedurales (procedimientos, funciones, subrutinas) y el esquema de clasificación básico lo definió con tres facetas:

- *función*: denota la acción del componente (p.e., buscar, mover, comparar, copiar)
- *objeto*: objeto sobre el que se realiza la acción (p.e., enteros, archivos, funciones)
- *contexto*: denota el medio en el que se realiza la acción (p.e., archivo, tabla, arreglo, base de datos)

Los componentes reutilizables OO encapsulan un nivel de granularidad diferente y representan otro modelo de componente. El esquema de facetas debe satisfacer un doble propósito: caracterizar el dominio y clasificar los componentes del dominio. El objetivo de la adaptación del esquema de facetas que proponemos busca construir un sistema de clasificación, búsqueda y recuperación adecuado para componentes OO. La consecuencia inmediata de este enfoque es que obtendremos una estructura de organización del conocimiento del dominio que está basada en el modelo de objetos, es decir, el resultado será un modelo OO para organizar el dominio.

Un punto de inicio coherente para determinar las facetas que clasificarán a los componentes, surge del análisis de las características generales de una clase. Una clase está compuesta básicamente por:

- un *nombre* que representa una abstracción del dominio.
- un conjunto de *servicios* que caracterizan el comportamiento del componente.
- una *estructura interna* o representación en la computadora.
- un conjunto de *relaciones* con otras clases:
  - relación de *herencia* que organiza las clases por categorías.
  - relación de *composición* en la estructura de la clase.
  - relación de *uso* cuando requiere instanciar objetos para implementar sus servicios.
  - relación de *manipulación* con los objetos que requiere como parámetros en la invocación de sus operaciones.
- una *documentación* que explique la abstracción y el contexto de su uso.

Con base en estas características estructuramos un conjunto de facetas para la clasificación de componentes reutilizables OO, que los explicaremos detalladamente.

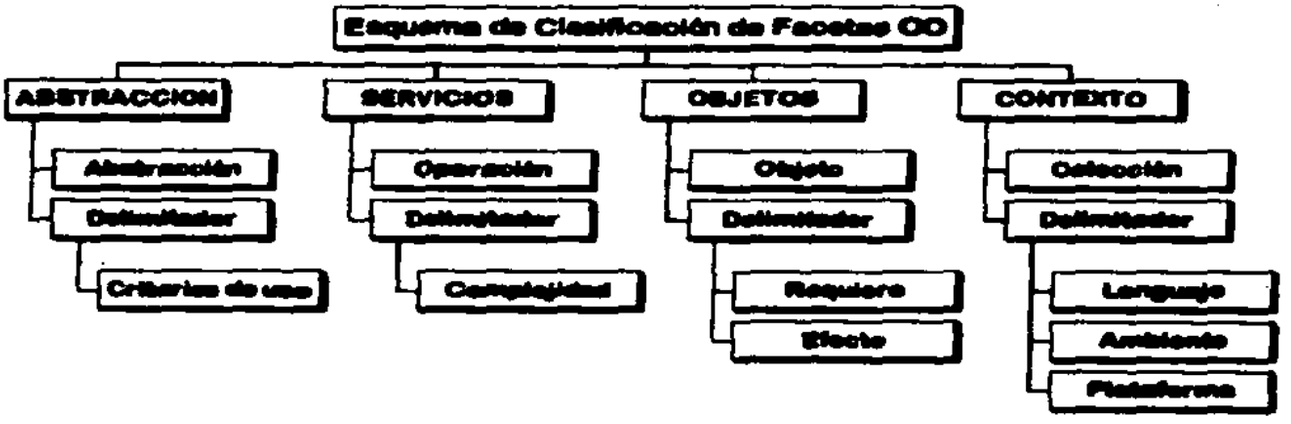


Fig. 3-5 *Esquema de clasificación de facetas para componentes reutilizables OO*

2.1.1 Faceta: *Abstracción*

La característica más importante de una clase es la abstracción que representa. El nombre, aunque no siempre es adecuado, puede ser ilustrativo para caracterizar la abstracción. El nombre puede estar formado por una o más palabras que pueden tener los siguientes roles:

- ***Abstracción***: el nombre identifica la abstracción del dominio con un sustantivo (colección, ventana, figura).
- ***Cualidades***: en el nombre puede añadirse cualidades que delimitan la representación de la abstracción con adjetivos (colección ordenada, figura geométrica) e inclusive con sustantivos (ventana de menú).
- ***Comportamiento***: en el nombre también puede estar representada la función global de la abstracción con sustantivos derivados de verbos (sincronizador de ventanas, manejador de errores, coordinador, administrador de memoria). Los verbos se utilizan generalmente para nombrar las operaciones de la clase y es raro encontrarlos en el nombre de una clase.

Además del nombre existen otros elementos que podemos determinar de las operaciones que conforman el comportamiento de la clase, de sus relaciones con otras clases y de su documentación. En cualquier caso, podemos caracterizar una abstracción del dominio con los siguientes elementos:

- ***Abstracción***: sustantivo que denota la abstracción principal representada.
- ***Delimitador***: uno o más modificadores (adjetivos y/o sustantivos) que denotan características particulares para el sustantivo. El delimitador acota el ámbito de la abstracción.

- ***Criterios de uso***: recomendaciones sobre cuando utilizar la abstracción. Denota el contexto de uso de la abstracción, por lo que es un delimitador.

1. Consideraciones

- **Nombres de clases**

Una abstracción se puede implementar con diferentes clases y los nombres de cada clase no necesariamente coincidirán. Un ejemplo frecuente de este caso se presenta cuando se utilizan diferentes bibliotecas de clases para un mismo dominio, bien sea porque se necesitan para diferentes plataformas o lenguajes o sencillamente por diferentes preferencias de los programadores. Los nombres de las clases que implementan una misma abstracción fueron asignados por sus desarrolladores por lo que seguramente diferirán, pero deben clasificarse con una misma abstracción. En concesuencia, no es adecuado utilizar los nombres de las clases como términos.

- **Tipos de clases**

En general, podemos distinguir dos tipos de clases en una colección:

- ***Principales***: representan las abstracciones principales del dominio del problema. La mayoría de éstas, sino todas, se reutilizan con frecuencia. Todas las clases principales se deben clasificar y cada una con una abstracción diferente. Las clases principales por lo general no están en las hojas de la jerarquía de herencia y pueden ser derivadas de clases principales y/o secundarias. Lo importante es la abstracción que representan en el dominio del problema desde un punto de vista conceptual, no de implementación.
- ***Secundarias***: las clases secundarias son de dos tipos:
  - ***dominio del problema***: representan características particulares o especiales para las abstracciones representadas por las clases principales y/o otras clases secundarias (p.e., lista doblemente ligada). Una clase secundaria se debe clasificar con una de las abstracciones que se utilizaron para clasificar una de sus clases principales (pueden ser varias si hay herencia múltiple).
  - ***dominio de la solución***: se utilizan como auxiliares para la implementación de las clases principales y/o secundarias (p.e., índice). Por lo general, no es de interés su clasificación, pues se utilizan como auxiliares en el reuso de otras clases.

La clase abstracta o diferida generalmente se utiliza para proporcionar flexibilidad en el reuso de la jerarquía, bien sea porque la abstracción puede tener muchas implementaciones o que no se pueden anticipar ciertos detalles de implementación. Con frecuencia, encontramos que una clase abstracta es una clase principal. Es importante que el esquema permita indicar si una clase es o no abstracta, pues una clase diferida requiere un esfuerzo adicional de implementación.

2.1.2 Faceta: *Servicios*

Los servicios (métodos u operaciones) representan las características funcionales de una clase. El comportamiento que proporciona un método lo podemos identificar por:

- ***Operación***: verbo que denota la abstracción funcional del método (insertar, remover).
- ***Delimitador***: modificador que delimita el ámbito de la operación (insertar al principio, remover todas las ocurrencias, buscar en tiempo *logn*).

Utilizar criterios de uso para cada servicio parece ser excesivamente detallado.

No es necesario clasificar todos los métodos de una clase, el criterio a aplicar es la relevancia de la operación para la abstracción y para el reutilizador. Otro factor relevante en la clasificación es que si una clase es derivada de otra, también hereda sus métodos. La herramienta debería poder representar esta relación de dependencia respecto a sus clasificaciones.

1. Consideraciones

- **Nombres de métodos**

Al igual que en el caso de la abstracción de la clase, la abstracción funcional de un método se puede implementar con diferentes nombres de funciones, por lo que no conviene utilizar los nombres de los métodos para clasificarlos. Por ejemplo, la operación *insertar* en una estructura de datos puede implementarse como *insertar*, o *agregar*.

- **Tipos de métodos**

Podemos identificar cuatro tipos de métodos desde la perspectiva de tipos de datos abstractos (TDA):

- *constructoras*: instancian un objeto de la clase
- *destructoras*: destruyen un objeto de la clase
- *transformadoras*: modifican el estado del objeto
- *observadoras*: no modifican el estado del objeto

Por lo general, las operaciones constructoras y destructoras no deben clasificarse pues no representan un criterio importante de reuso y todas las clases deben proporcionar una manera de instanciar y destruir objetos, bien sea implícita o explícitamente. Es importante identificar si una operación es transformadora u observadora.

- **Casos de clasificación**
  - Una clase puede proporcionar varios métodos para que el usuario los utilice en conjunto para realizar una operación mayor. En este caso, se debe clasificar el método de mayor nivel y no los métodos individuales. Cuando varias operaciones mantienen una relación de dependencia a través de una secuencia de ejecución se puede identificar una operación de mayor nivel.
  - En un método se puede identificar más de una operación, por ejemplo, en una prueba de membresía en una estructura de datos que regresa la posición del

elemento, se puede identificar la operación de membresía y la operación que regresa la posición. Es conveniente clasificar cada operación por separado si el usuario podría estar interesado en usar cualquiera de las dos operaciones.
- Es conveniente identificar si un método es o no diferido pues implica un esfuerzo adicional de implementación.

2.1.3 Faceta: *Objetos*

Contiene la información sobre los objetos que manipula. En particular interesa el efecto de las operaciones sobre los objetos que contiene la clase (que forman parte de su estado) y sobre los objetos que se le proporcionan como parámetros actuales en la invocación de los métodos. La manipulación de objetos la podemos identificar con:

- ***Objeto***: abstracciones de los objetos que manipula. En general, el conjunto de objetos que puede manipular está delimitado por las abstracciones del dominio más los tipos de datos básicos.
- ***Requiere***: denota las características que requiere de los elementos que va a manipular.
- ***Efecto***: denota el tipo de manipulación sobre el objeto: consulta, modifica, crea.

Nuevamente, la manipulación de objetos que interesa clasificar son únicamente las relevantes en el contexto del dominio. No es nada sensato indicar que un método modifica el valor de un parámetro poco relevante, no aporta nada a la clasificación.

1. Consideraciones

El ocultamiento de información sugiere que los detalles de implementación deben permanecer ocultos. Si una clase proporciona acceso directo (público) a su estado, existen operaciones implícitas que manipulan estos objetos. Una solución posible para representar esta situación es clasificar estas operaciones implícitas.

2.1.4 Faceta: *Contexto*

Para cada dominio, puede haber varias colecciones (bibliotecas) de clases que implementen total o parcialmente el dominio. Cada colección puede tener un contexto igual o diferente de uso que lo podemos representar con:

- ***Colección***: cada colección tiene un nombre (p.e., estructuras de datos para C++: COOL, NIH, LEDA, STL).
- ***Lenguaje***: lenguaje de programación (p.e., C++, SmallTalk, Java, Eiffel).
- ***Ambiente***: ambiente requerido (p.e., Motif, ODBC, ORACLE).
- ***Plataforma***: plataforma requerida (p.e., Windows NT, SCO Unix, HP UX, Macintosh, NextStep).

A diferencia de las tres primeras los valores que puede adoptar esta faceta no dependen del dominio. Podemos decir entonces que existen dos tipos de facetas: *facetas de dominio* (dependen del dominio) y *facetas de colección* (dependen de la colección a clasificar). Coherentemente, podemos identificar *términos de colección* y *términos de dominio*.

2.1.5 Consideraciones

En el proceso de estructuración del esquema de clasificación, encontramos una particularidad especial que se repite para cada faceta:

- En cada faceta podemos identificar grupos de términos que representan la ***abstracción*** principal (*abstracción, operación, objeto, colección*) y grupos de términos cuya finalidad es ***delimitar*** el alcance de la abstracción.

2.2 Relaciones para clasificar componentes OO

Existen algunas relaciones intrínsecas a la estructura del esquema de clasificación:

- cada esquema tiene un conjunto de facetas
- cada faceta tiene un conjunto de términos
- cada término puede tener uno o más sinónimos

Adicionalmente, hemos encontrado otras relaciones de interés entre términos que incrementan las posibilidades de representación y las posibilidades de búsqueda del esquema de clasificación. Estas son ***relaciones binarias***, es decir, relaciones que establecen una conexión entre exactamente dos términos.

Cualquier relación binaria la podemos clasificar en dirigidas y no dirigidas. Una relación es ***dirigida*** cuando es importante el sentido de la conexión, es decir, la semántica de la conexión cambia cuando se lee en un sentido o en otro. Cuando la relación es ***no dirigida***, la semántica de la conexión es independiente del sentido de lectura. Algunas características de las relaciones dirigidas son:

- existe un ***término origen*** y un ***término destino***.
- una relación dirigida es ***bidireccional*** cuando la misma relación se define en ambos sentidos para el mismo par de términos.
- una relación dirigida bidireccional es equivalente a una relación no dirigida, pues la semántica de la conexión no cambia con el sentido de la lectura.

2.2.1 Relaciones

Hemos encontrado cuatro relaciones binarias de importancia (***generalización-de, parecido-a, implica-a, excluye-a***). Las propiedades generales de estas relaciones son:

- ***Ortogonalidad***: las relaciones son ortogonales, es decir, dados un par de términos sólo se puede establecer un tipo de relación entre ellos.
- ***Participación múltiple***: un término puede participar en los cuatro tipos de relaciones simultáneamente. El resultado es un red de términos y relaciones que podemos representar con la estructura de datos ***gráfica dirigida***.
- ***Ciclos***: no se permite ciclos en los enlaces para un mismo tipo de relación. Un ciclo frecuente que se presenta entre relaciones de diferente tipo es una relación entre dos términos que tienen un ancestro común.

- ***Alcance***: para mantener la simplicidad en la construcción y mantenimiento del esquema, las relaciones sólo se pueden establecer entre términos de una misma faceta.

i. Relación: *generalización-de*

Denota una relación de clasificación entre términos. Es una relación dirigida donde el término origen clasifica al término destino. La relación interpretada en el sentido inverso recibe el nombre ***"especialización-de"***. Algunos sinónimos que pueden utilizarse son:

• *generalización-de*	abstracción-de, padre-de, categoría-de
• *especialización-de*	particularidad-de, hijo-de, elemento-de, es-un
• *término origen*	clasificador, padre
• *término destino*	especializador, hijo, particularidad

Este tipo de relación es similar a la relación de herencia entre clases. La diferencia básica es que la relación entre términos es una relación de conceptos y su principal finalidad es la agrupación de otros conceptos (términos).

Para mantener la simplicidad del esquema haremos las siguientes restricciones:

- un término puede estar agrupado a lo más por un término, es decir, puede tener un sólo padre.
- no se permiten ciclos en la agrupación de términos.

☞ ***la existencia de un término implica la existencia de todos sus ancestros.***

ii. Relación: *parecido-a*

Representa la similitud que puede existir entre dos términos, es decir, cuánto se parece un concepto o característica a otro. Es una relación no dirigida por cuanto ambos se parecen entre sí.

La relación de parecido puede tener grados de similitud, es decir, dos términos se pueden parecer entre sí con mayor o menor intensidad. Esta medida de similitud es muy subjetiva, por lo que nos parece conveniente restringir los valores a un conjunto discreto que sea indicativo: ***{alto, medio, bajo}***. En la construcción del esquema de clasificación para el dominio de estructuras de datos sólo se utilizaron los dos primeros valores. Es lógico, pues no tiene mucho sentido establecer una relación de parecido cuando el parecido es bajo.

iii. Relación: *implica-a*

la existencia de un término implica la existencia de otro término. Es una relación dirigida, donde el término origen implica la existencia del término destino. La relación interpretada en el sentido inverso recibe el nombre de ***"implicado-por"***. Algunos sinónimos que pueden utilizarse son:

• *implica-a*	determina-a
• *implicado-por*	determinado-por
• *término origen*	implicante, determinante
• *término destino*	implicado, determinado

Si la relación es bidireccional, ambos términos no pueden existir por separado en un descriptor.

- ☞ *cuando un término implica a otro, éste ni ninguno de sus implicados puede excluir al original o a un ancestro del original.*

iv. Relación: *excluye-a*

La existencia de un término implica la no existencia o exclusión de otro término. Es una relación dirigida donde el término origen excluye al término destino. La relación interpretada en el sentido inverso recibe el nombre de ***"excluído-por"***. Algunos sinónimos que pueden utilizarse son:

• *excluye-a*	descarta-a, rechaza-a
• *excluído-por*	descartado-por, rechazado-por
• *término origen*	excluyente, descartador, rechazador
• *término destino*	excluído, descartado, rechazado

Una relación de exclusión debe ser una relación bidireccional pues ambos términos no pueden coexistir en un descriptor. Como es una relación dirigida bidireccional, es equivalente a una relación no dirigida.

