TRABAJO DE GRADO

Desarrollo De Aplicaciones a Partir de Componentes Reusables.

Director:

-Prof. Lic. Gustavo Rossi.

Alumnos:

-Faiella Germán Darío.

-Duré Octavio César.

Contenidos:

Desarrollo de Aplicaciones a partir de Componentes Reusables.

ABSTRACT

El Reuso de Software es una de las alternativas que hoy se ofrecen como más atractivas para mejorar el proceso de desarrollo. Los sistemas actuales evolucionan a velocidades cada vez mayores porque los negocios actuales así lo requieren. Para poder seguir el ritmo de esta evolución es indispensable optimizar el proceso de desarrollo de soft en todas sus etapas. El negocio de hoy nos pide soluciones rápidas y capaces de evolucionar sin complicaciones. Para lograrlo es necesario un cambio tecnológico y cultural en lo que a desarrollo de soft se refiere. En este cambio, la tecnología de desarrollo orientada al reuso juega un papel muy importante, y hoy cuenta con conceptos, tecnologías paralelas, metodologías y herramientas que permiten implementarla en forma concreta. En este trabajo, tratamos de analizar cada uno de los aspectos involucrados en un proceso de desarrollo orientado al reuso, descubriendo las tecnologías que pueden realizar su aporte en cada una de las etapas de este proceso.

INTRODUCCION.

I. Objetivo.

El objetivo de este trabajo es el estudio de las características que un proceso de desarrollo de software orientado al reuso debería reunir. Cubriremos todo el ciclo de vida del soft, analizando las tecnologías que deberían utilizarse en cada etapa, teniendo en cuenta que lo que se busca es maximizar la eficiencia del proceso de desarrollo a partir del reuso.

Comenzaremos por analizar los aspectos involucrados en un traspaso a este tipo de tecnología, el costo y los beneficios que este cambio acarrea. En el capítulo II brindamos un informe sobre el estado actual de la ingeniería de soft en cuanto al Análisis de Dominios, ya que no es posible desarrollar soft genérico siguiendo las técnicas tradicionales de análisis de sistemas. En el capítulo III definimos las

características que debería reunir un **sistema de reuso**, indispensable en una organización que pretenda desarrollar software orientado a la reusabilidad. Luego brindamos algunos conceptos técnicos relacionados con el desarrollo concreto de código reusable. En el capítulo V cubrimos la etapa de composición de aplicaciones a partir de componentes reusables y en el VI estudiaremos algunas de las herramientas comerciales para composición.

Por último presentaremos una propuesta que trata de cubrir uno de los puntos en que hemos comprobado que el estado del arte actual en cuanto a tecnología de desarrollo de soft orientado al reuso carece de elementos, y que tiene que ver con el proceso de catálogo, recuperación y estudio de componentes reusables a partir de un repositorio que forme parte integral del sistema de reuso al que hacíamos referencia. Acompañaremos esta propuesta con un prototipo ejecutable que describimos en el capítulo VII.

II. El Paradigma de Orientación a Objetos.

A pesar de ser reconocida como una contribución crítica para el alivio de lo que se dio en llamar la ´Crisis del Software´, la tecnología de Orientación a Objetos no garantiza en absoluto que el soft desarrollado sea realmente reusable, o que pueda sobrevivir a cambios radicales en los requerimientos; y a pesar de que se obtienen beneficios innegables tales como una descomposición del código más limpia y manejable, nada asegura que un desarrollador de aplicaciones sea capaz de reutilizar una clase no trivial previamente desarrollada en una nueva aplicación.

La causa del poco aprovechamiento en cuanto a reusabilidad del código escrito, tanto dentro como fuera del paradigma de objetos, es que aún empleamos todo nuestro esfuerzo y creatividad en el desarrollo de aplicaciones individuales, en lugar de invertirlos en el desarrollo de componentes standard, interfaces y herramientas que transformarían la realización de una gran mayoría de las aplicaciones en la simple composición y customización de **componentes de soft** ya existentes.

El objetivo es desarrollar software reusable, para aplicaciones desconocidas, con requerimientos desconocidos. La tecnología de Orientación a Objetos brinda recursos tales como la encapsulación de estado y comportamiento, instanciación de objetos a través de clases, modificación incremental, reúso de código a través de herencia y mecanismos relacionados, que serán de gran utilidad en el camino hacia el objetivo de lograr ´**armar**´ aplicaciones en lugar de programarlas.

Lograr un equipo capacitado en el desarrollo de componentes reusables es un gran adelanto, pero no es suficiente. Será necesario un modelo dentro del cual estas componentes puedan ser ´conectadas´ entre sí para conformar aplicaciones individuales, o lo que es más, nuevas componentes de mayor complejidad. Este modelo, sumado a un sistema de información que permita el catálogo, recuperación

y estudio de componentes proveerá la base sobre la cual desarrollar aplicaciones a partir de módulos prefabricados.

Actualmente existen varios productos comerciales y propuestas académicas de herramientas visuales para composición, con interfaces de usuario muy atractivas para la manipulación directa de componentes en el desarrollo de aplicaciones. Sin embargo, carecen de todo tipo de sistematización para el acceso a componentes almacenadas. Un ambiente que soporte una tecnología de desarrollo orientado al reuso debe contar con las herramientas necesarias para la organización, manejo, almacenamiento y acceso a componentes. Y una vez que la componente deseada es obtenida, debe permitir su estudio y adecuación a los requerimientos.

III. Traspaso a la Tecnología de Orientación a Componentes.

El cambio hacia una tecnología orientada a la producción de componentes de software involucra aspectos económicos, operacionales, técnicos y hasta culturales. Nos detendremos a analizar algunos de ellos.

Supongamos que llega a nuestras manos un proyecto de desarrollo de una aplicación para un cliente; es de esperar que tenga una fecha tope de entrega. El lapso de tiempo asignado al proyecto, en la mayoría de los casos será considerado escaso por los desarrolladores y excesivo por el cliente. ¿Cómo tomar la decisión de asignar recursos al desarrollo de componentes genéricas cuando el tiempo y presupuesto asignados al proyecto aparentan ser insuficientes? Este es el paso inicial que se debe tomar, y el más crítico en el traspaso de un modelo de producción orientado a aplicaciones a un modelo orientado a componentes. La forma en que se adopte esta medida dependerá mucho de las características del grupo de desarrollo y de los recursos con que cuente. Hoy en día, pocos son los que se animan a dar este primer paso, y, en cambio, nos abocamos a la tarea de desarrollar la aplicación en el tiempo estipulado y cumpliendo con los requerimientos mínimos establecidos. En general, y si no ocurren eventos inesperados, la relación esfuerzo-ganancia se aproximará bastante a la planificada en el momento de gestación del proyecto. Obviamente, el próximo desarrollo, nos enfrentará a una situación muy similar, y ante la cual seguramente reaccionaremos de la misma forma.

La otra alternativa nos plantea la posibilidad de realizar una inversión de tiempo y recursos para que, a medida que se vayan presentando sucesivos proyectos, nos encuentren en mejor situación.

Esta inversión no siempre puede concretarse a través de la creación de un grupo especializado dedicado exclusivamente al desarrollo de componentes reusables, lo cual constituiría una situación ideal: un equipo capacitado en la realización, distribución y publicidad de componentes de soft que se encuentre a disposición del resto de los desarrolladores como proveedores de componentes ´a

pedido´ y administradores de un repositorio de módulos reusables. En circunstancias como esta, a medida que crezcan las bibliotecas de componentes, los desarrollos individuales requerirán de menor cantidad de recursos, pudiéndose asignar mayor personal y presupuesto al grupo encargado de generar componentes reusables.

En general, razones económicas y presupuestarias no hacen sencilla la creación de un grupo como el descripto anteriormente, sin embargo, existen posibilidades alternativas: Desarrollar un porcentaje del sistema a partir de componentes reusables generadas especialmente, comprar componentes relacionadas con el tipo de aplicaciones que se desarrollan generalmente, dedicar tiempos muertos entre proyecto y proyecto al desarrollo de componentes, etc. Aunque más lentamente, la tendencia a mejorar los tiempos de desarrollo de aplicaciones individuales descripta antes debería repetirse en estos casos, permitiendo dedicar mayores recursos a la producción orientada a componentes, que nos posicionará en condiciones ventajosas frente a futuros proyectos.

En cuanto a los aspectos culturales a los que nos referíamos antes, existe mucho por cambiar si es que pretendemos desarrollar software desde una perspectiva orientada al reúso. Es sorprendente hallar en oficinas supuestamente ´modernas´ un concepto muy antiguo de `propiedad` personal del código. Solemos encontrar que ´compartir´ el soft desarrollado cuesta mucho; aún en grupos jóvenes, que supuestamente se han formado entre ideas como las que discutimos en este trabajo. Revertir situaciones como éstas es trabajoso y requiere involucrarse con factores humanos y educacionales.

IV. Aspectos Técnicos.

La aplicación de técnicas de reusabilidad depende del previo almacenamiento de un conjunto de descripciones de problemas, soluciones y métodos para derivar soluciones. La existencia de tales recursos reusables no puede ser considerada un hecho. En la práctica, quienes han intentado probar los beneficios tan publicitados por las técnicas de reusabilidad, descubrieron que no es trivial identificar la clase de información reusable ´justa´ de la cual proveer a los desarrolladores. Lo que es más, una vez identificada tal información; existen cuestiones de granularidad, representación, índices, acceso a repositorios y demás, que deben ser resueltos para hacer que la información sea realmente reusable. En otras palabras, antes de que una organización pueda empezar a reutilizar, debe montarse en ella toda una ´infraestructura para el reuso´ , lo cual, como anunciáramos en párrafos anteriores, involucra distintos aspectos, es difícil y costoso.

El desarrollo de soft genérico implica un cambio de filosofía desde el mismo análisis. No es posible escribir componentes reusables si es que no se ha hecho un diseño genérico, basado en un análisis de requerimientos amplio y no sujeto a

un caso de estudio particular. Las estrategias de análisis de requerimientos, especificación y diseño Top-Down difícilmente nos conducirán a la implementación de componentes reusables. Este problema es esencialmente metodológico y es puntualmente atacado por distintos estudios acerca de **Análisis de Dominio**.

CAPITULO I: Costos y Beneficios del Traspaso a una Metodología Orientada al Reuso.

Introducción.

Planteado el desafío de lograr reusabilidad, se hace necesario medir en forma precisa sus costos y beneficios. Sin evidencia acerca de las ventajas del reuso, pocas organizaciones invertirán los recursos necesarios para afrontar los obstáculos involucrados. En este capítulo consideraremos los puntos y características fundamentales que deberán tenerse en cuenta en la medición del reuso, desde el punto de vista del B.S.I. (Beneficio Sobre Inversión) para componentes reusables, y examinaremos la relación clave existente entre reusabilidad y facilidad en el mantenimiento del soft.

2.1 Mediciones acerca del Reuso en el Paradigma de Orientación a Objetos.

La capacidad de reuso es una característica muy valorada dentro de la ingeniería de soft. Desafortunadamente, la evidencia cuantitativa del reuso en desarrollos orientados a objetos es limitada. La información disponible es a menudo obtenida a través de métodos de medición informales y focalizada sobre distintos conceptos del reuso. Ciertos proyectos reportan un alto grado de reuso, pero incluyen en sus mediciones, la utilización de componentes incluídas en el ambiente de desarrollo o librerías de clases de terceras partes. En otros casos, estas componentes primarias son omitidas, y sólo se consideran como reusables las componentes construídas a partir del dominio del problema.

Otro punto a considerar, es que el uso de herencia es en ciertos casos considerado como reuso del código heredado. Consideramos que el reuso está presente sólo en la utilización de componentes en distintas aplicaciones. La herencia en una única aplicación no puede ser tratada como reuso de ninguna manera, al menos para los fines de medición que estamos estudiando. Estas diferencias deben ser tenidas en cuenta en la comparación de los pocos resultados disponibles.

Muchas organizaciones han visto decepcionadas sus intenciones de trasladarse hacia una tecnología de desarrollo orientada al reuso a partir de estos inconvenientes. Sin embargo, el reuso permanece como uno de los puntos más atractivos en el momento de tratar de mejorar el proceso de desarrollo de software. Por ejemplo, el Departamento de Defensa de los Estados Unidos (DoD) ha estimado que un incremento del reuso de soft de tan solo un 1% repercutiría en un

ahorro de u$s 300.000. El DoD consideró el efecto del reuso en aplicaciones mayormente convencionales. Un dato interesante es el que nos provee la industria japonesa de software, que logró tasas impresionantes de reusabilidad, excediendo el 50%, con un incremento en la productividad del 20% anual.

Las aplicaciones desarrolladas dentro de la tecnología de orientación a objetos proveerán bases aún más sólidas para dedicar recursos a la construcción de componentes reusables. Los tipos de objetos basados en el dominio de problema del mundo real, y su implementación como clases encapsuladas proveen una estructura altamente modular y flexible para la construcción de software que permite mejorar el proceso de desarrollo, reducir los tiempos de implementación y optimizar la distribución de aplicaciones.

2.2 Fundamentos.

Reuso es la capacidad de construir tanto modelos como aplicaciones a partir de componentes. Las 'unidades' para el reuso las constituyen cualquier elemento bien definido del problema en el mundo real o su realización en software. Una componente puede ser una especificación (un modelo o diseño) compuesto de tipos de objetos, o la implementación (código ejecutable) en clases.

Una propiedad clave del reuso es que involucra la utilización de componentes para satisfacer requerimientos que no eran conocidos en el momento en que la componente fue desarrollada. Esta cualidad define al reuso como una forma de captar posibles nuevos requerimientos. Lo que es más, los requerimientos deberían aparecer en un contexto separado dentro de la aplicación. Esto es lo que distingue al reuso del ciclo de desarrollo de una aplicación standard. Esta distinción es útil por varias razones. Primero, la naturaleza iterativa del proceso de desarrollo implica que el entendimiento de requerimientos puede producirse en cualquier punto del desarrollo. Esto hace difícil separar un entendimiento evolutivo de la aplicación y obtener ventaja a partir de una componente existente. Segundo; el proceso de buscar, entender y reusar algunas componentes es muy diferente a las tareas típicas realizadas durante un desarrollo tradicional, en el que el desarrollador tiene usualmente un íntimo conocimiento de los requerimientos subyacentes y de la implementación de cada módulo. El proceso de desarrollo es de construcción más incremental cuando se utilizan partes bien conocidas (tal vez esto sea simplemente **uso**). El reuso afecta el ciclo de desarrollo tanto en el actual como en futuros proyectos.

2.3 Beneficios sobre Inversión.

El reuso debe ser evaluado en el contexto de múltiples equipos de desarrollo y organizaciones completas dedicadas a la construcción de software, no

sólo como la actividad de un programador individual. Cualquier medición de reuso debe apuntar a su "macro-proceso de desarrollo", el cual incluye el trabajo de definir y construir componentes reusables. El trabajo de encontrar, entender e implementar extensiones propias es responsabilidad del usuario. Además en una organización que desarrolla soft utilizando una metodología orientada al reuso, se hará necesario manejar el almacenamiento y mantenimiento de componentes. Por supuesto, este costo será compensado con beneficios para el usuario y la organización. Los beneficios incluirán el ahorro involucrado en no tener que redesarrollar funcionalidades similares en varias aplicaciones, y la utilización de componentes provistas por terceros.

Es este equilibrio entre costos y beneficios entre creadores de componentes, usuarios y la organización el que determinará si el reuso vale la pena. En un análisis excelente, Henderson-Sellers presenta un modelo para el B.S.I. del reuso. En él plantea un argumento cuantitativo para medir el reuso en función del costo de construir, entender y recuperar componentes reusables. Se asigna una variable a cada costo agregado en las tareas involucradas en el proceso de desarrollo con reuso:

$$R = \frac{S - (CR + CM + CD + CG)}{CG} = \frac{S - C}{CG} - 1$$

Los términos se definen como R = Beneficio sobre inversión; S = Costo del proyecto sin reuso; CR = Costo de encontrar las componentes reusables; CM = Costo de modificar las componentes; CD = Costo de desarrollar nuevas componentes; CG = Costo de generalización. Cada variable representa el costo total asociado con cada tarea en el desarrollo de una aplicación con cierto número de componentes. C es el costo total del reuso para el usuario de componentes (C = CR + CM + CD + CG). El costo de generalización (CG) representa el esfuerzo involucrado en construir componentes reusables; es un costo para el creador de componentes. Henderson-Sellers sólo considera el reuso de clases como componentes y asume que el esfuerzo de construir clases reusables radica en su generalización. Por esto, CG representa el trabajo de construir una jerarquía de clases con una o más superclases y relaciones de herencia.

La ecuación expresa que B.S.I. es positivo si el ahorro por reuso (S-C) es mayor que el costo de generalización. El análisis Henderson-Sellers provee además los resultados de utilizar esta ecuación en una simulación del B.S.I. para un número de proyectos y diferentes valores para las variables de costo. El efecto de una base de componentes reusables existentes también fue considerado. El resultado clave de la simulación se produce cuando el B.S.I. se hace positivo. Un número interesante es la cantidad de proyectos necesarios para justificar el costo (B.S.I. > 0). Las mediciones realizadas en este trabajo, indican que hasta la asignación de los más elevados valores a CG promete resultados optimistas : entre

3 y 10 proyectos para llegar a un valor positivo para S.

Las simulaciones mostraron que el costo de desarrollo y la disponibilidad de una librería de clases afectan fuertemente el B.S.I.

Pero el proceso de desarrollar componentes reusables no se limita a la generalización de código, sino que involucra todos los aspectos de análisis y diseño de manera que la componente final describa una abstracción convincente. El uso de otras técnicas tales como la composición y encapsulación requieren esfuerzo adicional.

Una nueva simulación, con los mismos valores para CG y CR, pero asumiendo que sólo un 10% de las componentes necesarias en cada proyecto puede ser encontrado en las librerías disponibles, nos muestra que son necesarios 33 proyectos para alcanzar un B.S.I. positivo. Este resultado enfatiza la importancia de utilizar una biblioteca básica de componentes reusables.

El análisis de este trabajo es realmente sorprendente. El reuso es una inversión a largo plazo, pero pocas organizaciones aceptarían el costo inicial de tal período de tiempo: 33 proyectos es una enorme cantidad de recursos invertidos. En los siguientes párrafos discutiremos cómo los costos de mantenimiento equilibran este aparente problema con el costo del reuso.

2.3.1 Mantenimiento y Reusabilidad.

El costo de mantenimiento es extremadamente alto. Una organización dedicada a Sistemas de Información puede ocupar hasta un 80% del total de sus recursos en mantenimiento, usualmente dividido en tres actividades diferentes: a) Corrección de errores en los desarrollos actuales, b) Implementación de nueva funcionalidad requerida por el negocio relacionado con los sistemas ya desarrollados y c) Modificaciones causadas por cambios en los requerimientos. El porcentaje del mantenimiento involucrado en cada actividad es difícil de determinar. Los sistemas nuevos tienden a tener más errores al entrar en producción y comenzar a ser testeados por los usuarios finales. Los sistemas en producción a menudo necesitan proveer mayor funcionalidad, ya sea debido a un análisis inadecuado o a la escasa participación de usuarios y otros expertos en el dominio durante esta etapa. Por último, los sistemas más maduros están sujetos a cambios en los requerimientos que se producen a medida que su entorno evoluciona.

Es cierto, entonces, que todo desarrollo involucra un costo posterior de mantenimiento. En uno de sus trabajos, G.P.W. Keen trata la magnitud de este costo. En él estima que cada dólar gastado en un nuevo desarrollo acarreará un gasto de $0.60 por año en concepto de mantenimiento durante el ciclo de vida de la aplicación. Aproximadamente $0.20 los constituyen costos operacionales y los restantes $0.40 actividades de mantenimiento. Keen sugiere además que estas actividades de mantenimiento se suman a la aplicación de manera tal que el

mantenimiento mismo acarreará un costo más alto de mantenimiento posterior. Esto se debe a que la extensión funcional de una aplicación es considerada una actividad de mantenimiento que además incrementa la complejidad de las aplicaciones.

Reuso y mantenimiento están íntimamente relacionados: desarrollar para reusar es facilitar el mantenimiento. Una componente reusable debe ser un módulo bien diseñado. La calidad del diseño de un módulo es generalmente función de la cohesión, encapsulamiento y acoplamiento débil. Y son precisamente estas características las que hacen a una componente orientada a objetos o a cualquier módulo más extensible y mantenible.

Una componente es cohesiva si está basada en un concepto del dominio del problema, por ejemplo, si la componente representa una abstracción que tiene un significado de importancia para el experto en el dominio. En un análisis detallado de ocho proyectos de desarrollo de aplicaciones medianas, D.N. Card establece una fuerte correlación entre la cohesión de módulos y una baja tasa de errores. Un total de 453 módulos fueron caracterizados por sus desarrolladores como de alta, media o baja cohesión. Para módulos altamente independientes, el 50% registraron cero fallas. En cambio, para módulos poco independientes, sólo el 18% registraron cero fallas en tanto el 44% registro más de 7 fallas por módulo. Dado que una parte significante del mantenimiento es la eliminación de fallas, es razonable asumir que las componentes de mayor cohesión requerirán menor mantenimiento.

La encapsulación es la separación de la interface de una componente de su implementación. Es la encapsulación de una componente lo que reduce el esfuerzo en el mantenimiento de una aplicación. Las aplicaciones desarrolladas dentro del paradigma de objetos tienen un fuerte soporte para el encapsulamiento, por lo que deberían ser más mantenibles que las desarrolladas en el paradigma procedural.

Por último, W. C. Lim acerca evidencia adicional sobre la relación entre reuso y mantenimiento del soft.. En su paper cita la densidad de defectos (promedio de defectos por cada 1000 sentencias no comentadas de código fuente) para cierto número de proyectos en Hewlett-Packard. Tomemos 2 trabajos a modo de ejemplo:

1) Defectos en el código reusado: 0.4
   Defectos en el código nuevo : 4.1
   Defectos en el producto : 1.0
   Nivel de reuso: 46 %

2) Defectos en el código reusado: 0.4
   Defectos en el código nuevo : 1.7
   Defectos en el producto : 1.3
   Nivel de reuso: 30 %

Nuevamente, es razonable asumir que el reuso de código, con la consiguiente baja en la tasa de errores, facilitará el mantenimiento.

2.3.2 Efectos sobre el B.S.I.

Dado el hecho de que la utilización del reuso afecta la etapa de mantenimiento del software, reduciendo su costo, es necesario capturar esta variable en la medida del B.S.I. La siguiente ecuación muestra una extensión de la fórmula de Henderson-Sellers:

$$R = \frac{S - C + M}{CG} - 1$$

En la fórmula revisada, M es la reducción del costo de mantenimiento sobre el ciclo de vida de la componente. De esta manera, obtendremos un B.S.I. positivo si el costo de reutilizar y mantener una componente es mayor que el costo de programarla. En este punto, sólo podemos dar un argumento cualitativo del ahorro en costo de mantenimiento mediante la utilización de técnicas de reuso. Sin evidencia empírica las diferencias entre las distintas pruebas realizadas son sólo estimativas. Sin embargo, la reducción en la tasa de errores observada en módulos de alta cohesión, la baja densidad de defectos hallada en el código reusado y la mejora en cuanto a facilidad de mantenimiento del código orientado a objetos sugieren una fuerte correlación entre el uso de componentes orientadas a objetos y el costo de mantenimiento de las aplicaciones que las utilizan.

En resumen, el costo inicial de un desarrollo orientado al reuso es mitigado por la reducción en el costo de mantenimiento. En otras palabras, desarrollar para reusar asegura que las componentes sean realmente más **usables.** Este incremento en la calidad tiene un efecto dramático sobre los costos de mantenimiento. Este punto debería afectar cualquier consideración acerca del B.S.I. del reuso. El B.S.I se incrementa no sólo debido al ahorro en construir nuevas componentes en los casos en que estas se encuentren disponibles, sino también por la mencionada reducción en cuanto a costos de mantenimiento.

A pesar de la posible variación en el costo de construir componentes reusables, hasta diez veces más esfuerzo en costo de desarrollo es aceptable dado el orden de magnitud aproximado en ahorro en costo de mantenimiento.

La implicación clave es que existe en este punto una fuerte motivación en construir soft de calidad, sea reusable o no. Si se logra el reuso, se obtienen aún mayores beneficios.

2.4 Próximos pasos en cuanto a Medición de la Reusabilidad.

Actualmente, el Grupo de Métrica de Virginia se encuentra trabajando en

esta nueva area de la ingeniería de software: Programación Orientada a Objetos y Reusabilidad del Soft. El paradigma de Orientación a Objetos ha demostrado soportar y promover el reuso. Se encuentran disponibles librerías (bases de datos) orientadas a objetos (GNU, Microsoft, Borland International, Smalltalk, etc.). Sin embargo, como mencionáramos en párrafos anteriores, existen muy pocas métricas que intenten evaluar el reuso potencial de una componente codificada (ya sea orientada a objetos o no). El Grupo está investigando qué hace al software reusable y cómo puede ser medida esta característica a través de una herramienta desarrollada en el Departamento de Ciencias de la Computación (Blacksburg, Virginia 24061. Fax: 703-231-6075. lattanzi at cs.vt.edu) : The Metrics Generator.

Se han establecido dos objetivos. El primero es medir la reusabilidad relativa entre dos componentes. Utilizando esta herramienta se pretende evaluar distintas piezas de software de acuerdo a su potencial reusabilidad. Para lograr este objetivo, se está realizando un estudio empírico en una Universidad de nivel Senior en Ciencias de la Computación. El paradigma de Orientación a Objetos, incluyendo Análisis y Diseño Orientado a Objetos, serán enseñados como materias del experimento. Los trabajos que resulten serán utilizados como mediciones que colaboren en el desarrollo de Metrics Generator.

El segundo objetivo es lograr predecir la complejidad y reusabilidad del código en la etapa de diseño. Claramente, para lograr esto será necesario obtener algún resultado en el objetivo anterior. La idea es poder volcar la documentación de análisis en alguna forma legible por la máquina, de manera que pueda ser analizada. Para este punto se utilizará al mismo grupo de programadores, a quienes se les pedirán documentos de análisis y diseño que se puedan vincular con el código resultante de forma tal de poder establecer ciertos parámetros en términos de complejidad y reusabilidad.

CAPITULO II : Aspectos del Análisis de Dominios.

1.1 Análisis de Dominio.

El Análisis de Dominio apunta a resolver cuestiones tales como la forma de identificar, capturar y documentar información reusable dentro de áreas de problemas particulares (el dominio del problema). Para hacerlo, se funden viejas y nuevas técnicas de análisis que cada vez más ingenieros de software interesados en una perspectiva de reusabilidad del desarrollo de soft adoptan como respuesta al problema de cómo, en una forma sistemática y confiable, lograr capturar la información que necesitamos ahora (y podemos necesitar en el futuro) en nuestro proceso de desarrollo de soft.

El trabajo sobre Análisis de Dominio da por hechos los siguientes dos puntos:

- Una clara especificación del dominio de la información reusable. La reusabilidad no es una propiedad universal, depende de un problema en particular y de un contexto de resolución. Así, una componente de información puede considerarse reusable dentro de un área de problemas bien definida.
- Una relativa estabilidad en los dominios de problema. El conocimiento acerca de soluciones y los métodos de hallazgo de soluciones dentro de estos dominios debe ser lo suficientemente estable como para justificar el esfuerzo de adquirirlo y representarlo. Este punto es el resultado de la observación empírica, una especificación e implementación de conocimiento modular hacen posible capturar el conocimiento necesario para resolver un gran número de problemas a partir de un conjunto relativamente pequeño de descripciones reusables. La estabilidad, hace posible amortizar el costo de capturar y representar la información reutilizándola a través de cierto período de tiempo o en distintos sitios sin que se transforme en obsoleta por cambios bruscos en el dominio del problema.

El análisis de dominio puede ser explicado por analogía con la ingeniería de software tradicional como el contexto en el cual, una organización, en lugar de dedicarse al desarrollo de sistemas de soft (aplicaciones), desarrolla un metasistema: el sistema de reuso. El propósito del sistema de reuso es, sí, la implementación de aplicaciones.

La infraestructura de reuso es la `base de datos` de ítems a ser reutilizados, por lo que será necesario definir para estos ítems la clase de especificaciones que el sistema de reuso espera. El análisis de dominio deberá identificar los conceptos de especificación e implementación requeridos. Estos conceptos deben ser representados en forma precisa; llamaremos **modelo de dominio** a esta representación. Dependiendo de la tecnología utilizada en el desarrollo de software, el modelo de dominio puede no ser directamente reusable en la

construcción de soft, pero provee las bases para el diseño de repositorios de items reusables -la infraestructura de reuso- consistente en módulos de soft, diseño de componentes, etc.

La ingeniería de software tradicional, nos enseña que, antes de implementar un sistema para resolver un problema, es deseable 1) Analizar el problema y describir los requerimientos de la solución de software, 2) Especificar qué es lo que el sistema debería hacer y 3) Diseñar el sistema de manera de facilitar su evolución futura.

El análisis de dominio es equivalente al analisis de requerimientos y especificación convencionales, pero se ubica en un metanivel con respecto a la construcción de software. El Análisis de Dominio trabaja con la identificación, adquisición, y representación de (potencialmente reusables) conocimientos acerca de la especificación e implementación de software para distintos tipos de problemas del mundo real. Es decir, no trabaja con estos problemas en forma directa, sino más bien, a través de los conocimientos que existen acerca de ellos desde el punto de vista de las soluciones informáticas.

Una organización decidida a establecer un sistema de producción de software basado en el reuso deberá implementar una `infraestructura para el reuso` que le permita realmente generar implementaciones bajo restricciones explícitas de recursos, tal como lo comentábamos en párrafos anteriores.

Las técnicas de Análisis de Dominio ayudan en la definición de esta infraestructura que permitirá establecer un sistema de reuso capaz de lograr y mantener el nivel de performace deseado.

