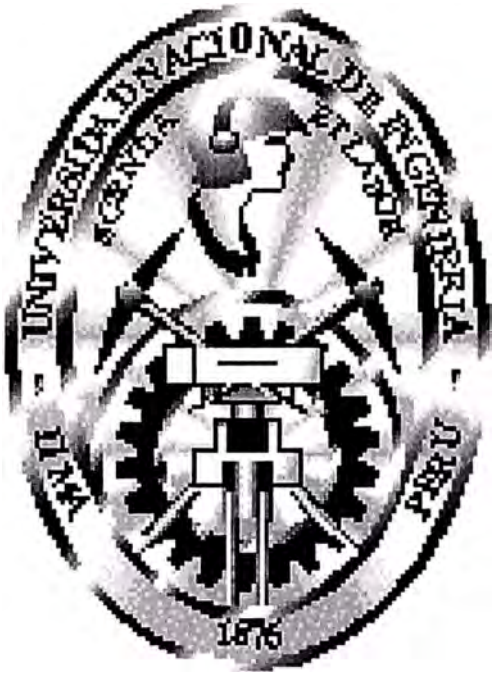
UNIVERSIDAD NACIONAL DE INGENIERIA

Facultad de Ingeniería Industrial y de Sistemas



PERSPECTIVA DEL DESARROLLO RÁPIDO DE APLICACIONES DE SOFTWARE

TESIS

Para Optar el Titulo Profesional de:

INGENIERO DE SISTEMAS

Presentado por el Bachiller:

Jaime Enrique Ortiz Castro

LIMA – PERU
1999

A Frank y Dora
mis amorosos padres.

...he aprendido a contentarme, cualquiera que sea mi Situación. Sé vivir humildemente y sé tener abundancia; en todo y por todo estoy enseñado, así para estar saciado como para tener hambre, así para tener abundancia como para padecer necesidad.
Todo lo puedo en Cristo que me fortalece.

Flp. 4:11-13

INDICE

SUMARIO

Dentro de las actividades más complejas desarrolladas por el hombre se encuentra entre las más notables, la dedicada a la generación o desarrollo de software. En esta actividad y a través de las diferentes etapas que se han experimentado (refiriéndome a los diferentes paradigmas que se han usado, se usan actualmente y al avance de la tecnología en hardware) han permanecido inmovibles o como característica de dicha actividad una serie de problemas relacionados intrínsicamente al desarrollo de proyectos en esta área.

Los especialistas enfrentan problemas diversos, los mismos que recaen o tienen su naturaleza en aquel fantasma conocido por todos como riesgo. El riesgo es inherente a los proyectos y se manifiesta de diferentes maneras, dependiendo básicamente del tamaño del proyecto, así tenemos que el riesgo puede estar asociado a costos, cronogramas, cumplimiento de objetivos, y a otras suertes inclusive de que ni siquiera se termine el proyecto. Cualquiera de las anteriores situaciones de riesgo conllevan a un sin número de malestares que concluyen en pérdidas monetarias, de reputación, confianza, fracasos, etc, etc, etc.

La presente investigación está orientada a conseguir que los cronogramas sean tratados eficientemente, con lo cual garantizaremos la reducción de los riesgos asociados al no cumplimiento de los mismos.

La tesis presenta un enfoque alternativo, para ampliar el espectro de tratamientos de los proyectos de desarrollo de aplicaciones de software.

DESCRIPTORES TEMATICOS

- RAD: DESARROLLO RÁPIDO DE APLICACIONES
- PROTOTIPEO RÁPIDO
- DESARROLLO INCREMENTAL
- DESARROLLO ORIENTADO AL CLIENTE
- RDL: LENGUAJE DE DESARROLLO RÁPIDO
- PRODUCTOS SHRINK-WRAP
- PROGRAMACION ORIENTADA AL OBJETO
- FRAMEWORKS
- CLIENTE/SERVIDOR

CAPITULO 1

INTRODUCCION

En la industria del desarrollo de software, sea este corporativo o comercial, los equipos responsables todos quieren soluciones para un problema importante: ¿Cómo tener los cronogramas de desarrollo sometidos a gran presión bajo control?

Muchos estudios han encontrado que más de las dos terceras partes de los proyectos superan sustancialmente los plazos estimados. En promedio los proyectos grandes fallan en ser desplegados en la fecha convenida, entre en un 40 y 60 por ciento, y el desface del cronograma en promedio aumenta con el tamaño del proyecto.

Año tras año los asuntos relacionados a la velocidad del desarrollo han aparecido como los principales problemas de entre los mas críticos relacionados a la comunidad de desarrollo de software.

Aunque el problema de cronograma de desarrollo de software está ampliamente difundido, algunas organizaciones están efectuando desarrollos rápidos.

En la presente investigación se trata de proveer a los grupos que no están incluidos en el desarrollo rápido con la información que ellos necesitan para moverse hacia el lado que esta desarrollando rápidamente, centrándonos en el cumplimiento de los cronogramas, tratando de encontrar un balance y de efectuar buenas prácticas de gestión para tener un desarrollo rápido de aplicaciones.

Esta elección se ha hecho en base a experiencias personales y de otros que han pasado por situaciones en las cuales el cronograma no se ha cumplido y por lo tanto el entregable no ha sido desplegado a tiempo. Demás esta señalar que en situaciones como ésta, cuando desplegar el artefacto de software se convierte en la única prioridad se desatiende la performance, usabilidad, mantenabilidad, pruebas, etc., lo que trae consecuencias predecibles.

En el estudio se pone especial énfasis en:

- Una estrategia de desarrollo rápido que puede ser aplicada a cualquier proyecto, y las mejores prácticas para hacer esta estrategia viable.
- Revisión de las mejores prácticas de desarrollo-rápido: prototipeo, prácticas orientadas al cliente, lenguajes de desarrollo-rápido, y otras técnicas.
- Una lista de errores clásicos a ser evitados en proyectos de desarrollo-rápido.

El principal objetivo es formular el marco que permite el uso exitoso de la perspectiva del desarrollo rápido de aplicaciones, proporcionándole al desarrollador (organización) de software un enfoque que amplíe el espectro en cuanto al tratamiento de los proyectos de desarrollo de aplicaciones.

De seguro que por medio de la mencionada perspectiva estaremos en condiciones de acortar el tiempo de desarrollo significantemente, posiblemente a la mitad, y potenciar la productividad significativamente. Estaremos en condiciones de hacer esto sin afectar la calidad, costos, performance, o mantenimiento. Pero la mejora no vendrá instantáneamente, no la conseguiremos de una nueva herramienta o método, y no la conseguiremos meramente desempaquetando un producto nuevo.

Ojalá y hubiera alguien que tuviera una solución simple al problema de velocidad de desarrollo. Pero las soluciones simples solo funcionan para problemas simples, y el desarrollo de software no es un problema simple.

Este estudio también incluye un prototipo el cual está basado en mi experiencia profesional en el sector empresarial.

CAPITULO 2

NATURALEZA DEL DESARROLLO RÁPIDO

Muchos proyectos son percibidos como lentos; sin embargo no todos los proyectos son lentos de la misma manera. Algunos esfuerzos de desarrollo son realmente lentos, y otros meramente aparentan ser lentos debido a los esfuerzos puestos en estimaciones inalcanzables.

Un enfoque de estimación de proyectos de software afirma que todo proyecto tiene un tiempo exacto en el cual este debería ser completado. Este enfoque sostiene que si el proyecto es bien desarrollado, habría un 100 por ciento de probabilidad de ser completado en una fecha particular. La figura 2.1, muestra una representación gráfica de ese enfoque.

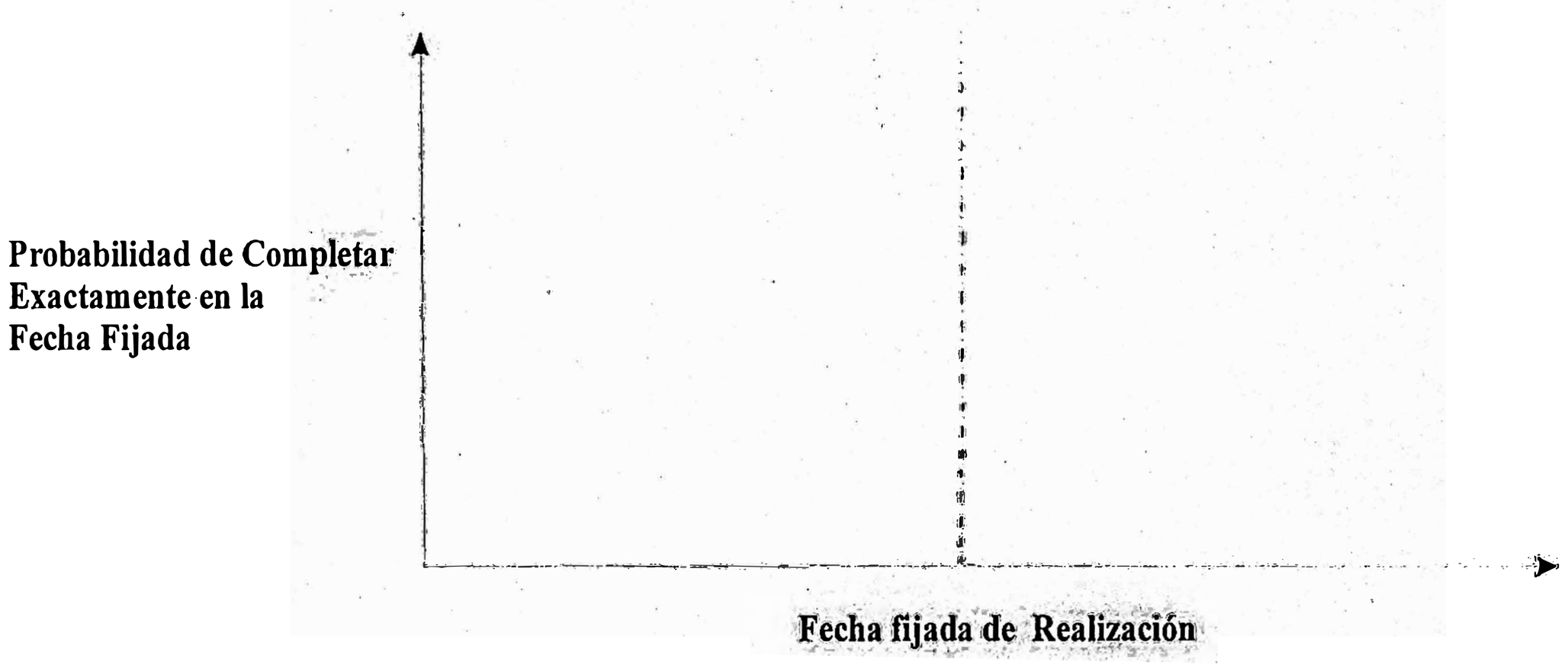


Figura 2-1. Se cree que el proyecto tiene un 100 por ciento de ser completado en una fecha especifica

La experiencia de una gran cantidad de desarrolladores no comparten esta afirmación. Muchos elementos desconocidos contribuyen al cronograma de software. Las circunstancias cambian. Los desarrolladores aprenden más del producto que ellos están construyendo, conforme lo construyen.

Los proyectos de software contienen muchas variables como para poder establecer un cronograma con un 100 por ciento de exactitud. Lejos de tener una fecha particular de finalización de un proyecto; para un proyecto dado hay un rango de fechas a finalizar, de las cuales unas son más probables que otras. La distribución de probabilidad del rango de fechas se muestra en la figura 2.2.

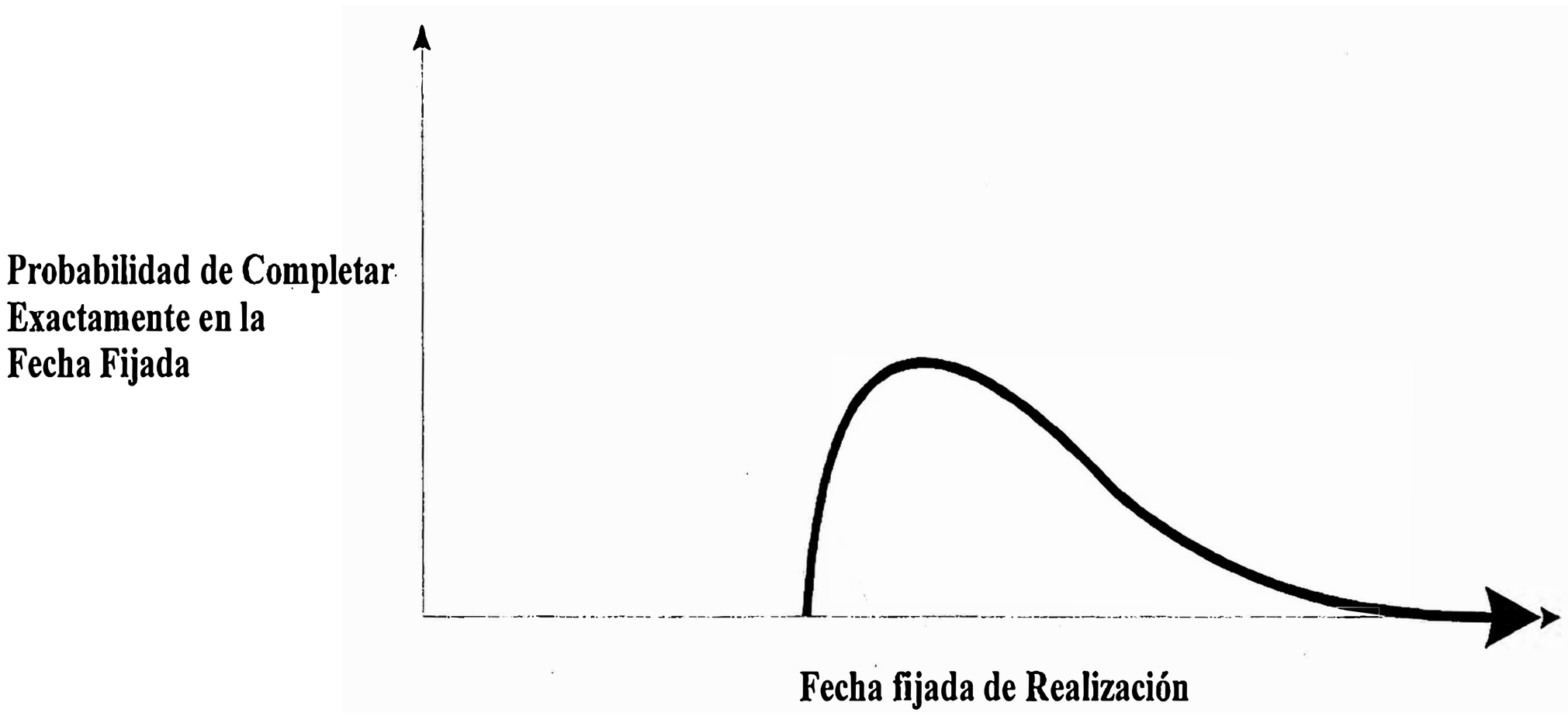


Figura 2-2. Debido a los imponderables, algunas fechas son más probables que otras pero no hay certeza en alguna.

La forma de esta curva de probabilidad expresa diversas asunciones. Una es que hay un límite absoluto en cuan rápido se puede completar un proyecto particular. Completar un proyecto en un tiempo más corto no sólo es dificultoso sino imposible. Otra asunción es que la forma de la curva en el lado "temprano" no es la misma que en el lado "tarde". Aún cuando hay un anguloso límite en cuan rápido un proyecto puede ser completado, no hay

un anguloso límite que muestre cuan lento el proyecto pueda ser completado. Desde que hay mas maneras de hacer que un proyecto sea finalizado tarde antes que temprano, la pendiente de la curva en el lado tarde es más gradual que en el lado temprano.

Estudiosos en la materia han propuesto que los proyectos deberían ser cronogramados de tal manera que la probabilidad de término temprano de los mismos sería igual a la probabilidad de término tarde. En otras palabras, establezcamos el cronograma para el proyecto de tal manera que tengamos una probabilidad de 50/50 de completar el proyecto a tiempo, como se muestra en la figura 2.3.

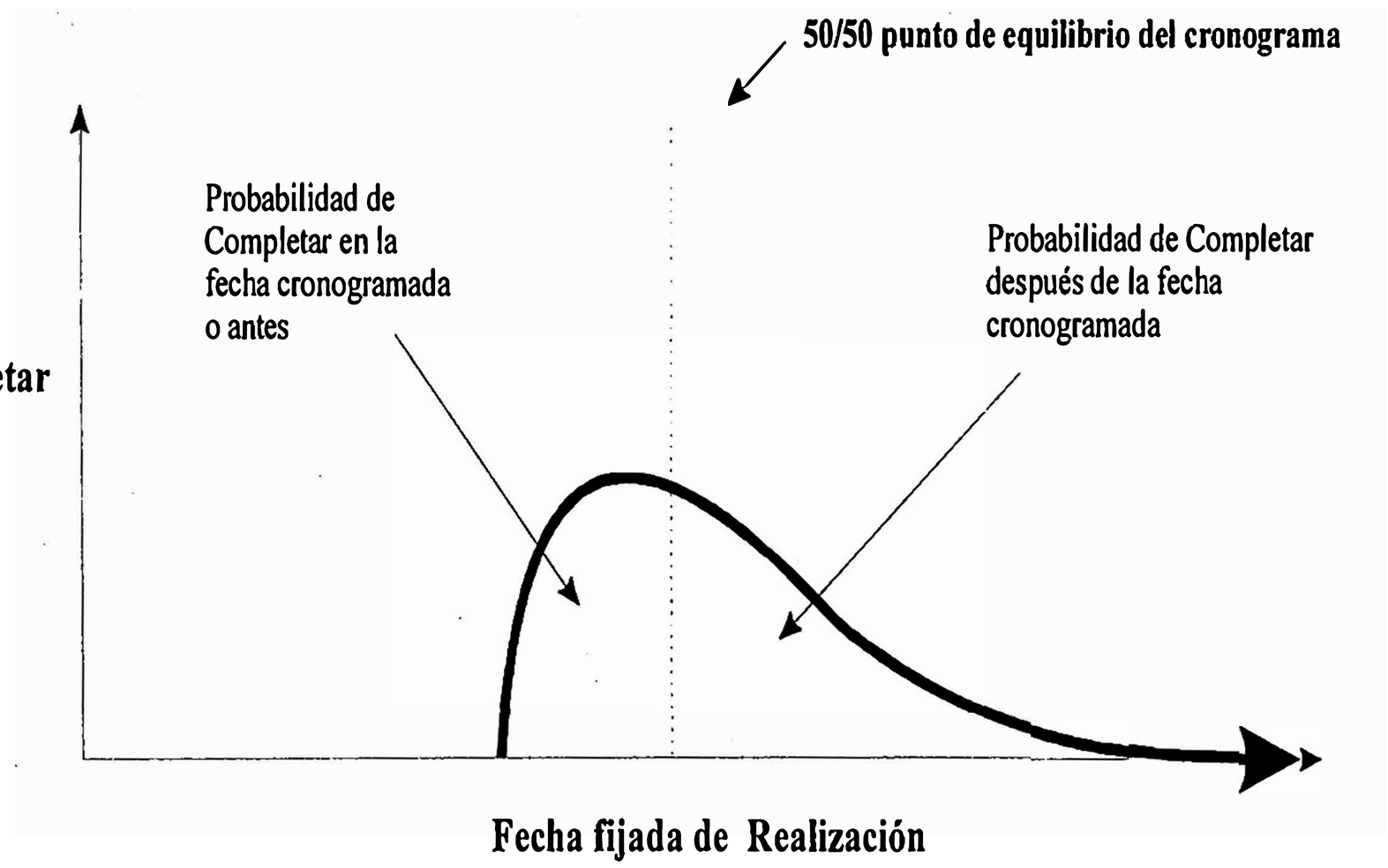


Figura 2-3. Hay un limite en cuan rápido se puede completar un proyecto, pero no hay un limite en cuan tarde pueda realizarse un proyecto. Este cronograma provee una probabilidad de 50/50 de finalizar a tiempo.

2.1 Las Zonas de Desarrollo

La estrategia del cronograma anterior es un punto de partida útil para tener una vista de la naturaleza del desarrollo rápido. Podemos empezar dividiendo el gráfico de probabilidad en diversas zonas, cada una representando una velocidad diferente de desarrollo, como se muestra en la figura 2.4.

El área lejana en el lado izquierdo fuera de la gráfica es la "zona de desarrollo imposible". Esta zona representa un nivel de productividad que ningún proyecto nunca haya alcanzado.

Los proyectos que son cronogramados en esta zona están garantizados a tener sus cronogramas reconsiderados.

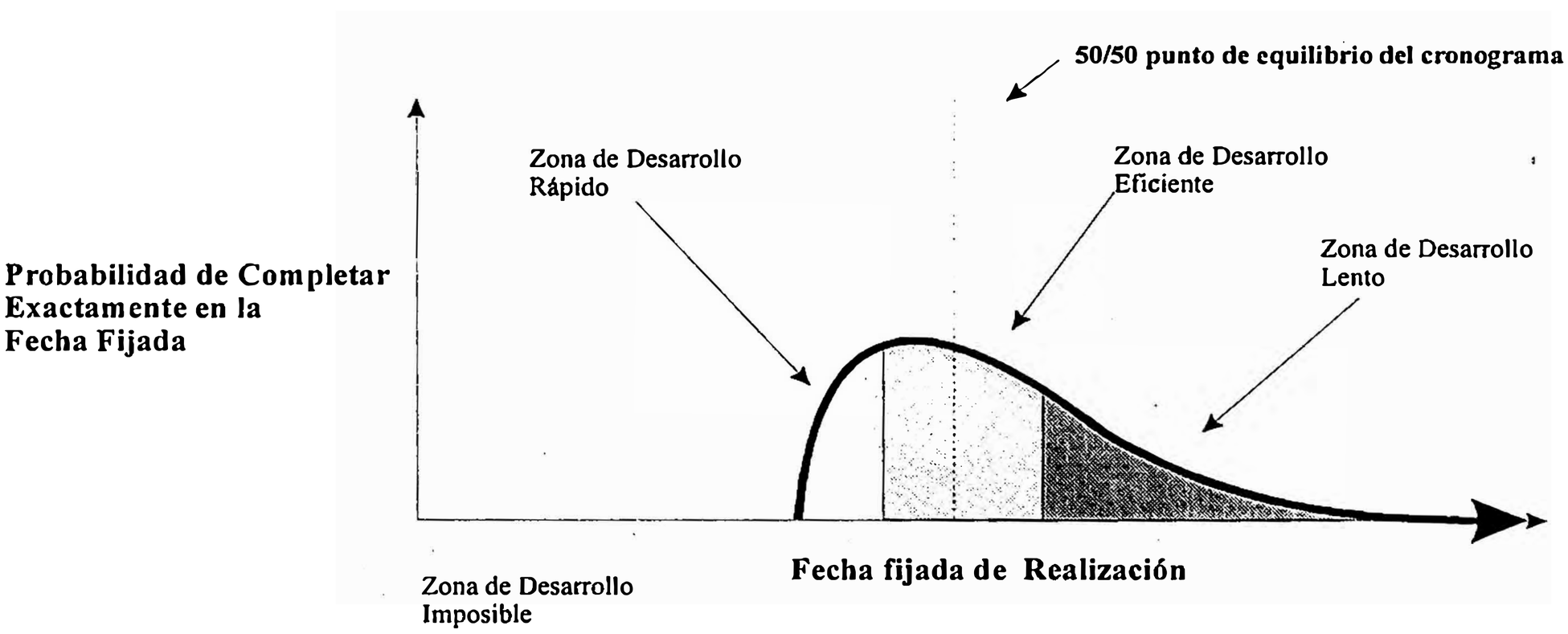


Figura 2-4. Las zonas de planeamiento bajo una curva de cronograma. Muchos proyectos apuntan a la zona imposible (sin saber que es imposible), y terminan en la zona de desarrollo lento (sin quererlo).