- ☞ *la exclusión de un término implica la exclusión de sus hijos.*
- ☞ *un término no puede excluir a uno de sus ancestros.*

2.3 Tipos de términos

El esquema de facetas cumple un objetivo doble, por un lado es un esquema de clasificación y por otro es una manera de organizar el conocimiento por dominios. La estructuración del conocimiento se logra identificando características para cada faceta. El primer requerimiento que podemos identificar para esta organización es que las características se puedan agrupar dentro de cada faceta en conjuntos de propiedades relacionadas.

En el contexto de la organización del conocimiento, la relación más importante entre términos es la *generalización-de* que cumple el papel de agrupar términos. La relevancia de las otras relaciones debe evaluarse en el contexto de la búsqueda y clasificación.

En esta organización del conocimiento del dominio necesitamos básicamente dos tipos de términos:

- ***noción***: su función principal es organizar características desde una perspectiva particular del dominio, es decir, una noción es un término que debe tener hijos. Denota un grupo de características, pero por sí misma no es una característica (p.e., criterios de uso, tipo de estructura). Una noción no se puede utilizar en la construcción de un descriptor.
- ***propiedad***: denota una característica específica que se puede utilizar en la construcción de un descriptor (p.e., acotada, ordenada). Una propiedad puede o no tener hijos.

Note que los padres o hijos de una noción o propiedad pueden ser una noción o propiedad, no hay restricción al respecto. En este contexto, es útil definir otro tipo de término que no es ortogonal a los anteriores:

- ***clasificador***: un término que agrupa a otros términos, puede ser una noción o una propiedad. Una propiedad interesante de los términos clasificadores es si los términos que agrupa son o no ortogonales entre sí:
  - ***clasificador ortogonal***: se debe escoger uno y sólo uno de sus términos agrupados para un mismo descriptor.
  - ***clasificador no ortogonal***: se puede escoger más de uno de sus términos clasificados en un descriptor. Puede existir relaciones de exclusión entre algunos de sus términos.

Finalmente, encontramos dos grupos de términos que diferencian a todos los términos en una faceta según el rol que tiene en la descripción de un componente:

- ***abstracción***: identifica abstracciones medulares o principales de una faceta. Cada faceta tiene un único hijo de nombre ***abstracción*** que agrupa a todos los términos abstracción de la faceta. Todos los términos hijos de un término abstracción también son de dicho tipo. Los valores para estos términos son en su mayoría sustantivos y verbos.
- ***delimitador***: es un término que existe para delimitar el ámbito un término abstracción (***término delimitado***). Los valores que puede adoptar un delimitador son propiedades que particularizan el significado de otro término y por lo general, son sustantivos y adjetivos. Cada faceta tiene un único hijo de nombre ***delimitador*** que agrupa a todos los delimitadores de la faceta. Los delimitadores presentan las siguientes propiedades:
  - un delimitador se puede aplicar a uno o más términos.
  - en un descriptor de clasificación un delimitador debe aplicar a un término abstracción.
  - en un descriptor de búsqueda un delimitador puede existir sin aplicarlo a un término abstracción, es decir, se puede buscar una característica sin saber a que abstracción delimita.
  - todos los hijos de un delimitador son delimitadores.
  - un delimitador no es necesariamente un término propiedad.
  - por lo general, los delimitadores se agrupan en nociones cuyo nombre puede comenzar con "Tipo de ...".

Resumiendo, en todo momento un término es del tipo ((***abstracción***, ***delimitador***), (***noción***, ***propiedad***), (***clasificador*** (***ortogonal***, ***no ortogonal***), ***no clasificador***)). De esta forma, tenemos tres grupos complementarios que caracterizan a un término en una faceta. Un término ***noción*** implica que el término es un ***clasificador***.

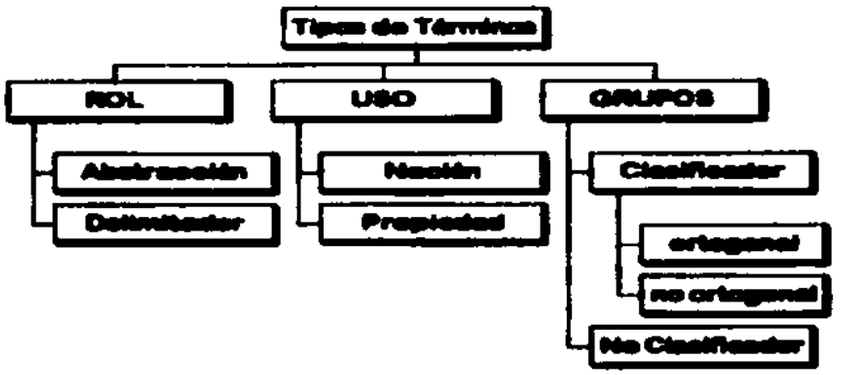


Fig. 3-6 Tipos de Términos

Finalmente, podríamos tener un tipo de término ***usuario*** para permitir que los usuarios puedan agregar términos al esquema y establecer relaciones con otros términos. Un término ***usuario*** no se puede utilizar en un descriptor, es una manera de que los usuarios soliciten la inclusión de un término especificando todas sus relaciones directamente sobre el esquema. Posteriormente, los administradores del esquema de clasificación evalúan estos términos y los incluyen o rechazan como términos utilizables.

2.4 Clasificación y búsqueda de componentes

2.4.1 Descriptores

La clasificación y búsqueda de componentes se realiza de manera uniforme, esto es, se construye un descriptor que contiene términos ***propiedad*** que describen las características del componente. De esta forma tenemos dos tipos de descriptores:

- ***descriptor de clasificación***: está asociado permanentemente a un componente en el depósito. Debe ser un ***descriptor completo***, es decir, debe tener términos ***abstracción*** en todas las facetas, y cada término ***delimitador*** debe aplicarse a un término ***abstracción*** en el descriptor.
- ***descriptor de búsqueda***: en el descriptor se colocan los términos ***propiedad*** que denotan las características que se desean de un componente. El descriptor no necesita tener términos en todas las facetas y puede especificar ***delimitadores*** sin aplicarlos a una ***abstracción***.

Los descriptores tienen las siguientes restricciones:

- sólo se puede incluir un término ***abstracción*** para la faceta abstracción, es decir, una clase representa a una única abstracción en el dominio.
- no se puede incluir un término que por sí mismo o mediante alguno de sus implicados excluya a alguno de los términos existentes en el descriptor.

- se puede incluir un término que por sí mismo o mediante alguno de sus implicados implique a alguno de los términos existentes en el descriptor.

Un caso importante en la organización del dominio es que algunas abstracciones presentan propiedades fijas que dependen del dominio y no de una implementación particular. Es decir a un término *abstracción* en la faceta abstracción se le puede asociar propiedades en la misma y/o en otras facetas.

Una manera de representar esta situación mediante el esquema propuesto es definir para el término *abstracción*, relaciones *implica-a* con todos los términos que representan sus propiedades. Otra manera de hacerlo, a nuestro parecer más coherente, es poder clasificar las abstracciones principales del dominio como si fueran componentes. Para lograr esto definiremos un tercer tipo de descriptor:

- ***descriptor de dominio***: es un descriptor de clasificación que se asocia a un término *abstracción* de la faceta abstracción.

Cuando un usuario construye un descriptor de búsqueda o clasificación y selecciona un término *abstracción* que tiene asociado un descriptor de dominio, se deben agregar sus términos al descriptor en construcción.

En teoría, cuando se utiliza un descriptor de dominio no debería poder excluirse sus términos, sin embargo, en la práctica eso depende de la calidad del espacio de términos definidos para el dominio y del descriptor asignado a la abstracción. En consecuencia, el usuario debe poder modificar los términos agregados a su descriptor.

2.4.2 Clasificación y construcción de descriptores

El proceso de clasificación se circunscribe a la construcción de un descriptor de clasificación para cada componente. La construcción de un descriptor consiste en identificar los términos que caracterizan a un componente en cada faceta. Es un proceso de selección en el que el usuario no puede especificar características que no pertenezcan al esquema. Existen tres procesos importantes en la construcción de un descriptor: ***normalización***, ***extensión*** y ***restricción***.

1. Normalización de sinónimos

Los términos pueden tener sinónimos y el usuario puede utilizarlos para construir un descriptor. Los sinónimos cumplen una función de ampliación de vocabulario orientada principalmente a los usuarios que no conocen el esquema de clasificación, el dominio o los componentes del depósito para el dominio.

En general, el proceso de clasificación se centraliza en un grupo responsable del dominio y/o del depósito que conoce el esquema de clasificación. Por esta razón, no es frecuente que se utilicen sinónimos para construir descriptores de clasificación. En cualquier caso, los sinónimos de un descriptor siempre se reemplazan por los términos principales. A este proceso se le llama ***normalización***.

ii. Extensión de implicados

Si un descriptor contiene términos que implican a otros, una estrategia adecuada para mejorar el proceso de búsqueda es incluir todos los implicados en el descriptor. De esta forma, no hay que buscar en la red de relaciones los términos que caracterizan al componente. Desde esta perspectiva, cuando un usuario selecciona un término para un descriptor, automáticamente los términos implicados se agregan al descriptor. A esta proceso se le llama ***extensión***.

No es muy claro si sería conveniente incluir en este proceso de extensión a los términos parecidos y los conectados por relaciones de generalización. Aparentemente no, porque de quererlos el usuario los podría haber especificado. En cualquier caso, representan relaciones y niveles de implicación menores.

iii. Restricción de excluidos

Cuando un usuario construye un descriptor, la herramienta automatizada no debe permitir que el usuario seleccione un término que ha sido excluído por otro. De esta forma, el rol de la relación de exclusión es ***restringir*** la inclusión de términos en el descriptor y el rol de la relación de implicación es incrementarla.

2.4.3 Búsqueda

Exploremos dos casos básicos de búsqueda:

1. el descriptor de búsqueda es un subconjunto de uno o más descriptores de clasificación.
2. algunos términos del descriptor de búsqueda conforman un subconjunto de uno o más descriptores de clasificación.

En el primer caso encontramos uno o más componentes que satisfacen todos los requerimientos buscados. Es el caso ideal, pero probablemente no sea el más frecuente. Todos los componentes encontrados pueden servir por igual, por lo que no importa el orden en que se muestren. Aún así, puede ser útil mostrarlos en un orden predefinido, por ejemplo agruparlos por colección o alguna otra característica de contexto (p.e., lenguaje, sistema operativo).

El segundo caso es el más interesante y donde se encuentran mayores posibilidades de encontrar componentes parecidos y no sólo exactos al buscado. Distinguimos dos tipos de parecido:

- ***parecido por intersección (PI)***: los componentes encontrados tienen un subconjunto de las características (términos) buscadas.
- ***parecido por cercanía (PC)***: los componentes encontrados tienen características similares a las buscadas.

Para cualquiera de los tipos de parecido, se requiere una medida de cuánto se parecen los componentes encontrados al buscado, para determinar cuáles componentes de los que están en el depósito son los que mejor satisfacen los requerimientos demandados.

Para ello necesitamos un ***criterio de comparación*** que proporcione una ***medida de parecido***. Primero exploraremos el parecido por intersección.

Antes de proponer un criterio de comparación, definiremos un estado que asigna el usuario y que denota la importancia del término en el descriptor. Lo hemos llamado ***estado de relevancia*** y puede adoptar los valores:

- ***imprescindible***: los componentes encontrados deben tener el término.
- ***prescindible***: los componentes encontrados podrían no tener el término.

Ciertamente, todas las características son importantes o no se habrían especificado para la búsqueda. Sin embargo, si el usuario quiere flexibilidad en la consulta, debe asignar el estado de imprescindible sólo a aquellos términos que son fundamentales para la búsqueda. Si todos los términos son imprescindibles, se desea una búsqueda exacta.

1. Medidas de peso

Obtener una medida implica ponderar el parecido entre componentes. Para hacer esto se requiere asignar ***medidas de peso*** que permitan calcular una ***medida de parecido***. Por simplicidad, restringiremos las medidas de peso a la estructura misma del esquema de clasificación y no a términos particulares de un dominio. Además, la medida de parecido debe adoptar valores entre [0, 1], para representar de manera directa el porcentaje de parecido.

En el esquema de clasificación propuesto encontramos dos conjuntos importantes que se pueden cuantificar independientemente del dominio:

- ***facetas***: existen facetas que son más importantes que otras. En particular, nos interesa cuantificar la importancia de las facetas de dominio. Asumimos que los términos de colección son imprescindibles. Las facetas de dominio por orden de importancia son: ***abstracción, servicios, objetos***.
- ***términos***: los términos por orden de importancia se pueden clasificar en: ***abstracción, delimitador***.

A la medida de peso la hemos llamado ***relevancia*** por ser más ilustrativo. Los valores que puede adoptar una relevancia deben estar entre [1, 100] y representan porcentajes. Dentro de cada conjunto la suma de las relevancias debe ser igual a 100. Las medidas de prueba iniciales son: $Rf = (Rfa=50, Rfs=30, Rfo=20)$ para las facetas (***abstracción, servicios, objetos***) y $Rt = (Rta=70, Rtd=30)$ para los términos (***abstracción, delimitador***). En el primer caso, se trata de ***relevancias de facetas (Rf)*** y en el segundo de ***relevancias de términos (Rt)***.

Las medidas de relevancia se aplican para ponderar la importancia de los términos encontrados en relación a los términos buscados. Son parámetros de entonación para la búsqueda y pueden irse modificando con el uso del esquema. En principio, estas medidas pueden ser las mismas para todos los dominios clasificados con el esquema, pero podría quererse una entonación particular por dominio. Esto puede suceder porque la importancia de las facetas podría variar entre dominios, p.e., en dominios matemáticos la faceta de servicios podría ser tan o más importante que la faceta abstracción.

ii. Criterios de comparación por intersección

Un criterio de comparación por intersección determina una medida de parecido por intersección (*PI*) para los componentes encontrados cuando satisfacen un subconjunto de los requerimientos buscados. Proponemos tres criterios de comparación iniciales que pueden utilizarse a elección del usuario en cada búsqueda para definir cuáles son los componentes que más se parecen al buscado.

- **Criterio 1: sin ponderación**

Es un criterio básico de comparación que determina el parecido por el número de términos encontrados respecto al número de términos buscados. Este criterio no pondera la importancia de los términos. De esta forma, tenemos:

$$PI = (\#\ términos\ encontrados) / (\#\ términos\ buscados)$$

- **Criterio 2: ponderación por facetas**

Se basa en el hecho de que cada faceta denota un nivel de importancia diferente en el esquema de clasificación. Se utiliza la relevancia de facetas (*Rf*) para ponderar los términos encontrados y buscados respecto a su importancia en la faceta. La medida de parecido se calcula como:

$$PI = (\sum Ei \times Rfi)/(\sum Bi \times Rfi)$$

Ei – # términos encontrados en la faceta *i*
Bi – # términos buscados en la faceta *i*
Rfi – relevancia de la faceta *i*

- **Criterio 3: ponderación por facetas y términos**

Además de diferenciar la importancia de las facetas, utiliza la relevancia de términos (*Rt*) para ponderar los términos encontrados. La medida de parecido se calcula como:

$$PI = \frac{\sum((AEi \times Rta) + (DEi \times Rtd)) \times Rfi}{\sum((ABi \times Rta) + (DBi \times Rtd)) \times Rfi}$$

AEi – # términos *abstracción* encontrados en la faceta *i*
DEi – # términos *delimitador* encontrados en la faceta *i*
ABi – # términos *abstracción* buscados en la faceta *i*
DBi – # términos *delimitador* buscados en la faceta *i*
Rta – relevancia de los términos *abstracción*
Rtd – relevancia de los términos *delimitador*
Rfi – relevancia de la faceta *i*

- **Criterio único**

Realmente los criterios *1* y *2* son casos particulares del criterio *3*. El criterio *2* resulta de tener relevancias de términos iguales (*Rfa = Rfd*). Por otro lado, el criterio *1* resulta de tener relevancias de facetas iguales (*Rfi*) para el criterio *2*.

De esta manera, tenemos un único criterio para medir el parecido entre componentes utilizando la comparación por intersección. En este criterio subyace una noción de peso para cada término que queda determinado por el esquema de clasificación, es decir, es independiente del descriptor de búsqueda o clasificación. El peso de un término es:

$$Peso\ término = Relevancia\ término \times Relevancia\ faceta$$

Podemos reescribir el cálculo del parecido por intersección como la relación entre los pesos de los términos encontrados respecto a los pesos de los términos buscados:

$$PI = \frac{\sum_{i=1}^{m} Rte_i \times Rfe_i}{\sum_{i=1}^{n} Rtb_i \times Rfb_i}$$

m = # términos encontrados
n = # términos buscados
Rte_i = relevancia del término encontrado
Rfe_i = relevancia de la faceta del término encontrado
Rtb_i = relevancia del término buscado
Rfe_i = relevancia de la faceta del término buscado

iii. Criterios de comparación por cercanía

- **Medida de cercanía**

Comencemos por definir una medida de cercanía (*Ce*) entre el término encontrado y el término buscado. El objetivo es determinar para un descriptor, cuán cercano está a un término que se está buscando. Esta medida sólo tiene sentido cuando no encontramos el término buscado. Disponemos básicamente de dos elementos:

- *relación parecido-a*: establece que un término se parece a otro. El parecido puede ser *alto* o *medio*.
- *relación generalización-de*: un padre se parece mucho al hijo porque el hijo hereda todas las características de su ancestro. Un hijo tiene un parecido menor con el padre porque el padre no contiene toda la información del hijo.