1.2 Utilizando Técnicas existentes, pero en un nuevo Contexto de Resolución de Problemas.

Hemos presentado al Análisis de Dominio como una version (en un metanivel) del análisis de requerimientos convencional. No es nada diferente a lo que ya conocemos; sin embargo, sus objetivos son distintos debido al enfoque que adopta frente al proceso de construcción de software. Lo que es nuevo acerca del Análisis de Dominio y lo que lo transforma en un área de estudio son las demandas que el nuevo contexto de resolución de problemas realiza frente a técnicas y principios de análisis ya conocidos. Muchas de las estrategias y técnicas aplicadas en el Análisis de Dominio no son más que transferencias o adaptaciones de técnicas que encontremos en otros tipos de ingenierías del conocimiento, modelación conceptual, o análisis de requerimientos tradicional. Lo mismo se puede decir acerca de la representación de dichas técnicas.

1.3 Conceptos involucrados en el Análisis de Dominios.

Al comenzar el capítulo hablabamos de las características que debería reunir un área de problema para ser objeto de un analisis de dominio: una especificación y definición claras del area y sus límites, y cierta estabilidad en la información que permita el aprovechamiento del resultado del análisis en cierta extensión geográfica o durante cierto lapso de tiempo de modo tal que permita amortizar el esfuerzo involucrado en adquirir y documentar su conocimiento.

Digamos que un conjunto de información será considerado un **Dominio de Problema** si: 1) Existe una profunda relación entre los items de información con respecto a cierta clase de problemas, 2) Existe una comunidad interesada en resolver esta clase de problemas, 3) Esta comunidad busca soluciones informáticas para estos problemas, 4) Esta comunidad tiene acceso al conocimiento que puede ser aplicado en la resolución de esta clase de problemas.

Un ejemplo de dominio de problema es el dominio del procesamiento de texto, y aunque distintas comunidades interesadas (editores, abogados, publicistas, estudiantes) tendrán una visión distinta del dominio, existe un conjunto de reglas, un vocabulario y espectativas acerca de cómo las distintas clases de documentos deberían ser procesados e impresos.

El objeto del Análisis de Dominio es producir un **Modelo del Dominio de Problema**. Asumiendo que estamos interesados en la construcción y evolución de sistemas de soft, un modelo de un dominio debería contener información de al menos tres aspectos del dominio de problema: a) Conceptos que permitan la especificación de sistemas en el dominio, b) Diagramas describiendo cómo mapear esa especificación en código y c) Las relaciones de los conceptos de la especificación entre sí, y con los diagramas de implementación. Entonces, podemos distinguir dos fases en el Análisis de Dominio: el **análisis conceptual** y **el análisis constructivo.** Es necesaria una representación explícita de la información para convertir el análisis y su evolución en algo concreto: el Modelo de Dominio, que deberá servir como fuente definitiva y unificada de referencia cuando se encuentren ambigüedades en el análisis de un problema o más tarde durante la implementación de componentes reusables; como repositorio de conocimientos para enseñanza y comunicación; y como especificación para el programador de componentes.

En la práctica, encontraremos que el volumen y la complejidad del conocimiento involucrado en un dominio aparentemente simple es grande; y aunque no existe niguna técnica de medición confiable para estos parámetros, veremos que dominios del mundo real, que inicialmente se muestran como `pequeños` se revelan como muy complejos cuando los colocamos bajo análisis.

En el dominio de los Manejadores de Discos, por ejemplo, escribir y leer sobre un disco magnético es un problema bien definido. Es cierto que un programa que maneje uno de estos dispositivos no debe ser muy grande ni complicado, y que existen unas pocas operaciones simples que debería manejar. Sin embargo, las posibles variaciones en cuanto a tecnologías son numerosas, y si analizamos algún sistema operativo encontraremos que soporta alrededor de unos cien

dispositivos diferentes. Cuando analizamos el impacto de tal variación en la forma en que la funcionalidad del manejador debe ser implementada, el volumen de información que el analista debe adquirir y organizar se vuelve inmenso. Es así que se harán necesarias herramientas de representación que ayuden al analista a clasificar, jerarquizar, especificar referencias cruzadas y obtener información almacenada previamente.

El tratar de manejar un conjunto de información interrelacionada es un problema simple, si lo comparamos con el desafío intelectual involucrado en el 'entendimiento' de un dominio. Los dominios utilizados como ejemplo, tales como el de los procesadores de texto o el de los controladores de ascensores o discos, ilustran sólo algunos de los aspectos a los que se enfrentan los analistas de dominios. No cubren las dificultades prácticas e intelectuales del análisis de dominios del mundo real. En el ejemplo de los controladores de ascensores, cuando temas de electrónica, mecánica, arquitectura de construcción y uso, interface con la administración del edificio y sistemas de seguridad, regulaciones de seguridad impuestas por el Estado, etc. deben ser tenidos en cuenta, la especificación de un controlador de ascensores pasa a depender de un centenar de parámetros diferentes. Sólo el manejar con fluidez el vocabulario del dominio llevará al analista meses de interacción con expertos en diferentes disciplinas. El resultado de tal análisis no es solo un modelo del dominio de los controladores de ascensores, sino en realidad un conjunto de modelos interrelacionados, incluyendo, por ejemplo, conceptos de sistemas operativos distribuídos, sensores para adquisición de datos, procesamiento en tiempo real, modelos matemáticos de tráfico. etc.

"Para sistemas tan complicados como los Sistemas Manejadores de Bases de Datos, el proceso de adquisición de conocimientos y diseño y verificación de la arquitectura en módulos podría fácilmente requerir varios años del trabajo de un experto. No parece haber soluciones fáciles y rápidas" (Batory D.)

1.4 Infraestructura para el reuso.

Para poder operar un configurador de módulos de software, que se alimente a partir de modelos de dominio, necesitaremos de una infraestructura consistente de:

- Bibliotecas de Componentes de Soft.
- Un sistema de indexación que permita acceder dichas componentes.
- Descriptores de componentes escritos en algún lenguaje.
- Sentencias de interconección de módulos capaces de describir qué módulos deben interconectarse y cómo.

Los propósitos de un modelo de dominio y de la infraestructura de reuso son, en principio, distintos. El modelo de dominio debería soportar el trabajo del analista de dominios; identificación de conocimiento, adquisición y análisis. La

infraestructura debería soportar una eficiente operación del sistema de reuso, que es el que realmente produce implementaciones. Por esto, debe adaptarse a las demandas de la organización y responder a la pregunta de Cómo la implementación del conocimiento debe ser empaquetada y almacenada para maximizar la eficiencia en el desarrollo de software. La infraestructura deberá además adaptarse a la tecnología del sistema de reuso. Las demandas de la organización y de la tecnología del sistema de reuso quedan fuera del foco del análisis de dominio; sin embargo ambos son complementarios en el proceso de convertir el conocimiento de una comunidad en repositorios de componentes reusables.

1.5 Principios y Aspectos Metodológicos del Análisis de Dominios.

Dada una clase de problemas para los cuales buscamos soluciones computacionales, tratamos de hallar un lenguaje único para describir dichos problemas e identificar soluciones generales válidas o métodos de hallazgo de soluciones. Esta situación puede ser comparada con la del científico que examina un fenómeno físico tratando de encontrar un conjunto de leyes generales o un modelo computacional. La forma en que estos científicos desarrollan sus teorías no es bien conocida. Es un proceso laborioso, que involucra observación, analogía, generación de hipótesis, testeo e inducción. En la práctica no existe una respuesta definitiva a la pregunta de Cuáles son los conceptos correctos para describir problemas y soluciones en un dominio determinado. Cada analista desarrolla su propio framework conceptual frente a un problema; y se maneja dentro del mismo de acuerdo a su capacidad.

En un extremo del espectro metodológico del Análisis de Dominios se encuentra la tendencia "crack analyst", en la cual el experto es el analista. Muchos han adoptado esta posición basándose en el hecho de que un analista de dominios exitoso es un experto en el dominio. Lo que es más, la práctica ha demostrado que un experto muchas veces no es capaz de articular sus conocimientos como un conjunto bien definido de conceptos para la especificación y reglas para la implementación. De hecho, durante el análisis el experto es forzado a reexaminar las bases de su conocimiento y el proceso a menudo se transforma en una experiencia de reaprendizaje.

En el otro extremo, tenemos la tendencia de la ingeniería del conocimiento. Asume que el conocimiento que necesitamos existe "en algún lado". Y según esta tendencia, los pasos claves son: a) Identificar las fuentes correctas de conocimiento, b) Adquirir el conocimiento existente y c) Representarlo explícitamente.

La experiencia obtenida en el area de Sistemas Expertos revela que la ejecución de estos pasos no es trivial. La tendencia de la ingeniería del conocimiento, se basa en expertos humanos, análisis de documentos o en alguna

combinación de ambos.

Como método, la ingeniería inversa, que utiliza el análisis de aplicaciones existentes, está obteniendo una considerable atención: Una familia representativa de aplicaciones en un determinado dominio es seleccionada y estudiada con el objeto de identificar puntos comunes e inducir abstracciones genéricas.

En la práctica, el análisis puede basarse exclusivamente en aplicaciones documentadas sólo cuando se trata de dominios bien conocidos, donde la teoría obtenida de la literatura técnica puede guiar al analista. En otros casos la consulta y asesoramiento de expertos humanos es esencial. El análisis de aplicaciones existentes aparece como una buena forma de organizar la tarea de adquisición de conocimientos a partir de expertos.

1.5.1 Evolución de Dominios y Sistemas de Reuso.

En el mundo real, los dominios no son absolutamente estables. Sin embargo, en la mayoría de los casos, los cambios tienden a ser graduales, y monótonos para una gran extensión del dominio. Es decir, los cambios involucran extensiones y variaciones que son consistentes con el conocimiento existente. Esta relativa estabilidad hace ventajosa una alternativa de reuso específica al dominio.

En otros casos, los cambios son no monótonos. La semántica de los conceptos cambia, las asumsiones son revisadas, o los planes de implementación son sustituídos o modificados para satisfacer las necesidades de las nuevas tecnologías. El dominio de problema cambia porque el mundo real cambia, porque las tecnologías de implementación cambia y porque nuestro entendimiento del problema y las soluciones evolucionan con el tiempo.

Una infraestructura debe evolucionar, aunque el dominio de problema original no lo haga. Por ejemplo, las componentes pueden tener que ser reimplementadas para eliminar bugs; o la arquitectura de repositorios y distribución de componentes en repositorios puede necesitar evolucionar para mejorar la eficiencia del sistema de reuso.

Un sistema de reuso se encuentra embebido en una organización dedicada al desarrollo de software. Su performance se define en relación a los objetivos de la organización. La eficiencia del sistema puede ser medida en relación a la cantidad de recursos necesarios en una implementación promedio, utilizando y sin utilizar el sistema de reuso.

Un sistema de reuso, en la práctica, experimenta una gradual pero contínua evolución. Las aplicaciones en dominios del mundo real necesitan de cambios de organización a lo largo del tiempo. La performance del sistema de reuso decae como consecuencia de estos cambios. Sin la correspondiente evolución de la infraestructura de reuso, la escalabilidad económica que justifica el reuso no puede ser lograda. Por esto, un sistema de reuso real, debe proveer de un loop de feedback desde el subsistema operacional al subsistema infraestructural. En la

mayoría de los sistemas de reuso hoy, este loop de feedback no es soportado, y algunas veces ni siquiera es conocido.

Los métodos de la ingeniería de dominio organizan la evolución de los repositorios del sistema de reuso de manera de mantener el nivel deseado de performance del sistema. Estos métodos incluyen un proceso de aprendizaje que puede incluir la adquisición de conocimientos de expertos externos o acerca de las necesidades del mercado. En otros casos, el aprendizaje involucra una forma de introspección por parte del sistema de reuso; observando sus propias operaciones, y memorizando instancias o generalizaciones de patrones recurrentes que le permitan mejorar su propia eficiencia. Así como el Análisis de Dominio esta relacionado con la adquisición de conocimientos acerca del dominio del problema, el análisis de infraestructura está relacionado con la adquisición de conocimiento acerca de las necesidades del mercado.

1.5.2 Inputs, Outputs Mecanismos y Actores.

El conocimiento acerca del dominio del problema y acerca de cómo implementar soluciones computacionales en ese dominio puede ser adquirido de diferentes fuentes. En general estas incluyen:

-Literatura Técnica: libros, manuales, papers, publicaciones científicas.

-Aplicaciones existentes, las cuales pueden ser investigadas como código fuente, documentos de diseño, manuales de usuario e implementaciones ejecutables.

-Análisis de mercado.

-Expertos humanos en el dominio del problema y en el diseño de sistemas en el dominio.

-Registros históricos acerca de la evolución del dominio.

En la práctica, cada fuente de información tiene ventajas y limitaciones. ¿Qué clase de información deberíamos obtener en cada una? Los expertos humanos son una buena fuente para obtener los razgos generales de la estructura conceptual del dominio del problema que permitan al analista navegar a través de la masa de información a ser examinada durante el análisis de dominio. Los expertos humanos son usualmente la única fuente de justificaciones, anécdotas o explicaciones de por qué las cosas son como son. Generalmente la memoria de un experto en el dominio es rica en conocimientos acerca de temas puntuales históricos que no encontraremos en otras fuentes de información.

La literatura técnica provee datos precisos y detallados, pero rara vez cubre puntos causales o de conocimiento histórico. Las aplicaciones existentes ayudarán a seleccionar el tipo de conocimiento que se intentará obtener de los expertos. La clase de información que ayudan a descubrir tiene que ver, por ejemplo, con variaciones en la definición de los objetos del dominio, relaciones, restricciones, diagramas de diseño especializados y conocimientos de implementación. El

estudio del mercado nos proveerá una base firme para establecer qué propiedades de especificación son esenciales, comunes o eventuales.

Cada fuente de información tiene sus desventajas. Por ejemplo, el tiempo de un experto humano es en general escaso y costoso. Además un experto se cansa, y suele ser un laborioso e ineficiente proceso el seleccionar conocimiento a partir de él. Aunque disponible y barata, la literatura técnica no puede ser utilizada para obtener justificaciones y elaboraciones. Además, ha de resultar sumamente engorroso para un analista, el encontrarse de buenas a primeras con decenas de tomos y publicaciones acerca de, por ejemplo, legislación impositiva. Las aplicaciones existentes y su documentación son usualmente muy específicas, sin la asistencia de un experto, los resultados son pobres. El estudio de mercado no provee más que la percepción y distribución de necesidades.

En todos los casos, las fuentes de información deben combinarse y complementarse, algunas proveerán información nueva, algunas nos ayudarán a seleccionar información de otras fuentes, otras nos ayudarán en la comparación y verificación.

El resultado del proceso de análisis de dominio es el modelo del dominio. Sus contenidos van determinados por las necesidades del trabajo de resolución del problema, en nuestro caso, la construcción de software. Entonces, el modelo debe contener, al menos:

- Una definición de los conceptos utilizados en la especificación de problemas y sistemas de software.
- Una definición que incluya decisiones de diseño de software típico, alternativas, trade-offs, justificaciones, etc.
- Un diagrama de implementación de software.

El modelo resultado tiene diferentes propósitos. Una taxonomía, por ejemplo, puede ser caracterizada como un modelo de definiciones, porque muestra qué cosas hay en el dominio y cómo están organizadas. Los modelos de representación de conocimientos como redes semánticas y frames, proveen la semántica del dominio. Los lenguajes específicos al dominio, cuando se expresan como gramáticas formales soportadas por parsers, son modelos que pueden soportar la traducción directa de especificación de software a código ejecutable.

Los modelos funcionales describen cómo los sistemas trabajan, utilizando diagramas de flujo de datos, o PDL (lenguage de descripción de programas). Hay además modelos que proveen información acerca de cómo construir sistemas en el dominio: los modelos estructurales.

Las representaciones de los modelos de dominios varían en complejidad y formalidad desde la simple taxonomía hasta la más rica estructura de una red semántica o la más elaborada definición de un lenguaje del dominio.

Cubrir el gap entre las fuentes de conocimiento y la representación formal o semi-formal del mismo es muy difícil. Expertos en dominios, expertos en diseño de software y analistas de dominios han desarrollado su papel en ese intento. Sin embargo, la fundamentación existente en la inteligencia humana y en métodos no

formales indica que a pesar de haber entendido las propiedades generales del proceso de análisis de dominio, estamos lejos de comprender los mecanismos que conducen a su éxito.

Al descomponer la actividad del analista en tareas más simples identificaremos herramientas y representaciones que faciliten dichas tareas.

Para sus representaciones, el análisis de dominios puede beneficiarse de la experiencia en otros campos. La representación de conocimiento y los lenguajes de modelación conceptual juegan un papel preponderante en el proceso de hacer explícito el resultado del análisis de dominio. La taxonomía y representaciones de la tecnología de orientación a objetos son muy utilizadas. Esquemas de representaciones convencionales en ingeniería de soft, tales como cartas estructuradas, diagramas de flujos de datos, diagramas de estados, o pseudocódigo son utilizados para expresar conocimientos de implementación. La especificación algebraica juega su papel cuando se trata de formalizar la semántica del dominio. Existe también un número de herramientas: editores, analizadores de lenguages de modelación, browsers de datos, etc. que son definitivamente de utilidad.

Así como existe un ingeniero de soft en el análisis tradicional, ¿es necesario el papel de un "ingeniero de dominio" ? A esta altura temprana en que nos encontramos en la materia, no estamos preparados para definir en detalle el skill, entrenamiento, o responsabilidades de un ingeniero de dominios. En cambio, podemos identificar cuatro roles funcionales bien definidos: expertos del dominio, analistas de dominio, analistas de infraestructura, implementador de infraestructura.

1.6 Conclusiones.

Una organización dedicada al desarrollo de soft interesada en incorporar técnicas de reusabilidad necesita establecer programas de ingeniería de dominio para manejar la identificación, captura y evolución de la información reusable.

La ingeniería de dominio no soluciona el problema infraestructural en reusabilidad más que lo que la ingeniería de soft hace respecto al problema de desarrollar y mantener software en forma efectiva y eficiente. La ingeniería de dominio debería proveer de modelos y direcciones para el trabajo de identificar, capturar y hacer evolucionar el conocimiento acumulado en una organización. El conocimiento es capturado en modelos de dominios, y almacenado en patrones de reuso evolutivos y en la infraestructura de reuso.

La infraestructura debe ser ajustable a las necesidades, recursos disponibles, técnicas y tecnología de cada organización. Y cada organización deberá evaluar sus procedimientos de construcción de soft, fuentes de información, oportunidades de reusabilidad, costos y beneficios para definir la conveniencia del esfuerzo involucrado en realizar el cambio a la ingeniería de dominio.

CAPITULO III : El Sistema de Reuso en Comunidades de Software.

Introducción.

El reuso y distribución de componentes de software entre los integrantes de una comunidad de desarrolladores requiere de una infraestructura de comunicación y servicios de información, tal como mencionáramos en el primer capítulo. En los siguientes párrafos trataremos los problemas involucrados en la utilización de componentes compartidas dentro de una **comunidad de software** y estudiamos una propuesta de desarrollo de software basado en el intercambio y reuso a gran escala de componentes de soft.

3.1. Comunidades de Software.

Definiremos una comunidad de software como un grupo de personas con una ¨cultura de soft similar¨, esto es, con un conjunto común de términos para comunicarse, un entendimiento común de los problemas involucrados en el proceso de desarrollo de soft.

Las comunidades profesionales existen en muchas ramas: abogados, médicos, ingenieros. La gente que trabaja en el desarrollo de soft no tiene, sin embargo, una única comunidad profesional con la cual se pueda identificar. Muchos reciben su entrenamiento en otras áreas y la naturaleza de su trabajo varía según el momento de su carrera y la actividad que desarrollen. Pueden identificarse más como matemáticos, ingenieros, artistas, contadores o científicos mas que como desarrolladores de soft. Existen, sin embargo dos fuertes vínculos que relacionan a todas estas personas: todos trabajan con un medio nuevo e interesante: el software, y tienen la necesidad de comunicarse con otras personas con ideas similares.

Hasta hoy, las comunidades de software se formaron mayormente en torno a un producto particular. El producto puede haber sido una máquina (AS-400, PC, Mainframes), un sitema operativo (por ejemplo Unix), un lenguaje o un sistema (por ejemplo un DBMS), un producto que fue principalmente utilizado para el desarrollo de soft. La gente que lo utilizó necesitó, entonces, juntarse e intercambiar experiencias en grupos de usuarios. Estos grupos de usuarios eran formados generalmente por los proveedores del producto, quienes los veían como un conjunto de actuales y futuros clientes consumidores. Estos grupos le resultaron a los proveedores de gran utilidad. Hoy en día es ampliamente aceptado que el éxito de una idea de soft, método, herramienta o producto está directamente relacionado con el tamaño y dinamismo de la comunidad de software con la cual

se identifica.

En los últimos años, el contexto en el cual se mueven las comunidades de software ha ido cambiando lentamente. Se nota un claro énfasis en sistemas de arquitectura abierta. Las organizaciones no quieren ser capturadas por un distribuidor o producto particular. Esta situación termina con el concepto de comunidad de soft como grupos de usuarios de un producto particular. Por otro lado, se ha incrementado la presencia de nuevos productos, y la gente se ha abierto a la influencia de distintas tecnologías. Esta apertura de mercado e ideas hace difícil que una comunidad permanezca con cierta cohesión y una cultura de soft particular "pura". La aparición de redes y sistemas avanzados de comunicación digital hacen de la geografía un hecho menos relevante. Así la ausencia de contacto humano hacen a las comunidades más volátiles. Por último la gente se mueve entre una comunidad y otra dependiendo del momento y sus intereses personales. LLamaremos comunidades abiertas a este tipo de grupos. En el caso de grupos pertenecientes a una misma organización dedicada al desarrollo de soft, hablaremos de comunidades cerradas. En éstas, la integración del individuo a la comunidad está dada por motivos muy diferentes a los mencionados en el caso de las comunidades abiertas. En estos grupos el objetivo es producir software de una forma eficiente, y con el menor costo, y es en este sentido en que el vuelco hacia una tecnología orientada al reuso y al desarrollo a partir de componentes se ofrece como una alternativa atractiva.

3.2. Sistemas de Información y Componentes de Software.

El resultado del cambio hacia sistemas más abiertos y la independencia de fabricantes y productos particulares es un entorno mucho más dinámico para la producción de software. Hoy es posible concebir comunidades realmente grandes, que vayan más allá de una única organización dedicada al desarrollo de soft. Estas comunidades proveen un entorno adecuado para el reuso de software a gran escala, donde el software es desarrollado y puesto a disponibilidad - por cierto precio en el caso de estas comunidades abiertas -.

Una comunidad que pretenda explotar el intercambio y reuso de software entre sus miembros, debe integrarse. Deben existir mecanismos y procedimientos bien establecidos para que sus miembros se comuniquen y compartan software. La forma más simple de proveer esto es a través del establecimiento de un **sistema de información de software.** Un sistema de información de software es un repositorio conteniendo toda la información: documentos, diseños y componentes de soft, disponibles en una comunidad particular. El sistema debería ser fácilmente accesible para los integrantes de la comunidad y contínuamente alimentado a través del desarrollo de nuevas componentes y del refinamiento de las existentes. En el último capítulo de este trabajo, proponemos un prototipo que apunta a mejorar el proceso de catálogo, publicidad y acceso a componentes de

soft.

La noción de **industria de componentes de soft** no es nueva. Sin embargo, desarrollos recientes dentro del paradigma de Orientación a Objetos y Networking, han brindado los fundamentos técnicos para el establecimiento de una industria de software basada en componentes. Hay, sin embargo, un conjunto de problemas a ser resueltos antes de poner en práctica el reuso de componentes a gran escala, en particular, el packaging, distribución, validación, y en el caso de comunidades abiertas, el marketing y pricing de componentes.

3.2.1. Packaging de Componentes.

El ¨packaging¨ de componentes se refiere al problema de estructurar y organizar la información asociada a una componente, de modo tal que la información necesaria para reusarla se encuentre agrupada. La necesidad de **empaquetar** componentes surge debido a que varios mecanismos como herencia y referencias externas, hacen que éstas a menudo dependan de otras componentes las cuales a su vez referencian a terceras.

Consideremos un programa que pretende hacer uso de una componente particular. El programador deberá localizar, al menos, la siguiente información: la especificación de la interface de la componente, el código fuente u objeto de las funciones utilizadas por la componente y el código fuente u objeto de las componentes de las cuales la componente a reusar depende. Además, el programador deberá considerar:

- Si los nombres utilizados por la componente entran en conflicto con los nombres ya existentes.
- Si el código objeto está disponible, si es compatible con el ambiente (procesador, sistema operativo) que el programa va a utilizar.
- Si está disponible el código fuente, si es posible compilarlo y ejecutarlo con las herramientas que el desarrollador está utilizando.

Es claro que si el Sistema de Información va a contener cientos de componentes, debería proveer alguna ayuda al programador en este sentido. Por ejemplo, el sistema debería, dada la información acerca del ambiente del programador y de las componentes de interés, proveer el código fuente y objeto necesarios y determinar si existe algún conflicto de nombres.

El packaging de componentes apunta a resolver estos problemas representando a la componente dentro del sistema de información en una forma que conduzca al reuso de la misma. El diseño del packaging de componentes es llamado ¨component modelling¨, y tiene en cuenta aspectos tales como las relaciones y dependencias entre componentes, interfaces y código fuente, versionamiento de componentes e interfaces, convensiones o stándares taxonómicos, e información descriptiva necesaria para acceder y navegar bibliotecas de componentes.

3.2.2. Distribución de Componentes.

Los sistemas de información de software, con sus conecciones de comunicación con los desarrolladores, forman la infraestructura que vincula a la comunidad de software. Sin embargo, si la comunidad pretende ser un todo y no un conjunto fragmentado de desarrolladores semi-independientes, debe existir alguna forma de integración entre el sistema de información y los ambientes de programación utilizados en la comunidad. En particular, debería ser posible transferir el software desde el sistema de información al ambiente local de cada desarrollador. Llamamos **distribución de componentes** a este proceso de transferencia de software desde el repositorio del sistema de información al ambiente de programación local.

Existen dos aspectos a ser estudiados en relación a este tema. Primero hay cuestiones relacionadas con el tipo de información a ser almacenado en el sistema de información de modo de cubrir los intereses de una comunidad grande y diversa. En muchos casos es necesario el almacenamiento de distintas variantes para una misma componente, cada una de las cuales es destinada a un sector particular de la comunidad. En otros casos es necesario versionar las componentes debido a optimizaciones; distintas plataformas de ejecución, sistemas operativos o compiladores. En general, los administradores de un sistema de información deben tener conocimiento acerca de las características demográficas de la comunidad, con el objeto de asegurar que el sistema ha sido alimentado con todas las variantes necesarias de una componente determinada.

El segundo aspecto en cuanto a la distribución de componentes tiene que ver con la naturaleza de las conecciones de comunicación entre el repositorio de software y los ambientes de desarrollo locales. Existe una serie de situaciones posibles, dependiendo de características tales como el ancho de banda de las conecciones, el esfuerzo requerido en establecer la comunicación y su costo.

La situación ideal consiste en una comunicación punto a punto entre el ambiente local del desarrollador y el repositorio de software. Debería haber un ancho de banda amplio y una conección barata y fácil de establecer. Esto se requiere sobre todo en comunidades cerradas en las que se pretende integrar el browsing y acceso a componentes del repositorio con los procedimientos locales habituales de desarrollo de software.

El acoplamiento entre el sistema de información y el ambiente en el cual la aplicación va a ser ejecutada es menos crítico. Sin embargo, si existe alguna forma de manejar las posibles incompatibilidades entre ambos ambientes, la distribución de componentes puede tornarse más dinámica y flexible. Por ejemplo, las nuevas versiones de una componente podrían distribuirse directamente a los lugares en los que se encuentran las aplicaciones, pudiéndose construir programas autoconfigurables capaces de obtener las nuevas variantes de componentes en tiempo de ejecución.

3.2.3. Validación de Componentes.

Cuando una componente es registrada en un sistema de información a través de algún procedimiento de catálogo, ésta debería ser sometida a un proceso de validación. El propósito de este proceso es garantizar que la componente sea "segura" para el reuso. Entre las características que deberían ser validadas figuran:

- Concordancia con la especificación: La componente debe proveer la implementación de todas las funciones descriptas en su interface.
- Standards: La componente debe respetar todas las convensiones establecidas en la comunidad en cuanto a documentación y taxonomía.
- Información de testeo y performance: Cualquier información acerca de la cantidad y calidad del testeo realizado y las características de performance esperadas será de interés para los eventuales usuarios de la componente.
- Virus: Un sistema de información sería un excelente distribuidor de virus al menos que las componentes sean inspeccionadas.
- Verificación de autoría: A efectos del pricing, y por cuestiones de seguridad y soporte sería beneficioso identificar al proveedor de una componente.

Cuando en una comunidad existe distribución de componentes en forma comercial, el proceso de validación se vuelve esencial. En estos casos, tanto los desarrolladores de componentes como los usuarios querrán asegurar la confiabilidad y calidad de las componentes que se distribuyen.

La validación de componentes plantea la necesidad de definir una política adecuada para resolver cuestiones como Quién o Quiénes serán los encargados de realizarla; Qué sucede si una componente pasa la validación pero falla en oportunidad de ser reutilizada; Cómo pueden ser probadas estas fallas; Cómo se realiza el proceso de solución de la falla y redistribución de la componente en estos casos.

3.3. Un caso real de traspaso a una tecnología orientada al reuso.

¿Es posible aplicar estos conceptos en casos reales? En la introducción de este trabajo hemos planteado algunas de las dificultades involucradas en el traspaso de la forma de trabajo de una organización hacia una tecnología de desarrollo orientado al reuso y bosquejamos algunas sugerencias para la forma en que podría darse el primer paso de este cambio. En el capítulo anterior brindamos algunos parámetros que nos permitirían evaluar la conveniencia o no de esta tranformación en la metodología de desarrollo. El siguiente caso ilustra cómo iniciar esta transformación a partir de un proyecto particular.