El área en el lado izquierdo de la curva es la "zona de desarrollo rápido". Un proyecto que es completado en esta zona es considerado rápido debido a que este tiene una probabilidad menor de 50 por ciento de ser completado en el tiempo especificado. Un equipo de desarrollo que complete un proyecto en esta zona habrá batido todos los récords.

El área en el medio de la curva es la "zona de desarrollo eficiente". Un proyecto que es completado en esta zona es considerado eficiente debido a que este no ha batido los récords ni ha sido batido por el cronograma. El proyecto ha sido desarrollado apegándose a la fecha estimada de completamiento. Las organizaciones efectivas en el desarrollo de software consistentemente cronograman y completan sus proyectos en esta zona, lo cual representa una buena combinación de cronograma y costos.

El área en el lado derecho de la curva es la zona de "desarrollo lento". El proyecto que es completado en esta zona es considerado lento, debido a que este tuvo una probabilidad mayor de 50 por ciento de terminar más temprano.

2.2 ¿Qué es Desarrollo Rápido?

Un Proyecto de Desarrollo Rápido, es aquel proyecto en el cual se incide o se enfatiza en la velocidad de desarrollo. La frase Desarrollo Rápido es solamente descriptiva para contrastar con desarrollos típicos y lentos. En otras palabras el Desarrollo Rápido esta asociado al hecho de desarrollar software más rápido de lo que se esta haciendo.

El desarrollo rápido toma en cuenta herramientas y metodologías que puedan contribuir a su éxito (Ciclos de vida del software, Case, Visual Basic, Delphi, etc.)

2.3 Percepción y Realidad

Algunas veces superar la percepción de desarrollo lento requiere más que proveer signos constantes de progreso. Muchos de los proyectos hoy en día son programados en las zonas rápida o imposible. A muchos proyectos les hacen falta el planeamiento y el compromiso de los recursos necesitados para alcanzar sus cronogramas agresivos. Los planeadores del proyecto aún no comprenden cuan ambiciosos son sus cronogramas, y como la figura 2-5 sugiere, estos son usualmente completados en la zona lenta.

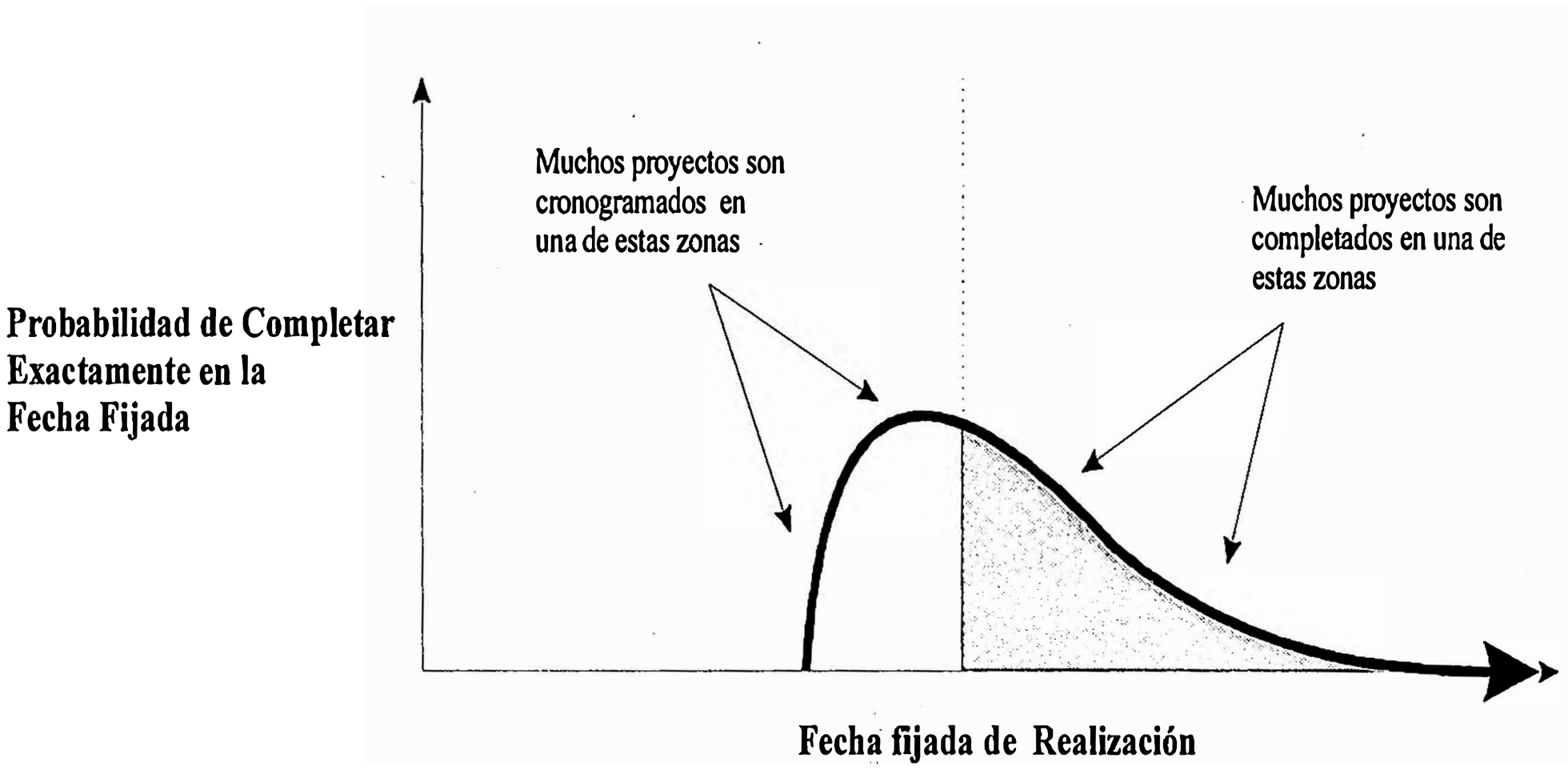


Figura 2-5. Debido a expectativas irreales, muchos proyectos serán percibidos lentos aun cuando son desarrollados en la zona rápida o eficiente.

La distancia entre la fecha programada de finalización y la fecha actual de finalización tiene mucho que ver con la percepción, de por que los proyectos de software son lentos. Si el proyecto en promedio es programado en la zona imposible, pero equipado y completado en la zona eficiente, la gente considerara que este ha fallado aún cuando los desarrolladores han completado este

eficientemente y podrían aún haberlo completado tan rápido como posible con los recursos disponibles.

En términos generales, podemos superar el problema del desarrollo lento en una de dos maneras:

- Direccionar la percepción del desarrollo rápido. Hacer los cronogramas actuales más cortos moviéndonos desde el desarrollo lento al desarrollo eficiente, o moviéndonos del desarrollo eficiente al desarrollo rápido.

- Direccionar la percepción del desarrollo lento. Eliminar los pensamientos positivos, y planear los cronogramas mas realísticamente, alargándolos para cerrar la brecha entre las fechas planeadas y fechas actuales de finalización. Usemos prácticas que resalten la visibilidad del proyecto. Algunas veces los clientes no quieren un incremento en la velocidad de desarrollo, sino que necesitan que los mantengamos informados.

CAPITULO 3

LOS MODELOS CICLO DE VIDA

Todo esfuerzo de desarrollo de software sigue un "ciclo de vida", el cual consiste de todas las actividades que se dan entre el momento en que la versión 1.0 de un sistema nace como un destello en la mente de alguien, y el momento en que la versión 6.2b finalmente toma su último respiro en la última máquina del cliente.

Para nosotros, la principal función de un modelo ciclo de vida es establecer el orden en el cual un proyecto especifica, prototipea, diseña, implanta, revisa, prueba, y ejecuta sus otras actividades. Este establece los criterios que usaremos para determinar el continuar de una tarea hacia la siguiente. Este capitulo enfoca una parte limitada del ciclo de vida completo, el período entre el nacimiento o primera concepción y el despliegue inicial (conocido también como ciclo de desarrollo).

Los estilos de desarrollo del producto varían tremendamente entre las diferentes clases de tareas y el diferente orden de las mismas. Los proyectos diferentes tienen necesidades diferentes, aún si todos necesitan ser desarrollados tan rápido como sea posible, el ciclo de vida que elijamos tiene tanta influencia sobre el éxito del proyecto como cualquier otra decisión de planeamiento que abordemos.

El ciclo de vida apropiado puede hacer más eficaz el proyecto y ayuda a asegurar que a cada paso que nos movamos estemos más cerca de nuestras metas. Dependiendo del ciclo de vida elegido, podemos incrementar la velocidad del desarrollo, calidad, seguimiento y control del proyecto, minimizar los gastos generales, el exponerse a riesgos o mejorar las relaciones con los clientes. El ciclo de vida errado puede ser una fuente constante de trabajo lento, trabajo repetido, trabajo innecesario, y frustración. El no elegir un modelo de ciclo de vida puede producir los mismos efectos.

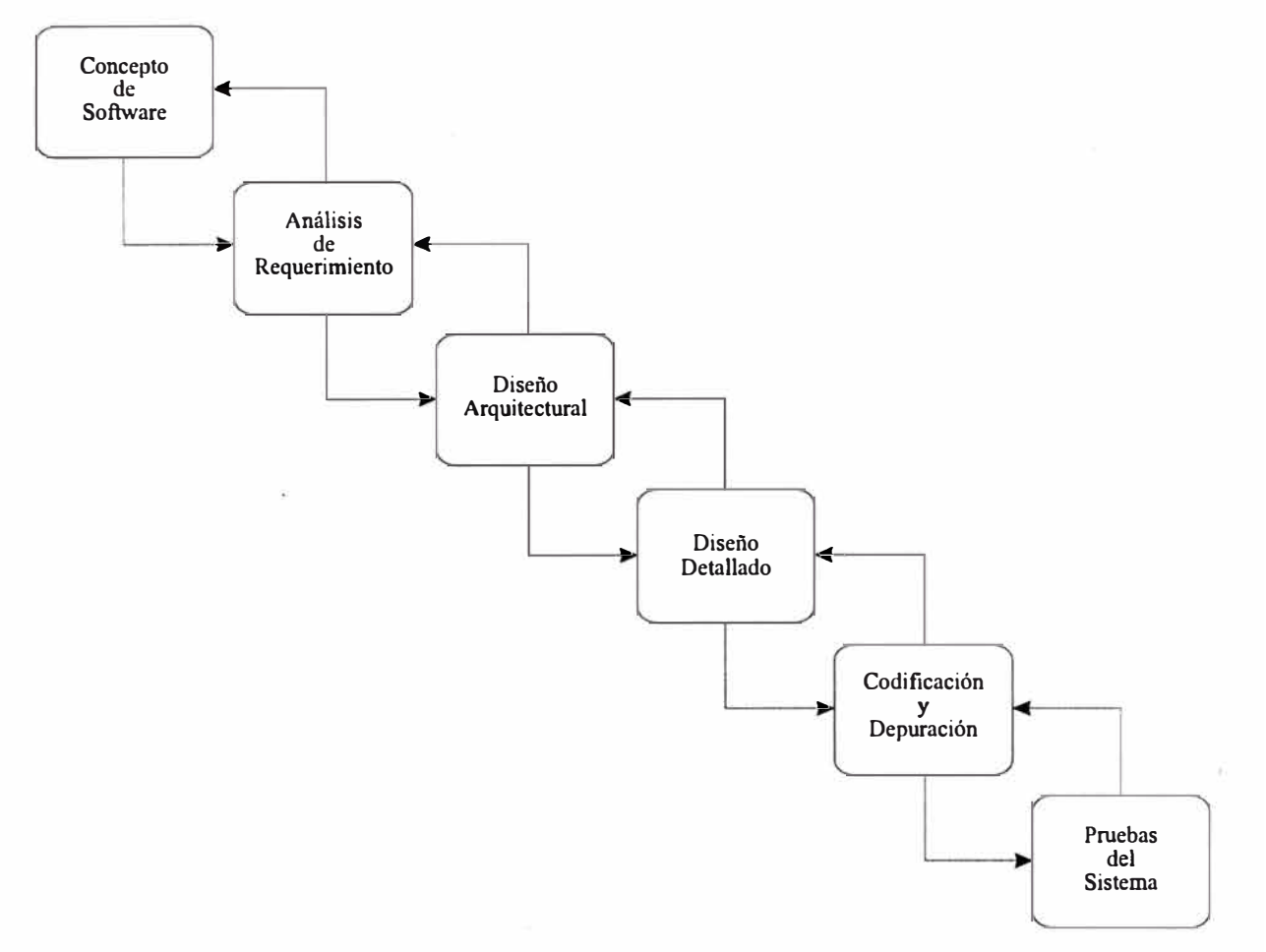
El ciclo de vida más familiar es el bien conocido modelo de ciclo de vida cascada, el mismo que adolece de fallas bien conocidas. Hay otros modelos ciclo de vida disponibles, y en muchos casos estos son mejores alternativas para el desarrollo rápido que el modelo cascada. (El modelo cascada se describe a continuación.)

3.1 EL MODELO CICLO DE VIDA CASCADA PURO

El abuelito de todos los modelos ciclo de vida es el Cascada Puro. Aunque tiene muchos problemas este sirve como base para otros modelos ciclos de vida más efectivos, por lo tanto lo presento primero.

En el modelo cascada un proyecto progresa a través de una ordenada secuencia de pasos desde los conceptos iniciales del software hasta las pruebas del sistema. El proyecto mantiene una revisión al final de cada fase para determinar si es que este esta listo para avanzar a la siguiente fase, por ejemplo del análisis de requerimientos al diseño de la arquitectura. Si la revisión determina que el proyecto no está listo para moverse a la siguiente fase, este permanece en tal fase hasta que esté listo.

El modelo cascada es orientado a la documentación, lo que significa que el principal producto del trabajo son los documentos que se manejan de fase a fase. En el modelo cascada puro, las fases son discontinuas, no hay traslape.



La figura 3.1 muestra el progreso del modelo ciclo de vida cascada puro.

El modelo cascada puro funciona bien con ciclos de productos de los cuales se tiene una definición estable del producto. En estos casos, el modelo cascada nos ayuda a encontrar errores en las etapas iniciales, y a bajo costo del proyecto. Este provee los requerimientos de estabilidad que los desarrolladores anhelan. Si estamos construyendo una emisión de mantenimiento bien definida de un producto existente o migrando un producto existente a nueva plataforma, el ciclo de vida cascada podría ser la elección correcta para el desarrollo rápido.

Este no provee resultados tangibles en la forma de software hasta el final del ciclo de vida, pero para cualquiera que sea familiar con esto, la documentación que se genera provee indicaciones significativas del progreso a través del ciclo de vida.

El modelo cascada trabaja bien para proyectos que son bien comprendidos pero complejos, debido a que podemos beneficiarnos en afrontar la complejidad de una manera ordenada. Este trabaja bien cuando los requerimientos de calidad dominan los requerimientos de costos y cronograma. La eliminación de cambios en la mitad del camino elimina una gran y común fuente de errores potenciales.

Las desventajas del modelo cascada puro surgen de la dificultad de especificar los requerimientos completamente en el comienzo del proyecto, antes de que algún trabajo de diseño haya sido hecho, y antes de que algo de código haya sido escrito.

Usualmente los desarrolladores se quejan de que los usuarios no saben lo que quieren, que los productos de software son complicados y que la gente responsable de la tarea de especificación del software no son comúnmente expertos en computación. Ellos pueden olvidar cosas que aparentemente son simples para ellos, hasta que ven el producto en funcionamiento. Si estamos usando el modelo cascada, olvidar algo puede ser un error costoso. No lo encontraremos sino hasta que estemos probando el sistema y descubrimos que uno de los requerimientos esta errado o no fue contemplado.

Así tenemos que el primer gran problema con el modelo cascada es que este no es flexible. Tenemos que especificar completamente los requerimientos al inicio del proyecto, lo cual puede tomar meses o años antes de tener algún software en funcionamiento. Esto va contra las necesidades de negocios modernos, en los cuales los desarrolladores consiguen implementar la máxima funcionalidad en la última etapa del proyecto. Como Roger Sherman de Microsoft señala, la meta es

siempre no alcanzar lo que se dijo al inicio del proyecto, sino alcanzar lo máximo posible dentro del tiempo especificado y con los recursos disponibles.

Hay quienes han criticado el modelo cascada por no permitir regresar a corregir errores. Pero eso no es completamente correcto. Una vista diferente del modelo cascada que podría tratar la materia dentro de una mejor perspectiva es el modelo ciclo de vida cascada mostrado en la figura 3.2 la cual sugiere que regresar es posible pero es dificultoso.

Esta permitido nadar contra la corriente, pero el esfuerzo podría matarnos. Por ejemplo si al final del diseño arquitectural se participo en muchos eventos mayores que declararon que habíamos terminado con esa fase; mantuvimos una revisión del diseño y firmamos la copia oficial del documento de la arquitectura. Si descubriéramos una falla durante la codificación y depurado, sería engorroso y difícil nadar contra la corriente y replantear la arquitectura.

El modelo ciclo de vida cascada tiene otras debilidades. Algunas actividades se extienden mas allá de las fases, y es difícil acomodar estas actividades en las fases discontinuas del modelo cascada. Para un proyecto de desarrollo rápido, el modelo cascada puede prescribir un monto excesivo de documentación. Si estamos tratando de retener flexibilidad, actualizar las especificaciones puede convertirse en un trabajo de tiempo completo. El modelo cascada genera pocos signos de progreso visible hasta el mismo final. Eso puede crear la percepción de un desarrollo lento aún si esto no es cierto. A los clientes les gusta tener una seguridad tangible de que sus proyectos serán entregados a tiempo.

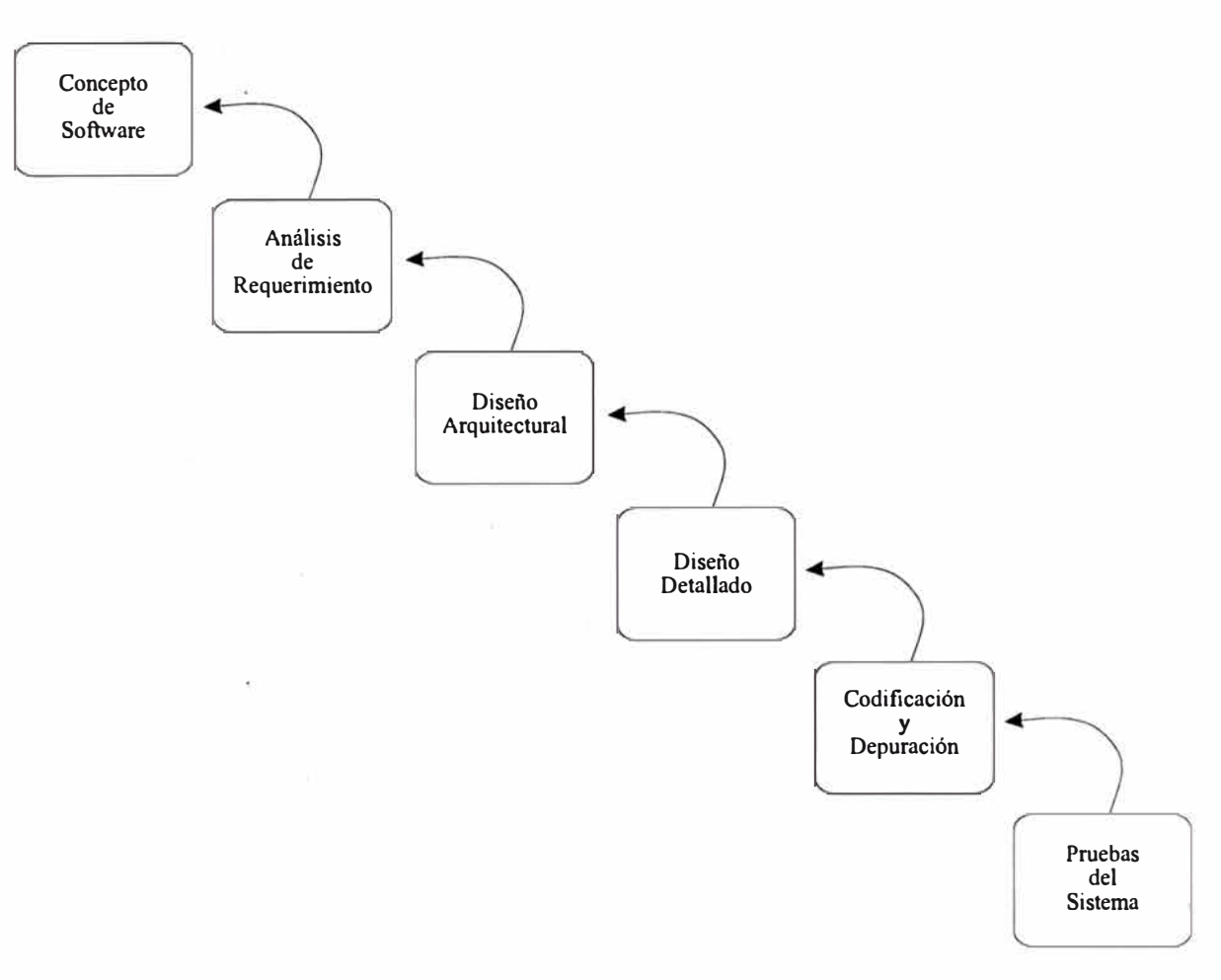


Figura 3-2. El ciclo de vida Salmón. No es imposible regresar, sino que es muy difícil

En resumen, las venerables debilidades del modelo cascada comúnmente lo empobrecen para ser adecuado a un proyecto de desarrollo rápido. Aún en casos en los cuales las fortalezas del modelo cascada puro superan las debilidades, los modelos cascada modificados pueden trabajar mejor.

3.2 LOS MODELOS CASCADA MODIFICADO

Las actividades identificadas en el modelo cascada puro son intrínsecas al desarrollo de software. No podemos evitarlas. De alguna manera tendremos que formular un concepto de software, y tendremos que conseguir los requerimientos de alguna parte. No tenemos que usar el modelo ciclo de vida cascada para recopilar los requerimientos, pero tendremos que usar algo. De la misma manera no podemos evitar tener una arquitectura, diseño, o código.

Más de las debilidades en el modelo cascada puro no emergen de problemas con las actividades sino del tratamiento de estas actividades en una secuencia de fases discontinuas. Por eso podemos corregir las principales debilidades en el modelo cascada puro con relativamente pequeñas modificaciones. Podemos modificar este de tal manera que las fases se traslapen. Podemos reducir el énfasis en la documentación. Podemos permitir mayor regresión.