Para cada término, podemos tener cuatro tipos de términos cercanos y a cada uno se le puede asignar una relevancia (*Rc*) que pondera la cercanía con el término buscado:

• término descendiente	$Rcd = 100$	
• término ancestro	$Rca = 70$	$(Rca \leq Rcd)$
• término parecido alto	$Rcpa = 90$	$(Rcpa \leq Rcd)$
• término parecido medio	$Rcpm = 60$	$(Rcpm \leq Rcpa)$

Los valores de la relevancia de cercanía pueden variar, pues forman parte de la entonación del sistema de búsqueda. La medida de relevancia determina el orden de búsqueda del término cercano, se toma el primer término encontrado.

Una vez que encontramos un término cercano, podemos determinar cuán cercano es al término buscado. La medida de cercanía la podemos calcular utilizando la proporción del peso de ambos términos (***Rte* x *Rfe* / *Rtb* x *Rfb***) y la medida de relevancia (***Rce***) del término encontrado para ponderar esta proporción. Note que sólo se pueden definir relaciones entre términos de una misma faceta (***Rfe* = *Rfb***) y que las relaciones de clasificación y de parecido se establecen en un grupo de términos abstracción o en un grupo de términos delimitadores (***Rte* = *Rtb***). Entonces nos queda:

$$Ce = \left(\frac{Rce}{100}\right)^{n}$$

Rce = relevancia de cercanía del término encontrado
n = # de conexiones entre el término buscado y el encontrado para las relaciones de padre e hijo. El ancestro o descendiente inmediato puede ser una *noción* y hay que buscar un término *propiedad*

Esta medida de cercanía es menor o igual que la unidad y aplica individualmente a cada término no encontrado. El paso siguiente es definir una medida de parecido por cercanía entre dos componentes (descriptores). La definiremos de la misma manera que el parecido por intersección agregando la medida de cercanía.

$$PC = \frac{\sum_{i=1}^{m} Rte_i \times Rfe_i \times Cei}{\sum_{i=1}^{n} Rtb_i \times Rfb_i}$$

m = # términos cercanos encontrados
n = # términos buscados

Tenemos dos medidas de parecido que podemos unificar para obtener una medida de parecido total (***PT* = *PI* + *PC***). Esta medida es menor o igual que 1, por lo que directamente representa el porcentaje de parecido entre dos componentes (descriptores). Note que podemos reescribir el cálculo del parecido total como:

$$PT = \frac{\sum_{i=1}^{m} Rte_i \times Rfe_i \times Cei}{\sum_{i=1}^{n} Rtb_i \times Rfb_i}$$

m = # términos encontrados
n = # términos buscados
Ce = 1 cuando se encontró un término igual al buscado

- **Aplicación de la medidas de intersección y de cercanía**

La búsqueda de términos cercanos sólo tiene sentido cuando no se encuentra el término buscado. Es una operación costosa para cualquier sistema de búsqueda, pues debe explorar las relaciones del término no encontrado en uno o más niveles. En consecuencia, no parece razonable aplicar la medida de cercanía para cada uno de los descriptores que existen en el depósito.

La alternativa es delimitar el conjunto de descriptores sobre los que se aplicará la medida de cercanía. Un enfoque posible es hacer primero una búsqueda por intersección y luego calcular el parecido por cercanía para cada componente. Bajo esta perspectiva, se hace un doble proceso de búsqueda, primero se determina cuáles son los componentes que satisfacen exactamente algunas o todas las características que se están buscando y luego se calcula las medidas de parecido por intersección y por cercanía para cada uno de los componentes encontrados.

Aún cuando se puede obtener una medida de parecido total, es conveniente observar por separado cada una de las medidas, pues tienen significados totalmente diferentes. El parecido por intersección determina en que proporción los componentes son exactamente iguales, es decir, es el porcentaje de características que podemos utilizar sin modificar el componente encontrado. El parecido por cercanía determina que porcentaje de características podemos utilizar con poco esfuerzo de modificación sobre el componente encontrado.

- **Medida de reuso**

Finalmente, consideraremos una medida adicional que proporciona información importante para la selección de uno u otro componente:

- ***medida de reuso (MR)***: es la proporción de las características encontradas respecto al total de características de cada componente.

Es una medida de reuso individual e indica la cantidad de características del componente que se utilizarán y las que no que se utilizarán, en relación a las características buscadas. Una medida de reuso baja probablemente indique que hay que construir un contexto de integración significativo respecto a lo que se reutilizará. Una medida alta de reuso indica que se utilizará todo el componente por lo que no es significativo el esfuerzo de la construcción del contexto de integración.

iv. Proceso de búsqueda

Resumiendo, el proceso de búsqueda de componentes involucra los siguientes procesos:

(1) ***Normalizar sinónimos***: reemplazar todos los sinónimos por sus términos.
(2) ***Extender implicados***: agregar al descriptor todos los términos implicados directamente, es decir, limitar la extensión a una conexión de implicación.
(3) ***Buscar por intersección***: buscar todos los descriptores que tengan términos iguales a los buscados. Los descriptores deben tener todos los términos ***imprescindibles*** buscados.

(4) ***Calcular medidas***: para cada componente recuperado, calcular las medidas de parecido por intersección (*PI*), de parecido por cercanía (*PC*) y de reuso (*MR*).
(5) ***Mostrar componentes***: mostrar los componentes recuperados ordenados por alguna de las medidas calculadas.
(6) ***Recuperar***: la herramienta debe recuperar la información completa de cada uno de los componentes que el usuario desee examinar y proporcionar todos los elementos que se requieran para su uso si decide utilizarlo.

Algunos parámetros que puede utilizar el usuario para configurar la búsqueda son:

- *relevancias*: relevancias de facetas (*Rf = {Rfa=50%, Rfs=30%, Rfo=20%}*) y relevancias de términos (*Rt = {Rta=70%, Rtd=30%}*).
- *mínima intersección (MI)*: medida que especifica el porcentaje mínimo de términos que deben coincidir exactamente para considerar que un descriptor podría satisfacer los requerimientos (*MI=30%*).
- *mínimo parecido por intersección(MPI)*: medida mínima del parecido por intersección (*MPI=40%*).
- *mínimo parecido total (MPT)*: medida mínima del parecido total (*MPT=50%*).
- *máximo número de componentes (MNC)*: que se quieren obtener para la consulta (*MNC=20*).

2.5 Definición de un prototipo

2.5.1 Elementos básicos

- Un ***dominio*** tiene un único ***esquema*** de facetas y cada esquema corresponde a un único dominio.

- Un ***esquema*** tiene ***N facetas***. En la propuesta actual hay cuatro facetas propuestas, pero podría evolucionar a un mayor número de facetas (p.e., para clasificar patrones).
  - La faceta ***contexto*** se puede aplicar a todos los dominios. La faceta contexto se debe crear con las siguientes nociones: *colección, lenguaje, ambiente, plataforma*.
  - Cuando se crea un esquema se deben crear todas sus facetas: *abstracción, servicios, objetos*. También debe incluirse:
    - Crear las nociones iniciales para cada faceta
      - *"Criterios de uso"* para la faceta *abstracción*
      - *"Complejidad"* para la faceta *servicios*
      - *"Requiere"* y *"Efecto"* para la faceta *objetos*
    - Indicar en la faceta *servicios* si una operación es transfomadora u observadora. Sólo aplica cuando el término es *propiedad*.

- Una ***faceta*** tiene ***N términos***. Cada término debe tener la siguiente información:

Información de herencia:

- ***Padre***: indica cuál es el padre (limita a un único padre)
- ***Hijos***: indica el número de hijos
- ***Nivel***: indica el nivel en que se encuentra en la jerarquía. Cuando se crea un hijo para un término, el nivel del hijo es del padre más 1. Cuando no tiene término padre el nivel es 1.
- ***Raíz***: indica el ancestro (abstracción o delimitador) que no tiene padre

Tipo de término:

- ***abstracción / delimitador*** es un atributo. Sólo se puede especificar cuando no tiene un término padre, cuando tiene un padre hereda el atributo.
  - para un término delimitador debe indicarse los términos abstracción que puede delimitar
- ***noción / propiedad*** es un atributo que siempre se puede asignar y modificar. Una noción se crea para agrupar términos, por lo que es raro que no tenga hijos.
- ***clasificador / no clasificador*** no es un atributo, pues queda determinado si tiene o no hijos. Cada vez que se crea un hijo para un término se incrementa la información de sus ***Hijos***.
- ***ortogonal / no ortogonal*** es un atributo que se puede asignar y modificar, aplica sólo cuando tiene hijos.
- ***usuario*** es un atributo que indica que es un término propuesto por un usuario para su inclusión en el esquema.

- Un ***término*** puede tener ***N sinónimos***. Un sinónimo puede tratarse como un término de un tipo especial, indicando de qué término es sinónimo.

2.5.2 Relaciones

- Puede definirse cuatro tipos de relación entre dos términos: ***implica-a, excluye-a, parecido-a, generalización-de*** (ésta última se puede almacenar con el mismo término).
  - debe indicarse si la relación es o no dirigida
  - en la relación de parecido se maneja dos niveles de parecido: alto, medio

- Restricciones para definir una relación:
  - debe establecerse entre dos términos diferentes. Puede manejarse un término origen y un término destino aún cuando la relación no sea dirigida
  - los términos deben pertenecer a una misma faceta
  - no debe existir una relación entre los dos términos, pues sólo puede existir una relación entre dos términos

- Definición de una relación ***Hijo-de*** (a un término se le especifica un padre)

Restricciones:

- el hijo no puede ser un ancestro del padre. No puede ser descendiente del padre ni de nadie porque sólo puede tener un padre.
- el hijo no puede excluir a un ancestro del padre, porque se excluiría a sí mismo

Implicaciones:

- el padre incrementa en 1 su número de hijos
- la raíz del hijo es la misma raíz del padre
- el nivel del hijo es el nivel del padre incrementado en 1
- el rol del hijo es el mismo rol del padre (*abstracción / delimitador*)
- en caso que el hijo tenga hijos, hay que actualizar el nivel, la raíz y el rol de todos los hijos

- Definición de una relación ***Implica-a***

Restricciones:

- el destino (implicado) no puede ser un ancestro del origen, porque un término implica a todos sus ancestros
- el destino no puede ser un descendiente del origen, porque la semántica del descendiente es que no queda determinado por la existencia del padre
- el destino ni alguno de sus implicados puede excluir al origen o a alguno de sus ancestros, porque estaría excluyendo al que lo está implicando
- ningún ancestro del destino puede estar excluido por el origen o alguno de sus ancestros, porque un término implica a todos sus padres y la exclusión de un término excluye a todos sus hijos

- Definición de una relación ***Excluye-a***

Restricciones:

- el destino (excluido) no puede ser un ancestro del origen, porque el origen implica a todos sus ancestros
- el destino no puede ser un descendiente el origen, porque un hijo implica a todos sus ancestros y se estaría excluyendo a sí mismo
- el destino o alguno de sus descendientes no puede estar implicado por el origen o alguno de sus ancestros

- Definición de una relación ***Parecido-a***

Restricciones:

- el destino no puede ser un ancestro o un descendiente del origen, porque existe una relación de parecido natural en la relación de herencia
- el destino no puede estar implicado por algún ancestro del origen, porque la relación de implicación es más fuerte que la de parecido

- el destino o uno de sus ancestros no puede estar excluido por el origen o uno de sus ancestros

2.5.3 Otros elementos

- Almacenamiento de los descriptores de clasificación
- Almacenamiento y enlace con los componentes reutilizables
- Estructura de documentación y empaque de todas las partes de los componentes
- Almacenamiento de los descriptores de búsqueda y estadísticas de éxito
- Características más avanzadas del funcionamiento de una biblioteca pueden encontrarse en el apartado "Bibliotecas de componentes reutilizables" (cap.1, sección 5, pag. 30-39).

Página intencionalmente en blanco

Conclusiones y resultados

Manifestamos de antemano nuestra animadversión al estilo enumerado de la presentación de las conclusiones de un trabajo. La razón es que se dificulta proporcionar una visión global, tiende a aislar resultados y lo que es peor, deja poco trabajo al lector. Esta investigación se estructuró en tres capítulos y cada una ha arrojado conclusiones propias, por lo que presentaremos las conclusiones en tres secciones.

1.1 Contexto

El inicio de este trabajo se centró en la estructuración del conocimiento acerca del reuso de software. Definimos una gran cantidad de conceptos y estructuramos múltiples categorías para organizarlos dentro del contexto del proceso global de desarrollo de software. La organización del conocimiento de este dominio aporta tres elementos fundamentales:

- ***definición de una estructura*** que permite delimitar diferentes aspectos del reuso, clasificar y evaluar propuestas.
- ***definición de un contexto global*** que permite visualizar las implicaciones y relaciones de elementos individuales de reuso.
- ***dimensionamiento*** de los alcances e implicaciones del reuso de software.

A nuestro entender, la extensión y ramificaciones de este dominio son enormes, a tal punto, que en la literatura casi cualquier aspecto de ingeniería de software se ha relacionado con el reuso. Esto dificulta lograr una visión global, pues el espectro es muy amplio. Note que la jerarquía propuesta no está acabada y puede aceptar numerosas ramificaciones e inclusive reorganizaciones.

Consideramos que esta organización del dominio del reuso es un aporte significativo porque el estado de estructuración de este conocimiento aún no es estable y está diseminado en una gran cantidad de artículos. Esta situación se ve plasmada en una carencia de libros acerca del tema y la mayoría son recopilaciones[1]. La cantidad de información disponible es considerable, pero el hecho de no tener libros de texto denota un estado de madurez incipiente. Esto por supuesto, representa un obstáculo significativo para la divulgación de este conocimiento a nivel académico y comercial.

Una consecuencia lateral, pero importante, de lo anterior fue el redimensionamiento de los alcances de este trabajo. Las motivaciones iniciales de esta investigación tuvieron como polo de atracción la relevancia de sus implicaciones técnicas. La delimitación de las dimensiones del reuso ubicaron a este trabajo en los niveles inferiores de la jerarquía (bibliotecas, componentes, esquemas de clasificación, lenguajes OO).

---

[1] Estuvimos esperando la aparición del libro "*Software Reuse*" de Ivar Jacobson antes de escribir estas conclusiones, pero no llegó (estaba anunciado para los primeros días de marzo '97).

Aunque parezca sólo un ejercicio de ubicación, dejó claro que el tema de trabajo aborda una parte de la infraestructura de soporte para niveles superiores de reuso. Es importante resaltar, que la investigación de infraestructura lleva una década de recorrido tratando este tipo de problemas y no ha logrado uniformizar sus resultados. Lo que intentamos decir con esto es que, aunque dispersos, ya existen los elementos necesarios para definir la infraestructura de soporte y que debemos enfocarnos a los niveles superiores pues éstos reportarán mayores beneficios. De ninguna manera estamos disminuyendo el valor de éste ni otros trabajos por abocarse a los niveles inferiores, sólo creemos que es tiempo de elevar el centro del análisis en los trabajos futuros.

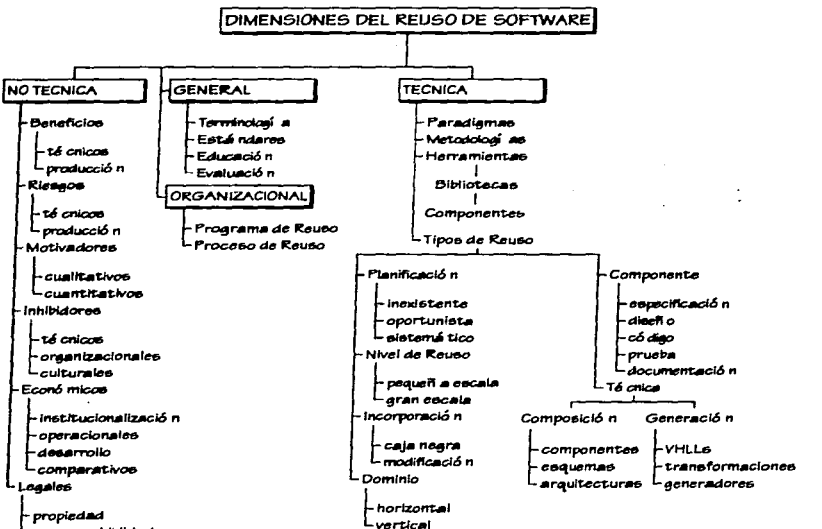


Fig. 4-1 Dimensiones del Reuso de Software

Finalmente, observamos una tendencia curiosa en los avances que se han ido generando: las prácticas y metodologías están orientadas a organizaciones con una infraestructura muy grande de desarrollo de software (múltiples equipos de desarrollo, grupos que monitorean y evalúan el proceso, grandes depósitos de componentes, costosas y complejas herramientas y ambientes, etc.) . Esta característica no es común a la mayoría de las organizaciones en México, por lo que una tarea importante a realizar es adaptar estos elementos para aplicarlos con éxito en organizaciones de menor tamaño. Los beneficios del reuso no están limitados a un tamaño de la organización, su mayor dependencia gira en torno al proceso de desarrollo.

