UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE INGENIERIA

INGENIERIA DE SOFTWARE ORIENTADA A OBJETOS

T E S I S

QUE PARA OBTENER EL TITULO DE

INGENIERO EN COMPUTACION

P R E S E N T A N :

ELIAS OMAR CRUZ RAMIREZ
PEDRO ESPINDOLA CORRAL
ADRIAN FELIPE NOVOA MARTINEZ
PEDRO SEDAS ZAMORA

FACULTAD DE INGENIERIA

U N A M

DIR. ING. JUAN JOSE CARREON GRANADOS

MEXICO, D. F.

DICIEMBRE 1994

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

A mis padres y hermanas: porque el esfuerzo fue suyo.
E. O. C. R.

A Diana, Pablo, Hugo, Martha y ✞Pedro.
P. E. C.

A mis padres y hermano: por su cariño y apoyo.
A. F. N. M.

A mi familia, especialmente a mi madre y a mis abuelos:
por el apoyo y confianza que me brindaron
P. S. Z.

INDICE

INTRODUCCION GENERAL 1

CAPITULO 1. ANTECEDENTES 3

Introducción 3
1.1 Ingeniería de Software 4
1.2 El Enfoque de Sistemas 5
1.3 El Enfoque Modular-Estructurado 6
1.4 El Enfoque Orientado a Objetos 7
1.5 Expectativas 9
Resumen 11

CAPITULO 2. ENFOQUE ORIENTADO A OBJETOS 13

Introducción 13
2.1 Objetos 14
2.2 Encapsulamiento 14
2.3 Acceso a Objetos 17
2.4 Clases e Instancias 20
2.5 Herencia 22
2.6 Polimorfismo 25
2.7 El Ciclo de Vida del Desarrollo de Software 26
2.8 Limitaciones y Problemas con el Método de Cascada 32
2.9 Disminución del Tiempo de Desarrollo 33
Resumen 34

CAPITULO 3. DEL PROTOTIPO AL DESARROLLO DEL SISTEMA 37

Introducción 37
3.1 Prototipos 37
3.2 Taxonomías de Prototipos 38
3.3 Del Prototipo al Desarrollo 40
3.4 Ventajas 41
3.5 Herramientas de Auxilio 42
3.6 Tablas Históricas 43
3.7 Factores Importantes 44
Resumen 45

CAPITULO 4. FRAMEWORKS 49

Introducción 49
4.1 Frameworks 49
4.2 Desarrollo por Capas de Hardware 50
4.3 Construcción del Software como Hardware: Software por Capas 51
4.4 Frameworks para Prototipos 54
4.5 Mantener los Controles Tradicionales 55
4.6 Constructores de Interfaces Orientadas a Objetos 57
Resumen 60

CAPITULO 5. DESARROLLO DEL SISTEMA 63

Introducción 63
5.1 Requerimientos del Sistema 63
5.2 Identificación de Clases 64
5.3 Identificación de Responsabilidades 65
5.4 Identificación de Contratos 67
5.5 Identificación de Colaboraciones 69
5.6 Identificación de Jerarquías 73

5.7 Identificación de Subsistemas 84
5.8 Documentación del Diseño 93
Resumen 94

CAPITULO 6. PRUEBA 99

Introducción 99
6.1 El Propósito de la Prueba 100
6.2 Tipos de Prueba 102
6.3 Técnicas de Prueba 106
6.4 Unidad de Prueba 108
6.5 Prueba de Bloques y Paquetes de Servicio 112
6.6 Prueba de Integración 113
6.7 Planeación de la Prueba 115
6.8 Identificación de Prueba 116
Resumen 117

CAPITULO 7. REUSO 119

Introducción 119
7.1 Ingeniería de Software Reusable 119
7.2 Componentes como mecanismo de Reforzamiento 119
7.3 Definición de Componente 123
7.4 Uso de Componentes 125
7.5 Implementación con Componentes 127
7.6 Manejo de Componentes 129
7.7 Construcción de Componentes 129
7.8 El Sistema de Componentes 131
7.9 Documentación de Componentes 134
Resumen 137

APENDICE. TDRAWING SISTEMA PROTOTIPO 139

CONCLUSIONES 155

BIBLIOGRAFIA 157

INTRODUCCION GENERAL

El desarrollo de nuevos sistemas de software orientados a objetos se ha incrementado durante los últimos años, al grado de convertirse en una nueva disciplina y una actividad necesaria.

Desarrollar nuevos sistemas se convierte, sobre todo en las grandes empresas que manejan grandes cantidades de información, en una de las principales tareas de las mismas.

Los objetivos generales de esta tesis son los siguientes:

- Servir como base para el diseño conceptual de la construcción de un sistema usando el enfoque orientado a objetos.
- Usarse como una guía de referencia para el líder de proyecto, que norme las actividades y procedimientos que deben cumplirse para el desarrollo eficiente de sistemas orientados a objetos.
- Utilizarse como una referencia didáctica en clases de sistemas computacionales relacionadas con el análisis y diseño de programas de software.

La presente tesis fue desarrollada tomando en cuenta el orden de sus capítulos, siguiendo los pasos necesarios al generar una aplicación mediante un enfoque *"orientado a objetos"*.

El capitulo 1 establece el medio y ubicación del trabajo escrito, exponiendo la problemática a resolver. En el capítulo 2 se definen los conceptos y terminología usados a lo

largo del documento. El capítulo 3 describe el uso de un prototipo para el desarrollo rápido del sistema. En el capítulo 4 se tratan las estructuras de programas que se incrustan en la aplicación, conocidas como *frameworks* o *modelos,* y que son parte fundamental del diseño orientados a objetos. El capítulo 5 propone las actividades y pasos a seguir durante el desarrollo de una aplicación completa. El capítulo 6 expone las tareas que se pueden desarrollar para probar el software construido. El capítulo 7 enumera los tipos de programas que pueden ser reusados para posteriores aplicaciones. El apéndice A describe brevemente el sistema prototipo TDrawing. y finalmente se exponen las conclusiones generales de la tesis.

Se da una propuesta de las etapas que se consideran básicas, las cuales pueden variar o ensimarse, ya que el enfoque utilizado es flexible y no rígido, algunos autores que se han adentrado al tema, exponen diversas descripciones del ciclo de desarrollo orientado a objetos, usando diferente semántica, sin embargo, describen en general las mismas fases del ciclo, aunque no coincidan en los límites de las mismas. Se hace notar, que lo importante en el ciclo de desarrollo es que las actividades a realizar sean claras para los miembros del equipo responsable de la aplicación.

AGRADECIMIENTOS

Agradecemos al Ing. Juan José Carreón Granados, a María Rosa Orduña Galván y a nuestros profesores por su ayuda y orientación para la realización de este trabajo.

También agradecemos a nuestros familiares y amigos por el apoyo que nos brindaron durante estos años.

ANTECEDENTES

Introducción

Las modernas corporaciones enfrentan en la actualidad un gran dilema, están volviéndose organizaciones basadas en información, dependientes de un continuo flujo de datos virtualmente para cada aspecto de sus operaciones. Sin embargo, su capacidad para manejar la información está disminuyendo porque la cantidad está creciendo a un ritmo mayor que su capacidad para procesarla, lo cual trae como consecuencia serios problemas para manejar sus propios datos.

"En la actualidad, el problema no radica en el hardware, ya que las computadoras continúan mejorando en velocidad y potencia a un grado excepcional. El problema se encuentra en el software, pues su desarrollo para aprovechar todo el potencial de las nuevas computadoras parece un reto más difícil que construir máquinas cada vez más rápidas" (Taylor [1992]).

Este desperdicio del potencial del hardware debido al software afecta a todos los usuarios de computadoras, pero se concentra especialmente en grandes organizaciones, las cuales dependen en gran medida de su habilidad para construir grandes sistemas de información.

Para el desarrollo de software se debe contar con presupuesto y un tiempo de desarrollo, los cuales son usualmente sobrepasados, incrementando el costo originalmente

planeado. El esfuerzo de sacar el proyecto de software a tiempo y con el presupuesto esperado, da como resultado, en la mayoría de los casos un software con muchos defectos, y como los programas también son estructurados rígidamente, es casi imposible hacer cambios mayores sin un rediseño total.

1.1 Ingeniería de Software

Debido a las condiciones de cambio continuo y a las exigencias de los negocios, se necesita un software mejor y más rápido; ya que en muchos casos el software corporativo que se desarrolla esta obsoleto antes de que se entregue una versión liberada, y comúnmente no se pueden involucrar las futuras necesidades reconocidas. Estudios de este problema indican que en muy pocos casos los proyectos de software terminan con sistemas que trabajan, mientras que la gran mayoría tienen que ser arduamente reconstruidos o abandonados antes de completarse.

En la industria esta situación es conocida con el nombre de "Crisis de Software". Para enfrentar al conjunto de problemas característicos de la crisis del software, se debe contar con una disciplina efectiva que conjunte métodos y procedimientos para el correcto desarrollo de software de computadora, tal disciplina es llamada "Ingeniería de Software".

La ingeniería de software es una metodología que integra elementos en etapas definidas, para obtener programas de alta calidad. Existen diferentes metodologías que se pueden aplicar para resolver el problema. El método que se presenta en este trabajo para el diseño de programas es llamado "Orientado a Objetos", el cual se explicará posteriormente.

1.2 El Enfoque de Sistemas

Para enmarcar los paradigmas de desarrollo estructurado y orientado a objetos es necesario conocer lo que es un sistema, pues son sistemas de computadora los que se desarrollan y programan para obtener sistemas de información que sirvan a los propósitos de una empresa.

Sistema es, en general, un conjunto de elementos interrelacionados que se distinguen de un medio entorno, de tal manera que un cambio en las propiedades del entorno afecta al sistema y un cambio del sistema actúa sobre él.

El concepto es aplicado en cualquier materia, disciplina o especie. Hay sistemas de todo tipo, por ejemplo: sistemas orgánicos en el cuerpo humano, numéricos, de comunicación, computacionales, sociales, de transportes, educativos, bancarios, entre otros.

Un sistema es en sí una unidad completa, que tiene propiedades y componentes específicos que se pueden identificar del entorno, pudiendo formar parte de un macrosistema o que sus componentes formen subsistemas.

La eficiencia en el funcionamiento de un sistema depende de la adecuada integración de sus elementos o componentes y de si es o no afectado por el medio que lo rodea.

El enfoque de sistemas influye en la ingeniería de software, ya que los grandes programas siempre representan un sistema; también influye la forma de "ser atacado" el

problema, ya que tiene que ser dividido en sus componentes esenciales para poder ser comprendido más a fondo, sin olvidar que el problema entero es también una unidad.

Los sistemas de programas para computadoras van dirigidos a la resolución de problemas, que se tienen que analizar y a los que hay necesidad de asignar recursos de la empresa para su elaboración.

Bajo el enfoque de sistemas se han desarrollado otros enfoques para el tratamiento de grandes programas que son y representan un sistema, estos son el "Modular-Estructurado" y el "Orientado a Objetos".

1.3 El Enfoque Modular-Estructurado.

El enfoque de sistemas ha influido en la metodología desarrollada para la elaboración de programas para computadoras, ya que constituyen la solución a un problema dado; los grandes programas representan la solución a un problema de sistema, con tal fin se dio un "enfoque modular" a la programación.

La programación modular consiste en la partición del programa entero en subprogramas menores llamados "módulos", ya que es más fácil resolver un problema muy grande y entero cuando se divide en trozos más manejables. Cada módulo o subrutina realiza una tarea específica y ofrece una división natural de trabajo que se puede repartir entre varios programadores. La estructura del sistema define una jerarquía de control, en donde, el

procesamiento descrito por cada módulo o subrutina incluye una referencia a todos los módulos subordinados a él.

Un diseño modular ofrece ventajas de reducción de complejidad, ayuda a hacer cambios fáciles y permite el desarrollo en paralelo de diferentes partes de un sistema.

El enfoque estructurado es un estilo de programación más disciplinado y consistente, que incluye dentro de su ámbito el enfoque modular (por lo que podría ser llamado enfoque modular-estructurado, en lugar de simplemente "estructurado"), el cual lo refina y se apoya de "estructuras lógicas", con las que se forma cualquier tipo de programa. Dichas estructuras facilitan determinar el dominio funcional, ya que cada una tiene un "principio" y un "fin", que ayudan a seguir fácilmente el flujo de instrucciones en un algoritmo dado.

Así, el enfoque estructurado se basa en la descomposición funcional en un diseño de programa de arriba a abajo, en el cual el programa se descompone sistemáticamente en componentes estructurados, los cuales a su vez son descompuestos en subcomponentes y continuando así hasta llegar a un nivel de subrutinas individuales.

1.4 El Enfoque Orientado a Objetos

La evolución de los lenguajes de programación ha pasado de lo no estructurado a lo estructurado, y de lo estructurado a lo orientado a objetos. La programación orientada a objetos (OOP por sus siglas en inglés), es un nuevo enfoque que trata de representar en forma más adecuada a la realidad, a los sistemas como realmente son. Su meta es facilitar la

programación por medio de un alto grado de abstracción del sistema u objeto, descubriendo otros sistemas u objetos dentro de él.

Mediante objetos programados con sus características y/o propiedades específicas, se puede modelar fielmente a los correspondientes objetos reales. El enfoque orientado a objetos se concentra en los datos que lo forman o lo hacen ser un objeto, e identificando las funciones asociadas a sus elementos.

Ya que, "el reconocimiento de los problemas y sus causas, así como el desenmascaramiento de los mitos del software, son los primeros pasos hacia las soluciones. Luego, las soluciones deben dar asistencia práctica al que desarrolla el software, mejorar su calidad y finalmente permitir al 'mundo del software' emparejarse con el 'mundo del hardware'"(Pressman [1990]).

La diferencia entre la programación orientada a objetos y la ya tradicional programación estructurada radica en el aprovechamiento de un punto de vista más acorde con la realidad, que permite clasificar elementos en conjuntos de pertenencia. El enfoque estructurado tradicional se enfoca en las funciones o procedimientos a utilizar en un programa, mientras que el enfoque orientado a objetos comienza, no con la tarea a ser desempeñada, sino más bien con aspectos del mundo real que necesitan ser modelados en orden para desempeñar dicha tarea. Una vez que son correctamente representados los objetos, el modelo puede ser usado para solucionar una gran variedad de tareas.

En la pasada década los lenguajes de programación fueron diseñados para tomar ventajas del diseño estructurado, sin embargo, estos lenguajes no dieron la flexibilidad suficiente para manejar las complejidades de los requerimientos del software moderno. Los lenguajes de programación estructurados han ayudado ciertamente a organizar el código, pero no se ha podido mantener fácilmente. Con el enfoque orientado a objetos y con lenguajes especiales para este propósito, los programas pueden ser modificados y mantenidos fácilmente. De esta manera, el enfoque orientado a objetos provee el soporte necesario para ayudar a desarrollar mejores programas.

1.5 Expectativas

La orientación a objetos aparece para constituirse en el camino principal de la computación comercial para desarrolladores de software y usuarios finales. Su crecimiento está ocurriendo dentro de un amplio rango de componentes de software, incluyendo lenguajes, interfaces de usuarios, bases de datos y sistemas operativos. La programación orientada a objetos es para los 90's, lo que fue la programación estructurada para los 70's: un nuevo e importante paradigma para la construcción, mantenimiento y uso de software. Este tipo de programación cambiará la forma de trabajar de los programadores e incrementará la velocidad con la que se producirá la siguiente generación de software; pudiendo también expandir la capacidad de programación del usuario final, la funcionalidad que se puede construir dentro de las aplicaciones habilitará a los usuarios a ganar acceso a los actuales y expandidos tipos de datos alrededor de heterogéneas plataformas de computación. Los lenguajes orientados a objetos estándar están comenzando a ser cristalizados con extensiones de lenguajes de programación populares, como Pascal, C y COBOL.

Las herramientas de desarrollo, ambientes de sistemas y aplicaciones que soportan información multimedia, computación de usuario final y procesamiento distribuido, son fuerzas imperantes para la orientación a objetos.

Los sistemas de hoy continúan incrementándose en complejidad en un gran número de dimensiones, no sólo en términos de requerimientos para extender la funcionalidad, sino también en términos de diversos tipos de datos. Los sistemas del futuro llevarán procesamiento de imágenes, sonido y video, además de texto y números como hasta ahora. Las áreas de aplicación como CAD, CAM, CIM, CASE, etc., deben de tener la capacidad de la programación actual y sistemas de arquitectura de software, con sus requerimientos para la simulación, representación del mundo real, manipulación y relaciones complejas entre diferentes tipos de datos.

Invariablemente los usuarios de las computadoras de los 90's experimentarán cambios para proveer la capacidad del soporte integrado de multimedia, como parte de la configuración básica de su equipo de cómputo, incluyéndose también procesamiento digital de señales para permitir la manipulación de datos de audio, para cualquier voz o aplicaciones musicales, dispositivo integrado de CD/ROM, movimiento completo de video y gráficos de color enlazados, lo que implicará diseño de herramientas y desarrollos especiales para el soporte de sistemas de producción multimedia.

Por otra parte, los retos de los 90's implican la exploración de oportunidades para múltiples aplicaciones, mientras que se oculta la complejidad de todo lo que está involucrado con el software, incluyendo al usuario y al desarrollador de la aplicación. A través de interfaces

gráficas se tienen mayores beneficios debido a la complejidad oculta para los usuarios, haciéndolas fáciles de usar, el requerimiento para soportar éstas interfaces gráficas y su complejo desarrollo ha incrementado la carga de programación y la tarea de mantenimiento.

En la presente década las redes de área local se incrementarán perdiendo su naturaleza normal y se requerirán gateways para alargarse en amplias redes de áreas corporativas. La robustez de las arquitecturas firmes de LAN's también se aprovecharán, incluyendo interfaces de datos distribuidas por fibras para soportar transmisión de datos de multimedia como un firme protocolo de red. Estas redes desarrollarán nuevas jerarquías para cada capa, ofreciendo sofisticación en comunicaciones, aplicaciones y administraciones de redes punto a punto.

Las técnicas orientadas a objetos proveen la flexibilidad esencial para el desarrollo de estos complejos sistemas. El software orientado a objetos promete ser mucho más que el familiar "apunte" y "oprima" de un interface de ventanas. Proveerá entornos en los cuales los usuarios pueden comunicarse entre aplicaciones y navegar fácilmente sobre arquitecturas distribuidas y diferentes. El diseño orientado a objetos y sus herramientas involucradas son la base para implementar el cada vez más complejo e innovador software de los 90's.

RESUMEN

- La ingeniería de software es una metodología que integra elementos en etapas definidas, para obtener programas de alta calidad.

- La programación orientada a objetos, es un enfoque que trata de resolver algún problema de sistemas simulando la realidad del mismo.

- Los lenguajes basados en el diseño estructurado no son lo suficientemente flexibles para manejar adecuadamente los requerimientos del complejo software de hoy y el futuro.

- El diseño orientado a objetos provee el soporte y flexibilidad necesarios, para hacer mejores diseños de programas, que son sistemas complejos de hacer.

ENFOQUE ORIENTADO A OBJETOS

Introducción

La programación orientada a objetos significa, esencialmente, el desarrollo de programas utilizando objetos. Este estilo de programación ha sido usado desde hace poco tiempo y sólo en pequeñas escalas. Su mayor poder radica en estimular el reuso del código y en la facilidad de entenderlo y mantenerlo, a diferencia de otros tipos de programación.

A menudo se dice que la OOP, es una forma más natural para programar que la programación tradicional, y es verdad, pues permite organizar la información en forma más familiar, al tratar de ser un reflejo de nuestras técnicas para manejar la complejidad. Es por eso que el definir qué es y qué no es la OOP es una tarea difícil, si es un estilo de programación o depende del lenguaje. Dos de las cualidades más importantes de la OOP son las siguientes:

- La comprensión de un sistema es más fácil, pues la diferencia entre la semántica del sistema y la realidad es muy pequeña.
- Las modificaciones al modelo tienden a ser locales, resultado de un desarrollo individual, el cual se representa por medio de un objeto.

Otra de las grandes ventajas de la OOP es que nos permite pensar al nivel de un sistema del mundo real y no al nivel de un lenguaje de programación. Pero para poder entender mejor este "estilo" de programación se requiere un entendimiento completo de los conceptos básicos.

2.1 Objetos

El primer y más importante concepto que debemos describir es por supuesto, el concepto de *objeto*. Esta palabra se mal interpreta y se usa en todos los contextos. Un objeto es una entidad capaz de guardar un cierto estado (información) y que a su vez responde a un cierto número de operaciones (comportamiento) para examinar o afectar ese estado.

En forma más concreta: Un objeto se caracteriza por ser un conjunto de operaciones y un estado resultado de los efectos de estas operaciones.

Un modelo orientado a objetos se forma por un número de objetos que son parte de un sistema modelado. Cada objeto de la vida real (un carro, un teléfono, etc.), contiene determinada información. Se debe definir para cada objeto un conjunto de operaciones para modificar o leer la información almacenada, no obstante, se pueden definir operaciones que no afecten la información, sino que solo actúen dentro del ambiente.

2.2 Encapsulamiento

Un objeto puede ser visto como un "paquete" que contiene una colección de datos y procedimientos relacionados. La acción del empacado que nos permite contener estos elementos relacionados es llamado *encapsulación* y puede ser visto así:

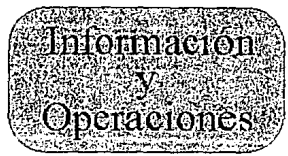


Fig. 2.1 Encapsulación

El concepto de encapsulación usado en un contexto orientado a objetos, no difiere del señalado en un diccionario. Se refiere a construir una cápsula, una barrera conceptual alrededor de una colección de cosas; en la OOP es el empacar datos y procedimientos juntos. "La encapsulación transforma múltiples elementos en un solo objeto por adhesión a un nuevo nivel del significado. De esta forma, la encapsulación usa nuestras habilidades semánticas para auxiliarnos en la conceptualización y trato con la complejidad" (Wirfs-Brock [1990]).

2.2.1 Ocultamiento de Información

El mecanismo de encapsulación de la OOP es una extensión natural de la estrategia de ocultamiento de información desarrollada en la programación estructurada. "La OOP mejora esta estrategia mediante mejores mecanismos para poner en orden los tipos de información juntos y esconder sus detalles en forma más efectiva" (Taylor [1992]).

La encapsulación dibuja un círculo alrededor de cosas relacionadas, pero estos objetos no son vistos tan fácilmente como si estuvieran encapsulados. El objeto tiene una interface

pública, que deja obtener información que necesitan otros objetos y una representación *privada*, que permite ocultar la información que no deba de estar disponible para otros objetos y programadores.

El ocultamiento de información nos permite quitar de la vista una porción de lás cosas que han sido encapsuladas por el objeto. Esto es útil para incrementar el alcance de la abstracción y para diseñar un código que pueda ser más fácil de modificar, mantener y extender.

El ocultamiento de información distingue la capacidad para realizar algún acto de los pasos específicos que realiza para conseguirlo; es decir, un objeto puede decir públicamente lo que puede hacer, pero no cómo lo hace. Además tiene que conocer que acciones puede pedir a otros objetos.