3.2.1 EL MODELO SASHIMI (CASCADA CON FASES TRASLAPADAS)

El modelo tradicional de cascada solo nos permite un mínimo traslape entre fases en la revisión al final de la fase. Este modelo modificado sugiere un grado más fuerte de traslape, por ejemplo sugiriendo que podríamos estar bien en el diseño arquitectural y posiblemente en parte en el diseño detallado antes de considerar los requerimientos de análisis completos. Este es un enfoque razonable para muchos proyectos, los cuales tienden a ganar la comprensión verdadera de lo que se está haciendo según se avance en el ciclo de desarrollo los mismos que funcionan pobremente con el desarrollo de planes estrictamente secuenciales.

En el modelo cascada puro, la documentación ideal es documentación que un equipo puede alcanzar a otro equipo completamente separado entre dos fases cualesquiera.

La pregunta es, ¿Por qué? Si podemos proveer continuidad entre los conceptos del software, análisis de requerimientos, diseño arquitectural, diseño detallado, codificación y depuración, no necesitamos mucha documentación. Podemos seguir un modelo cascada modificado y sustancialmente reducir las necesidades de documentación.

El modelo sashimi no esta extento de problemas. Debido a que hay traslape entre dos fases, los hitos son más ambiguos, y es más difícil hacer un seguimiento exacto. La ejecución de actividades en paralelo puede guiar a mala comunicación, suposiciones erradas, e ineficiencia.

La figura 3-3 muestra el modelo sashimi.

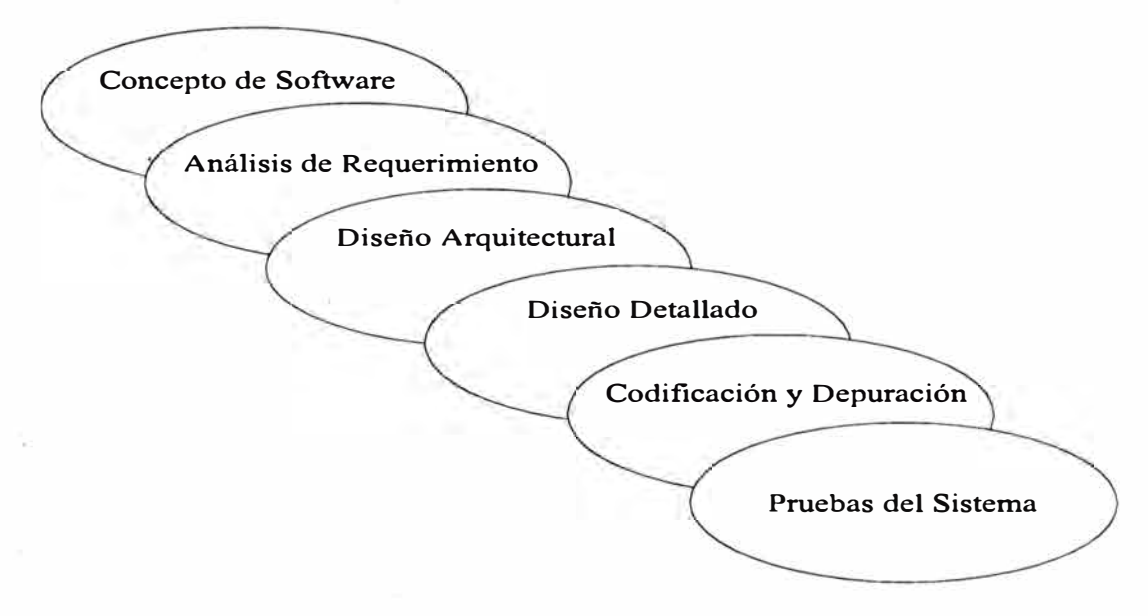


Figura 3-3. El modelo sashimi. Se pueden superar algunas de las debilidades del modelo cascada traslapando sus etapas, pero se enfrentan nuevos problemas.

3.2.2 EL MODELO CASCADA CON SUBPROYECTOS

Otro problema con el modelo cascada puro desde un punto de vista del desarrollo rápido es que debemos completar el diseño arquitectural antes de empezar con el diseño detallado, y tenemos que completar el diseño detallado antes de empezar a codificar y depurar. Los sistemas tienen algunas áreas que contienen sorpresas de diseño, pero ellos tienen otras áreas que hemos implementado muchas veces antes y que no contienen sorpresas. ¿Por qué demorar la implantación de las áreas que son fáciles de diseñar, solo debido a que estamos esperando por el diseño de una área dificultosa? Si la arquitectura ha dividido el sistema en subsistemas lógicamente independientes, podemos concentrarnos en proyectos separados, cada uno de los cuales pueden proceder a su propio paso . La figura 3-4 muestra a vuelo de pájaro lo antes dicho.

El riesgo principal con este enfoque son las interdependencias no previstas. Podemos parcialmente tomar en cuenta eso eliminando dependencias en el momento que trabajemos en la arquitectura o durante la fase del diseño detallado, para separar el proyecto en subproyectos.

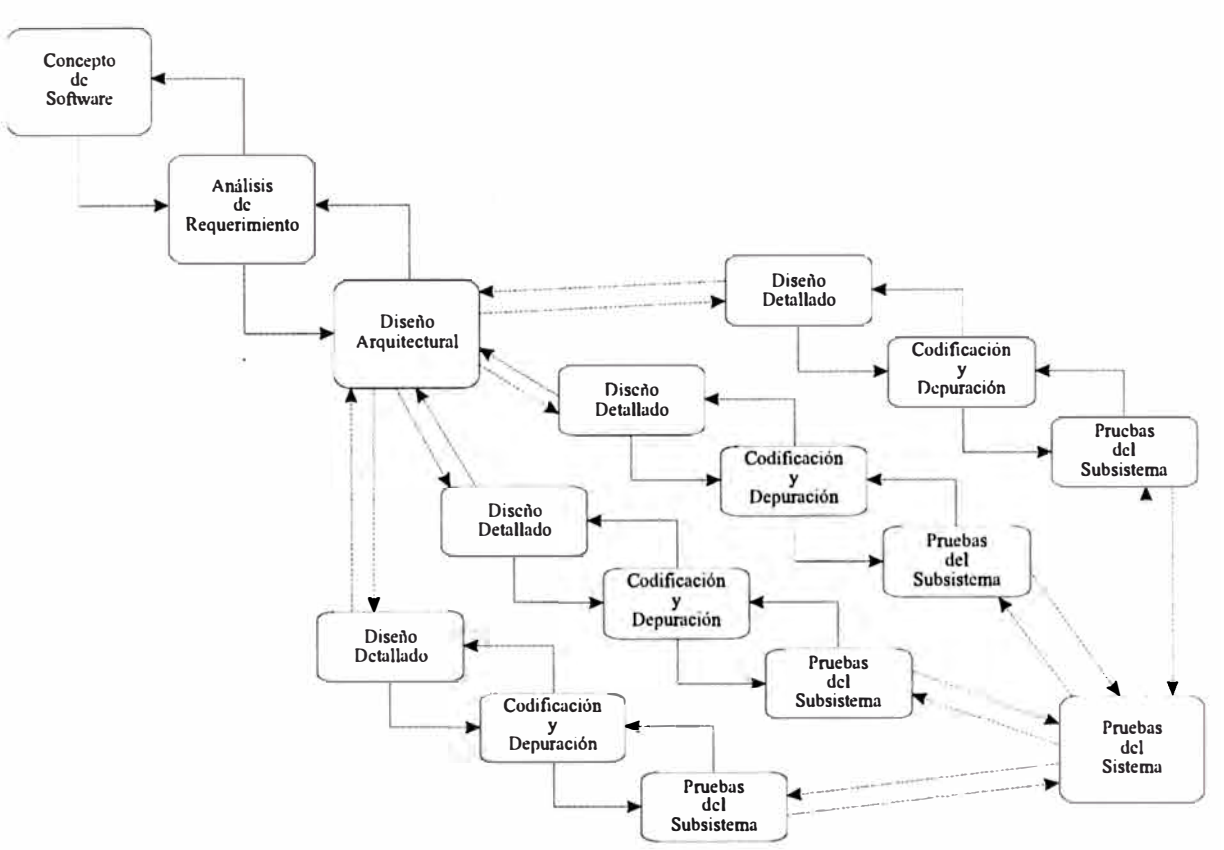


Figura 3-4. El modelo cascada con subproyectos. Un planeamiento cuidadoso puede permitir realizar algunas de las tares del modelo cascada puro en paralelo.

3.2.3 EL MODELO CASCADA CON REDUCCION DE RIESGO

Otra de las debilidades del modelo cascada es que este requiere definir completamente los requerimientos antes de empezar el diseño arquitectural.

Modificando otra vez el modelo cascada solo levemente, podemos poner una espiral de reducción del riesgo al inicio de la cascada para enfrentar el riesgo de requerimientos. Podemos desarrollar un prototipo de la interface del usuario, usar esketches del sistema, conducir conferencias con los usuarios, grabar las interacciones del usuario con sistemas existentes, o usar cualquier otra práctica de recolección de requerimientos que estimemos apropiada.

La figura 3-5 muestra el modelo cascada con reducción de riesgo. El análisis de requerimientos, y el diseño arquitectural se muestran sombreados para indicar que estos podrían ser encarados durante la fase de reducción de riesgo en vez que durante la fase de la cascada.

El preámbulo de reducción del riesgo para el ciclo cascada, no esta limitado a los requerimientos. Podríamos usar este para reducir el riesgo en la arquitectura o cualquier otro riesgo del proyecto. Si el producto depende del desarrollo de un núcleo altamente riesgoso para el sistema, podríamos usar un ciclo de reducción de riesgo para desarrollar completamente tal núcleo, antes de completar el proyecto a escala completa.

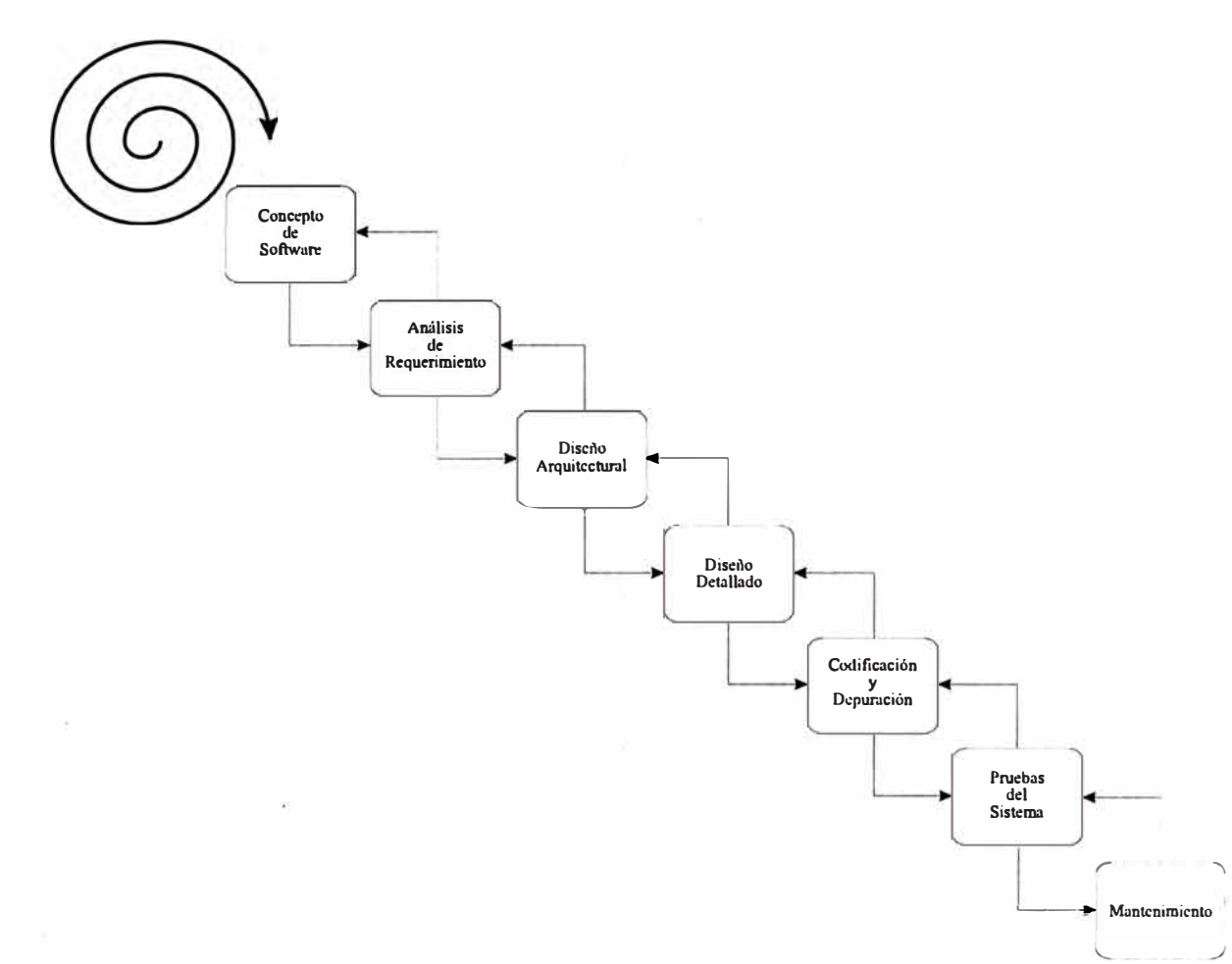


Figura 3-5. Modelo Cascada con reducción de riesgo. Para superar los problemas asociados con la rigidez del modelo cascada, podemos preceder al modelo cascada con una espiral de reducción de riesgo para el análisis de requerimientos o el diseño arquitectural.

3.3 EL MODELO CICLO DE VIDA ESPIRAL

El modelo espiral es un sofisticado modelo de ciclo de vida orientado al riesgo, que desmenuza un proyecto de software en mini proyectos, los cuales resultan de la identificación de riesgos para su posterior reducción. Cada miniproyecto direcciona uno o más riesgos mayores hasta que todos los principales riesgos han sido direcccionados. El concepto de riesgo está ampliamente definido en este contexto, y puede estar referido a requerimientos pobremente comprendidos, arquitectura pobremente comprendida, problemas potenciales de performance, problemas en la tecnología de soporte, etc.

Un proyecto espiral empieza en pequeña escala, se exploran los riesgos, se planea para manipular los riesgos y luego se decide si se tomara el siguiente paso en el proyecto para hacer la siguiente iteración de la espiral. El beneficio en el desarrollo rápido se deriva no de un incremento en la velocidad de desarrollo sino de la reducción continua del nivel de riesgo, el cual tiene un efecto en el tiempo requerido para entregar el producto.

Después que todos los riesgos principales han sido direccionados, el modelo espiral termina como un modelo de ciclo de vida cascada. La figura 3-6 ilustra el modelo espiral, al cual algunos se refieren también afectivamente como "el rollo de canela".

La figura 3-6 es un diagrama complicado, y merece ser estudiado. La idea básica en el diagrama es que empecemos a una pequeña escala en la mitad de la espiral, exploremos los riesgos, confeccionemos un plan para manejar los riesgos, y luego enfoquemos la siguiente iteración. Cada iteración nos mueve en el proyecto hacia una escala mayor. Desenrollamos una vuelta del rollo de canela, chequeamos para asegurarnos lo que queremos, y luego empezamos trabajando en la siguiente vuelta.

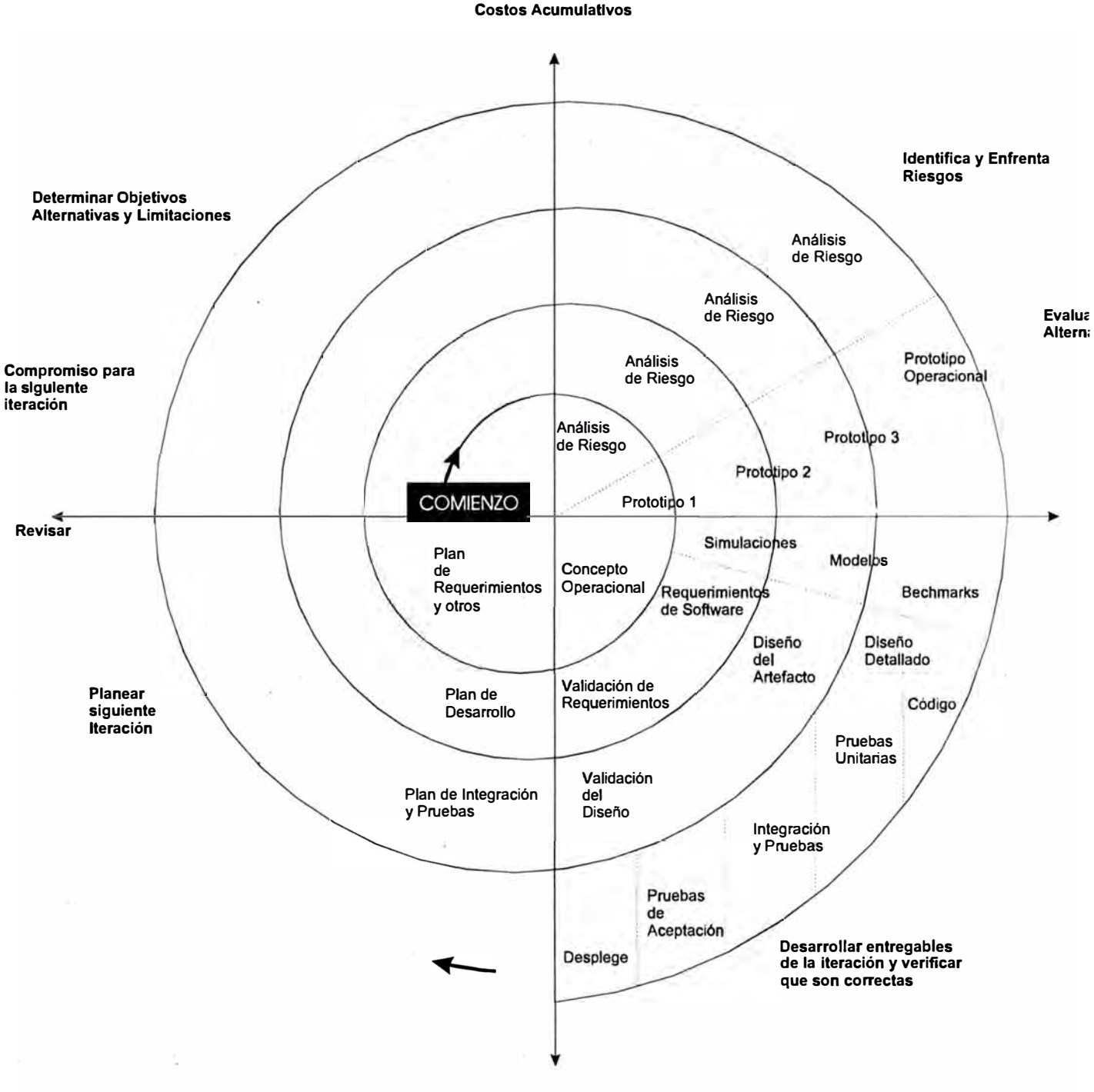


Figura 3-6. El modelo espiral. Se empieza de a pocos y se expande el alcanze del proyecto en incrementos. Se expande el alcanze solo despues de haber reducido los riesgos para el siguiente incremento a un nivel aceptable.

Cada iteración envuelve los seis pasos mostrados en negritas al filo de la espiral:

1. Determinar los objetivos, alternativas, y restricciones
2. Identificar y resolver riesgos
3. Evaluar alternativas
4. Desarrollar los entregables por cada iteración, y verificar que ellos estén correctos
5. Planear la siguiente iteración
6. Enfoque de la siguiente iteración (si se decide tener una más).

En el modelo espiral, las primeras iteraciones son las menos costosas. Se dedica menos desarrollando el concepto de operación que lo que se dedica desarrollando los requerimientos, y menos desarrollando los requerimientos que lo que se dedica desarrollando el diseño, implementando el producto y haciendo las pruebas.

No tomemos el diagrama más literalmente de lo que debe ser tomado. No es importante que tengamos exactamente cuatro bucles alrededor de la espiral, y no es tampoco importante ejecutar los seis pasos exactamente como se indican, aunque es usualmente un buen orden a seguir. Podemos adecuar cada iteración de la espiral a las necesidades del proyecto.

Se puede combinar el modelo con otros modelos de ciclo de vida en un par de maneras diferentes. Se puede empezar el proyecto con una serie de iteraciones de reducción del riesgo, hasta que el riesgo sea reducido a un nivel aceptable, luego podemos concluir el desarrollo con un ciclo de vida cascada.

Se puede incorporar otros modelos de ciclo de vida dentro del modelo espiral. Por ejemplo, si uno de los riesgos es que no estamos seguros de alcanzar las metas de performance, podríamos incorporar una iteración de prototipeo para investigar si podemos alcanzar las metas.

Una de las ventajas más importantes del modelo espiral es que según el costo se incremente, el riesgo decrece. A más dinero y tiempo invertido, el riesgo decrece, lo cual es justamente lo necesitado en un proyecto de desarrollo rápido.

El modelo espiral provee al menos tanta gestión de control como el modelo cascada tradicional. Tenemos los puntos de control al final de cada iteración. Debido a que el modelo es orientado al riesgo, este nos provee de indicaciones tempranas acerca de riesgos superables. Si el proyecto no puede ser hecho por razones técnicas u otras, esto será detectado tempranamente y la corrección no costará mucho.

La única desventaja del modelo espiral es su complicación. Este requiere concientización, atención y administración de alto nivel. Puede ser dificultoso definir objetivos y hitos verificables que indiquen si estamos listos para adicionar la siguiente capa del rollo de canela. En algunos casos el desarrollo del producto es directo y los riesgos del proyecto son lo suficientemente modestos, que no necesitamos la flexibilidad y gestión de riesgo que el modelo de espiral provee.

3.4 MODELO PROTOTIPEO EVOLUTIVO

El prototipeo evolutivo es un modelo ciclo de vida en el cual el sistema es desarrollado en incrementos, de tal manera que este puede ser modificado fácilmente en respuesta a la retroalimentación del cliente.

El Prototipeo Evolutivo es un enfoque de desarrollo en el cual desarrollamos primero partes seleccionadas del sistema, para después evolucionar el resto del sistema a partir de estas partes. Generalmente los esfuerzos en el prototipeo evolutivo empiezan prototipeando la interface del usuario, y luego evolucionando hacia el sistema final a partir de este.

A diferencia de otras clases de prototipeo, en el Prototipeo Evolutivo no desechamos el código del prototipo, si no que este también evoluciona hacia el código final que se despliega. La figura 3-7 muestra como este trabaja.