1.2 El modelo de objetos y el reuso de software

El enfoque de análisis del paradigma orientado a objetos se centró en el estudio de sus mecanismos de programación y sus aportes al reuso de software en el contexto de la interconexión de componentes. El logro fundamental de este estudio fue mostrar que el modelo de objetos proporciona mecanismos adicionales que incrementan las posibilidades de composición y que en principio, incrementan las posibilidades de reuso. Sin embargo, existe una acotación significativa:

- el hecho de utilizar el paradigma OO no implica que se obtendrán automáticamente los beneficios del reuso de componentes. Hace falta un conocimiento sólido de la manera como los mecanismos del modelo de composición OO contribuyen a este objetivo y de cuándo y cómo utilizarlos.

Otro resultado importante lo obtuvimos al dimensionar este análisis respecto a la visión general obtenida en el primer capítulo. No es difícil percatarse de que la incidencia de la elección de uno u otro paradigma de programación es marginal en relación al resto de los elementos, sobre todo a los de la dimensión organizacional. Si bien este estudio análisis está limitado a los componentes de menor nivel (código), nos atrevemos a generalizar el siguiente resultado: ***el uso del modelo de objetos no trae consigo los beneficios del reuso.*** Es una aseveración fuerte sobre todo cuando es un lugar común escuchar que el reuso y el modelo de objetos son hermanos siameses, por lo que aclararemos nuestra posición.

Estamos convencidos que existe una "afinidad" entre el modelo de objetos y el reuso de software manifestada como un conjunto de posibilidades, es decir, elementos que podrían ser adecuados para la implementación de una estrategia de reuso. Sin embargo, son cosas separadas e independientes y el trabajo futuro debe enfocarse a la integración de ambos elementos. Esta situación queda claramente plasmada en el hecho de que la mayoría de las metodologías de análisis y diseño OO no incluyen elementos específicos que unifiquen el reuso de software a todas las fases del proceso de desarrollo.

En este sentido, el factor *reuso de software* no es el mayor beneficio del modelo de objetos como paradigma de construcción. Su mayor atractivo (entre otros) recae en la elevación significativa del nivel de abstracción, la organización jerárquica de las abstracciones del dominio y una reducción en la distancia que existe entre el análisis, diseño e implementación.

Una de las propuestas de reuso más relevantes, independiente de un paradigma particular, es el *análisis de dominio*[2] y pensamos que se puede integrar con el modelo de objetos. El principal producto de este proceso es una arquitectura de software, es decir, una síntesis de las principales estructuras y decisiones de diseño de un sistema. La arquitectura de software está formada por componentes de diferentes granularidades que pueden implementarse consistentemente con frameworks, categorías de clases, patrones de diseño y clases. Esta integración requiere que la metodología de desarrollo incluya el proceso de análisis de dominio y una correspondencia entre los elementos de la arquitectura de software y los elementos del modelo de objetos. En particular, ya existen algunas propuestas de análisis de dominio OO.

De esta manera, tendríamos una metodología de desarrollo integral en la dimensión técnica del reuso (note que esta integración no aborda directamente las otras dimensiones). Por último, recalcamos nuevamente que los beneficios que se pueden obtener de una propuesta de reuso concreta dependen en poco de la selección de un paradigma y depende en mucho del programa global de reuso que se diseñe para cada organización en particular.

Fig. 4-2 Integración del Modelo OO con el Análisis de Dominio

1.3 Clasificación de componentes reutilizables OO

Para clasificar componentes reutilizables OO propusimos una adaptación del esquema de facetas basados en las propiedades de las clases y un sistema de búsqueda y clasificación adecuado a dicha estructura de clasificación. La importancia de esta adaptación radica en dos elementos:

- permite clasificar componentes OO de manera natural, pues existe una correspondencia entre las propiedades de las clases y la estructura del esquema de clasificación.
- permite organizar el conocimiento del dominio utilizando la descomposición en abstracciones del modelo de objetos. También es un medio de divulgación del conocimiento del dominio.

Adicionalmente, el esquema de facetas es uno de los elementos que se utilizan para controlar el vocabulario y organizar las características que resultan de un proceso de

[2] El análisis de dominio estudia clases de sistemas dentro de un dominio de aplicación para determinar los requerimientos globales del conjunto de sistemas y para construir una arquitectura de software general que oriente la construcción de cualquier aplicación dentro del dominio (*ver Anexo B*).

análisis de dominio. Por lo tanto, un esquema de facetas OO facilitaría la integración entre el modelo de objetos y la propuesta de reuso de análisis de dominio que fue esbozada anteriormente.

A nuestro parecer, el uso de un esquema de facetas es una alternativa a las especificaciones formales para representar la semántica de los componentes. Efectivamente, la caracterización (clasificación) de un componente con los términos del dominio al que pertenece es una especificación semántica, pues identifica las propiedades que lo definen. En este sentido, probablemente lo más importante es que permite hacer búsquedas semánticas mencionando las propiedades que se desean de un componente. Una diferencia notable con un esquema de clasificación jerárquico es que el esquema de facetas captura todas las propiedades del dominio y no sólo un subconjunto.

Adentrándonos en los detalles de la adaptación propuesta, los aportes más significativos son los siguientes:

- identificación de dos grupos principales y excluyentes entre sí de propiedades: abstracciones y delimitadores. Esta división de términos aplica uniformemente a todas las facetas.
- identificación de relaciones que permiten organizar las propiedades del dominio. La relación de mayor importancia es la clasificación o agrupamiento. Las relaciones de implicación y exclusión son útiles para representar relaciones de dependencia entre propiedades.
- identificación de medidas de parecido entre componentes que dependen de medidas que se asignan a la estructura de clasificación, no a las propiedades del dominio.

La relación de parecido se propuso inicialmente como una manera de representar similitudes entre componentes mediante el parecido de sus propiedades. Es una relación problemática porque es demasiado subjetiva y no representa una relación de dependencia significativa en la organización de las propiedades del dominio. Además, al momento de diseñar las estrategias de búsqueda surgió una medida de parecido que no depende de esta relación (parecido por intersección). Esta era la motivación de la relación de parecido, por lo que ya no se justifica su existencia[3], es mejor prescindir de esta relación en el esquema propuesto. Con ello se simplifica el esquema y su uso sin pérdida significativa de las posibilidades de búsqueda.

A pesar de las ventajas de esta adaptación, el esquema de clasificación tiene limitaciones importantes:

- no puede expresar dependencias de tiempo
- no puede representar precondiciones, poscondiciones e invariantes
- no aborda la clasificación de componentes de mayor nivel

[3] El número de relaciones de parecido que surgieron en el esquema de facetas para el dominio de estructuras de datos fue poco significativo en relación a la cantidad de términos.

Por último, un resultado lateral del proceso de investigación fue la construcción de un esquema de facetas para el dominio de estructuras de datos. Es un resultado importante porque una de las utilidades de un esquema de facetas es la divulgación del conocimiento del dominio. En consecuencia, podría utilizarse en la enseñanza, sobre todo para sintetizar las propiedades fundamentales de las estructuras de datos y discernir cuál es la importancia de cada una.

Referencias

[Atkinson 91] Atkinson, C. "Object-Oriented Reuse, Concurrency and Distribution". Addison-Wesley Publishing Company, Reading, Massachusetts, 1991.

[Batory 92] Batory, D., Singhal, V., Sirkin, M. "Implementing a Domain Model for Data Structures". International Journal of Software Engineering and Knowledge Engineering, Vol. 2, No. 3, 1992, pp. 375-402.

[Beck 94] Beck, K. "Patterns and Software Development". Dr. Dobb's Journal, February 1994.

[Biggerstaff+ 87] Biggerstaff, T., Richter, C. "Reusability Framework, Assessment, and Directions". IEEE Software, vol. 4, no. 2, March 1987, pp. 41-49.

[Biggerstaff+ 89a] Biggerstaff, T., Richter, C. (Eds.). "Software Reusability, Volume I: Concepts and Models". ACM Press Frontier Series. Addison-Wesley, Reading, Mass., 1989.

[Biggerstaff+ 89b] Biggerstaff, T., Richter, C. (Eds.). "Software Reusability, Volume II: Applications and Experience". ACM Press Frontier Series. Addison-Wesley, Reading, Mass., 1989.

[Booch 86] Booch, G. "Software Components with ADA: Structures, Tools, and Subsystems". Benjamin / Cummins, 1986. Second Edition.

[Booch 94] Booch, G. "Object-Oriented Analysis and Design with Applications", segunda edición. The Benjamin/Cummins Publishing Company, Inc., 1994.

[Caldiera+ 91a] Caldiera, G. and Basili, V. "Implementing Faceted Classification for Software Reuse". Computer, 24(2), February 1991.

[Caldiera+ 91b] Caldiera, G., Basili, V. "Identifying and Qualifying Reusable Software Components". IEEE Computer, vol. 24, no. 2, Feb. 1991, pp. 61-69.

[CARDS 93a] "Application Engineering with Domain-Specific Reuse: Instructor's Guide" (vol. I). Central Archive for Reusable Defense (CARDS), Marzo 1993.

[Chen+ 93] Chen, D., Huang, S. "Interface for reusable software components". Journal of Object Oriented Programming (JOOP), vol. 5, no. 8, jan. 1993, pp. 42-53.

[Coad+ 94] Coad, P., Mayfield, M. "Object Model Pattern: Workshop Report". Addendum to the Proceedings of OOPSLA'94.

[Creech+ 91] Creech, M., Freeze, D., Griss, M. "Using Hypertext in Selecting Reusable Software Components". In Proceedings of Hypertext '91, December, 1991.

[Deng-Jyi+ 93] Deng-Jyi, C., Shih-Kun, H. "Interface for Reusable Software Components". Journal of Object Oriented Programming (JOOP), vol. 5, no. 8, January 1993, pp. 42-53.

[Edwards 90] Edwards, S. "The 3C Model of Reusable Software Components", Third Annual Workshop: Methods and Tools for Reuse, Syracuse, 1990.

[Freeman 87] Freeman, P. (Ed.) "Tutorial: Software Reusability". IEEE Computer Society Press, Washington, D.C., 1987.

[Gamma+ 94] Gamma, E., Helm, R., Johnson, R., Vlissides, J. "Design Patterns: Elements of Reusable Object-Oriented Software". Addison-Wesley. Reading, Mass. 1994.

[GAO/IMTEC 93] "Issues Facing Software Reuse". GAO/IMTEC - 93 - 16.

[Greiff 93] Greiff, W. "El Ocaso del Sol a través de una Ventana: Un discurso sobre los paradigmas de programación". Soluciones Avanzadas, año 1, no. 5, Septiembre/Octubre 1993, pp. 35-40.

[Greiff 93b] Greiff, W. "Paradigma vs. Metodología: El caso de la Programación Orientada a Objetos (Parte I)". Soluciones Avanzadas, año 1, no. 6, Noviembre/Diciembre 1993, pp. 10-16.

[Helm 92] Helm, R. "Semantic Integrity of Reusable Objects: Re-use and Abuse of Software Components". En el panel de "Ensuring Semantic Integrity of Reusable Objects" en OOPSLA'92.

[Helm 95] Helm, R. "Patterns in practice". Proceedings of OOPSLA'95.

[Helm+ 91] Helm, R., Maarek, Y. "Integrating Information Retrieval and Domain Specific Approaches for Browsing and Retrieval in Object-Oriented Class Libraries". In Proc. of OOPSLA'91. ACM Sigplan notices, 1991. pp 47-61.

[Helm+ 95] Helm, R., Gamma, E. "Patterns and Software Design: Patterns for Reusable Object-Oriented Software". Dr. Dobb's Journal, "Windows Programming" (special issue), Mar/Apr 1995.

[Henderson+ 92] Henderson-Sellers, B., Freeman, C. "Cataloguing and Classification for Object Libraries". ACM SIGSOFT Software Engineering Notes, vol. 17, no. 1, Jan. 1992. pp. 62-64

[Hernández 91] Hernández, Armando. "Catálogo de Componentes Reusables de Software. Prototipo Algres". Tesis de Maestría, IIMAS-CCH-UNAM, México, 1991.

[Hong+ 95] Hong, S., Koetzer, B. "A Comparison of Software Reuse Support in Object-Oriented Methodologies". Proceedings of IRMA'95.

[Hooper+ 91] Hooper, J., Chester, R. "Software Reuse: Guidelines and Methods". Plenum Press, 1991.

[Hsian-Chou+ 93] Hsian-Chou, L., Feng-Jian, W. "Softare reuse based on a large objec-oriented library". ACM SIGSOFT Software Engineering Notes, vol. 18, no. 1, Jan. 1993. pp. 74-80

[Humphrey 92] Humphrey, S. "SIG/CR Calssification and Retrieval for Software Reuse". En Proceedings of ASIS'92.

[Johnson 88] Johnson, R., Foote, B. "Designing Reusable Classes". Journal of Object-Oriented Programming (JOOP), vol. 1, no. 2, 1988. pp. 22-35.

[Johnson+ 88] Johnson, R., Foote, B. "Designing Reusable Classes". Journal of Object-Oriented Programming (JOOP), vol. 1, no. 2, July/August 1988, pp. 22-35.

[Karlsson+ 92] Karlsson, E., Sindre, G., Sorumgard, S., Tryggseth, E. "Weighted term spaces for relaxed search". 1st International Conference on Information and Knowledge Management (CIKM'92), Baltimore, Nov. 5-8, 1992.

[Karlsson+ 93] Karlsson, E., Sorumgard, S., Tryggseth, E. "Classification of object-oriented components for reuse". In Proceedings of Technology of Object-Oriented Languages and Systems, TOOLS'93.

[Kennedy 92] Kennedy, B. "Design for Object-Oriented Reuse in the OATH Library". Journal of Object Oriented Programming, vol. 5, no. 4, pp. 51-57, July/August 1992.

[King 89] King, R. "My Cat is Object-Oriented". En Object-Oriented Concepts, Databases and Applications, Kim, W., Lochovsky, F. (Eds.), ACM Press and Addison-Wesley, 1989, pp. 23-30.

[Krueger 92] Krueger, C. "Software Reuse". ACM Computing Surveys, vol. 24, no. 2, June 1992.

[Lewis+ 91] Lewis, J., Henry, S., Kafura, D., Schulman, R. "An Empirical Study of the Object-Oriented Paradigm and Software Reuse". Proceedings of OOPSLA '91, special issue of SIGPLAN Notices, vol. 26, no. 11, November 1991, pp. 184 - 196.

[Lewis+ 92] Lewis, J., Henry, S., Kafura, D., Shulman, R. "On the Relationship Between the Object-Oriented Paradigm and Software Reuse: An Empirical Investigation". Journal of Object-Oriented Programing (JOOP), vol. 5, no.4, July/August 1992, pp. 35-41.

[Meyer 87] Meyer, Bertrand. "Reusability: the case for object-oriented design". En Software Reusability, Volume I: Concepts and Models, T. J. Biggerstaff, A. J. Perlis, Eds. ACM Press Frontier Series, Addison-Wesley, Reading, Mass., 1989.

[Meyer 88] Meyer, B. "Object-Oriented Software Construction". Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[Meyer 92] Meyer, B. "Eiffel: The Language". Prentice Hall, 1992. Fragmento del libro traducido en "Invitación a Eiffel: primera parte", Soluciones Avanzadas, año 1, no. 6, Noviembre/Diciembre

1993, pp. 37-40.

[Meyer 92b] Meyer, B. "Achieving semantic integrity through 'Design by Contract'". En el panel de "Ensuring Semantic Integrity of Reusable Objects" en OOPSLA'92.

[Meyer 94] Meyer, B. "Reusable software: The Base object-oriented component libraries". Prentice Hall Object-Oriented Series, 1994.

[Meyer 95] Meyer, B. "Object Success: A manager's guide to object orientation, its impact on the corporation, and its use for reengineering the software process". Prentice Hall Object-Oriented Series, 1995.

[NATO 1] "Standard for the Development of Reusable Software Components". NATO Communications and Information Systems Agency.

[NATO 2] "Standard for Software Reuse Procedures". NATO Communications and Information Systems Agency.

[Oktaba 90] Oktaba, H. "Componentes Reusables en Modula-2". Aportaciones Matemáticas, comunicaciones 8, 1990. pp. 103-115.

[Oktaba 93] Oktaba, H. "Programación Orientada a Objetos: ¿moda o realidad?". Soluciones Avanzadas, año 1, no. 3, Marzo/Abril 1993, pp. 39-42.

[Oktaba+ 93b] Oktaba, H., Quintanilla, G. "Abstracción de Datos en Lenguajes Orientados a Objetos". Soluciones Avanzadas, año 1, no. 5, Septiembre/Octubre 1993, pp. 25-31.