Esta estrategia ubica a la organización en condiciones de ir incrementando en forma gradual la cantidad de recursos dedicados al desarrollo de componentes

reusables a medida que se van obteniendo los primeros beneficios. Es una alternativa realmente factible y que minimiza el costo de la transformación. El proyecto que describimos tuvo lugar en la unidad de Warton de la Fuerza Aeroespacial Británica. La propuesta utilizada se basa en la construcción de bloques de software reusables en determinados puntos de la aplicación. En este caso, la reusabilidad de las componentes será determinada recién cuando el desarrollo haya concluído, momento en el cual podrán ser catalogadas en una biblioteca de componentes. El proceso de desarrollo centró su atención en el estudio de las siguientes cuatro áreas: a) La identificación de componentes reusables, b) El diseño de las componentes de soft de modo de lograr reusabilidad, c) El catálogo y recuperación de componentes y c) El diseño de sistemas capaces de incorporar las componentes reusables.

Para identificar las componentes reusables, se realizó un análisis de dominio, basado en el estudio de sistemas existentes en el área de interés. La tarea estuvo a cargo de un analista de reuso y un especialista en el dominio. Al final del análisis, se formuló un conjunto de preguntas capaces de identificar componentes reusables aún en áreas tan dependientes del hardware como el software de los controladores de vuelo. Estas preguntas son consideradas como el primer paso en la documentación de los atributos que definen a una componente como reusable. El análisis de dominio, ademas de identificar los atributos de una componente reusable, fue capaz de separar 41 componentes reusables en la etapa de definición de requerimientos.

Para soportar la infraestructura de reuso, se utilizó una herramienta de catálogo desarrollada por la Universidad de Strathclyde, acerca de la cual brindamos una breve evaluación en este capítulo.

3.3.1. El análisis del dominio del problema.

El dominio de problema elegido fue una simulación del sistema de manejo de instrumentos para el sistema del Programa Experimental Aéreo (EAP). El dominio comprende nueve subdominios, tres de los cuales fueron analizados para reuso. Los subdominios elegidos fueron los sistemas de propulsión, de manejo de combustible y el de tren de aterrizaje. El sistema de propulsión fue elegido por el especialista en el dominio debido a que el hardware controlado (por ejemplo, los motores) no cambiarían en forma significativa entre la implementación actual (EAP) y la siguiente (EAF- Euro-fighter). El dominio de los manejadores de sistemas de combustibles parecía tener una duplicación funcional importante a nivel de definición de requerimientos, por lo que fue escogido por el analista de reuso. La operación involucrada con el manejo del tren de aterrizaje no variaría en futuras implementaciones, por lo que, según el experto en el dominio, era adecuada para ser analizada en términos de reuso.

La idea de analizar un dominio de problema es que los procesos que tienen lugar en él, los objetos que lo componen, y los vínculos existentes entre ellos

puedan ser entendidos y documentados. El desarrollo de un método para realizar el análisis fue conducido con los siguientes objetivos:
-Descubrir las funciones que involucran reusabilidad.
-Enfocar la atención del especialista en el reuso.
-Ayudar al especialista a determinar parámetros de reuso.
-Descubrir cómo rediseñar componentes existentes para que sean reusables.

Estos objetivos llevaron a una técnica de análisis de dominio estructurado, basado en preguntas dirigidas a determinar la reusabilidad de una componente de software. La estructura utilizada fue un árbol de decisión, en el cual las preguntas iniciales estaban influenciadas por los requerimientos funcionales de futuras implementaciones, y las preguntas en los nodos tenían que ver con el reuso y la implementación de las dependencias en el código. A pesar de parecer demasiado rígida para las primeras versiones, esta estructura era necesaria para forzar al especialista en el dominio a considerar el reuso por sobre todo y en una forma no intuitiva. El árbol de decisión además tiene la ventaja de ser fácilmente transformable en una red semántica para una futura implementación como un sistema inteligente basado en conocimiento.

3.3.2. Identificación de atributos que hacen a una componente reusable.

Para cuantificar la reusabilidad de una componente de software, se formularon 12 preguntas a ser presentadas ante el experto en el dominio. Estas preguntas lo conducirán a determinar el grado de reusabilidad de una componente de una manera clara:
-¿Es requerida la funcionalidad de la componente en futuras implementaciones?
-¿Cuán común es la función de la componente en el dominio bajo estudio?
-¿Qué grado de dependencia del hardware tiene la componente?
-¿Permanece el hardware invariante entre una implementación y otra?
-¿Pueden trasladarse las especificaciones de hardware a otra componente?
-¿Está el diseño lo suficientemente optimizado como para utilizarse en la próxima implementación?
-¿Es posible parametrizar una componente no reusable de modo de transformarla en reusable?
-¿Es posible reutilizar la componente en varias implementaciones sólo realizando modificaciones menores?
-¿Es posible el reuso a través de la modificación?
-¿Es posible descomponer una componente no reusable, en un conjunto de componentes reusables?
-¿Cuán válida es la descomposición de componentes para el reuso?

3.3.3. Resultados del Análisis de Dominio.

En el dominio del manejo de propulsión, de las 19 componentes identificadas a nivel de requerimientos funcionales, 1 componente fue clasificada como reusable sin necesidad de cambios, 14 como reusables a través de pequeñas modificaciones y 4 como no reusables. En el dominio de manejadores de sistemas de combustible, de las 26 componentes identificadas a nivel de requerimiento funcional, 2 fueron clasificadas como reusables sin necesidad de cambios, 14 fueron clasificadas como reusables a partir de leves modificaciones y 10 como no reusables. En el caso del manejo del tren de aterrizaje, las 10 componentes fueron clasificadas como reusables a través de pequeñas modificaciones.

Queremos subrayar este resultado, ya que nos muestra la importancia de facilitar la modificación de componentes de software. El sistema de información que soporta la infraestructura para el reuso debería proveernos de las herramientas necesarias no sólo para **catalogar** y **recuperar** componentes, sino también para **entender** el funcionamiento de una componente una vez recuperada sin lo cual es imposible su modificación. La práctica nos muestra que la **adaptación** de la componente a las necesidades de cada aplicación particular es un punto fundamental. Rara vez una componente puede reutilizarse sin realizar modificaciones (adaptativas o extensivas). En el capítulo VII volvemos a tratar este punto y diagramamos e implementamos una propuesta.

3.3.4. Catalogando y Recuperando componentes.

Este trabajo incluyó el uso y evaluación de un catálogo de componentes desarrollado por la Universidad de Strathclyde. Esto llevó a investigar ciertos problemas relacionados con el proceso de catálogo y el futuro rol de los catálogos de componentes en los ambientes de soporte para el reuso de software. La actividad principal del proceso de catálogo tiene que ver con el ingreso de las componentes a una base de datos, y la navegación de dicha base a través del mecanismo de recuperación. La evaluación se basó en la facilidad en el proceso de catálogo y en la exactitud en la recuperación de la componente deseada.

El catálogo estudiado corre bajo UNIX en una estación de trabajo SUN-3. La interface se realiza a través del sistema de ventanas de SUN. La facilidad de manejo en el llenado de formularios permite que el catálogo sea utilizado aún con muy poco entrenamiento. Sin embargo, se encontró que es necesario un conocimiento profundo acerca de la teoría utilizada en los descriptores de componentes para catalogar componentes y utilizar los mecanismos de recuperación con todo su potencial.

El catálogo de las componentes involucró la creación de un frame descriptor para cada una. Los frames descriptores de 41 componentes fueron ingresados en la base de datos. El esfuerzo requerido en este proceso fue de aproximadamente una semana-hombre, e incluyó el entendimiento de la teoría utilizada en los frames

descriptores de componentes, la creación de los descriptores y la edición de los archivos de datos que los contienen. Los problemas encontrados tienen que ver con errores de tipeo o manejo de sinónimos que repercuten en fallas en el momento de recuperación de componentes.

En la evaluación del proceso de recuperación, se sabía que la base no contaba con un volumen de información lo suficientemente representativo como el disponible en un ambiente de soporte de reuso real. Por otro lado, la familiaridad de los analistas con las componentes almacenadas en el repositorio influirían en el proceso de recuperación. Por esto, se pidió a terceros que intentaran recuperar componentes del repositorio. Durante la evaluación, el número de componentes retornadas fue, en promedio, 15. Es una cantidad razonable de ser presentada, sin embargo, el número de componentes catalogadas era del orden de 60 (se incluyeron algunas pertenecientes a un análisis de dominio previo), lo cual transforma a las 15 componentes recuperadas en un porcentaje demasiado elevado. Por otro lado, el orden en que las componentes son presentadas tiene que ver con el nivel de matching que existe con el pedido, esto hace que baste con analizar una o dos para encontrar la componente deseada. Si la proporción se mantuviera en un sistema real, con cientos de componentes almacenadas, se deberían presentar al usuario cientos de componentes recuperadas. Existen distintas alternativas para solucionar este punto: Limitar la búsqueda a las N componentes que tengan mejor matching con la solicitada, profundizar el proceso de catálogo, invirtiendo más tiempo y esfuerzo en él para facilitar la recuperación, implementando alguna forma de catálogo automático, etc.

3.3.5. Conclusiones.

En la utilización de la técnica de análisis de dominio para lograr los objetivos originales, se encontró que el criterio más importante fue la aceptación de la existencia de un analista de reuso capaz de enfocar la atención del experto en el dominio en el reuso y educarlo en sus beneficios. Se consideran logrados los objetivos en un dominio limitado, con pocos especialistas, pero se plantearon una serie de problemas que entran en perspectiva al pretender lograr reusabilidad a gran escala, específicamente, la necesidad de formar ingenieros en reuso, analistas de dominios y desarrollar herramientas capaces de soportar eficientemente una infraestructura para el reuso.

CAPITULO IV : ¿ Por qué Objetos ?

4.1 El Paradigma de Orientación a Objetos.

Son conocidas las características involucradas en el paradigma de orientación a objetos que promueven la creación de software reusable: La abstracción de datos nos lleva a desarrollar sistemas modulares que son fáciles de entender. La herencia brinda el mecanismo para que varias subclases compartan métodos definidos en superclases y permitir la ¨programación por diferencia¨. El polimorfismo permite a una componente comportarse correctamente en nuevos contextos de ejecución. La Programación Orientada a Objetos promueve el reuso. Lenguajes como Smalltalk no sólo reducen el tiempo de desarrollo sino también el costo de mantenimiento, simplificando la creación de nuevos sistemas y de nuevas versiones de sistemas ya existentes. Esto es cierto, pero la P.O.O. no es la panacea. Las componentes deben ser diseñadas con un concepto de reusabilidad. Los diseñadores de sistemas deben planear el reuso de componentes existentes y tratar de desarrollar nuevas componentes reusables.

Una de las razones por las cuales la P.O.O. se ha tornado popular es el hecho de que el reuso de software se ha tornado popular. Desarrollar nuevos sistemas es caro, y mantenerlos cuesta aún más. En el capítulo anterior hemos mencionado algunos datos relacionados con el costo de mantenimiento del soft. Un estudio reciente (de Wilma Osborne, del Bureau Nacional de Standards) sugiere que entre el 60 y el 85% del costo total de un sistema se atribuye al mantenimiento. Las características de la P.O.O. mencionadas antes son útiles también en esta etapa. La modularidad hace más fácil entender el efecto que producen los cambios al resto del sistema. El polimorfismo reduce el número de procedimientos, por lo cual el tamaño del programa que debe ser entendido por quien realiza el mantenimiento es menor. La herencia de clases permite la construcción de una nueva versión de un programa sin afectar a la anterior.

Muchas de las técnicas de reuso de software escrito en lenguajes convencionales son paralelizadas por técnicas de la orientación a objetos. Por ejemplo, los ´esqueletos´ de programas son reemplazados por la creación de clases abstractas. El "copy & paste" es mejorado mediante la herencia a partir de una clase y la reimplementación en la subclase de algunos de sus métodos. Aún así, el reuso no ocurrirá por arte de magia. El concepto implica un cambio de actitud. No se debe temer el utilizar clases desarrolladas por otros programadores. Reescribir una clase para hacerla más reusable o fácil de utilizar es tan importante como desarrollar una nueva componente. Una nueva clase que no es compatible con las clases existentes pierde sentido. Se debe estar dispuesto a invertir tanto tiempo en leer código existente para estudiar la forma de reutilizarlo como en

escribir código nuevo.

Existen un conjunto de técnicas de diseño que hacen al software orientado a objetos más reusable. En este capítulo describiremos algunas de estas técnicas.

4.2 Proceso de desarrollo de un Producto Reusable.

Lo que sucede en la realidad, durante el desarrollo de una aplicación, es que las necesidades de nuestro cliente casi siempre cambian. El conjunto de requerimientos finales que una aplicación debe satisfacer es distinto al conjunto de requerimientos originalmente percibido. Se suma a esto el hecho de que el resultado obtenido es la base para un complejo proceso de "cut & paste" para que futuras aplicaciones satisfagan sus propios requerimientos. Todo esto resulta en un costo de mantenimiento significativamente desproporcionado con respecto al costo original de desarrollo. Lo que es más, nuevos desarrollos mantienen un costo alto a pesar de su aparente similtud con desarrollos previos. La causa de estos problemas es la carencia de planeamiento en el desarrollo. En la mayoría de los casos, el único producto que los desarrolladores de software están construyendo es la aplicación del usuario final. Comúnmente no se reconoce que cada componente en un sistema de software es también un producto final para otro tipo de ´cliente´: los desarrolladores encargados del mantenimiento y quienes puedan necesitar reusar dichas componentes. El dominio potencial de aplicabilidad de una componente puede ser mucho más amplio que el de la aplicación para la cual fue desarrollada. Es necesario entender y explotar este punto para obtener el máximo beneficio de cada desarrollo.

Para desarrollar componentes de software reusables, es necesario entender el proceso básico de desarrollo de cualquier clase de producto reusable. Cuando se desarrollan esta clase de productos, ya sea que se trate de hardware, software comercial, componentes mecánicas o cualquier otro producto, se ejecutan una serie de pasos similares. Entonces, podemos examinar el modelo de desarrollo de un producto reusable y trasladarlo al desarrollo de soft, estableciendo algunas consideraciones especiales que nos permitan dirigir nuestro proceso de desarrollo a un proceso orientado a la componente de soft como producto.

La metodología utilizada para crear componentes de software reusable debe asumir que dichas componentes son productos destinados a los desarrolladores que las mantendrán o reusarán. El reuso potencial de una componente es influenciado por la misma clase de factores que afectan el reuso de cualquier otro producto: La habilidad de adaptarse a las necesidades variantes del usuario.

Los siguientes puntos deberían ser cubiertos por una buena metodología de desarrollo de componentes de software reusable:

-Realización de un "Análisis de Mercado": Antes de proceder con el desarrollo de una componente de software reusable, debería establecerse la necesidad de la

misma. Es necesario definir el dominio de aplicación de la componente y cuantificar su reuso potencial. Como con cualquier otro producto, el "mercado" debe ser maximizado para obtener un óptimo nivel de reuso, manteniento la restricción de un costo de desarrollo razonable. Por otro lado, se deberá planear la asignación de recursos adecuada al desarrollo de la componente.

-Realización de Análisis de Dominio: Debe establecerse un entendimiento apropiado del dominio de problema de la componente, de modo tal de poder formular los requerimientos en forma adecuada. El proceso de análisis de dominio consta de dos tareas bien diferenciadas: el análisis de dominio de la aplicación, que nos permitirá identificar las componentes a ser implementadas y su clasificación asociada, y el análisis de dominio de la componente, que será utilizado para formular los requerimientos de implementación de la componente que satisfagan un dominio de reuso óptimo.

-Clasificación de las Abstracciones Reusables: Esta clasificación formará las bases para manejar del acceso a las componentes. El proceso de clasificación debería asegurar un mínimo número de pasos de selección. Hacia el final de este trabajo presentamos una propuesta de catálogo de componentes basado en clasificación.

-Definición de una Especificación de Interface Abstracta: Incluyendo una enumeración explícita de todas las interfaces asociadas con el uso de cada componente reusable. Esta especificación debería minimizar la posibilidad de cambios en la interface como resultado de cambios en la implementación de la componente. Tanto interfaces funcionales como de ´customización´ deberán ser definidas: las interfaces funcionales definen las entradas y salidas asociadas a cualquier instanciación de la componente; en tanto las de ´customización´ permitirán que diferentes implementaciones de una componente sean construídas para satisfacer las necesidades específicas de cada usuario.

-Implementación de un Prototipo: Debería implementarse y utilizarse un prototipo con el objetivo de evaluar la potencial reusabilidad de cada componente. El prototipo debería posibilitar la customización de la componente. Su utilización ayudará a lograr un entendimiento apropiado de los cambios en el dominio y los requerimientos deseados para maximizar la reusabilidad de la componente.

-Implementación de la Componente para Producción: Las necesidades descubiertas a través del uso del prototipo deberían ser implementadas en una versión final.

-Publicidad de la Componente: Se deberá asegurar que el producto resultado sea accesible y visible, esté disponible, y sea percibido como claramente aplicable por el conjunto de potenciales usuarios. Este paso podría ser cumplimentado a través

del catálogo de la componente de software en un sistema de información apropiado.

-Soporte para el Uso de la Componente: El soporte de mantenimiento, incluyendo análisis y corrección de problemas deberían ser provistos.

4.3 Proceso de Desarrollo de Clases Reusables.

Bajando el nivel de abstracción, y concentrándonos particularmente en una metodología de desarrollo dentro de la Tecnología de Orientación a Objetos, nos encontramos con que, a nivel de implementación, debemos comenzar por crear clases reusables. Muchos de los puntos mecionados antes para el desarrollo de un producto reusable, deben ser mapeados en actividades puntuales durante el desarrollo de clases que maximicen el reuso.

4.3.1 Utilización de Protocolos Standard.

Es muy importante que el proceso de diseño resulte en protocolos standard. La creación de protocolos standard facilitará la composición de aplicaciones a partir de componentes reusables.

Los protocolos standard son desarrollados eligiendo los nombres cuidadosamente. La necesidad de utilizar protocolos standard es una de las razones por las cuales lleva tiempo convertirse en un programador Smalltalk experto. Muchos de los protocolos están descriptos en los manuales, pero la mayoría sólo están documentados en el código fuente y se incorporan con la experiencia. Muchas clases deberían tener casi idénticas interfaces externas, y debería haber conjuntos de operaciones que muchas clases implementan.

Existe un número de reglas que ayudan a desarrollar protocolos standard. El adoptarlas ayudará a evitar el dar diferentes nombres a las mismas operaciones en distintas clases. Estas reglas minimizan el número de nombres diferentes y maximizan el número de nombres compartidos por un conjunto de clases:

a) Introducción de Recursión: Si una clase se comunica con un número de otras clases, la interface con cada una de ellas debería ser la misma. Si una operación X se implementa realizando una operación similar sobre las componentes del receptor, entonces dicha operación también debería llamarse X. Aún si el nombre debiera ser cambiado debido al número de argumentos, tiene sentido hacer que los nombres sean similares de modo que quien lee el programa note dicha conección. El resultado es que un método para un mensaje envía el mismo mensaje a otros objetos. Si los otros objetos pertenecen a la misma clase que el emisor, entonces el método es recursivo, aún cuando no exista recursión real. Esta

regla ayudará a decidir en qué clases una operación debería ser implementada como método.

b) Eliminación de Análisis de Casos: Es casi siempre un error checkear la clase de un objeto. Código de la forma:

(obj class == ClassA) ifTrue: [obj haceEstoA] ifFalse: [obj haceEstoB]

debería ser reemplazado por un mismo mensaje al objeto cuya clase está siendo checkeada. Los métodos deberán ser creados en las posibles clases a la cual pueda pertenecer el objeto, en la forma adecuada a cada caso.

c) Reducción del Número de Argumentos: Mensajes con muchos argumentos son difíciles de leer. Excepto para métodos de creación de instancias, este tipo de mensajes deberían ser redefinidos. Cuanto menor número de argumentos tiene un mensaje, se incrementa la probabilidad de que sea similar a algún otro mensaje, pudiéndoles asignar igual nombre. El número de argumentos puede ser reducido particionando el mensaje en varios mensajes más pequeños o creando una nueva clase que represente el conjunto de argumentos. Muchas veces existen muchos mensajes que manipulan el mismo conjunto de argumentos. Este conjunto de argumentos es esencialmente un nuevo objeto, y el diseño puede ser cambiado para reflejar este hecho.

d) Reducción del Tamaño de los Métodos: Los métodos en Smalltalk son casi siempre pequeños. Es más fácil especializar clases con métodos pequeños, ya que el comportamiento puede ser alterado a través de reimplementaciones de estos métodos simples en la subclase creada. Un método de 30 líneas es largo y probablemente necesita ser particionado. A menudo sucede que un método en una clase es dividido cuando se crea una subclase. La mayor parte del método heredado es correcto, pero una parte necesita ser modificada. En lugar de reescribir el método por completo, se divide en varios métodos, algunos de los cuales deben ser redefinidos.

4.3.2 Utilización de Clases Abstractas.

Los protocolos standard son a menudo representados por clases abstractas. En muchos casos, las clases abstractas pueden ser utilizadas como el ´esqueleto´ del programa, donde el usuario completa ciertos vacíos y reutiliza el código estructural. En general es conveniente heredar de una clase abstracta. A diferencia de las clases concretas, no necesitan proveer una definición para su representación de datos, por lo que cada subclase puede adoptar la representación que más le convenga sin temer entrar en conflicto con la

representación de datos heredada.

Crear nuevas clases abstractas es importante, pero no es fácil. Es siempre más fácil reusar una abstracción bien empaquetada que inventarla. En general tratamos de crear nuevas clases a partir de las existentes, esto es menos trabajoso que crearlas desde cero. Como resultado solemos obtener jerarquías cuya raíz es una clase concreta. Ante este tipo de situaciones es conveniente encontrar la forma de reestructurar la jerarquía, encontrando la clase abstracta escondida en la raíz de la jerarquía de herencia.

Existen un conjunto de reglas que nos ayudan a encontrar clases abstractas:

a) Una jerarquía de clases debería ser profunda y angosta: Una jerarquía de clases bien desarrollada debería tener varios niveles de profundidad. Una jerarquía con una superclase y 27 subclases necesita evidentemente un cambio, aunque no nos da una idea de cómo realizarlo. Las posibles modificaciones consisten en particionamiento de métodos, división de clases, movimiento de métodos y clases, etc.

b) La raíz de una jerarquía de clases debería ser una clase abstracta: La herencia por generalización a menudo indica la necesidad de creación de nuevas subclases. Si una clase B reimplementa un método X que hereda de una clase A, entonces puede resultar conveniente crear una nueva clase C, con los métodos que tanto A como B compartirán (vía herencia) e implementar en A y B los métodos que son particulares a ellas (por ejemplo X).

c) Minimizar el acceso a variables: Dado que una de las principales diferencias entre una clase abstracta y una concreta es la presencia de una representación de datos, las clases pueden hacerse "más abstractas" eliminando la dependencia sobre su representación de datos. Una de las formas en que podría hacerse esto es accediendo a todas las variables a través de mensajes. La representación de datos puede ser cambiada redefiniendo los métodos de acceso.

d) Las subclases deberían ser especializaciones: Existen varias formas diferentes de utilizar la herencia. La usualmente descripta como ideal es la especialización, en la que los objetos de las subclases pueden ser pensados como elementos de la superclase. La subclase no debería redefinir métodos heredados, sino agregar métodos nuevos. Un caso distinto de especialización es la creación de clases concretas a partir de clases abstractas. Como las clases abstractas no son ejecutables, crear una subclase de una clase abstracta es diferente a especializar una clase concreta. Una clase abstracta requiere que sus subclases implementen ciertas operaciones, por lo que crear una clase concreta es como llenar espacios blancos en un esqueleto de programa. Una clase abstracta puede definir ciertas operaciones en una forma muy general, y requerir que sus subclases las redefinan.

4.4 Frameworks.

Una de las formas más importantes de reuso es el reuso de diseños. Una colección de clases abstractas puede ser utilizada para expresar un diseño abstracto. El diseño de un programa puede generalmente ser descripto en términos de sus componentes y de la forma en que interactúan. Un diseño abstracto orientado a objetos se denomina framework. Un framework consiste de una clase abstracta por cada una de las componentes significativas de un diseño. Las interfaces entre las componentes del diseño se definen en términos de conjuntos de mensajes. Usualmente existirá una librería de subclases que pueden ser utilizadas como componentes en el diseño.

MacApp es un framework para aplicaciones Macintosh. Una aplicación abstracta MacApp consiste en una o más ventanas, uno o más documentos y un objeto aplicación. Una ventana contiene un conjunto de vistas cada una de las cuales muestra parte del estado de un documento. MacApp también contiene comandos con mecanismos de undo/redo automáticos, manejadores de impresión que proveen independencia del hardware utilizado. La mayoría de las aplicaciones deben hacer poco más que definir la clase de sus documentos. Heredan el intérprete de comandos y opciones de menú. Las clases de documentos hacen poco más que definir sus ventanas y la forma en que se leen y escriben en disco. Heredan las opciones de menú para grabar y abrir documentos. Un programador rara vez crea nuevas clases de ventana, pero usualmente debe definir una clase view para contener la imagen de un documento. MacApp no sólo asegura que las aplicaciones tendrán la interface de usuario standard Macintosh, sino que además hace más facil el desarrollo de programas interactivos.

Los frameworks son útiles para reusar no sólo el codigo principal de una aplicación, sino que también pueden describir diseños abstractos para librerías de componentes. La habilidad de un framework para permitir la extensión de estas librerías es una de las características más deseadas.

Los frameworks son más que librerías de clases bien escritas. Un ejemplo de librería de clases es el conjunto de clases definido en Smalltalk para Colecciones. Estas clases proveen formas de manipular colecciones de objetos tales como arreglos, diccionarios, bags, conjuntos. En cierta forma, uno podría encontrar esta clase de herramientas en una librería de un sistema de programación convencional. Cada elemento de esta librería es independiente de contexto, sirve para una amplia variedad de problemas diferentes. Son componentes independientes de la aplicación en que se utilicen. Un framework, en cambio, es un diseño abstracto para una clase párticular de aplicaciones, y usualmente consiste en un conjunto de clases, muchas de ellas abstractas. Los frameworks proveen una forma de reusar código que es muy difícil de reutilizar con las alternativas convencionales. Las componentes independientes de la aplicación pueden ser reutilizadas de una manera más sencilla. Pero reutilizar una estructura

en la que las componentes interactúan entre sí plantea seguramente nuevos inconvenientes como el tratar de mantener la consistencia a la vez de satisfacer los requerimientos particulares de una aplicación.

Una característica importante de los frameworks es que los métodos definidos por el usuario serán a menudo invocados desde el mismo framework más que desde el código de la aplicación del usuario. El framework juega el rol del programa principal en la coordinación y secuenciamiento de las actividades de la aplicación. Los métodos provistos por el usuario adaptan los algoritmos genéricos definidos en el framework a una aplicación particular.

El comportamiento específico de un framework para una aplicación se define agregando métodos a las subclases de una o más de sus clases. Cada método agregado a una subclase debe concordar con las convenciones de sus superclases. Llamamos a éstos frameworks de caja blanca, ya que su implementación debe ser entendida para poder utilizarlos. El mayor inconveniente con tales frameworks es que toda aplicación requiere de la creación de muchas nuevas subclases. A pesar de que la mayoría de estas subclases son simples, su número puede hacer difícil a un nuevo programador comprender el diseño de una aplicación lo suficientemente bien como para cambiarlo. El segundo problema es que aprender a usar un framework de caja blanca es tan difícil como comprender cómo está construído. (El problema del ´entendimiento´ de una componente a reutilizar se repite también fuera del ámbito de los frameworks. Durante el desarrollo de este trabajo presentaremos una propuesta que intenta atacar el problema de comprender el funcionamiento de una componente lo suficiente como para permitir su reutilización y eventual modificación.)

Otra forma de customizar un framework es mediante un conjunto de componentes que proveen el comportamiento específico de la aplicación. Cada una de estas componentes deberá entender y satisfacer un protocolo particular. Todas o la mayoría de las componentes pueden ser provistas por una librería. La interface entre componentes puede ser definida por protocolo, de modo que el usuario sólo necesita entender la interface externa de las componentes. A esta clase de frameworks se lo denomina de caja negra.

Los frameworks de caja negra son más fáciles de aprender a utilizar que los de caja blanca, aunque son menos flexibles.

Dado que los frameworks proveen reuso en la mayor granularidad posible, no es sorprendente que sean mucho más difíciles de diseñar que otro tipo de componentes reusables. Diseñar un framework requiere de gran habilidad y experiencia.

4.5. Más que componentes reusables...

En el ambiente competitivo del negocio de hoy, es crítico para las organizaciones saber aprovechar al máximo su potencial informático. Las

tecnologías orientadas al reuso apuntan a esto. Sin embargo, está surgiendo un nuevo problema: La variedad de entornos y arquitecturas impiden a muchas organizaciones aprovechar las ventajas del reuso. Aquellas que sean capaces de traspasar las barreras impuestas por la heterogeneidad de arquitecturas que aparecen día a día, en forma económica y efectiva en cuanto a tiempo y dinero, obtendrán una ventaja competitiva importante en el mercado. La reciente aparición de DOM (Distributed Object Managment) se presenta como una oportunidad para maximizar el potencial computacional de una organización dedicada al desarrollo de soft. Se trata de algo más que componentes reusables, la tecnología DOM permite a estas componentes interactuar a través de plataformas heterogéneas en un ambiente de procesamiento distribuído. DOM intenta definir standards para interfaces de objetos abiertas. El logro más significante de OMG (Object Management Group) fue el desarrollo del standard CORBA (Common Object Requester Broker Architecture) el cual define una interface de lenguaje neutral que permite a un objeto cliente requerir servicios de un objeto servidor sin tener conocimiento explícito acerca de la ubicación o identificación del mismo. ORB (Object Request Broker) es el mecanismo definido en CORBA para manejar el intercambio de mensajes entre objetos que residen en plataformas heterogéneas.