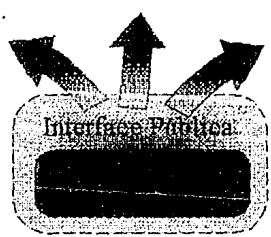


Fig. 2.2 Ocultamiento de Información

La encapsulación y el ocultamiento de información trabajan juntos para aislar una parte

del sistema de los otros, permitiendo que se modifique o extienda el código y se arreglen los errores, sin introducir efectos colaterales indeseados.

Estos dos principios se ponen en práctica en los objetos:

1. Se abstrae la funcionalidad e información que estén relacionados y se encapsulan en un objeto.
2. Se decide que funcionalidad e información necesitarán otros objetos de él, el resto se oculta. Se diseña la interface pública que permitirá a otros objetos accesar lo que requieran. La representación privada es protegida por omisión del acceso de otros objetos.

2.3 Acceso a Objetos

Si los objetos sólo pueden ser accesados desde sus interfaces públicas, entonces, ¿Cómo es que ese acceso es permitido?. Un objeto accesa a otro enviándole un *mensaje*.

2.3.1 Mensaje

La anatomía de un mensaje consta de tres partes: el nombre de un objeto receptor, el nombre de la operación que el receptor sabe como ejecutar, y cualquier parámetro que ésta operación requiera para llevar a cabo esa función. La tercera parte es opcional, si la operación no necesita alguna información adicional no hay parámetros en el mensaje.

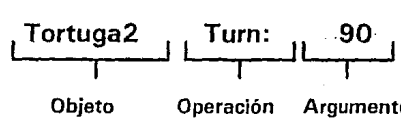


Fig. 2.3 Estructura de un mensaje

Esta es la estructura básica del mensaje, dentro de ésta hay muchas variaciones. La forma en que un mensaje es escrito depende del lenguaje que se use.

Normalmente, los tres componentes de un mensaje aparecen en un orden fijo - *receptor* + *método* + *parámetros* - pero la manera en que se nombran los componentes y se separan tiende a variar en los lenguajes de programación. Un mensaje en Smalltalk se vería así:

tortuga2 turn:90

El mismo mensaje en C++ u Object Pascal se escribiría así:

tortuga2.turn(90)

En cada caso, *tortuga2* es el nombre del objeto receptor, *turn* es el método que al receptor se le está pidiendo ejecutar, y *90* es un parámetro que especifica un ángulo de rotación.

Cuando un objeto envía un mensaje a otro, el emisor está requiriendo que el receptor del mensaje realice la operación mencionada y (posiblemente) regrese alguna información. Cuando el receptor recibe el mensaje, realiza la operación en la forma que él sabe. El requerimiento no especifica como una operación será realizada ya que la información siempre es escondida del emisor.

El conjunto de mensajes a los que un objeto puede responder, es conocido como el comportamiento del objeto. No todos los mensajes que un objeto responde necesitan ser parte de su interface pública. Un objeto puede enviarse mensajes privados para implementar operaciones accesibles públicamente.

2.3.2 Nombre del Mensaje

Un mensaje incluye el nombre de una operación y cualquier argumento que se requiera para esa operación. Algunas veces es usual referirse a una operación por su nombre, sin considerar sus argumentos. Al nombre de una operación se le llamará *nombre del mensaje.*

2.3.3 Firma

Un concepto relacionado con el envío de mensajes es una *firma*. Mientras un mensaje consiste del nombre de una operación y sus argumentos requeridos, una firma es el nombre de una operación, el tipo de sus parámetros y el tipo de objeto que la operación regresa, es como en lenguaje "C" el prototipo de la función. La firma identifica completamente un mensaje.

2.3.4 Método

Un *método* es un procedimiento o función que se involucra para actuar sobre un objeto. Un método especifica "*cómo*" se ejecuta un mensaje.

Cuando un objeto recibe un mensaje, realiza la operación requerida ejecutando un método. "Un método es un algoritmo ejecutado paso a paso en respuesta a la recepción de un mensaje, cuyo nombre es igual al nombre del método" (Wirfs-Brock [1990]). Como se especifica por el principio de ocultamiento de información, un método es siempre parte de la representación privada de un objeto y nunca de la parte pública.

2.4 Clases e Instancias

Algunos objetos durante una aplicación se comportarán en forma diferentemente a otros, así como otros lo harán de manera similar.

2.4.1 Clases

En los sistemas que podemos modelar hay un número de objetos que se comunican. Algunos de estos tienen características similares y se pueden agrupar de acuerdo a esas características, tal grupo representa una *clase*.

Una clase es una especificación genérica para un número arbitrario de objetos que tienen un comportamiento y una estructura de información similar. Esto nos permite construir una taxonomía de objetos sobre un nivel abstracto y conceptual.

En forma más concreta podemos decir que una clase representa un modelo para muchos objetos y describe como están estructurados estos objetos internamente. Objetos de la misma clase tienen la misma definición para sus operaciones y para sus estructuras de

información.

Una clase es a veces llamada el tipo de un objeto. Sin embargo, son dos cosas realmente diferentes, pues un tipo de datos abstracto se define por un grupo de operaciones mientras que una clase es más que eso, pues se puede ver dentro de una clase también sus estructuras de información.

La función básica de una clase es definir un tipo particular de objeto. Una vez que se ha definido una clase, se puede crear cualquier número de instancias únicas de esa clase.

2.4.2 Instancias

En un sistema orientado a objetos, cada objeto pertenece a una clase. Un objeto que pertenece a una cierta clase es llamada una *instancia* de esa clase, por lo que todos los objetos son instancias de alguna clase.

Una instancia, es entonces, un objeto creado a partir de una clase. La clase describe la estructura de la instancia (comportamiento e información), mientras que el estado actual de la instancia esta definido por las operaciones realizadas sobre la misma.

Una clase define las operaciones que pueden ser realizadas por una instancia y también define sus variables. Una vez que una instancia de una clase es creada, se comporta como cualquier otra, capaz de recibir un mensaje para realizar cualquier operación para la que tiene un método. No obstante, cada instancia tiene una identidad única, pueden crearse muchas

dentro de una cierta clase, donde cada instancia es manipulada por la operación definida por la clase. El único aspecto de una clase que difiere de una instancia a otra, son los valores de sus variables, llamadas variables de instancia.

2.4.3 Reuso de Nombres

Dentro de un programa, una tarea puede llevarse a cabo de diferentes formas, esta situación trae una importante pregunta acerca del nombre de la tarea: ¿Se puede usar el mismo nombre para todas las variantes o se tienen que usar nombres diferentes? A través de una técnica llamada *sobrecarga*, se puede usar el mismo nombre para un mismo método en diferentes clases, lo que simplifica los programas al permitirnos usar el mismo nombre para la misma operación en cualquier parte del programa.

2.5 Herencia

Cuando se describen clases, se nota rápidamente que muchas tienen características en común. La OOP soporta otro mecanismo de abstracción llamado *herencia*. La herencia es la propiedad que tiene una clase para definir el comportamiento y estructura de datos de sus instancias como superconjunto de la definición de otra u otras clases. En otras palabras se puede decir que una clase es igual a otra excepto que la nueva clase incluye algo extra. Con esto, la herencia nos da un mecanismo de clasificación que nos permite crear una *jerarquía de clases*.

La herencia permite concebir una nueva clase de objetos como el refinamiento de otra, al diseñar y especificar sólo las diferencias para la nueva clase, creando rápidamente otras nuevas. En otras palabras, la herencia es un mecanismo donde una clase de objetos pueden ser definidos como un caso especial de una clase más general, incluyéndose automáticamente las definiciones de datos y métodos de la clase general.

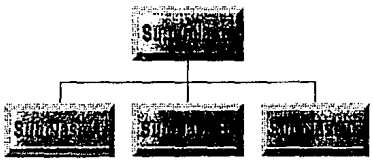


Fig. 2.4 Superclases y Subclases

2.5.1 Subclase

Una *subclase* es una clase que hereda el comportamiento de otra clase. Una subclase usualmente agrega nuevos métodos a su propio comportamiento para definir su propio y único tipo de objeto.

2.5.2 Superclase.

Las características comunes pueden estar repartidas en varias clases; si las juntamos en una clase específica y esta hereda su comportamiento, se dice que es una *superclase*. Una

superclase es entonces una clase cuyo ambiente específico es heredado a otras clases.

Como la herencia jerárquica incluye muchas clases podemos enfatizar la relación entre éstas. Si una clase hereda directamente su comportamiento de otra, se llama descendiente directo. La primera clase es entonces el ancestro directo de la segunda. Un ancestro directo es llamado a veces el padre y al descendiente directo hijo.

HERENCIA SIMPLE HERENCIA MULTIPLE

Fig. 2.5 Tipos de Herencia

Los ancestros desarrollados con el propósito principal de heredar a otros son llamados frecuentemente *clases abstractas*. Una clase desarrollada con el propósito principal de crear instancias de él, es llamada una *clase concreta*, las cuales también pueden ser ancestros de otras clases.

2.5.3 Herencia Múltiple

Cuando se describe una nueva clase y se desean usar características de dos o más clases existentes, se puede heredar de ambas clases, a lo cual se le llama herencia múltiple. Esto significa que una clase puede tener más de un ancestro directo. Una desventaja de la herencia

múltiple es que frecuentemente reduce la comprensibilidad de una jerarquía de clases.

2.6 Polimorfismo

El acceso a los objetos limitado a una interface definida estrictamente por el envío de mensajes, permite otro uso de la abstracción conocida como polimorfismo, del griego *polis* que significa "muchas formas". El polimorfismo es la habilidad de que dos o más clases de objetos respondan al mismo mensaje, cada una en su propia forma. Esto significa que un objeto no necesita saber como enviar un mensaje, sino solamente saber que varios tipos de objetos han sido definidos para responder a ese mensaje particular.

El polimorfismo significa entonces que el emisor de un estímulo (mensaje) no necesita saber la clase de la instancia receptora, pues ésta puede pertenecer a cualquier clase arbitraria. Así, un estímulo puede ser interpretado en diferente forma, dependiendo de la clase receptora, es por eso que la clase instanciada que recibe el estímulo es la que determina su interpretación y no la instancia transmisora.

El polimorfismo permite reconocer y explotar similitudes entre diferentes clases de objetos. Cuando se reconocen varios tipos diferentes de objetos que pueden responder al mismo mensaje, se reconoce la distinción entre el nombre del mensaje y un método.

Es tan importante que es considerado una de las características que define a la programación orientada a objetos. Su principal ventaja es que hace a los objetos más independientes unos de otros y permite que nuevos objetos se agreguen con mínimos cambios

a los objetos ya existentes. Todo esto de la simple capacidad de usar el mismo método en más de una clase.

2.7 El Ciclo de Vida del Desarrollo de Software

El ciclo de vida tradicional del desarrollo de software continuamente es llamado método de "cascada", porque asume que el esfuerzo de desarrollo fluye en una serie de etapas, siendo las más comunes el análisis de requerimientos, diseño del sistema, implementación, prueba y mantenimiento.

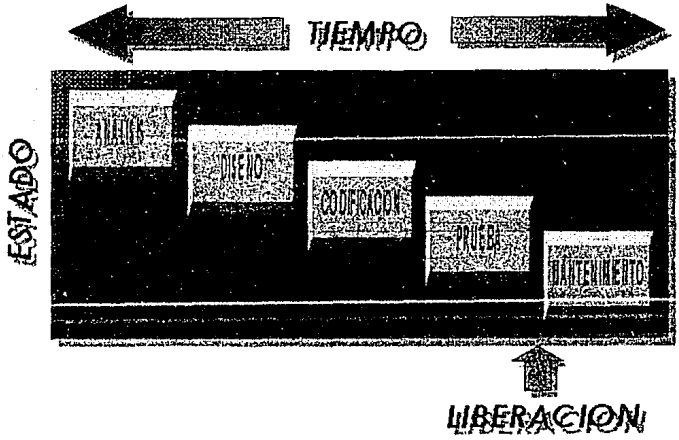


Fig. 2.6 El Ciclo de Vida del Desarrollo de Software

La intención de la metodología es que cada etapa en la secuencia debe ser activada cuando la anterior finalice, con signos explícitos en la conclusión de cada etapa para asegurarse que el proyecto está en camino. Sin embargo, la metodología permite la realimentación de cada etapa con respecto a la anterior por cuestión de flexibilidad.

Por ejemplo, cualquier defecto encontrado en la fase de pruebas es regresado a los programadores, para que puedan arreglar los problemas.

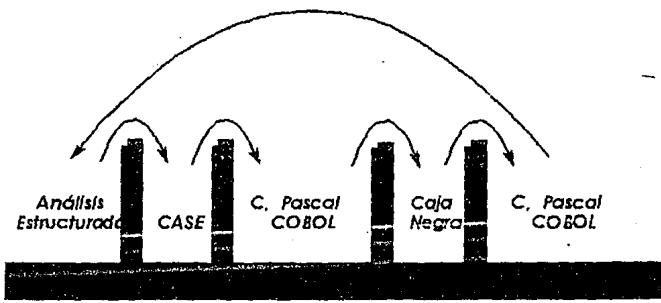


Fig. 2.7 Barreras entre Etapas

La OOP puede mejorar el ciclo de vida tradicional en dos formas. Primero, puede reducir las barreras entre etapas al dar un conjunto común de acciones a usar en cada una. En el presente, se usan términos muy diferentes en cada etapa del desarrollo, incluyendo descripciones verbales y modelos de entidad-relación para análisis de requerimientos, diagramas de descomposición funcional para diseño de sistemas, y código de programas de

lenguajes específicos para la implementación.

"La tecnología de objetos da un lenguaje común que ayuda a reducir las barreras entre las etapas" (Taylor [1992]). Los objetos del mundo real discutidos en el análisis de requerimientos son trasladados directamente a objetos del sistema en la fase de diseño, que son implementados como objetos de software en la fase de programación. Esta aproximación permite mapear en forma más directa un programa, desempeñando los requerimientos originales, haciendo un sistema más fácil de mantener y modificar.

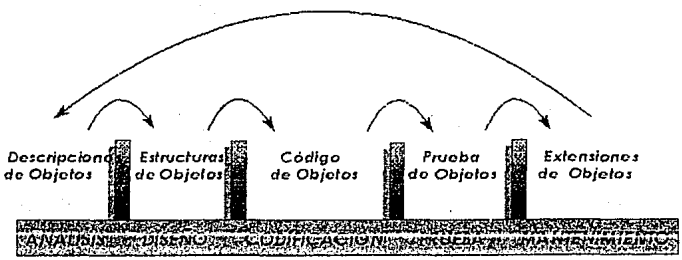


Fig. 2.8 Disminución de las Barreras Usando OOP.

La segunda forma en que la OOP ayuda al ciclo de vida del desarrollo tradicional, es dando herramientas específicas para asistir cada etapa.

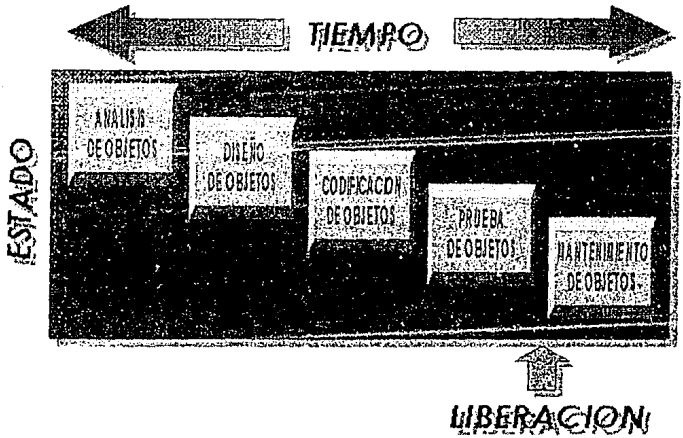


Fig. 2.9 Técnicas de Objetos para cada Etapa.

2.7.1 Análisis Orientado a Objetos

El análisis de requerimientos basado en objetos consiste esencialmente en ver todos los objetos potenciales en un sistema y, capturar sus características y relaciones en una notación formal. En esencia, consiste en crear un modelo abstracto de como se vería una solución al problema del negocio, manteniendo una correspondencia estrecha entre los objetos abstractos y sus contrapartes en el mundo real.

2.7.2 Diseño Orientado a Objetos

Una vez que los requerimientos para un sistema están bien entendidos, puede ser diseñada su estructura actual. Tradicionalmente, este es un paso muy distinto del análisis, pero la OOP lleva estas etapas muy juntas por lo que frecuentemente se confunden.

No hay notación estándar para expresar los diseños orientados a objetos. Algunos autores como Rebeca Wirsf-Brock, Brian Wilkerson y Lauren Wiener (1990) dan una notación simple para el diseño orientado a objetos junto con una nueva técnica llamada diseño manejando responsabilidades.

2.7.3 Programación Orientada a Objetos

La fase de implementación del desarrollo de software es, por supuesto, transformada en forma más radical por la OOP. Parte de la diferencia viene de la programación por sí misma, con el uso de objetos, mensajes y clases en lugar de funciones.

Una diferencia muy importante en el proceso de programación, radica en el reuso extensivo del código existente. Esto permite a los programas basados en objetos ser desarrollados extensamente al ensamblar clases existentes de objetos en nuevas formas. Además, acelera el proceso de programación considerablemente, porque ensamblar objetos existentes es mucho más rápido que escribir software desde cero.

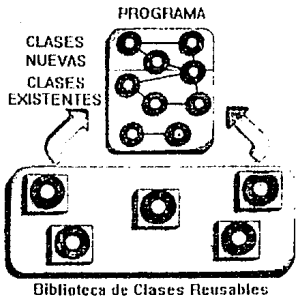


Fig. 2.10 Un Ejemplo de Reuso de Objetos.

2.7.4 Prueba Orientada a Objetos

El objetivo principal de la OOP en la prueba de software es precisamente mejorarlo, porque los nuevos programas consisten en su mayoría de objetos existentes ya probados, que han demostrado ser confiables en aplicaciones previas.

Otro beneficio importante es que la buena modularidad del software de objetos, hace más fácil aislar la causa de un mal funcionamiento.

2.7.5 Mantenimiento Orientado a Objetos

La OOP facilita el mantenimiento de programas en al menos tres formas importantes:

1.- Puede reducir la cantidad de mantenimiento, porque la calidad del software es más alta

debido al reuso extensivo de componentes ya probados.

2.- El mapeo entre objetos de software y los del mundo real, es más cercano y fácil para alguien diferente del programador original, para que pueda entender y mantener el sistema.

3.- La extensibilidad natural de los objetos de software facilitan usualmente la adición de características en una fase posterior.

El "mantenimiento de software" actualmente consiste en modificaciones y extensiones de la funcionalidad original. La extensibilidad es especialmente importante.

2.8 Limitaciones y Problemas con el Método de Cascada

El método de cascada es limitado. Los problemas que se han observado son:

- Raramente entregan las soluciones que los gerentes quieren, pues realmente no saben lo que quieren hasta que no lo ven, en este punto es imposible hacer modificaciones significativas sin ir hacia atrás o al principio del proceso.
- Se requiere personal especializado para el análisis, diseño, programación, pruebas y mantenimiento. Esto significa que todo desarrollo significativo debe ser centralizado, un requerimiento que es difícil de cumplir en el ambiente de hoy de la computación distribuida.
- Es muy difícil ejecutar como se describe. Los estudios muestran que las etapas raramente empiezan en el itinerario y casi nunca terminan cuando se supone. También, se están estableciendo nuevos requerimientos aún como partes de un programa que está sobrellevando la prueba final.

- Lleva demasiado tiempo, requiere de meses ó años desarrollar una aplicación, por lo que la mayoría de las aplicaciones son obsoletas cuando son liberadas.

2.9 Disminución del Tiempo de Desarrollo

Los objetos pueden aliviar algunos de estos problemas, pero no resolverlos totalmente.

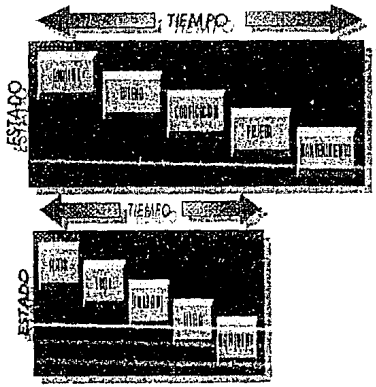


Fig. 2.11 Disminución del Tiempo de Desarrollo.

Lo que se necesita es una alternativa del método de cascada que preserve todas sus buenas cualidades, aunque hay que vencer sus crecientes limitaciones. Necesitamos ver más adentro, en la manera en que el software es construido para encontrar una solución a este dilema.

RESUMEN

- La OOP estimula el reuso de código y es más fácil de mantener que otros tipos de programación.

- Dos de sus principales cualidades son:
  1. La comprensión del sistema es fácil, ya que el modelo se parece más al sistema real.
  2. Las modificaciones al modelo del sistema tienden a ser locales.

- Un objeto es una entidad formada por datos y operaciones que actúan sobre estos.

- La delimitación definida de los datos y operaciones de un objeto es denominada encapsulamiento.

- Una parte del encapsulamiento del objeto es determinada como privada, permitiendo ocultar esta información a otros programas o desarrolladores del software, evitándose indeseados problemas colaterales.

- Un mensaje es la manera en que un objeto puede accesar o pedir la información u operaciones que se desean de otro objeto.

- Un método es un algoritmo paso a paso ejecutado en respuesta a la recepción de un mensaje, cuyo nombre es igual al nombre del método.

- Una clase es una definición o especificación genérica para un número arbitrario de objetos similares que concuerdan con tal especificación. De esta manera, una clase nos permite construir una taxonomía (o clasificación) de objetos a un nivel abstracto y conceptual.

- Una instancia es un objeto que se ha asignado y que pertenece a un tipo de clase. Como todos los objetos pertenecen a alguna clase, entonces una instancia es un objeto en si.

- El polimorfismo es la habilidad que tienen dos o más clases de objetos de responder a un mensaje con el mismo nombre, pero cada una en su propia forma.

- La herencia permite que una clase pueda tomar o heredar de otra clase ancestra (superclase) todas sus características, permitiendo que el programador sólo especifique las nuevas características distintas de las heredadas.

- Cuando una clase hereda de un solo ancestro directo en un lenguaje determinado, decimos que sólo permite herencia simple; pero, si se puede heredar de varias clases directamente entonces se dice que permite herencia múltiple.

FALTA PAGINA

No. 36

DEL PROTOTIPO AL DESARROLLO DEL SISTEMA

Introducción

El desarrollo de prototipos es un método para incrementar la utilidad del conocimiento del usuario para propósitos del desarrollo del software. Los prototipos tienen la capacidad de incrementar la productividad del software por medio del modelado del proceso de desarrollo de software. Algunas de las maneras en las cuales los prototipos afectan la productividad del software son por provisión de soporte para:

1. El proceso de requerimientos, en términos de identificación de las necesidades del usuario para transformar éstas en especificaciones del sistema.
2. Comprensión del ambiente operacional.
3. Comprensión del nivel de funcionalidad del sistema deseado en términos de lo que el sistema llevará a cambio.
4. Asistir al diseñador de software al examinar varios aproximamientos estructurales al diseño de software para las aplicaciones prometidas.

3.1 Prototipos

Por medio del uso de prototipos, los detalles del ciclo de vida del desarrollo de software son en sí modificados para acomodar las potencialidades de esta técnica. Uno de sus más importantes usos, es el aumentar la conciencia del usuario del producto de software que será desarrollado.