[Ostertag+ 92] Ostertag, E., Hendler, J., Prieto-Díaz, R., Braun, C. "Computing Similarity in a Reuse Library System: An AI-Based Approach". ACM Transactions on Software Engineering and Methodology, Vol. 1, No. 3, Julio 1992, page. 205-228.

[Poulin+ 93] Poulin, J., Caruso, J. "A Reuse Metrics and Return on Investment Model". En Proceedings of "Second International Workshop on Software Reusability: Advances in Software Reuse". IEEE Computer Society Press, 1993.

[Prieto-Díaz 85] Prieto-Díaz, R. "A software classification schema". PhD. thesis, Department of Information and Computer Science, University of California, Irvine, 1985.

[Prieto-Díaz 87] Prieto-Díaz, R., Freeman, P. "Classifying software for software reusability". IEEE Software, pp. 6-16, January 1987.

[Prieto-Díaz 89] Prieto-Díaz, R. "Classification of Reusable Modules". En "Software Reusability: Concepts and Models", vol. I, 1989, ACM Press, Addison-Wesley.

[Prieto-Díaz 91a] Prieto-Díaz, R. "Making software reuse work: an implementation model". ACM SIGSOFT Software Engineering Notes, vol.16, no.3, pp. 61-68, July 1991.

[Prieto-Díaz 91b] Prieto-Díaz, R. "Implementing Faceted Classification for Software Reuse". Communications of the ACM, 34(5), May 1991.

[Prieto-Díaz 91c] Prieto-Díaz, R. "Making Software Reuse Work: An Implementation Model". ACM SIGSOFT, Software Engineering Notes, 16(3), July 1991.

[Prieto-Díaz+ 93] Prieto-Díaz, R., Frakes, W. (Eds.). "Advances in Software Reuse: selected papers from the Second International Workshop on Software Reusability". IEEE Computer Society Press, Los Alamitos, California.

[Quintanilla+ 93] Quintanilla, G., Silva, S. "Elementos de la programación Orientada a Objetos". Soluciones Avanzadas, año 1, no. 4, Julio/Agosto 1993, pp. 5-7.

[Ranghanathan 67] Ranghanathan, S. R. "Prolegomena to Library Classification". Asia Publishing House, Bombay, India, 1967.

[REW 92] Proceedings for the Reuse Education Workshop. 23-24 September, West Virginia University, 1992

[REW 93] Proceedings of the Second Annual Reuse Education and Training Workshop, 25-27 October

1993, West Virginia University.

[Sanchez 93] Sanchez, S. "Adquisición del Conocimiento". Tesis de Grado, Maestría en Ciencias de la Computación, UACPyP del CHH, UNAM, 1993.

[Silva 94] Silva, S. "Polimorfismo en los Lenguajes de Programación". Soluciones Avanzadas, año 2, no. 8, Marzo/Abril 1994, pp. 47-49.

[Sindre 92] Sindre, G., Karlsson, E. "Heuristics for maintaining term structures for relaxed search". En Proceedings of the 3rd International Conference on Database and Expert Systems Applications, DEXA'92, Springer Verlag, 1992.

[Sitamaran 96] Sitamaran, M. (Ed.). "Fourth International Conference on Software Reuse (Proceedings)". IEEE Computer Society Press, 1996.

[Sorumgard+ 93] Sorumgard, L., Sindre, G., Stokke, F. "Experiences from Application of a Faceted Classification Scheme". Second International Workshop on Software Reusability, (REUSE'93), Lucca, Italy, March 24-26, 1993.

[Steigerwald+ 91] Steigerwald, R., Luqi, McDowell, J. "CASE tool for reusable software component storage and retrieval in rapid prototyping". Information and Software Technology, vol.33, no. 9, nov. 1991.

[Tsenzer+ 89] David Tsenzer, Murthy Ganti, and Sunil Podar. "Object-Oriented Software Reuse: The Yoyo Problem". Journal of Object-Oriented Programing, vol. 2, no.3, Sept./Oct. 1989.

[Tracz 88] Tracz, W. (Ed.). "Tutorial: Software Reuse: Emerging Technology". IEEE Computer Society Press, Los Alamitos Calif., 1988.

[Tracz 95] Tracz, W. "Confessions of a Used Program Salesman: Institutionalizing Software Reuse". Addison - Wesley, 1995.

[Weide+ 91] Weide, B., Ogden, W., Zweben, S. "Reusable Software Components". Advances in Computers, vol. 33, no. 1, 1991.

[WISR 91] Workshop on Institutionalizing Software Reuse, 1991.

[WISR 92] Workshop on Institutionalizing Software Reuse, 1992.

[WISR 93] Workshop on Institutionalizing Software Reuse, 1993.

[WISR 94] Workshop on Institutionalizing Software Reuse, 1994.

[WISR 96] Workshop on Institutionalizing Software Reuse, 1996.

ANEXO A:
Aplicación del esquema de clasificación de facetas OO para el dominio de estructuras de datos

Recordemos que para construir un esquema de clasificación de facetas se requiere realizar un análisis de dominio. Este análisis descubre las características y relaciones más importantes del dominio y las organiza en un esquema de clasificación. Esto permite identificar a cada elemento del dominio asociándole un conjunto de las características del esquema de clasificación.

Analizamos el dominio de las estructuras de datos y organizamos este conocimiento en el esquema de clasificación que estructuramos de la siguiente manera.

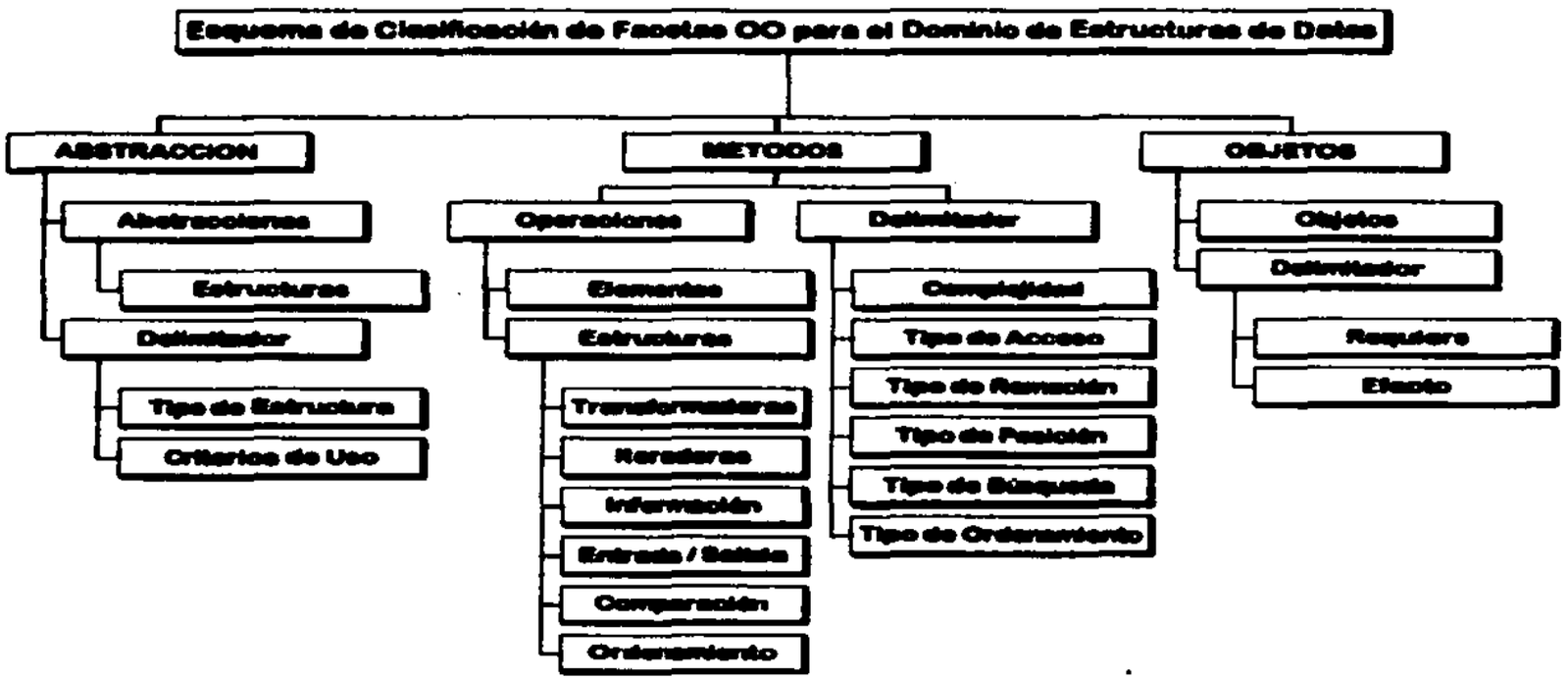


Fig. A-1 Esquema de clasificación de facetas OO para el dominio de Estructuras de Datos

1. Faceta: *Abstracción*

Para clasificar las abstracciones de las estructuras contenedoras utilizaremos las principales categorías que se encuentran en la literatura: estructuras lineales, estructuras jerárquicas, conjuntos, diccionarios, gráficas y almacenamiento secundario o externo.

a) Estructuras: *(abstracción, noción, clasificador ortogonal)*

- las categorías son excluyentes entre sí
- los términos dentro de cada categoría son excluyentes entre sí

Fig. A-2 Estructuras contenedoras

Por razones de extensión no definiremos cada término de esta noción. En lugar de ello, describiremos algunos criterios de clasificación que podrían prestarse a confusión.

Término	Descripción
Lineales	la estructura está compuesta por elementos que están organizados de manera que para cualquier elemento (salvo los extremos en estructuras no circulares) se puede identificar un único sucesor y un único predecesor. Una estructura lineal tiene dos extremos (cabeza-cola, principio-final, frente-atrás, tope-fondo). Las diferencias principales entre las estructuras lineales están dadas por el tipo de acceso a los elementos. Existen estructuras que permiten accesar todos sus elementos (acceso aleatorio), unas de manera directa (*arreglo*) y otras de manera secuencial (*lista, string*). En contraste, existen estructuras que no permiten accesar todos sus elementos (*pila, cola, bicola*) a estas estructuras también se les llama ***dispensadores*** porque sólo se pueden remover los elementos en una posición determinada. En la literatura encontramos con frecuencia que se asigna la misma semántica a estructuras lineales y secuenciales, aquí preferimos asignar significados diferentes. Una ***estructura secuencial*** es una estructura lineal que proporciona un tipo de acceso secuencial, es decir, permite accesar todos sus elementos sin removerlos y de manera secuencial.
Mapeo	una ***estructura de mapeo*** representa un mapeo de un conjunto dominio a otro conjunto imagen o rango. Se considera que en lugar de contener elementos individuales, contiene pares compuestos por un valor del dominio y un valor del rango. Dos características adicionales distinguen las estructuras de mapeo:

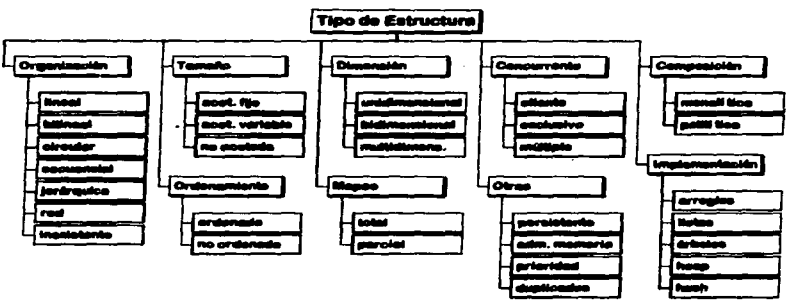
- siempre están ordenadas respecto al dominio, por lo que debe existir una relación de orden lineal total para los valores del dominio.
- los valores del dominio no se repiten, es decir, la estructura no contiene dos pares de valores para los que coincidan los valores del dominio.

Note que una estructura de mapeo también podría clasificarse como lineal si puede identificarse dos extremos, un predecesor y un sucesor para cada elemento. Por otro lado, un *arreglo* es también una estructura de mapeo.

Finalmente, una característica importante para este tipo de estructuras es si mapea o no todos los valores del dominio, por ejemplo, se sabe que para los enteros entre [1, 10^5] existen 10^5 elementos, por lo tanto un arreglo no es una estructura adecuada para contenerlos.

b) Tipo de Estructura: *(delimitador, noción, clasificador no ortogonal)*

Denota las características estructurales globales del contenedor.



- las categorías no son excluyentes entre sí
- los términos de la categoría *Otras* no son excluyentes entre sí
- ***Mapeo implica-a bidimensional*** o ***multidimensional*** y viceversa
- ***unidimensional excluye-a Mapeo***
- ***Implementación con arreglos implica-a acotada***
- ***Organización inexistente excluye-a ordenada***
- ***ordenada*** es ***parcial/o-a(A) Mapeo***
- ***Organización jerárquica*** y de ***red implica-a política***

Fig. A-3 Tipos de estructuras

1. Términos

Término	Descripción
Organización	indica la organización de los elementos en la estructura.
lineal	la estructura es lineal.
bilineal	es una estructura lineal que se puede recorrer en ambos sentidos.
circular	es una estructura lineal que tiene sus extremos unidos. Todos los elementos de la estructura tienen un sucesor y un predecesor.
secuencial	es una estructura lineal que proporciona un tipo de acceso secuencial. Una estructura secuencial tiene una ***posición actual*** que determina el elemento sobre el que se aplicará las operaciones de acceso. Ejemplos de estructuras secuenciales son: *pilas*, *colas*, y *listas*.
jerárquica (árbol)	existe la noción de niveles en la estructura. Los enlaces entre elementos están restringidos a uno hacia el nivel anterior y sólo existe un único elemento que no tiene un enlace hacia el nivel anterior, la raíz.
red	la organización de los elementos queda determinada por las relaciones que se establecen entre ellos, resultando en una red de relaciones (p.e., *gráficas*)
inexistente	no existe la noción de organización de los elementos, p.e. en un *conjunto* los elementos existen en la estructura pero no están organizados de ninguna manera en particular (desde un punto de vista abstracto).
Tamaño	indica si la estructura tiene o no una cota máxima predefinida para la cantidad de elementos que puede contener.
acotada	el tamaño de la estructura es determinado estáticamente, por lo que sólo puede contener un número predeterminado de elementos.
acotada fijo	no se puede modificar dinámicamente el tamaño de la estructura acotada.
acotada variable	se puede modificar dinámicamente el tamaño de la estructura acotada.
no acotada	el tamaño de la estructura es determinado dinámicamente, por lo que el número de elementos que puede contener queda determinado por la disponibilidad de memoria en tiempo de ejecución.
Ordenamiento	indica si la estructura es o no ordenada.
ordenada	los elementos en la estructura siempre mantienen un orden lineal, que debe estar definido para los elementos de la estructura.
no ordenada	los elementos no mantienen un orden lineal. Es irrelevante si existe definido un orden lineal para los elementos.
Dimensión	indica cuántas dimensiones abarca en la información que contiene.
unidimensional	la estructura sólo contiene elementos, no contiene información adicional para un elemento. La mayoría de las estructuras lineales son unidimensionales.
bidimensional	es una estructura de mapeo que establece una relación entre un valor de un dominio y un valor de un rango. Ejemplos de estructuras bidimensionales son *arreglos*, *tablas* y *diccionarios*.
multi-dimensional	es una estructura de mapeo que establece una relación entre varios valores de un dominio y un valor de un rango. Ejemplos de estructuras multidimensionales son *matrices*, *vectores* y *arreglos* con doble índice.
Mapeo	indica el tipo de mapeo que hace sobre el dominio de una estructura de mapeo.
total	es una estructura de mapeo que asigna un valor del rango para cada valor del dominio (p.e. *arreglo*).

parcial	es una estructura de mapeo que asigna un valor del rango para algunos de los valores del dominio (p.e. *diccionario*)
Concurrente[1]	la semántica de la estructura se preserva en la presencia de múltiples hilos de control.
cliente	la estructura proporciona algún mecanismo de control de concurrencia (p.e. un *semáforo*) para que la exclusión mutua sea lograda por la cooperación de todos los clientes de la estructura.
exclusivo	la exclusión mutua la controla la estructura y proporciona acceso exclusivo a cada usuario.
múltiple	la exclusión mutua la controla la estructura y proporciona acceso múltiple a varios clientes para operaciones de lectura y acceso exclusivo para operaciones de escritura.
Otras	características para las que no se encontró una noción adecuada.
persistente	permite almacenar y recuperar el contenido de toda la estructura a un medio de almacenamiento permanente (p.e., *archivos, bases de datos*).
administración de memoria	la estructura no acotada administra la memoria para las asignaciones y liberaciones de espacio. El motivo es proporcionar un mecanismo de administración de memoria más eficiente que el del ambiente de operación subyacente.
prioridad	la estructura proporciona acceso a los elementos con mayor prioridad.
duplicados	la estructura puede contener dos o más elementos con el mismo valor o estado.
Composición[2]	la justificación de esta clasificación recae en el concepto de *estructura compartida*, que significa que una estructura puede tener partes (subestructuras) que pueden ser referenciadas por más de un nombre. Si se modifica una subestructura a través de un nombre produce el efecto lateral de modificar las subestructuras para los otros nombres, por lo tanto, para estructuras polilíticas hay que tener cuidado con estos efectos laterales.
polilítica	una estructura polilítica contiene subestructuras que pueden ser manejadas independientemente. Ejemplos de estas estructuras son *listas, árboles* y *gráficas*.
monolítica	una estructura monolítica siempre se trata como una unidad, no se puede manipular subpartes de la estructura. Ejemplos de estas estructuras son *pilas, colas, anillos, mapas, conjuntos* y *bolsa*.
Implementación	indica la estructura o técnica principal de implementación de la estructura de datos.

c) Criterios de Uso: *(delimitador, noción, clasificador no ortogonal)*

Engloba recomendaciones prácticas para el uso de la estructura contenedora.