El standard CORBA ya ha sido incorporado en productos comerciales de empresas como IBM, DEC, HP, SunSoft, IONA y Nec. Actualmente, ORB opera entre DOS, Windows, OS/2, Unix y Macintoch. Los lenguajes soportados son principalmente C++, Smalltalk, Cobol y Rexx. Muchas empresas han formado alianzas con las anteriores con el objeto de lanzar productos que interactúen vía ORB (IBM, Apple, Novell, WordPerfect, Tivoli, Borland, Lotus, Digitalk).

Quienes sean capaces de incorporar esta tecnología en el desarrollo de componentes reusables, obtendrán un beneficio extra: por un lado los inherentes a la reusabilidad, por el otro un mayor grado de independencia del hardware y entorno de ejecución y la disponibilidad de conección con la gran cantidad de componentes existentes en el mercado desarrolladas por empresas que ya se han plegado a esta tecnología.

En cuanto a los lenguejes de programación, Smalltalk permanece como el paraíso de los programadores que desarrollan su actividad dentro de la tecnología de Orientación a Objetos. Sin embargo, aún no ha satisfecho los requerimientos de performance esperados por los usuarios de aplicaciones. Por otro lado, a pesar de las ventajas que le son inherentes como lenguaje de programación orientada a objetos puro, el traspaso se hace muy difícil para programadores habituados a lenguajes procedurales como COBOL o C.

A pesar de sus recursos de Orientación a Objetos, C++ no ha brindado claramente una de las mayores promesas de la P.O.O.: la reusabilidad del software. En términos de empaquetado y distribución de módulos binarios, C++ representa un gran paso hacia atrás. Las bibliotecas linkeadas estáticamente siempre han sido una manera efectiva de intercambiar código reutilizable. Con la aparición de Librerías Dinámicas y sistemas operativos basados en DLL (Windows

y OS/2) las bibliotecas funcionaron mucho mejor aún. Sin embargo, cuando se modifican las bibliotecas de clases de C++, sus clientes deben recompilar para ordenarlas. Existe además un engorroso problema de compatibilidad de clases entre compiladores. Esto hace que hasta el momento C++ no logre resolver el problema de intercambio de componentes a gran escala, aunque sí a nivel de proyecto, departamento u organización.

Objective-C, de StepStone constituyó una respuesta a este problema, SOM (System Object Model) de IBM es otra respuesta. Cada una de estas tecnologías emplea un motor runtime para permitir que los objetos se vinculen dinámicamente mientras mantiene preservado el flujo de herencia a través de los límites del objeto. Con SOM se pueden agregar funciones virtuales o incluso redefinir la jerarquía de clases.

El recurso DirectToSOM que Pennello está agregando a High C/C++ acerca el modelo de objeto interno del compilador a la funcionalidad de SOM, resolviendo los problemas de intercambio de componentes y compatibilidad de enlaces. Los beneficios de esta tecnología compensan con creces los costos en performance que ocasionan. Pronto se podrán realizar intercambios binarios directos entre objetos de diferentes compiladores.

En síntesis, la tecnología del software apunta desde todos los sectores a maximizar la reutilización de componentes, a partir de técnicas de ingeniería de soft, del desarrollo de herramientas y lenguajes adecuados y de la standarización de protocolos de comunicación entre objetos que maximizan las posibilidades de reuso.

4.6. Conclusión

La experiencia de desarrollar aplicaciones complicadas en unos pocos días utilizando técnicas y métodos del Paradigma de Orientación a Objetos es posible sólo si los desarrolladores son capaces de reutilizar componentes de software y diseños abstractos en una forma eficiente.

Construir componentes y diseños reusables lleva mucho tiempo, sin embargo es una inversión recuperable en el mediano plazo.

Hemos mencionado los factores que juegan su rol en el logro de un alto grado de reusabilidad en componentes orientadas a objetos. El polimorfismo nos da herramientas para garantizar que una componente se comportará adecuadamente en distintos contextos. La herencia promueve el surgimiento de protocolos standard y permite customizar componentes existentes. Por otro lado introduce el concepto de clase abstracta. Los frameworks permiten a un conjunto de objetos servir como molde de solución para determinada clase de problemas.

Las técnicas de la Tecnología de Orientación a Objetos nos ofrecen una alternativa a escribir los mismos programas una y otra vez. Podemos, en cambio utilizar ese tiempo en crear y perfeccionar componentes generales con la

convicción de que nuestras metodologías de desarrollo nos permitirán reexplotarlas.

CAPITULO V: Reutilizando e Interconectando Componentes.

Introducción.

En el capítulo anterior mencionamos algunos temas a considerar al tratar de desarrollar clases reusables, punto de partida para el desarrollo de componentes reusables. Pero esto no es todo. Los usuarios necesitan aplicaciones, es difícil pensar que una aplicación pueda consistir de un único objeto o una única componente por más poderosa que ésta sea. En general, una aplicación estará formada por muchas componentes trabajando juntas. Por esto, necesitamos algún mecanismo que nos permita interconectar componentes para formar aplicaciones.

La composición es una actividad que aparece en casi todos los dominios. Se compone texto en literatura, se componen sonidos para obtener música, se componen escenas para obtener una película. La programación misma trabaja con composición de sentencias y procedimientos. Aún cuando todos los objetos que necesitamos para desarrollar una aplicación se encuentren preprogramados y sean reusables, necesitaremos componerlos e interconectarlos. El proceso de composición es, por esto, crítico en el desarrollo de aplicaciones a partir de componentes reusables. La posibilidad de incorporar a nuestro proceso de desarrollo orientado al reuso, algún mecanismo de composición automático de componentes de soft redundará en un nuevo incremento en la productividad.

La definición de tal mecanismo de composición es difícil por varios motivos. Primero, los niveles de funcionalidad de las componentes a ser interconectadas suelen ser muy distintos. Necesitamos formas de componer objetos básicos, componentes y subsistemas. Segundo, nos encontraremos con que los objetos reusables pueden tener interfaces bastante diferentes. A pesar de estarse desarrollando ciertos standards, como mencionáramos en capítulos anteriores (por ej. CORBA), éstos aún no se han popularizado. Necesitaremos resolver eventuales incompatibilidades en las interfaces de las componentes a ser interconectadas. Tercero, la concurrencia y la posibilidad de contar con objetos distribuídos hacen aún más difícil la composición.

Los principales problemas pero también oportunidades en la interconección de componentes derivan de la reusabilidad. Es muy distinto componer objetos hechos a medida a componer componentes reusables prefabricadas. Centraremos nuestra atención en esta segunda actividad.

La creatividad necesaria se expresa de tres maneras diferentes. Primero, necesitaremos decidir qué componentes reusaremos; segundo, cómo deberían ser modificadas y tercero, cómo serán interconectadas. Estos tres aspectos se

entrelazan y no pueden ser resueltos en forma independiente.

En este capítulo presentaremos algunas propuestas que intentan resolver el problema de la composición de objetos, punto de fundamental importancia en el proceso de desarrollo de aplicaciones a partir de componentes reusables.

5.1. Scripts.

La programación orientada a objetos y el desarrollo de componentes reusables es solo parte de la solución para lograr el ´armado´ de aplicaciones. Una aplicación puede ser vista como una colección de componentes reusables, que satisfacen ciertos standares de interfaces, y se interconectan a través de **scripts** para realizar cierta tarea. La técnica de scripting es una propuesta para el desarrollo de aplicaciones orientado a componentes en la cual frameworks de componentes reusables (objetos y scripts) son cuidadosamente estructurados en un ciclo de vida de soft cuyo principal objetivo es soportar la construcción de aplicaciones a partir de estas componentes prefabricadas. La actividad de construir la aplicación ejecutable es soportada por una herramienta de scripting visual que reemplaza el paradigma textual de programación y composición (muy complicado en ciertos casos, tales como la propuesta de Joseph A. Goguen, LIL -Library Interconnection Language- en "Reusing and Interconecting Software Components") por un paradigma visual con manipulación directa de componentes. Una herramienta de scripting visual provee un mecanismo interactivo para interconectar y editar componentes de software gráficamente representadas. Ayuda al desarrollador de soft en la construcción y definición tanto del comportamiento interno de la aplicación como así también de la interface de usuario a partir de componentes de software prefabricadas.

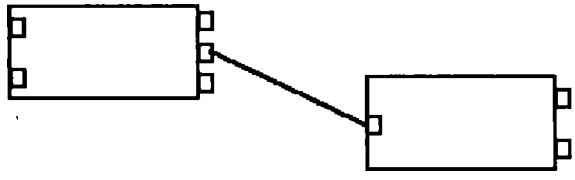
Los scripts proveen el ´pegamento´ que interconecta componentes de software compatibles. Una componente de soft es vista como una pieza que importa o exporta servicios. Los objetos son componentes porque exportan servicios (a través de sus métodos) y los importan desde servidores externos de los cuales son clientes. Las clases son también componentes de software, ya que proveen el servicio de crear instancias de objetos e importan servicios de sus superclases. Las componentes serán adecuadas para la interconección si los servicios requeridos y ofrecidos son compatibles y las componentes son correctas aún antes de ser interconectadas. Estas dos propiedades pueden entonces ser explotadas en un script que especifica las conecciones entre componentes compatibles.

Consideremos por el momento a los scripts como entidades puramente sintácticas. No necesitaríamos conocer nada acerca de los servicios de una componente más que los nombres a través de los cuales ésta puede ser interconectada: los **ports** de la componente. Habrá ports de entrada (cada uno representando un conjunto de servicios disponibles en una componente) y ports de

salida (representando servicios requeridos a otras componentes).

En la figura representamos una componente con un port de entrada y dos ports de salida.

Un script especifica las interconecciones entre un conjunto de componentes. Un link entre el port de salida de una componente y el port de entrada de otra significa que la primer componente utiliza un servicio provisto por la segunda. En la figura representamos dos componentes interconectadas:

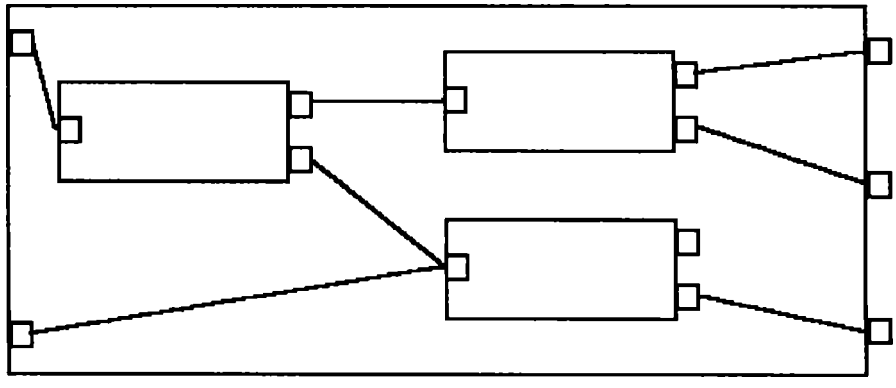


Aparte de la restricción de que los ports de salida sólo pueden ser conectados a ports de entrada, aún no hemos definido ninguna restricción acerca de la sintaxis de los scripts que podemos definir de esta manera. ¿Existen diferentes clases de ports y links? , ¿Podemos interconectar varios ports de salida a uno de entrada o viceversa? , ¿Puede el grafo de componentes contener ciclos?. Estas preguntas no tienen una respuesta en términos absolutos, pero deben ser consideradas en base a cada framework de componentes particular. El conjunto de componentes a interconectar conforman el modelo de script, y formaliza tanto la sintaxis del script como su interpretación. Consideremos algunas reglas sintácticas que podríamos imponer a los scripts:

-Se podrían definir distintos **tipos** de ports, cada uno representando un conjunto diferente de servicios. Sólo ports de entrada y salida **compatibles** podrían ser conectados. Dado que un tipo de port representaría un subconjunto de servicios

provistos por otra componente, las reglas de polimorfismo standard podrían ser aplicadas.
-Se podrían combinar ports con el objeto de combinar conjuntos de servicios. También sería posible definir ports bidireccionales, en cuyo caso, la interconección representaría una relación cliente servidor de ida y vuelta.
-Dado que los ports de entrada representan disponibilidad de servicios, normalmente se encontrarían conectados a múltiples ports de salida; es decir, a múltiples clientes. Llamaremos **multiports** a los ports que permiten múltiples links. Ciertos servicios, sin embargo, funcionarán sólo para un único cliente, en cuyo caso, uno podría restringir el uso de un port de entrada definiéndolo como **simple**. Dado que los ports de salida representan el uso de servicios, generalmente serán simples. Sin embargo, en el caso en que una componente itera sobre un conjunto de servidores, puede resultar útil definir un multiport de salida.
-La interconección de un port puede ser opcional u obligatoria. Los ports de salida en general deberán ser conectados a servidores para completar el comportamiento de la componente, en tanto que los ports de entrada pueden quedar libres, representando servicios disponibles que no serán requeridos. Podríamos definir además links por defecto para determinados ports.

Hay dos factores más a considerar. El primero es que a través de la definición de un script, un conjunto de componentes puede pasar a conformar una nueva componente de mayor complejidad. Es posible encapsular un script en forma de componente con el objeto de reutilizarla en futuros scripts:



Un script componente encapsula un conjunto de componentes parcialmente linkeados y determina qué ports serán visibles desde el exterior. Notemos que los links de exportación/importación simplemente renombran ports, no implican el pedido real de un servicio desde una componente a otra.

Una componente de este tipo puede también contener **huecos**, a ser llenados con componentes que satisfagan ciertas condiciones (por ejemplo,

componentes con determinados ports de entrada, capaces de brindar servicios de cierto tipo).

El segundo factor a tener en cuenta es que los scripts (y las componentes) pueden tener múltiples vistas. No nos referimos sólo a la visión encapsulada o expandida de un script, sino también la forma en que los distintos tipos de componentes, links y ports son visualmente presentados. Esta es la clave para una herramienta de scripting visual que soporte especificación interactiva e interpretación de scripts: las componentes mejor representadas como grafos pueden ser vistas como tales, en tanto componentes más adecuadas para otro tipo de representación, como las componentes para interface de usuario deberían ser representadas de manera más natural.

Análogamente, los links pueden ser vistos como líneas o flechas, pero en ciertos casos la relación puede ser representada más naturalmente como el posicionamiento de una componente sobre otra, la conección estilo rompecabezas, etc. Explotando adecuadamente el potencial de las vistas múltiples para un script, una herramienta podrá soportar manipulación directa de componentes y aplicaciones y controlar la complejidad, reduciendo la necesidad de nombres textuales, mostrando selectivamente sólo los aspectos de una aplicación que son de interés en determinadas circunstancias.

Hasta aquí nos hemos concentrado en los aspectos sintácticos de los scripts: su representación visual y las reglas para componerlos. La interpretación de un script, es decir su semántica, es igualmente importante, ya que es el punto en el cual se establece la conección entre los scripts y los objetos, completando de este modo la definición del **modelo de script**. El modelo de script actúa, además como un puente entre el desarrollador de aplicaciones y el de componentes.

Los links que interconectan componentes tienen dos semánticas: una para la interpretación de script, y otra para las componentes que hacen uso de esos links. Desde el punto de vista de la interpretación del script, un link entre dos componentes nos indicará que una de las partes hará uso de los servicios brindados por la otra. Desde el punto de vista de las componentes, un link le indica a la componente cliente a qué objeto y a través de qué método debe requerir determinado servicio.

Hoy existen en el mercado una serie de herramientas para desarrollo de aplicaciones a partir de componentes que utilizan esta técnica de composición.

Las ideas principales de esta propuesta: visualización y manipulación directa de componentes, scripting tanto del modelo como de la interface de usuario de una aplicación, tipos de links y ports predefinidos, y utilización de scripts como componentes han sido muy bien logradas en productos como Parts y Visual Age. A pesar de esto, el trabajo con estas herraminetas nos demuestra que el ambiente de composición visual no es lo sufucientemente poderoso como para soportar la construcción de aplicaciones medianas o grandes. El abuso de scripts gráficos nos lleva a resultados poco claros e incluso a aplicaciones con tiempos de respuesta pobres. La solución consiste en combinar la técnica de script visual con un

lenguaje de script textual (En el caso de Visual Age esto se logra a través de Smalltalk o C++). Esto clarifica el código de la aplicación facilitando su mantenimiento. En el capítulo VI presentamos algunos aspectos importantes de estas herramientas y estudiamos la implementación real de los links de un script en la composición de una aplicación.

5.2. Conciliando Diferencias de Interfaces.

Existe una propuesta que intenta resolver eventuales incompatibilidades en las interfaces de dos componentes que se interconectan en un script. Hasta aquí hemos visto a la actividad de composición como una actividad de ´matching´ de funcionalidades entre las componentes que requieren un servicio y aquellas que lo proveen. Esto implica el establecimiento de un protocolo standard que deberá ser respetado en forma sistemática. De este modo, el desarrollador de componentes debe proveer interfaces que matcheen exactamente con las esperadas por las componentes existentes. Si bien esto no es malo, la existencia de una entidad mediadora, encargada de conciliar diferencias de interfaces y establecer el link entre dos componentes brindaría cierta flexibilidad en la comunicación.

El concepto de un objeto mediador en la comunicación (denominado gluón por el autor de la propuesta, X. Pintado), tiene otros aspectos interesantes. Estas entidades podrían ser utilizadas, por ejemplo para soportar interoperabilidad entre sistemas, asumiendo la responsabilidad de realizar las conversiones necesarias sin necesidad de modificar componentes incompatibles. Otro dominio de utilidad para estos objetos mediadores es el de las aplicaciones distribuídas o el de auqellos que requieren acceso controlado a componentes y recursos.

El **gluon** sería responsable de todo intercambio de servicios entre componentes de software, conocería la forma de requerir un servicio y manejaría los parámetros involucrados. Toda invocación de un servicio se haría al gluón correspondiente, de modo que dos componentes no se interconectarían directamente.

Hoy no existe en el mercado ninguna herramienta basada en este concepto. Si bien la propuesta es interesante se vuelve poco práctica en el momento de la implementación debido al overhead generado por la sobrecarga en la comunicación entre objetos y la réplica de información.

CAPITULO VI: Estudio de Herramientas Para Desarrollo de Aplicaciones a Partir de Componentes Prefabricadas.

6.1 Herramientas para Composición.

El utilizar una tecnología orientada al reuso tiene como objetivo incrementar la productividad de una organización dedicada al desarrollo de soft. Este tipo de tecnología de desarrollo implica el uso de técnicas y herramientas especializadas, desde el análisis hasta el momento de la programación. En este capítulo analizamos algunas de las herramientas disponibles en el mercado para cubrir la etapa de composición de aplicaciones. Existen tres puntos de fundamental importancia que una herramienta para composición debería cubrir: a) Contar con un lenguaje de scripting, b) Proveer un ambiente de programación visual y c) Soportar el Paradigma de Objetos. En los siguientes párrafos analizaremos estos temas para cada una de las herramientas estudiadas.

6.1.1 Visual Basic.

El aporte de Visual Basic, un producto para programación visual lanzado por Microsoft, fue proporcionar masivamente la posibilidad de reutilización. Es una herramienta de fácil utilización, corre sobre Windows y provee manipulación directa de componentes a través de una interface gráfica clara y sencilla de manejar. Por otro lado posee un lenguaje de scripting similar a Basic. Visual Basic permite desarrollar aplicaciones de escasa complejidad en poco tiempo.

Una barra de herramientas con 16 controles en la versión 1.0 permite seleccionar componentes (fundamentalmente de interface) para componer una aplicación. Cuando se lanzó la versión 2.0, surgieron controles (componentes) desarrollados por terceras partes que pueden adquirirse por separado. De este modo, la barra de herramientas llena la pantalla con un denso mosaico compuesto por todos los controles que la memoria y el disco de la máquina admitan.

Los controles de Visual Basic vienen en formato VBX, y pueden ser considerados como segmentos de software reusable hasta cierto punto; un VBX no puede ser modificado para adaptarse a las necesidades de una aplicación particular, y el grado de customización que proveen, en general, es pobre. Sin embargo, hay que reconocer el acceso masivo que brindaron los VBX a componentes de soft prefabricadas y reusables. Los VBX no sólo son botones, agujas y grillas. National Instruments (Austin, TX) está incluyendo un VBX que controla instrumentos GPIB (bus de interface de propósitos generales). Cimflex Teknowledge (Palo Alto, CA) ofrece un sistema experto basado en VBX. Distinct

(Saratoga, CA) empaqueta su kit de programación TCP/IP en un VBX. Diamond Head Software (Honolulu, HI) ofrece un conjunto de VBX para manejo de imágenes. Stylus Innovation (Cambridge, MA) vende un producto de estas características que se utiliza para crear aplicaciones de respuesta de voz y manejo de fax. Todas estas son componentes reales de venta masiva que se pueden utilizar para crear aplicaciones pequeñas y medianas en poco tiempo.

Sin embargo, manteniendo la coherencia con el entorno Windows creado por Microsoft, las VBX no son objetos. La POO se basa en la trilogía de herencia, polimorfismo y encapsulamiento. Las componentes que permite utilizar Visual Basic sólo cuentan con encapsulamiento. De este modo, pierde las ventajas inherentes al paradigma de objetos en cuanto a reusabilidad y modificabilidad del código. Por otro lado, el desarrollo de una VBX no es trivial, lo cual dificulta la creación de nuestras propias componentes. A esto se agrega el hecho de que, a medida que crece la aplicación, un desarrollo Visual Basic (orientado a eventos) va perdiendo claridad y el mantenimiento se hace dificultoso.

En cuanto a la herramienta en sí, un punto débil (que encontraremos en todas las herramientas analizadas) es el relacionado con la recuperación y estudio de componentes. Recuperar una VBX a partir de una barra con 100 controles no es lo óptimo, además de la limitación en cuanto al número de componentes con que es posible trabajar. VisualBasic puede resultar de utilidad para un programador individual desarrollando aplicaciones chicas.

A pesar de su éxito, VBX es una arquitectura de componentes deficiente. Lo más notorio es que está absolutamente ligada a Windows y a Visual Basic, y a las limitaciones inherentes a Windows (al menos 3.x) en cuanto a segmentación, multitarea cooperativa y fragilidad. Otra crítica es que el límite entre la arquitectura de VBX y su entorno es demasiado rígido. El atractivo de la tecnología de objetos reales es que se puede modificar una componente que realiza el 90% de lo que se necesita y para ello sólo debe agregar el 10%. Es ridículo que los programadores no puedan ampliar los VBX de este modo.

6.1.2 PowerSoft PowerBuilder

PowerBuilder es una herramienta para construcción de aplicaciones a partir de componentes. Brinda un entorno gráfico basado en ventanas. Asume la presencia de productos instalados para Networking y Data Servers. Utiliza SQL para acceder y manipular recursos de Bases de Datos. Además incorpora los recursos de desarrollo e integración del entorno Windows, incluyendo DDE(Dynamic Data Exchange), DLL (Dynamic Link Library) y OLE (Object Linking and Embedding). La connección de Network client-server es manejada por las facilidades del RDBMS. Provee soporte para muchos de los RDBMS conocidos, incluyendo SQL server, Gupta SQLBase, Sybase SQL Server, Oracle, Allbase/SQL, XDB, Informix On-Line y DB2 via Database Gateway de Micro

Decision Ware. Los datos o procedimientos pueden ser accedidos a través de requerimientos SQL especificados en scripts o generados gráficamente a través de objetos Data Window.

PowerBuilder soporta workstations Windows del lado del cliente. Entre las facilidades que provee para el armado de aplicaciones podemos mencionar una barra de herramientas para la creación de interfaces GUI, el lenguaje PowerScript y una librería de objetos. Su arquitectura, llamada CODE (Client/Server Open Development Environment) provee mecanismos para expandir las facilidades de la herramienta.

El ambiente de desarrollo se compone de tres partes: Painters, Objects y el Lenguaje de Scripting. Para crear una aplicación, el programador crea una serie de ventanas, que interactúan con objetos y controles. Los objetos son herramientas para el desarrollo de aplicaciones que han sido iconificadas. Los Painters son herramientas gráficas WYSIWYG (What you see is what you get) que permiten realizar el desarrollo, al crear, manejar y mantener las ventanas de la aplicación, los objetos y los scripts. El Application Painter define el ambiente, incluyendo el look and feel y las estructuras de datos a ser accedidas por la aplicación en desarrollo. El Window Painter permite construir ventanas de interface, incluyendo controles y menúes que pueden ser linkeados estática o dinámicamente. El Data Window Painter se utiliza para presentar y manipular datos residentes en bases de datos relacionales. La Structure Painter define las estructuras del programa (arreglos, collecciones, etc.). La Library Painter crea y mantiene librerías que son el repositorio para aplicaciones y objetos PowerBuilder. El Function Painter permite escribir funciones definidas por el usuario para extender el lenguaje de script. El User Object Painter crea objetos definidos por el usuario, incluyendo atributos, scripts y eventos asociados con ellos. El Database Painter se utiliza para crear tablas y vistas de una base de datos. Todas estas herramientas permiten la construcción de una aplicación a partir de componentes almacenadas, en un ambiente gráfico que permite manipulación directa.

A pesar de no ser realmente orientado a objetos, PowerBuilder hace un esfuerzo por brindar algunos beneficios importantes del paradigma. Los programadores customizan sus aplicaciones utilizando atributos, eventos y scripts que se asocian a objetos y controles. Sin embargo, el producto no permite una programación orientada a objetos para la lógica de la aplicación. Esto significa que no es posible un análisis, modelo y diseño orientado a objetos. Por esto, y a pesar de la terminología de objetos, PowerBuilder no proporciona los beneficios de modelar las particularidades de un negocio como una serie de objetos. A pesar de esto brinda cierto grado de reusabilidad y la posibilidad de utilizar herencia (simple).

La Library Painter permite crear y mantener librerías. Se proveen funciones para crear, borrar, organizar y mantenerlas, estudiar, exportar, importar y regenerar objetos. Si bien estas características son adecuadas para proyectos pequeños, otras capacidades más sofisticadas como versionamiento y seguridad para

aplicaciones en gran escala pueden ser adquiridas a través de la integración con otros productos. (Se ha llegado a un acuerdo con LBMS, Inc., una empresa desarrolladora de herramientas CASE, para proveer estas mejoras).

Al igual que Visual-Basic, PowerBuilder pierde muchas de las ventajas del paradigma de Orientación a Objetos en cuanto a reusabilidad y no cuenta con un mecanismo adecuado para recuperación de componentes.

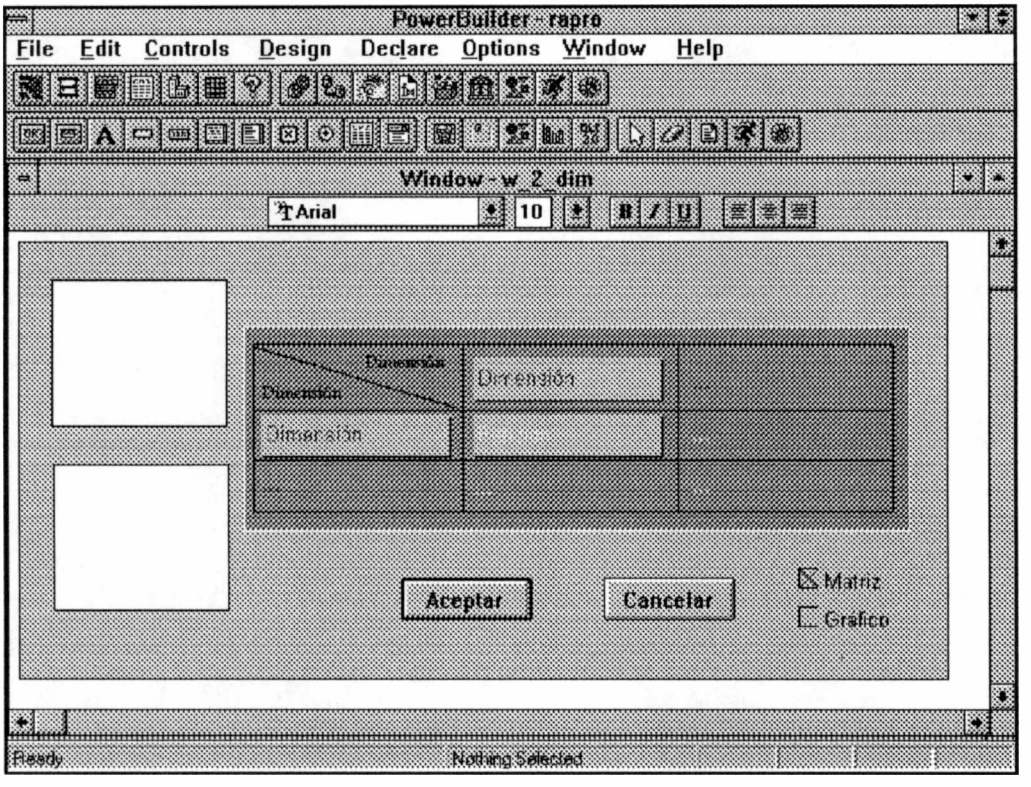


Figure 4 Ambiente de composición de PowerBuilder.

6.1.3. ENVY/400

ENVY/400 es un ambiente de desarrollo Smalltalk para equipos de desarrolladores. Sus aplicaciones corren tanto en Windows como en OS/2, y acceden a los recursos y servicios de un server AS/400. Sin embargo, no provee facilidades de desarrollo para la lógica de la aplicación sobre la AS/400. En esencia, ENVY/400 es Smalltalk con extensiones que hacen posible el desarrollo en equipo y pone a disposición de las aplicaciones clientes los recursos de una AS/400. Históricamente, Smalltalk ha sido limitado a desarrolladores individuales por requerir que desarrolladores que trabajan aún en una parte pequeña de una

aplicación, deban contar con una parte significativa de la lógica total en sus máquinas. IBM ha sobrepuesto esta limitación proveyendo un repositorio basado en AS/400 con capacidades de versionamiento y control de componentes.

ENVY/400 provee muchas componentes Smalltalk prefabricadas y algunas capacidades gráficas para los desarrolladores. Sin embargo, un desarrollador debe aprender Smalltalk para explotar las posibilidades de la herramienta. Para staffs con fuerte arraigo procedural la inversión en entrenamiento puede ser considerable.