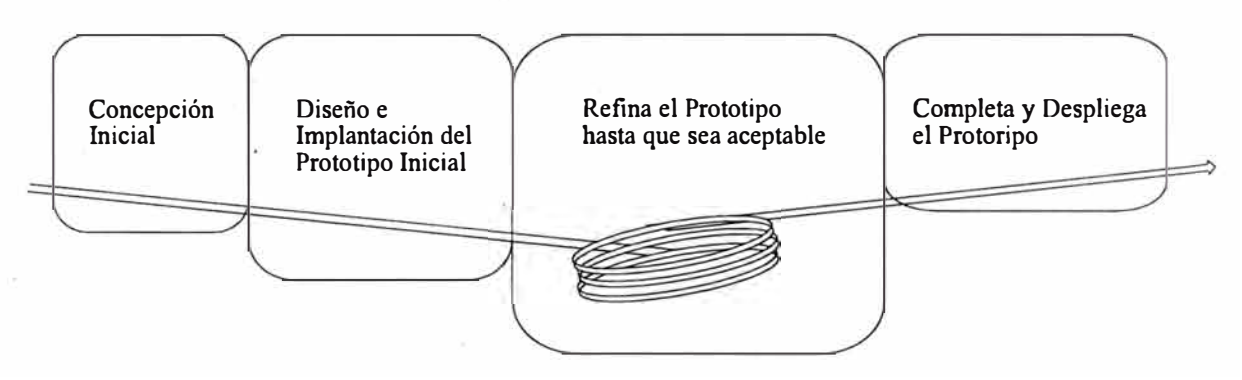


Figura 3-6. El modelo prototipeo evolutivo. Con el prototipeo evolutivo se empieza diseñando e implementando las partes más prominentes del sistema, y luego se refina el prototipo hasta que se completa. El prototipo se convierte en el artefacto desplegado.

El Prototipeo Evolutivo soporta el desarrollo rápido enfrentando los riesgos al principio, ya que se empieza el desarrollo con las áreas más riesgosas del sistema.

Si podemos superar los obstáculos, entonces podemos evolucionar el resto del sistema desde el prototipo. Si no podemos, entonces estamos en posibilidad de cancelar el proyecto sin haber desperdiciado mas recursos que los necesarios para descubrir que los obstáculos fueron insuperables.

El Prototipeo Evolutivo es una de un conjunto de prácticas que son comúnmente descritas como "Prototipeo Rápido". El término "Prototipeo Rápido" generalmente se refiere al Prototipeo Evolutivo, pero algunas veces también se refiere a cualquier práctica de prototipeo.

Uso del Prototipeo Evolutivo

El Prototipeo Evolutivo es una actividad exploratoria que se usa cuando no sabemos en primera instancia exactamente que es lo que necesitamos construir. Debido a las áreas de incertidumbre de un desarrollo, el esfuerzo varía de un proyecto a otro, el primer paso es identificar la parte del sistema a usar como punto de partida. Las dos opciones principales son empezar con las partes más visibles o las partes más riesgosas.

La interface del usuario es comúnmente ambas, la parte más visible y la más riesgosa, de tal manera que esta es la parte más obvia para empezar.

Después de que hemos construido la primera parte del sistema, demostramos ésta y luego continuamos el desarrollo del prototipo basados en la retroalimentación recibida. Debido a que el esfuerzo del prototipeo es orientado al usuario, continuaremos solicitando retroalimentación del cliente y refinando la interface del usuario, hasta que todos convienen que el prototipo es suficientemente bueno.

Luego se desarrolla el resto del sistema usando el diseño y código del prototipo como fundamento.

Paralelamente a la refinación del prototipo se van desarrollando otras partes del sistema que tienen que ver con esfuerzos técnicos, tales como la base de datos. En este caso el feedback que buscaremos no viene necesariamente del cliente. La misión aquí podría ser el establecer la performance de la base de datos en cuanto a tamaño con un número realístico de usuarios finales. Los mismos principios se aplican a calculaciones complejas, performance de interacción, restricciones en tiempo de respuesta en tiempo real, tamaño de los datos, y los aspectos de prueba de conceptos en sistemas de última tecnología.

Se puede usar el Prototipeo Evolutivo en mayor o menor grado en una amplia gama de proyectos. Este es probablemente bien adecuado para sistemas de negocios en el cual los desarrolladores pueden tener frecuentemente, interacciones informales con los usuarios finales. Pero este también es adecuado para productos comerciales, productos shrink-wrap y productos de sistemas si es que se puede conseguir la participación de los usuarios finales. La interacción del usuario para esta clase de proyectos generalmente necesitará ser más estructurada y formal.

Los productos de software *shrink-wrap* son aquellos empaquetados y vendidos comercialmente empaquetados. Esto incluye productos del mercado horizontal como procesadores de texto, y hojas de cálculo, y productos del mercado vertical tales como programas de análisis financiero, gestión de casos legales etc.

Gestión del riesgo en el Prototipeo Evolutivo

En vez de tener una gran lista de riesgos por que preocuparse, el prototipeo Evolutivo es una práctica de relativamente bajo nivel de riesgo.

A continuación detallo algunos riesgos a mantener presentes.

Cronogramas irreales y expectaciones de presupuestos. El Prototipeo Evolutivo puede crear expectaciones irrealizables del desarrollo global del cronograma.

Cuando los usuarios finales, gerentes, o distribuidores ven progresos rápidos en un prototipo, estos generalmente hacen Asunciones irreales acerca de que tan rápido podemos desplegar el producto final. En algunos casos, la gerencia notifica al cliente con expectaciones de cronogramas irreales. Luego los clientes reclaman cuando ellos escuchan de que el software lo recibirán mas tarde de lo que lo esperaban, o fueron dichos.

El trabajo de las partes más visibles es fácil de completar rápido, pero gran parte del trabajo en proyectos de programación no es obvio a los clientes o usuarios finales. El trabajo de baja visibilidad incluye acceso robusto a bases de datos, mantenimiento de bases de datos, integridad, seguridad, redes, conversión de datos entre sucesivas versiones de productos, diseño para uso de grandes volúmenes, soporte multiusuario, soporte para múltiples plataformas, etc. Lo mejor que un trabajo se haga, lo menos que éste se notara.

Otro problema con las expectaciones de esfuerzo creado cuando el prototipeo empieza con la interface del usuario es que la funcionalidad que aparenta ser la más fácil desde el punto de vista del usuario es la más dificultosa de implantar desde el punto de vista del desarrollador.

Finalmente los prototipos son generalmente diseñados para manipular solo los casos nominales, ellos no son expectados a manipular los casos excepcionales. Lo anterior sugiere que aún cuando el prototipo este funcionalmente listo para todos los casos nominales, aún habrá que poner mucho esfuerzo para implantar los casos excepcionales.

Si la gerencia no comprende la diferencia entre crear un prototipo limitado y un producto a escala completa, esto creará un riesgo de sub-presupuestación en proyectos de prototipeo. Las estadísticas han reportado casos en los cuales las gerencias han presupuestado proyectos por 10 semanas, los cuales han tomado hasta dos años. Las gerencias creyeron que el prototipeo produciría resultados funcionales a escala completa en un lapso de tiempo que estuvo fuera de la realidad por un factor de 10.

Podemos gestionar este riesgo manejando explícitamente las expectaciones de usuarios finales, gerentes, desarrolladores, y distribuidores. Asegurémonos de interactuar con el prototipo de una manera controlada en el cual podamos explicar las limitaciones del prototipo, y asegurarnos que ellos comprendan la diferencia entre la creación de un prototipo y la creación de software completamente funcional.

Extensión del proyecto. Con el prototipeo evolutivo es imposible saber al comienzo del proyecto cuanto tiempo tomará la creación de un producto aceptable. No se sabe cuántas iteraciones se darán o cuanto durará cada iteración.

El riesgo es mitigado de alguna manera por el hecho de que los clientes pueden ver signos uniformes de progreso y tienden a estar menos nerviosos que si se tratara de esperar por un producto desarrollado con un enfoque tradicional.

Feedbak pobre del cliente o usuario final. El prototipeo no garantiza una calidad alta en la interacción con el usuario o en el feedback con el cliente. Los usuarios finales y los clientes no siempre saben lo que están viendo cuando tienen al frente un prototipo. Un fenómeno común es que ellos están arrollados por la demostración en vivo del software, que no ven mas allá para comprender lo que el prototipo realmente representa. Si ellos nos dan un sello de aprobación a primera vista, podemos estar virtualmente seguros que ellos no comprenden completamente lo que

están viendo. Asegurémonos que los usuarios finales y clientes cuidadosamente estudien el prototipo lo suficiente, para proveernos con feedback significativo.

Pobre Performance del producto. Varios factores pueden contribuir a un pobre performance del producto, incluyendo:

- No considerar la performance en el diseño del producto
- La mantención de código ineficiente, pobremente estructurado que fue desarrollado rápidamente para el prototipo, en vez de evolucionar este hacia un producto de calidad final.

Podemos dar dos pasos para minimizar los riesgos de performance pobre:

- Considerar la performance temprano al inicio. Asegurémonos de inferir, y chequear que el diseño del prototipo soporta un adecuado nivel de performance.
- No desarrollemos cualquier tipo de código para el prototipo. La calidad del código del prototipo necesita ser lo suficiente buena para soportar extensivas modificaciones, lo que significa que este tiene que ser tan bueno como el código de un sistema desarrollado tradicionalmente.

Expectativas de performance irreales. El riesgo de performance irreal esta relacionado al riesgo de pobre performance. Este riesgo se origina cuando el prototipo tiene mucha mejor performance que el producto final. Debido a que un prototipo no tiene que hacer todo el trabajo que si le compete al producto final, este puede algunas veces ser mucho más rápido comparado al producto final. Cuando los clientes ven el producto final, este puede parecer ser inaceptablemente lento.

Muchos casos estudio han reportado la falla del proyecto como resultado de expectativas lentas creadas por el prototipo.

Podemos encarar este riesgo explícitamente manejando las expectaciones que los clientes desarrollan a partir de la interacción con el prototipo. Si el prototipo es muy rápido, adicionemos retrasos artificiales, de tal manera que la performance del prototipo se aproxime a la performance del producto final. Si el prototipo es muy lento, expliquemos a los clientes que la performance del prototipo no es un indicador de la performance del producto final.

Diseño pobre. Muchos proyectos de prototipeo tienen mejores diseños que proyectos tradicionales, pero varios factores contribuyen al riesgo de diseño pobre.

- El diseño de un prototipo puede deteriorarse, y su implantación puede apartar el diseño original durante sucesiva etapas (así como la mayoría de productos de software tienden a deteriorarse durante el mantenimiento). El producto final puede heredar parches del diseño desde el prototipo del sistema.
- Algunas veces el feedback del cliente o el usuario direccionan el producto en una dirección inesperada. Si el diseño no anticipa la dirección, este podría ser enderezado para adecuarlo sin tener que hacer un rediseño completo.
- El diseño que fluye fuera de la actividad del prototipeo enfocado en la interface del usuario podría enfocar excesivamente en la interface del usuario; se podría fallar en no tomar en cuenta otras partes del sistema que deberían también tener fuertes influencias en el diseño del sistema.
- El uso de un lenguaje de propósito especial para prototipeo puede resultar en un diseño pobre del sistema. Los entornos que son adecuados para obtener progresos rápidos durante las etapas de inicio del proyecto son algunas veces pobres en adecuación para mantener la integridad del diseño en las etapas finales del desarrollo.

Se pueden mitigar los riesgos de calidad de diseño atacando a la raíz del problema.

Limitemos el alcance del prototipo a áreas especificas del producto, por ejemplo, a la interface del usuario. Desarrollemos esta parte limitada del producto usando un RDL, y evolucionemos este, luego continuemos el desarrollo del resto del producto como si se tratara de un desarrollo tradicional sin tener en cuenta el enfoque de prototipeo.

Cuando creemos el diseño global del sistema, hagamos un esfuerzo consciente para no poner demasiado énfasis en la interface del producto o cualquier otra parte del sistema que va a ser prototipeado. Minimicemos el impacto que puede tener el diseño de una área en el diseño como un todo.
Incluyamos una fase de diseño con cada iteración del prototipo para asegurar que el diseño también evoluciona y no solo el código. Usemos una hoja de chequeos de diseño en cada etapa para chequear la calidad del producto evolucionado.

Evitemos usar desarrolladores no experimentados en un proyecto de prototipeo. El Prototipeo Evolutivo requiere que los desarrolladores visionen decisiones de diseño mas temprano en el desarrollo del proceso que con otros enfoques. Los desarrolladores inexpertos comúnmente están pobremente equipados para tomar decisiones acertadas de diseño bajo ciertas circunstancias. El estudio de casos de proyectos con prototipeo ha reportado proyectos que han fallado debido a la inclusión de desarrolladores inexpertos. Estos también han reportado proyectos que han tenido éxito solo debido a los desarrolladores expertos incluidos en los proyectos.

Mantenimiento pobre. Algunas veces el Prototipeo Evolutivo contribuye a un pobre mantenimiento. El alto grado de modularidad necesitado por el prototipeo evolutivo efectivo es propicio para el mantenimiento del código. Pero cuando el

prototipo es desarrollado extremadamente rápido, este es también algunas veces desarrollado extremadamente enredado. En la literatura publicada en prototipeo, se dan mas casos de mantenimiento engorroso con enfoques tradicionales que con prototipeo evolutivo.

Minimicemos el riesgo de mantenimiento tomando unos pasos simples. Asegurémonos que el diseño es el adecuado usando las líneas guías listados en la discusión del riesgo de "diseño pobre" (en la sección anterior). Sigamos convenciones normales para nombrar objetos, métodos, funciones, datos, elementos de la base de datos, librerías, y cualquier otros componentes que serán mantenidos conforme el prototipo madure. Sigamos otras convenciones de código para obtener buenas estructuras (layout) y comentarios. Recuerde a su equipo que ellos tendrán que usar su propio código a través de múltiples ciclos de emisión del producto, lo cual les dará un fuerte incentivo para hacer el código mantenible.

Características adicionales. Los clientes y los usuarios finales típicamente tienen acceso directo al prototipo durante un proyecto de prototipeo evolutivo, y esto puede algunas veces guiar a un incrementado deseo por características adicionales. En adición a gestionar las interacciones con el prototipo, usemos prácticas normales de administración de cambios para lo cual debemos tener en cuenta:

- Permitir cambios que ayuden a producir el mejor posible producto en el tiempo disponible. No permitamos los demás cambios.
- Permitamos a todas las partes que serán afectadas por el cambio propuesto evaluar el cronograma, recursos, y el impacto del cambio en el producto.
- Notifiquemos a las partes en las periferias del proyecto de cada cambio propuesto, su impacto evaluado, y si es que este fue aceptado o rechazado.
- Proveamos una auditoria de las decisiones relacionadas al contenido del producto.

Ineficiente uso del tiempo de prototipeo. El prototipeo es usualmente dado para acortar el cronograma de desarrollo. Paradójicamente, los proyectos comúnmente desperdician tiempo durante el prototipeo y no ahorran tiempo como deberían. El prototipeo es un proceso exploratorio e iterativo que debe ser desarrollado cuidadosamente. Los desarrolladores no siempre saben cuan cercanos están de un prototipo al inicio, y desperdician tiempo ocupándose de características que después son excluidas del producto. Los desarrolladores algunas veces le dan al prototipo una robustez que este no necesita, o desperdician tiempo trabajando en un prototipo sin moverse en una clara dirección.

Para usar el tiempo del prototipeo efectivamente, hagamos del seguimiento y control del proyecto una prioridad. Cuidadosamente desarrollemos el cronograma de actividades del prototipo. Al final en el proyecto podríamos separar el proyecto en partes de días o semanas, pero durante el prototipeo separemos este en horas o días. Asegurémonos de no poner mucho esfuerzo dentro del prototipo ni en la evolución de alguna parte del sistema hasta que no estemos seguros que esa parte del sistema va a permanecer.

Efectos colaterales del prototipeo evolutivo

En adición a sus beneficios de desarrollo rápido, el Prototipeo Evolutivo produce varios efectos colaterales, la mayoría de los cuales son beneficiosos. El prototipeo tiende a:

- Mejorar la moral de los usuarios finales, clientes y desarrolladores debido a la visibilidad del progreso que se va obteniendo
- Darnos un temprano feedback indicativo de sí es que el producto final será aceptado
- Decrementar la longitud del código global debido a mejores diseños y más reusabilidad

- Mas baja tasa de defectos debido a la mejor definición de requerimientos
- Curvas de esfuerzo más suaves, reduciendo el efecto de entrega a tiempo (el cual es común con enfoques de desarrollo tradicional).

Razones de uso del Prototipeo Evolutivo

En casos de estudio, el Prototipeo Evolutivo ha disminuido el esfuerzo del desarrollo dramáticamente, desde un 45 a un 80 por ciento. Para alcanzar esta clase de reducción, debemos manejar los riesgos de desarrollo cuidadosamente, particularmente los riesgos de diseño pobre, mantenimiento pobre, y adición de características. Si no hacemos esto, el esfuerzo global puede actualmente incrementarse.

El prototipeo evolutivo es un modelo ciclo de vida en el cual desarrollamos el concepto del sistema según nos movamos a través del proyecto, y este es especialmente útil cuando los requerimientos cambian rápidamente, cuando el usuario es adverso a establecer un juego de requerimientos, o cuando ni nosotros ni el cliente comprende bien el área de la aplicación. Este es también útil cuando los desarrolladores no están seguros de la arquitectura o algoritmos óptimos a usar.

El Prototipeo Evolutivo produce visibles signos de progreso uniformes, los cuales pueden ser especialmente útiles cuando hay una demanda fuerte por velocidad de desarrollo.

Claves para tener éxito en el uso del Prototipeo Evolutivo

Las siguientes son algunas claves para el éxito en el Prototipeo Evolutivo:

- Explícitamente manejemos las expectaciones del cliente y usuario final, relacionadas al cronograma y performance

- Limitemos la interacción del usuario final con el prototipo para controlar las características
- Usemos desarrolladores experimentados. No usemos practicantes
- Enfoquemos el diseño en cada etapa para asegurar la calidad del sistema prototipeado.
- Consideremos las cuestiones de performance temprano en el desarrollo
- Manejemos cuidadosamente la actividad misma del prototipeo.
- Consideremos si es que el Prototipeo Evolutivo proveerá el mayor beneficio o si es que la Entrega Evolutiva, o la Entrega por Etapas serian mejor.

3.5 Modelo Entrega por Etapas

El modelo entrega por etapas es otro modelo ciclo de vida en el cual mostramos sucesivamente software al cliente en etapas refinadas. A diferencia del modelo prototipeo evolutivo, cuando usamos la entrega por etapas, conocemos exactamente que es lo que vamos a construir al momento de construcción. Lo que hace al modelo entrega por etapas distinguirse es que el software no es entregado al final del proyecto de un solo golpe. Los artefactos son entregados en sucesivas etapas a través del proyecto. (Modelo conocido también como "implantación incremental.")

La figura 3-8 muestra como trabaja el modelo.

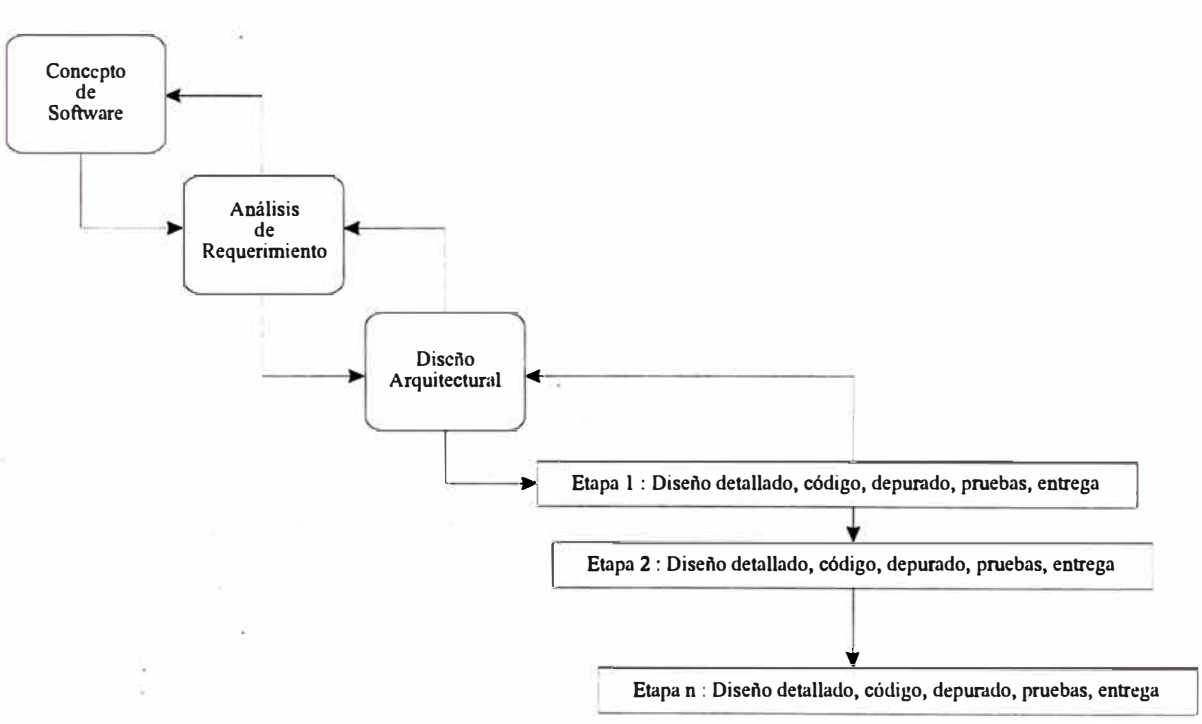


Figura 3-8. El modelo de entrega por etapas. Este modelo evita los problemas del modelo cascada, relacionados a que alguna parte del sistema no pueda hacerse sino hasta que todas las partes del mismo estén listas. Una vez que se ha finalizado el diseño se puede implementar y desplegar el sistema en etapas.

Según la figura 3-8 sugiere, con la entrega por etapas vamos a través de los pasos del modelo cascada para la definición del concepto de software, análisis de requerimientos, y la creación de un diseño arquitectural para el producto completo que se intenta construir. Luego procedemos al diseño detallado, codificación, depuración, y pruebas dentro de cada etapa.

La primera ventaja de la entrega por etapas es que nos permite poner funcionalidad útil en las manos de los clientes mas temprano que si entregáramos el 100 por ciento al final del proyecto. Si planeamos las etapas cuidadosamente, podemos estar hábiles para entregar la funcionalidad más importante lo mas temprano posible, y los usuarios pueden empezar a usar el software en ese momento.

La entrega por etapas también nos provee de señales tangibles de progreso temprano en el proyecto, lo que puede ser un aliado valioso en mantener la presión del cronograma a un nivel manejable.

La principal desventaja de la entrega por etapas es que este no trabaja sin un planeamiento cuidadoso al nivel gerencial y técnico. Al nivel gerencial, asegurémonos que las etapas planeadas son significativas para el cliente y que distribuimos el trabajo entre el personal del proyecto de una manera que ellos pueden completar su trabajo a tiempo para cumplir con los plazos fijados. En el nivel técnico, asegurémonos que hemos previsto todas las dependencias técnicas entre los diferentes componentes del producto. Un error común es diferir el desarrollo de un componente hasta la etapa 4 y después encontrar que un componente planeado para la etapa 2 no puede trabajar sin este.