- las categorías no son excluyentes entre sí
- los términos de la categoría *Velocidad* no son excluyentes entre sí
- los términos de la categoría *Memoria* no son excluyentes entre sí

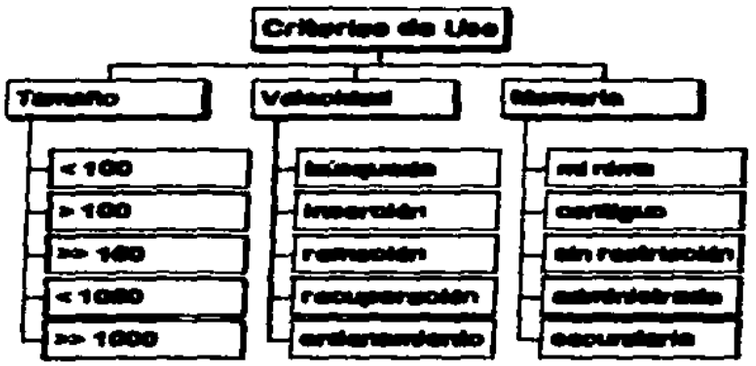


Fig. A-4 Criterios de uso

---

[1] Clasificación encontrada en [Booch 86:41]. Establece tres categorías de concurrencia con los nombres de *guardado, concurrente* y *múltiple*, que hemos renombrado a *cliente, exclusivo* y *multiple* respectivamente.

[2] Clasificación encontrada en [Booch 86:38].

1. Términos

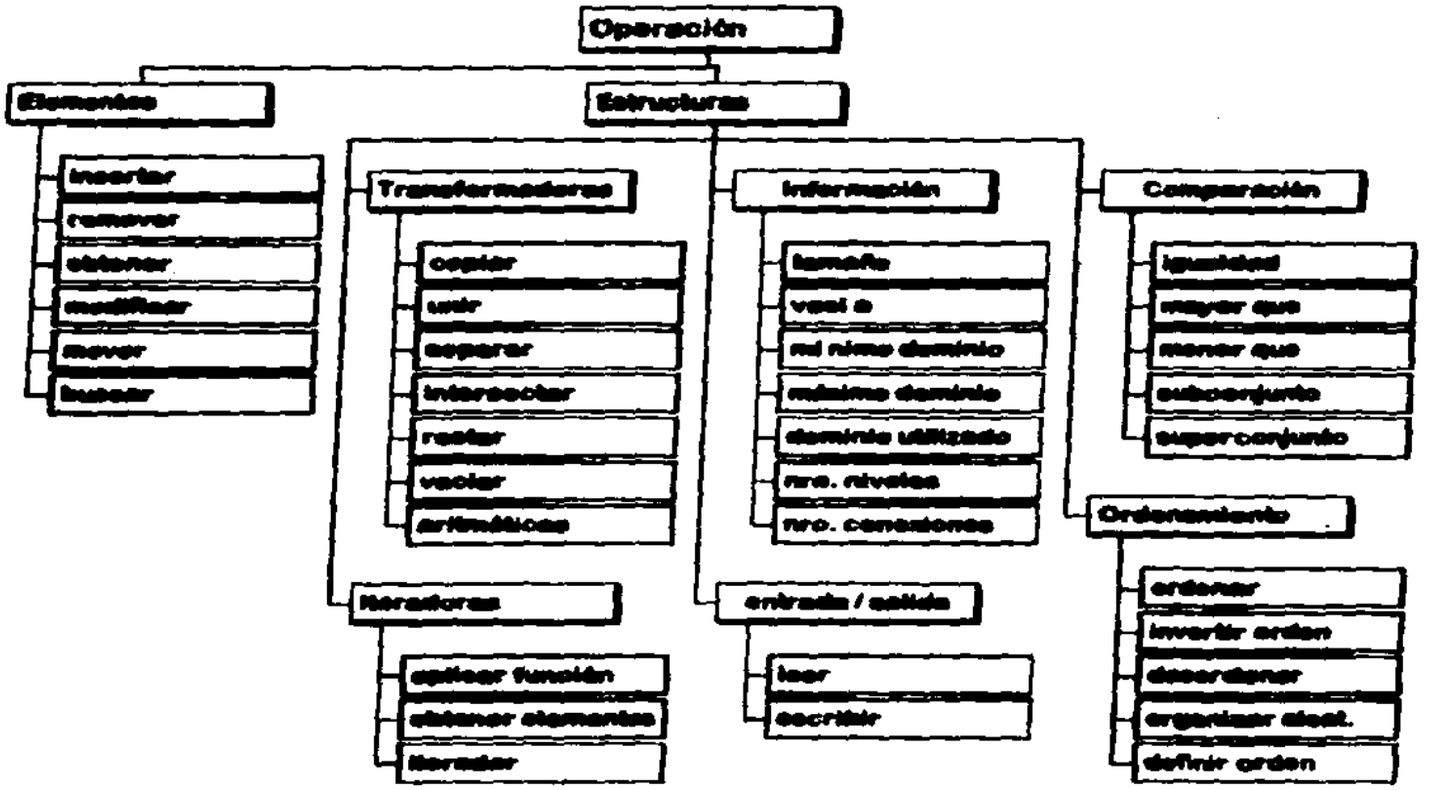
No se da una explicación para cada término (sólo para su categoría), pero todos los términos deben incluirse en el esquema.

Término	Descripción
Tamaño	indica la cantidad de elementos recomendada para obtener un buen rendimiento y/o el número de elementos que puede contener de manera eficiente.
Velocidad	indica cuáles son las operaciones sobre las que ofrece mayor velocidad.
Memoria	indica criterios para el uso de memoria de la estructura.

2. Faceta: *Servicios*

a) Operaciones: *(abstracción, noción, clasificador no ortogonal)*

Las operaciones las podemos dividir en operaciones sobre los elementos y operaciones sobre las estructuras. La mayor cantidad de métodos son operaciones de estructuras.



- las categorías no son excluyentes entre sí
- los términos dentro de cada categoría no son excluyentes entre sí

Fig. A-5 *Operaciones disponibles en estructuras de datos*

1. Términos

Término	Descripción
Elementos	operaciones que se aplican sobre elementos
insertar	agrega un elemento a la estructura, es un acceso constructivo
remover	remueve un elemento de la estructura, la mayoría de las veces regresa el elemento removido, es un acceso destructivo
obtener	regresa un elemento de la estructura sin removerlo, no es un acceso destructivo
modificar	modifica un elemento en la estructura sin removerlo, no es un acceso destructivo
mover	mueve un elemento dentro de la estructura
buscar	busca un elemento en la estructura
Transformadores	modifican el estado de una de las estructuras sobre las que opera o crea una nueva estructura en la que guarda el resultado de la operación.
copiar	en una estructura el contenido de otra estructura
unir	reúne los elementos de dos estructuras en una
separar	separa una estructura en dos, cada una conteniendo un subgrupo de los elementos de la colección original.
intersectar	crea una nueva estructura con los elementos de intersección de dos o más estructuras.
restar	la estructura resultante contiene los elementos de una estructura que no están en otra estructura.
vaciar	remueve todos los elementos de una estructura
aritméticos	se utilizan operadores aritméticos (+, -, *, /) para algunas operaciones de estructuras.
Iteradores	la estructura proporciona operaciones iteradoras
aplicar función	el usuario puede especificar una función o procedimiento que se aplicará sucesivamente a cada elemento.
obtener elementos	proporciona una manera de asignar sucesivamente a una variable el valor o la referencia de cada elemento en la estructura.
iterador	la estructura proporciona un elemento iterador junto con todas las operaciones básicas para recorrer todos los elementos: asignar el iterador a una posición inicial, avanzar a la siguiente posición, obtener el elemento en la posición del iterador y determinar si ya terminó de recorrer todos los elementos. También podría proporcionar operaciones para recorridos bidireccionales.
Información	operaciones que regresan información acerca de la estructura.
tamaño	determina el número de elementos que contiene la estructura.
vacía	determina si la estructura no contiene elementos.
mínimo dominio	para estructuras de mapeo regresa el mínimo valor del dominio utilizado. Cuando la estructura es indexada regresa el valor del mínimo índice.
máximo dominio	para estructuras de mapeo regresa el máximo valor del dominio utilizado. Cuando la estructura es indexada regresa el valor del máximo índice.
dominio utilizado	para una estructura de mapeo determina si se ha utilizado un valor del dominio para establecer una relación con el rango. En un *arreglo* todos los valores del dominio son utilizados.
nro. de niveles	para una estructura jerárquica determina el número de niveles que tiene la estructura
nro. conexiones	para una estructura con organización de red determina el número de conexiones que existen entre los nodos.

entrada / salida	proporciona operaciones de entrada / salida para la estructura, en general las operaciones de entrada / salida existen en pares, es decir, existe una operación de entrada y otra de salida. Los medios para leer y escribir pueden ser varios (archivo, display, base de datos)
Comparación	operaciones que comparan estructuras y regresan un valor booleano
igualdad	determina si dos estructuras del mismo tipo tienen el mismo contenido.
mayor que	determina si una estructura es mayor que otra (p.e., nro. de elementos)
menor que	determina si una estructura es menor que otra (p.e., la suma de las prioridades)
subconjunto	determina si todos los elementos de una estructura están contenidos en otra
superconjunto	determina si la estructura contiene todos los elementos de otra estructura.
Ordenamiento	operaciones de ordenamiento en la estructura.
ordenar	permite ordenar todos o algunos elementos en la estructura.
invertir orden	invierte el orden de los elementos para un recorrido determinado. La operación tiene mayor sentido en estructuras secuenciales.
desordenar	reorganiza los elementos de manera que para un orden de recorrido, dos elementos contiguos no satisfacen la relación de orden lineal definida para los elementos.
organizar aleatoriamente	reorganiza los elementos en la estructura aleatoriamente. No existe la restricción de que dos elementos contiguos no deben satisfacer la relación de orden.
definir orden	permite especificar la función de comparación que define la relación de orden lineal.

b) Complejidad de Tiempo: *(delimitador, noción, clasificador ortogonal)*

Una de las características de mayor peso en la selección de una estructura de datos es una medida del tiempo de ejecución de las operaciones que proporciona, pues determina el tamaño del problema que podemos resolver.

La ***complejidad de tiempo*** es el tiempo requerido por una operación, expresada como una función del tamaño (número de elementos) de la estructura. Una notación común para esta medida es la ***"O mayúscula"*** (big-oh), que permite expresar una cota máxima para la complejidad en tiempo $T(n)$ de una operación en una estructura de tamaño n:

$T(n) \leq c \cdot O(f(n))$, c es una constante

que expresa que el tiempo de ejecución máximo de la operación es proporcional a una función que depende del número de elementos de la estructura. Por lo general, no es necesario determinar el valor de la constante c, lo más importante es determinar $f(n)$ pues da la información de cómo se comporta la operación con variaciones del número de elementos de la estructura.

En la práctica, se ha encontrado que existen funciones $f(n)$ típicas que representan el comportamiento en tiempo de muchas de las operaciones [Booch 86:45]:

$f(n)$	comportamiento	duración de la operación
1	constante	no depende del tamaño de la estructura
$\log n$	logarítmico	se duplica cuando n aumenta a n^2
n	lineal	se duplica cuando n se duplica
$n.\log n$	linearítmico	no es lineal, cada vez que n se duplica la duración aumenta por la relación $(2 + 2/(\log n))$. A medida que n aumenta se acerca más a la conducta lineal: para duplicar a (n = 4, 8, 32, 128, 256, 1024) la duración de la operación aumenta en (4, 3, 2.5, 2.29, 2.25, 2.2) respecto a la duración anterior.

n^2	cuadrático	se cuadruplica cuando n se duplica
n^3	cúbico	se octuplica cuando n se duplica
2^n	exponencial	excesivamente grande, pues se duplica el exponente

Estos son algunos de los valores que puede adoptar una noción de complejidad en el esquema de clasificación. También podríamos incluir otros, por ejemplo: $n^{1/4}$, $n^{1/3}$, $n^{2/3}$, $n^2.\log n$, $(\log n)^2$.

c) Tipo de Acceso: *(delimitador, noción, clasificador no ortogonal)*

El ***tipo de acceso*** de una estructura de datos determina las formas que se pueden utilizar para manipular sus elementos individuales. El tipo de acceso aplica a las operaciones de elementos en la estructura.

- las categorías no son excluyentes entre sí
- la categoría *Restringido excluye-a* la categoría *Aleatorio*
- la categoría *Restringido excluye-a* la categoría *Con búsqueda*
- *búsqueda por valor implica-a búsqueda por condición*
- *mínimo y máximo valor implica-a por valor*
- los términos de la categoría *Aleatorio* no son excluyentes entre sí
- *sucesor* y *predecesor implica-a secuencial*
- *iterativo* es *parecido-a(bf) secuencial*
- el acceso *no permitido* excluye a los otros tipos de acceso

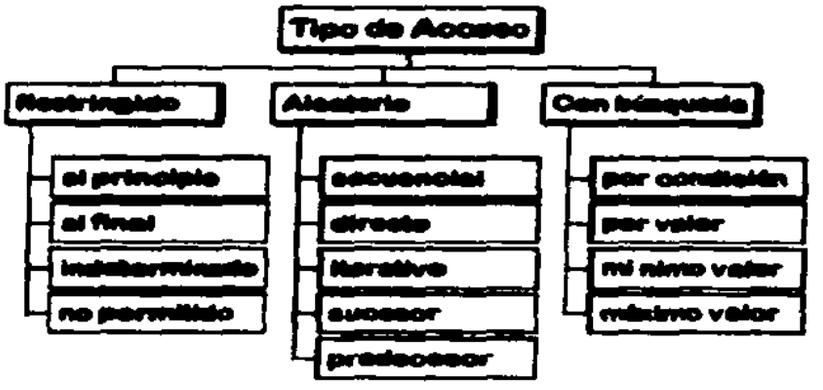


Fig. A-6 Tipos de acceso

1. Términos

Término	Descripción
Restringido	permite accesar un elemento sólo en ciertos lugares de la estructura.
al principio	permite accesar un elemento en el extremo de inicio de una estructura secuencial (cabeza, frente, tope).
al final	permite accesar un elemento en el extremo final de una estructura secuencial (cola, atrás, fondo).
indeterminado	remueve o recupera algún elemento de la estructura, pero el usuario no indica cuál.
no permitido	puede ser que no permita una de las operaciones de acceso, p.e., un *arreglo* no proporciona remoción.
Aleatorio	permite accesar cualquier elemento.
secuencial	hay que accesar los elementos anteriores o posteriores para accesar un elemento.
directo	se puede accesar cualquier elemento sin recorrer los predecesores o sucesores.
iterativo	proporciona una operación iteradora que permite recorrer los elementos de la estructura.
sucesor	permite accesar el elemento posterior a otro.

predecesor	permite accesar el elemento anterior a otro.
Con búsqueda	el acceso a un elemento implica buscarlo en la estructura.
por condición	indica que el valor de uno o de un conjunto de atributos de un elemento deben satisfacer una condición determinada.
por valor	indica que el valor de uno o de un conjunto de sus atributos sea igual a un valor o conjunto de valores. El acceso por valor es un caso particular del acceso por condición, donde la condición es la igualdad.

d) Tipo de Remoción: *(delimitador, noción, clasificador no ortogonal)*

Adicionalmente a las propiedades denotadas en el tipo de acceso, podemos identificar tipos diferentes de remoción.

una ocurrencia	remueve una ocurrencia de un elemento en estructuras que permiten duplicados.
todas ocurrencias	remueve todas las ocurrencias de un elemento en estructuras que permiten duplicados.
global	remueve todos los elementos de la estructura.
física	elimina el elemento del contenedor.
lógica	indica que el elemento ha sido removido pero no hace la remoción física. Esta técnica se utiliza para acelerar la operación de remoción y está asociada con una operación iteradora de depuración que realiza la remoción física de todos los elementos con indicación de remoción lógica.