Las comunicaciones se logran a través de llamadas a programas remotos que permiten al cliente comunicarse con programas en la AS/400. En tiempo de ejecución, las aplicaciones ENVY/400 usan agentes para interceptar y rutear llamadas a funciones a través del mecanismo de RPC.

Existen librerías de clases especializadas que contienen objetos para acceder a bases de datos en AS/400. Los archivos basados en AS/400 (incluyendo bases de datos, secuenciales, indexados y directos) son soportados a través de componentes DDM (Distributed Data Management). Además provee acceso a bases de datos relacionales a través del DRDA de IBM (Distributed Relational Database Architecture).

El producto consta de 5 componentes:

A) Visual Program Generator. Es una herramienta para diseño de interface de usuarios que permite a los desarrolladores seleccionar, modificar y utilizar controles CUA (Common User Access, el standard para GUI de IBM) a partir de una librería portable de componentes GUI. Se dispone de más de 20 controles (check boxes, push buttons, scroll bars, etc.). Cada uno posee un esquema de propiedades para guiar al programador a través del proceso de customización del look and feel y comportamiento del control.

B) Lenguaje de Programación Smalltalk. Es el vehículo para construir aplicaciones en ENVY/400. Una aplicación Smalltalk se compone de objetos reusables que interactúan para realizar las tareas del programa. Los objetos tienen atributos (o características) y pueden tener tanto datos como métodos para realizar operaciones. Los objetos utilizan mensajes para comunicarse entre sí, transmitirse datos, determinar los métodos de otros objetos y requerir servicios. Una aplicación Smalltalk en ejecución es una colección de objetos encapsulados ejecutando procedimientos que se activan a través de mensajes. Todas las construcciones de programación y los elementos de datos son objetos que utilizan la herencia de clases, que le permiten a instancias (objetos) de clases hijas heredar atributos y capacidades definidas en clases ancestras. Además de los beneficios del reuso de objetos, Smalltalk reduce la complejidad de programar en un ambiente multiplataforma. Por ejemplo, en ENVY/400, un objeto layer se encarga de proveer los servicios de comunicación, interceptando un mensaje de requerimiento, dirigiéndolo al server apropiado y retornando el resultado a la aplicación. ENVY/400 incluye numerosos browsers especializados que proveen diferentes vistas del repositorio. Los desarrolladores pueden usar estos browsers para

acceder y modificar porciones de una aplicación.
C) Librería de Componentes AS/400. Es un conjunto de objetos que permiten a las aplicaciones clientes acceder a los servicios de una AS/400, objetos y datos. ENVY/400 utiliza agentes y brokers para extender los servicios de la AS/400 a los clientes. Los agentes interceptan los mensajes y requerimientos de los objetos en las aplicaciones y manejan el acceso a la AS/400. El efecto es una reducción importantísima en la complejidad de comunicación. Los objetos en la aplicación cliente no necesitan hablar directamente con los recursos de la AS/400. Los brokers crean agentes en respuesta a los requerimientos de servicios por parte de las aplicaciones y controlan el funcionamiento de los agentes creados.
D) Team Development Facilities. Para facilitar el desarrollo en equipo, ENVY/400 proporciona un repositorio en AS/400 basado en Smalltalk. El repositorio maneja múltiples ediciones de aplicación, permitiendo a varios desarrolladores acceder al código de la misma aplicación. Maneja configuración y control de versiones y accesos. Permite control de cambios y comparaciones entre versiones.
E) Application Packaging. ENVY/400 incluye un editor de configuración para especificar versiones runtime. Utiliza una utilidad de packaging para separar clases y métodos sin relevancia para la aplicación ejecutable como así también el compilador y el código fuente. Por otro lado permite llevar una aplicación a diferentes plataformas.

Esta herramienta, pensada para desarrollar aplicaciones medianas y grandes, maximiza las posibilidades de reuso y provee facilidades para composición de aplicaciones a partir de componentes Smalltalk, por lo que ofrece un ambiente de desarrollo Orientado a Objetos puro. Dejando de lado cuestiones económicas, se presenta como una de las alternativas más interesantes, cubriendo puntos de fundamental relevancia en una metodología de desarrollo orientada al reuso.

6.1.4 Visual Age.

Visual Age es un ambiente de desarrollo de aplicaciones visual, de muy fácil manejo, totalmente orientado a objetos. La herramienta permite desarrollar rápidamente aplicaciones Cliente-Servidor para OS/2 y Windows. Incluye un poderoso editor visual, para composición de componentes, una extensa librería de componentes prefabricadas, tanto para interface gráfica de usuario como para programar la lógica de las aplicaciones, un lenguaje de Scripting poderoso (Smalltalk o C++, según la versión), browsers de clases y componentes, debuggers, componentes para acceso a bases de datos relacionales (DB2/2, DB2/6000, SQL/400, Oracle), soporte para integrar programas preexistentes escritos en otros lenguajes, total interacción con programas C, soporte para desarrollo en equipo con control de versiones, utilidades para packaging, y componentes para multimedia y comunicaciones (usando TCP/IP, NetBios, CPIC,

APPC en OS/2 y CICS OS/2).

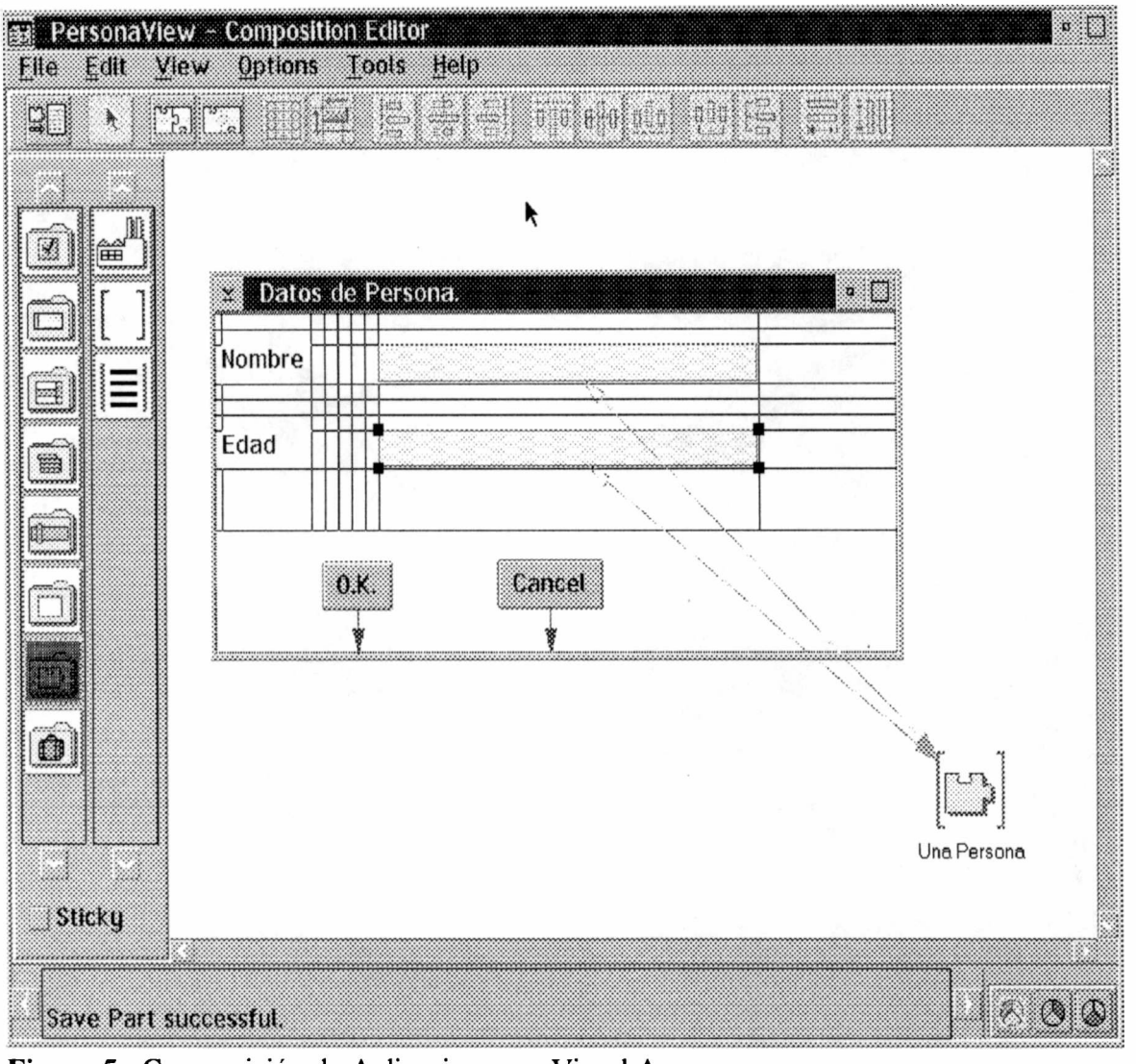


Figure 5 Composición de Aplicaciones en Visual Age.

Con Visual Age es posible desarrollar aplicaciones a través de la conección gráfica de partes prefabricadas. Por otro lado ofrece una ventaja importante que es la posibilidad de extender la capacidad creando nuevas partes y agregándolas a la paleta de componentes reusables. Las aplicaciones pueden extenderse codificando nuevas clases (en Smalltalk o C++) o extendiendo las existentes.

Visual Age es un ambiente de desarrollo orientado a objetos puro, con lo que obtenemos la posibilidad de crear partes reusables heredando de las

existentes. Soporta encapsulamiento, polimorfismo y herencia.

Hemos comprobado que permite no sólo el desarrollo rápido de aplicaciones simples, sino también el desarrollo incremental de aplicaciones medianas y grandes.

Los puntos débiles encontrados son, por un lado la selección de componentes a utilizar, y por el otro la implementación real que hace de los links entre componentes a nivel de código. La selección de componentes se realiza a partir de una paleta en la que las partes reusables se dividen en categorías (Ventanas, Botones, Partes para BD, etc.) Si bien es posible extender esta paleta, no es una forma de browsing adecuada cuando se trabaja en un ambiente de reuso a gran escala, en el que el número de componentes es grande. El otro punto implica que una componente, para poder ser utilizada en el ambiente de composición, debe reunir ciertas características que nada tienen que ver con su funcionalidad. En el ejemplo que mostramos, una ventana que muestra los datos de una persona, para poder ser compuesta debe almacenar en variables de instancia las conecciones que se establezcan. Esto no es grave cuando se trata de una componente integrada al producto, pero se hace molesto cuando pretendemos crear nuestras propias partes, sobre todo en el caso de Visual Age C++. En el caso de la versión Smalltalk es más sencillo, ya que basta con que nuestras clases hereden de una clase particular (IPart) para poder ser manipulada directamente a través del Composition Editor.

```
/*****************************************************************************
* FILE NAME: PrsnView.hpp                                                    *
*                                                                            *
* DESCRIPTION:                                                               *
*   Declaration of the class:                                                *
*     PersonaView                                                            *
* -------------------------------------------------------------------------- *
* Warning: This file was generated by the VisualAge C++ Visual Builder.      *
* Modifications to this source file will be lost when the part is regenerated. *
*****************************************************************************/
#ifndef _PERSONAVIEW_
#define _PERSONAVIEW_

.
.
.

class PersonaViewConn0;
class PersonaViewConn1;
class PersonaViewConn2;
class PersonaViewConn3;


.
.
.

//*****************************************************************************
// Class definition for PersonaView
//*****************************************************************************
class PersonaView : public IFrameWindow {
public:
   //-------------------------------------------------------------------------
```

```
   // Constructors / destructors
   //------------------------------------------------------------------------
   PersonaView(
      unsigned long id = WND_PersonaView,
      IWindow* parent = IWindow::desktopWindow(),
      IWindow* owner = 0,
      const IRectangle& rect = defaultFramingSpec(),
      const IFrameWindow::Style& style = IFrameWindow::defaultStyle ( ),
      const char* title = defaultTitle());

   virtual ~PersonaView();

   //------------------------------------------------------------------------
   // public member functions
   //------------------------------------------------------------------------
   static const IRectangle defaultFramingSpec();
   static IString defaultTitle();
   virtual PersonaView & initializePart();
   PersonaView * getFrameWindow() { return this; };

   //------------------------------------------------------------------------
   // public member data
   //------------------------------------------------------------------------
   static const INotificationId readyId;

protected:
   //------------------------------------------------------------------------
   // protected member functions
   //------------------------------------------------------------------------
   virtual Boolean makeConnections();

private:
   //------------------------------------------------------------------------
   // private member data
   //------------------------------------------------------------------------
   ICanvas * iCanvas;
   IMultiCellCanvas * iMultiCellCanvas1;
   IEntryField * iEntryEdad;
   IEntryField * iEntryNombre;
   IStaticText * iNombre;
   IStaticText * iEdad;
   IPushButton * iButtonOk;
   IPushButton * iButtonCancel;
   IVBVariablePart<Persona> * iUnaPersona;
   PersonaViewConn0 * conn0;
   PersonaViewConn1 * conn1;
   PersonaViewConn2 * conn2;
   PersonaViewConn3 * conn3;


};   //PersonaView


/*--------------------------------------------------------------------------*/
/* Resume compiler default packing.                                         */
/*--------------------------------------------------------------------------*/
#pragma pack()

#endif
```

Mas allá de ese problema, que es posible apreciar gracias a que las componentes son totalmente abiertas, Visual Age, que es muy similar a Parts (producto que utiliza Smalltalk y corre sobre Windows) es la alternativa más atractiva como herramienta de scripting que complemente un proceso de desarrollo de aplicaciones orientado a componentes.

CAPITULO VII : Presentación de una Propuesta para Catálogo, Estudio y Recuperación de Componentes.

7.1 Resumen del Objetivo.

En el capítulo anterior estudiamos varios modelos y herramientas para construcción de aplicaciones a partir de componentes prefabricadas. Hemos visto que a través de la manipulación directa, una interface atractiva y sencilla de manejar y un mecanismo de composición adecuado, muchas de estas herramientas cubren una parte importante de los requerimientos que una infraestructura de reuso debe satisfacer en cuanto a facilidades para el ¨armado¨ de la aplicación a partir de componentes preexistentes. Sin embargo, existen puntos no cubiertos por ninguno de estos productos. ¿Cómo obtenemos estas componentes? , ¿Cómo las mantenemos? , ¿Cómo las adaptamos a las necesidades de nuestra aplicación?

Para trabajar en un entorno de reuso, es necesario resolver cuatro problemas fundamentales: 1) Localización de componentes, 2) Entendimiento de componentes, 3) Alteración de componentes y 4) Composición de componentes. Hemos visto herramientas capaces de solucionar la última de las cuestiones. Estudiaremos los primeros tres problemas y presentaremos una propuesta que intenta resolverlos.

El primer punto débil encontrado es el modo de acceso y recuperación de componentes. Ninguna de las herramientas estudiadas está preparada como para soportar una gran biblioteca de componentes, ni brinda facilidades para el catálogo y recuperación de las mismas. Como consecuencia, no existe un soporte de base de datos adecuado. En todos los casos, el desarrollador debe conocer cuáles son las componentes de las que dispone y recuperar la que sea de su interés a través de un simple listado. Si bien algunas de estas herramientas son adecuadas para el desarrollo de aplicaciones a partir de componentes reusables, carecen de un medio que permita integrar un sistema de información de componentes al proceso de desarrollo.

7.2 Visión del Usuario.

A lo largo de este trabajo hemos estudiado los distintos pasos involucrados en un proceso de desarrollo orientado al reuso. Comenzamos por la realización de análisis de dominios. En este momento ya es posible utilizar una herramienta para catálogo de componentes reusable. Hemos mencionado que el resultado de un análisis de dominios es un conjunto de conocimientos acerca de un dominio de problemas y de cómo implementar soluciones informáticas dentro de ese dominio.

Lo que obtenemos es un modelo del dominio. Este resultado no es más que una componente que deberá ser reutilizada cada vez que se requiera una solución para un problema dentro del dominio analizado, y debería ser catalogada adecuadamente de manera que todos los integrantes de una organización conozcan de su existencia y puedan utilizarla.

Cuando un equipo de desarrollo va más allá del análisis, y llega a diseñar o implementar un conjunto de componentes reusables, éstas deben ser catalogadas para posibilitar su reutilización. El desarrollo de componentes reusables en una organización es una inversión que debe ser recuperada a través del reuso de dichas componentes por parte de la mayor cantidad de miembros de esa organización; y esto sólo es posible si se implementa un mecanismo de catálogo sistemático de cada una de las componentes desarrolladas.

Por último, cada desarrollador que se enfrente con un nuevo proyecto debe acudir al repositorio en busca de componentes que tengan relación con el dominio del problema que se dispone a solucionar. La biblioteca de componentes debe transformarse en uno más de los canales de comunicación propios de cada organización y debería ser objeto de estudio y consulta permanente de los desarrolladores de soft.

7.3 Presentación de la Propuesta.

En este capítulo, presentamos una propuesta para el catálogo, recuperación y estudio de componentes de soft. Esta propuesta es acompañada por un prototipo ejecutable que corre en entorno Windows. En la definición e implementación de la propuesta centraremos nuestra atención en los siguientes puntos:

a) Flexibilidad en el Almacenamiento: Deberá ser posible almacenar componentes reusables tanto a nivel de análisis como de código fuente u objeto. No siempre una componente reusable se traduce en un fragmento de código fuente. Tal como mencionáramos en el capítulo II (Análisis de Dominios), se considerará importante la posibilidad de guardar resultados obtenidos en la identificación, adquisición y representación de conocimientos, potencialmente reusables, acerca de la especificación e implementación de software para distintos problemas del mundo real, acompañados o no por su implementación.

b) Tecnología de Orientación a Objetos: Trabajaremos basándonos en notaciones y técnicas pertenecientes al paradigma de Orientación a Objetos. A lo largo de este trabajo hemos mencionado en más de una oportunidad las ventajas involucradas en esta tecnología para el desarrollo de software orientado al reuso.

c) Facilidad de Integración: Teniendo en cuenta que en una organización dedicada al desarrollo de soft encontraremos diversas arquitecturas, lenguajes, y entornos

de desarrollo y ejecución, el catálogo deberá independizarse de estas cuestiones permitiendo de este modo una mayor integración entre los miembros de la organización y entre el sistema de información y los ambientes individuales de desarrollo. Se deberá permitir el acceso a código implementado en distintos lenguajes y preparado para correr en distintos ambientes.

d) Facilidad para el Estudio de Componentes Almacenadas: En el capítulo III presentamos el desarrollo real de una aplicación utilizando una metodología orientada al reuso. Este caso, así como la experiencia personal, nos muestra que un alto porcentaje de las componentes recuperadas con la intención de ser reusadas necesitan antes de una adaptación a los requerimientos de nuestra aplicación. En muy pocas oportunidades nos bastará con estudiar la interface de una componente para utilizarla. En general, y sobre todo cuando se trate de componentes no triviales, la modificación es indispensable. En este sentido existen dos tipos de modificaciones: extensivas o adaptativas. Las modificaciones extensivas amplían la funcionalidad de la componente o generalizan su uso. Ocurren cuando una componente necesita mayor funcionalidad o no es lo suficientemente general como para satisfacer los requerimientos de nuestra aplicación. Este tipo de modificaciones enriquecen a la componente, debiéndose recatalogar la nueva versión en el sistema de información. Por otro lado las modificaciones adaptativas 'pegan' a la componente particularidades propias (pero necesarias) de nuestra aplicación. Generalmente este tipo de modificaciones quitan reusabilidad a la componente y deberían ser evitadas. La calidad de la componente y del diseño de nuestra aplicación hacen que este tipo de modificaciones sean o no necesarias.

Para posibilitar la modificación de componentes, el sistema de información debería permitir el estudio de las mismas desde el punto de vista de su funcionamiento, jerarquía de clases que la componen, interacción entre estas clases, comportamiento y estados de la componente, etc. De otro modo, una modificación que enriquezca a la componente sería muy difícil de lograr. En este punto surge un tema delicado: el usuario se transforma en desarrollador de componentes reusables, perdiendo la ventaja de ver a la componente como un módulo encapsulado y debiendo acceder a sus características internas. Sólo deberá ser posible acceder a esta visión de la componente ante la firme convicción de que ésta necesita ser modificada y la aceptación del cambio de rol del usuario que pasa a ser desarrollador de componentes. El sistema de información deberá manejar cuestiones como esta.

e) Proceso de Catálogo Claro y Efectivo: El proceso de catálogo de componentes deberá ser fácil de utilizar. El desarrollador de una componente, a través de una serie de preguntas o formularios de clara lectura será guiado durante el proceso de catálogo.

f) Recuperación Rankeada: Durante el proceso de recuperación es posible que más de una componente satisfaga los parámetros de búsqueda brindados. Es importante que las componentes halladas sean presentadas en forma ordenada de acuerdo al grado de matching con el tipo de componente pedida.

g) Descripción para Composición: La situación ideal consiste en recuperar una componente y acceder de inmediato a la información que permita componerla con el resto de la aplicación de la cual va a formar parte. La descripción principal de una componente debería proveer la información de cómo realizar esta composición.

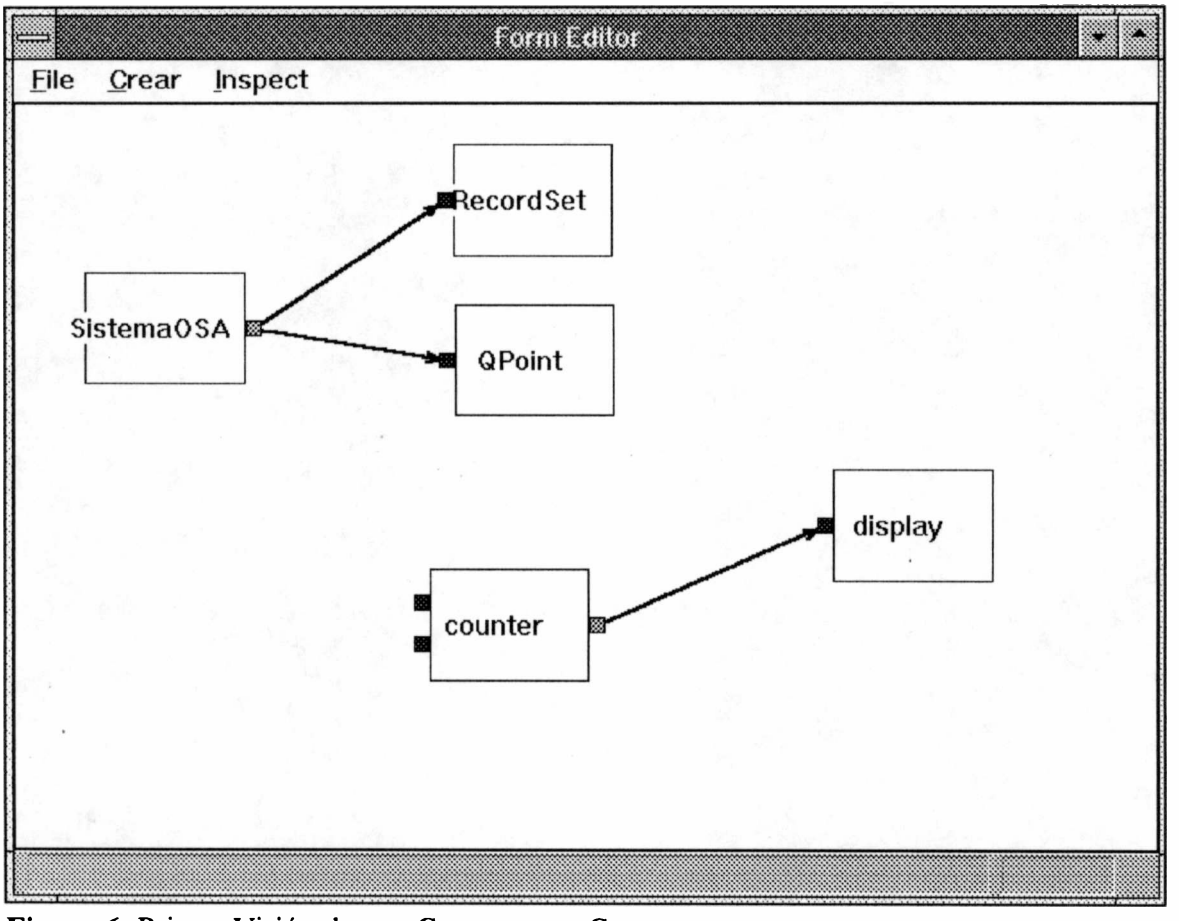


Figure 6. Primer Visión de una Componente Compuesta.

h) Focalización en los procesos de Catálogo, Recuperación y Estudio de Componentes: No se implementará un soporte de Base de Datos Real preparado para el manejo de gran cantidad de componentes, con utilización de índices, claves

de acceso, etc; aunque sí se separa el módulo de almacenamiento de manera que pueda ser implementado mediante la utilización de un DBMS relacional (estamos estudiando el caso de Oracle como una posibilidad factible vía ODBC e incluso hemos implementado una interface entre Smalltalk for Windows, lenguage utilizado en la implementación del prototipo, y Paradox como segunda alternativa relacional) o a través de un DBMS Orientado a Objetos (analizamos Object Store).

I) Manipulación directa de objetos: Tal como lo hacen las herramientas para composición, se deberán proveer facilidades de interface gráfica y manipulación directa de componentes y clases durante el proceso de catálogo.

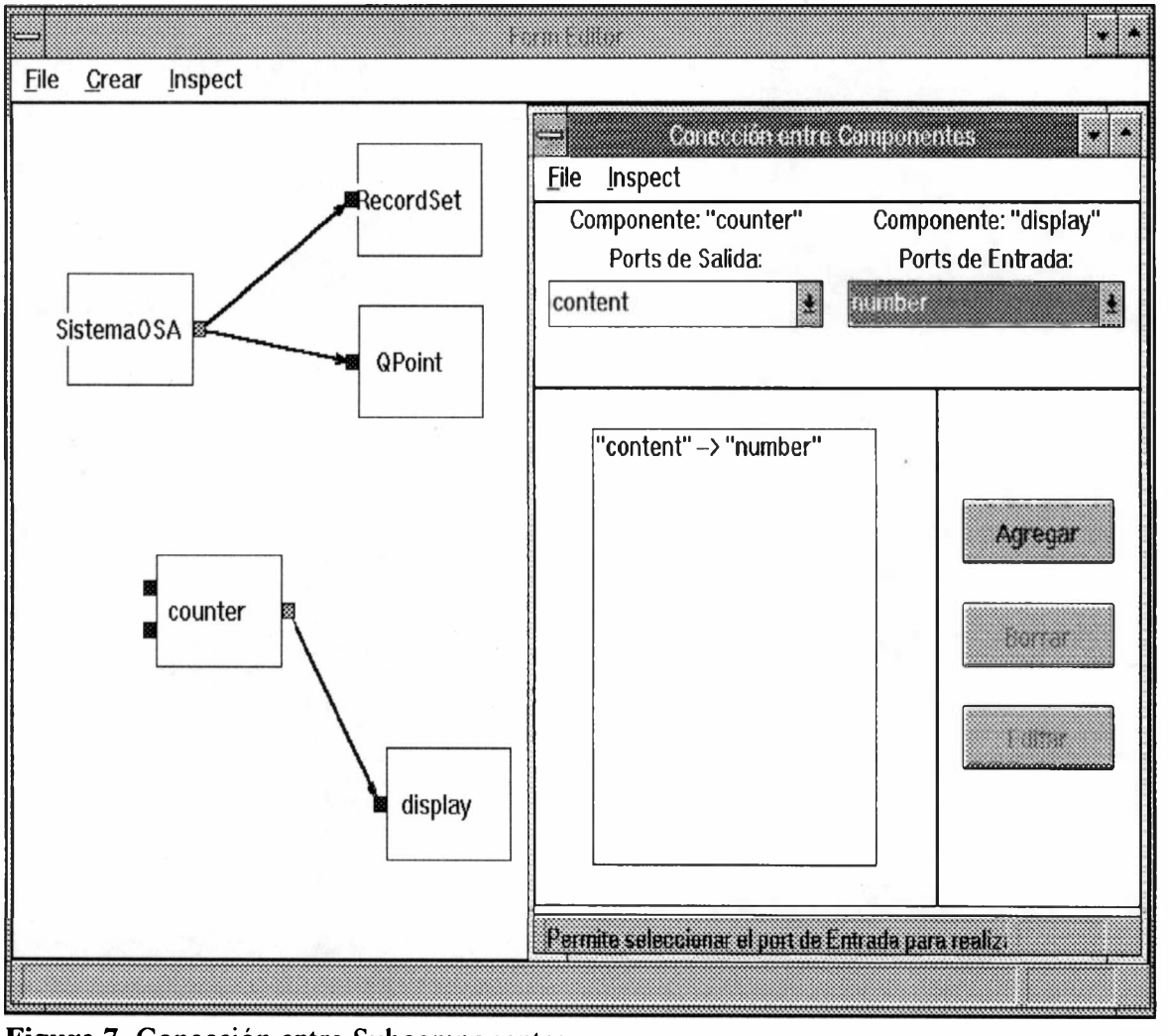


Figure 7. Conección entre Subcomponentes.

j) Vocabulario consistente: Se establecerá un vocabulario básico (extensible) para los términos involucrados en los procesos de catálogo y recuperación de componentes, como así también un mecanismo para el manejo de sinónimos.

7.3.1 Principios de Clasificación.

Para resolver el problema de catálogo y recuperación de componentes, nos basaremos en un esquema denominado de Clasificación. El esquema de clasificación es la base de la accesibilidad a las componentes almacenadas en el sistema de información y estará basado en los atributos de reutilización de dichas componentes y en un mecanismo de selección automática. La clasificación del conjunto de componentes almacenadas en una biblioteca permitirá reducir el esfuerzo necesario en ubicar, entender y adaptar una componente. La biblioteca se organiza de acuerdo a atributos que definen en forma unívoca los requisitos de una componente buscada, reduciendo la probabilidad de recuperar componentes no relevantes para el problema que se intenta resolver.