3.6 Modelo Entrega Evolutiva

La entrega evolutiva es un modelo ciclo de vida que se apoya en el prototipeo evolutivo y la entrega por etapas. En la secuencia de pasos desarrollamos una versión del producto, mostramos este al cliente, y refinamos el producto basados en la retroalimentación del cliente. En cuanto la entrega evolutiva se asemeja al prototipeo evolutivo realmente depende en el punto al cual planeamos acomodar los requerimientos del usuario. Si planeamos acomodar la mayor parte de requerimientos, la entrega evolutiva se asemejara al prototipeo evolutivo. Si planeamos acomodar algunos cambios de requerimientos, la entrega evolutiva se asemejara mucho a la entrega por etapas. La figura 3-9 ilustra como trabaja el proceso.

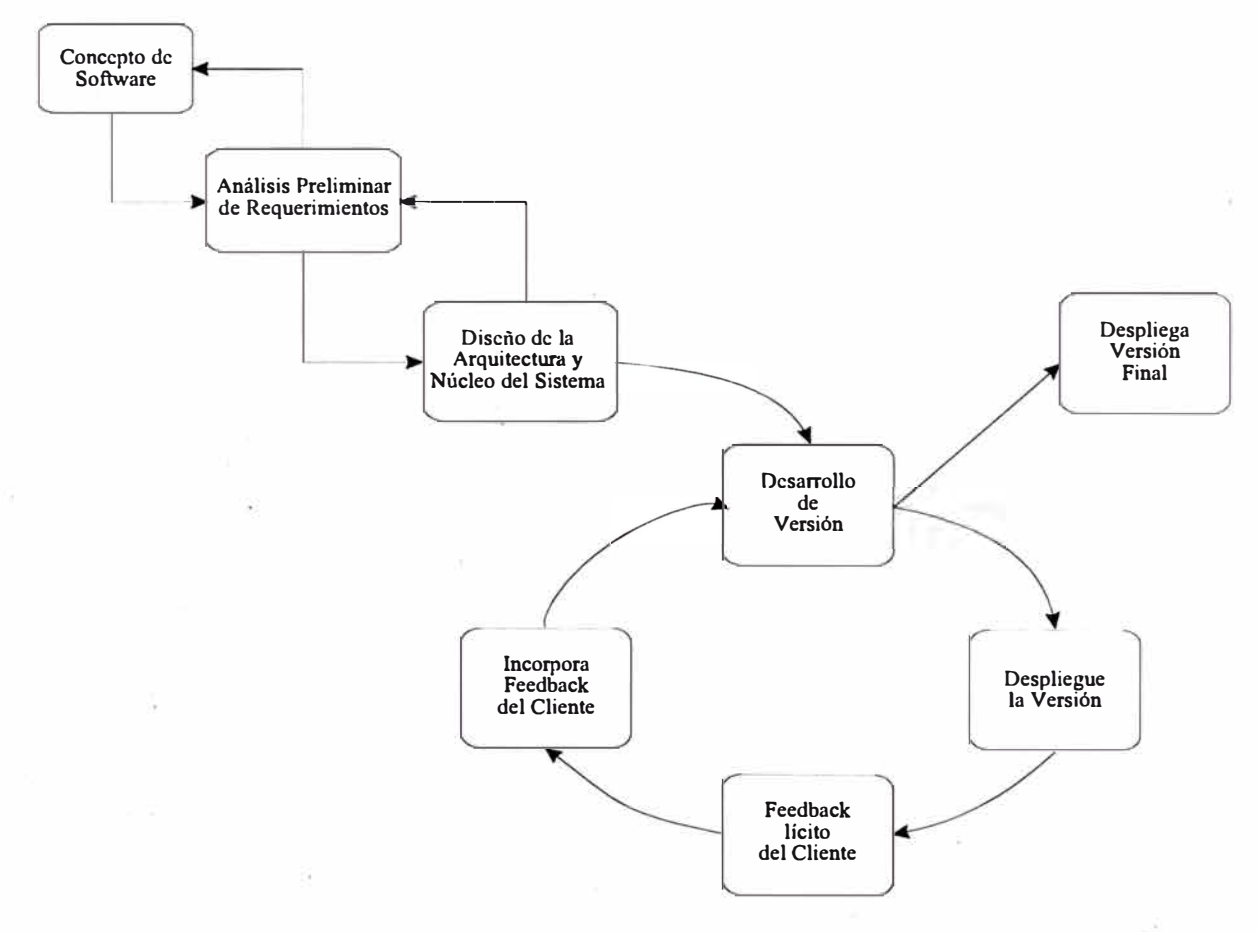


Figura 3-9. El modelo entrega evolutiva

Las principales diferencias entre el prototipeo evolutivo y la entrega por etapas son mas diferencias en énfasis que en el enfoque fundamental. En el prototipeo evolutivo, el enfoque inicial se da en los aspectos visibles del sistema; luego que estos son comprendidos mas regresamos y enfocamos en la fundación del mismo.

En la entrega evolutiva, el énfasis inicial es en el núcleo del sistema, el cual consiste de funciones del sistema del mas bajo nivel que son improbables de ser cambiadas por la retroalimentación del cliente.

3.7 Selección del Modelo Ciclo de Desarrollo

La elección errada del modelo ciclo de vida puede resultar en omisión de tareas e inapropiado orden de las mismas, lo cual merma el planeamiento del proyecto y su eficiencia. La elección de un modelo ciclo de vida apropiado tiene el efecto opuesto, asegurando que todo el esfuerzo sea usado eficientemente. Todo proyecto usa un ciclo de vida de una u otra clase explícitamente o implícitamente y la práctica de selección asegura que la elección es hecha explícitamente y es necesaria para obtener la máxima ventaja.

¿Cuál es el ciclo de vida más rápido?. No hay algo llamado "modelo del ciclo de vida de desarrollo rápido", debido a que el modelo más efectivo depende del contexto en el cual este es usado. Ciertos modelos de ciclo de vida son algunas veces tildados de ser más rápidos que otros, pero cada uno será más rápido en algunas situaciones, mas lento en otras. También un modelo de ciclo de vida que comúnmente trabaja bien, puede no dar los resultados esperados si no se aplica correctamente.

Para elegir el modelo de ciclo de vida más efectivo para el proyecto, examinemos el proyecto y contestemos las siguientes preguntas:

- ¿Qué también nuestros clientes y nosotros comprendemos los requerimientos al comienzo del proyecto? ¿Está nuestra comprensión propensa a cambiar significantemente a medida que avance el proyecto?
- ¿Cuán bien comprendemos la arquitectura del sistema? ¿Es probable que necesitemos hacer mayores cambios a medio camino en el proyecto?
- ¿Cuánto de confianza necesitamos?
- ¿Cuánto planeamiento y diseño adelantado necesitamos durante el proyecto para futuras versiones?
- ¿Cuánto es el riesgo envuelto en este proyecto?

- ¿Estamos sujetos a un cronograma establecido?
- ¿Estamos capacitados a hacer correcciones en medio camino?
- ¿Necesitamos mostrar a nuestros clientes progreso visible a través del desarrollo del proyecto?
- ¿Necesitamos mostrar a la gerencia progreso visible a través del proyecto?
- ¿Cuánta sofisticación necesitamos para usar un modelo de ciclo de vida exitosamente?

Después de contestar estas preguntas la tabla siguiente debería ayudarnos a decidir el modelo ciclo de vida a usar.

Cada valuación es "Pobre", "Regular", o "Excelente". La valuación está basada en el mejor potencial del modelo. La efectividad de cualquier modelo de ciclo de vida depende en como lo implementamos. Es posible que lo hagamos peor de lo que se indica en la tabla. De otro lado, si sabemos que el modelo es débil en un area en particular, podemos direccionar esa debilidad temprano en el planeamiento y compensar este, posiblemente creando un híbrido de uno o más de los modelos descritos. Por cierto, varios de los criterios en la tabla estarán fuertemente influenciados por las consideraciones de desarrollo y no únicamente por la elección del modelo de ciclo de vida.

A continuación se detallan las descripciones del modelo de ciclo de vida descrito en la tabla:

Trabaja con una pobre comprensión de requerimientos. Se refiere a que tan bien el modelo de ciclo de vida trabaja cuando el cliente o nosotros tenemos una comprensión pobre de los requerimientos del sistema, o cuando el cliente es propenso a cambiar los requerimientos. Esto indica que tan bien adecuado es el modelo al desarrollo de software exploratorio.

Caracteristicas del Ciclo de Vida	Cascada Puro	Espiral	Cascada Modificado	Prototipo Evolutivo	Entrega por Etapas	Entrega Evolutiva
Trabaja con pobre comprensión de requerimientos	Pobre	Excelente	Regular a Excelente	Excelente	Pobre	Regular a Excelente
Trabaja con pobre comprensión de arquitectura	Excelente	Excelente	Regular a Excelente	Regular a Excelente	Pobre	Pobre
Produce sistemas altamente confiables	Excelente	Excelente	Excelente	Regular	Excelente	Regular a Excelente
Produce sistemas con gran envoltura	Excelente	Excelente	Regular	Excelente	Excelente	Excelente
Gestion de riesgos	Pobre	Excelente	Regular	Regular	Regular	Regular
Restringido a un cronograma determinado	Regular	Regular	Regular	Pobre	Regular	Regular
Bajo overhead	Pobre	Regular	Excelente	Regular	Regular	Regular
Permite correccion a mitad del camino	Pobre	Regular	Regular	Excelente	Pobre	Regular a Excelente
Provee al cliente con visibilidad del progreso	Pobre	Excelente	Regular	Excelente	Regular	Excelente
Provee a la adm. con visibilidad del progreso	Regular	Excelente	Regular a Excelente	Regular	Excelente	Excelente
Requiere poca Supervisión o poca Sofisticación	Regular	Pobre	Pobre a Regular	Pobre	Regular	Pobre

Trabaja con pobre comprensión de la arquitectura. Se refiere a cuan bien el modelo del ciclo de vida trabaja cuando desarrollamos en una nueva area de aplicación, o cuando estamos desarrollando aplicaciones de una arrea familiar, pero estamos desarrollando características no familiares.

Produce sistemas altamente confiables. Se refiere al número de defectos que un sistema desarrollado con el modelo del ciclo de vida es probable que tenga, cuando lo pongamos en operación.

Produce sistemas con una gran envoltura. Se refiere a cuan fácil estaremos capacitados a modificar el sistema en tamaño y diversidad sobre su tiempo de vida. Esto incluye modificar el sistema en maneras que no fueron especificadas por el diseñador original.

Gestión de riesgo. Se refiere al soporte del modelo para identificar y controlar los riesgos del cronograma, riesgos del producto, y otros riesgos.

Puede ser restringido a un cronograma definitivo. Se refiere a cuan bien el modelo del ciclo de vida soporta la entrega del software en una fecha no renovable.

Tiene bajo gastos generales. Se refiere al monto de administración y overhead técnico para usar efectivamente el modelo. El overhead incluye planeamiento, seguimiento del estatus, producción de documentos, adquisición de paquetes, y otras actividades que no están directamente envueltas en la producción del software.

Permite correcciones a medio camino. Se refiere a la habilidad de cambiar significantemente los aspectos de un producto en el curso del desarrollo. Esto no incluye cambiar la misión básica del producto sino que incluye extender significantemente el mismo.

Permite al cliente tener visibilidad del progreso. Se refiere al punto hasta el cual el modelo automáticamente genera signos de progreso que el cliente puede usar para seguir el estatus del proyecto.

Permite a la administración tener visibilidad del progreso. Se refiere al punto hasta el cual el modelo automáticamente genera signos de progreso que la administración puede usar para seguir el estatus del proyecto.

Requiere poca gestión o desarrolladores no muy sofisticados. Se refiere al nivel de educación y entrenamiento que se necesita para usar el modelo efectivamente. Esto incluye el nivel de sofisticación que se necesita para el seguimiento del progreso usando el modelo, para evitar riesgos inherentes al modelo, para evitar desperdicio de tiempo usando el modelo, y para comprender los beneficios que nos guían a usar el modelo desde el primer momento.

CAPITULO 4

DESARROLLO ORIENTADO AL CLIENTE

Aparentemente se piensa que la orientación al cliente es demasiado intangible como para tener mucha influencia en la velocidad de desarrollo. Intangible o no, las compañías que han dado a sus relaciones con los clientes una alta prioridad, han hecho desaparecer muchos de sus problemas, incluyendo el problema del desarrollo lento.

La necesidad por prestar atención a los clientes se hace obvia cuando comprendemos eso, al final lo que importa es la percepción lenta o rápida que tiene el cliente en cuanto a la velocidad de desarrollo. Si a los clientes no les gusta el producto, ellos no pagaran por este, y si ellos no pagan, ninguna otra cosa que hagamos importa. Gerentes, distribuidores y ejecutivos están interesados en la velocidad del desarrollo debido a que ellos piensan que a los clientes también les importa. Si podemos satisfacer a nuestros clientes, podemos satisfacer a nuestros gerentes, y a todos los demás.

Quien es el "cliente" varía considerablemente de proyecto en proyecto. En algunos proyectos, el cliente es la persona que paga $200.00 por un producto de software shrink-wrap. En otros proyectos, es una organización la que paga directamente los costos de desarrollo del proyecto. Aún en otros proyectos es un grupo o departamento dentro de la organización. En todos los casos, los principios de incremento en la velocidad de desarrollo mejorando las relaciones con el cliente son casi las mismas.

4.1 Importancia de los Clientes al Desarrollo Rápido

Es bien sabido en la industria del desarrollo de software que una de las razones importantes en el éxito de los proyectos de software es el envolvimiento del cliente en los mismos.

Algunos expertos en el desarrollo rápido afirman que el fácil acceso a los usuarios finales es uno de los tres factores críticos de éxito en los proyectos de desarrollo rápido.

En estudios conducidos por investigadores se ha concluído que las tres razones por las cuales los proyectos no se completan en la fecha establecida, los presupuestos crecen escandalosamente, y el producto final es desplegado con menos funcionalidad que la deseada; es debida a la falta de feedback del usuario, a las especificaciones de requerimientos incompletos, y al cambio de requerimientos y especificaciones. Todos estos problemas los podemos manejar a través de prácticas orientadas al cliente.

A continuación se indican dos razones principales por las cuales deberíamos prestar atención a las relaciones con los clientes en un proyecto de desarrollo rápido:

- La buena relación con los clientes mejora la velocidad actual de desarrollo. Si se tiene una relación cooperativa antes que antagónica, y una buena comunicación con el cliente, se elimina una fuente significante de ineficiencia y muchos de los errores en el desarrollo.
- Las buenas relaciones con el cliente mejoran la percepción que se tiene de la velocidad de desarrollo. Muchas de las preocupaciones de los clientes referidas a la velocidad de desarrollo surgen del temor de que ni siquiera podamos completar el proyecto. Si estructuramos el proyecto para proveer alta visibilidad del progreso, aumentaremos la confianza del cliente en

nosotros, y la velocidad por si misma se convierte en una preocupación de menor importancia. La atención del cliente se direccionará hacia la funcionalidad, calidad y otros asuntos, y la velocidad de desarrollo será solo una mas de otras muchas prioridades.

Las siguientes secciones describen como enfocando en los clientes mejora la velocidad de desarrollo, tanto la real como la percibida.

4.1.1 Impacto en la Eficiencia

La participación del cliente esta ligada al camino critico en un proyecto de software. Los clientes comúnmente no comprenden lo que ellos necesitan hacer para dar soporte al desarrollo rápido. Ellos no preveen tiempo para revisiones, gestión, monitoreo del progreso, o aún considerar lo que están pidiendo. Los clientes no ven que una semana de retraso en la revisión de un documento clave pueda trasladarse como una semana de retraso en el despliegue del producto. Un problema común también es que los clientes proveen varios puntos de contacto, y muchas veces no sabemos a quien deberíamos contactar para conseguir una aprobación de un asunto en particular. Enfocando en las relaciones con el cliente temprano en el proyecto, podemos seleccionar prácticas orientadas al cliente que eliminen estas ineficiencias.

4.1.2 Disminución de Duplicidad de Esfuerzos

Uno de los más costosos errores en el desarrollo de software, es desarrollar software que al final es rechazado por el cliente. No podemos alcanzar desarrollo rápido si tenemos que desarrollar el software mas de una vez.

Usualmente los clientes no rechazan sistemas completos de software, sino solo partes, lo que significa que debemos rediseñar y reimplantar esas partes. El efecto global es que el sistema es desplegado tarde. Evitando estos desperdicios de esfuerzos es una clave para alcanzar el desarrollo rápido

4.2 Reducción del riesgo

A continuación se señalan algunas maneras por las cuales los clientes adicionan riesgo al cronograma:

- Los clientes no comprenden lo que ellos quieren
- Los clientes no proporcionan por escrito el conjunto de requerimientos
- Los clientes insisten en nuevos requerimientos después que los costos y el cronograma han sido fijados
- La comunicación con el cliente es lenta
- Los clientes no participan en revisiones o son incapaces de hacerlo
- Los clientes no son técnicamente sofisticados
- Los clientes no dejan que la gente haga sus tareas
- Los clientes no comprenden el proceso de desarrollo de software
- Un cliente nuevo es una nueva entidad, y los riesgos específicos son desconocidos

Estableciendo buenas relaciones con los clientes nos permite hacer un mejor trabajo de identificación de riesgos y monitorearlos a través del proyecto.

4.3 Naturaleza de la Fricción

Cuando no avanzamos de la mano con los clientes, dedicamos mas tiempo a manejar nuestras relaciones con el cliente. Eso toma tiempo, y puede distraernos. Mientras estamos pensando acerca de la arquitectura del software, en el fondo de nuestra mente también estamos pensando como decirle al cliente que el software estará retrasado un par de semanas. Esas distracciones nos hacen menos eficientes y son desmotivadoras. Es difícil trabajar horas extras para clientes que no nos simpatizan.

El problema de fricción con el cliente es endémico para la industria. Para proyectos outsourcing de software (proyectos con clientes reales), la severidad de la fricción entre los clientes y los contratistas de software es de tal manera que en promedio ambas partes tienen igual posibilidad de cancelar el proyecto.

La fricción puede surgir del lado del desarrollador o del lado del cliente. Del lado del cliente las fuentes de fricción para los desarrolladores incluyen la demanda de fechas de despliegue imposibles, demanda de nuevos requerimientos y negarse a pagar por ellos, omitir criterios de una clara aceptación del contrato, inadecuado monitoreo del progreso del contracto, insistencia en que sea quitada toda falla trivial en el primer despliegue.

Del lado del desarrollador, las fuentes de fricción para los clientes pueden incluir prometer fechas de despliegue imposibles, ofertar presupuestos bajos, falta de conocimientos necesarios para el proyecto, desarrollo de productos de baja calidad, no respetar las fechas convenidas, y proveer reportes de estatus inadecuados.

Hacernos compañeros de los clientes significa que ellos estén más propensos a comprender las restricciones técnicas. Empezaremos a desembarazarnos del

fenómeno "necesito todo esto ahora", y los clientes empezaran cooperando para encontrar soluciones técnicas realistas, mutuamente satisfactorias.

4.4 Prácticas Orientadas al Cliente

Las prácticas orientadas al cliente vienen en varias categorías.

- *Planeamiento*—Las prácticas orientadas al cliente nos ayudan a construir satisfacción para el usuario dentro del proyecto.
- *Análisis de requerimientos*—Las prácticas orientadas al cliente nos ayudan a comprender los requerimientos reales y a evitar el rehacer nuevamente lo hecho.
- *Diseño*—Las prácticas orientadas al cliente nos ayudan a construir la flexibilidad necesitada para responder rápidamente a las requisiciones de cambio generadas por el cliente.
- *Construcción*—Las prácticas orientadas al cliente ayudan a mantener al cliente confiado con nuestro progreso

4.4.1 Planeamiento

A continuación se abordan algunas prácticas que pueden ser usadas para construir la satisfacción del cliente dentro de nuestro proyecto:

- *Seleccionar un modelo ciclo de vida adecuado.* Proveamos a nuestro cliente con signos de progreso uniformes y tangibles. Las posibilidades incluyen la entrega evolutiva, el prototipeo evolutivo, y la entrega por etapas.
- *Identificar al cliente real.* Algunas veces la persona que necesitamos mantener feliz no es la persona con la cual tenemos contacto. Si

estamos construyendo software para otro grupo dentro de nuestra organización, podríamos dedicar mas de nuestro tiempo con la persona contacto de ese grupo, pero la persona que realmente debemos mantener feliz podría ser nuestro jefe. Si estamos trabajando con un cliente externo, el representante del cliente podría no ser el que toma las decisiones, quien decide si debe continuarse el proyecto o cancelarse. Asegurémonos de identificar al tomador de decisiones y también mantengámoslo feliz.

- *Establecer un método eficiente para interactuar con el cliente.* Si es posible, insistamos en que el cliente provea un único punto de contacto. Esa persona ocasionalmente necesitara conseguir algún feedback de otras personas o generar algún consenso en el lado del cliente, pero no hay un proyecto de desarrollo rápido en el cual tengamos que conseguir la aprobación de media docena de representantes del cliente por cada decisión.
- *Gestión del riesgo.* Pongamos especial atención a los riesgos relacionados al cliente, para poder direccionarlos de la manera adecuada.

4.4.2 Análisis de requerimientos

Cada vez que recopilamos los requerimientos, el reto es recopilar los requerimientos reales. Algunas veces los requerimientos reales están en conflicto con los requerimientos que recopilamos. Los requerimientos son comúnmente declarados vagamente, lo que crea posibilidades de confusión. Los clientes tienden a interpretar los requerimientos ampliamente, y los desarrolladores tienden a interpretar estos de una manera mas estrecha. Aquí tenemos otra fuente de fricción.

Las prácticas de recopilación de requerimientos orientadas al cliente nos ayudan a descubrir mas de los requerimientos reales y maximizar nuestra comprensión de todos los requerimientos. Obviamente, mientras más tiempo nos dediquemos trabajando en los requerimientos reales menos tiempo nos dedicaremos en requerimientos superfluos, y más rápido desplegaremos el software que el cliente quiere.

La figura 4-1 muestra las diferencias entre los requerimientos que recopilamos con y sin prácticas orientadas al cliente.