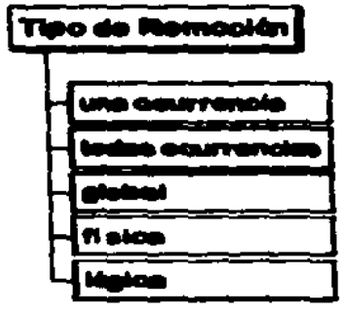
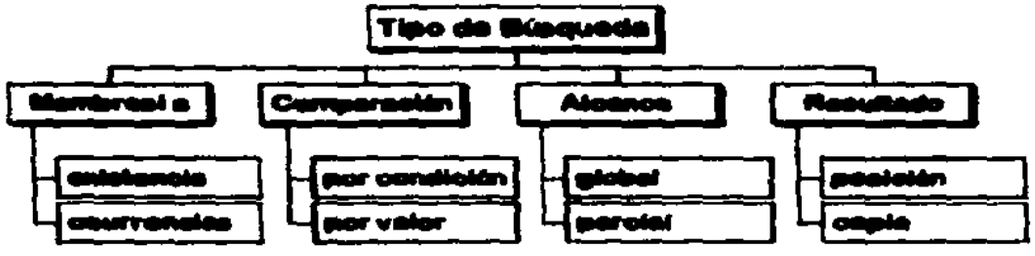


Fig. A-7 Tipos de remoción

- los términos dentro de la categoría no son excluyentes entre sí.

e) Tipo de Búsqueda: *(delimitador, noción, clasificador no ortogonal)*

Formas que se pueden utilizar para buscar un elemento en la estructura. Aplica a la operación de búsqueda de elementos.



- las categorías no son excluyentes entre sí
- los términos dentro cada categoría no son excluyentes entre sí
- *búsqueda por valor implica-a búsqueda por condición*

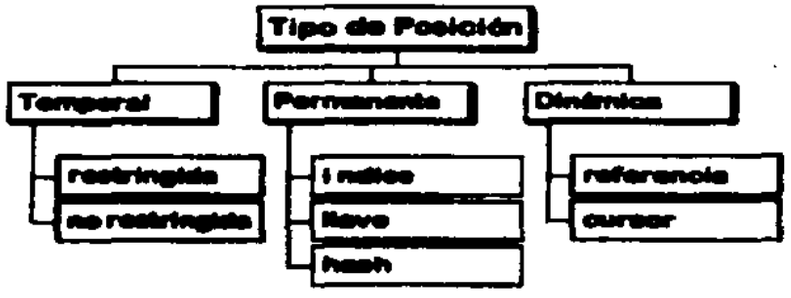
Fig. A-8 Tipos de búsqueda

i. Términos

Término	Descripción
existencia (membresía)	determina si un elemento está o no en la estructura. La prueba de existencia puede o no regresar la ubicación del elemento para futuros accesos.
ocurrencias	función numérica que determina el número de ocurrencias de un elemento en una estructura que permite duplicados.
por condición	busca un elemento para el que el valor de uno o de un conjunto de sus atributos satisfaga una condición determinada
por valor	busca un elemento para el que el valor de uno o de un conjunto de sus atributos sea igual a un valor o conjunto de valores. Este es un caso particular de la búsqueda por condición, donde la condición es la igualdad.
global	busca en toda la estructura.
parcial	busca en un subgrupo de elementos.
posición	regresa la posición del elemento para futuros accesos.
copia	regresa una copia del elemento encontrado

f) Tipo de Posición: *(delimitador, noción, clasificador no ortogonal)*

Un criterio relacionado estrechamente con el tipo de acceso a una estructura de datos es la ***noción de posición***, es decir, si podemos asociar o no una ubicación específica para cada elemento dentro de la estructura. Aplica a las operaciones de elementos.



- las categorías no son excluyentes entre sí
- la categoría *temporal excluye-a permanente*
- *hash implica-a llave*
- los términos de la categoría *Dinámica* no son excluyentes entre sí

Fig. A-9 *Tipos de posición*

i. Posición temporal

Estamos ante una ***posición temporal*** cuando la inserción o remoción de elementos de la estructura modifica la posición de otros elementos. Todas las estructuras ligadas tienen una posición temporal, pues se modifican las ligas con las operaciones de inserción y remoción. Note que esta afirmación no implica que las estructuras implementadas con estructuras ligadas tengan una posición temporal. Aclaremos.

La noción de posición se refiere a una ubicación en la estructura como abstracción, no a una ubicación que se podría identificar para una implementación particular. Por ejemplo:

- una lista se puede implementar con un arreglo, las operaciones de inserción y remoción no modifican la posición de los elementos en el arreglo, pero si modifican la posición en la lista.
- un diccionario se puede implementar con una estructura de árbol ligada, pero la posición que interesa para la abstracción es la llave y no la posición en el árbol.

La posición temporal la podemos clasificar en:

- ***Restringida***: permite accesar elementos sólo en ciertos lugares de la estructura. La estructura determina la posición para el acceso (p.e., ***pilas***, ***colas*** y ***bicolas***).
- ***No restringida***: las operaciones de acceso no están restringidas a una posición.

ii. Posición permanente

El concepto de ***posición permanente*** está presente en una estructura cuando el usuario debe identificar explícitamente una posición específica para insertar, remover o recuperar un elemento. La noción de posición permanente implica un mapeo de un conjunto de valores (llave, índice) a otro (elemento, información), por lo tanto, siempre que está presente el concepto de posición estamos ante una estructura de mapeo y viceversa.

Definimos las siguientes condiciones para el concepto de ***posición***:

- no puede haber dos posiciones con el mismo valor en una misma estructura.
- debe existir un orden lineal total definido para el tipo de dato posición[3].
- todos los elementos de una estructura se pueden accesar especificando su posición.

Los accesos por posición los podemos clasificar en:

- ***índice***: la posición es un entero.
- ***llave***: existe un componente del elemento que se puede utilizar para identificarlo de manera única.
- ***hash***: debe existir una función de dispersión para los valores del dominio.

iii. Posición dinámica

Una ***posición dinámica*** es una ubicación que se puede asociar a un elemento dinámicamente, es decir, su valor queda determinado en tiempo de ejecución y varía de una ejecución particular a otra de un programa.

- Referencia

Es un mecanismo de acceso que consiste en una ***referencia*** a la ubicación en memoria de un elemento, por lo tanto, proporciona un tiempo de acceso constante ***O(1)*** una vez que se ha encontrado el elemento en la estructura. La ubicación dinámica sólo puede existir adicionalmente a otro mecanismo de acceso principal. Además, no hay restricciones para el número de referencias, es decir, puede existir más de una referencia para un mismo elemento y todos los elementos pueden estar referenciados. Ejemplos de este mecanismo son:

- si en un ***diccionario*** un usuario recuperó una información ***I*** con una llave ***K***, para futuras operaciones de acceso podría utilizar su posición dinámica en lugar de la llave.

---

[3] Aunque exista este orden lineal, la estructura no necesariamente proporciona las operaciones de ir al elemento predecesor o sucesor.

- si en una *lista* se ha recorrido *m-1* elementos para accesar el elemento *m*, en futuras operaciones de acceso se puede utilizar su ubicación dinámica en lugar del recorrido.

Finalmente, si se quiere aplicar una operación de acceso a una referencia se debe especificar explícitamente la referencia que indica la posición. Note que el acceso por referencia es un mecanismo auxiliar, no el principal que caracteriza a la estructura.

- Cursor

La existencia de un ***cursor*** está asociada a una ***posición actual*** en la estructura sobre la que se aplican las operaciones de acceso. Si existe un sólo cursor para una estructura no es necesario especificar explícitamente el valor del cursor para realizar la operación de acceso, se aplica en la posición del cursor.

El cursor implica la noción de recorrido de la estructura y es la base de las operaciones iteradoras. Para ello, existen operaciones que permiten manejar el cursor: ubicarlo al principio, avanzar al siguiente elemento, retroceder al anterior y saber cuando se han recorrido todos los elementos. Podríamos pensar un ***cursor restringido*** que sólo se posicione en los elementos que satisfagan cierta condición.

Note que para estructuras secuenciales las operaciones de cursor tienen un significado natural, pero para otro tipo de estructuras no es tan sencillo, por ejemplo, para una estructura jerárquica hay que definir un orden de recorrido de los nodos.

g) Tipo de Ordenamiento: *(delimitador, noción, clasificador no ortogonal)*

El objetivo fundamental del ordenamiento es acelerar la búsqueda de un elemento. Una estructura ***está*** ordenada si mantiene una relación de orden lineal total entre sus elementos. Una estructura ***es*** ordenada si siempre mantienen la relación de orden, en particular, bajo la aplicación de las operaciones que modifican la estructura (inserción y remoción básicamente). Una estructura no ordenada se puede ordenar temporalmente si tiene definida una relación de orden lineal total. Note que:

- en una estructura ordenada, el usuario no puede especificar una posición de inserción o remoción
- una estructura que proporcione una operación de ordenamiento no es una estructura ordenada.

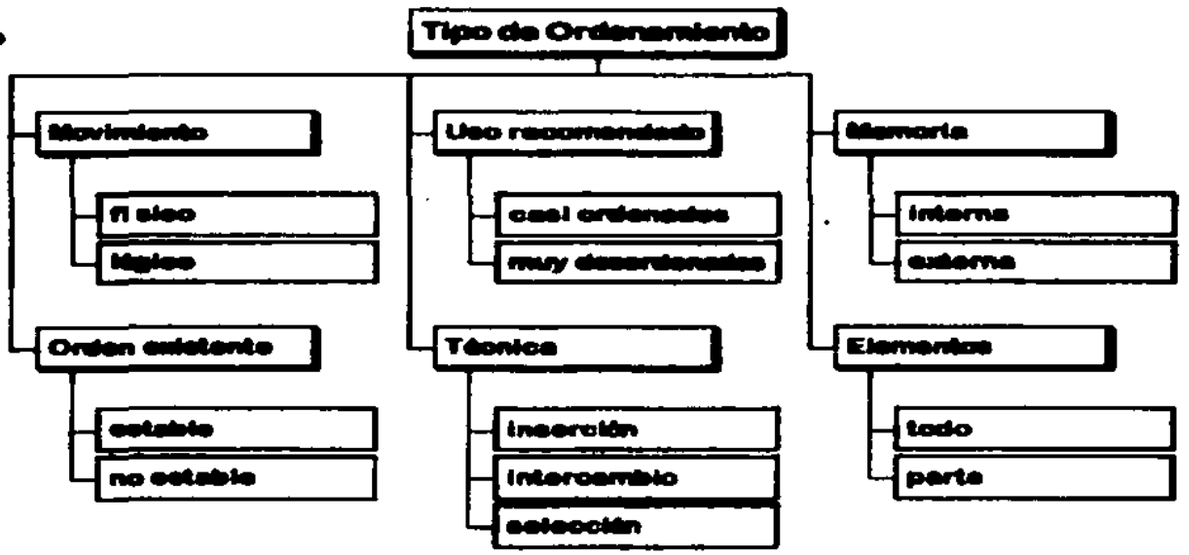


Fig. A-10
Tipos de Ordenamiento

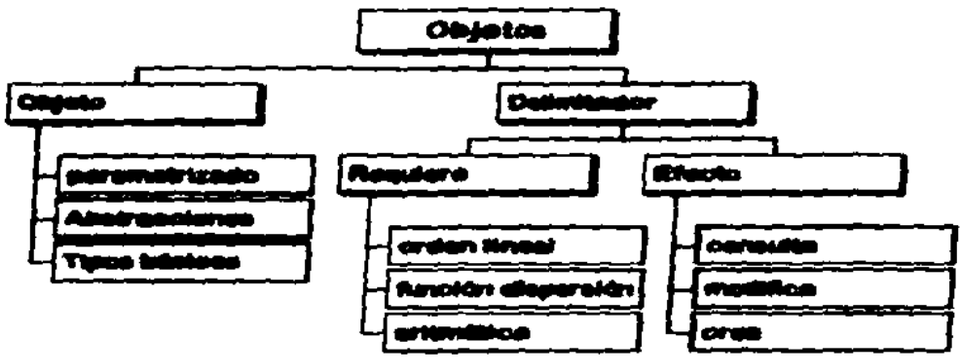
2. Términos

Término	Descripción
físico	(o *directo*) se cambia de lugar el elemento.
lógico	(o *indirecto*) se reorganiza una característica (identificador, llave) del elemento.
estable	preserva el orden original de los elementos, es decir, si encuentra dos elementos iguales no cambia la posición de uno respecto al otro.
no estable	no preserva el orden original de los elementos.
casi ordenados	se recomienda su uso cuando los elementos en la estructura están casi ordenados.
muy desordenados	se recomienda su uso cuando los elementos en la estructura están muy desordenados.
inserción	examina un elemento a la vez y lo inserta en la estructura en el orden relativo a todos los elementos previamente examinados.
intercambio	si existen elementos fuera de orden selecciona dos y los intercambia.
selección	encuentra el siguiente ítem más grande (o pequeño) y lo coloca en su posición definitiva.
interna	los elementos están en memoria principal.
externa	los elementos están en memoria secundaria.

3. Faceta: *Objetos*

a) Objeto: ***(abstracción, noción, clasificador ortogonal)***
Requiere: ***(delimitador, noción, clasificador no ortogonal)***
Efecto: ***(delimitador, noción, clasificador ortogonal)***

Denota características de los elementos que puede contener la estructura.



- las categorías no son excluyentes entre sí
- los términos en cada categoría no son excluyentes entre sí

Fig. A-11 Requerimientos y servicios para elementos

1. Términos

Término	Descripción
Objeto	*indica el tipo de objeto que manipula.*
parametrizado	*la estructura puede manipular cualquier tipo de elemento.*
Abstracciones	*puede manipular las mismas abstracciones del dominio.*
Tipos básicos	*puede manipular los tipos de datos básicos.*
Requiere	*indica que característica requiere sobre el elemento que manipula.*
orden lineal	*debe existir una definición de orden lineal para el tipo de elemento. Para estructuras de mapeo se requiere un orden lineal para los valores del dominio.*
función de dispersión	*debe existir una función de dispersión para el tipo de elemento.*
operaciones aritméticas	*requiere que se pueda aplicar las operaciones aritméticas comunes (+, -, *, /) sobre el tipo de elemento.*
Efecto	*indica el efecto que tiene sobre el objeto que manipula.*
consulta	*sólo consulta el valor o estado del objeto.*
modifica	*el estado del objeto.*
crea	*construye un nuevo objeto.*

4. Comparación de elementos

La comparación de elementos se utiliza para determinar la existencia de una relación de igualdad o una relación de orden lineal entre elementos. La primera se utiliza para la búsqueda de elementos y la segunda para ordenamiento y búsqueda.

a) Relación de igualdad

Es importante saber si la estructura contenedora contiene referencias a elementos o los elementos mismos, pues ello determina dos maneras diferentes de considerar la igualdad de elementos:

- ***Igualdad de referencias***: *determina que dos elementos son iguales cuando las referencias son iguales, es decir, cuando están referenciando a un mismo elemento. Sólo aplica a estructuras que contienen referencias.*
- ***Igualdad de estados***: *determina que dos elementos son iguales cuando sus estados son iguales*[4]*. Puede aplicar cuando la estructura contiene referencias o elementos.*

La igualdad de referencias implica la igualdad de objetos pero el inverso generalmente no es cierto. Por otro lado, la igualdad de referencias utiliza el criterio de igualdad de un tipo de dato primitivo (básico), mientras que la igualdad de estados utiliza una operación de comparación redefinible para cada tipo de elemento. La implementación por omisión de esta operación es la

[4] Si el elemento tiene datos que son referencias, la comparación de estado puede extenderse en profundidad a los elementos que referencia y así sucesivamente.

comparación atributo por atributo de los estados de los elementos, invocando las operaciones de igualdad definidas para los tipos extendidos (clases) y la igualdad por omisión para los tipos primitivos.

Una propiedad interesante para estructuras que almacenan referencias es poder indicar dinámicamente qué tipo de comparación se utilizará. De esta manera se pueden utilizar algoritmos comunes con diferente semántica, la determinada por el estado de la estructura.

b) Relación de orden lineal

Frecuentemente se requiere una relación de orden lineal para los elementos: para mantener la estructura ordenada, para ordenar la estructura, para buscar un elemento en una estructura ordenada.

Una relación de orden lineal total es una relación binaria (entre dos elementos) que debe satisfacer las siguientes propiedades: [Meyer 94:136]

Irreflexibidad	$a<a$ siempre es falso
Transitividad	$a<c$ siempre que $a<b$ y $b<c$
Asimetría	$a<b$ y $b<a$ no son nunca ciertos simultáneamente
Completez	para cualesquiera a y b, una de las relaciones debe ser cierta: $a<b$, $b<a$, $a=b$ (igualdad de estados)

Las otras operaciones de comparaciónse pueden estructurar como: $(a<=b) = a<b$ *or* $(a=b)$; $(a>=b) = (b<=a)$; $(a>b) = (b<a)$. De esta manera las operaciones de comparación quedan determinadas por las relaciones *'menor que'* e *'igual que'*. Estas son las relaciones que debe definir el usuario sobre los elementos que quiere almacenar cuando se requiere una relación de orden lineal[5].

[5] Los tipos básicos o primitivos (enteros, caracteres, strings, apuntadores, etc.) tienen definida una relación de orden lineal.

ANEXO B:
Análisis de Dominio

La evolución de la ingeniería de software ha traído consigo el surgimiento de variadas metodologías para el desarrollo de sistemas (cascada, espiral, construcción de prototipos, etcétera). Estas metodologías han sido ampliamente probadas en la práctica con mayor o menor éxito dependiendo de las características particulares de las demandas computacionales y de la manera como se han utilizado. En el contexto del reuso de software, surgen las interrogantes: ¿cuál es la mejor metodología existente para aprovechar al máximo la tecnología del reuso? o ¿será necesario inventar una nueva metodología?.