Además, es factible encontrar incertidumbres en el sistema y requerimientos del software, y estas pueden ser reducidas a través del uso de prototipos que proveen una retroalimentación, tanto para el desarrollador como para el usuario. Por medio de esta retroalimentación, el usuario puede participar en el desarrollo de software como una parte integral del equipo de desarrollo del mismo. Aquí hay al menos tres maneras en las cuales el usuario puede interactuar para ayudar a revelar las especificaciones originales de requerimientos:

1. Siendo parte del equipo de identificación de requerimientos, para enfrentarse mejor, tanto en la identificación inicial de requerimientos como de los que cambian durante el proceso de desarrollo.
2. Estando comprometido con la identificación de necesidades de funcionamiento operacionales, incluyendo los cambios que pueden influenciar fuertemente la instalación y mantenimiento del sistema.
3. Trabajando con los diseñadores de software, para asistir y ganar una mejor y mutua comprensión de la aplicación actual del producto, así como el tiempo extra envuelto.

Realizando este último acto, se puede asegurar que el sistema operacional incorpora la funcionalidad necesitada.

3.2 Taxonomías de Prototipos

La construcción de prototipos puede ser hecha con lenguajes estructurados u orientados a objetos, independientemente del enfoque de diseño se puede dar una clasificación somera de la construcción, sin descartar las ventajas que el enfoque orientado a objetos puede dar.

Una clasificación de prototipos es dada por Carey y Mason (1987), que identifican tres categorías de técnicas de prototipos:

1. *Prototipo versión 0.* Es un sistema funcional limitado para el cliente o usuario para evaluar e identificar los refinamientos potenciales que se necesitan.
2. *Prototipo de demostración.* Procesa un limitado rango de datos del usuario, usando un limitado número de archivos. Frecuentemente alguna porción del prototipo de demostración es llevada hacia el sistema de producción.
3. *Prototipo de simulación.* Presenta para el usuario un escenario o la interface de usuario. En éste la función eventual, aplicación orientada o solución no es desarrollada. Generalmente, sólo una demostración o versión fingida es la que esta presente. Esta interface de usuario puede ser usada en la producción actual del sistema, dependiendo de la herramienta usada para construir el escenario.

Por otra parte, Riddle y Williams (1986) identifican tres tipos de prototipos.

1. *Prototipos Evolucionados.* Son usados para la creación interactiva y evolución de un sistema. Esto representa una aproximación estructural para la creación del software y el aumento de las capacidades funcionales, permitiendo la mejor convergencia entre el desarrollador y el usuario para el apropiado sistema de software.
2. *Prototipos Experimentales.* Son usados para investigación de soluciones alternativas aproximadas. Como un resultado del prototipo experimental, el desarrollador y el usuario tendrán conductas para analizar varias soluciones funcionales para el caso bajo estudio.
3. *Prototipos Exploradores.* Con ellos el usuario explorará mayores aspectos de identificación de problemas, con asistencia del equipo desarrollador y el prototipo exploratorio.

Como las anteriores clasificaciones de prototipos podemos encontrar muchas otras, lo importante no es la clasificación, si no ver como usar los prototipos para que realmente ayuden al desarrollo del sistema global.

Un enfoque de desarrollo de software puede ser el desarrollo incremental de subsistemas, donde el ambiente para construcción de prototipos provee una aproximación para el desarrollo de subsistemas, en el cual la administración advertida permitirá el proceso entero para asegurar que el uso de recursos es efectivo y eficiente.

La seguridad del producto igualmente implementada a lo largo del proceso, hace cierto que la operación del prototipo mantenga la conciencia necesaria de los requerimientos de subsistemas. Aquí, el análisis incremental de requerimientos son puestos en el prototipo y revisados, entonces las especificaciones incrementales son desarrolladas y repasadas, continuando con el diseño de las especificaciones aprobadas, y completado por la implementación del producto.

3.3 Del Prototipo al Desarrollo

Los prototipos de software se pueden hacer para sistemas de pequeña escala en los cuales la interface de usuario es crítica. Para pequeños sistemas que contienen algunos cientos de líneas de código, los equipos para prototipos dejarán producir sistemas que son más pequeños, menos complejos y más fáciles de usar que programas desarrollados por equipos de especificación convencionales.

Los sistemas grandes pueden ser descompuestos en un número de pequeños proyectos de desarrollo, se puede sugerir usar un prototipo sólo para identificar los requerimientos de usuario y entonces cambiar al desarrollo completo de software orientado a objetos.

La decisión concerniente de cuando un prototipo será refinado en un sistema completo o será desechado, se basa en los costos y beneficios de cada alternativa. Dos factores claves son:

1. ¿Qué tanta funcionalidad se encuentra presente en el prototipo desarrollado?
2. El prototipo diseñado soportará un sistema mantenible o ¿es realmente digno de más investigación, esfuerzo y dinero?

3.4 Ventajas

Las mayores ventajas de un prototipo general para desarrollo de software son:

1. Realza la descripción de modelo resultante del ciclo de vida.
2. Es más parecido a la realidad del proceso de desarrollo y permite una interacción usual de un sistema para tomar lugar en forma más natural que en un ciclo de vida del desarrollo de software que no tenga un prototipo.
3. Se deja para control del proceso de desarrollo de software, para incorporar los aspectos de administración de recursos, configuración, verificación y validación en fases tempranas del ciclo de vida.
4. Habilita la construcción de sistemas grandes y más complejos con equipos de desarrollo más pequeños, aumentando la productividad y comunicación entre desarrolladores.
5. Habilita la retroalimentación del usuario desde las primeras etapas el proceso de desarrollo del sistema.

Para que el desarrollo de prototipos pueda ser efectivo, un prototipo debe ser desarrollado sobre un período de tiempo corto para revisión y asistencia del usuario. De esta manera, una estimación del ajuste de los requerimientos identificados y el diseño de software resultante llevan a modificaciones producto de la revisión del usuario, que son incorporadas a la siguiente interacción del prototipo.

Hay por lo menos tres clases genéricas de herramientas que serían disponibles. Estas incluyen técnicas y herramientas 4GL, partes y componentes de software reusable, especificaciones apropiadas formales y un entorno para el desarrollo de prototipos.

3.5 Herramientas de Auxilio

En los últimos años se ha innovado la ingeniería de software por medio de CASE (Ingeniería de Software Asistida por Computadora). Con CASE, las computadoras manejan el proceso de la descomposición funcional, se definen gráficamente las subrutinas en diagramas y se verifica que todas las interrelaciones entre subrutinas sigan una forma específica correcta . Los sistemas avanzados de CASE pueden actualmente construir programas completos desde estos diagramas una vez que la información del diseño ha sido introducida.

Otro acercamiento a la programación automática es representada por los lenguajes de cuarta generación (4GLs), los cuales incluyen una gran variedad de herramientas que ayudan a automatizar la generación de aplicaciones de negocios rutinarios, incluyendo generación de formas, reportes y menús.

Actualmente podemos encontrar sistemas CASE y 4GL's que generen código orientado a objetos, dando una mayor flexibilidad y poder a la creación de prototipos, permitiéndose una mejor adaptación al enfoque de diseño, y pasar sin problemas al desarrollo usando directamente el prototipo.

3.6 Tablas Históricas

Una técnica exploratoria para la construcción de prototipos es la tabla histórica (Storyboard). Este acercamiento usa una mezcla interactiva de análisis de requerimientos y simulación. La disposición de las microcomputadoras posibilita la rápida construcción de prototipos de tabla histórica en forma de pantallas desplegadas y escenarios operacionales. El prototipo de tabla histórica es una forma de validación y verificación de requerimientos para los sistemas de procesos ingenieriles. Este método provee un diseño flexible y orientado a objetos para la captura de requerimientos y especificaciones para el sistema.

Las tablas históricas son sólo uno de los propósitos de enfoque de prototipos, el propósito general es la identificación de los requerimientos de los usuarios o del software. El problema general de la identificación y especificación de requerimientos, es la pequeña necesidad de una vista detallada de lo que es realmente. Fundamentalmente, dos subniveles de requerimientos son necesarios para la determinación de los requerimientos de usuarios para el sistema de software:

1. *Requerimientos de niveles organizacionales.* Especifican la estructura del sistema, portafolios de aplicaciones y ventajas de la interface.
2. *Requerimientos de niveles de la aplicación.* Determinan requerimientos específicos del

sistema a ser implementados en la aplicación específica en el orden para satisfacer las necesidades del usuario.

Un prototipo puede ser usado también como un vehículo para entrenar usuarios sobre el uso del sistema. Sin embargo, esto requiere que el prototipo este actualizado, más que ser retirado cuando el sistema este funcionando y así contribuirá siendo operacional al mismo tiempo. Las ventajas de este enfoque para capacitar será medido contra el costo adicional del mantenimiento del prototipo por un largo período de tiempo.

3.7 Factores Importantes

En general, un mayor énfasis es encontrado en una implementación rápida y costos de desarrollo bajos en aquellos aspectos del ciclo de vida que específicamente se relacionaron con el prototipo. Gomaa (1987) cita un numero de factores que pueden asistir para lograr estos objetivos:

- *Enfasis en la interface de usuario.* Como un objetivo del paradigma de prototipos es el maximizar la interacción del usuario con el sistema, el prototipo enfatizará la interface de usuario a expensas del nivel bajo de software que no es visible al usuario.
- *Equipo pequeño de desarrollo.* Típicamente, el esfuerzo del desarrollo de software que involucra prototipos podría ser realizado por un equipo pequeño de trabajo, lo que minimizará los problemas potenciales de comunicación entre usuarios y desarrolladores.
- *Lenguaje de desarrollo para prototipos.* Un lenguaje de programación orientado a objetos o de cuarta generación facilitará el rápido desarrollo del prototipo. Debe haber énfasis en reducir el tiempo de desarrollo para obtener el prototipo y no en el desempeño del

producto terminado. Un lenguaje interpretado es ventajoso, desde él se puede llevar una rápida detección de errores de programación. Un lenguaje orientado a objetos es ventajoso por las poderosas funciones de manipulación de datos.

- *Herramientas para desarrollo rápido de prototipos.* Si la construcción ó uso de prototipos es muy usual, debe haber herramientas que posibiliten el rápido desarrollo de prototipos.

En los prototipos de software el costo inicial puede ser mayor en términos de tiempo y dinero en relación con otras aproximaciones. Sin embargo, el apropiado uso de los prototipos proveerá mejores resultados, debido a las interacciones con el usuario.

"La construcción de prototipos requiere de un compromiso por parte del usuario para proveer suficiente información durante todo el proceso de desarrollo, para que las actividades puedan proceder tranquilamente" (Sage - Palmer [1990]).

Se requiere que los desarrolladores proporcionen también información al usuario durante el proceso de desarrollo para cualquiera de los beneficios reales a ser derivado.

RESUMEN

- El uso de prototipos en el diseño y desarrollo de software, permite conocer más ampliamente los requerimientos del usuario, para una mejor calidad en la construcción del proyecto a realizar.

- La decisión concerniente a si un prototipo será refinado en un sistema completo o

desechado, se basa en los costos y beneficios del mismo; por lo que hay que evaluar su funcionalidad y si soportará el ser mantenido.

- Entre las ventajas del uso de prototipos están:
  1. Realza la descripción del modelo resultante del ciclo de vida.
  2. Es lo más parecido a la realidad del proceso de desarrollo.
  3. Se deja para control del proceso de desarrollo de software.
  4. Habilita la construcción de sistemas grandes y más complejos con equipos de desarrollo más pequeños.
  5. Habilita la retroalimentación del usuario en el proceso del desarrollo del sistema.

- El uso de sistemas CASE y 4GLs que generen código orientado a objetos, puede dar mayor flexibilidad y poder a la creación de prototipos rápidos.

- Las tablas históricas (Storyboards) son un tipo de prototipos que presentan la secuencia de pantallas y escenarios operativos que forman el sistema, ya sea impresas o por medio de una presentación por computadora.

- Se pueden considerar como factores que ayuden a lograr los objetivos del uso de prototipos a:
  1. Enfasis en la interface de usuario.
  2. Equipo de desarrollo pequeño.
  3. Lenguaje de desarrollo para prototipos.
  4. Herramientas para desarrollo de prototipos rápidos.

- Los prototipos pueden ser costosos inicialmente en términos de tiempo y dinero, pero su uso apropiado dará mejores resultados debido a la mayor comunicación con el usuario.

FRAMEWORKS

Introducción

Hay algo que es muy necesario aún con toda la metodología, lenguajes y herramientas CASE orientadas a objetos desarrollados hasta hoy. Se necesita incrementar la productividad y calidad del software por lo menos al doble. Esto requiere un nivel consistente de reuso de los requerimientos, diseño, código y prueba del software en un 80% o más, algo poco usual hoy en día. Los objetos por si mismos no pueden hacerlo, cada vez las clases de objetos son mejores, pero aún insuficientes. Ellos envuelven bloques de construcción muy pequeños para incrementar el nivel de reuso necesario. Es por ello que se necesitan bloques de construcción más grandes. Se necesitan Frameworks.

4.1 Frameworks

Los frameworks son grandes colecciones de clases colaborantes que contienen los patrones a pequeña escala y mecanismos mayores que implementan requerimientos y diseño común en un dominio específico de la aplicación.

Esto incluye no solo los frameworks actuales, que soportan una plataforma específica desarrollada (como la Mac App), sino todos los aparatos integrados de frameworks incluyendo:

1) Frameworks del dominio. Contienen las clases, patrones, mecanismos y escenarios esenciales en un dominio específico de la aplicación, tales como automatización de la industria,

telecomunicaciones, etc.

2) Frameworks de interface con el usuario. Dan una vista y un sentido común a los productos

3) Frameworks de bases de datos. Proveen independencia e interoperabilidad con múltiples sistemas operativos y plataformas de hardware.

4.2 Desarrollo por Capas de Hardware

La meta del desarrollo de software ha sido siempre la aplicación monolítica, un sistema de software contenido en sí mismo, cerrado, que es creado de cero para responder a un conjunto específico de requerimientos de una aplicación.

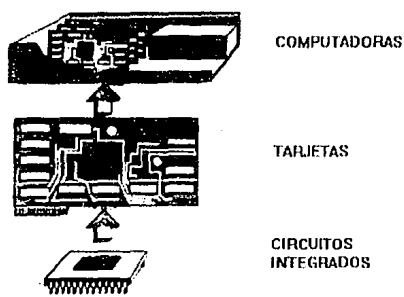


Fig. 4.1 Construcción de Hardware por Capas.

La computadora fue una vez una máquina monolítica, pero ahora son diseñadas a base de componentes ya existentes. Las computadoras son construidas en capas y cada capa usa

componentes estándares. El sistema final está abierto a modificación.

4.3 Construcción del Software como el Hardware: Software por Capas.

La aproximación de hardware tiene algunas ventajas poderosas. Ofrece más funcionalidad confiable y adaptable, mientras se obtiene máximo reuso de los componentes existentes. Los resultados son mejores para cada uno, desde el vendedor al revendedor o al usuario final.

La OOP ofrece una nueva oportunidad para mover el software fuera de la era de aplicaciones monolíticas. Usando esta, podemos tomar un acercamiento de capas al desarrollo de software usando componentes estándar, donde sea posible en cada capa.

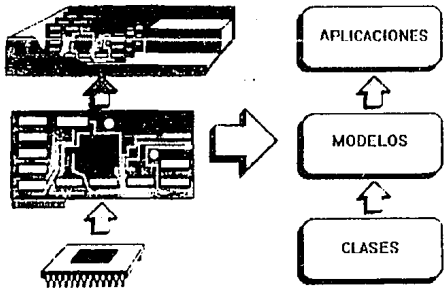


Fig. 4.2 Desarrollo de Software por Capas.

En la aproximación que se considera se asume que hay 3 capas, como con el hardware. Las capas son: *clases, modelos y aplicaciones.*

- Capa 1: Clases

La capa 1 consiste de clases estándar. El nivel más bajo de software en este acercamiento, está hecho de clases que ofrecen paquetes estándar de funcionalidad. Las clases pueden reflejar los objetos de los negocios.

Fig. 4.3 Clases simples de negocios.

- Capa 2: Modelos o Frameworks

La capa intermedia está hecha de modelos reusables de operaciones de negocios.

El uso de capas intermedias no es aún una práctica universal en el desarrollo de objetos, pero muchas compañías están cambiando a ella porque permite que muchas aplicaciones

diferentes sean construidas con las mismas estructuras básicas.

Construir modelos separados para manejar operaciones de rutina puede hacer a los sistemas de información más estables.

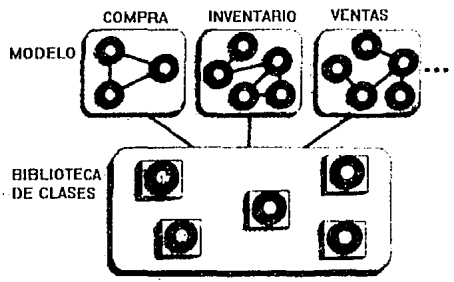


Fig. 4.4 Ejemplos de Modelos.

- Capa 3: Aplicaciones

La capa más alta en un sistema de capas consiste de las aplicaciones actuales que cumplen las necesidades actuales del negocio.

Estas aplicaciones son muy diferentes de las tradicionales aplicaciones monolíticas, lo más importante, requieren muy poco desarrollo nuevo. Como una computadora es un ensamble de tarjetas de PC, las aplicaciones de objetos quizá no sean más que nuevas formas de ejercitar modelos de negocios existentes.

También, estas aplicaciones son independientes de los detalles de cómo funcionan los negocios.

4.4 Frameworks para Prototipos

Como las aplicaciones requieren relativamente muy poca nueva construcción en el aproximamiento de capas, pueden ser generadas y desplegadas más rápido en comparación a lo tradicional de las aplicaciones monolíticas. El ensamblar aplicaciones nuevas está acompañado de un proceso conocido como prototipo rápido.

En la programación de objetos, el prototipo es el programa. El software orientado a objetos es suficientemente maleable que aún las decisiones de diseño relativamente profundas pueden ser revisadas usualmente sin reconstrucción.

Las aplicaciones basadas en objetos pueden ser usualmente desarrolladas más rápido. De hecho, si una compañía tiene una sólida fundación de clases y modelos construidos, los sistemas de trabajo completo pueden ser entregados frecuentemente en menos tiempo del que tomaría para completar un análisis de requerimientos tradicional.

Aún después de hacer la transición del prototipo a la producción del sistema, una aplicación orientada a objetos puede seguir creciendo. Como el negocio necesita cambiar, la aplicación continua para desarrollar esas necesidades sin ir hacia atrás en el ciclo de vida de desarrollo entero. Esta capacidad para cambio evolucionario es una ventaja valuada de la OOP.

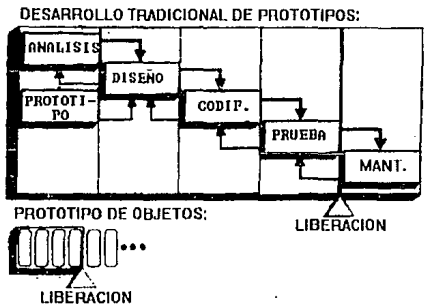


Fig. 4.5 Una nueva clase de Prototipo.

El uso de frameworks da velocidad a la construcción de prototipos rápidos. Los cuales son fácilmente extendidos. Como una aplicación orientada a objetos usa una colección relativamente pequeña de código comparada con una aplicación monolítica, es fácilmente modificada para cumplir necesidades de cambios.

Es importante enfatizar que el aumento de velocidad ganado del prototipo rápido no viene de hacer nada más rápido. En vez de esto, la velocidad se gana al hacer menos.

4.5 Mantener los Controles tradicionales

El prototipo rápido puede ser peligroso. La solución a este problema es, por supuesto, mantener los mismos controles de manejo sobre el desarrollo del software del pasado:

- El desarrollo de cada clase requiere de las etapas tradicionales de análisis, diseño,

implementación, pruebas y mantenimiento para asegurar que incluye la funcionalidad correcta.

- Similarmente los constructores de modelos deben ir a través del ciclo de vida entero y deben asegurar que sus modelos cumplen los requerimientos del negocio y que están propiamente diseñados, implementados, probados y mantenidos.
- Cada prototipo debe pasar las mismas etapas y se puede hacer tantas veces como el prototipo sea revisado y mejorado. Pero como ya se ha hecho mucho trabajo en los niveles más bajos, el proceso es muy rápido.

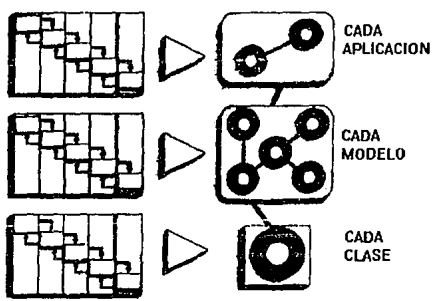


Fig. 4.6 Controles en cada Nivel.

La solución del dilema radica en realizar beneficios de orden de magnitud mientras se mantienen los controles tradicionales. No se está sólo reusando el código. Se está reusando el análisis, el diseño, las pruebas y el mantenimiento también.

4.6 Constructores de Interfaces Orientadas a Objetos.

En la construcción de entornos de Interface de Usuarios es de gran ayuda el uso de frameworks. Los frameworks son utilizados dentro del entorno orientado a objetos por herramientas visuales para la construcción de Interfaces de Usuario. Dichas herramientas abastecen de los frameworks necesarios para la construcción de la aplicación, fundamentalmente cuentan con aspectos como:

- **Manipulación Directa.**

  La manipulación directa debe estar bien adaptada a la interacción con una herramienta de construcción de Interface de Usuario, es decir, cuando la parte gráfica de la Interface de Usuario esta siendo manejada. Es sin embargo difícil de aplicar a la adecuación de un objeto de propósito general.

- **Imposición del Modelo.**

  La importancia de un modelo flexible para Interface de Usuario se debe poner de manifiesto desde el comienzo. La herramienta de construcción de Interface de Usuario se impone durante el proceso de construcción de dicha interface, dando direcciones para construir la aplicación interactiva del usuario.

- **Componente Computacional Obligatoria.**

  El resultado de la sesión laboral es un conjunto de archivos que describe la Interface de Usuario, sin embargo, los mecanismos tienen que estar provistos para definir la parte obligatoria computacional de esa aplicación. Dichos mecanismos ofrecen un frente-final

flexible que sostiene conexiones a múltiples lenguajes de programación como son Smalltalk, C++ o Eiffel, sin requerir de un lenguaje específico por el programador de Interface de Usuario.

- Ayuda y Guías.

La flexibilidad suministrada por una herramienta interactiva, con manipulación directa de múltiples entidades de Interface de Usuario, puede presentar al diseñador un conjunto muy grande de opciones, ayuda y guías tienen que estar presentes a través del proceso de construcción de Interface de Usuario, ya sea para señalar la solución más adecuada, definir un enfoque estructurado a la organización de la interface que sea coherente con el modelo, o hasta limitando las posibilidades válidas en el contexto actual.