El esquema de clasificación se basa en la suposición de que las componentes nunca ¨matchean¨ los requisitos perfectamente, tornándose siempre necesaria algún tipo de adaptación. De esta manera, el esquema supone la existencia de un ambiente apropiado que ayude a localizar componentes y a estimar de alguna manera el esfuerzo necesario en la adaptación.

El proceso de reutilización tiene en cuenta la existencia de un conjunto de especificaciones provistas por el programador de componentes o analista de dominios que definen los atributos que podrán ser exigidos a las componentes que serán recuperadas de la biblioteca para ´armar´ una aplicación.

De este modo, si existen componentes que satisfagan todas las especificaciones, la recuperación se torna trivial; si existen varias candidatas, cada una de las cuales satisface algunas especificaciones, el problema consiste en hacer una selección y clasificación de las componentes disponibles, basada en el grado de matching con los requisitos por orden de relevancia. Una vez seleccionada una lista ordenada de las componentes candidatas, se deberá facilitar al usuario el estudio de las mismas para reconocer las adaptaciones que será necesario realizar y seleccionar la que más se adecúe a sus necesidades.

La selección de la lista de componentes candidatas es una tarea compleja. El grado de matching con los requerimientos dependerá de cómo el catálogo fue organizado. La organización de la biblioteca de componentes y la selección de los atributos relevantes se tornan en este punto la componente central en el proceso de reutilización.

7.3.2 El esquema de clasificación.

Una clasificación es un agrupamiento de elementos similares que comparten al menos una característica común, que miembros de otros grupos no poseen. Una clasificación muestra así, una relación entre elementos y entre clases de elementos. El resultado es una red de relaciones. De este modo, la definición de un elemento hace referencia a la clase que contiene a dicho objeto y establece diferencias con objetos de otras clases.

Un esquema de clasificación es una herramienta destinada a producir un órden sistemático, basado en un vocabulario de índices, estructurado y controlado. El esquema consiste en un conjunto de símbolos -representando conceptos o clases- sistemáticamente listados, para mostrar las relaciones entre clases.

El esquema de clasificación que utilizaremos es multifacético. Cada ítem a ser clasificado consta de múltiples facetas, cada una de las cuales corresponde a un aspecto en la especificación de la componente. Las facetas pueden ser consideradas como perspectivas, puntos de vista o dimensiones de un dominio particular. Un esquema multifacético tiene varias facetas, cada una de las cuales puede tener varios términos posibles. Las facetas son ordenadas por orden de relevancia por los usuarios.

Otra suposición hecha en este modelo es que las colecciones de componentes reusables son muy grandes y crecen contínuamente, existiendo grupos enormes de componentes similares. El esquema debe ser flexible y extensible, permitiendo la incorporación de nuevas clases a las ya existentes sin grandes dificultades.

Para esto proponemos un formato de descripción de componentes, basado en un vocabulario padre de términos para cada faceta y un orden predeterminado el cuanto a la relevancia de las mismas que puede ser alterado fácilmente por el usuario al realizar una búsqueda.

7.3.4 Formato de descripción de Componentes.

En un primer acercamiento, podemos advertir que la descripción de una componente de software consiste de tres partes: la funcionalidad o ¨qué hace¨ la componente, el ambiente o ¨dónde lo hace¨ y la realización o ¨cómo lo hace¨. El ¨cómo¨ no es incluído en el formato de descripción de componentes, el cual debe ser suscinto como para ser utilizado como clave de recuperación. Sin embargo, al estudiar una componente recuperada, la adaptación requiere de cierta información acerca de ¨cómo¨ funciona, por lo que esta segunda descripción será almacenada también por cada componente en un descriptor especial.

Para describir la funcionalidad y ambiente de una componente de software hemos establecido un conjunto base de facetas y términos. La herramienta que presentamos deberá permitir extender este conjunto de conceptos a medida que la biblioteca de componentes crezca y surjan nuevas necesidades.

7.3.5 Facetas y Términos.

Al catalogar una componente se deberá completar un descriptor seleccionando uno de los términos existentes para cada faceta. Otro punto a tener en cuenta es que puede ocurrir que una componente sea caracterizada por más de un término en una faceta dada. El sistema de información deberá manejar estas cuestiones. En caso de que ningun término se adecúe a la componente a catalogar, es posible introducir nuevos conceptos. De este modo, el sistema de información se enriquecerá a medida que sea utilizado.

La siguiente es una lista de las facetas y términos propuestos, y los conceptos involucrados en cada ítem.

-Tipo: Hace referencia al tipo de componente que se desea catalogar. Desde el punto de vista del tipo, una componente puede pertenecer a las siguientes clases: Interfaces de Usuarios, Estructuras de Datos, Manejadores de E/S en Tiempo Real, Manejadores de Dispositivos Externos, Manejadores de Memoria, Manejadores de Video, Componentes para Cálculo Matemático, Manejador de Sensores para Adquisición de Datos, Módulos para Acceso a Bases de Datos, Manejadores de Discos, Manipuladores de Texto, Módulos para Procesamiento Paralelo, Componentes para Debugging, Graficadores, Módulos para Animación Computada, Componentes Multimediales, Hipertextos, Componentes para Cálculo de Estadísticas, Listadores de Datos, Browsers, etc.

-Area Funcional: Se refiere al tipo de aplicaciones en las cuales la componente puede ser incluída. Seguramente existirán componentes capaces de formar parte de aplicaciones en cualquier área, pero existen componentes típicas de áreas particulares, como por ejemplo, Sistemas de Personal, Sistemas de Gestión de Compras, Planillas de Cálculo Contable, Gestión Comercial, Sistemas de Seguridad, Sistemas Operativos, Procesadores de Textos, Sistemas Recaudadores, Controladores de Tráfico, Sistemas para Manejo de Socios, Manejo de Turnos, Sistemas de Sueldos, Charts, Controladores de Stock, Sistemas de Abastecimiento, Graficadores CAD, etc.

-Situación: Se refiere a la Industria o Negocio dentro del cual es apropiada la componente. Puede ocurrir que una componente con la misma área funcional sea distinta en dos situaciones diferentes. Por ejemplo, una componente útil en CAD en la industria de las construcciones puede ser obsoleta en la industria automotriz. Otros ejemplos de situaciones son Salud, Electrónica, Comercio, Física y Matemática, Publicidad, Medios de Comunicación, Banking, etc.-

-Ambiente: Esta faceta describe el ¨dónde¨ que mencionábamos anteriormente. Es una faceta particular ya que describirá varios puntos: Hardware que utiliza la componente, Sistema Operativo o Entorno, Lenguaje en que está programada en

caso de que se provea el código, Compilador utilizado, Descripción de la forma en que la componente está disponible (fuente-objeto-librería estática o dinámica, etc), Soft adicional que necesita para ser utilizada (protocolos de comunicación, base de datos), etc.

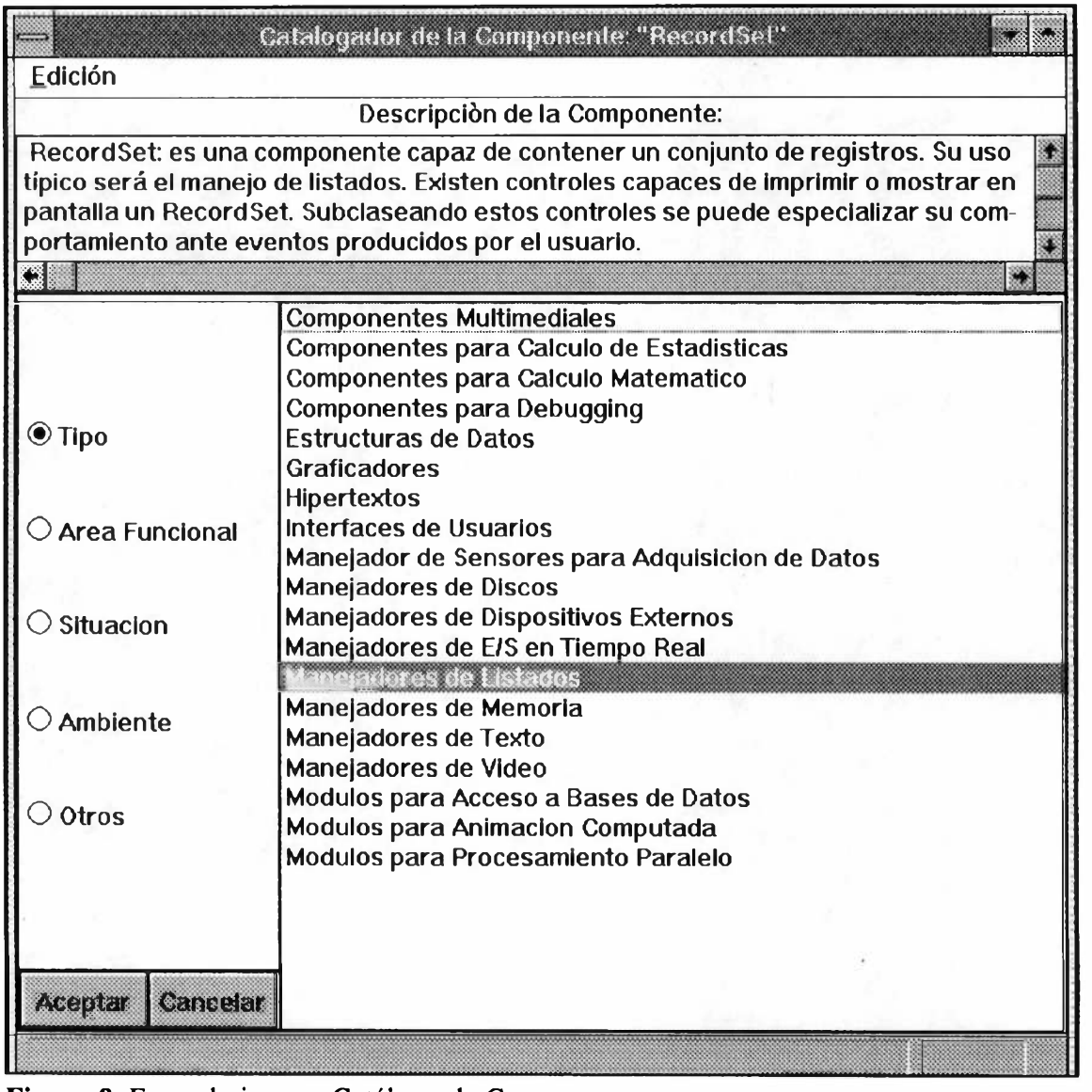


Figure 8. Formulario para Catálogo de Componentes.

7.3.6 Descripción Interna de la Componente.

Hemos mencionado ya la importancia de facilitar el estudio de la componente para, una vez recuperada, permitir su adaptación. Es importante subrayar que este estudio se realiza sólo si se asume que la componente necesita ser adaptada o extendida; y que el acceso a los detalles internos de la misma es inevitable. Para esto, nuestra herramienta permite 'abrir' una componente y acceder a un descriptor adicional.

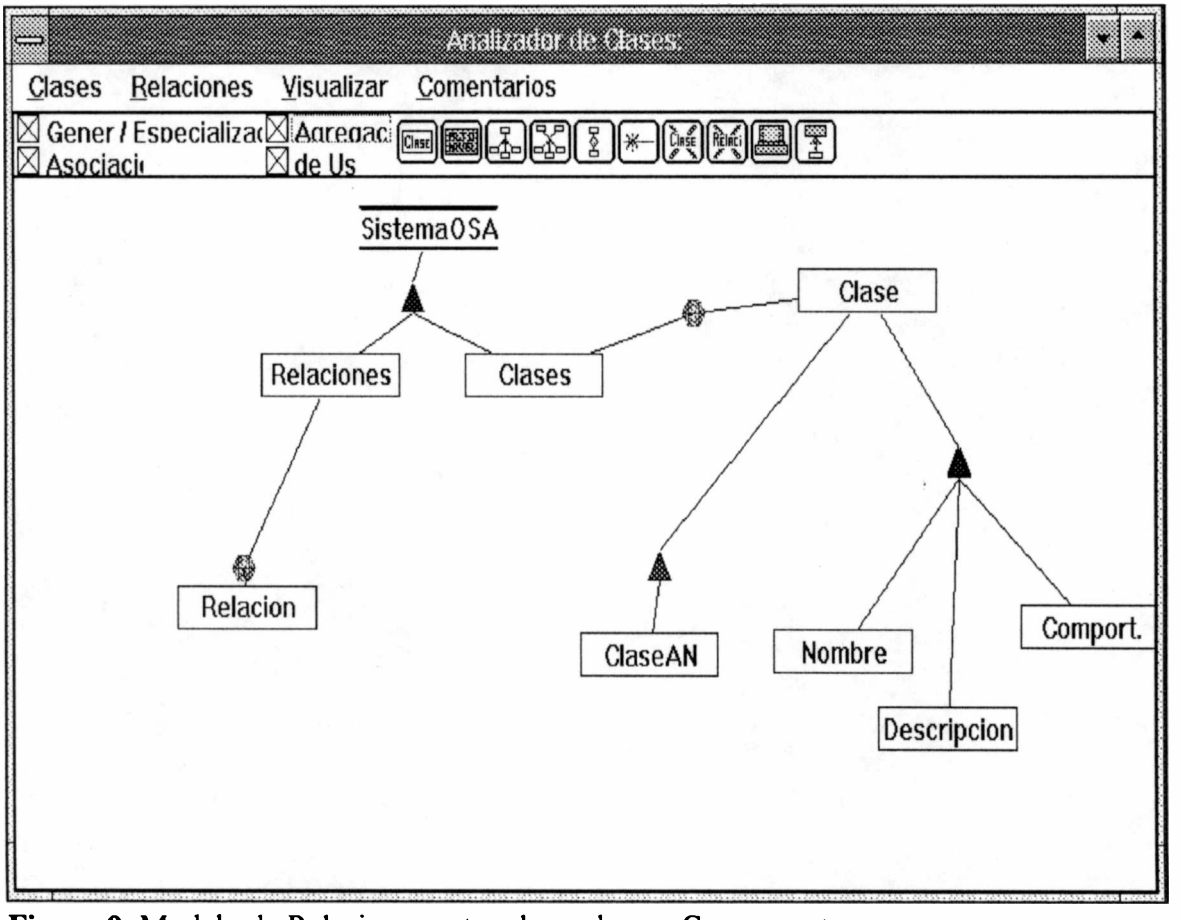


Figure 9. Modelo de Relaciones entre clases de una Componente.

Este descriptor plantea un par de modelos que el analista o programador de componentes deberá completar. El primer modelo es el de Relaciones entre las clases que interactúan dentro de la componente, y el segundo consiste en un modelo de Comportamiento.

Para especificar el modelo de relaciones entre las clases de una componente, se proveen las herramientas necesarias para representar Generalización, Especialización, Agregación, Membrecía, etc. Además es posible agrupar un conjunto de clases cooperantes en lo que llamamos clases de alto nivel, lo cual permite cierto grado de abstracción en el análisis de una componente. Cada una de las clases y relaciones representadas puede ser manipulada y estudiada en particular, todo a través de una interface gráfica amigable, con manejo de mouse y drag and drop similar al de las herramientas para composición estudiadas en el capítulo anterior. Desde este modelo se puede acceder al modelo de comportamiento de cada una de las clases.

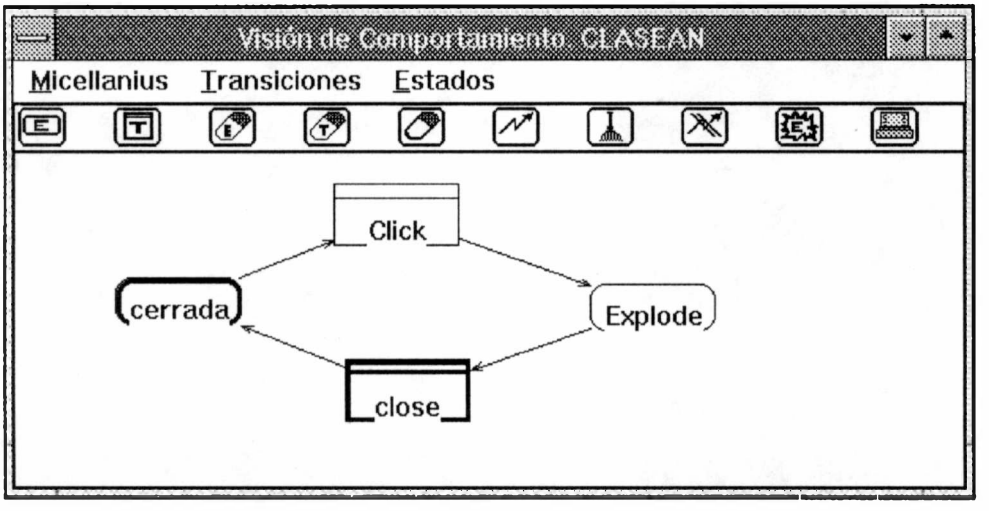


Figure 10. Diagrama de Estados y Transiciones.

En él, es posible crear el grafo de estados y transiciones cuyos nodos se unen a través de flechas. Como en el caso anterior, es posible profundizar el estudio de un estado o una transición mediante la creación de nodos de alto nivel.

Hasta aquí hemos presentado los dos modelos estáticos que describen los objetos y las relaciones entre ellos en la componente que se analiza. La herramienta logra controles automáticos de consistencia de los datos que describen la componente tales como herencia de comportamiento, links permitidos en el grafo de estados y transiciones, etc.

Para completar el estudio, hemos incluído un tercer modelo que describe la dinámica de los objetos participantes y sus interacciones. La importancia de un modelo de interacciones entre objetos radica en la posibilidad de apreciar la dinámica de la componente en funcionamiento, y ver cómo, con qué objetivo y en qué momento los objetos se comunican y establecen relaciones entre sí y en qué momento dejan de hacerlo. Nuestra propuesta incluye, entonces, un tercer

ambiente el cual permite ver a las instancias de las clases definidas interactuando entre sí en forma dinámica.

Con la información requerida al crear el modelo de comportamiento, la herramienta muestra en este tercer modelo cómo y con qué objetos interactúa una clase a medida que recorre su grafo de estados y transiciones. A través de flechas de diferentes colores se aprecian las interacciones entre un objeto estudiado y los demas objetos de la componente. De este modo es posible ver a la componente ´en acción´.

7.3.6.1 Los Modelos de Descripción Interna.

1- Grado de Formalidad.

Los desarrolladores no son matemáticos. No podemos esperar que transmitan sus conocimientos en cálculos de predicados de primer orden ni ninguna otra notación altamente formal. Además de lo dificultoso que esto sería en una organización dedicada al desarrollo de soft, quitaría flexibilidad a la herramienta. Es necesario que los desarrolladores puedan volcar, en sus descripciones, notas, esquemas y conceptos que completen los modelos sin necesidad de ajustar sus ideas a notaciones matemáticas predeterminadas.

2- El Modelo de Relaciones.

Este modelo presenta a los objetos que forman parte de una componente y a las relaciones estáticas que tienen lugar entre ellos. Cada objeto puede ser físico o conceptual. Un objeto puede representar a una entidad del mundo real, a una estructura de datos o a una entidad conceptual, sobre todo en el caso de componentes de análisis.

Las relaciones establecen conecciones lógicas entre los objetos asociando objetos entre sí. Cada conección tiene un tipo determinado y características implícitas como cardinalidad, restricciones de participación (número de veces que un objeto puede participar de determinada relación), de concurrencia, etc. La herramienta no obliga a definir estas restricciones, aunque brinda la posibilidad de expresarlas.

Las relaciones incluídas hasta el momento son: a) **de Uso**: Aunque el nombre no sea muy feliz, se utiliza para indicar relaciones generales, no propias del paradigma de objetos, tales como *trabaja para, es dueño de, maneja a,* e infinitas alternativas. Es importante especificar una completa descripción para este tipo de relaciones. Son de fundamental importancia cuando se intenta describir una componente de análisis. b) **de Generalización-Especialización**: Plasman la relación de herencia entre clases u objetos participantes en una componente. Como es sabido, representan la relación es *un*. Es posible especificar herencia

simple y múltiple. Respetando el concepto de herencia, el relacionar dos clases mediante este tipo de conección hace que la clase hija herede el modelo de comportamiento de la clase ancestra. c) **de Agregación**: establece una relación *es parte de*. Esta es una relación muy común en diseños orientados a objetos. d) **de Membrecía**: sirve para definir relaciones con semántica *es miembro de* y establece una relacion entre una clase conjunto y una clase miembro.

3- El Modelo de Comportamiento.

El modelo de comportamiento es similiar a una descripción de trabajo para un objeto, ambos describen qué es lo que el objeto debe hacer dentro de la componente. En el modelo de comportamiento se especifican los estados del objeto y las condiciones y eventos que hacen que pase de un estado a otro. Para definirlo, utilizamos una red de estados. Es importante especificar correctamente las características de cada transición, las condiciones necesarias para que esta se dispare, las acciones involucradas en su ejecución y los objetos con los que se interactúa al realizar estas acciones, ya que a partir de esta información, el sistema generará el tercer modelo: el Modelo de Interacciones.

4- El Modelo de Interacciones.

Este modelo tiene por finalidad mostrar a la componente ´en funcionamiento´. A través de él es posible seleccionar un objeto, guiarlo en el recorrido a través de su diagrama de estados y transiciones y ver gráficamente la forma en que interactúa con el resto de la componente.

Toda esta información permitirá a los desarrolladores que recuperen una componente el estudio de la misma. Sin esto, en un gran porcentaje de los casos es imposible adaptarla y menos aún extenderla para brindarle mayor funcionalidad y un grado mayor de reusabilidad.

7.3.7 Recuperación de Componentes.

A través de un fromulario similar al utilizado en el proceso de catálogo es posible especificar los requerimientos que se pretenden de las componentes a recuperar. Así, para cada una de las facetas, se especificarán uno o más términos. Por otro lado, se podrá determinar un órden de relevancia, tanto para las facetas como para los términos especificados. Esto determinará el órden en que se presentarán las componentes recuperadas. Por último, existen parámetros adicionales para guiar la búsqueda de componentes tales como palabras claves, nombres de clases, estados o transiciones, etc.

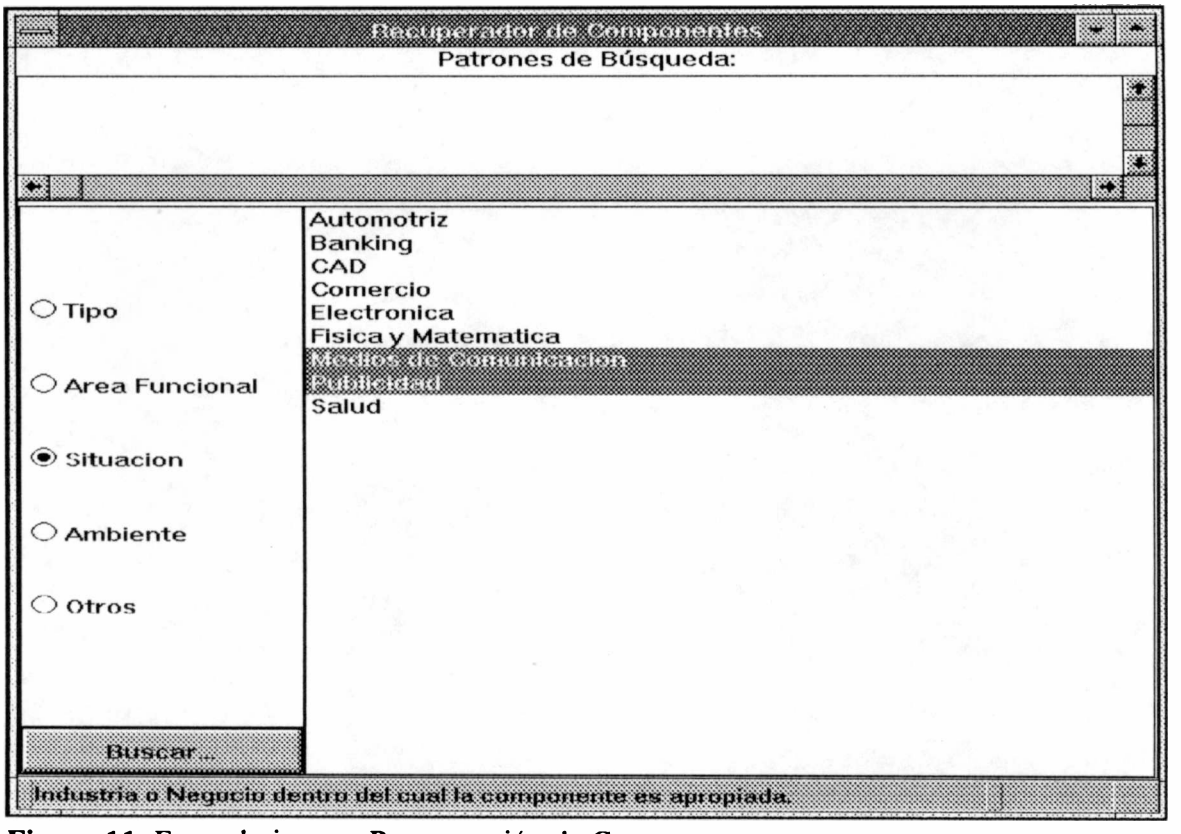


Figure 11. Formulario para Recuperación de Componentes.

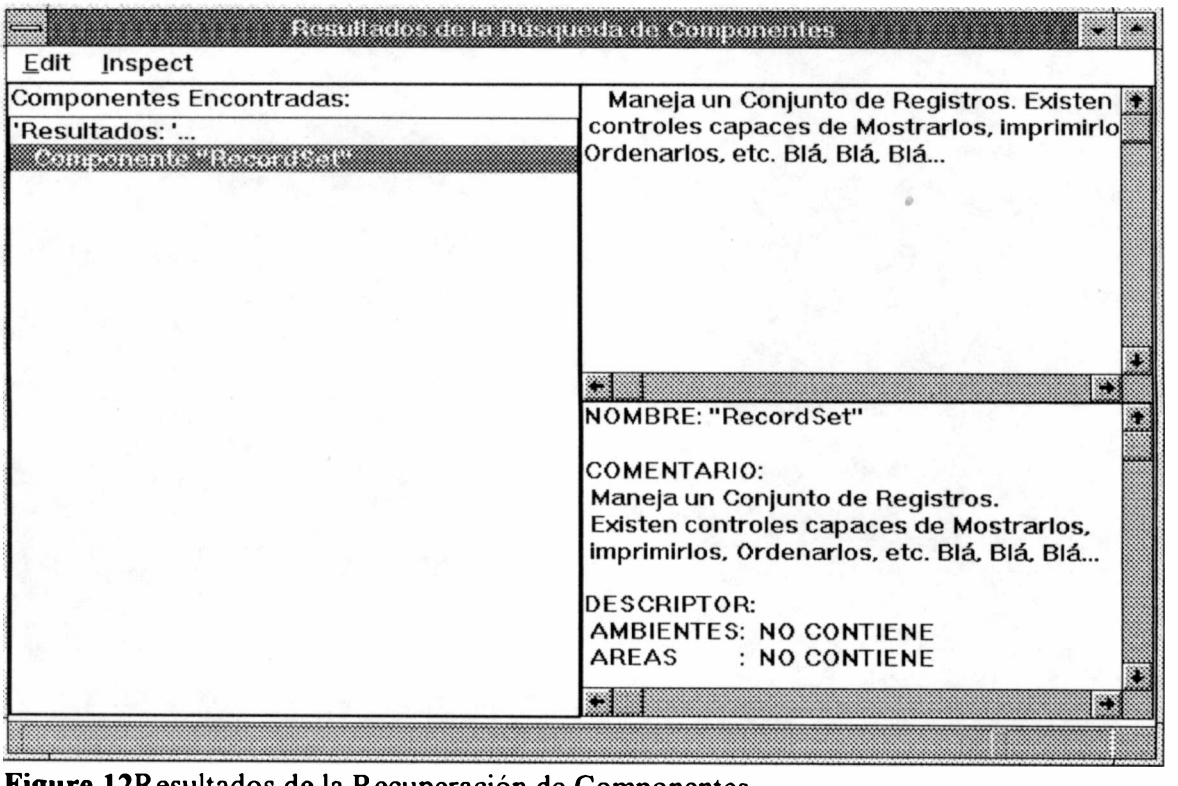


Figure 12 Resultados de la Recuperación de Componentes.

Una vez recuperada la lista de componentes candidatas, el usuario podrá estudiar cada una a través del descriptor completo utilizado al catalogarla, una primer visión gráfica, que muestra un resumen de la información necesaria para componerla y herramientas que permiten obtener el código fuente, objeto o documentos adicionales, y de ser necesario, podrá acceder al descriptor interno de las componentes recuperadas.

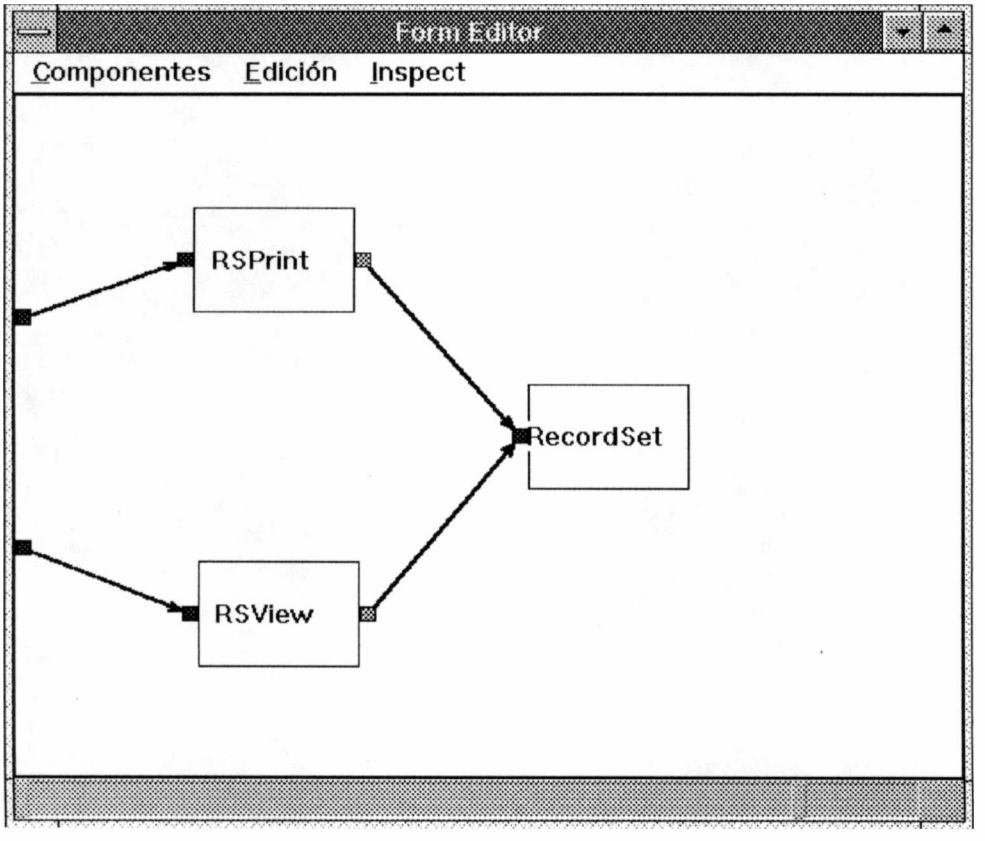
7.4 Un ejemplo de utilización de la herramienta.