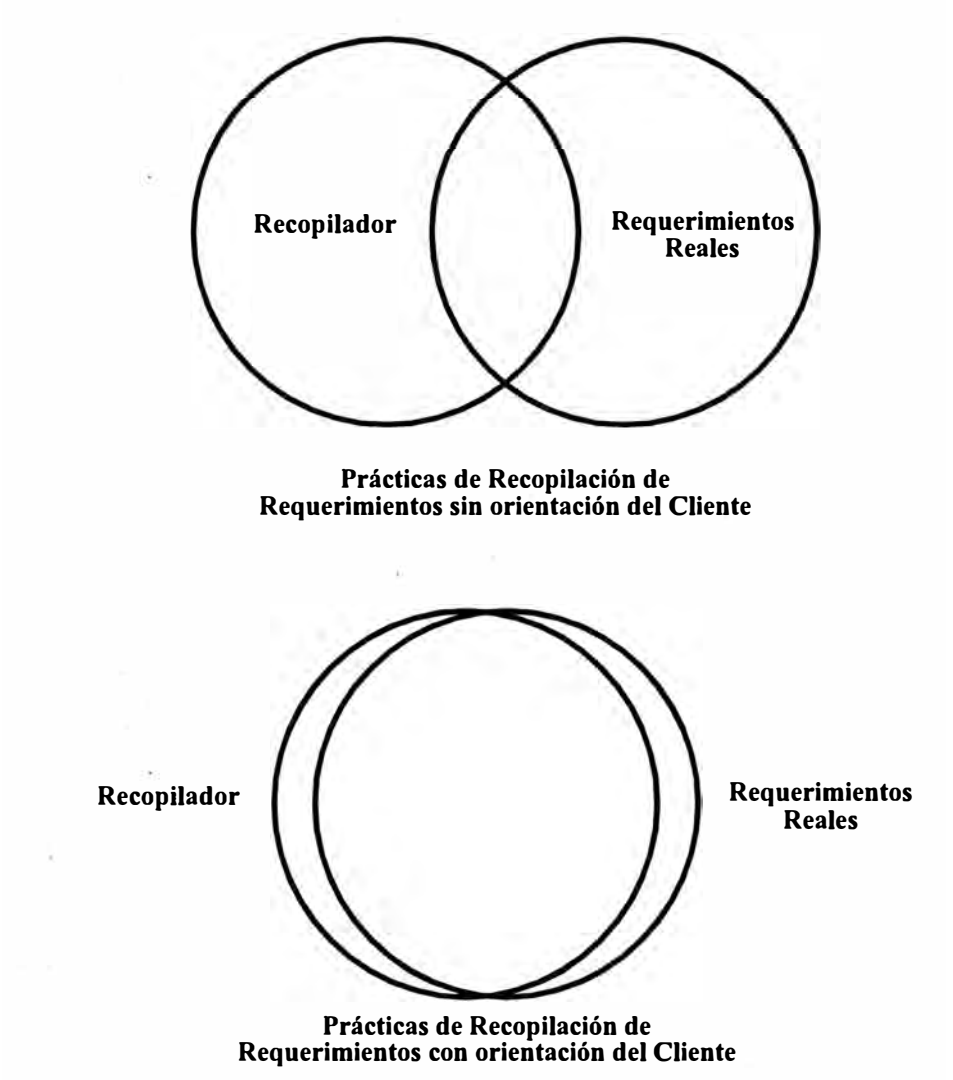


Figura 4-1. Incremento de la proporción de requerimientos reales que podemos recopilar

Estudios empíricos han mostrado que la productividad fue cerca de 50 por ciento más alta que el promedio cuando los clientes tuvieron un nivel "alto" de participación en la especificación de los requerimientos.

Tan importante como envolver a los clientes en la especificación de requerimientos, es también importante hacer un control de calidad de los mismos, evitemos que los clientes escriban las especificaciones de requerimientos completamente. Los mismos estudios que han encontrado que la productividad se incrementa con la alta participación del cliente, también han encontrado que la productividad fue mas baja cuando los clientes escribieron completamente las especificaciones. En efecto, mas de la mitad de las especificaciones escritas por los clientes tuvieron que ser reescritas.

A continuación se señalan algunas prácticas que podemos usar para envolver a los clientes en la recopilación de requerimientos:

- Usemos prácticas atractivas de requerimientos que ayuden al cliente a determinar estructuradamente lo que ellos quieren, como por ejemplo el prototipeo de la interface con los modelos de prototipeo evolutivo, entrega por etapas, o entrega evolutiva.
- Conduzcamos grupos de enfoque que ayuden a determinar lo que los clientes quieren.
- Grabemos en cintas de video a los clientes usando el software.
- Conduzcamos estudios de satisfacción del cliente, para obtener una medida cuantitativa de la relación con nuestros clientes.

4.4.3 Diseño

Podríamos haber hecho un trabajo perfecto en la recopilación de requerimientos, o tal vez no lo hemos hecho tan perfecto. De cualquier manera a continuación enuncio la práctica más productiva que podemos hacer durante el diseño para mantener la orientación al cliente:

- Empleemos prácticas de diseño que permitan a nuestros clientes cambiar ocasionalmente de parecer.

Esto se refiere a que identifiquemos los cambios que pensamos son probables con anterioridad, y luego hagamos uso de todo el encapsulamiento que sea posible para proveer al proyecto de la flexibilidad necesaria a ser usada en el desarrollo. Muchas veces la falta de flexibilidad es aparentemente una de las debilidades de los proyectos de software. Hay casos en los cuales acomodar un par de nuevos reportes conlleva a incurrir en traumas mayores en el sistema.

4.4.4 Construcción

Si nosotros hemos sentado bien las bases durante el planeamiento, el análisis de requerimientos, y el diseño, para el momento de conseguir la construcción a escala completa, nuestro cliente estará tan comprometido en el proceso de desarrollo que no tendremos que preocuparnos de este.

A continuación se enuncian algunas prácticas orientadas al cliente que trabajan bien especialmente durante la construcción:

- Empleemos prácticas de implantación que creen código de fácil lectura, modificable, lo cual mejorara nuestra habilidad para responder a los requerimientos de cambio de nuestros clientes.
- Usemos prácticas de monitoreo del progreso de tal manera que podamos informar al cliente de nuestros progresos.
- Seleccionemos un modelo ciclo de vida que provea al cliente con signos de progreso tangibles y continuos. Esto se convierte en especialmente importante al momento de implantación debido a que sin este el proyecto empieza a aparentar como si estuviera en una etapa mas de iteración sin percibir realmente que este se esta moviendo hacia adelante y que se encuentra ya en la etapa final de desarrollo.

Un aspecto interesante de elegir un modelo ciclo de vida incremental es que este nos permite frecuentemente entregar a nuestros clientes software trabajando. El mero acto de entregar un producto trabajando a nuestros clientes cada semana o cada mes, provee de comunicación más efectiva a nuestro progreso que un reporte tradicional de estatus. A los clientes les gusta mas el progreso que las promesas, y a ellos especialmente les gusta el progreso que pueden mantener en sus manos o ver en sus pantallas.

4.5 Gestión de las expectativas del cliente

Muchos problemas en el desarrollo de software, especialmente en el área de velocidad de desarrollo surgen de expectaciones irreales y no declaradas. Un estudio ha encontrado que el 10 por ciento de todos los proyectos son cancelados debido a expectativas irreales. Nuestro interés esta en tratar de establecer expectativas explícitamente de tal manera que podamos traer a claridad cualquier asunción irreal que los clientes podrían tener acerca del cronograma o de los entregables.

Una clase de expectativas irreales pueden surgir del cronograma. Muchos proyectos tienen sus cronogramas establecidos por los clientes antes que los requerimientos y los recursos sean completamente conocidos. Convenir en cronogramas irreales crea expectativas irreales en la mente de los clientes. Trabajar con el cliente para establecer expectativas reales acerca de los cronogramas es una clave para el éxito.

Esforzarnos por comprender las expectativas del cliente puede ahorrarnos fricción y trabajo extra.

Posiblemente mas de una vez hemos tenido experiencias en las cuales nuestros clientes erradamente percibieron algo tan básico, que ni siquiera se nos había ocurrido que esto fuera propenso a ser mal comprendido. Los clientes comúnmente piensan que si hemos acabado el prototipo, entonces hemos ya acabado con el producto mismo. Ellos ponen los diskettes dentro de los drivers patas para arriba, y al revés. Ellos piensan que la intensidad de los colores que ven en las pantallas deberían también ser impresos con sus impresoras láser. Ellos mueven el mouse al borde del escritorio y no saben que hacer. Ellos piensan que nosotros deberíamos conocer todos sus requerimientos sin tener que explicarnos los mismos.

Aún a pesar de lo que ellos a veces aparentan, ellos no son tontos; sino que no comprenden que está sucediendo en el desarrollo de software. Eso es justo debido a que algunas veces los desarrolladores no comprendemos el entorno del negocio de los clientes y ellos piensan que por eso somos bobos.

Una parte de nuestro trabajo como desarrolladores de software es educar a nuestros clientes de tal manera que ellos comprendan mejor el desarrollo de software, lo que hace posible para nosotros evaluar sus expectativas. Cuando los clientes se educan, cuando ellos tienen experiencia con software en general, con automatización en las áreas de su trabajo, y como participantes en proyectos de software es cuando la productividad del proyecto global mejora.

Algunas veces las expectativas del cliente hacen que el éxito del proyecto sea imposible de alcanzar. Así tenemos casos en los cuales si el proyecto termina antes del tiempo especificado, el cliente acusa al desarrollador de haber estimado el cronograma con demasiada holgura.

Si se termina a tiempo, el cliente acusa al desarrollador de haber estimado el cronograma conservadoramente y que el trabajo se ha retrasado solo para cumplir a tiempo. Si se termina tarde, el cliente acusa al desarrollador de incumplimiento y falta de responsabilidad.

La creación de expectativas irreales en el cliente acerca del cronograma, costos, o funcionalidad por cualquier razón para conseguir un proyecto aprobado, o fondos adecuados para un proyecto, o ser un ofertante de servicios a bajos montos, crea virtualmente inevitable riesgo para cualquier proyecto. La creación de expectativas infladas establece una situación en la cual el proyecto parecerá que esta en problemas aún cuando su desarrollo es normal. Con expectativas infladas, los desarrolladores son vistos como perdedores aún cuando ellos están haciendo un buen trabajo. La gente que infla las expectativas malogra la credibilidad de los desarrolladores, y estos sub estiman sus relaciones laborales con sus clientes. La gente que promete mas de lo real puede tener inicialmente éxito, pero ellos estarán en problema en el largo plazo. Por eso, parte del trabajo del desarrollador es establecer expectativas realísticas.

CAPITULO 5

REUSABILIDAD

La reusabilidad es una estrategia a largo plazo con la que la organización construye una librería de componentes frecuentemente usados, permitiendo que los nuevos programas sean ensamblados rápidamente a partir de componentes existentes. Cuando esta estrategia es respaldada en el largo plazo por un compromiso de la gerencia, la reusabilidad puede producir mejores cronogramas y ahorro de esfuerzos que ninguna otra práctica de desarrollo rápido. Más aún ésta puede ser usada por virtualmente cualquier clase de organización y para cualquier clase de software. La reusabilidad puede también ser implementada oportunísticamente, como una práctica de corto plazo, usando código para un programa nuevo a partir de programas existentes. El enfoque de corto plazo puede también producir significantes ahorros de esfuerzos y de cronograma, pero los ahorros potenciales son de lejos más dramáticos con la reusabilidad planeada.

Algunas veces se piensa que la reusabilidad puede solo ser aplicada al código, pero esta envuelve cualquiera de los esfuerzos realizados en desarrollos previos como código, diseños, datos, documentación, materiales de pruebas, especificaciones y planes. Para aplicaciones de sistemas de información la planificación para el reuso de datos puede ser tan importante como el planeamiento para el reuso de código.

En este estudio se enfoca dos categorías básicas de reuso:

- Reuso planeado
- Reuso oportunista

La reusabilidad planeada es acerca de lo que expertos usualmente hablan cuando ellos se refieren a la reusabilidad, pero la reusabilidad oportunística puede proveer también reducciones del cronograma. La reusabilidad oportunística se compone de dos tópicos:

- Reuso de componentes de fuentes internas
- Reuso de componentes de fuentes externas

El uso de componentes externos no es contemplado típicamente como reuso; sino que es típicamente contemplado como una decisión de comprar vs. desarrollar. Pero todo lo relacionado es similar por eso abordaremos algunos aspectos de este tópico.

El reuso produce ahorro en el cronograma por la razón obvia de que es comúnmente más rápido, más fácil, y más confiable de reusar algo que ya ha sido construido y probado que crear algo nuevo. Podemos emplear el reuso en proyectos de virtualmente cualquier tamaño, y este es apropiado para desarrollo interno de sistemas comerciales, software de sistemas, y para distribución masiva de aplicaciones shrink-wrap.

Las consideraciones envueltas en el reuso oportunista y el reuso planeado son completamente diferentes y son abordadas a continuación.

5.1 Reusabilidad oportunista

Podemos reusar oportunísticamente cuando descubrimos que un sistema existente tiene algo en común con un sistema que estamos a punto de construir. Luego

podemos ahorrar esfuerzos seleccionando piezas del sistema existente y usando estas en el sistema nuevo.

5.1.1 ¿Adaptar o Salvar?

Si queremos reusar oportunísticamente, tenemos dos opciones: adaptar el sistema viejo al uso del nuevo, o diseñar el nuevo sistema desde el comienzo y salvar componentes del viejo sistema. En muchos casos se ha dado que lo mejor es crear un diseño fresco para el sistema nuevo y luego salvar partes del sistema existente. La razón de por que este enfoque trabaja mejor es que este requiere que comprendamos solo pequeñas piezas aisladas del programa existente. La adaptación de un programa existente a uno nuevo requiere que comprendamos en su totalidad el programa viejo, lo cual es un trabajo más difícil. Por supuesto que si la gente que desarrollo el sistema existente son los mismos que están trabajando en el nuevo sistema, o si el nuevo sistema es extremadamente similar al ya existente, sería mucho mejor adaptar el viejo sistema.

El enfoque de salvamento es oportunista debido a que es un hecho de suerte el tener la oportunidad de reusar los diseños y código de un sistema previo sin planear esto con anticipación. Podemos tener éxito si el sistema existente ha sido bien diseñado e implementado. Es extremadamente útil si el sistema previo ha hecho uso de modularidad y encapsulamiento. Podemos también tener éxito si el staff se traslapa entre el sistema previo y el nuevo. Tratar de salvar partes de un sistema previo sin tomar en cuenta el traslape, puede ser mas un ejercicio de criptografía que de reuso.

5.1.2 Ahorros sobre estimados

El problema más grande con el reuso oportunista es que comúnmente se sobre estima con facilidad el esfuerzo potencial y los ahorros en el cronograma. Aún si el sistema viejo es completamente similar al nuevo, digamos que un 80 por ciento del código podría potencialmente ser rehusado, tenemos que considerar el análisis de requerimientos, diseño, construcción, pruebas, documentación, y otras actividades en nuestro esfuerzo y en el planeamiento del cronograma.

Dependiendo de la situación especifica, podríamos estar aptos para reusar todo el análisis de requerimientos y diseño o ninguno de estos; si el proyecto previo no fue creado y empaquetado para ser rehusado, no deberíamos contar con reusar este. Si el código es lo único que podemos reusar, ese 80 por ciento de traslape en código pude encogerse a una reducción de 20 por ciento en esfuerzo y posiblemente menos reducción en el cronograma, aún cuando los sistemas son 80 por ciento similares.

Parte del trabajo de reusar el 80 por ciento del código será el comprender que 80 por ciento se reusará. Eso puede ser engañosamente un desperdicio de tiempo y puede consumir el 20 por ciento de ahorros.

Otro problema común ocurre cuando las partes del sistema previo son usadas de maneras que no fueron anticipadas en el diseño e implantación original, lo que podría llevarnos a encontrar defectos en el código del viejo sistema. Cuando esto pase, si el staff no es íntimamente familiar con el sistema viejo, ellos repentinamente se encontraran depurando y adecuando código no familiar, y eso consumirá también el 20 por ciento de ahorros.

5.1.3 Experiencias con el reuso oportunista

Algunos proyectos que han reusado oportunísticamente han tenido gran éxito. La NASA ha reportado que en algunos proyectos se han producido hasta 37 por ciento de incremento en la productividad por el hecho de salvar código de un sistema similar existente. Los lideres de proyecto acreditaron el éxito al uso de modularidad, encapsulación en el primer sistema, alcances similares del viejo y nuevo sistema, así como la mitad del staff trabajando en ambos sistemas.

Un estudio en el Laboratorio de Ingeniería de Software en la NASA estudió diez proyectos que perseguían un reuso agresivo. Los proyectos iniciales no estuvieron aptos para usar mucho del código de los proyectos previos debido a que esos proyectos previos no habían establecido una base de código suficiente. En proyectos subsecuentes, sin embargo, los proyectos que usaron diseño funcional estuvieron aptos para tomar cerca del 35 por ciento de su código de proyectos previos. Los proyectos que usaron el diseño orientado al objeto estuvieron aptos para tomar alrededor del 70 por ciento de su código de proyectos previos.

Una de las ventajas de esta clase de reuso es que este no necesariamente requiere un alto nivel de compromiso por parte de las gerencias. Hay reportes que indican que los desarrolladores individuales comúnmente rehusan su propio código el cual puede alcanzar hasta 35 por ciento de un programa dado. Nosotros podríamos estar capacitados para alentar a los desarrolladores a reusar tanto como sea posible al nivel de la dirección técnica o al nivel de contribución individual.

5.1.4 Reuso Externo

No hay una razón técnica para desarrollar internamente un componente cuando podemos comprarlo empaquetado externamente, pero podría haber una razón comercial, tal como querer controlar una tecnología que es critica a nuestro negocio. Desde un punto de vista técnico, un proveedor externo puede probablemente poner mas recursos dentro de un componente empaquetado que lo que nosotros podríamos y si lo construiríamos este se haría a un mayor costo. Los módulos de código reusables son típicamente ofrecidos desde diez a veinte por ciento de lo que ellos costarían si los desarrolláramos.

Para mucha gente, en la industria no hay otra manera de pensar mas que en el reuso. Este ha sido el sujeto de investigaciones académicas e industriales por años, y los proveedores por largo tiempo han proveído comercialmente librerías de código para dominios de aplicaciones especificas tales como interfaces, bases de datos, aplicaciones científicas, etc. Pero el reuso comercial finalmente empezó de una manera seria en la plataforma PC con Visual Basic de Microsoft y VBXs (Visual Basic Control). Reconociendo su popularidad, Microsoft dedico soporte para el reuso con OCXs(OLE Controls). Proveedores de lenguajes tales como Borland, Gupta, Microsoft, and PowerSoft han convenido en soportar ambos VBXs y OCXs en los productos de sus lenguajes.

Los problemas tradicionalmente citados como obstáculos para el reuso están también siendo direccionados por el mercado. Por ejemplo hay compañías que no necesitan crear sus propios depósitos debido a que los vendedores de VBXs los confeccionan por ellos. El éxito de los vendedores depende parcialmente de cuan bien ellos organizan, catalogan y publican sus

productos reusables de tal manera que los clientes puedan encontrar los componentes que necesitan para después comprarlos.

Algunas personas han estado desconcertadas por el éxito de los VBXs, y la razón es aparente. Los VBXs proveen encapsulación e información oculta extremadamente buena, las cuales son las claves de un reuso exitoso. El desarrollador de un VBXs no tiene otra elección si no que proveer una hoja de propiedades, la cual esencialmente documente la interface del componente VBX. La hoja de propiedades oculta completamente la implementación del componente, lo que fuerza a que se dé encapsulación e información oculta. VBX ha tenido éxito simplemente debido a que estos proveen mejor soporte para encapsulación que incluso muchos lenguajes orientados al objeto. La modularidad y el encapsulamiento proveen el soporte para el reuso que la gente quiere. Este hallazgo es consistente con los reportes de organizaciones que han establecido sus propios programas de reuso planeado.

5.2 Reusabilidad Planeada

El reuso planeado como su nombre implica es una estrategia de largo plazo que no nos ayudara en el primer proyecto en el cual lo usemos. A pesar de esto, ninguna otra práctica tiene la capacidad de reducir tanto los cronogramas de desarrollo en el largo plazo.

Para empezar a reusar programas, primero debemos estudiar el software existente en nuestra organización e identificar los componentes que ocurren frecuentemente. Luego deberíamos planear hacer estos componentes reusables en el largo plazo, comprando estos de vendedores confiables o desarrollando versiones reusables nosotros mismos.

5.2.1 Consideraciones de Gestión

Reusemos proyectos de enlace que previamente pudieron ser conducidos aisladamente, lo que amplia el alcance de las decisiones acerca de los proyectos. Esto implica que los diferentes proyectos tendrán que estandarizar sus procesos de software, lenguajes, y herramientas. El reuso efectivo requiere una inversión de largo tiempo en entrenamiento y un plan de proyectos múltiples para construir y mantener componentes reusables. Estos son retos difíciles, y virtualmente todo estudio en el reuso de programas o reportes en programas específicos indica que la clave para el éxito es el compromiso de la gerencia antes que la destreza técnica.

A continuación señalo algunas de las tareas de la gerencia en la conducción de un programa de reuso:

- Designemos un gerente de entre los gerentes de alto nivel para el programa de reuso.
- Aseguremos un compromiso a largo plazo para el programa. Es importante que el reuso no sea tratado como una obsesión, desde que este puede tomar 2 o más años para que el desarrollo de componentes reusables empiece a pagar.
- Hagamos del reuso una parte explícita e integral del proceso de desarrollo.

  El reuso no se dará por sí mismo, y este no sucederá como consecuencia de buenos procesos. Para hacer clara su prioridad, hagamos el soporte de reuso una actividad al nivel de revisión de la performance del desarrollador.
- Convirtamos los programas de medida de productividad de software de medir cuanto software es desarrollado a cuanto software es

desplegado. Esto ayudará a asegurar que los desarrolladores reciben créditos por usar componentes reusables.

- Establezcamos un grupo separado de reuso que se responsabilice del cuidado y generación de componentes reusables. En una pequeña organización el "grupo" podría ser una única persona.
- Proveamos entrenamiento para el grupo de reuso y para los clientes potenciales del grupo.
- Levantemos la conciencia de la organización en la iniciativa del reuso a través de una campaña activa de relaciones publicas.
- Crear y mantener una lista formal del personal que esta rehusando componentes, de tal manera que ellos puedan ser notificados en el evento de que haya problemas, con los componentes que ellos están usando.

5.2.2 Consideraciones Técnicas

El grupo a cargo del cuidado y de la alimentación de componentes reusables tendrá bastante trabajo que hacer. A continuación algunas de sus tareas:

- Evaluar si es que las arquitecturas de software en uso de la organización soportan reuso, y desarrollar arquitecturas que soporten reuso si es necesario.
- Evaluar si es que los estándares de codificación soportan reuso, y recomendar nuevos estándares si apropiado.
- Crear estándares para los lenguajes de programación y las interfaces que soporten reuso. Es casi imposible reusar componentes si ellos son escritos en diferentes lenguajes, si se usa completamente diferentes interfaces para llamar funciones, y si se usa diferentes convenciones de codificación.

- Establecer procesos que soporten reuso. Recomendar que cada proyecto empiece averiguando los componentes que este podría reusar.
- Crear una librería formal de componentes reusables, y proveer algunos medios de barrer la librería para buscar los componentes que podrían ser rehusados.

Mas allá de estas tareas generales, el grupo necesitara trabajar en el diseño, implantación, y calidad de los componentes reusables.

Enfocar en componentes de dominio especifico. Los programas de reuso que son los más exitosos enfocan la construcción de componentes reusables para dominios de aplicaciones especificas. Si estamos trabajando en aplicaciones financieras, enfoquemos la construcción de componentes reusables financieros. En programas de presupuestación de seguros enfoquemos en la construcción de componentes reusables de seguros. Tratemos de enfocar en el reuso al nivel del "dominio de la aplicación" o "componente del negocio".