De acuerdo con estos requisitos, los objetivos de programación interactiva, podemos considerar que la herramienta de Interface de Usuario debe tener cinco componentes incorporados:

1. **Organizador de Interface.** Permite acceso global a los objetos de interface, logra la creación de vínculos entre objetos de Interface de Usuario, y provee la interface a funciones globales como la generación de código de lenguaje orientado a objetos con todos los frameworks que se necesiten incorporar, ayuda en línea, etc.
2. **Editor de Desplegado.** Manipula objetos con propiedades de presentación como posición, tamaño, texto y mapas de bits. El editor es dividido en tres áreas: la paleta, canvas y programador de atributos, estas áreas son la interface para la instanciación, colocación y operaciones de parametrización de los objetos usados para construir la

interface.

3. **Editor de Datos.** Los objetos de datos representan estructuras abstractas usadas por los componentes interactivos y computacionales de la aplicación. El editor de datos provee un mecanismo de identificación que asocia un nombre de usuario a cada objeto. Después, estos objetos pueden ser seleccionados desde un menú de instancias creadas. La instanciación y parametrización pueden también ser presentadas desde menús que se refieren a las clases de datos.
4. **Editor Manejador.** La naturaleza de objetos manejadores de otros objetos dificulta el uso de una directa manipulación adoptada a su parametrización, en este caso, los menús permiten instanciación y parametrización, la identificación es similar al usado para objetos de datos.
5. **Editor de Diálogos.** Depende del modelo de control de diálogos que es provisto por un kit de herramientas, una primera aproximación define un dialogo manejado por eventos orientados a objetos. Por los cuales un evento o secuencia de eventos llegan a un objeto pudiendo llevar a cabo una acción predeterminada. El diálogo puede ser representado con una gráfica directa, ajustable por programación interactiva usando técnicas de manipulación directa.

La arquitectura y componentes frameworks de un entorno de construcción de interfaces, están relacionados estrechamente con los sistemas existentes. Smalltalk-80 ha sido una inspiración al acercamiento de orientación a objetos y el sistema de soporte en tiempo de ejecución.

Las herramientas y arquitecturas proveen frameworks comprensivos para el desarrollo de herramientas más especializadas que manipulen además de objetos de interface de usuario de propósito general (menús, cuadros de diálogos, cajas de listas, botones, listas de opciones, barras de desplazamiento, paletas de colores, iconos, etc.), dominios específicos de frameworks. La evolución de sistemas en esta dirección dará herramientas para la construcción de dominios específicos de aplicaciones, las cuales pueden ser conocidas como herramientas CASE orientadas a objetos.

RESUMEN

- Los frameworks son grandes colecciones de clases colaborantes que contienen los patrones a pequeña escala y mecanismos que implementan requerimientos y diseño en un dominio específico de la aplicación.

- Existen frameworks de diferentes tipos como:
  1. De dominio específico.
  2. De interface de usuario, que son los más difundidos.
  3. De base de datos.

- La colección de frameworks (usualmente de Interface de Usuario) que llevan consigo un lenguaje de programación, son comúnmente utilizados dentro de un entorno orientado a objetos por herramientas visuales y/o CASE que pueden generar código de la parte de interface de usuario, sin embargo se cuentan con pocas de estas herramientas para la construcción de otros dominios específicos. La evolución de sistemas orientados a objetos

en esta dirección dará las herramientas necesarias para el uso automatizado de más frameworks.

DESARROLLO DEL SISTEMA

Introducción

El diseño orientado a objetos es el proceso por el cual los requerimientos de software son puestos sobre una especificación detallada de objetos. Esta especificación incluye una descripción completa de los papeles respectivos, las responsabilidades de los objetos, así como la comunican con cualquier otro.

5.1 Requerimientos del Sistema

El primer paso para poder desarrollar el sistema es saber los requerimientos de este, en esta etapa se deben detallar las funciones que debe realizar el sistema, no importa que sea sencillo o complejo, una descripción detallada y concreta del problema ayudará a comprenderlo completamente. Para tener una especificación de requerimientos buena, es necesario que el analista o analistas interactuen con el usuario o usuarios para que se llegue a comprender lo que se necesita.

El usuario debe indicar que características son obligatorias y cuáles opcionales. Los requerimientos del sistema deben contemplar: el alcance del problema, lo que se necesita, el contexto de la aplicación y los requerimientos de rendimiento.

Los usuarios no deben describir los detalles internos del sistema ya que esto causa que se limite la flexibilidad de la implementación.

La forma de escribir la especificación de requerimientos también será importante, ya que con una sintaxis adecuada se podrá comprender mejor el problema y se evitarán dificultades posteriores.

Al tener la especificación de requerimientos se tiene una de las partes más importantes en el desarrollo del sistema, ya que esta sirve de base para las etapas posteriores del sistema y al concluir esta se podrá desarrollar el sistema correctamente y se evitará pagar costos posteriores en etapas subsecuentes.

5.2 Identificación de Clases

Para poder identificar las clases dentro del sistema es necesario leer cuidadosamente los requerimientos y buscar los sustantivos. Cambiar todos los plurales a singulares y hacer una lista preliminar. Después de tener una lista de todos los sustantivos hay que escoger cuáles serán las clases, para hacer esto hay que eliminar las clases que no tengan relevancia dentro del sistema y escoger como candidatas a las que si tengan relevancia, para escoger las clases del sistema se recomienda:

- Si se usa más de una palabra para el mismo concepto, hay que escoger la que sea más significativa en términos del sistema.
- Hay que tener cuidado con el uso de adjetivos. Los adjetivos se pueden usar de muchas formas. Un adjetivo puede sugerir un tipo diferente de objeto o un uso diferente del mismo. Si el uso de un adjetivo señala que el comportamiento del objeto es diferente, entonces hay que realizar una nueva clase.

- Hay que tener cuidado de las oraciones en voz pasiva, o en las que los sujetos no son parte del sistema ya que a pesar de que no se mencione este sujeto hay veces que es importante y otras en las que oraciones en voz activa mencionan sujetos o cosas que están fuera del sistema como: "el usuario".
- Modelar categorías de clases. Estas categorías se pueden convertir en superclases abstractas, pero en esta etapa hay que modelarlas como individuales, clases específicas.
- Modelar las interfaces conocidas al mundo exterior, como la interface con el usuario, o las interfaces con otros programas o para el sistema operativo tanto como sea posible.
- Modelar los valores de los atributos de los objetos, pero no los atributos por sí mismos.

El resultado de este procedimiento es la lista tentativa de clases en el sistema. Faltarán algunas clases y otras se eliminarán después.

Al descomponer el sistema en subsistemas se pueden diseñar primero algunas partes de este y posteriormente otras, así como más personas pueden diseñar una u otra parte. Esto depende del entendimiento que se tenga del problema.

5.3 Identificación de Responsabilidades

Las responsabilidades de un objeto son todos los servicios que da para todos los contratos que soporta. Un objeto juega el papel de un servidor cuando cumple una petición hecha por otro objeto. Un contrato entre dos clases representa una lista de servicios, una instancia de una clase puede pedirle un servicio a la instancia de otra clase. Todos los servicios

listados en un contrato particular son las responsabilidades del servidor para ese contrato. El contrato entre cliente y servidor no especifica como se hagan las cosas, sólo se dice que se haga.

Para poder determinar las responsabilidades de los objetos se puede usar la lista de requerimientos, de esta se escogen los verbos, de los que se seleccionan los que representan las acciones que realiza el sistema. Los nombres de las clases pueden sugerir una o más responsabilidades.

El comparar y constatar los papeles de las clases puede generar nuevas responsabilidades.

Se pueden identificar nuevas responsabilidades al examinar las relaciones entre las clases. Para identificar estas se pueden usar estas relaciones:

- La relación *"es clase de"*
- La relación *"es análoga a"*
- La relación *"es parte de"*

- La relación "es clase de"

Al hacer clases de objetos, se determinan atributos para la clasificación que se lleva a cabo; cada uno de los atributos que se encuentran implica una responsabilidad especifica, o un conjunto de estas. Los nombres de estos atributos pueden generar más responsabilidades.

Cuando se hace esta categoría de clases para especificar atributos se puede encontrar que algunas informaciones o comportamientos son comunes a todas estas clases, y todas las clases

que son *"una clase de"* alguna otra clase comparten algunas responsabilidades. Cuando una responsabilidad implicada por un atributo es identificado, se asigna la responsabilidad a la superclase de las clases que comparten el atributo.

- La relación "es análoga a"

Cuando varias clases parecen ser análogas, es un señal frecuente de que comparten una superclase común. Encontrar tal relación puede permitir el identificar una clase omitida, y asignar las responsabilidades comunes a la superclase.

- La relación "es parte de"

Distinguir entre *el todo* y *las partes* puede ayudar a determinar responsabilidades para ciertos comportamientos que deben cumplirse, al reducir el ámbito de posibilidades. Un objeto compuesto de otros objetos siempre conoce acerca de sus componentes o partes que lo forman.

Para asignar las responsabilidades de las clases es aconsejable usar tarjetas de clases y en ellas poner las responsabilidades que tienen. Con una frase para cada responsabilidad será suficiente.

5.4 Identificación de Contratos

Las colaboraciones representan peticiones de un cliente a un servidor en cumplimiento de una responsabilidad del cliente. Una colaboración es la incorporación del contrato entre un cliente y un servidor, un objeto puede cumplir una responsabilidad por sí mismo, o puede

requerir la asistencia de otros objetos. Un objeto colabora con otro, si para cumplir una responsabilidad, necesita enviar a otros objetos mensajes. Una colaboración va en una dirección si representa una petición de un cliente a un servidor. Desde el punto de vista del cliente, cada una de sus colaboraciones están asociadas con una responsabilidad particular implementada por el servidor.

Se pueden tener varias colaboraciones entre clases para poder cumplir una responsabilidad, o puede haber objetos que cumplan sus responsabilidades sin colaborar con otros.

Las colaboraciones son importantes porque revelan el flujo de control y la información durante su ejecución. Identificar las colaboraciones entre las clases permite tomar mejores decisiones acerca del diseño de la aplicación. Al identificar las colaboraciones se identifican las rutas de comunicación entre las clases. Al encontrar estas rutas se pueden identificar subsistemas de clases colaborantes. Encontrar estos subsistemas es importante para posteriormente encapsular comportamiento y conocimiento dentro del diseño.

El identificar colaboraciones entre clases forza a determinar qué clases juegan el papel de clientes y cuales de servidores, para cada contrato. Se puede usar esta información para identificar responsabilidades que no estén correctas o que falten. Se pueden encontrar responsabilidades que falten al identificar una colaboración. Si se descubre una colaboración sin una responsabilidad asociada, entonces falta una responsabilidad. Por lo tanto, hay que regresar y agregar la responsabilidad a la clase cliente. Si se descubre una colaboración donde la

clase del servidor no tiene responsabilidad de responder, se puede agregar esa responsabilidad a la clase del servidor.

El analizar los patrones de comunicación entre los objetos dentro de la aplicación también puede revelar cuando una responsabilidad ha sido asignada incorrectamente.

5.5 Identificación de Colaboraciones

Para poder determinar colaboraciones entre clases, hay que empezar por analizar las interacciones de cada clase. Examinar las responsabilidades para dependencias. Por ejemplo, si una clase es responsable por una acción específica, pero no posee todo el conocimiento necesario para cumplir esa acción, debe colaborar con otra clase (o clases) que posea el conocimiento.

Para identificar las colaboraciones, hay que hacer las siguientes preguntas para cada responsabilidad de cada clase:

1. ¿Es la clase capaz de cumplir esta responsabilidad por sí misma?
2. Si no, ¿qué necesita?
3. ¿De qué otra clase puede adquirir lo que necesita?

Cada responsabilidad que sea compartida entre clases también representa una colaboración entre esas clases. También hay que preguntar para cada clase:

1. ¿Qué hace o sabe esta clase?

2. ¿Qué otras clases necesitan el resultado de la información? Hay que checar para asegurarse de que cada clase que necesita el resultado colabora con esta clase para obtenerla.
3. Si una clase parece no tener interacciones con otras clases, debe ser descartada. Antes de hacer esto hay que revisar perfectamente las clases.

Por "no interacciones" se entiende que la clase no colabora con otra clase, y que ninguna otra clase colabora con ella. En la mayoría de los casos las clases que representan interfaces externas no son clientes de otros objetos en el sistema.

Al examinar las relaciones entre las clases se pueden identificar las colaboraciones. Hay tres relaciones útiles que son:

- La relación *"es parte de"*
- La relación *"tiene conocimiento de"*, y
- La relación *"depende de"*

- La relación "es parte de"

Esta relación puede implicar una responsabilidad para mantener información. Estas relaciones son de dos tipos: las relaciones entre clases compuestas y los objetos que las componen, y las relaciones entre clases contenedoras y sus elementos.

Una clase compuesta es responsable de contener los objetos que la componen, como manejar sus partes de alguna manera, o mantener relaciones específicas entre ellas. Una clase compuesta por lo tanto es responsable de conocer sus partes.

Las clases compuestas comúnmente cumplen una responsabilidad al delegarla a una ó más de sus partes. Las relaciones entre clases contenedoras y los elementos que contienen pueden o no requerir de una colaboración. Las interacciones entre una clase contenedora y sus elementos necesitan estar especificados claramente.

- La relación "tiene conocimiento de"

Las clases pueden saber algunas veces acerca de otras, aunque no estén compuestas de ellas. Las relaciones "tiene conocimiento de" pueden estar indicadas en la especificación por frases como "que obtiene de". Tales relaciones pueden implicar responsabilidades para conocer información y por lo tanto implican una colaboración entre la clase que tiene conocimiento y la clase que es conocida.

- La relación "depende de"

Las clases están frecuentemente conectadas con otras formas. Las relaciones "depende de" algunas veces están indicadas en la especificación por frases como "cambia con". Tales relaciones pueden implicar una relación "tiene conocimiento de" o pueden implicar la existencia de una tercera parte que forma la conexión.

5.5.1 Registro de Colaboraciones

Para poner las colaboraciones que se han encontrado, hay que tomar la tarjeta para la clase que juega el papel de cliente. En esta, hay que escribir el nombre de la clase que juega el papel de servidor. Hay que escribir ese nombre directamente a la derecha de la responsabilidad de la colaboración que sirve para ayudar a cumplir.

Si una responsabilidad requiere algunas colaboraciones, hay que escribir el nombre de cada clase requerida para cumplir la responsabilidad. Si algunas responsabilidades requieren una clase para colaborar con otra misma clase, hay que poner algunas colaboraciones, una para cada responsabilidad.

Hay que asegurarse de que una responsabilidad existe para cada colaboración asignada. Si una colaboración ocurre con una superclase que da un servicio definido por una superclase, la responsabilidad correspondiente será asignada en la tarjeta de la superclase. Los servicios dados por una clase incluyen los listados en su tarjeta y las responsabilidades que hereda de su superclase.

Si cumplir una responsabilidad requiere colaboración con otras instancias de la misma clase (o instancias de sus superclases), hay que asignar esa colaboración también. Para cumplir esta responsabilidad se necesita comunicación entre dos objetos distintos, un hecho que debe ser registrado.

5.6 Identificación de Jerarquías

Hay tres herramientas que se usan para tener una perspectiva más global de las relaciones de herencia en un sistema:

- Gráficas de Jerarquía
- Diagramas de Venn
- Contratos

5.6.1 Gráficas de Jerarquía

Una gráfica de jerarquía es una herramienta que presenta una representación gráfica de las relaciones de herencia entre las clases relacionadas.

Las clases están representadas por rectángulos etiquetados con los nombres de las clases. La herencia se indicada por una línea de la superclase a la subclase.

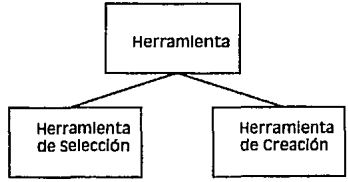


Fig. 5.1 Gráfica de Jerarquía Simple

Esta gráfica indica que *Herramienta de Selección* y *Herramienta de Creación* son subclases de *Herramienta* y heredan de ella.

Un sistema dado puede requerir más de una gráfica para describir las jerarquías de herencia de todas las clases dentro de el.

5.6.2 Distinción entre clases abstractas y concretas

Las clases abstractas están diseñadas sólo para heredar. Las instancias de las clases abstractas nunca se crean como el sistema trabaja.

Las clases concretas están diseñadas para ser instanciadas. Están diseñadas primero, así que sus instancias son útiles, y también pueden ser útiles para heredar.

Las clases abstractas existen solamente para agrupar comportamiento que es común para más de una clase y ponerlo en un lugar para cualquier número de subclases para usarlas. En general, las clases representan categorías de clases ya sean abstractas o concretas.

5.6.3 Diagramas de Venn

Otra herramienta usada para ver las relaciones de herencia son los diagramas de Venn. Si se ven las clases como conjuntos de responsabilidades, podemos usar diagramas de Venn para mostrar que responsabilidades están contenidas entre clases comunes. Por ejemplo la jerarquía de clases de la fig. 5.1 tendría el diagrama de Venn como se muestra en la fig. 5.2

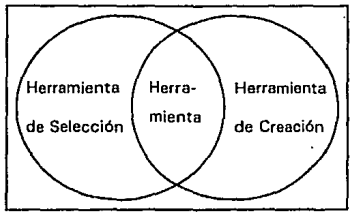
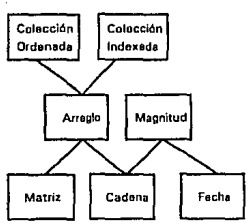


Fig. 5.2 Diagrama de Venn

El diagrama de Venn siguiente muestra una gráfica de jerarquía más compleja.



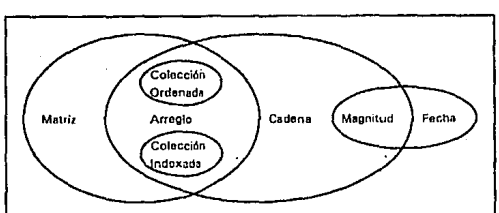


Fig. 5.3 Diagrama de Venn más complejo

5.6.4 Relaciones Jerárquicas

Para identificar y resolver los problemas referentes a las relaciones de herencia entre clases se recomienda:

- Modelar una jerarquía "tipo de"
- Agrupar las responsabilidades comunes tanto como sea posible.
- Asegurarse de que las clases abstractas no hereden de las clases concretas
- Eliminar las clases que no agregan funcionalidad

Una clase debería heredar de otra clase sólo si soporta todas las responsabilidades definidas por esa otra clase. Otra forma de decir esto es que la herencia debería modelar relaciones es "tipo de": cada clase debería ser un tipo específico de sus superclases. Las subclases deberían soportar todas las responsabilidades definidas por sus superclases, y posiblemente más. Asegurar esto hará las clases más reusables porque esto hace más fácil ver dónde, en una jerarquía existente, una nueva clase debería ser puesta.

Los diagramas de Venn pueden ayudarnos a ver donde fallan las subclases para soportar todas las responsabilidades de sus superclases. Cuando una subclase soporta correctamente las responsabilidades definidas por sus superclases, sus responsabilidades cubrirán completamente las de sus superclases como en la fig. 5.4.

Si una subclase soporta sólo parte de las responsabilidades definidas por sus superclases, aparecerá como se muestra en la fig. 5.5.

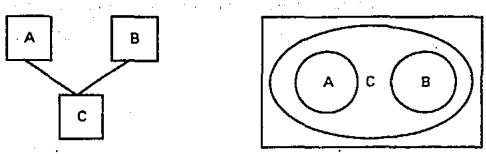


Fig. 5.4 Forma correcta de las responsabilidades de una subclase

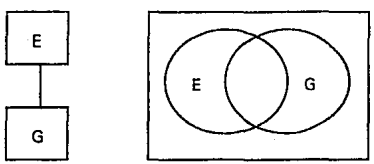


Fig. 5.5 La clase G solo soporta parte de las responsabilidades

Cuando el comportamiento de una subclase incluye sólo parte de las responsabilidades definidas por sus superclases, hay que crear una clase abstracta con todas las responsabilidades comunes a la clase y a la superclase de la que se hereda, como se muestra en la fig. 5.6

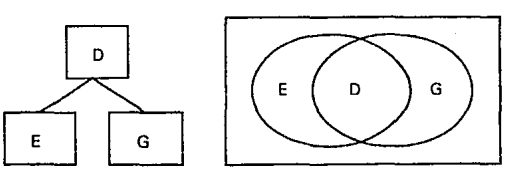


Fig. 5.6 Las clases E y G heredan todas las características de D

Agrupar las responsabilidades comunes tanto como sea posible. Si un conjunto de clases soportan una responsabilidad común, deben heredar esa responsabilidad de una superclase común.

Si una superclase común aún no existe, hay que crear una, y mover las responsabilidades comunes a ellas. Una manera de agrupar las responsabilidades al máximo en la jerarquía es diseñar tantas clases abstractas como sea posible. Cuando se ha determinado cuantas clases abstractas están presentes en el diseño, hay que especular en las clases abstractas que pueden encapsular comportamiento que podría ser reusado por subclases existentes y futuras. Hay que buscar atributos comunes y responsabilidades duplicadas. Hay que definir tantas clases abstractas como parezca razonable para capturar las abstracciones en el diseño presente o futuro.

En general, los usos más concretos que se pueden pensar para abstraer funcionalidad, probablemente sean las pruebas de tiempo y las mejoras de software. Sólo se necesita una responsabilidad para definir una superclase abstracta, pero se necesitan al menos dos subclases específicas antes de que se desee diseñar una abstracción útil.

El beneficio principal de diseñar cuidadosamente jerarquías con clases abstractas se evidencia por sí misma cuando se quiere agregar más funcionalidad a la aplicación existente. Se reusa más comportamiento si se diseñan tantas clases abstractas como sea posible. Definir tantas clases abstractas como sea posible significa que se ha agrupado tanto comportamiento común como se ha podido. La tarea de agregar nueva funcionalidad puede empezar con una cantidad clara de comportamiento ya definido por uno. Uno puede hacer subclases de estas

clases abstractas y definir sólo el nuevo comportamiento necesario para capturar las especificaciones de la nueva funcionalidad. Con ese nuevo comportamiento agregado, la implementación, la prueba y el mantenimiento del diseño es más fácil.

La alternativa involucra crear una subclase para una jerarquía en la que el comportamiento no ha sido abstraído tan claramente. Esto requiere más esfuerzo porque es más difícil entender el comportamiento de cada clase existente, y es más difícil asegurar que la nueva clase no está violando cualquier comportamiento previamente establecido.

Hay que asegurarse de que las clases abstractas no hereden de las clases concretas. Las clases abstractas, por su naturaleza, soportan sus responsabilidades de manera independiente a la implementación, por lo tanto no deben heredar de las clases concretas, que dependen específicamente de la implementación. Si el diseño actual tiene tal herencia, se puede resolver el problema al hacer otra clase abstracta de donde las clases abstracta y concreta puedan heredar su comportamiento común.

Las clases que no tengan responsabilidades deben descartarse. Si una clase hereda una responsabilidad que se implementará en una forma única, entonces añade funcionalidad a pesar de no tener responsabilidades por sí misma, y deber ser conservada. Por otra parte, las clases abstractas que no definen responsabilidades no tienen uso.

5.6.5 Contratos

Un contrato sirve para agrupar las responsabilidades de una clase que están relacionadas de alguna forma. Ayuda a asignar y reasignar responsabilidades, como grupos de responsabilidades dentro de un contrato sencillo que deben ser reasignadas como un grupo.