En el siguiente ejemplo catalogaremos una componente desarrollada en C++ llamada RecordSet. Un RecordSet es básicamente un conjunto de registros contenidos en memoria. La idea de utilización de un RecordSet es poder contener datos (obtenidos, por ejemplo, a partir de una base de datos), que se pretendan mostrar en pantalla o imprimir. Para construirlo, se especifican los tipos de datos que se almacenarán en cada una de las columnas, y se le asigna un título y una función que se dipara en el momento en que sea invocado su método read(). La tarea de esta función será obtener los datos y cargarlos en el RecordSet.

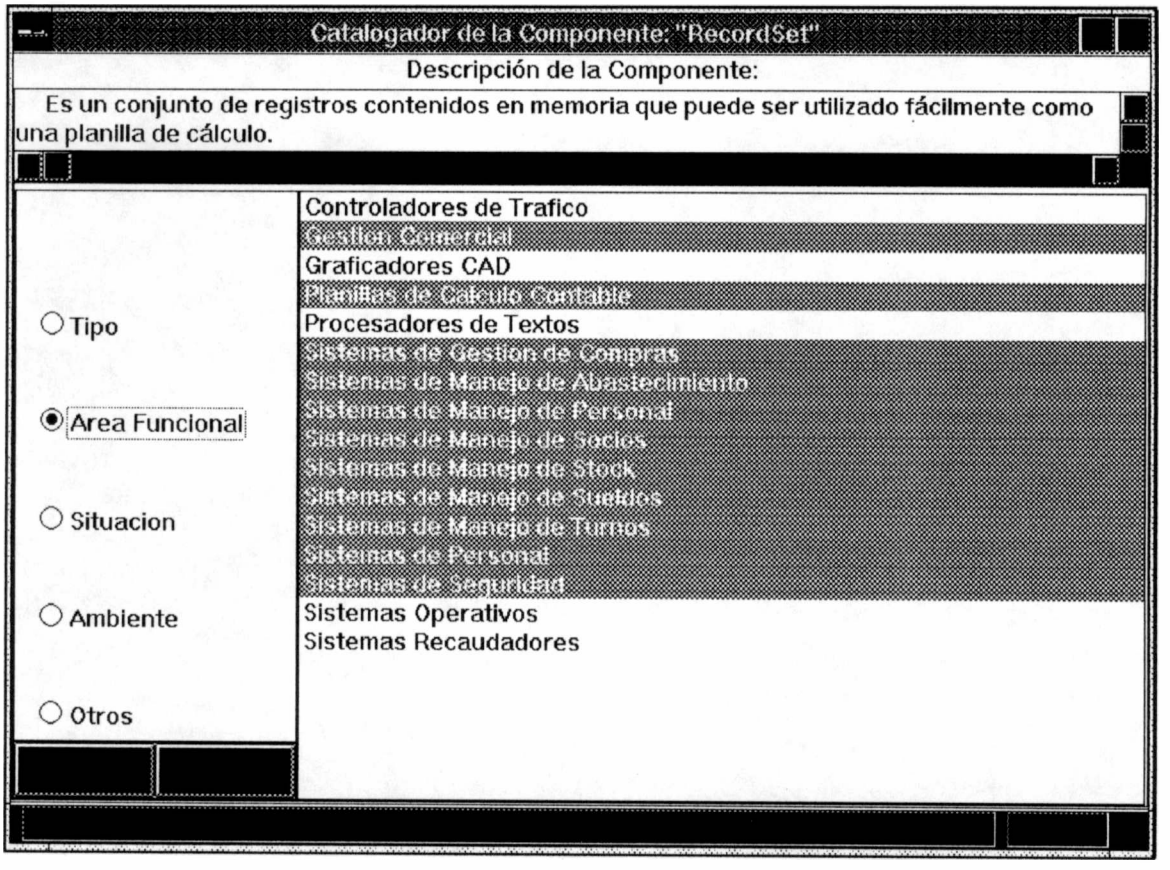
Como componentes complementarias, existen RSView y RSPrint. La clase RSView es capaz de mostrar los datos almacenados en un RecordSet. Es una clase de interface, y es posible ubicarla en cualquier ventana C++. La clase RSPrint, nos ofrece la posibilidad de Imprimir los datos contenidos en un RecordSet.

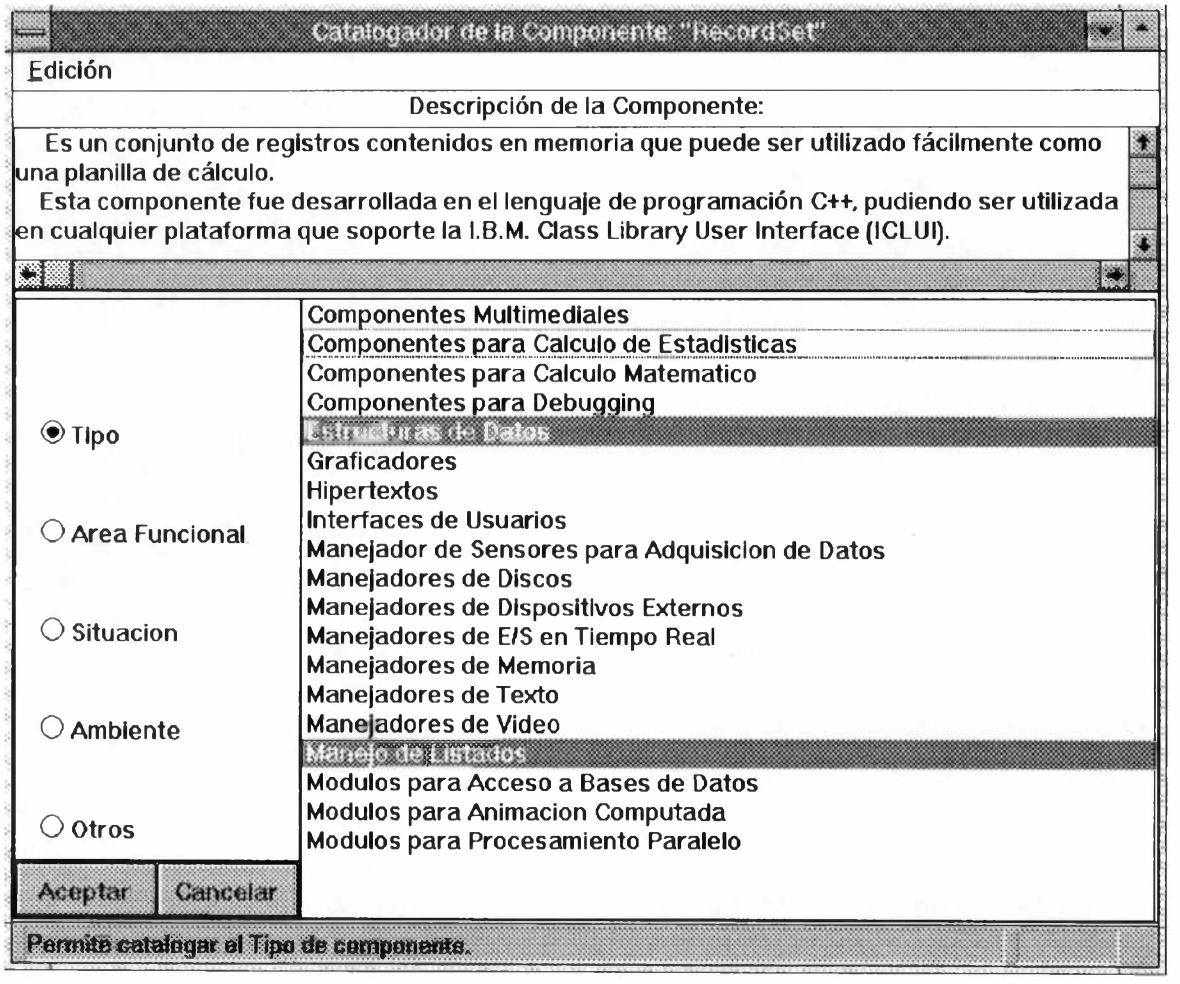
Como vemos, es una componente sumamente genérica, utilizable en una amplia variedad de aplicaciones, en los gráficos siguientes mostramos la forma en que podría ser catalogada.

El RecordSet posee un port de entrada a través del cual, RSView y RSPrint obtienen los datos para mostrar e imprimir respectivamente. A su vez, RSView posee un port de Entrada llamado show (implementado por el método show()) y RSPrint uno llamado print (implementado por el método print()). En la implementación real, ambos se construyen a partir de un RecordSet del cual son clientes.

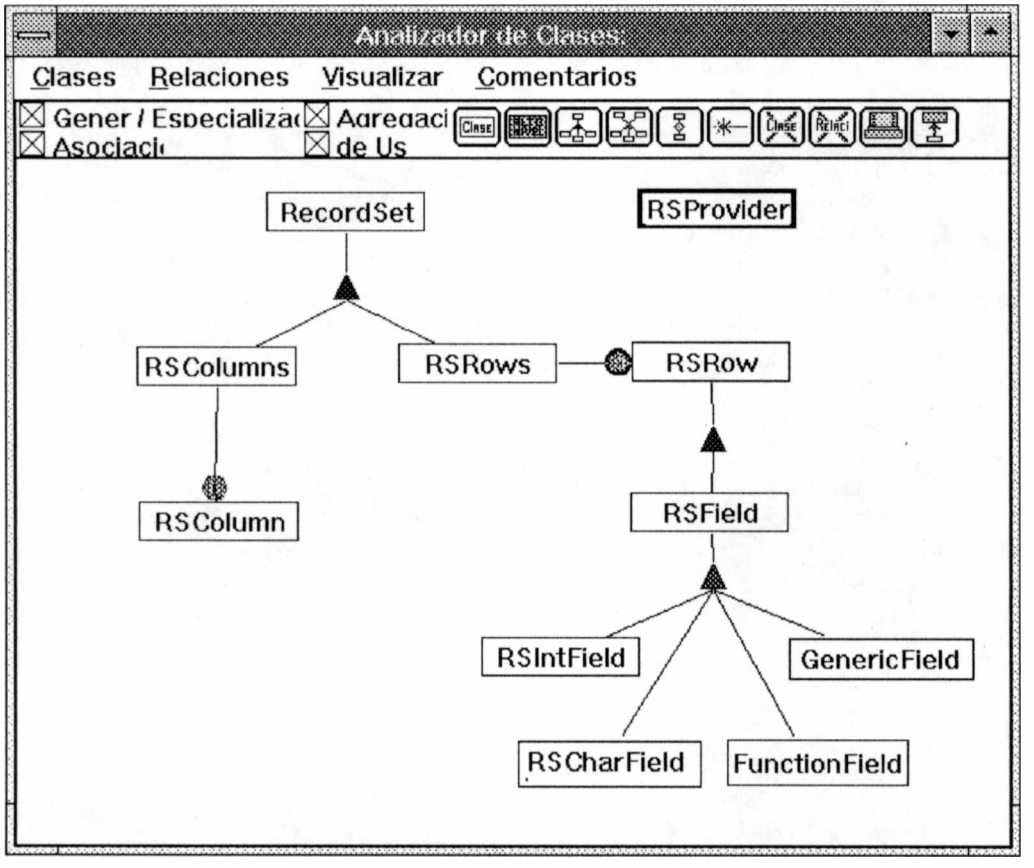


En la pantalla de catálogo, puede suceder que ninguno de los términos en alguna de las facetas sea adecuado como para describir a nuestra componente. En ese caso, es posible definir nuevos conceptos, tanto para Area Funcional como para Tipo, Situación y Ambiente.

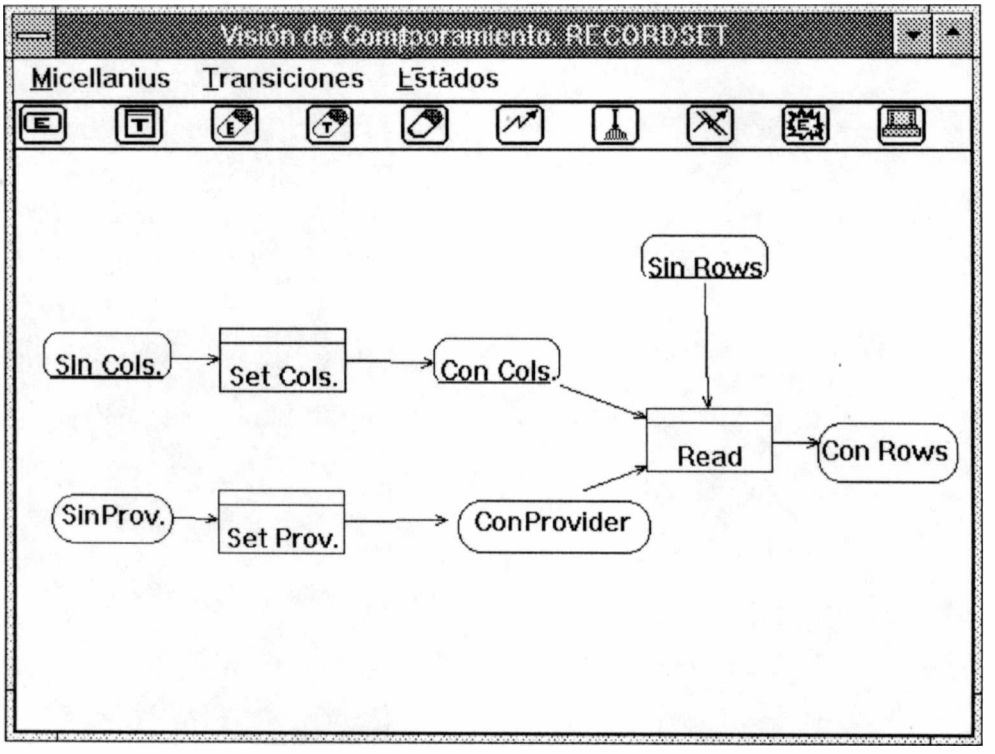


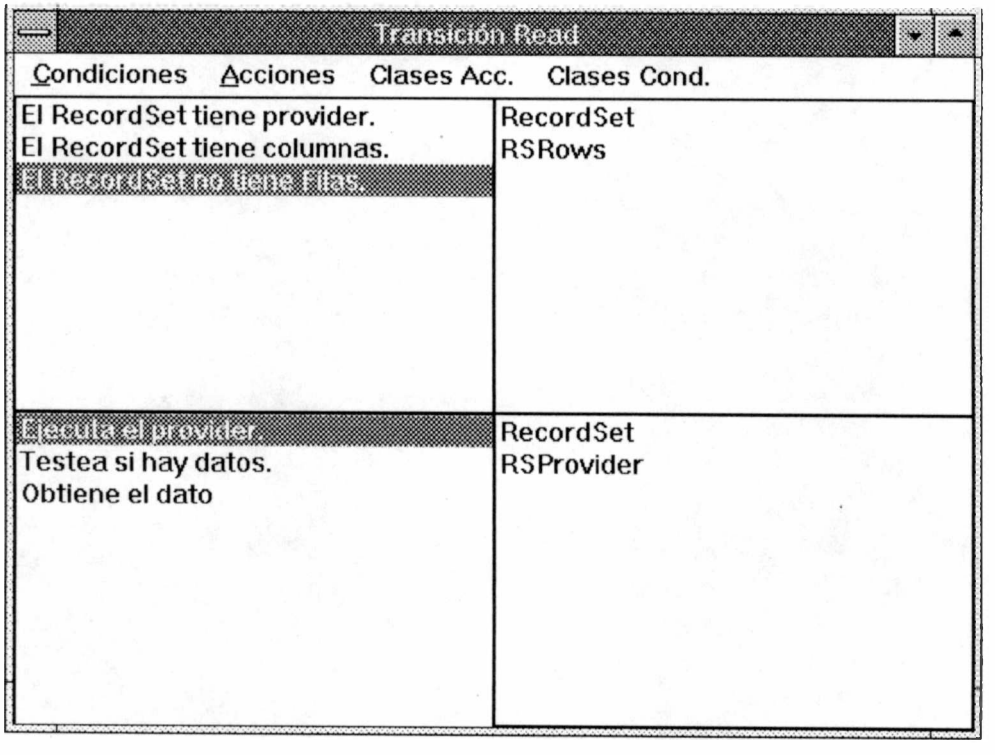


El próximo paso es describir a la componente internamente, completando el Modelo de Relaciones entre clases y el de Comportamiento de cada clase de una componente. En el gráfico, se puede apreciar la relación entre la clase RecordSet y clases cooperantes como RSColumns, RSFields y RSProvider.

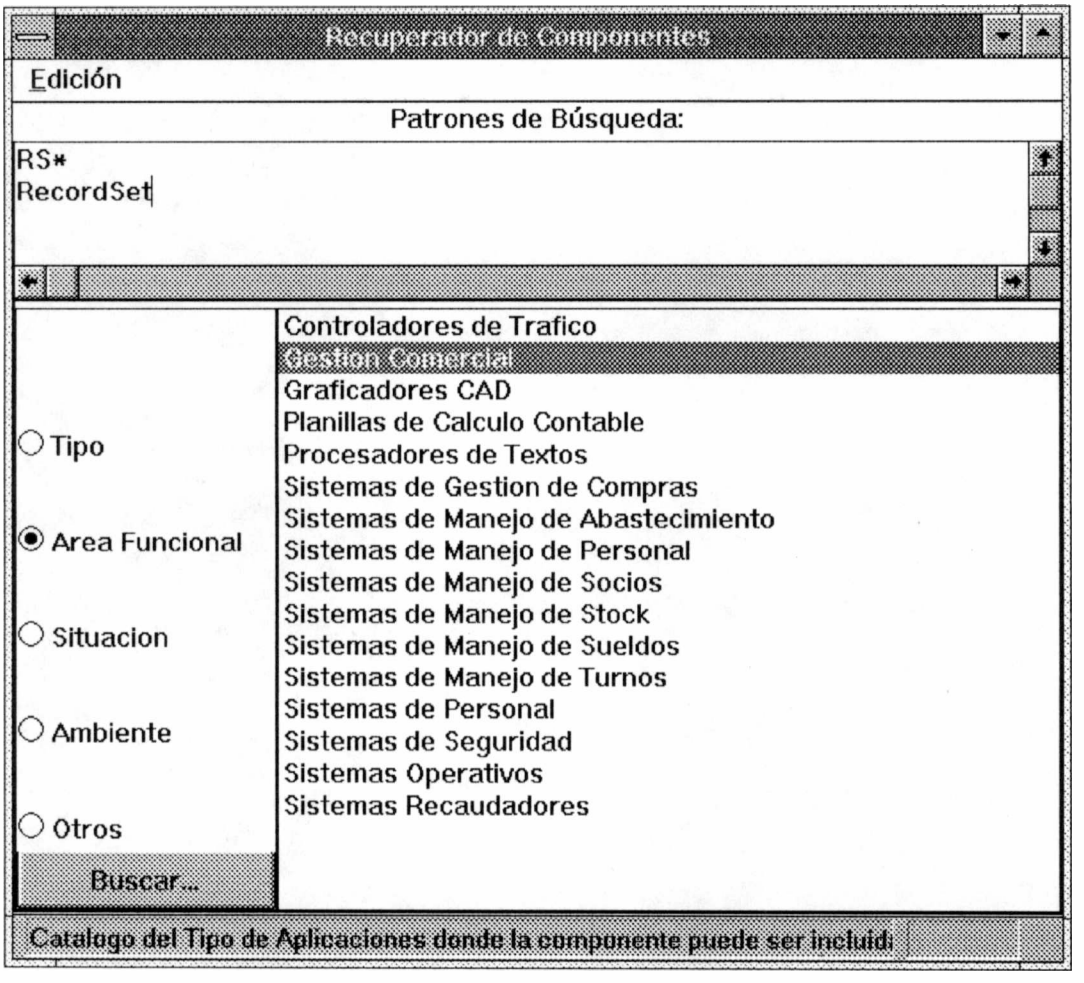


Como ejemplo de un diagrama de estados y transiciones, las siguientes figuras muestran una parte del comportamiento de una instancia de la clase RecordSet y la descripción de una de sus Transiciones, en la cual se especifican las condiciones necesarias para que sea disparada y las acciones involucradas en su ejecución, así como las clases que intervienen en la evaluación de estas condiciones y la realización de estas acciones.


Visión de Comportamiento. RECORDSET
Micellanius
Transiciones
Estados
Sin Rows
Sin Cols.
Set Cols.
Con Cols.
Read
Con Rows
SinProv.
Set Prov.
ConProvider

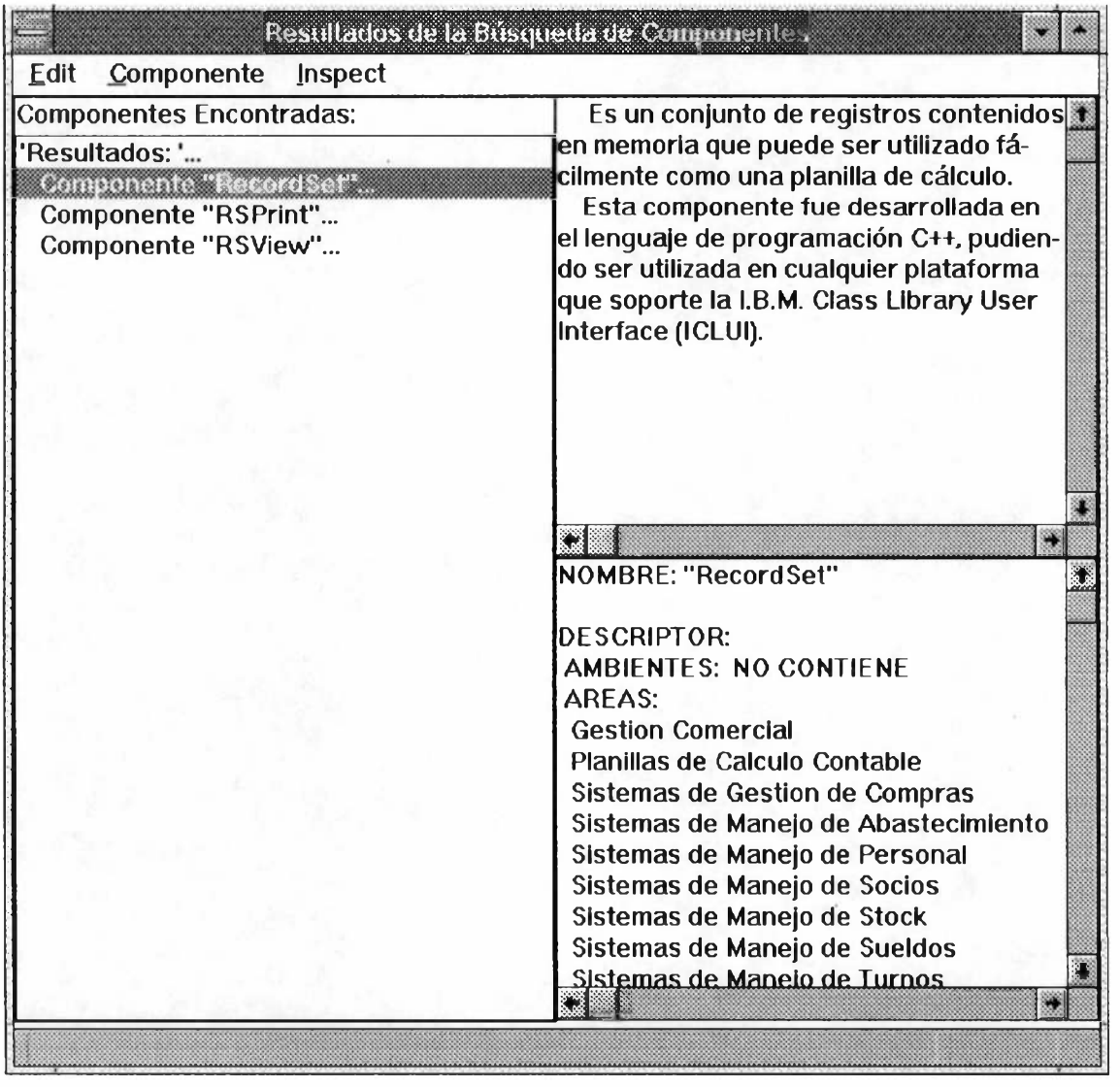

Transición Read
Condiciones
Acciones
Clases Acc.
Clases Cond.
El RecordSet tiene provider.
El RecordSet tiene columnas.
El RecordSet no tiene Filas.
RecordSet
RSRows
Ejecuta el provider
Testea si hay datos.
Obtiene el dato
RecordSet
RSProvider

Un desarrollador en busca de una componente que maneje los listados de su aplicación de gestión comercial, completaría un formulario del siguiente tipo:



Especificando, por ejemplo, Area Funcional: ´Gestión Comercial´, Tipo:´Manejadores de Listados´, Ambiente: ´OS/2, C++, AIX´. El hecho de que los mismos términos utilizados por el catalogador de la componente aparezcan en el momento de la recuperación, guiará al reutilizador de componentes en su búsqueda.

En la siguiente pantalla, se muestra el resultado de la búsqueda descripta antes.

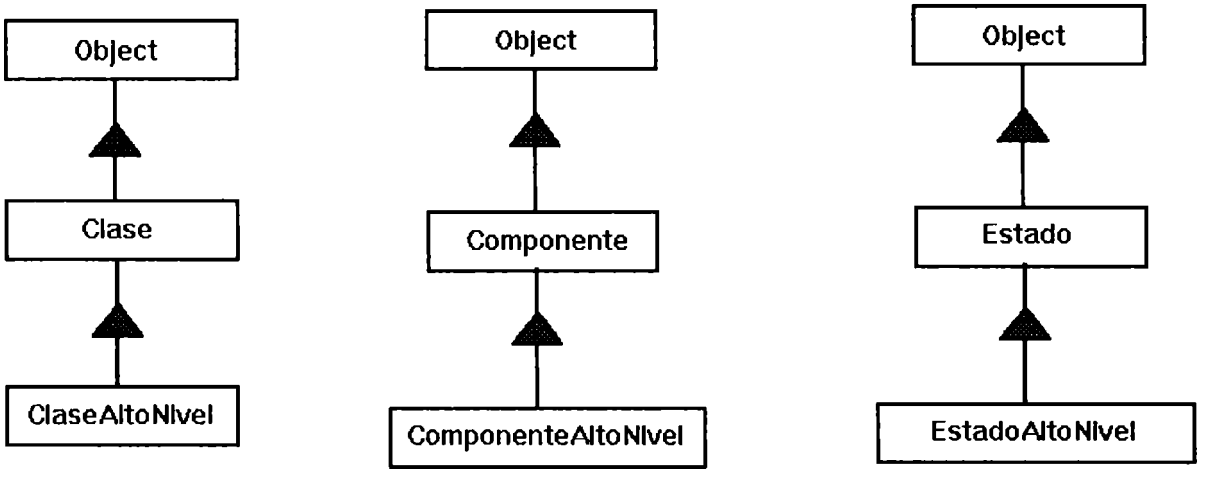


CAPITULO VIII : Implementación.

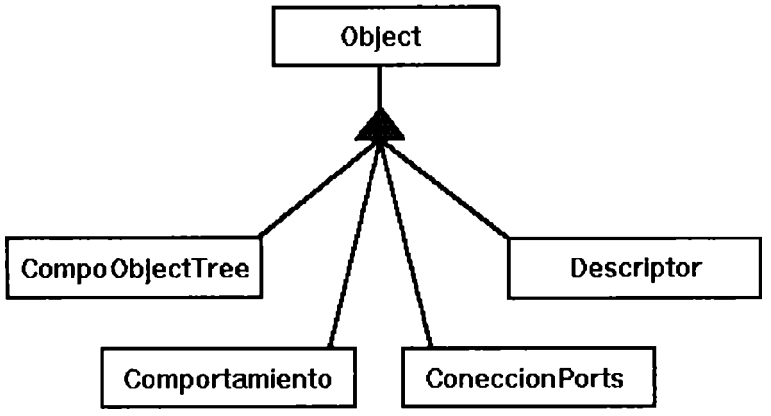
8.1 Arquitectura Interna de la Herramienta.

La herramienta ha sido implementada en Smalltalk, un lenguaje orientado a objetos puro que describiremos en los próximos párrafos. Debido a las características del lenguaje, es posible describir la arquitectura de la implementación en términos de las clases que la componen.