Especificar cómo incorporar el reuso de software a los procesos de desarrollo existentes o más aún, definir lineamientos para la creación de una nueva metodología, son pretensiones que exceden los alcances perseguidos en este trabajo. Sin embargo, como punto de partida consideramos que el reuso debe ser un elemento orientador y en consecuencia debe estar inmerso en una visión integral del proceso de desarrollo. Esta percepción demanda reformular planteamientos existentes y/o crear nuevas metodologías y procedimientos. Sólo de esta manera se podrá alcanzar todos los beneficios esperados.

Otro lineamiento básico a considerar es el(los) modelo(s) de componente(s) reutilizable(s) que propone la metodología, es decir, cómo representar los elementos que se utilizan en la construcción de un sistema. Hay que considerar cuidadosamente las técnicas de agrupamiento o empaquetamiento y el nivel de granularidad y flexibilidad que permiten. De especial interés es la evaluación de las interfaces y formas de interconexión de componentes, pues determinarán el éxito de la construcción y de la integración de los componentes reutilizables.

Son igual de importantes también, el nivel de automatización y promoción de la actividad de reuso mediante ambientes y herramientas de desarrollo integradas a la metodología. Esta automatización debe estar presente en todas las fases del desarrollo, debe obligar a seguir la metodología y sobre todo ayudar a identificar las oportunidades de reuso y a promover la construcción de componentes reutilizables.

Finalmente, existe otra serie de elementos que considerar y que dependen en mucho de las características particulares de cada organización y proyecto. Entre estos podemos citar, las metodologías existentes de desarrollo, el tipo y capacitación del personal, el tipo de aplicaciones, la inversión que puede hacerse, los beneficios que se esperan, las demandas de tiempos de entrega, el tamaño de los proyectos y de la organización, la infraestructura que se desea aprovechar, los costos de modificación o reemplazo, etcétera.

Adicionalmente a estos aspectos generales, quisimos mencionar una propuesta metodológica concreta por tener una relación muy estrecha con la construcción y uso de componentes reutilizables, nos referimos al análisis de dominio.

1. Análisis de dominio

El análisis de dominio estudia clases de sistemas dentro de un dominio de aplicación para determinar los requerimientos globales del conjunto de sistemas y para construir una arquitectura de software general que oriente la construcción de cualquier aplicación dentro del dominio.

Además de determinar los requerimientos existentes puede anticipar requerimientos futuros para el conjunto de sistemas.

El proceso de análisis de dominio genera una arquitectura, subsistemas, módulos, funciones e interfaces comunes a todos los sistemas bajo estudio. Los componentes y sistemas que se construyen siguiendo la arquitectura descubierta tienen un gran potencial de reuso, pues sus características pertenecen a una clase de sistemas y no a alguno en particular. Es importante destacar que aunque el análisis de dominio lo realizan expertos del dominio, los productos resultantes evolucionan con el uso y se enriquecen con los aportes de los usuarios, por lo que debe contemplarse un sistema explícito de realimentación que registre los defectos y nuevas necesidades.

I. ¿Cuándo realizar un análisis de dominio?

Un análisis de dominio requiere un esfuerzo bastante mayor que el análisis de requerimientos de un sólo sistema y debe justificarse esta inversión adicional. Algunos criterios básicos para tal justificación son:

- comprobar la necesidad de construir más sistemas en el mismo dominio
- verificar que el dominio sea muy bien comprendido y que exista documentación suficiente, inclusive podría existir algún análisis para el dominio de interés
- verificar que se disponga de suficiente experiencia como para generalizar el conocimiento del dominio. Se recomienda disponer de expertos en el dominio.
- asegurar que los productos del análisis de dominio serán utilizados por todos los que construyan sistemas en el mismo dominio

Existe la opción de realizar un análisis de dominio parcial para identificar algunos componentes medulares y diseñarlos para reuso. Los candidatos inmediatos pueden ser las interfaces a dispositivos estándares, interfaces con otros sistemas, protocolos de comunicación, interfaz con el usuario, algoritmos específicos, manejo de mensajes, manejo de errores, etcétera. Aún sin construir la arquitectura genérica para todos los sistemas del dominio, se puede obtener componentes genéricos y flexibles con buenas características de reusabilidad.

II. ¿Qué se obtiene de un análisis de dominio?

PRODUCTOS DEL ANALISIS DE DOMINIO

Modelo de dominio

el paso inicial es modelar el dominio, es decir, definir funciones, datos, dependencias y relaciones. Esto conforma los requerimientos generales del conjunto de aplicaciones y define las reglas y principios del dominio. Debe indicar los alcances del dominio, las entradas y salidas principales y especificar un vocabulario estándar.

Arquitectura de software

después de definir el modelo general se plantea soluciones a los problemas del dominio. La arquitectura de software define los componentes de software fundamentales, las interfaces de conexión y el control de ejecución. La arquitectura es un diseño genérico de alto nivel y se utiliza como base para construir aplicaciones y para mapear los requerimientos del modelo a los componentes de diseño. La mayor

influencia de la arquitectura está en la definición de los requerimientos junto con el modelo del dominio.

Componentes de diseño

se derivan de las especificaciones de la arquitectura y se utilizan para la construcción del diseño de la aplicación.

Componentes de implementación

se derivan de los componentes de diseño y están en el nivel más bajo de abstracción. Un componente de implementación puede ser código, pruebas, documentación e inclusive ejecutables.

III. ¿Cuándo utilizar los productos de un análisis de dominio?

Lo primero que hay que determinar es si el sistema que se desea construir es efectivamente un miembro representativo del dominio estudiado y verificar que la arquitectura propuesta es la adecuada para el sistema concreto. Por otro lado, puede que sólo exista un modelo de la arquitectura pero no exista diseño detallado ni componentes de código, en cuyo caso no vale mucho la pena apegarse estrictamente al modelo.

Por otro lado, si existen componentes de diseño y de código hay que verificar cuan bien se ajustan al análisis de dominio, es decir, cuánta funcionalidad proporcionan y si son fieles a las interfaces de comunicación. Es importante sobre todo saber si se han utilizado en la práctica, que nivel de evolución tienen y si son productos confiables. Si ya se han utilizado en la práctica, es conveniente saber si han logrado estandarizar la construcción de sistemas para el dominio o si hay diferencias sustanciales entre la arquitectura genérica y las implementaciones concretas.

Finalmente, aún cuando los resultados de este examen inicial no arroje resultados muy satisfactorios, por lo general un análisis de dominio es muy valioso y se podrá reutilizar información para la especificación del sistema y/o componentes para las diferentes fases.

Página intencionalmente en blanco

ANEXO C:
Sugerencias para la construcción de CSR

Los aspectos que giran alrededor de la noción de un CSR son extensos: construcción de CSRs, modelos de componentes y modelos de composición, certificación y evaluación de componentes, niveles de abstracción, correspondencia entre abstracción y realización, especificación de componentes y documentación.

Desde la perspectiva del reuso de componentes concretos los elementos más relevantes son el modelo de componente y el de composición, que los abordamos en el capítulo 3. Por ahora, nos centraremos en las generalidades de la construcción de componentes reutilizables. Para ello, asumiremos un esquema general de construcción de sistemas organizado en fases de análisis, diseño y programación, con lo que tendremos tres tipos de componentes claramente diferenciados. Note que el impacto del reuso de software es mayor en las primeras fases del desarrollo, pues por lo general el reuso de un componente a este nivel implicará el reuso de otros componentes en las fases posteriores.

1.1 Reuso en la fase de análisis

El análisis abarca la captura de los requerimientos del sistema y los plasma en especificaciones que indican sin ambigüedades (el ideal) lo que se debe desarrollar. Desde la perspectiva de reuso esta fase debe:

- reutilizar especificaciones de requerimientos existentes
- construir especificaciones de requerimientos reutilizables
- identificar oportunidades de reuso en las siguientes fases y sugerir el uso de componentes existentes en el depósito
- especificar las características de componentes reutilizables que deben construirse en las siguientes fases
- especificar el nivel de reuso que se espera lograr dentro del proyecto y la manera como se garantizará y medirá el logro de este objetivo

1.1.1 Componentes reutilizables de requerimientos

Una *especificación de requerimientos* puede corresponder a uno o un conjunto de requerimientos concretos. Si esta especificación se repite o se espera que se necesite en otros sistemas, podemos tratarla como un *componente reutilizable de requerimientos* y en general debe estar asociado con uno o más diseños. El reutilizar una especificación de requerimientos implica un ahorro significativo en la especificación del sistema, disminuye la posibilidad de error en la captura de los requerimientos y estandariza las especificaciones. Adicionalmente, es posible que se reutilicen todos o algunos de sus elementos asociados (documentación, diseños, programas, pruebas), con lo que potencia desde el principio el reuso en las siguientes fases.

1.1.2 Sugerencias para especificar requerimientos reutilizables

Para tratar los requerimientos como componentes reutilizables, es conveniente hacer la especificación siguiendo algunas convenciones que faciliten su reutilización.

Sugerencias para especificar requerimientos reutilizables

Sugerencias de notación

- emplear un modelo de componente para las especificaciones de requerimientos, de manera que puedan agruparse requerimientos atómicos y no atómicos, contener o referenciar documentación, especificaciones de diseños y programas asociados en el depósito o sugeridos por el analista
- indicar con alguna notación la diferencia entre requerimientos específicos de un sistema, requerimientos reutilizados y requerimientos con potencial de reutilización
- asegurar la unicidad de especificación de un mismo requerimiento. Si se tiene más de una especificación equivalente se puede dejar sólo una en el depósito o mantener referencias ente ellas
- adoptar un método de especificación uniforme para poder reutilizar las especificaciones en diferentes sistemas
- si una especificación se reutiliza sin modificación es conveniente mantener una referencia a la especificación en lugar de incluirla textualmente

Sugerencias de elaboración

- no especificar ningún mecanismo de implementación para un requerimiento, de manera que se amplíe las posibilidades de elección de los componentes reutilizables de diseño y sus programas asociados
- evaluar la posibilidad de cambiar parte de los requerimientos para aumentar la posibilidad de reuso de especificaciones y/o componentes en las siguientes fases
- plantear como un requerimiento si parte del software tiene un reuso potencial en otros sistemas (esto garantiza el desarrollo de componentes software reutilizables). Debe especificarse explícitamente si se requiere un alto nivel de generalidad, si será reutilizado en varias plataformas o si deberá adaptarse a diferentes interfaces de interconexión. Deberá especificarse la documentación que acompañará al componente, las pruebas que debe satisfacer y la conformancia a algún estándar
- plantear como un requerimiento si se espera que para una especificación se utilice alguna implementación particular del depósito en las siguientes fases
- especificar la reusabilidad como un requerimiento del proyecto y asegurar que exista una manera de medir y evaluar la satisfacción de este requerimiento

1.2 Reuso en la fase de diseño

El diseño construye especificaciones de diseño para las demandas de las especificaciones de requerimientos. Desde la perspectiva de reuso esta fase debe:

- seguir las orientaciones para el reuso especificadas en la fase anterior
- seleccionar o desarrollar un diseño para cada especificación de requerimientos que fue reutilizada

- identificar oportunidades de reuso de componentes de diseño existentes adicionales a los identificados en la fase anterior
- identificar oportunidades futuras de reuso y especificar y definir nuevos componentes de diseño reutilizables
- identificar oportunidades de reuso en las siguientes fases y sugerir el uso de componentes existentes en el depósito
- determinar la construcción de nuevos componentes reutilizables en las siguientes fases y determinar las características que deben presentar para satisfacer las especificaciones de un diseño

1.2.1 Componentes reutilizables de diseño

Una especificación de diseño describe la manera como se debe implementar las demandas operacionales (función, datos, desempeño) demandada por una especificación de requerimiento. Si la especificación de diseño se utilizará en el diseño de varios sistemas, podemos considerar a la especificación de diseño como *un componente reutilizable de diseño*. Como cualquier componente reutilizable un componente de diseño tiene asociados otros componentes, puede ser el diseño de un componente reutilizable de requerimientos, puede tener un componente de código que implemente el diseño y especificaciones de prueba que deben satisfacer las implementaciones.

El reutilizar componentes de diseño implica disminuir el esfuerzo en el diseño del sistema, estandarizar su construcción y disminuir los errores en el diseño. Adicionalmente, potencia el reuso de componentes en la fase de implementación. Se puede reutilizar un componente de diseño sin tener que utilizar su componente de código asociado, por ejemplo cuando se requiere el mismo diseño en una plataforma y lenguaje diferente. Lo inverso no se cumple, si reutilizamos un componente de código (sin modificarlo), debemos reutilizar también el componente de diseño asociado.

1.2.2 Sugerencias para especificar diseños reutilizables

Para tratar los diseños como componentes reutilizables, es conveniente construir la especificación siguiendo ciertas orientaciones que faciliten su reutilización.

Sugerencias para especificar diseños reutilizables

- emplear un modelo de componente para las especificaciones de diseño para uniformizar la estructuración, el formato y la descripción de los componentes de diseño, también debe contener o referenciar la documentación apropiada, componentes de código y componentes de pruebas del depósito. El modelo de componente debe facilitar la extracción e incorporación de componentes de diseño en otras especificaciones de diseño
- utilizar modelos para mapear algunos requerimientos a componentes de diseño específicos, de manera que aumente su potencial de reuso
- utilizar una arquitectura de software si se dispone como un producto de análisis, pues proporciona una división de capas y especificación de interfaces que facilita la incorporación de componentes de diseño existentes
- utilizar un diseño por capas, de manera que cada capa implemente una abstracción específica y pueda ser reemplazada fácilmente. Cada capa debe proporcionar un conjunto de servicios a la capa superior y tener interfaces bien definidas para las capas inmediatamente

superiores e inferiores

- especificar la interface de cada componente de diseño reutilizable. La interface determina las características comunes de interconexión que tendrá el componente en diferentes sistemas
- distinguir entre la interface de un componente y el diseño propiamente dicho. Para una interface puede haber varios diseños y la sola interface podría utilizarse como un componente reutilizable
- identificar las modificaciones potenciales que pueden afectar al componente y documentarlas, p.e. cambio de plataforma, interface de usuario o sistema operativo, nuevas demandas para el componente, nuevo manejo de errores, etc. Algunos de estos cambios se pueden prever con el uso de parametrización.

1.3 Reuso en la fase de programación (implementación)

La fase de programación parte de las especificaciones de diseño y construye implementaciones que ejecutarán las demandas operacionales del diseño. Los componentes de implementación abarcan código, pruebas (planes, procedimientos y resultados) y documentación. Desde la perspectiva de reuso esta fase debe:

- adecuar las demandas de diseño para maximizar la reutilización de componentes de código
- evaluar y construir los componentes reutilizables de programación que fueron sugeridos en las fases anteriores y construir casos de prueba para cada uno
- determinar qué componentes pueden llegar a tener demandas en otros sistemas y desarrollarlos o adquirirlos

1.3.1 Sugerencias para construir componentes de código reutilizables

Sugerencias para construir componentes de código reutilizables

- especificar una interfaz para el componente, es decir, separar las características y operaciones de utilización de los detalles de implementación. La interfaz es una base para determinar si el reuso es posible bajo determinadas condiciones
- mantener buena documentación de la interfaz y de sus implementaciones. Incluir características de desempeño, descripciones textuales, rangos de valores permitidos, restricciones y explicaciones de qué sucede bajo situaciones anormales (errores, excepciones)
- mantener las propiedades especificadas en el diseño mapeando los constructores de diseño al lenguaje de implementación. En el caso que el lenguaje no soporte algunos conceptos de diseño debe especificarse como se manejará cada situación
- cada componente debe ser lo más completo posible y minimizar las dependencias con otros componentes. Desde la perspectiva de tipos de datos abstractos, debe proporcionar como mínimo operaciones de creación, destrucción, consulta y modificación de estado
- minimizar las dependencias de la plataforma y sistema operativo

1.4 Certificación y calificación

Un componente reutilizable debe garantizar una cierta seguridad a quien lo reutiliza. Idealmente, debe satisfacer exactamente las funciones y desempeño que especifica, no tener efectos colaterales y no presentar errores (o por lo menos especificar cuándo se presentan y cuales son las consecuencias).

Como siempre el ideal es difícil de lograr, sin embargo, debemos definir un mínimo de requerimientos de manera que el reuso de componentes no agregue riesgos significativos al proyecto y que por el contrario reduzca las posibilidades de aparición de errores. Para ello disponemos de dos elementos fundamentales: *certificación* y *calificación*.

- **Certificación**

Es el proceso que determina en qué medida un componente reutilizable satisface los requerimientos básicos que impone la biblioteca. Un componente que no satisfaga todos los requerimientos también puede reutilizarse, en cuyo caso el proceso de certificación debe proporcionar la máxima cantidad de información posible para medir el riesgo en que se incurre. Los criterios de evaluación para el proceso de certificación pueden aplicarse a cualquier dominio y tipo de componente

- **Calificación**

Es el proceso de determinar si un componente satisface los requerimientos particulares de un dominio de aplicación en el que se tiene predefinida una cierta arquitectura que especifica interfaces de interconexión entre componentes requeridos. Nótese que en este caso los criterios de evaluación son propios del dominio de aplicación