Un contrato define un conjunto de peticiones que un cliente puede hacer a un servidor. El servidor garantiza responder a las peticiones. Las responsabilidades encontradas en la fase exploratoria son la base para determinar los contratos soportados por una clase.

Una clase puede soportar uno ó más contratos distintos. La palabra "contrato" no es sólo un nombre para una responsabilidad: una responsabilidad es algo que un objeto hace para otros objetos, ya sea ejecutar una acción ó responder con alguna información. Un contrato define un conjunto de responsabilidades cohesivas de las que un cliente puede depender. La cohesión entre responsabilidades es una medida de que tan relacionadas están esas responsabilidades con otras.

Una clase puede soportar cualquier número de contratos. Cada responsabilidad será parte cuando mucho de un contrato, pero no todas las responsabilidades serán parte de un contrato. Algunas responsabilidades representan el comportamiento que una clase debe tener, pero que no puede ser solicitado por otros objetos. Estas son responsabilidades privadas. Se pueden determinar qué responsabilidades pertenecen a qué contratos al seguir los siguientes puntos:

- Agrupar las responsabilidades por los mismos clientes
- Maximizar la cohesividad entre las clases
- Minimizar el número de contratos

- Agrupar las responsabilidades por los mismos clientes

Una manera de encontrar las responsabilidades cohesivas es buscar las responsabilidades que serán usadas por los mismos clientes. Mientras sea posible que algunas clases sirvan a clientes diferentes con cada responsabilidad, normalmente es el caso de que dos o más responsabilidades definidas para una clase servirán a los mismos clientes. Puede ser significativo que esas mismas responsabilidades sean usadas siempre juntas y al ser así se deben contemplar en el diseño. Al ocurrir esto sólo será necesario hacer un sólo contrato en el que se agrupen las responsabilidades usadas por los mismos clientes.

- Maximizar la cohesividad de las clases

Se puede hacer una jerarquía de clases más fácil de refinar al maximizar la cohesividad de los contratos soportados por las clases dentro de ella. Una clase debe soportar un conjunto cohesivo de contratos.

Maximizar la cohesión minimizará el número de contratos soportados por cada clase. Pero también puede ser que la jerarquía de clases se haga demasiado grande y confusa y que se pierda encapsulamiento de la inteligencia del sistema al tener que crear un número mayor de clases.

Esto hace que se deba tener un balance entre un número pequeño de clases y la facilidad de entender y reusar las clases. El principio de cohesión ayuda a lograr este balance.

- Minimizar el número de contratos

Una clave para probar y mantener el diseño es la comprensibilidad. El sistema será más comprensible si hay menos detalles que comprender. Por esta razón se debe minimizar el número de contratos en un diseño, sin violar la cohesión de los contratos; la mejor manera de reducir el número de contratos es buscar responsabilidades similares que puedan ser generalizadas.

Para maximizar el reuso, un conjunto de clases que soportan un contrato común deben heredar ese contrato de una superclase común. Los contratos deben estar en términos generales. Por lo tanto hay que definir los contratos soportados por una superclase. De hecho, un contrato debe ser definido por una clase, y soportada por todas sus subclases.

5.6.6 Aplicar las recomendaciones

Una técnica simple para definir contratos es empezar definiendo contratos para las clases principales de la jerarquía. Los nuevos contratos necesitan ser definidos sólo para las subclases que añadan nueva funcionalidad, que sean usados por clientes distintos. Hay que examinar las responsabilidades agregadas por cada subclase y determinar si representan nueva funcionalidad, o si sólo son formas específicas de expresar responsabilidades heredadas y son por lo tanto parte del contrato heredado.

Hay que examinar cada tarjeta de clase. Determinar el o los servicios distintos ofrecidos por la clase, y asignar cada responsabilidad listada en un contrato apropiado. Hay que escribir una oración que describa cada contrato, y asignarle un número único. Hay que numerar las responsabilidades de acuerdo al contrato al que han sido asignadas.

En este momento se han identificado los servidores para cada uno de los contratos listados. Los clientes son las clases que confían estas responsabilidades . Por cada colaboración, hay que determinar que contrato representa esa colaboración.

5.6.7 Modificar el Diseño

Usando las recomendaciones, hay que identificar y resolver los problemas de la jerarquía de herencia. Hay que crear superclases abstractas o concretas, tal como sea necesario y asignarles las responsabilidades como sea apropiado. Hay que remover las clases innecesarias y reasignar otras responsabilidades como se requiera para producir jerarquías de clases que puedan ser reusadas y extendidas fácilmente.

Una vez que se ha modificado el diseño, hay que rehacer las tarjetas y las gráficas para que correspondan al nuevo estado. Hay que revisar el sistema. Para cada colaborador, hay que asegurarse de que hay una responsabilidad correspondiente y viceversa. Una vez más hay que asegurarse de que cada objeto se está comunicando con el resto del sistema de una manera apropiada.

Después de que se ha retrabajado la jerarquía de clases, hay que definir contratos para todas las clases en el diseño. En esta etapa, los contratos sirven como indicadores generales de los distintos usos de cada clase.

5.7 Identificación de Subsistemas

Hay dos herramientas útiles para ayudarnos en el diseño de un sistema y son:

- Gráficas de colaboraciones
- Tarjetas de Subsistemas

5.7.1 Gráficas de colaboraciones

Una gráfica de colaboraciones despliega las colaboraciones entre clases y subsistemas en forma gráfica. Se pueden usar la gráficas para ayudar a identificar áreas de complejidad innecesaria, duplicación o lugares donde la encapsulación se viole. Con una gráfica de colaboraciones de la aplicación, se pueden identificar los subsistemas, simplificar los patrones de comunicación y finalmente producir un diseño más limpio y comprensible.

Las gráficas de colaboraciones representan clases, contratos y colaboraciones. Además, muestran relaciones de superclase-subclase. Una subclase soporta todos los contratos definidos por su superclase. Por lo tanto, en una gráfica de colaboraciones, una superclase representa los contratos soportados por todas sus subclases.

Las clases están mostradas como rectángulos etiquetados, como en la fig. 5.7.

Fig. 5.7 Representación de una clase

Las subclases están gráficamente anidadas dentro de los lados de sus superclases. Por ejemplo, la jerarquía de clases mostrada en la fig. 5.1 está representada en la gráfica de colaboraciones de la fig. 5.8

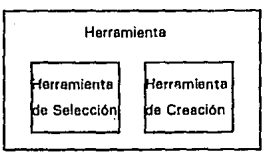


Fig. 5.8 Representación de la figura 5.1

Los contratos se muestran como pequeños semicírculos dentro de los límites de la clase a la que pertenecen. Hay que dibujar un semicírculo por contrato. Hay que numerar el semicírculo que representa cada contrato con el número asignado a él, como en la fig. 5.9.

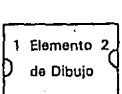


Fig. 5.9 Clase con sus contratos

Las colaboraciones entre clases están representadas por una flecha de un cliente a un contrato soportado por un servidor, como en la fig. 5.10. Si dos objetos colaboran con una clase por medio del mismo contrato, hay que dibujar una flecha al mismo semicírculo. De otra forma, hay que dibujar las flechas a los semicírculos que representan los contratos diferentes.

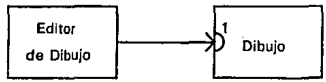


Fig. 5.10 Representación de un contrato

5.7.2 Subsistemas

Los subsistemas son grupos de clases, o grupos de clases y otros subsistemas, que colaboran entre sí mismos para soportar un conjunto de contratos. Por fuera del subsistema, el grupo de clases puede ser visto trabajando junto para dar una unidad delimitada de funcionalidad claramente. Por dentro los subsistemas revelan tener una estructura compleja, consisten de clases y subsistemas que colaboran uno con otro para soportar contratos distintos que contribuyen al comportamiento total del sistema.

Los subsistemas se identifican al encontrar un grupo de clases, cada uno de los cuales cumple diferentes responsabilidades, colaboran cercanamente con otras clases en el grupo para poder cumplir conjuntamente una gran responsabilidad. Tales responsabilidades aparecen por fuera del subsistema como contratos discretos. Si el grupo de clases colabora para cumplir un

propósito común, forman un subsistema. Un subsistema no es sólo un conjunto de clases. Debe formar una buena abstracción.

No hay diferencia conceptual entre las responsabilidades de una clase, un subsistema de clases y aún en una aplicación entera; es simplemente un problema de escala, y la cantidad de riqueza y detalle en el modelo.

5.7.3 Contratos de Subsistemas

Los subsistemas como las clases soportan contratos. Para determinar los contratos soportados por un subsistema, hay que encontrar todas las clases que proveen servicios a clientes fuera del subsistema.

Los subsistemas son un concepto usado para simplificar un diseño. La complejidad de una aplicación grande puede ser tratada primero al identificar los subsistemas dentro de ella, y tratando esos subsistemas como clases. Se puede descomponer la aplicación en subsistemas, y repetidamente descomponer esos subsistemas hasta que toda la riqueza requerida y detalle haya sido modelada.

5.7.4 Tarjetas

Una tarjeta de clase contiene información de un objeto que se registrará dentro del desarrollo del sistema. La tarjeta muestra el nombre de la clase, indica si es de tipo abstracta o

concreta, menciona cuales son las superclases y subclases inmediatas, lista sus responsabilidades asignadas, y si hay alguna colaboración con cada una de las responsabilidades.

Clase: *Nombre*	
Superclases: *Lista de Superclases*	
Subclases: *Lista de Subclases*	
Responsabilidades	**Colaboraciones**
...	...

Fig. 5.11 Tarjeta de una clase

Cuando se han identificado subsistemas, hay que escribir sus nombres debajo del índice de las tarjetas, un subsistema por tarjeta. Luego se hace una descripción pequeña del propósito general atrás de la tarjeta del subsistema. Hay que poner cada contrato requerido por los clientes externos al subsistema. Además de cada contrato, anotar la delegación a la clase interna o al sistema que soporta el contrato actualmente. Las tarjetas de subsistemas aparecen en la fig. 5.12

Subsistema : *Subsistema de dibujo*	
Accesar un dibujo	Dibujo
Modificar parte de un dibujo	Elemento de Dibujo
Desplegar un dibujo	Dibujo

Fig. 5.12 Tarjeta de un subsistema

Cuando se han identificado los subsistemas, hay que regresar a las tarjetas de clase y modificar sus colaboraciones para que reflejen esos cambios. Si una clase fuera de un subsistema colabora con una clase dentro del subsistema, luego se cambia esta a una colaboración con el subsistema. Anotando en la tarjeta del subsistema la delegación a esta clase agente.

5.7.5 Representación gráfica de las colaboraciones

Los subsistemas se muestran en la gráfica de colaboraciones al dibujar un rectángulo con esquinas redondeadas encerrando las clases y los subsistemas que las comprenden, como en la fig. 5.13

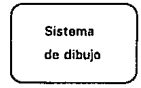


Fig. 5.13 Representación de un subsistema

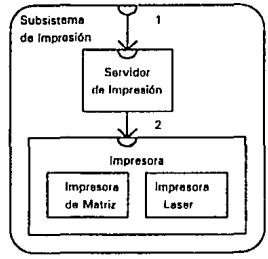


Fig. 5.14 Representación de un contrato en un subsistema

Cuando se dibuja un subsistema, las clases y los subsistemas que lo encapsulan, hay que dibujar una flecha del contrato del subsistema a la clase que actualmente soporta el contrato, como en la fig. 5.14.

Cuando se ha dibujado la gráfica, hay que buscar clases acopladas fuertemente. El acoplamiento entre dos clases es una medida de que tanto dependen una de la otra.

Las clases fuertemente interdependientes pueden ser conectadas por una colaboración frecuentemente usada y conectarla al resto de la aplicación por una menos usada.

Una manera de determinar si un grupo forma un subsistema es tratar de nombrarlo. Si se puede nombrar un número de clases, se ha nombrado el papel más grande que cooperan para cumplir. Además se debe ser capaz de establecer el propósito del subsistema.

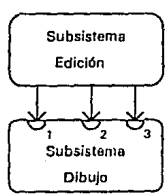


Fig. 5.15 Colaboraciones de alto nivel

Como antes, podemos aplicar el principio de abstracción de ocultamiento de información a las gráficas que muestran subsistemas. Se hace esto al abstenerse de dibujar las

clases y los subsistemas dentro de un subsistema, por lo que nos permite pensar en el nivel más alto de diseño. Una vez que se ha identificado un subsistema, se puede dibujar la gráfica, ocultando sus clases componentes y subsistemas. Esto da una vista a un nivel más alto de las colaboraciones en la aplicación.

5.7.6 Simplificar Interacciones

Los puntos básicos para simplificar los patrones de colaboración son los siguientes:

1. - Minimizar el número de colaboraciones que una clase tiene con otras clases o subsistemas.
2. - Minimizar el número de clases y subsistemas a las que un subsistema delega.
3. - Minimizar el número de contratos diferentes soportados por una clase o subsistema.

1.- Una manera de cumplir esto es centralizar las comunicaciones que fluyen en un subsistema. Se puede crear una nueva clase o subsistema para que sea el intermediario principal de las comunicaciones o se puede usar una existente que cumpla este papel.

2.- En un subsistema bien diseñado el número de clases o subsistemas que tienen contratos delegados a ellos debe ser mínimo. Para encapsular las clases y los subsistemas apropiadamente dentro de un subsistema es más fácil manejar la complejidad y adaptarlo al cambio. Un subsistema bien diseñado tiene pocas clases o subsistemas que soportan directamente sus contratos, y un gran número de colaboraciones entre clases internas y subsistemas.

Este principio va de la mano con la recomendación de centralizar comunicaciones ya que si una clase o subsistema sirve como intermediario de comunicaciones para un subsistema, entonces los contratos son delegados principalmente o solamente al intermediario.

3.- Un subsistema que tiene muchos contratos concentra mucha inteligencia de la aplicación. Por ejemplo si un subsistema soporta demasiados contratos, quizá sus clases podrían ser divididas en varios subsistemas cohesivos, cada uno soportando uno o dos contratos. Una vez que estos nuevos subsistemas han sido identificados, quizá las comunicaciones entre ellos puedan ser simplificadas como se muestra en la parte derecha de la Fig. 5.16

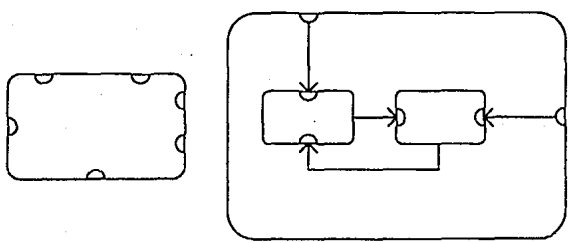


Fig. 5.16 Simplificación de las comunicaciones entre subsistemas

Una vez que se han dividido los contratos de una clase compleja entre algunas más simples, se necesitan examinar los nuevos patrones de comunicación. Estas nuevas clases forman nuevas colaboraciones. Al redibujar la gráfica de colaboraciones se pueden identificar

los lugares donde las nuevas clases pueden ser encapsuladas fácilmente dentro de subsistemas existentes o lugares donde éstos no son muy apropiados.

5.7.7 Verificación del Diseño

El objetivo de analizar colaboraciones es simplificar el diseño y reducir el número de clases que son dependientes de otras. Si las interacciones complejas requieren trabajo extensivo, los cambios al diseño inicial pueden ser extensivos. Hay que redibujar las gráficas de colaboraciones para reflejarlo en el nuevo diseño. Una vez que se ha reajustado la gráfica de colaboraciones, hay que ir a través de un conjunto completo de escenarios para asegurarse de que todas las nuevas rutas de comunicación trabajan como deben. Verificar si los cambios que se hicieron han simplificado las colaboraciones y han reducido el acoplamiento entre las clases y los subsistemas, asegurandose de actualizar las tarjetas de clases y subsistemas para que reflejen las nuevas colaboraciones y responsabilidades. Las jerarquías de clases también necesitan ser redibujadas para reflejar las nuevas abstracciones o clases unidas.

5.8 Documentación del Diseño

Esta etapa busca refinar y formalizar lo elaborado en las etapas previas, para obtener la documentación completa del sistema a la que se ha llegado.

Primero hay que obtener el protocolo o el conjunto de firmas correspondiente a cada clase del sistema global. Para lograr esto, podemos realizar las siguientes acciones:

- Usar el mismo nombre para cada operación conceptual, cada vez que sea encontrada dentro del sistema.
- Asociar una misma operación conceptual con cada nombre del método.
- Si las clases llenan las mismas responsabilidades específicas, hay que mostrarlos explícitamente en la jerarquía de herencia.
- Construir firmas generalmente usuales.
- Definir valores por default.

Una vez hecho esto se puede hacer una descripción más detallada y formal de cada una de las clases y subsistemas. Se refinan las tarjetas correspondientes, se ordenan y se les agregan referencias a otras tarjetas y gráficas. Para tener más opciones de búsqueda se pueden añadir tarjetas que especifiquen los contratos entre clases, estas deben contener:

- El nombre del contrato y su descripción.
- El nombre de la clase servidora.
- Los nombres de las clases clientes.

RESUMEN

- Identificar Requerimientos

  El comienzo del diseño orientado a objetos es exploratorio. Se debe empezar conociendo los requerimientos del sistema, haciendo una descripción detallada y concreta del problema, y de las funciones que el sistema a implantar debe desarrollar; contemplándose: el ámbito del problema, lo que realmente se necesita, el contexto de la aplicación y el rendimiento que debe presentar.

- Identificación Inicial de Clases

  Para identificar las clases que forman el sistema del análisis de requerimientos, se procede a:

  Se debe leer dicha descripción cuidadosamente y buscar los *sustantivos*; cambiar los plurales a singulares y hacer una lista preliminar.

  Identificar de la lista cuales pueden ser candidatos a superclases abstractas al agrupar aquellas clases que comparten atributos comunes.

  Escribir una sentencia pequeña del propósito de cada clase.

  Al final se tendrá un conjunto de clases, de las cuales faltarán algunas y sobraran otras que posteriormente se eliminaran.

- Identificar Responsabilidades

  Las responsabilidades de un objeto son todos los servicios que da para los contratos que soporta. Para poder determinarlas se puede comenzar desde la descripción detallada de requerimientos y:

  Leer la lista de requerimientos y seleccionar los *verbos* que acompañan a los sustantivos, de los que se seleccionaran las acciones que realizan las clases.

  Observar los nombres de las clases que pueden sugerir responsabilidades.

  Identificar responsabilidades implicadas de las relaciones entre clases.

- Identificar Contratos y Colaboraciones

  Las colaboraciones representan peticiones de un cliente (un objeto) a un servidor (otro objeto) en cumplimiento de una responsabilidad del cliente. De esta manera la colaboración entre clases es una incorporación de contrato entre cliente y servidor. Un

objeto colabora con otro(s), si para cumplir una responsabilidad necesita enviar a otro(s) objeto(s) mensajes. Las colaboraciones revelan el flujo de control e información del sistema; al identificarlas se encuentran las rutas de comunicaciones entre clases. Para poder determinar las colaboraciones se debe:

Examinar las responsabilidades asociadas con otras clases.

Preguntarse lo siguiente: la clase es capaz de cumplir la responsabilidad por si misma?; De que otra clase puede adquirir el servicio? y A que otra clase le puede dar servicio?.

Descartar las clases que no colaboran y que colaboran con otras clases diferentes.

- Identificar Jerarquías

Las jerarquías presentan las relaciones de herencia entre las clases conectadas; para hallarlas se debe contar con:

Gráficas de jerarquías (como organigramas).

Distinguir las clases abstractas (diseñadas solo para heredar) de las clases concretas (diseñadas para heredar y ser instanciadas)

Ver las clases como conjunto de responsabilidades y usar diagramas de Venn para mostrar que responsabilidades están contenidas entre clases.

Agrupar las responsabilidades que sean comunes, lo más posible.

Verificar que las clases abstractas no hereden de las clases concretas.

Eliminar clases que no añadan funcionalidad.

- Identificar Subsistemas

  Los subsistemas son grupos de clases que colaboran entre si mismos para soportar un conjunto de contratos; para identificarlos se debe contemplar lo siguiente:

  Dibujar una gráfica de colaboración completa del sistema entero.

  Nombrar los subsistemas dentro del diseño señalando las colaboraciones que son frecuentes y complejas.

  Simplificar las colaboraciones con y entre subsistemas.

- Construcción de Protocolos

  A través de las etapas anteriores de desarrollo que se han citado, se han refinado poco a poco el conjunto de clases que forman el sistema global; quedando solo por hacer:

  Refinar las responsabilidades en conjunto de signaturas que incrementen la utilidad de las clases.

  Escribir la especificación formal de diseño para cada clase.

  Escribir la especificación formal de diseño para cada subsistema.

  Escribir la especificación formal para cada contrato.

  Al final el diseño debe contar con toda la documentación que incluye:

  Gráfica de cada jerarquía de clases.

  Diagramas de Venn que muestren las responsabilidades que son comunes entre clases.

  Gráfica de las rutas de colaboración para cada subsistema.

  Tarjeta de especificación para cada clase.

  Tarjeta de especificación de contrato por cada clase y subsistema.

PRUEBA

Introducción

Probar un producto es relativamente independiente del método de desarrollo usado.

Las actividades de prueba se dividen normalmente en verificación y validación. La verificación es ver si el resultado final es lo que se quería.

La prueba está integrada actualmente en todas las actividades, la prueba trabaja mejor si empieza al mismo tiempo que el trabajo de análisis. Entre más integrada este la prueba, es mejor. El desarrollo de software es una actividad incremental que consiste de análisis, construcción y prueba.

Las actividades de prueba concernientes al producto final durante la ejecución del código del programa son llamadas algunas veces certificación para diferenciarlas de otros tipos de actividades de calidad.

La prueba también puede ser hecha por ejemplo, a nivel de documento al verificar que el análisis y el diseño están hechos apropiadamente.

El aspecto más importante de la prueba es actualmente la actitud hacia ella. Se debe reconocer que la prueba lleva tiempo y el costo debe ser considerado. "Las actividades de prueba toman quizá más del 30% del costo del desarrollo entero, pero pueden exceder algunas

veces el 50%" (Jacobson, [1992]). La prueba es consecuentemente una gran parte del desarrollo y debe ser planeada de la misma manera que el análisis y la construcción. La prueba debe por lo tanto estar incluida en el plan del proyecto. Se hace mucho esfuerzo al planear la prueba, no debe ser algo que se realice todo junto al final de la fase de desarrollo.

6.1 El Propósito de la Prueba.

Hay algunos conceptos importantes cuando se aplica una prueba. Una falla ocurre cuando un programa falla en su comportamiento. Por lo tanto una falla es una propiedad (estática) del sistema en ejecución. Una falla existe en el código del programa. Cuando el código es incorrecto, la falla puede ser arreglada al cambiar el código. Una falla, si se encuentra, puede causar una avería. No hay falla si el programa no puede averiarse. Un error es una acción humana que provoca fallas en el software, por lo tanto un error puede llevar a la inclusión de una falla en el sistema, haciendo que el sistema se averíe.

La primera lección que se aprende acerca de la prueba es que uno nunca puede probar que el programa nunca fallará, se puede mostrar sólo que contiene fallas. Una prueba que ha encontrado muchas fallas es una prueba exitosa y no lo opuesto, si se ha hecho una construcción exitosa, se tendrá una prueba no exitosa, y si la construcción fue deficiente, se tendrá una prueba exitosa.