Crear componentes pequeños y nítidos. Tengamos en cuenta la creación de componentes pequeños, nítidos y especializados en vez de componentes grandes, cargados y generales. Los desarrolladores que tratan de crear software reusable creando componentes generales raramente anticipan adecuadamente las necesidades de los usuarios. Los usuarios futuros buscaran en los componentes grandes y voluminosos; y verán que estos no satisfacen todas sus necesidades y decidirán no usar los componentes. "Grande y voluminoso" significa "demasiado difícil de comprender", y eso significa "demasiado propenso a errores para ser usado". Datos del Laboratorio de Ingeniería de Software de la NASA sugieren que si un

componente necesita ser modificado en mas de 25 por ciento, es tan costoso como desarrollar un componente nuevo a reusar el viejo.

Si se incurrió en la construcción de componentes grandes, es mejor dividirlos en varios componentes pequeños y nítidos y enfocar los esfuerzos de reuso en estos. En lenguaje de diseño estructurado, *factoricemos completamente* cualquier componente que se intenta usar. No presentemos componentes grandes como entidades únicas y monolíticas.

Enfocar en encapsulación e información oculta. Otra clave para el éxito es enfocar en encapsulación e información oculta, el núcleo del diseño orientado al objeto. El diseño orientado al objeto adiciona las ideas de herencia y polimorfismo al diseño basado en objetos lo que hace la reusabilidad más alcanzable.

Crear buena Documentación. El reuso también requiere buena documentación. Cuando creamos un componente reusable, no estamos solo creando un programa, sino que estamos creando un producto. Para que este sea usado el nivel que necesita tiene que ser comparable a los productos que nuestra organización compra comercialmente. Por eso hay quienes estiman que el desarrollo de un componente reusable costara cerca de tres veces el costo de desarrollarlo sin reuso en mente.

Una clase de documentación que es especialmente valiosa para el reuso es una lista de limitaciones conocidas para cada componente. El software es comúnmente desplegado con defectos conocidos debido a que algunas fallas son de una prioridad demasiada baja para ser depuradas. Pero un defecto que es de prioridad baja en un contexto puede convertirse de prioridad alta cuando es rehusado en otro producto. No hagamos que la gente descubra los

defectos por ellos mismos cuando ya sabemos de ellos; documentemos las limitaciones conocidas.

Construir los componentes reusables a prueba de errores. El reuso exitoso requiere la creación de componentes que sean virtualmente a prueba de errores. Si un desarrollador tratando de emplear un componente reusable encuentra que este contiene defectos, el programa de reuso rápidamente perderá su brillo. Un programa de reuso basado en programas de baja calidad puede incrementar el costo del desarrollo de software.

Enfocar en la calidad de la librería de los componentes reusables y no en el tamaño. El tamaño completo de código disponible a ser rehusado aparentemente no afecta el nivel de reuso; las librerías pequeñas de código son usadas al menos tanto como las grandes. Elegir desde un menú grande de existentes componentes reusables puede ser tan dificultoso que algunos desarrolladores se refieren a este fenómeno como escalar una montaña de código. Si se quiere implantar reusabilidad, enfoquemos en calidad y no en cantidad.

No tengamos en cuenta el hecho de si los desarrolladores aceptaran la reusabilidad. Finalmente hay mucho menos de que preocuparnos por el lado del usuario en la relación de reusabilidad, de lo que mucha gente piensa. La apreciación convencional es de que a los desarrolladores no les gusta usar el código al menos de que este haya sido desarrollado por ellos mismos. Pero la tendencia actual (por experiencias e investigaciones) es que alrededor del 70 por ciento de los desarrolladores actualmente preferirían reusar componentes que desarrollar desde el inicio.

5.3 Riesgos de la Reusabilidad

La reusabilidad generalmente mejora la calidad, productividad y baja los costos.

Al nivel del proyecto, la reusabilidad tiende a reducir el riesgo debido a que menos del producto necesita ser codificado, y la calidad de componentes rehusados es generalmente mas alta que si se desarrollara las partes correspondientes de manera convencional. Al nivel de la organización, sin embargo una familia de riesgos merodean alrededor de la dificultad de predecir cuales componentes necesitaran ser rehusados.

Desperdicio de Esfuerzo. La creación de componentes reusables cuestan dos a tres veces más de los que no lo son. Una vez que el componente es desarrollado, tenemos que usar este al menos dos o tres veces solo para recuperar los costos. De aquí que tengamos la "regla del tres": antes de recibir cualquier beneficio de la reusabilidad de un componente, tenemos que reusar este tres veces. Si no estamos seguros de reusar un componente al menos tres veces, todavía podría tener sentido tener disponible componentes preconstruídos desde un punto de vista de velocidad de desarrollos futuros, pero esa velocidad se obtendrá a un precio.

Esfuerzo mal dirigido. Si se establece un grupo separado para desarrollar componentes reusables, existe la posibilidad de que el grupo desarrolle componentes que nunca son usados. Si el grupo de reuso no esta en lo correcto y uno de cuatro de sus componentes no es nunca usado, tendríamos que planear usar cada componente que rehusamos al menos cuatro veces solo para alcanzar el punto de equilibrio. Dentro de una única organización que no desarrolla una notable cantidad de sistemas realmente similares, ese puede ser un punto de equilibrio difícil de alcanzar.

Una estrategia práctica que reduce el riesgo podría ser implementar una "regla de dos": implementar todo componente como si no fuera reusable la primera vez, y luego la segunda vez que se necesite, consideremos hacerlo reusable. En ese punto, sabemos que vamos a estar usando este al menos dos veces, así estamos implícitamente aceptando menos riesgo si pensamos que algún reuso ahora implica mas reuso posterior. Pero tengamos cuidado. Algunos expertos previenen que si aún no hemos construido tres sistemas reales en un dominio particular, probablemente no conozcamos aún lo suficiente de este como para crear componentes reusables.

Cambio de tecnología. Un riesgo de reuso planeado resulta del hecho que este es una estrategia de largo plazo. No solamente necesitamos usar un componente muchas veces para alcanzar el punto de equilibrio, si no que necesitamos usarlo antes de que la tecnología en la que este basado sea obsoleta. "No podemos dedicar demasiado tiempo haciendo algo compartible, solo para tenerlo como obsoleto dentro de 18 meses mas tarde. En un mundo donde las cosas cambian tan rápido y furiosamente, puntualicemos las partes que tienen que ser compartidas".

Ahorros Sobreestimados. No asumamos que la reusabilidad de código producirá un ahorro de gran cantidad de tiempo. Si usamos solo código, mayormente ahorraremos solo tiempo en la codificación. Si rehusamos otros artefactos del proyecto, podemos ahorrar también otro tiempo adicional.

Mantengamos presente que siempre hay un costo por reusar un componente debido al tiempo necesitado para encontrar ese componente y aprender como usarlo. Planeemos dedicar cierto tiempo necesario de acuerdo al caso para poder medir este tiempo y ponerlo en relación a lo que seria desarrollar desde el principio.

5.4 Efectos Colaterales de la Reusabilidad.

La reusabilidad oportunística no tiene efectos colaterales notables, pero los programas de reuso planeado tienen uno.

Mejora de performance. Los componentes rehusados mejoran la performance de dos maneras:

- Estos son comúnmente mejor diseñados y más funcionales que si no fueran componentes, lo que significa que son comúnmente más rápidos.
- Estos también permiten a los sistemas ser ensamblados más rápido, lo que permite a los sistemas ser probados por performance más rápido. Pero debido a que estos tienden a ser mas generales, los componentes reusables pueden algunas veces ser más lentos que los que no son componentes, así que necesitamos evaluar la performance de cualquier componente particular en el que estemos interesados en vez de asumir que este sea mas rápido o mas lento.

5.5 Beneficio de la Reusabilidad

El beneficio en el reuso oportunista varia considerablemente, dependiendo en cuan grande es el reuso oportunista. Pequeños montos de reuso oportunista producirán pequeños ahorros. Grandes montos de reuso podría probablemente producir ahorros de esfuerzo en el rango de 20 a 25 por ciento sobe la vida de un proyecto completo, asumiendo que montos grandes de código y diseño pueden ser rehusados y que hay alguna continuidad del staff entre el nuevo programa y el que esta siendo salvado.

El reuso planeado no es una práctica de corto plazo, pero en el largo plazo es cuando revierte y se convierte en una estrategia atractiva. En un estudio de mejora de procesos de software en 13 organizaciones, algunas organizaciones demostraron mucho mas ganancia en productividad que otras. Una compañía mejoró su productividad en 58 por ciento anual por cuatro años. Otra compañía redujo su tiempo de despliegue en un 23 por ciento anual por 6 años, eso es una reducción total de 79 por ciento. Los autores del estudio atribuyeron ambas ganancias extraordinarias a los programas de reusabilidad.

Debido a los retos organizacionales y a la calidad puede tomar hasta dos años para un programa de reuso desarrollar cualquier componente que sea verdaderamente reusable. Debido a que el uso de componentes reusables completamente elimina el diseño, la construcción y reduce el monto de pruebas necesitadas para esos componentes, un programa de reuso exitoso es de lejos la práctica de productividad disponible más efectiva.

5.6 Claves para el éxito de Reusabilidad

Las claves de éxito para el reuso oportunista son simples:

- Tomar ventaja de la continuidad del personal entre sistemas viejos y nuevos.
- No subestimar los ahorros.

Las claves de éxito para el reuso planeado demandan mas:

- Asegurar el compromiso de las gerencias de alto nivel en el largo plazo para el programa de reuso.
- Hacer del reuso una parte integral del proceso de desarrollo.

- Establecer un grupo separado responsable de identificar candidatos a componentes reusables, crear estándares que soporten reusabilidad, y diseminar información acerca de los componentes reusables a los usuarios potenciales.
- Enfocar en componentes pequeños, nítidos y de dominio especifico.
- Enfocar los esfuerzos de diseño en ocultamiento de información y encapsulamiento.
- Llevar la calidad de los componentes reusables al nivel de "productos" documentándolos bien y asegurándose de que ellos sean construidos virtualmente libre de errores.

CAPITULO 6

LENGUAJES DE DESARROLLO RÁPIDO (RDLs)

El término Lenguaje de Desarrollo rápido es un término general que se refiere a cualquier lenguaje de programación que ofrece una implantación más rápida que con lenguajes tradicionales tales como Basic, Cobol, Pascal, Fortran, C etc. A continuación señalo los tipos de entornos de desarrollo que caen bajo esta categoría:

- Lenguajes de cuarta generación (4GLs) tales como Focus y Ramis
- Sistemas de administración de bases de datos tales como Microsoft Access, Microsoft FoxPro, Oracle, Paradox y Sybase.
- Lenguajes de programación visual tales como Borland Delphi, Borland C++ Builder, CA Visual Objects, Microsoft Visual Basic, Power Builder, Realizer y Visual Age.
- Herramientas de dominio específico, tales como hojas de cálculo, paquetes estadísticos, editores de ecuaciones, y otras herramientas que resuelven problemas que de otra manera tendrían que ser solucionados creando un programa de computadora.

De las plataformas anteriormente señaladas las mas impactantes son aquellas de la nueva casta de programación visual, las cuales están basadas en la tecnología orientada al objeto.

6.1 Los Lenguajes de Programación Visual

En este ultimo tiempo de más velocidad y más interfaces visuales han aparecido una variada gama de lenguajes de programación visual (LPV).

La Programación Visual (PV) nos trajo el arte de la programación de aquel misticismo en el cual estaba el desarrollo de software, a algo más manipulable por cualquier mero mortal. Estas interfaces nuevas capacitan al desarrollador de software a construir "visualmente" la interface del usuario con el mouse, en vez de construir este con código, y después compilar y correr el programa para apreciar la apariencia y comportamiento del mismo.

Estas herramientas por ser visuales son más intuitivas, así es difícil mirar una pieza de código que genera una ventana y visualizar esa ventana, opuesto a la facilidad de crear una ventana con un par de clicks del mouse.

El énfasis no solo es puesto en las herramientas empleadas por el desarrollador sino también en la metodología que este usara, debido a que un lenguaje de desarrollo de aplicaciones rápidas nos lleva por defecto a una metodología de prototipos rápidos. Lo anterior se refiere a la práctica de crear modelos funcionales de la GUI (por sus siglas en ingles de Graphical User Interface) que los usuarios quieren para sus aplicaciones, al inicio de la etapa de diseño.

Dicha interface de construcción temprana es presentada al usuario para beneficiar el desarrollo del proyecto con retroalimentación desde el punto de vista del usuario, en cuanto al tamaño, los objetos, apariencia, la inclusión de características, algún maquillaje y sobre todo para darle al usuario un sentimiento de que estamos trabajando de acuerdo a sus necesidades. Sobre los pasados años, muchos productos han emergido como herramientas de prototipeo, pero estas eran solo pantallas sin ningún sustento de código. Las cosas han cambiado ahora con el

advenimiento de esta nueva generación RDL, así que aquellas herramientas de prototipeo no son mas necesarias. Las nuevas herramientas RDL incluyen un juego de Componentes Visuales que nos habilitan primero a crear las pantallas, e inmediatamente después a implementar todo lo relacionado al procesamiento de datos. Podemos también incluir OCXs, producidos por vendedores de software en nuestras ventanas. Muchos de estos pueden existir en la aplicación para propósitos de demostración con poco código, o sin este (solo tírelos en la ventana).

Con estas herramientas el desarrollador desarrolla aplicaciones rápidamente, lo cual es logrado gracias a las características que le permiten dar saltos considerables en la funcionalidad con una pequeña cantidad de trabajo. Por ejemplo si usamos el contenedor OLE para embeber una hoja de calculo Excel dentro de la aplicación, en vez de crear la funcionalidad de la hoja. La idea es crear buenas aplicaciones en el menor tiempo posible y con la menor cantidad de código.

En este punto, volviendo al prototipeo podemos asegurar que la desventaja de duplicidad de esfuerzo ha sido superada. En el pasado cuando creábamos los prototipos con las herramientas tradicionales, tan luego como se conseguía la aprobación de la interface por parte del cliente, teníamos que recodificarla para generar la aplicación real, lo cual era un desperdicio. En la actualidad eso ya no es necesario, por que al momento que se tira el componente en la ventana se crea el objeto.

Los lenguajes de Desarrollo Rápido producen sus ahorros reduciendo el monto de construcción necesitada para construir un producto. Aunque los ahorros son comprendidos durante la construcción, la habilidad de acortar el ciclo de construcción tiene implicaciones a lo largo del proyecto: ciclos de construcción más pequeños hacen prácticos los ciclos de vida incrementales tales como el Prototipeo Evolutivo.

Los lenguajes del desarrollo rápido soportan desarrollo rápido permitiendo a los desarrolladores, desarrollar programas a un nivel de abstracción mayor de lo que ellos lo harían con lenguajes tradicionales. Una operación que tomaría 100 líneas de código en C podría tomar solo 25 líneas de código en Visual Basic. Una operación que requeriría abrir un archivo, avanzar el puntero de archivo, escribir un registro y cerrar el archivo en C podría requerir solo una simple sentencia Store( ) en un RDL.

Los lenguajes de programación visual han sido posibles gracias al desarrollo de la tecnología orientada al objeto sobre la cual se basan, y es necesario conocer dicha tecnología para obtener ventajas en el desarrollo rápido de aplicaciones. En lo que sigue abordaremos el paradigma de los lenguajes tradicionales de programación, para luego tocar lo referente al paradigma de la orientación al objeto.

6.2 Los Lenguajes Procedimentales

Pascal, C, Cobol, Fortran, Basic, y lenguajes similares son lenguajes procedimentales. Lo que significa que cada declaración en el lenguaje le indica a la computadora hacer algo: Captar algunos datos (entrada), operar estos datos (proceso), mostrar los datos (salida). Un programa en un lenguaje procedimental es una lista de instrucciones.

Para programas pequeños no se necesita otro paradigma. El programador crea una lista de instrucciones, y la computadora las ejecuta.

6.2.1 División en funciones

Cuando los programas se van haciendo grandes, una simple lista de instrucciones se hace inmanejable. Pocos programadores pueden comprender un programa de mas de unas pocas centenas de declaraciones, a menos que

se divida en unidades más pequeñas. Por esta razón la función fue adoptada como una manera de hacer los programas más comprensibles a los creadores humanos. Un programa es dividido en funciones, e (idealmente, al menos) cada función tiene un propósito definido y una interface definida claramente, para las otras funciones del programa.

La idea de dividir un programa en funciones puede ser extendido agrupando un número de funciones dentro de una entidad mas larga llamada modulo, pero el principio es el mismo: un grupo de componentes que llevan a cabo una tarea especifica.

El dividir un programa en funciones y módulos es una de las piedras angulares de la programación estructurada.

6.2.2 Problemas con la programación estructurada

Según como los programas vayan creciendo y se conviertan en más complejos los riesgos en el desarrollo se convierten en amenazas. No es extraño haber escuchado que el proyecto es demasiado complejo, que no se puede cumplir el cronograma, que hay que adicionar mas programadores, que la complejidad se esta incrementando, los costos se elevan, y los desastres aparecen.

Analizando las razones para estas fallas encontramos que hay debilidades en el paradigma procedimental. No interesa cuan bien el enfoque de programación estructurada este implementado, los programas grandes se convierten en excesivamente complejos.

¿Cuáles son las razones de la falla de los lenguajes procedimentales? Uno de los más cruciales es el rol de los datos.

6.2.2.1 Datos Subvaluados

En un lenguaje procedimental el énfasis es en hacer cosas; leer el teclado, invertir una matriz, chequear errores, abrir archivos, etc. La subdivisión de un programa en funciones continúa este énfasis. Las funciones hacen cosas de la misma manera que una simple instrucción. Lo que ellas pueden hacer puede ser más complejo o abstracto, pero el énfasis esta aún en la acción.

¿Qué pasa con los datos en este paradigma? Los datos son después de todo la razón de la existencia de un programa. La parte importante de un programa de inventarios no es la función que muestra los datos en pantalla, o una función que valida el ingreso de datos; sino son los datos del inventario. Los datos tienen un segundo nivel de estatus en la organización de los lenguajes procedimentales.

Por ejemplo, en un programa de inventarios, los datos son probablemente leídos desde un disco a memoria, donde estos son tratados como variables globales. Por global quiero decir que las variables que constituyen los datos son declarados afuera de las funciones, de tal manera que estos están accesibles a todas las funciones. Estas funciones ejecutan varias operaciones en los datos. Estas leen los datos, analizan, actualizan, ordenan, muestran, los reescriben en el disco, etc.

Deberíamos notar que la mayoría de lenguajes, tales como Pascal y C, también soportan variables locales, las cuales están ocultas dentro de una función. Pero las variables locales no son útiles para almacenar datos importantes que deben ser accesados por muchas funciones diferentes. La figura 6-1 muestra la relación entre variables globales y locales.

Supongamos ahora que un programador nuevo es contratado para escribir una función que de alguna manera analice los datos del inventario. Debido a la falta de familiaridad con el programa, el programador crea una función que corrompe los datos accidentalmente. Esto es fácil de hacer debido a que todas las funciones tienen acceso completo a los datos. Es como dejar nuestros documentos personales en la sala de recepción de la empresa donde trabajamos: cualquiera puede confundirlos o destruirlos. De la misma manera los datos globales pueden ser corrompidos por las funciones que no tienen que hacer nada con estos datos.

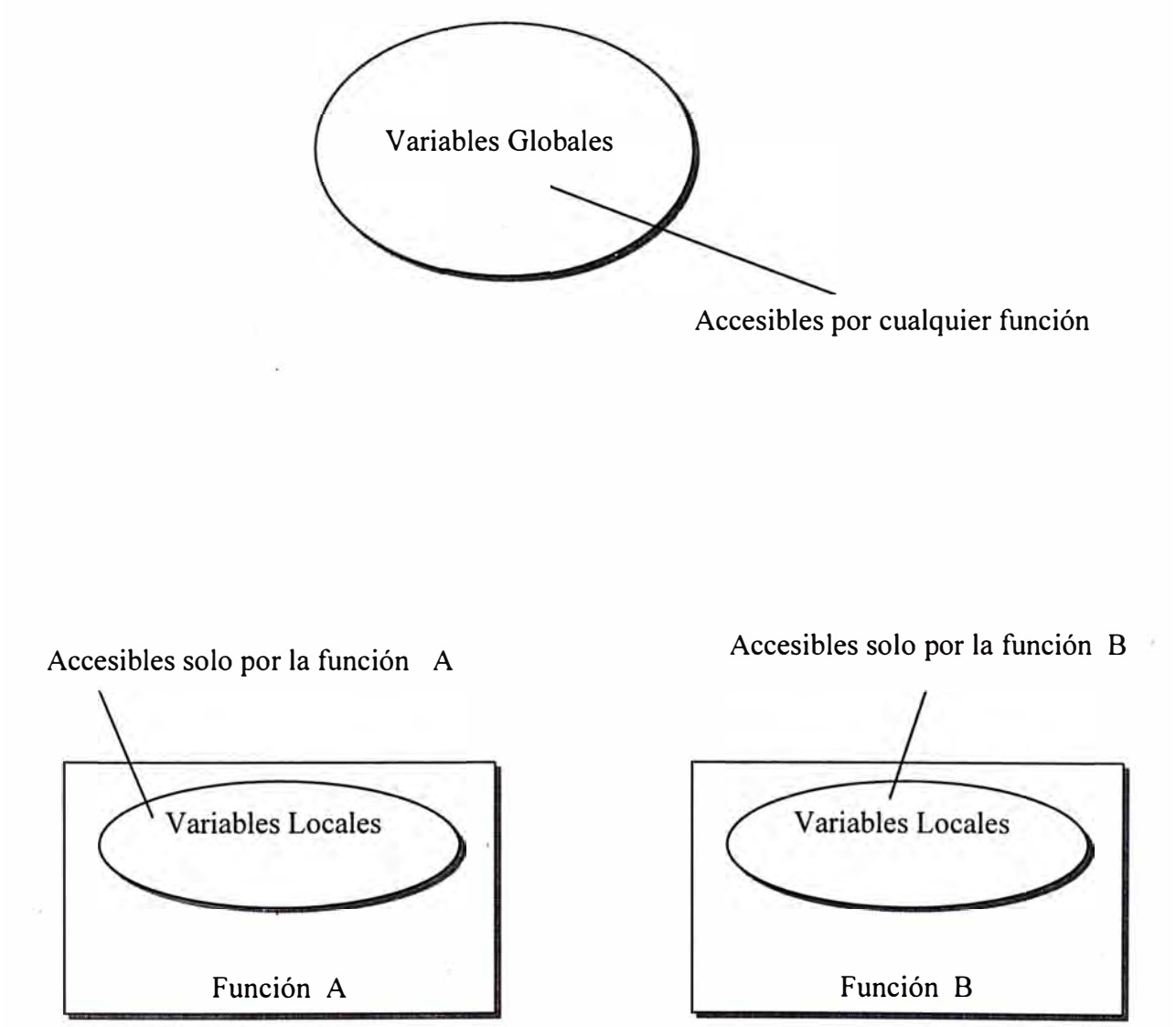


Figura 6-1 Variables Globales y Locales

Otro problema es que debido a las muchas funciones que accesan los datos, la manera como estos se almacenan es critica. El arreglo de los datos no puede cambiarse sin modificar todas las funciones que accesan los datos. Si se adicionan nuevos items de datos, por ejemplo se necesita modificar todas las funciones que accesan los datos para que accesen estos nuevos items. Será muy difícil encontrar todas las funciones, y más aún modificar todas ellas correctamente.