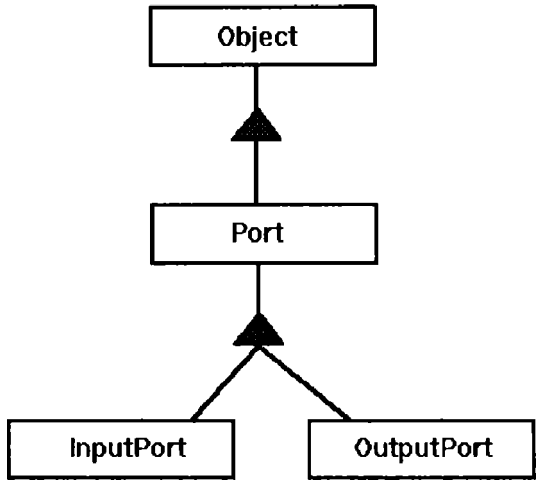
Para describir lo que desde el punto de vista del usuario es una Componente, hemos definido la clase 'Componente'. Dentro de la misma, existe un descriptor en el cual aparecen las 'Clases' que la componen. Cada 'Clase' a su vez, posee un 'Comportamiento', integrado por una red de 'Estados' y 'Transiciones'.



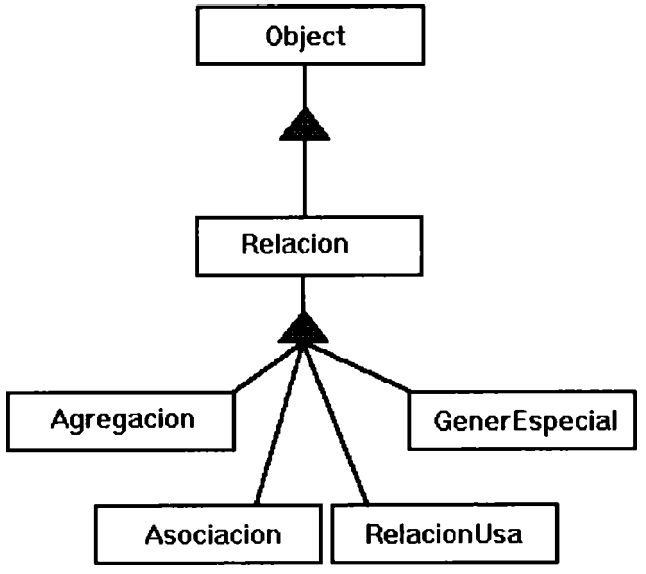
Tanto 'Componentes' como 'Clases' y 'Estados' pueden ser creadas con un concepto de abstracción, representado internamente a través de las clases 'ComponenteAltoNivel', 'ClaseAltoNivel' y 'EstadoAltoNivel' respectivamente. Una ClaseAltoNivel tiene un conjunto de 'Clases' vinculadas a través de 'Relaciones'. Asimismo, una 'ComponenteAltoNivel' posee un conjunto de subcomponentes vinculadas a través de 'ConeccionPorts'; y un 'EstadoAltoNivel' encierra dentro suyo a un conjunto de 'Estados' y 'Transiciones'.



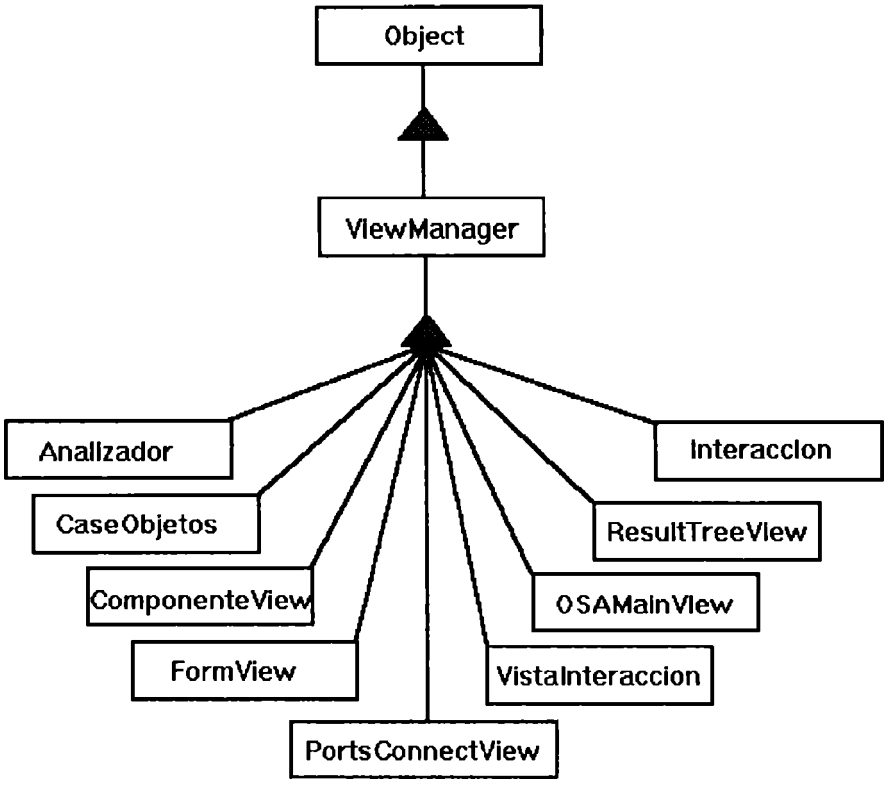
Un ´ConeccionPort´ conecta componentes a través de los ´InputPorts´ y ´OutputPorts´ de dos ´Componentes´.

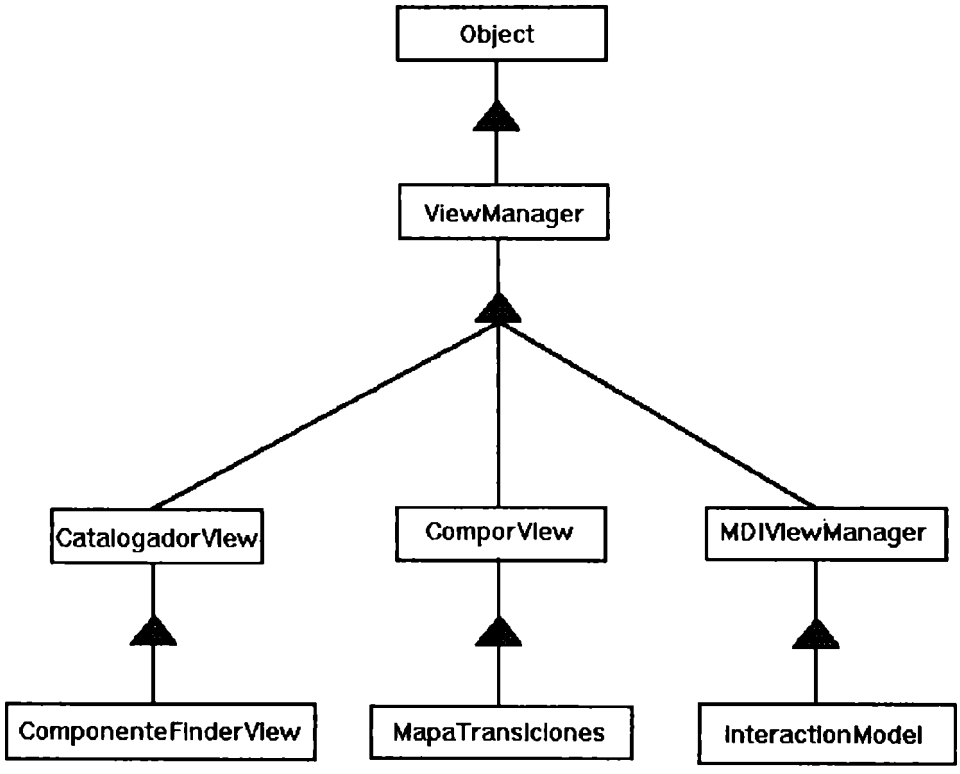


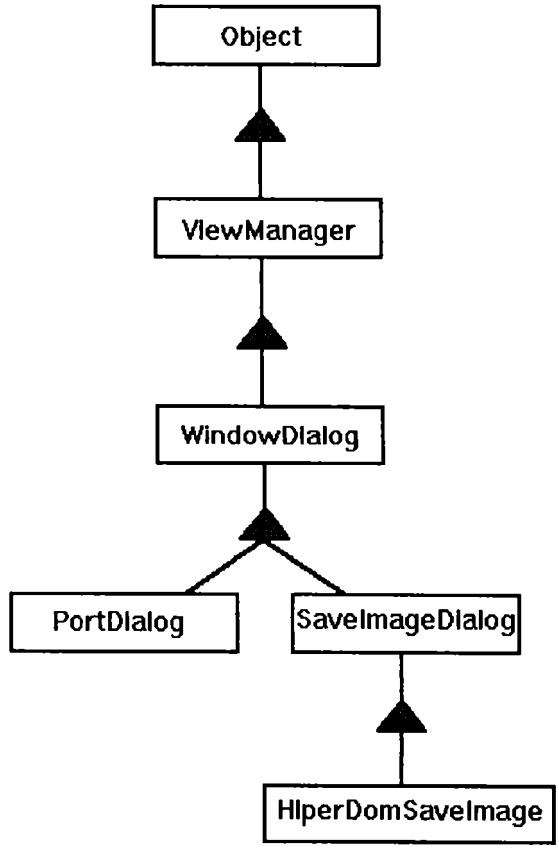
Las ´Clases´ dentro de una ´Componente´ se vinculan entre sí mediante distintos tipos de ´Relaciones´.



Todo esto conforma un modelo que se transmite al usuario a través de clases de interface especializadas:




Object
ViewManager
CatalogadorView
ComporView
MDIViewManager
ComponenteFinderView
MapaTransiciones
InteractionModel


Object
ViewManager
WindowDialog
PortDialog
SaveImageDialog
HIperDomSaveImage

8.2 La herramienta de Construcción: Smalltalk.

Hemos descripto un prototipo para catálogo, recuperación y estudio de componentes, de fácil integración con otras herramientas existentes en el mercado para desarrollo de aplicaciones orientado al reuso. Nuestra herramienta ha sido implementada en Smalltalk/V, de Digitalk y corre en entorno Windows.

Smalltalk es un lenguaje de programación para el desarrollo de soluciones tanto a problemas simples como de gran complejidad. Ha sido exitosamente utilizado para simulación, desarrollo de sistemas expertos, y de ambientes integrados de programación. Un ejemplo de este último punto es Visual Age. Smalltalk permite testear fragmentos de código sin necesidad de largas sesiones de compilación y linkeado. Es posible testear fragmentos de grandes programas a medida que se van desarrollando.

Smalltalk es un lenguaje orientado a objetos puro, es decir que permite fácilmente modelar problemas en términos de objetos y mensajes entre objetos. Soporta realmente el polimorfismo (es un lenguaje no tipado), herencia simple y encapsulamineto. Smalltalk **es** orientado a objetos, a diferencia de lenguages como C++ que simplemente **soportan** el paradigma de objetos. Esto se traduce en programas más elegantes, prolijos y fáciles de entender.

Sin embargo, la curva de aprendizaje de Smalltalk es lenta. Es necesario aprender a leer código Smalltalk para llegar a dominarlo, y leer código no resulta para muchos una de las tareas más agradables. Gran parte del ambiente de desarrollo Smalltalk está hecha en Smalltalk, y es posible acceder al código fuente de cada uno de los browsers y ventanas de trabajo; lo cual resulta de suma utilidad en el proceso de aprendizaje. Existen algo más de 150 clases y 2500 métodos incluídos en el ambiente de desarrollo. Examinando este código es posible aprender el lenguaje, ver cómo son resueltos una serie interesante de problemas, y familiarizarse con clases y métodos que deberemos reutilizar en la construcción de nuestras aplicaciones.

Una aplicación Smalltalk se compone de objetos que interactúan para realizar las tareas del programa. Los objetos tienen atributos (o características) y pueden tener tanto datos como métodos para realizar operaciones. Los objetos utilizan mensajes para comunicarse entre sí, transmitirse datos, determinar los métodos de otros objetos y requerir servicios. Una aplicación Smalltalk en ejecución es una colección de objetos encapsulados ejecutando procedimientos que se activan a través de mensajes. Todas las construcciones de programación y los elementos de datos son objetos que utilizan la herencia de clases, que le permiten a instancias (objetos) de clases hijas heredar atributos y capacidades definidas en clases ancestras.

Al comienzo del desarrollo utilizamos una herramienta para la generación automática de las interfaces de usuario, WindowBuilder. Sin embargo, surgieron necesidades que WindowBuilder no lograba cubrir, por lo que decidimos

abandonarla.

CAPITULO IX : Futuras Extensiones.

Los próximos pasos tienen que ver con brindarle a esta propuesta un soporte de Base de Datos adecuado, para lo cual estamos estudiando la posibilidad de utilizar Oracle vía ODBC u Object Store. Por el momento, y como una primer aproximación, hemos implementado una interface que permite utilizar Paradox desde Smalltalk, aunque creemos que no es la alternativa más eficaz. Es necesario poder almacenar datos tan complejos como código fuente, resultados de análisis de dominios, etc, y una base de datos relacional dificulta esta tarea. Con vistas a esta extensión, hoy existe una clase que maneja el tema de persistencia para los objetos de la herramienta, cuando contemos con los elementos necesarios, reemplazaremos su implementación por una más eficiente.

Por otro lado, queremos ampliar la funcionalidad y expresividad de los descriptores internos de componentes como para que en algún momento lleguen a soportar la captura del conocimiento contenido en un modelo de dominio. Hasta hoy, la herramienta está más relacionada con la parte de componentes implementadas, y sería de utilidad llevarla a las restantes etapas del desarrollo con más posibilidades. De todas formas, como conocemos las dificultades involucradas en la documentación del resultado de un análisis de dominio, planteamos la alternativa de incluir links entre la herramienta de recuperación y la utilizada para generar los documentos de análisis, integrando de esta manera las distintas tecnologías utilizadas durante el proceso de desarrollo.

Un tema que ha quedado sin tocar es el de la administración del repositorio de componentes. Existen cuestiones como la validación de componentes, quién está autorizado a catalogar una componente, quién aprueba el catálogo, cómo manejar versiones y extensiones de componentes, seguimiento y evaluación de componentes y publicidad de componentes. Para cubrir estos temas hemos pensado en las siguientes extensiones:

-Establecer un sistema de autorizaciones de forma tal de identificar a los usuarios que catalogan componentes y preveer su evaluación futura.

-Versionar la información de componentes, con el objeto de mantener los datos de la componente original y su evolución a medida que es reutilizada.

-Realimentar el sistema a partir de la información que aporten los desarrolladores que utilicen cada una de las componentes. Esto permitirá obtener una medición de la reusabilidad de las componentes almacenadas.

-Establecer un sistema de publicidad de componentes, es decir, que quienes las cataloguen puedan indicar al sistema que desean darla a conocer en forma especial, según el grado de relevancia que dichas componentes tengan. Por ejemplo, supongamos que un desarrollador cataloga una componente que cree que sería de utilidad en el desarrollo de aplicaciones de Banca, entonces podría pedir

al sistema que la distribuya (o al menos haga un delivery de mails) entre la gente de la organización dedicada a este tipo de desarrollos.

En síntesis, todas las extensiones que siguen apuntan a la integración de la herramienta con el resto del proceso de desarrollo y los miembros de una organización.

CAPITULO X: Conclusiones.

Hemos estudiado el Reuso de Software como una de las alternativas más atractivas para optimizar el proceso de desarrollo. Los sistemas actuales requieren soluciones rápidas y capaces de evolucionar al ritmo de sus negocios sin complicaciones. Para lograrlo es necesario un cambio tecnológico y hasta cultural en lo que a desarrollo de soft se refiere.

En este trabajo, hemos analizado los aspectos fundamentales a tener en cuenta en cada una de las etapas de un proceso de desarrollo orientado al reuso, descubriendo las tecnologías que pueden realizar su aporte en cada una de estas etapas.

Comenzamos por analizar los aspectos involucrados en un traspaso a este tipo de tecnología, y hemos comprobado que no sólo intervienen factores de índole técnico, sino también económicos, infraestructurales y culturales; y no es un paso trivial, para una organización dedicada al desarrollo de soft, el cambio hacia un proceso de desarrollo orientado al reuso. Hemos analizado los costos y beneficios asociados a este cambio, y podemos concluir que la inversión es ampliamente recuperable en el corto o mediano plazo, dependiendo del tamaño de la organización que enfrente esta transformación.

Hemos comprobado que una organización interesada en incorporar técnicas de reusabilidad necesita establecer programas de ingeniería de dominio para manejar la identificación, captura y evolución de la información reusable como así también montar una infraestructura para el reuso. Esta infraestructura debe ser ajustable a las necesidades, recursos disponibles, técnicas y tecnología de cada organización y se deberán evaluar los procedimientos de construcción de soft, fuentes de información, oportunidades de reusabilidad, inversión y posibilidades de recuperación para definir la conveniencia del esfuerzo involucrado en realizar el cambio.

En el capítulo III hemos planteado una serie de problemas que entran en perspectiva al pretender lograr reusabilidad a gran escala, específicamente, la necesidad de formar ingenieros en reuso, analistas de dominios y desarrollar herramientas capaces de soportar eficientemente la infraestructura de reuso.

Ha quedado plasmada en este informe la relevancia de las técnicas y métodos del Paradigma de Orientación a Objetos en un proceso de desarrollo orientado al reuso; aunque el éxito solo es posible si los desarrolladores son capaces de reutilizar componentes de software y diseños abstractos en una forma eficiente. Hemos mencionado los factores que juegan su rol en el logro de un alto grado de reusabilidad en componentes orientadas a objetos. El polimorfismo nos da herramientas para garantizar que una componente se comportará adecuadamente en distintos contextos. La herencia promueve el surgimiento de

protocolos standard y permite customizar componentes existentes e introduce el concepto de clase abstracta. Los frameworks permiten a un conjunto de objetos servir como molde de solución para determinada clase de problemas. Las técnicas de la Tecnología de Orientación a Objetos nos ofrecen una alternativa a escribir los mismos programas una y otra vez, pudiendo, en cambio, utilizar ese tiempo en crear y perfeccionar componentes generales con la convicción de que nuestras metodologías de desarrollo nos permitirán reexplotarlas.

En los capítulos V y VI cubrimos la etapa de composición de aplicaciones a partir de componentes reusables y subrayamos la importancia de la posibilidad de incorporar a nuestro proceso de desarrollo orientado al reuso, algún mecanismo de composición automático de componentes de soft, lo cual redunda en un nuevo incremento en la productividad.

Por último presentamos una propuesta que trata de cubrir uno de los puntos en que hemos comprobado que el estado del arte actual en cuanto a tecnología de desarrollo de soft orientado al reuso carece de elementos, y que tiene que ver con el proceso de catálogo, recuperación y estudio de componentes reusables a partir de un repositorio compartido por los integrantes de una organización dedicada al desarrollo de software.

APENDICE I : Descripción de las Clases Implementadas en la Herramienta.

A continuación se ofrece una breve descripción de la implementación de las clases que hacen al modelo de nuestra herramienta para catálogo, recuperación y estudio de componentes. Debido al modelo utilizado en la implementación (MVC, Model View Controller), las clases de interface conforman un módulo separado e independiente.

-Clase 'Componente': Superclase: 'Object'

Una componente tiene un nombre (#name) para identificarla; una descripción (#comentario) para explicar su funcionalidad; un conjunto de Ports, ya sean de entrada a la componente o de salida (#inPorts y #outPorts respectivamente). Para describir la funcionalidad de la componente, así como las distintas facetas en la que la misma fue catalogada, tiene un #descriptor. Por otro lado el conjunto de clases y relaciones que forman parte de la componente se guardan en la variable de instancia #sistemaOSA. Como se debe poder guardar tanto el código objeto como el código fuente de una componente desarrollada, existe también una variable de instancia #files que describe la ubicación de dichos archivos para que puedan ser accedidos.

-Clase ´ComponenteAltoNivel': Superclase: 'Componente'.

Es una componente que esta formada por otras componentes reusables (#subComponentes) donde los ports de entrada y de salida de cada subcomponente estan relacionados a través de ConeccionPorts (#conecciones).

-Clase 'Descriptor': Superclase: 'Object'

Una instancia de la clase Descriptor guarda la forma en que la componente fue catalogada en las distintas facetas. Cada una de estas facetas están identificadas con una variable de instancia: los Tipo de la componente se guarda en la variables de instancia #tipos, mientras que las Areas en #areas, y, análogamente las Situaciones, Ambientes y Otros en #situaciones, #ambientes y #otros.

-Clase 'Port': Superclase: 'Object'

Esta es una clase abstracta que describe el comportamiento común de sus subclases (InputPort y OutputPort). Básicamente, un Port tiene un nombre (#name)

para identificarlo, un tipo (#type) que es de mucha utilidad en lenguajes fuertemente tipados como C o C++ al describir la forma en que se comunica ese port con otras componentes. Un port a su vez conoce por medio de la variable de instancia #componente a la componente a la cual éste pertenece.

-Clases 'InputPort' y 'OutputPort': Superclase: 'Port'

Estas clases implementan a su manera la forma en que se imprimen.

-Clase 'ConeccionPort': Superclase: 'Object'

La responsabilidad de los objetos de esta clase es conectar las distintas componentes a través de sus ports, por lo que debe conocer, por medio de sus variables de instancia, la componente origen de la conección (#compoFrom) y el port de salida dentro de esa componente (#from), así como la componente destino (#compoTo) y el port de entrada a la misma (#to). A su vez, una conección puede ser descripta a través de la variable de instancia #comentario.

-Clase 'SistemaOSA': Superclase: 'Object'

Una instancia de esta clase contiene el conjunto de clases (#clases) que forman la componente. Todas las relaciones existentes entre estas clases están en la variable de instancia #relaciones.

-Clase 'Clase': Superclase: 'Object'

Una 'Clase' contiene un nombre (#nombre), una descripción (#contenido), un conjunto de relaciones (#relaciones) describiendo la herencia, agregación, uso y membrecía descriptas en los capítulos anteriores. Una clase posee un Comportamiento (#comportamiento), a través de la que se describen los Estados y Transiciones en el que los objetos de esta clase pueden estar.

-Clase 'ClaseAltoNivel': Superclase: 'Clase'

Un objeto de esta clase se comporta posibilita un grado de abstracción superior a un conjunto de clases y relaciones, agrupándolos en una única clase que hemos decidido denominar 'Clase de Alto Nivel'. Por este motivo, una instancia de esta clase posee un SistemaOSA (#sistemita) que contiene las clases y relaciones de más bajo nivel. También necesita conocer a qué sistema de alto nivel pertenece, por lo que debe conocer a su #padre.

-Clase 'Relacion': Superclase: 'Object'

Esta clase es una clase abstracta que muestra el comportamiento común a todas sus subclases. Toda relación tiene un nombre (#nombre), una descripción (#contenido) y un conjunto de clases que relaciona (#clasesRel).

-Clase 'Agregación': Superclase: 'Relacion'

Una relación de agregación contiene un conjunto de clases denominadas 'subpartes' y una clase llamada 'superparte. Las subpartes están representadas a través de la variable de instancia #clasesRel (heredada de la clase Relacion) y la 'superparte' es la variable #origen.

-Clase 'Asociacion': Superclase: 'Relacion'

Un objeto de esta clase representa la relación de membrecía.

-Clase 'GenerEspecial' Superclase: 'Relacion'

Las instancias de esta clase representan una relacion de Generalización-Especialización. Las clases que son generalizaciones están representadas a traves de la variable de instancia #clasesRel que es heredada de la clase Relacion, mientras que las clases que son especializaciones forman parte de la variable de instancia #especializ.

-Clase 'RelacionUsa': Superclase: 'Relacion'

Esta clase representa las relaciones de colaboración entre clases. Las clases que relaciona se encuentran en la variable de instancia #clasesRel.

-Clase 'Comportamiento': Superclase: 'Object'

Un Comportamiento contiene un conjunto de Estados (#listaEstados) y un conjunto de Transiciones (#listaTransiciones) en los que se puede encontrar una instancia de una clase en un momento dado.

-Clase 'Estado': Superclase: 'Object'

Una instancia de la clase Estado posee un nombre que la identifica (#name), una descripción (#comentario) acerca del mismo, y un conjunto de transiciones con las que interactúa. Las transiciones desde las cuales puede llegar un objeto de una clase a este estado están en la variable de instancia #desde, mientras que las transiciones habilitadas para pasar de este estado a otro son guardadas en #hasta.

-Clase 'EstadoAltoNivel': Superclase:'Estado'

Un estado de alto nivel, al igual que una clase de alto nivel y una componente de alto nivel, permite realizar un nivel de abstracción en el modelo, posibilitando encerrar un conjunto de estados y un conjunto de transiciones en un estado único para facilitar el estudio. Por ese motivo, un estado de alto nivel posee un Comportamiento en la variable de instancia #comportamiento.

-Clase 'Transicion': Superclase: 'Object'

Una transicion permite especificar un conjunto de condiciones (#condiciones) que se deben cumplir para que una instancia de una clase pueda cambiar de estado. La transición, una vez que se cumplen esas condiciones, debe ejecutar un conjunto de acciones (#acciones) que modificarán el estado de la instancia. Tanto las condiciones como las acciones pertenecen a una clase llamada Pasaje. Los estados desde los cuales se puede habilitar esta transición son guardados en la variable de instancia #desde, mientras que los estados a los cuales se pasa por medio de las acciones mencionadas son guardados en la variable de instancia #hacia. Un objeto de esta clase es identificado por su nombre (#name).

-Clase 'Pasaje': Superclase: 'Object'

Un pasaje tiene un nombre (#nombre), una descripción (#comentario) y una lista de clases con las que interactúa, ya que en una transición, tanto para evaluar una condición como una acción, es necesario interactuar con otros objetos de otras clases.

CAPITULO XI: Bibliografía.

-Measuring of the ROI of Reuse.
J. Bradford Kain. Object Magazine. Jun 94. Pag.49

-Object Oriented Metrics: Generation and Application.
Mark Lattanzi-Sallie Henry, Computer Science Department. Virginia Tech. Technical Reports from the Virginia Tech Department of Computer Science. Enero 1995.

-Object Oriented Software Engineering.
Addison-Wesley. I Jacobsen. Mayo, 1992.

-Cost estimation models for the reuse and prototype software life-cycles.
D. M. Balda-D. A. Gustafson. ACM SIGSOFT Software Engineering Notes.

-Pragmatics of reuse in the enterprise.
J. Bradford. Kain. Object Magazine. Diciembre 1994.

-Shaping the Future: Business Design Through Information Technology.
P.G.W. Keen. Harvard Business School Press, Boston, 1991.

-Measuring Software Design Quality.
D.N. Card-R. L. Glass. Prentice Hall, Englewood Cliffs, NJ, 1990.

-Findings on the impact of reuse on software quality at Hewlett-Packard.
W. C. Lim, 2nd International Workshop on Software Reuse, Lucca, Italia, 1993.

-Some experiences in Promoting Reusable Software: Presentation in Higher Abstract Levels.
Y. Matsumoto. IEEE Trans. Software Engineering, Vol SE-10, N°5. Sept. 1984, pag. 502-513.

-Domain Analysis and Software System Modeling.
G. Arango-Prieto Díaz. IEEE, 1991.

-Domain Engineering for Software Reuse.
G. Arango. Tech. Report. Dept. Of Information and Computer Science, Univ. de California, Irvine, Calif., 1981.

-Domain Análisis -From Art Form to Engineering Discipline.

G. Arango., CS Press, Los Alamitos, Calif. 1989 pag. 152-159.

-Genesis: An Extensible Database Management Sysrtem.
D. Batory. IEEE Trans. Software Eng. Vol 14. N°11 Nov. 1988.

-Towards Integrated Software Communities.
D. Tsichritzis - S. Gibbs. 1991.

-Strategic Planing for Distributed Object Management.
David S. Newman. Object Magazine. Jun. 1994, pag. 74-78.

-A preliminary study of large-scale software reuse.
J. W. Hutchinson-P. G. Hindley. Software Engineering Journal, Vol. 3 N° 5, Sept. 1988.

-ScrollController Explained: An Example of Literate Programing in Smalltalk.
W. Cunningham-K. Beck. Tech. Report, Tektronix, 1986.

-Reusability in the Smalltalk-80 Programming System.
L. P. Deutsch. De Tutorial on Software Reusability, IEEE Computer Society Press, 1987.

-Cognitive view of reuse and redesign.
G. Fischer. IEEE Software, 1987.

-Object Oriented Programming for the Macintosh.
K. J. Schmucker. Hayden Book Company, 1986.

-Interview with Wilma Osborne.
Ware Meyers. IEEE Software, 1988.

-Using types and inheritance in object-oriented programs.
D. C. Halbert-P. D. O´Brien. IEEE Software, 1987.

-Object Composition.
Dennis Tsichritzis, Universidad de Génova, Jun. 1991.

-Reusing and Interconnecting Software Components.
J. Goguen. Computer, pag. 16-28. Feb. 1986.

-The Right Criteria for Choosing a Client-Server Application Development Environment.
An Executive White Paper. May 94. Aberdeen Group, Inc.

-Software de Componentes.
Jon Udell. Byte, Mayo 94.

-Application Development Tools For Distributed Computing.
Hurwitz Consulting Group, Inc. Dic 93.

-Visual Age Users Guide. IBM Corporation. 2nd. Edition Feb 1995.

-Smalltalk/V.
Digitalk, Inc. 1986.

-Object Oriented System Analysis. A Model Driven Approach.
D. Embley-B. Kurtz-S. Woodfield. Yourdon Press. Prentice Hall Building. 1992.

-Reutilizaçào de software: aspectos de tecnologia de composiçào e ambiente de reutilizaçào.
Pujatti. Artículo presentado en JAIIO, 1994. Pag. 167.

NOTA:

En el momento de ser designada la Comisión Evaluadora, nos comprometemos a contactar a sus integrantes con el objetivo de brindar demostraciones de la herramienta que se describe en el capítulo VII de este documento.

Asimismo, brindamos a continuación la forma de contactarnos en caso de surgir la necesidad de demostraciones extras, exposiciones o aclaraciones respecto de este informe.

Teléfonos Particulares:
(021)- 51-0850 (Flia. Faiella)
(01)-322-4255 (Octavio-Germán)

Teléfonos Laborales:
(01)-313-0014 int. 3243 (Octavio)
(01)-313-0014 int. 3292 (Germán)

E-Mail:
FAIELLA AT BUEVM1.VNET.IBM.COM
OCTAVIO AT BUEVM1.VNET.IBM.COM