El propósito de probar es encontrar fallas, la prueba es por lo tanto un proceso destructivo para alguna extensión del sistema, se debe mostrar que algo es incorrecto. Por lo tanto es inapropiado que uno mismo pruebe su propia construcción ya que no se siente natural

trabajar solo para probar que uno mismo ha hecho un error, sin embargo, se puede ayudar a probar partes principales en un departamento especial de pruebas. Esto se hace normalmente durante la prueba de integración. A los desarrolladores se les debe dar la oportunidad de identificar estas faltas.

Una vista alternativa de calidad en el software es hacerlo correctamente la primera vez. Con lo que las faltas que se introduzcan deben ser eliminadas tan rápido como sea posible. Esta es una de las principales ideas en el aproximamiento de Ingeniería de Software de cuarto limpio.

La prueba se hace generalmente de abajo hacia arriba. Se empieza por probar los módulos más pequeños, y como se va completando su construcción y no se encuentran más defectos en estos, se ponen juntos y se prueba el siguiente nivel y así sucesivamente.

Hay algunos métodos donde se usa la prueba de arriba hacia abajo, estos métodos están basados sólo en el hecho de que el sistema está diseñado de arriba hacia abajo y que es probado continuamente. Los módulos subyacentes consisten sólo de fragmentos que pueden regresar valores aleatorios y luego ser remplazados con el código real.

Es un fenómeno bien conocido que cuando se corrigen fallas detectadas se introducen nuevas fallas en el sistema. Cuando una falla ha sido corregida, se debe volver a probar el sistema desde el inicio, incluyendo los bloques que usan el bloque corregido. Es de esperarse que se introduzcan menos errores.

6.2 Tipos de Prueba

Hay muchos tipos de prueba, entre estos podemos mencionar:

- *Prueba de unidades*: significa que una y solo una unidad es probada como tal. Esta prueba requiere que la unidad sea independiente de otras. La unidad puede ser un procedimiento o una clase, pero también puede ser un módulo o grupo de módulos. En estas unidades se incluyen clases, bloques y paquetes de servicio.

- *Prueba de integración:* involucra pruebas con el propósito de verificar que las unidades están trabajando juntas correctamente. La integración y las pruebas de unidades puedan ser las mismas pruebas, el método de prueba es el que difiere. Se usan casos de uso para este tipo de prueba. Los casos de uso son una herramienta excelente para este tipo de prueba. Los bloques, paquetes de servicio, subsistemas y el sistema entero se prueba de esta manera.

- *Prueba de regresión*: es hecha cuando se han realizado cambios en el sistema, por ejemplo, al corregir una falla, el propósito de la prueba es verificar que continua el viejo funcionamiento. Esta prueba es una de las razones por las que se deben usar pruebas automatizadas. Cuando una unidad probada se cambia se necesitarán cambiar también las especificaciones de prueba, cualquier cambio del código puede llevar a que viejos casos de prueba no realicen lo que era su propósito original.

- *Prueba de operación* es la prueba más común a gran escala. Aquí el sistema es probado en operación normal por un gran tiempo, el sistema es usado en la manera pensada, sólo se reproducen los errores normales, los errores que el usuario normal puede esperar hacer, si

el sistema va ser reconfigurado durante la operación esto debe ser probado también. Este tipo de prueba sirve para ver la confiabilidad del sistema, y por lo tanto se pueden usar las medidas estáticas, como la medida de tiempo para falla (MTTF).

- *Prueba completa de escala*: es la continuación natural de una operación de prueba, la prueba significa que se corre el sistema a su máxima escala. Todos los parámetros del sistema se aproximan a sus valores límite, se conectan todos los tipos de equipo, muchos usuarios simultáneos están presentes en el sistema y muchos casos de uso están en operación al mismo tiempo. Esta prueba requiere mucho del sistema, pero estos requerimientos deben ser manejados por el sistema. Se requiere que una planta a escala completa este disponible y la prueba por lo tanto es muy cara, pero la experiencia muestra que aparecen nuevas fallas con esta prueba.

- *Prueba de rendimiento o prueba de capacidad*: tiene el propósito de medir la habilidad de procesamiento del sistema. La prueba deberá ser diseñada con la finalidad de medir el rendimiento con cargas diferentes. Lo que se quiera medir puede incluir almacenamiento, utilización del CPU o quizá solo la velocidad en un caso de uso específico. Los valores medidos deberán ser comparados con los valores requeridos.

- *Prueba de stress*: se prueban los límites extremos del sistema. Una prueba de sobrecarga es un tipo especial de prueba de stress y está relacionada también con la prueba de rendimiento. Su propósito es ver como se comporta el sistema cuando es sujetado a sobrecarga y consecuentemente se va un paso adelante con respecto a las pruebas completas de escala. No se puede esperar que el sistema maneje el procesamiento de estas pruebas,

pero debe ejecutar bien y no se debe detener. El sistema debe sobrevivir a picos de carga ocasionales.

- *Prueba negativa*: es un tipo de prueba de stress pensada para sujetar al sistema a esfuerzos más alla de lo que ha sido diseñado. Si hay barreras especiales deben ser probadas y verificadas y los casos de uso que normalmente no serán invocados simultáneamente, deberán ser ejecutados al mismo tiempo y así el sistema deberá intencional y sistemáticamente ser usado en una manera incorrecta. Este mal tratamiento debe ser planeado cuidadosamente para que así lugares especialmente críticos sean probados.

- *Pruebas basadas en especificaciones de requerimientos*: son los que pueden ser trazados directamente de la especificación de requerimientos. Pueden ser una de la primeras pruebas, por ejemplo, una prueba de rendimiento o una prueba completa de escala pueden ser un requerimiento particular que no se quiere revisar explícitamente.

- *Pruebas ergonómicas*: se están convirtiendo importantes a medida que los sistemas computacionales maduran. Si el sistema tiene una interface hombre-máquina entonces se deben probar los aspectos ergonómicos. Se deben hacer las siguientes preguntas: ¿Es la interface consistente en un caso de uso?, ¿Es la interface consistente entre algunas interfaces? ¿Son leíbles los menús?, ¿Son visibles los mensajes del sistema?, ¿Se pueden entender los mensajes de fallas?, ¿El sistema da un cuadro lógico?

- *Prueba de la documentación del usuario:* es un tipo de prueba ergonómica en la que se prueba la documentación del sistema. Tanto el manual del usuario, la documentación para

mantenimiento y servicio deben ser probadas. Raramente los manuales y el comportamiento del sistema no son consistentes. A los documentos se les deberá checar el lenguaje, el balance entre los capítulos y el balance entre el texto y dibujos deberá fijarse.

- *Prueba de aceptación*; es ejecutada normalmente por la organización ordenando el sistema y es el chequeo final del cliente consumidor. El sistema es probado con datos reales. Es comúnmente también la validación del sistema. Este chequeo es hecho contra la especificación de requerimientos original. Algunas veces la especificación de requerimientos es deficiente y consecuentemente este chequeo no debería hacerse.

Este tipo de prueba es comúnmente llamada prueba alfa. "Puede ser hecha por un largo tiempo cuando el sistema está trabajando en el ambiente para el cual ha sido desarrollado. Cuando la prueba ha sido hecha, la decisión es hecha como si el producto fuera aceptado o no" (Jacobson, [1992]). Si no hay ordenador específico, por ejemplo, un producto compilado, se usa frecuentemente la prueba beta. Esto significa que el producto es probado por clientes especialmente seleccionados o clientes potenciales que usan el sistema y reportan las fallas que detectan. La prueba beta es hecha antes de que el producto sea puesto en circulación en el mercado y es una forma de preliberación.

Por supuesto hay algunas variantes para estos tipos de pruebas. La inspección de código es un método que puede usarse. Su propósito es que alguien inspeccione el desarrollo del código y evalúe su calidad. Es un método muy caro, pero comúnmente produce un código de calidad.

Es importante planear la prueba. Cada prueba a ser hecha, con la excepción de las unidades de prueba de más bajo nivel, deben estar documentadas. Las condiciones deben ser especificadas, por ejemplo si la prueba debe ser hecha en un ambiente de desarrollo específico, se debe determinar como va a ser realizada, los resultados esperados, los resultados actuales, etc.

6.3 Técnicas de Prueba.

La meta debe ser automatizar tanto como sea posible la prueba. Esto puede ser hecho a través de programas de prueba especiales con los datos de prueba asociados. Es simple repetir pruebas, esto es, realizar pruebas de regresión en las nuevas versiones o cuando alguna parte ha sido corregida desde la prueba precedente.

El programa de prueba busca secuencias y datos de la entrada de datos. Entonces la unidad de sistema es alimentada con la secuencia y la prueba de la respuesta del sistema es observada por el programa de prueba. Esta salida puede ser almacenada directamente en un archivo de salida o ser comparada con alguna salida esperada, por lo que se puede hacer un análisis de falla (primitiva) automáticamente. Los datos de prueba son secuencias de estímulos que son enviados.

El programa de prueba y los datos son puestos en archivos separados y pueden de esta manera ser reusados con algunas combinaciones diferentes. El programa de prueba esta sujeto usualmente al ambiente donde se ejecuta, pero esto no se aplica a la prueba de datos. Es fácil modificar una prueba, pues solamente se cambian los archivos de datos. La meta es tener el

programa de prueba tan general como sea posible e independiente de la prueba de datos. Se debe ser capaz de reusarla para algunas pruebas diferentes y si se puede reusarla para algunos productos diferentes es aún mejor.

Comúnmente los sistemas orientados a objetos están altamente integrados y algunos objetos son dependientes de otros, por lo tanto, puede ser necesario desarrollar simuladores de objetos que simulen el comportamiento de objetos adyacentes.

Para obtener una interface simple para el sistema y hacer la prueba del programa independiente del sistema, normalmente se construye un simulador de interface o manejador de prueba. Este simulador regularmente ahorra trabajo, ya que el manejo de la interface del sistema no tiene que estar descrito en cada prueba del programa.

Para escribir buenos programas de prueba se pueden construir un conjunto de pruebas reusables, tanto para programas de prueba como para prueba de datos, de la misma manera que se hace para los componentes. Estos programas de prueba, sin embargo, deben ser desarrollados cuando el sistema esta listo para probarse. Una vez más se ve que la prueba debe ser parte del proceso regular de desarrollo.

Si el sistema va a trabajar en áreas críticas como aplicaciones militares o bancarias, la inclusión de programas de prueba en el producto debe ser evitada ya que puede ser fácil evadir las rutinas de seguridad a través de un bloque de prueba. Otra desventaja de esto es que el bloque de prueba puede dañar el rendimiento del sistema o crear otros problemas. Sin

embargo, esto se puede resolver al tener un módulo separado que es cargado si se requiere y tiene un lugar especial en la memoria.

6.4 Unidad de Prueba

Una unidad de prueba es la forma más primitiva de prueba y la hace normalmente el desarrollador, en las unidades de prueba se incluyen la prueba de clases, bloques y paquetes de servicio. En un sistema tradicional una unidad de prueba comúnmente es una prueba de procedimientos. Por lo tanto, las unidades de prueba en los sistemas orientados a objetos son hechos comúnmente en un nivel más alto.

Hacer una unidad de prueba de un código orientado a objetos es más complejo que la prueba ordinaria del código (procedural). Este es un resultado del aproximamiento orientado a objetos, el programa tiene un estructura plana que hace que el flujo del programa y su estado sea distribuido. Es difícil para el desarrollador que es dependiente de los objetos de otros diseñadores probar lo suyo.

Los requerimientos para herramientas de depuración son también más grandes para los sistemas orientados a objetos. Normalmente los ambientes contienen soporte para inspeccionar la estructura del objeto durante la ejecución y otros servicios de paquete. Esto es (usualmente) un estándar en ambientes para Smalltalk, C++, Simula y Eiffel (Jacobson, [1992]).

Cuando se prueba una unidad hay dos métodos generalmente. La prueba estructural (o prueba de caja blanca) significa que uno usa el conocimiento de como esta diseñada la unidad internamente para la prueba. La prueba de especificación (o prueba de caja negra) significa lo opuesto, se prueba sin ningún conocimiento de como luce internamente la unidad y la única cosa que se tiene es una especificación de la unidad. Normalmente se necesitan las dos ya que se complementan una a la otra. Comúnmente se empieza con la prueba de cada caja blanca y cuando se ha completado ésta se continua con la de prueba de caja negra.

6.4.1 Prueba Estructural.

El propósito de la prueba estructural es probar que la estructura interna está correcta. Esto significa que uno usa su conocimiento de cómo esta implementada la unidad cuando se prueba. Sería deseable cubrir todas las combinaciones posibles de parámetros, valores de variables y rutas en el código durante la prueba pero esto es casi siempre imposible ya que se tendría un número enorme de casos de prueba. La prueba estructural es algunas veces llamada prueba basada en el programa o prueba de caja blanca.

Para examinar la efectividad de los casos de prueba se pueden usar medidas de cobertura. La menor cobertura es ejercitar la ruta de cada decisión a cada decisión al menos una vez. Una decisión es típicamente una proposición. *IF*. Esta cobertura lleva a la situación de que todas las proposiciones son ejecutadas y por lo tanto también cada salida de la decisión ha sido probada. El requerimiento mínimo debería ser que todas las proposiciones han sido ejecutadas. Normalmente este es un objetivo razonable para la cobertura de la prueba.

El polimorfismo ayuda a aislar las pruebas necesitadas cuando se introducen cambios. Es una herramienta muy fuerte que comúnmente hace más fácil de incorporar los cambios. Si se agrega una clase descendiente y se prueba, no se necesitará hacerlo con las clases clientes de nuevo.

Otro beneficio específico de la prueba de la Ingeniería de Software Orientada a Objetos (OOSE) es el mecanismo de herencia. Esto comúnmente lleva a menos código, como las operaciones definidas en los ancestros también se usan en los descendientes, no obstante la cantidad de pruebas de tal código no es necesariamente menor. Para probar clases abstractas hay que enfocarse en dos propiedades, primeramente se debe probar que se puede heredar a la clase y que se pueden crear instancias de los descendientes. En segundo lugar se debe probar que cualquier estímulo enviado al objeto por si mismo trabaje apropiadamente.

Cuando una clase hereda a otra clase, la operación heredada puede necesitar ser probada otra vez en el nuevo contexto. Se tienen al menos dos razones para que una operación heredada no funcione en un descendiente.

- Si la clase descendiente modifica las variables de instancia para la que la operación heredada asume ciertos valores.
- Si las operaciones en el ancestro invocan operaciones implantadas en el descendiente.

El primer caso puede evitarse solo con usar la clase ancestro de tal manera que sea siempre consistente, especialmente al no cambiar las variables de instancia en una manera inapropiada. El segundo caso puede ser evitado al probar la operación en los descendientes que son invocados para asegurarse de que se cumplen realmente las especificaciones, Si el nuevo

descendiente no interactua de ninguna manera con las variables de instancia o las operaciones heredadas, no se necesitan volver a probar estos en el descendiente.

La herencia puede llevar a una prueba más extensiva. No solo cuando se modifica una clase ancestro se deben volver a probar los descendientes, también cuando se agrega un nuevo descendiente se puede necesitar volver a probar las operaciones heredadas. En el peor caso, podemos necesitar desarrollar casos de pruebas únicas para cada nivel en la jerarquía de herencia.

6.4.2 Prueba de la Especificación

La prueba de la especificación o de caja negra, tienen el propósito de verificar las relaciones de entrada, salida de una unidad, la meta es verificar el comportamiento especificado de la unidad, esto es, lo que hace la unidad, pero no se está interesado en como soluciona la unidad esto. Se envían estímulos con diferentes parámetros a la unidad y como salida se reciben nuevos estímulos o quizá se ve algún cambio en alguna variable. Es esencial probar la relación de entrada-salida en diferentes estados de la unidad.

Para el código orientado a objetos se debe tomar en consideración el estado del objeto a ser probado. Las operaciones de un objeto no son independientes y no se deben probar separadamente. También hay que ver como se usan los objetos en su ciclo de vida. Esto es especialmente importante para controlar el estado de los objetos.

La partición de equivalencia es una técnica para reducir el número de pruebas que se necesitan realizar. El objetivo es seleccionar un número razonablemente de casos de prueba de un gran número de posibles, para que la probabilidad de encontrar fallas sea alta. Una clase de equivalencia es por lo tanto un conjunto de valores para los que un objeto se supone se comporta similarmente.

La técnica de partición de equivalencia debe usarse en la prueba de la especificación. Para cada operación se identifican las clases de equivalencia de los parámetros y el estado del objeto, luego se seleccionan los datos típicos para cada una de las clases de equivalencia. Lo más importante es que la unidad trabaje con valores de entrada normales.

Una herramienta muy buena para este tipo de prueba es una matriz de estados. La matriz indica todos los estados que pueden ser adoptados por la unidad y todos los estímulos que la unidad espera recibir en los varios estados.

Un error común cuando se prueba código es que no se verifican los datos de salida, pues el recibir datos de salida no es suficiente, uno se debe asegurar de que estos están correctos.

6.5 Prueba de Bloques y Paquetes de Servicio.

Las pruebas de paquetes de servicio y de bloques, significan pruebas de unidades y pruebas de integración.

Para probar paquetes de servicio y grandes bloques se debe especificar la prueba en una prueba de especificación. La especificación indica lo que se probará y como se hará.

Las pruebas de bloques se hacen al usar los principios anteriores y sin escribir ninguna especificación de prueba. La regla que usualmente se aplica es que el nivel de bloque más grande puede ser conectado con un solo programador (normalmente un paquete de servicio) debe ser probado con documentación formal. El programador normalmente hace esta prueba solo y se hace usualmente en el ambiente de desarrollo.

6.6 Prueba de Integración.

El propósito de la prueba de integración es probar si unidades diferentes que han sido desarrolladas están trabajando juntas apropiadamente. En la Ingeniería de Software Orientada a Objetos la prueba de integración viene suavemente y es introducida temprano en el desarrollo (Jacobson, [1992]).

En la prueba de integración se incluyen la prueba de los casos de uso, los subsistemas y el sistema entero. Las pruebas de los paquetes de servicio y bloques pueden ser introducidos en una cierta extensión ya que también integran unidades. No hay consecuentemente prueba de integración en un desarrollo, en cambio la prueba es realizada algunas veces en diferentes niveles. Normalmente la prueba de integración es hecha por un equipo de prueba especial en el proyecto. Aquí la documentación es frecuentemente más formal que las pruebas de unidades.

Antes de empezar la prueba de integración los bloques incluidos deben estar completamente diseñados probados y aprobados. Esto no significa que se tenga que esperar la prueba de integración hasta que el sistema entero ha sido diseñado. Al contrario se puede empezar la prueba de integración cuando ya se tienen pocos bloques, se puede, por ejemplo probar casos de uso cuando esos bloques que participarán están diseñados completamente y aprobados.

La prueba de integración es una actividad que se puede describir como se muestra en la Fig. 6.1. Se empieza el trabajo al planear la prueba. Este plan es la base para identificar lo que deberá ser probado y especificado en más detalle. Esta especificación es la base del rendimiento actual.

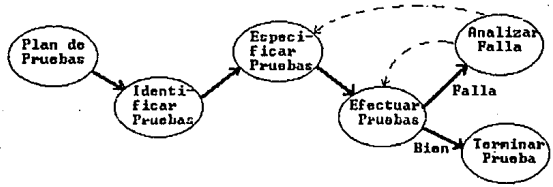


Fig. 6.1. Actividades en la prueba de integración

6.7 Planeación de la Prueba.

La planeación puede empezar cuando inicia el desarrollo. Se puede trabajar en esto en general durante la fase de análisis, pero no se puede empezar a preparar seriamente la prueba hasta que se empiece la construcción.

Las directrices de la prueba se establecen pronto. Al determinar el método y el nivel de combinación se crea la fundación de la prueba. Se debe determinar si la prueba será hecha automáticamente o manualmente y se puede hacer también una estimación temprana de los recursos que se requerirán.

Cuando el modelo de requerimientos está listo se puede empezar la planeación. Al mirar que partes estarán listas primero se puede decir también que pruebas serán hechas primero. Nunca hay que empezar la prueba de integración hasta que la unidad de prueba este lista, las desviaciones serían cruciales.

La planeación de la prueba debe considerar también los estándares para la prueba y los recursos requeridos para cada subprueba. El plan no debe controlar la prueba en detalle, pero debe funcionar como una base para las actividades de prueba (Jacobson, [1992]).

Se debe llevar una bitácora durante toda la prueba, que debe estar conectada a una versión del sistema. El propósito de esta es dar una historia breve de todas las actividades de la prueba, tanto los éxitos como los fracasos.

6.8 Identificación de la Prueba.

La identificación es resolver lo que será probado. La primera vez algunas clases, bloques de paquetes de servicio y subsistemas son llevados juntos, por lo tanto la prueba se debe concentrar en esto. Cada caso de uso se prueba separado inicialmente. Los casos de uso constituyen una herramienta excelente para la prueba de integración ya que se interconectan explícitamente algunas clases y bloques. Cuando todos los casos de uso han sido probados (en varios niveles) el sistema está probado enteramente.

Para un caso de uso podemos tener las siguientes pruebas:

1) Pruebas de curso básico
2) Pruebas de curso particulares
3) Pruebas basadas en la especificación de requerimientos
4) Prueba de la documentación del usuario

Las pruebas del sistema se pueden dividir de la siguiente forma:

1) Pruebas de operación
2) Pruebas a escala completa
3) Pruebas negativas
4) Pruebas basadas en la especificación de requerimientos
5) Prueba de la documentación del usuario.

La prueba de operación es una ejecución del sistema entero, la prueba a escala completa aumenta los parámetros del sistema a sus límites específicos y la prueba negativa tiene el propósito de romper el sistema entero al romperlo a través de estos límites.

La prueba de rendimiento que puede ser de un interés extra para los sistemas orientados a objetos, es puesta entre las pruebas basadas en los requerimientos, si se mencionan estos, de otra manera es puesta bajo las pruebas de operación. Cada tipo de prueba se divide en subpruebas con diferentes condiciones.

RESUMEN

- El propósito de probar es encontrar fallas.

- La prueba de software es generalmente una gran parte del desarrollo y debe ser planeada de la misma manera que el análisis y la construcción de programas.

- Entre los tipos de prueba podemos mencionar:
  - Prueba de unidades
  - Prueba de integración
  - Prueba de regresión
  - Prueba de operación
  - Prueba completa de escala
  - Prueba de rendimiento o capacidad
  - Prueba de stress

- Prueba negativa
- Pruebas basadas en requerimientos
- Pruebas ergonómicas
- Prueba de aceptación

Es de suma importancia que la prueba o series de pruebas que se deban realizar estén documentadas, dicha documentación nos servirá de referencia para saber como fue hecha, los resultados que se esperaban, y los que arrojo.

REUSO

Introducción

Ser capaz de reusar código ha sido desde hace tiempo una noción en el mundo de la ingeniería del software. Este ha sido uno de los objetivos principales para aumentar dramáticamente la productividad de la programación, sin embargo, aunque la mayor parte de la gente está de acuerdo sobre la importancia del reuso, ha sido practicado muy poco en el desarrollo de software de las organizaciones.

7.1 Ingeniería de Software Reusable

Cuando se habla de reuso en la ingeniería de software esto se refiere a cualquier cosa que pueda ser reusado en cualquier momento (Jacobson, [1992]).

Esto incluye toda la información y el conocimiento que ha sido desarrollado, además de la experiencia de proyectos anteriores, arquitecturas de sistemas que han probado ser buenos, métodos de desarrollo que han sido útiles, código de programa escrito y algoritmos completos.

7.2 Componentes como Mecanismo de Reforzamiento

Cuando se trabaja con componentes de software, se deben ver estos como parte del lenguaje de programación, los componentes constituyen un mecanismo de reforzamiento para

el lenguaje de implementación, en lugar de trabajar con construcciones primitivas, se trabajará con construcciones de niveles de abstracción más altos como listas y ventanas. Se pueden tener niveles de desarrollo de construcciones primitivas o más complejos. Teniendo los componentes construidos en el lenguaje de programación se obtendrá un nivel más alto de la capa de aplicación en la que se desarrollarán las aplicaciones.

Construir con componentes y con objetos son dos actividades totalmente diferentes. Se empieza con los objetos de los casos de uso y se diseñan los objetos conforme el sistema se va desarrollando. Construir con componentes es una actividad de abajo a arriba. Posteriormente los bloques manejarán unidades manipuladas por los desarrolladores de la aplicación y algunas aplicaciones quizá se desarrollen en una organización. Los componentes están documentados para este propósito, manejarán unidades de un sistema componente, están compartidos entre algunas aplicaciones.

Trabajar con componentes no es único para la ingeniería de software. El mismo modo de trabajo es usado en todas las otras disciplinas de la ingeniería.

Muchos sienten que los componentes son una de las soluciones más importantes para la crisis del software. La crisis consiste en que cuando la complejidad de los sistemas crece, se vuelven más caros y más difíciles de desarrollar. El problema es que la productividad de los ingenieros de software no crece al mismo ritmo. Si uno no se vuelve más productivo esos grandes sistemas no serán desarrollados.

Se espera que los componentes dirijan el diseño de los grandes sistemas. "Hay dos criterios importantes para los que los componentes son de gran ayuda: reducir el tiempo de desarrollo (costo) y elevar la calidad" (Jacobson, [1992]). Se reduce el tiempo de desarrollo conforme se tienen más componentes poderosos como base, esto reduce la complejidad y se consigue escribir menos código, al usar componentes que ya han sido usados en otras aplicaciones. Si se cumplen estos dos criterios esto hará que los sistemas de software sean menos caros y mejores.

¿Por qué no se tienen componentes hoy? Las causas son principalmente:

- Los proyectos son conducidos comúnmente con un presupuesto estrecho y con un itinerario muy apretado. Diseñar buenos componentes lleva tiempo. Como no es rentable diseñar componentes no hay incentivo para realizar este proyecto.
- Existe una actitud de desconfianza hacia lo que los demás escriben y se siente que no se puede tener control sobre lo que otro escribe por lo cual no se confía mucho en los componentes.
- No hay componentes estándar reconocidos que sean usados ampliamente, pues el establecer un estándar para los componentes es una tarea muy difícil.
- Los componentes ya existen, pero no se pueden encontrar. Se pueden diseñar componentes completos en un proyecto pero no son salvados.
- Se siente productividad cuando se escribe código fuente. Como la medida más común del software es la linea de código, se siente un progreso cuando se escribe código. Esto es un abuso de la tarea, pues se deben resolver los problemas difíciles, no solo los fáciles.

- Si un componente de mercado aparece habrá problemas con los componentes que son copiados y distribuidos gratis, lo que significará que nadie quiera pagar sus altos costos y las compañías no se interesen en diseñar componentes a gran escala.

Una razón importante por la cual no se han diseñado buenos componentes es que los ingenieros han tratado de encontrarlos de funciones y no de objetos.

El método de desarrollo para diseñar componentes útiles no ha sido estimulado. Un método de desarrollo tradicional tiene la forma de descomposición funcional o al menos un proceso de arriba a abajo. Esto no beneficia del todo el uso de componentes. Con los métodos orientados a objetos se construyen tanto de abajo a arriba como de arriba a abajo. Construir con componentes es consecuentemente más natural con un método orientado a objetos que con uno de función de datos.

Los componentes son usados hoy en día en ciertas áreas. Un ejemplo es el mundo de Smalltalk en el que mucho de la programación es encontrar la clase correcta para la aplicación. Por lo que se tiene la actitud de "una buena clase es una clase reusable" (Jacobson [1992]).

Los componentes de software también son usados en algunas industrias. Algunas compañías (entre ellas Toshiba, IBM y NobelTech) han creado departamentos especiales de componentes de software.

El propósito es darles a todos los otros departamentos componentes que no tienen requerimientos de rentabilidad directamente. Esto significa que el trabajar en diseñar un

componente bueno y útil no disminuirá el presupuesto de algún proyecto crítico, en vez de esto se distribuirá sobre diferentes proyectos.

7.3 Definición de un Componente

Un componente es una unidad de construcción estándar en una organización que es usado para desarrollar aplicaciones.

Para hacer un componente reusable en varias aplicaciones es necesario que sea independiente de la aplicación para la que fue diseñado. Esto no siempre es necesario cumplirlo, algunas veces los componentes que son dependientes de la aplicación son de interés para reuso. Entre más dependiente es el componente en una aplicación, comúnmente se adapta para poder usarlo.

Usar un componente significa que podemos ahorrar tiempo ya que no necesitamos saber como trabaja por dentro sólo necesitamos usarlo por fuera. Un componente complejo que es fácil de usar incrementa por lo tanto el nivel de abstracción para el desarrollador. Forma por lo tanto una simplificación conceptual de lo que se ha implementado.

Los componentes deben ser diseñados para ser reusados. Por lo que se necesitan incluir las operaciones para que todo el uso razonable del componente sea satisfecho; sin embargo, no se deben incluir muchas operaciones, ya que esto hace que el componente sea difícil de entender y usar. El componente debe capturar todas las operaciones significativas de uso para cualquier desarrollador y no otras operaciones.

"Los componentes deben ser de una calidad extraordinaria. Por eso necesitan probarse bien, ser eficientes y documentarse bien. Los componentes son por lo tanto normalmente más caros al desarrollar que el software ordinario" (Jacobson [1992]).

Los componentes también deben invitar al reuso. Los componentes por lo tanto deben tener una interface bien diseñada, de fácil recuperación y acompañados de buena documentación, correcta y fácil de usar. Los componentes por lo tanto deben estar empaquetados para el reuso.

"En general existen dos escuelas de pensamiento relativas a los componentes primitivos. Una es que los componentes deben ser flexibles, lo que significa que deben ser fáciles de cambiar para poder adaptarlos a las necesidades" (Parnas, [1983]). Los argumentos para esto son razones de eficiencia, los componentes generales son regularmente ineficientes.

"El otro punto de vista es opuesto, en este, los componentes ya deben estar especializados y no se les debe hacer cambios" (Booch, [1991]) Aquí se resuelve el problema de la eficiencia al tener diferentes tipos del mismo componente.

Se pueden tener dos tipos de componentes: un componente de caja blanca y uno de caja negra. En un componente de caja blanca se tiene que ir hacia dentro para reusarlo, estos regularmente tienen la tendencia a estar orientados hacia un dominio específico de aplicación. Para desarrollar estos componentes de caja blanca es necesario por lo tanto tener un buen conocimiento del dominio de la aplicación.

Los componentes de caja negra son más fáciles de usar que los de caja blanca, ya que son usados para ser conectados como componentes y no necesitan modificación, además deben estar bien probados, son comúnmente de alta calidad en comparación con el software ordinario ya que los requerimientos son más fuertes que los de los típicos componentes de caja negra que son los tipos de datos abstractos como las pilas, los strings y los árboles de búsqueda.

Una forma especial de componente de caja blanca es un diseño de un esquema donde se tiene un esqueleto entero para construir dentro la aplicación. Este es un diseño reusable, los componentes típicos de caja blanca incluyen un algoritmo de ordenamiento, donde se puede cambiar el operador "mayor-que" y el Modelo/vista/controlador en Smalltalk que es una clase esquema (una de las primeras) para construcción de interfaces.

7.4 Uso de Componentes

Los componentes deben ser vistos como una parte importante del lenguaje de implantación usado. Deben dar un nivel para incrementar la productividad. Como el desarrollador debe tener un buen conocimiento y habilidades del lenguaje de programación, también debe tener un buen conocimiento de la librería de componentes para poder usarlos efectivamente.

Los modelos de análisis y diseño dan un esquema fuerte para encontrar esos lugares. En particular, objetos, asociaciones y los tipos de atributos deben revisarse. Si el componente existe en la librería se puede usar directamente, de otra manera se necesitan propuestas de nuevos componentes.

Estos son algunos lugares donde se pueden usar los componentes:

- Los objetos de entidad general que son usados para desarrollar otros objetos entidad, por ejemplo, si son heredados o accesados por otros, son comúnmente llamados componentes orientados a la aplicación.
- Los objetos de interface pueden ser implementados comúnmente usando componentes para sistemas de ventanas. Las ventanas, botones y las barras de scroll son componentes típicos. De la misma manera, una área de aplicación podría tener interfaces similares que podrían ser un esquema general para herramientas de interfaces gráficas de usuarios (GUIS), también se deberían usar cuando se desarrollan las interfaces del sistemas.
- Algunos objetos de control tendrán funciones generales como actividades de corte, colección de datos para estadísticas y funciones de ayuda en línea. Estas son modeladas comúnmente usando una relación extendida y son utilizadas en diferentes lugares y posiblemente también en diferentes aplicaciones. Podemos tener por ejemplo una función general de ayuda en línea que es un esquema reusable en varias aplicaciones.
- Una asociación relacionada es comúnmente implementada con una referencia, si la cardinalidad tiene un intervalo fijado se usa una estructura estática como un arreglo o una colección para mantener estas referencias.
- Las clases de diferentes tipos ocurren en varias fases, por ejemplo, tipos de atributos, de parámetros y locales cuando se implementan bloques. Algunos de estos tipos quizá sean generales y por lo tanto podrían ser implementados con componentes.
- Durante la construcción, el sistema debe ser robusto para cambios en el ambiente de implementación. Esto se logra al encapsular el ambiente. El encapsulamiento se puede hacer al usar componentes.

Para poder usar componentes, es esencial que desde el comienzo del desarrollo se conozca con que componentes se cuenta. El mejor caso es cuando los componentes están disponibles desde el inicio. Entonces el desarrollador los puede ver como una herramienta natural durante todo el trabajo de desarrollo.

En algunas grandes organizaciones o grandes proyectos es algunas veces apropiado empezar con un desarrollo de aplicación orientado a componentes de caja blanca y esquemas. Luego se necesita un buen conocimiento del dominio de la aplicación para desarrollar componentes que serán usados generalmente por diferentes aplicaciones. El manejo de la versión, el manejo de la configuración y la entrega de tales sistemas a las organizaciones de desarrollo debe ser manejado con cuidado.

7.5 Implementación con componentes.

Los componentes son usados para construir bloques. El uso de componentes no está documentado de la misma manera como con el diseño de objetos. Cuando se usan componentes en el modelo de diseño se pueden usar varias técnicas para documentar su uso. El uso de los componentes más primitivos normalmente no se documenta en el modelo de diseño, estos son usados más como primitivas del lenguaje de programación. Los componentes son introducidos normalmente como un caso especial de módulos de objetos, especialmente en el nivel más primitivo en el diseño, sin embargo, para componentes más complejos es significativo incluirlos en el modelo de diseño. Los esquemas y otros componentes que involucran decisiones de diseño importantes deben incluirse en el modelo del diseño.

Los bloques pueden también ser reusados, pero son reusados para configurar el sistema para diferentes clientes.

Comúnmente se dice que la herencia es fundamental para el reuso ya que se pueden heredar componentes y por lo tanto reusarlos. Aunque la herencia es una herramienta fuerte en muchos contextos, esto es una mala concepción. La herencia no es un requisito indispensables para el reuso. Es posible reusar sin herencia. Actualmente la herencia no ha sido usada frecuentemente para decisiones de reuso.

Una necesidad común cuando se usan componentes es que se deben adaptar o especializar a la aplicación en cuestión, esto no se debe hacer normalmente al realizar cambios al componente, en lugar de esto hay que encapsular ya sea al componente en otro objeto que actúa como una interface entre la aplicación y el componente, o hacer una especialización del componente.

Otro problema al usar componentes es que están escritos en un lenguaje o ambiente diferente al del desarrollo. Esto será probablemente de menor importancia en el futuro cuando sea más fácil evitar estos problemas.

El uso de componentes lleva a un mejor código. Esto es un resultado del uso frecuente y del hecho de que han sido probados totalmente, pero también es un resultado del hecho de que se ha puesto más trabajo al diseñar el componente de lo que normalmente se hace con el código de programas ordinarios.

Se debe fomentar el uso de componentes y cuestionar cualquier diseño o implementación que no los use. Es obvio que al juzgar la productividad de un desarrollador en términos del número de líneas de código es opuesta a esta idea. Entre más se reusa menos se tiene que escribir. Por lo tanto una mejor medida de la productividad es el número de componentes usado en el diseño o implementación. Esto también aumentará la calidad del sistema.

7.6 Manejo de Componentes

Un sistema componente no es normalmente notable sólo para un proyecto. Tales sistemas deben ser compartidos entre algunos proyectos. La razón es que el desarrollo de componentes es comúnmente más caro que el del software ordinario y esto lo hace no rentable en términos del horizonte de un proyecto. Los beneficios reales vienen cuando un componente puede ser usado en varios proyectos y productos. Por lo tanto el manejo de componentes debe estar basado en proyectos múltiples.

7.7 Construcción de Componentes

La construcción de componentes es fundamental para su reuso. El reuso no viene como un efecto lateral, la especificación, la construcción y la prueba deben ser hechas para reusarse. Esto hace que un componente sea más caro (hasta 10 veces) de desarrollar que otro software (Jacobson, [1992]).

Algunos criterios para buenos componentes son:

- La entendibilidad. El componente deberá representar una abstracción. Deberá tener alta cohesión y ofrecer sólo las operaciones necesarias para hacerlo útil de una manera eficiente. Deberá tener una interface bien definida, tanto sintácticamente como semánticamente. Si dos operaciones en dos componentes diferentes tienen el mismo nombre, deberán actuar de manera similar para facilitar el entendimiento.
- El componente debe ser independiente de entidades cercanas. Debe estar conectado débilmente y por lo tanto el acoplamiento debe ser bajo con otras unidades. Una filosofía orientada a objetos lleva a esta independencia.
- El componente debe ser una abstracción general que es útil en algunas aplicaciones sin tener que sobrellevar cambios. Es estandarizado con respecto al nombre, con manejo de errores, estructura, etc..
- La entendibilidad no debe ser sólo externa, también debe ser interna. Como los buenos componentes tendrán una larga vida, serán mantenidos por largo tiempo.
- El criterio más importante para el reuso consiste en la interface del componente. Debe ser lo suficientemente completa y general para reusar fácilmente el componente.
- Se puede usar la herencia en el diseño del componente. El mecanismo de herencia es muy útil para construir una librería de componentes poderosa. Sin embargo, el uso de la herencia es también fundamental para la calidad del sistema componente.
- El rendimiento de tiempo y/o memoria de los componentes es también un resultado crítico.

Como se dijo, una implementación general que lleva a un componente ineficiente debe evitarse ya que probablemente no será usado.

Algunos criterios para diseños de buenos componentes y reusables han sido propuestos, por Johnson y Foote (1988), Meyer (1990) y Lieberherr y Holland (1989).

Los criterios incluyen:

- Reducir el número de parámetros. Menos argumentos llevarán a operaciones más cohesivas que implicarán más operaciones primitivas.
- Evitar usar opciones en los parámetros. Cualquier opción debe ser puesta en la creación del procedimiento y las operaciones especiales deben ser usadas para cambiar estas opciones.
- No accesar directamente a las variables de instancia. Para accesar a las variables de instancia se deben usar operaciones especiales, esto libera la implementación de la interface actual.
- El nombre debe ser consistente. Las operaciones en diferentes componentes debe tener nombres similares si realizan la misma tarea.

7.8 El Sistema de Componentes

Una librería de componentes es algo que todas las organizaciones maduras de desarrollo deberían tener. Esta actividad incluye seleccionar, clasificar y manejar los componentes incluidos y también el desarrollo de nuevos. Alguien se debería de responsabilizar de asegurarse de que la información acerca de la librería de componentes está disponible, extendida totalmente para toda la organización de desarrollo y que los componente son accesibles.

Una librería de componentes debe ser compartida preferiblemente entre diferentes productos. Por lo que el sistema componente debe servir varios proyectos. Los propósitos para nuevos componentes normalmente vienen de los proyectos que el sistema de componentes soporta. Es importante no ser pasivo en la búsqueda de nuevos componentes y no sólo esperar propuestas, también hay que buscar activamente nuevos componentes que se puedan ajustar.

Un componente tendrá un ciclo de vida como el que se muestra en la figura.

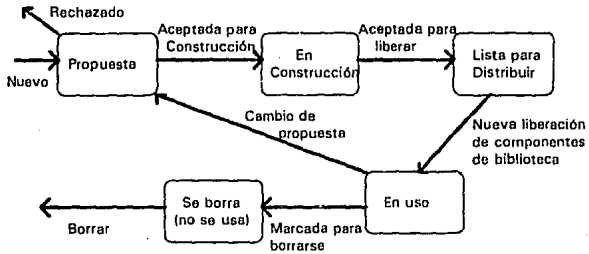


Figura 7.1. El ciclo de vida de un componente

Los componentes básicos como las estructuras de datos, las facilidades gráficas, deben existir en la librería. Estos son comprados normalmente de un vendedor externo o acompañar el ambiente de desarrollo usado.

El factor más esencial es que los componentes deben ser rentables para la organización. Aquí los criterios importantes son el reuso potencial del componente y su tamaño. Entre más lugares lo usen, será más rentable (Jacobson, [1992]).

Una propuesta de componente consiste de la funcionalidad deseada (interface y características), su reuso potencial (como y donde será reusado) y una fecha de entrega, la propuesta puede concernir al desarrollo de un nuevo componente, comprar un componente o una propuesta de cambio para componente.

Una propuesta de cambio puede ser de tres tipos. Un reporte de error siempre es aceptado. Un cambio de la implementación es aceptado si es razonable, como una optimización. Un cambio de la interface es aceptado sólo si hay tres muy buenas razones, ya que afecta a todos los dependientes del componente a los desarrolladores de la aplicación y a otros componentes.

Para identificar componentes reusables se pueden usar los siguiente conceptos.

- *Tamaño.* Esto afecta tanto al costo del reuso como la calidad. Si es demasiado pequeño los beneficios no excederán el costo de manejarlo. Si es demasiado grande, es difícil tener alta calidad.
- *Complejidad.* Esta afecta también al costo del reuso y a la calidad. Un componente demasiado trivial no es rentable para reusar y con un componente demasiado complejo es difícil tener alta calidad.
- *Frecuencia de reuso.* El número de lugares donde un componente es usado es demasiado importante. Ya que el costo del componente se destribuye entre varios proyectos.

7.9 Documentación de Componentes

Un componente es documentado para uso de una hoja de datos de componentes. Esta hoja tiene el mismo propósito que una hoja de datos para componentes de hardware, describe enteramente lo que el usuario necesita saber para poder usar el componente. Para encontrar el componente correcto se usa un estudio de componentes (automatizado posiblemente). Este documento presenta la librería de componentes entera y el desarrollador busca en este documento para encontrar el componente correcto para usar. Este documento deber ser generado en base a información característica relacionada con cada componente. Para propósitos de mantenimiento también hay una descripción del componente. Este documento describe la implementación del componente y es usado por la organización para mantener los componentes. Las relaciones se muestran en la figura.

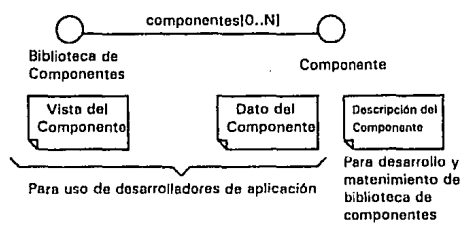


Figura 7.2. Documentación para un sistema de componentes

El elemento que diferencia al desarrollo de software industrial del de otra programación es la gran escala. Para obtener beneficios reales del reuso se necesita por lo tanto un gran número de componentes. Ya se han discutido los problemas de una biblioteca de

componentes grande. Otro problema es la tecnología para manejarlos, simplemente no existe. El problema de clasificación es una parte de tal tecnología. Esto se puede comparar con los componentes de hardware. Los componentes de software pueden ser estructurados siguiendo dos métodos. Se pueden usar criterios de eficiencia, velocidad y requerimientos de almacenamiento.

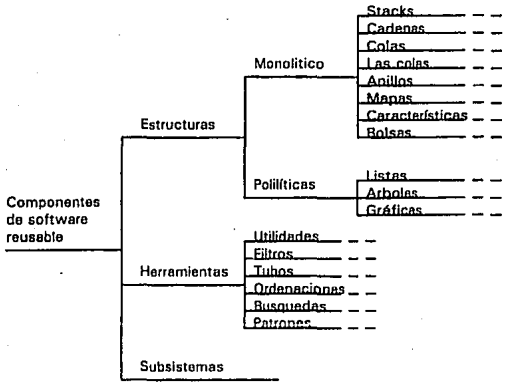


Figura 7.3. Clasificación de componentes

En los métodos jerárquicos los componentes se clasifican en una estructura de árbol donde se empieza la búsqueda desde la raíz y conforme se va bajando en el árbol se indica que propiedades debe tener el componente. Cuando se ha alcanzado una hoja nodo en el árbol, se ha encontrado un componente. Un ejemplo de método jerárquico es sugerido por Booch (1987). Su taxonomía empieza al clasificar primero los componentes en estructuras,

herramientas y subsistemas. Las estructuras son componentes que pueden ser descritos como tipos de datos abstractos o máquinas de estado abstracto. Las herramientas son componentes que describen una abstracción de algoritmo. Los subsistemas son componentes que son construidos de estructuras y herramientas (ver figura 7.3). Las estructuras se dividen en monolíticas y polilíticas. En el siguiente nivel se encuentran los tipos de datos abstractos.

El otro método de clasificación se basa en la búsqueda por clave. Es importante por sí mismo tener una terminología estandarizada que todos puedan usar. Ejemplos de métodos para nombrar componentes han sido propuestos por Prieto-Diaz y Freeman (1987) y por Cox (1986). El primero está basado en el hecho de que cada componente puede ser descrito en términos de su función, cómo es ejecutado y sus detalles de implementación. Cada componente puede ser descrito con seis tuplas (función, objeto, medio, tipo de sistema, aplicación, campo de aplicación). Una lista de palabras de sinónimos es una ayuda al usar palabras estandarizadas.

Un método jerárquico es bueno ya que es simple y lógico de usar y es relativamente fácil de introducir nuevos componentes si la estructura es buena. El problema es que no pueden describir muchas relaciones de muchos a muchos con dicho método, lo que significa que si se ha entrado al árbol incorrecto en la taxonomía nunca se encontrará el componente correcto. El método basado en claves tiene sus ventajas ya que es muy flexible cuando es controlado por lenguaje ordinario. Las desventajas son que se está forzando a cambiar o al menos extender, las claves de búsqueda tal como crece la librería. El método también requiere un vocabulario uniforme. Para remediar las deficiencias de ambos métodos, se han sugerido nuevos métodos en los que las propiedades de los dos anteriores han sido mezclados.

RESUMEN

- El reuso en la ingeniería de software denota que cualquier cosa puede ser reusada en cualquier momento; incluyéndose: información, conocimiento, experiencia, métodos, procedimientos, arquitecturas, código y clases de componentes.

- Hay dos criterios importantes para tomar en cuenta el uso de componentes reusables: reducir el tiempo de desarrollo (costo) y elevar la calidad.

- Un componente bueno y útil no disminuirá el presupuesto de algún proyecto crítico, en lugar de esto se distribuirá sobre diferentes proyectos.

- Un componente de software es una unidad estándar en una organización que es usada para desarrollar diferentes aplicaciones.

- Los componentes deben ser de una extraordinaria calidad, por lo que necesitan ser eficientes, bien probados, y bien documentados. Los componentes son por lo tanto normalmente más caros de desarrollar que el software ordinario.

- Algunos criterios para hacer buenos componentes son:
  1. Entendibilidad, ya que el componente representa una abstracción, debe tener una interface bien definida, alta cohesión y ofrecer sólo las operaciones necesarias.
  2. El componente debe ser independiente de entidades cercanas, con poco acoplamiento a otras unidades.

- Los componentes deben ser documentados con una hoja de datos para componentes, la cual debe describir enteramente lo que el usuario necesita saber para poder usar el componente, dicha información se puede automatizar para usarlo como una ayuda en línea.

TDRAWING SISTEMA PROTOTIPO

El TDrawing fue desarrollado bajo SMALLTALK /V de Digitalk, versión para Windows, para su desarrollo se uso WindowBuilder que es un constructor de interfaces gráficas integrado a Smalltalk. WindowBuilder permite de una manera muy sencilla la construcción de interfaces de muchos tipos, manejando diversos tipos de ventanas y permite manejar interfaces de un alto grado de dificultad. En el desarrollo de TDrawing se usó gran parte de las facilidades gráficas que permite Smalltalk, usando principalmente las clases: Window, ViewManager, FreeDrawing, ScrollBar, GraphicsMedium, así como otras, debido a los mecanismos de la Programación Orientada a Objetos, como son la herencia y el polimorfismo, el usar código de otras clases puede hacerse de una manera muy sencilla y obteniendo muchas ventajas de esto.

El TDrawing necesita para correr los siguientes requerimientos: 4Mb en RAM, procesador 386 o superior, 4Mb de espacio en disco duro o más y contar con mouse.

El nombre del programa para correr TDrawing es V.EXE. El TDrawing esta inicialmente en modo de dibujo con el cursor con una flecha, tiene el color negro y el ancho del punto que se dibuja es de un pixel, que es el mínimo. Para dibujar hay que mover el mouse y mantener apretado el botón izquierdo del mouse.

El TDrawing tiene cinco menús:

1) El menú de *archivo* que permite cargar archivos, abrir nuevos archivos, salvar archivos y salir del sistema.

2) El menú de *opciones* que da diferentes facilidades de edición, como limpiar, copiar, pegar, el tamaño de la pluma, el tipo de fuentes para el texto y un zoom (acercamiento).
3) El menú de *modo*, que permite usar las diferentes opciones para dibujar.
4) El menú de *color* que permite escoger un color diferente para la pluma, por default el color de inicio es el negro.
5) El menú de *ayuda* que da información acerca de TDrawing.

1) El menú de archivo tiene las siguientes opciones:

Nuevo: Crea una ventana en la que se puede empezar a dibujar, no tiene nombre y puede salvarse lo que se haga posteriormente.

Abrir: Hace la carga de un archivo con extensión BMP, a una ventana en la cual se despliega el archivo que ha sido abierto.

Guardar: Sirve para salvar un archivo con formato BMP, si el archivo no ha sido salvado anteriormente pide el nombre del archivo, por default da la extensión BMP.

Guardar como: Sirve para salvar un archivo con formato BMP, la ventana de dialogo puede el nombre del archivo que va a salvar.

Salir: Es la salida del sistema, si se han hecho cambios al área de trabajo en el momento de salir del sistema, pregunta si los cambios serán salvados o si se desea salir del sistema sin salvar nada.

2) Menú de opciones cuenta con las siguientes operaciones:

Limpiar: Se usa para poner el área de trabajo en blanco, es decir, si había algo se elimina.

Copiar: Hace la copia de una sección del área de trabajo, para esto se indican 2 puntos, que se dan por medio del mouse, al apretar el botón izquierdo del mouse se marca el primer

punto a continuación manteniéndolo apretado y luego soltando el botón izquierdo para seleccionar el segundo punto.

Pegar: Una vez que se ha copiado algo, esta opción sirve para ponerla en alguna parte del área de trabajo, solo se necesita, escoger la opción y apretar el botón izquierdo del mouse en el lugar donde se desee pegar lo que se copio.

Tamaño pluma: Es el tamaño de la pluma con el cual se trabajara, para escoger un tamaño de la pluma se necesita seleccionar esta opción y apretar el botón izquierdo del mouse, mover el mouse manteniendo apretado este botón y luego soltar este botón, pudiendo hacer esto en cualquier parte del área de trabajo.

Fuentes: Al seleccionar esta opción se puede cambiar el tipo de letra que se usa al escribir texto, pudiendo cambiar tanto el tipo de letra como el tamaño de esta.

Zoom: Sirve para hacer un acercamiento del área de trabajo, para hacer esto, se debe seleccionar una parte del área de trabajo donde se desee ver este acercamiento, después de que se ha seleccionado esta área, aparecerá una ventana donde se vera la parte que se selecciono ampliada, en esta ventana se pueden hacer cambios, del área que se selecciono punto a punto, se pueden cambiar los colores que se usan, se puede rellenar una parte o todo lo que se selecciona, estos cambios se pueden salvar o no, ya que se cuenta con estas 2 opciones, para salir de esta ventana y regresar a la ventana original se debe de apretar el botón de salir y se regresara a la ventana con los cambios que se hayan hecho en el zoom, si estos fueron salvados. El área de trabajo del zoom aparece cuadriculada y es ahí donde se puede cambiar el dibujo; para hacer algún cambio hay que apretar el botón izquierdo del mouse y dibujar lo que se desee, las opciones que hay son el color de la pluma y el relleno, que se puede usar en una área cerrada que tenga un mismo color.

3) El menú de modo cuenta con las siguientes opciones:

Dibujar: Es el modo por default al iniciar la aplicación, esta opción dibuja mientras se mantenga apretado el botón izquierdo del mouse, dibujando con el color y el tamaño de la pluma que se haya escogido.

Linea: Hace una linea, esto se hace al escoger 2 puntos, el primer punto se tiene cuando se presiona el botón izquierdo del mouse, el segundo punto esta donde se suelta el botón izquierdo del mouse. Si se presiona solo una vez el mouse en el mismo lugar se dibuja solo un punto.

Curva: Realiza una curva, esto se hace al dibujar una linea, después se toma algún punto exterior a la linea y hacia este punto se dibuja la curva, la curvatura depende del punto que se escoja, al presionar el botón izquierdo del mouse y soltarlo en el punto deseado.

Rectángulo: Un rectángulo se dibuja al seleccionar 2 puntos, las 2 esquinas del rectángulo, la primera se escoge al apretar el botón izquierdo del mouse y la segunda se escoge al soltar el botón izquierdo del mouse en el lugar deseado. Si se aprieta y se suelta el botón izquierdo del mouse en el mismo lugar solo se dibuja un solo punto.

Circulo: Para el circulo se da un centro y un radio, el centro es el punto en el que se presiona el botón izquierdo del mouse por primera vez y el radio es al distancia que hay entre el centro y el punto en el que se soltó el botón izquierdo del mouse, en el lugar deseado.

Elipse: El centro de la elipse es el punto donde se presiona el botón izquierdo del mouse por primera vez y a donde se suelta el botón izquierdo del mouse en el lugar seleccionado, determina el tamaño y la forma de la elipse.

Texto: Permite introducir texto, sin embargo, esta aplicación no cuenta con características de edición de texto que permitan borrar por medio del teclado retrocediendo por

medio del backspace. Este texto puede tener diferentes colores y también diferentes tipos de letra de acuerdo a lo que se quiera.

Borra: Sirve para borrar lo que se seleccione, la selección se hace presionando el botón izquierdo del mouse y arrastrándolo a través de lo que se quiera borrar, el tamaño de la pluma para borrar es de 8 pixeles.

Descripción de la clase TDrawing

TDrawing: Es una subclase de ViewManager, en la cual se implementaron los métodos correspondientes a la aplicación; pudiendo obtener funcionalidad de las clases necesarias para lograr el funcionamiento para el manejo gráfico requerido. A continuación se describen brevemente los métodos de instancia implementados.

```
ViewManager subclass: #TDrawing
 instanceVariableNames:
 'pen state start previous penSize penColor pane modo fileName changeFlag fila wihe '
 classVariableNames: ''
 poolDictionaries:
 'ColorConstants WBConstants WinConstants ' !
```

TDrawing métodos de instancia:

```
activate: aPane
        "Activa el área de trabajo a usar"
```

backupRelative: aPoint

"Privado- Contesta un punto relativo a la ventana de respaldo"

bitEdit

"Pide al usuario un rectángulo y abre BitEditor en el bitmap asociado con el rectángulo."

borra

"Llama al procedimiento erase:"

circle: aPoint

"Se encarga de dibujar el circulo"

circulo: aPane

"Llama al método circle:"

clearMouseCapture

"Termina la captura del mouse"

closed

"Privado - Pregunta al usuario si desea salvar el trabajo"

colorSelected: aColor

"Se encarga de seleccionar el color que se va a usar"

copiar

"Copia la porción escogida por el usuario del contenido del receptor al clipboard"

curva

"Llama al método curve:"

curve: aPoint

"Dibuja la curva, el usuario da los extremos de la curva y el punto exterior que da la curvatura de esta"

dibujar: aPane

"Llama al método tDraw:"

elipse

"Llama al método ellipse:"

ellipse: aPoint

"Se encarga de hacer el dibujo de la Elipse"

erase: aPoint

"Borra el punto que ha sido seleccionado"

fuentes

"Cambia el tipo de letra, por la elección del usuario"

goto: aPoint

"Privada - Dibuja una linea a un punto"

infoAcerca

"Despliega la ventana de Acerca de..."

iniPluma: aPane

"Da las características de inicio a la pluma"

limpia

"Llama al método erase"

line: aPoint

"Dibuja una linea de 2 puntos pedidos al usuario"

linea: aPane

"Llama al método line:"

mouseDown: aPane

"Se ejecuta cuando el usuario presiona el botón izquierdo del mouse"

mouseLocation

"Privado - Contesta la localización del mouse dentro del área de trabajo"

nuevo

"Privado - Crea una nueva área de trabajo"

open

"Se encarga de dibujar la ventana del TDrawing, hace los menús y toda la funcionalidad de esta ventana así como la conexión con los métodos a llamar por los menús, es generado en gran parte por WindowBuider"

openBMP

"Abre un archivo con formato BMP"

pegar

"Pega el área que ha sido copiada en el clipboard al receptor"

place: aPoint

"Privado - Pone a la pluma en el punto seleccionado"

rectangle: aPoint

"Se encarga de dibujar el rectángulo, pidiendo al usuario las esquinas de este"

rectángulo

"Llama al método rectangle:"

reset

"Restablece los atributos de la pluma"

saveBMP

"Privado - Al seleccionar esta opción se salva el área de trabajo en un archivo BMP, si este no existe, pregunta al usuario el nombre de este"

saveBMPas

"Privado - pregunta al usuario el nombre del archivo BMP donde será guardada el área de trabajo"

tamPluma

"Se encarga de cambiar el tamaño de la pluma de acuerdo a la elección del usuario"

tDraw: aPoint

"Dibujo en modo libre"

text: aPoint

"En el punto de elección cambia las características del cursor a modo texto para poder escribir el texto que desea escribir el usuario"

texto

"Llama al método text:"

Clases usadas por el TDrawing:

Window: Es una clase abstracta que da el protocolo y los datos comunes que son heredados por todas las subclases de window. Implementa todos los mensajes del anfitrión(mensajes "VM" de windows) que son respondidos por Smalltalk. Si el usuario desea responder a ciertos mensajes nuevos del sistema, deberá primero implementarlos en la clase y entonces agregar los métodos de selector en el orden apropiado del arreglo global de eventos (WinEvents).

Collection: Es la superclase de todas las subclases de collection. Es una clase abstracta que define el protocolo común para todas sus subclases, las colecciones son las estructuras de datos básicas usados para almacenar objetos en grupos de manera lineal o no lineal. Esta clase proporciona el protocolo para el acceso directo o almacenar un elemento en particular en una colección, para accesar todos .los elementos de una colección en un orden particular, o para ejecutar cierto bloque de código para cada elemento accesado.

IdentityDictionary: como en la clase Dictionary, la diferencia entre estos dos está en el almacenamiento interno de las parejas llave/valor. Para esta clase 2 llaves son iguales cuando realmente son el mismo objeto.

SubPane: Es una clase abstracta que proporciona las funciones comunes a los subpanes o controles construidos en una ventana o caja de diálogo.

DialogBox: Es una ventana 'pop-up' usada para desplegar mensajes y recuperar entradas del usuario puede ser modal o sin modo. Una caja de diálogo modal requiere que el usuario finalice la caja de diálogo antes de usar la ventana que abre la caja. Una caja sin modo permite que el usuario continúe usando la ventana sin finalizar la caja de diálogo. Una caja de diálogo puede ser creada utilizando el DialogEditor que viene incluido en el SDK de Windows. Una vez convertido puede ser abierto en Smalltalk usando DialogBox, fromDLLFile:, templateName: o fromModule: id:. Alternativamente las cajas de diálogo pueden ser creadas en Smalltalk usando los métodos de WindowDialog como un pane normal.

FileDialog: Implementa una caja de diálogo que permite al usuario un archivo.

GraphPane: Permite el dibujo de gráficos en un pane, GraphPane tiene como herramienta gráfica a RecordingPen.

RecordingPane: Da las facilidades para dibujo segmentado. Al usar dibujo segmentado una aplicación puede dibujar en un segmento y después duplicar ese segmento. Al duplicar la localización y la escala del dibujo este puede ser modificado. El dibujo segmentado es útil cuando una aplicación necesita dibujar una imagen varias veces, como al hacer un scroll o redesplegar una ventana gráfica. También es útil cuando una imagen esta compuesta por componentes pequeños o cuando varias imágenes comparten los mismos componentes.

WindowDialog: Es una clase abstracta que da el protocolo común para aplicaciones con cajas de diálogo que no usan archivos de recursos. Como WindowDialog es una subclase de ViewManager las cajas de diálogo se implementan casi de la misma forma que las ventanas regulares. La diferencia más importante es que una ventana regular crea sus propios SubPanes como ventanas individuales y cada una recibe sus mensajes de eventos, mientras que una caja de diálogo crea una plantilla describiendo todos los controles y los mensajes son recibidos primero por la ventana principal que los envía al control destino. Las cajas de diálogo implementadas con la subclase WindowDialog cuando sean desplegadas serán módulos en su ventana dueña a menos que estén procesando mensajes del sistema (Windows 'VM' mensajes).

ControlPane: Es una clase abstracta para todas las clases que presentan controles del sistema anfitrión.

ScrollBar: Permite que el usuario cree barras de scroll verticales, horizontales que pueden ser usadas como controles genéricos de desplazamiento. Las barras de scroll generan eventos para indicar # nextLine, # prevLines, # nextPage, # prevPage, #sliderTrack, # sliderPosition y # endscroll. El dueño puede fijar un rango arbitrario, incrementos de linea y de página para cada instancia de la clase ScrollBar.

ViewManager: Es una clase abstracta que implementa los protocolos comunes para la parte de las aplicaciones que involucran la interface con el usuario. Puede manejar una o varias vistas en una misma aplicación. Cada vista no se limita a la parte visual, puede ser auditiva u otra clase de interface soportada por la computadora en cuestión. Es muy

recomendable que se implemente la interface de la aplicación como una subclase de ViewManager. En cada una de estas subclases usualmente se implementa un método open u openon: que especifican todos los componentes incluidos en las vistas, sus atributos y que métodos serán invocados al recibir los eventos del sistema. El resto del trabajo es implementar estos métodos en la misma subclase de ViewManager para responder a diversos eventos.

Los métodos de instancia usados por la aplicación son: addSubpane:, addView:, asParameter, backColor:, changed:, changed: with:, changed: with: with:, clearTextModified, close, foreColor:, label:, labelWithowtPrefix:, mainView:, newTitled:, menuTitled: inView:, model:, modifiedPanes:, openWindow, owner, owner:, paneAt:, panesAt:, parent:, style:, textModified, textModifiedIn:, when: perform:.

No hay métodos de clase para la aplicación.

Object: es la superclase de todas las otras clases y define el protocolo común a todos los objetos. Define el comportamiento por default para desplegar, comparar, copiar, hashing, inspeccionar objetos, acceso de variables de instancia indexadas y manejo de errores. Incluye la capacidad de mantener relaciones de dependencia entre objetos y de mandar mensajes de un objeto a sus dependientes. También da el punto de entrada para el manejo de interrupciones.

Los métodos de instancia usados por la aplicación son: changed, isNil, perform:, yourself.

Set: Representa una colección desordenada de objetos sin llaves externas. Todos los elementos de un set son únicos, por lo tanto no se permiten los duplicados, los sets son "hasheados" para una búsqueda rápida.

Dictionary: Es una colección de pares de objetos llave/valor. Las llaves en un diccionario son únicas del mismo modo los valores pueden ser aplicados. La búsqueda en un diccionario puede ser por llave o por valor. Las búsquedas por llave usan 'hashing' para mayor eficiencia. Los elementos pueden ser introducidos o extraídos de un diccionario como un par de objetos o como una asociación. Internamente un diccionario guarda las parejas de llave/valor como un conjunto de asociaciones mientras que IdentityDictionary guarda las parejas de llave/valor en elementos contiguos de un arreglo.

CONCLUSIONES

En base a lo expuesto en los capítulos anteriores podemos concluir que el software de los 90's tenderá hacia la orientación a objetos, como se está experimentando actualmente, debido principalmente a que el enfoque modular-estructurado no logra satisfacer la demanda de la crisis del software.

El enfoque orientado a objetos es la alternativa a la crisis del software que vivimos, ya que los objetos poseen la flexibilidad suficiente para el desarrollo y mantenimiento de los sistemas que se buscan.

Algunas de las ventajas más importantes que se observan de la Programación Orientada a Objetos es el reuso, tanto de código (a través de componentes estándar), como de algunas etapas del desarrollo del sistema como son análisis, diseño, pruebas y mantenimiento. Lo que consigue un aumento significativo en la calidad y productividad del software.

Otras de las ventajas que se observó, es que el mantenimiento del sistema no implica tener que rediseñar todas las partes del mismo como en la programación estructurada, puesto que las modificaciones requeridas por el sistema se hacen de manera local.

Con las ventajas anteriormente expuestas y con características tales como abstracciones a nivel conceptual, polimorfismo, herencia, entre otros, se puede decir que la programación orientada a objetos será el camino a seguir por las herramientas de desarrollo del futuro.

Sin embargo, a la fecha la programación orientada a objetos no ha sido difundida a nivel internacional de la forma en la que se esperaba, esto debido principalmente a dos razones:

1. El aspecto comercial, debido a que las grandes empresas del software aún ofrecen en su mayoría herramientas de desarrollo basadas en programación estructurada.
2. Aún no se cuenta con un gran conocimiento internacional de objetos, pues solo en algunos países como E.U.A., Inglaterra, entre otros, la programación orientada a objetos se ha desarrollado de manera rápida.

No obstante, aún con estas limitantes o restricciones, la popularidad de los lenguajes orientados a objetos va en aumento, por lo que en un futuro no muy lejano las aplicaciones serán desarrolladas en forma total en una perspectiva orientadas a objetos.

BIBLIOGRAFIA

Atwood, Thomas. **The Object DBMS Standard.** Object Magazine, September/October. 1993.

Booch, G. **Object Oriented Design with Applications.** Redwood City. Ed: Benjamin/Cummings. 1991.

Booch, G. **Software Components with ADA.** Redwood City. Ed: Benjamin/Cummings. 1987.

Carey, T. T. and Mason, R. E. **Information System Prototyping: Techniques, Tools and Metholodogies.** New Paradigms for Softaware Development. IEEE Computer Society Press. 1987.

Coad, P. and Yourdon, E. **Object Oriented Design.** Englewood Cliffs, NJ. Ed: Prentice Hall. 1991.

Coad, P. and Yourdon, E. **Object Oriented Analysis.** 2da. Edición. Englewood Cliffs, NJ. Ed: Prentice Hall. 1991.

Conklin, Dick. **Spotlight on Digitalk: Objects Pioneer Pushes the Envelope.** OS/2 Developer. September/October 1993..

Cox, B. J. Object oriented Programming -An evolutionary Approach. Reading. MA. Ed: Addison Wesley. 1986.

Gomaa, H. The Role of Prototyping in Large Scale Software System Development. Large Scale Systems in Information and Decision Technologies Vol. 12, No. 3. 1987

Jacobson, Ivar. Object-Oriented Software Engineering. ACM Press - Addison Wesley. USA, 1992

Johnson, R. E. and Foote, B. Designing Reusable Classes. Journal of Object-Oriented Programming. June/July 1988.

Lalonde, W. R. and Pugh, J. R. Inside Smalltalk, vol I, Englewood Cliffs, NJ. Ed: Prentice Hall. 1990.

Lalonde, W. R. and Pugh, J. R. Inside Smalltalk, vol II, Englewood Cliffs, NJ. Ed: Prentice Hall. 1991.

Lieberherr, K. J. and Holland, I.M. OAssuring Good Style for Object-Oriented Programs. IEEE Software. September/October 1989.

Meyer, B. Object Oriented Software Construction. Englewood Cliffs, NJ. Ed: Prentice Hall. 1988.

Meyer, B. **Lessons from the Design of the Eiffel Libraries.** Communications of the ACM. 1990.

Parnas, D. L., Clements, P. C. and Weiss, D. M. **Enhancing reusability with information hiding.** ITT Proceedings of the Workshop on Reusability. 1983.

Pressman, Roger S. **Ingeniería de Software. Un Enfoque Practico.** McGraw - Hill. España, 1990

Prieto Diaz, R. and Freeman, P. **Classifying software for reusability.** IEEE Software. January 1987.

Riddle, W. and Williams, L.G. **Software Envioronments Workshop Report.** ACM SIGSOFT Software Engineering Notes, Vol 11, No. 1. 1986

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and Lorensen, W. **Object Oriented Modeling and Design.** Englewood Cliffs, NJ. Ed: Prentice Hall. 1991.

Sage, Andrew P. and Palmer, James D. **Software Systems Engineering.** Wiley-Interscience. USA, 1990

Taylor, David E. **Object Oriented Technology: A Manager's Guide.** Prentice Hall. USA, 1992

Wirfs-Brock, Rebecca, Wilkerson, Brian and Wiener, Lauren. **Designing Object - Oriented Software.** Prentice Hall. USA, 1990