La relación de funciones y datos en programas procedimentales se muestra en la figura 6-2

De esta manera lo que se necesitaba era una manera de restringir el acceso a los datos, para ocultarlos de todas aquellas funciones que nada tenía que hacer con estos. Esto protegería los datos, simplificaría el mantenimiento y ofrecería otros beneficios.

6.2.2.2 Relaciones con el mundo real

Los programas procedimentales son comúnmente difíciles de diseñar. El problema se debe a que sus principales componentes funciones y estructuras de datos no modelan el mundo real correctamente. Estas limitaciones pueden verse a la hora de escribir un programa para crear los elementos de una interface de usuario gráfica: menús, Windows, etc. Que funciones se necesitaran?, ¿Cuál es la estructura de datos?. Las respuestas no son tan obvias.

Todo esto sería más comprensible si los windows y menús correspondieran mas cercanamente a elementos del programa.

6.2.2.3 Tipos de Datos Nuevos

Hay otros problemas con los lenguajes tradicionales. Uno es la dificultad de crear nuevos tipos de datos. Normalmente los lenguajes de computadoras tienen varios tipos de datos incluidos: enteros, punto flotante, caracteres, etc. Que pasa si queremos inventar nuestro propio tipo de datos? Pueda ser que queramos trabajar con números complejos, o coordenadas en un espacio físico, o cualquier otro tipo imaginable tal que los incluidos no manejan fácilmente. La extensibilidad o habilidad de crear nuevos tipos de datos nos permiten extender las capacidades del lenguaje. Los lenguajes tradicionales no son usualmente extensibles. Es imposible combinar coordenadas x, y, z en una variable simple llamada Punto, y luego sumar y restar valores de este tipo. El resultado es que en lenguajes tradicionales es más difícil de manejar esto.

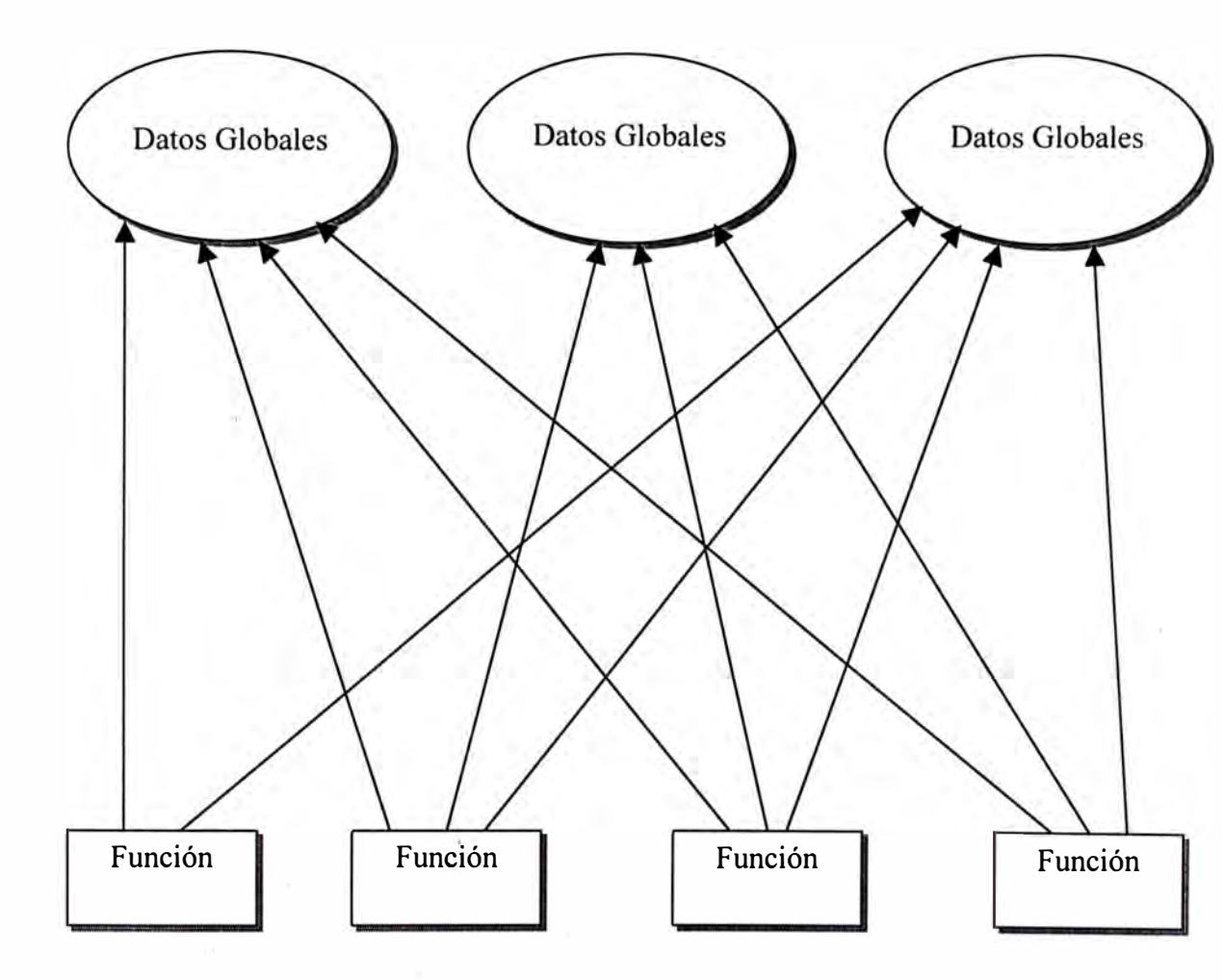


Figura 6-2 El Paradigma Procedimental

6.2.3 Programación Tradicional vs. Programación Manejada por Eventos

En un programa procedimental tradicional, las porciones de código a ejecutarse están controladas por la aplicación en vez de serlo por un control de eventos. La aplicación empieza con la primera línea de código y sigue un camino definido, llamando procedimientos según se necesiten.

En una aplicación manejada por eventos, el usuario o el sistema ejecuta un procedimiento activado por un evento. Así el orden en el cual el código se ejecuta depende del orden en el cual el evento ocurre; el orden en el cual los eventos ocurren esta determinado por las acciones de los usuarios. Esta es la esencia de las interfaces gráficas y la programación manejada por eventos: el usuario es responsable y el código responde acordemente.

Debido a que no podemos predecir lo que el usuario hará, el código debe hacer sus suposiciones acerca del "estado del mundo" cuando este se ejecuta.

Un evento es una acción reconocida por un objeto. Cada objeto es capaz de reconocer un conjunto de eventos, por lo que se dice que cada objeto tiene su propio dominio de eventos.

6.3 Necesidad de la Programación Orientada al Objeto.

La programación orientada al objeto es la innovación mas dramática en el desarrollo de software de la ultima década. La importancia del impacto de la programación orientada al objeto esta en el rango del desarrollo de los primeros lenguajes de alto nivel, en los inicios de la edad de la computación. Tarde o temprano todo desarrollador será afectado por el enfoque orientado al objeto para el desarrollo de artefactos de software.

En la actualidad hay quienes aún se preguntan:

- ¿Porqué necesitamos la programación orientada al objeto?
- ¿Qué obtenemos con los Lenguajes Orientados al Objeto?
- ¿Qué no podemos obtener con los lenguajes tradicionales tales como Cobol, 'C', Pascal, o BASIC?
- ¿Cuáles son los principios detrás de la programación orientada al objeto?
- ¿Cuál es la relación entre C++ y el viejo 'C'?

La Programación orientada al objeto fue desarrollada debido a las limitaciones que se descubrieron en los enfoques iniciales de la programación. Para apreciar lo que el paradigma de la programación orientada al objeto nos ofrece, necesitamos comprender cuales fueron estas limitaciones y como se originaron en los paradigmas tradicionales de programación, lo cual ha sido explicado líneas arriba.

6.3.1 Perspectiva de la Orientación al Objeto

La idea fundamental detrás de los lenguajes orientados al objeto, es la combinación que tiene lugar dentro de una única unidad de dos elementos: *datos y funciones que operan sobre esos datos.* A dicha unidad se le llama Objeto.

En C++, los datos de los objetos son llamados *item de datos o datos miembros* y las funciones del objeto son llamadas *funciones miembro*. Las funciones miembro típicamente proveen la única manera de accesar los items de datos. Si se quiere leer un item de datos en un objeto, tenemos que llamar una función miembro del objeto. La función leerá el dato y retornara su valor. No se pueden accesar los datos directamente. Los datos del objeto están ocultos de tal manera que están asegurados contra alteraciones accidentales. Los datos y sus funciones se dice que están *encapsuladas* dentro de una única entidad.

Los términos *Datos encapsulados* y *datos ocultos* son claves en la descripción de lenguajes orientados al objeto.

Si queremos modificar los datos en el objeto, sabemos que funciones exactamente interactúan con estos: las funciones miembro en el objeto. Ninguna otra función puede accesar los datos. Esto simplifica los esfuerzos, a través de las diversas etapas del ciclo de vida de los sistemas. El paradigma orientado al objeto se muestra en al figura 6-3

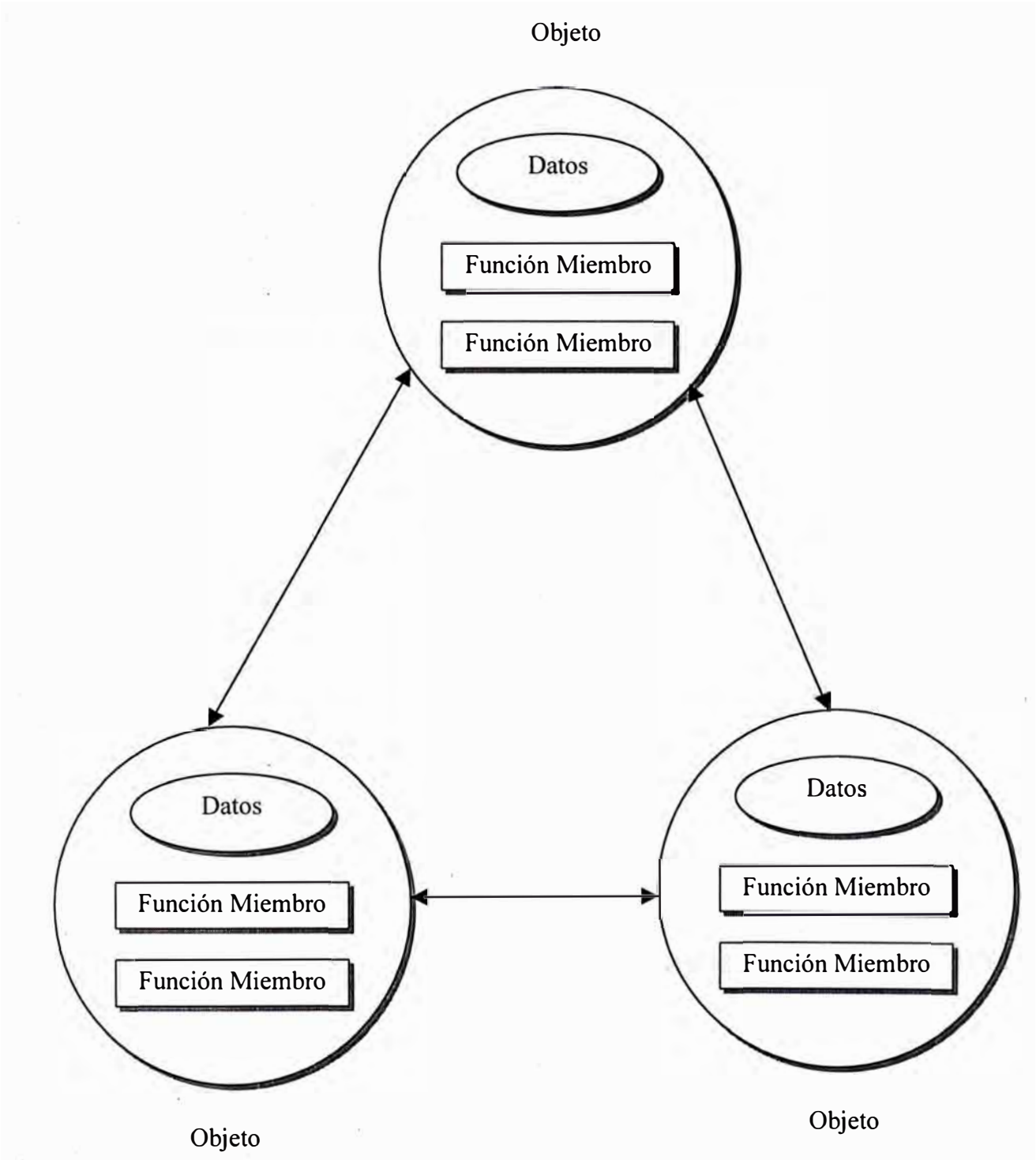


Figura 6-3 El Paradigma Orientado al Objeto

6.3.2 Una Analogía

Supongamos que nos encontramos en una empresa con un grado de organización tal que el tipo de la misma es por departamentos. Nosotros podríamos pensar como que los objetos son los departamentos de la compañía tales como ventas, contabilidad, personal etc. Sabemos por experiencia que en este tipo de organizaciones los empleados no trabajan un día resolviendo problemas en el departamento de personal, en contabilidad al siguiente día, para luego trabajar en ventas la semana que viene.

Cada departamento tiene su propio personal, con tareas asignadas claramente. Así mismo cada departamento tiene sus propios datos: planillas, montos de ventas, datos de personal, o inventario dependiendo del departamento.

Los empleados en cada departamento controlan y operan con los datos de su departamento. Dividiendo la compañía en departamentos facilita la comprensión y control de actividades de la misma, y ayuda a mantener la integridad de la información usada por la compañía. El departamento de planilla por ejemplo es responsable por los datos de planillas. Si alguien del departamento de ventas, necesita saber el total de salarios pagados a los vendedores de la sucursal del Cuzco por ejemplo en marzo, esa persona no va directo al departamento de planillas y empieza a buscar por si solo en los gabinetes del departamento de planillas, pues esa no es la manera correcta. La manera correcta es enviar un memo a la persona apropiada en el departamento, y luego se tendrá que esperar para que dicha persona accese los datos, para después enviar la información a quien la necesite. Esto asegura que la información sea accesada correctamente y que no es corrompida por personal extraño e inepto. De la misma manera, los objetos proveen una manera de organización del programa, ayudando a mantener la

integridad de los datos del programa. La vista de la organización corporativa se muestra en la figura 6-4.

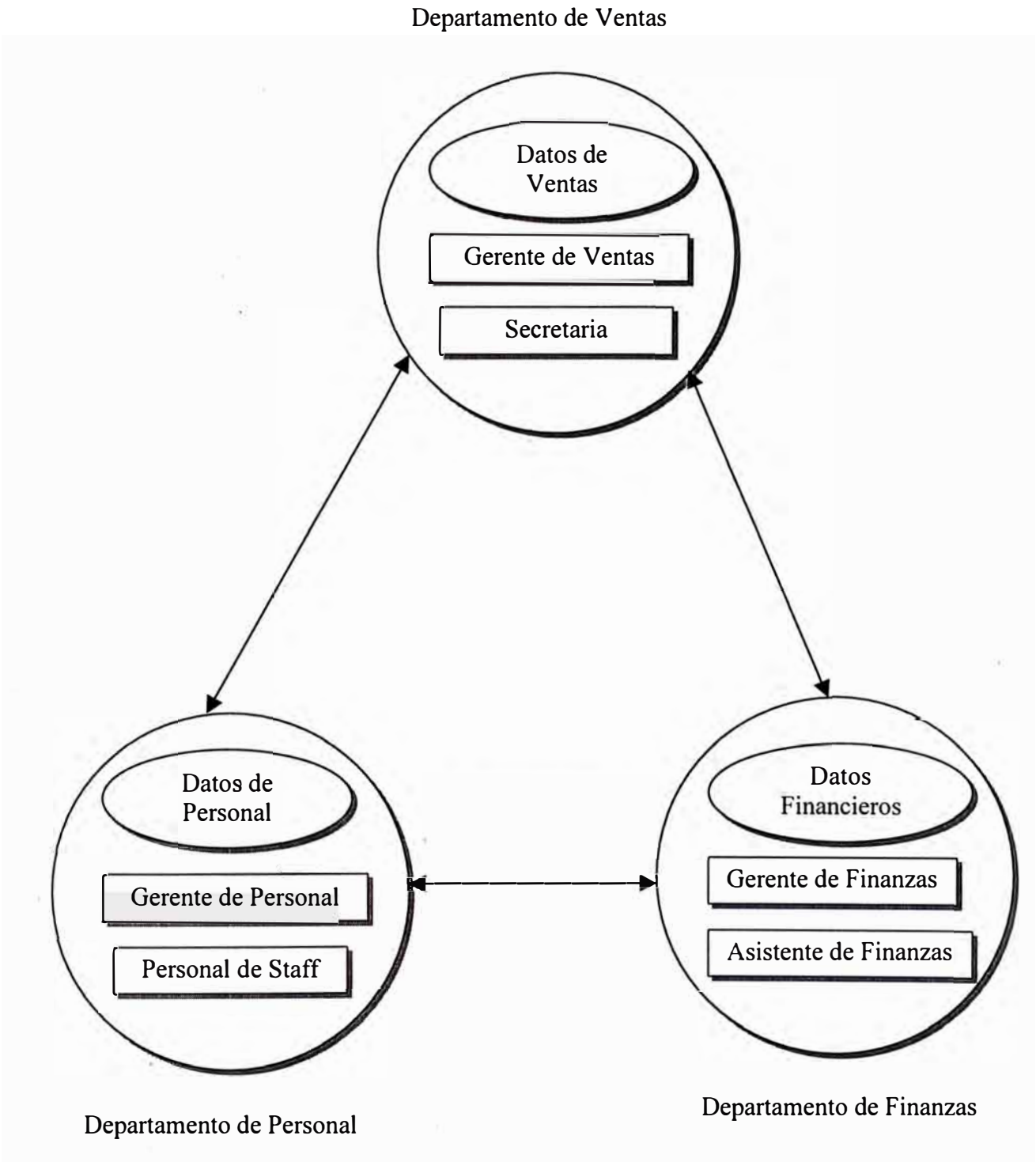


Figura 6-4 El Paradigma Corporativo

6.3.3 Características de los Lenguajes Orientados al Objeto

Examinemos brevemente unos cuantos de los principales elementos de los lenguajes orientados al objeto en general y a C++ en particular.

6.3.3.1 Objetos

Cuando abordamos un problema de programación en un lenguaje orientado al objeto, no preguntamos mas si el problema será dividido en funciones, sino como será este dividido en objetos. Pensar en términos de objetos, antes que en funciones tiene una sorprendente ayuda en la manera fácil de diseñar los programas. Esto como resultado de la comparación entre objetos en el sentido de programación y objetos en el mundo real.

Veamos algunas categorías típicas de objetos:

- Objetos físicos
  Automóviles en una simulación de flujo de trafico
  Componentes eléctricos en un programa de diseño de circuitos
  Aviones en un sistema de control de trafican aéreo

- Elementos en el entorno del usuario de computadoras
  Ventanas
  Menús
  Objetos gráficos(líneas, rectángulos, círculos)
  El ratón, teclado, impresora, disk drivers

- Almacenamiento de datos
  Arreglos de elementos
  Pilas

Listas

Arboles Binarios

- Entidades Humanas
  Empleados
  Estudiante
  Clientes
  Vendedores

- Colecciones de datos
  Un Inventario
  Un archivo de personal
  Un diccionario
  Una tabla de latitudes y longitudes de ciudades del mundo

- Tipos definidos por el usuario
  Tiempo
  Angulos
  Números Complejos
  Puntos en el plano

- Componentes en juegos de computadoras
  Fantasmas en un laberinto
  Posiciones en una tablero de juego(damas, ajedrez)
  Animales en una simulación ecológica
  Oponentes y amigos en juegos de aventura

El enlace entre los objetos en programación y los objetos en el mundo real es el feliz resultado de combinar datos y funciones. Los objetos resultantes ofrecen una revolución en los problemas de diseño.

En ciencias de la computación, un objeto es una entidad abstracta que encarna las características de un *algo* de la vida real. La programación basada en objetos no es dependiente de lenguaje alguno tal como C++, o Pascal Orientado al Objeto. La programación basada en objetos tiene ya algún tiempo en el medio, y es usada con lenguajes tan diversos como **FORTRAN, LISP, COBOL, ASSEMBLER**.

Los objetos son el resultado de un paradigma, de una metodología de programación, y no de un lenguaje.

Específicamente en C++ los objetos son elevados al más importante nivel. Con esta nueva metodología, todo lo que se manipula en un programa es considerado un objeto. Los objetos son creados y manipulados durante la ejecución del programa. Los objetos tienen vida, intercalan entre sí y todos pueden ser puestos en grupos, arreglos, colecciones, listas etc.

En este nuevo paradigma los objetos existen como producto de un modelo, o como la entidad creada a partir de un patrón llamado *clase*.

Los objetos son construcciones programadas o moldeadas de acuerdo a las *clases,* es decir un objeto es una variable de una determinada clase.

6.3.3.2 Clases

Clases son modelos y como tal abstracciones de la realidad. Una clase es en si misma un plano, patrón o plantilla a partir de la cual se dará vida a los objetos que vayamos a definir. Así una clase es una colección de objetos similares, esto cumple con nuestra concepción no técnica de la realidad. Por ejemplo Bill Clinton, Fujimori, Menen son miembros de la clase presidentes de un país. No hay una sola persona llamada "presidente de un país", pero gente especifica con nombres específicos, son miembros de esta clase si ellos poseen ciertas características.

6.3.3.2.1 Items de Datos (Data Items)

Vienen a ser las variables incluidas dentro de la clase, en donde se almacenaran los valores referidos a la naturaleza de la clase. Sus tipos son todos los permitidos por el lenguaje, e incluso pueden ser los definidos por el desarrollador.

6.3.3.2.2 Funciones Miembros (Member Functions)

Las funciones miembro (métodos, mensajes, funciones, etc.) son las funciones incluidas dentro de una clase. Estas funciones son las encargadas de manipular o llevar a cabo las diferentes operaciones sobre los datos miembros.

6.3.3.2.3 Encapsulación

Los items de datos especificados en la clase así como las funciones que accesan dichos datos forman una unidad (cápsula), dentro de una misma entidad.

La figura 6-5 muestra en la página siguiente muestra la encapsulación

6.3.3.2.4 Datos Ocultos (Data Hiding)

Esta es una característica clave de la programación orientada al objeto, y significa que los datos están escondidos o protegidos dentro de una clase, de tal manera que estos no pueden ser accesados erróneamente por funciones que están fuera de la clase. Los datos no tienen visibilidad fuera de la clase. El ocultamiento de los datos se da a través de los siguientes especificadores de alcance o visibilidad:

- *Private*: Los datos o funciones solo pueden ser accesados desde dentro de la clase. Por defecto los datos dentro de una clase son privados.
- *Public*: Los datos o funciones son accesibles desde dentro de la clase y además fuera de la clase.
- *Protected*: Los datos o funciones son accesibles desde dentro de la clase y además por las clases heredadas. La figura 6-5 muestra la protección de datos.

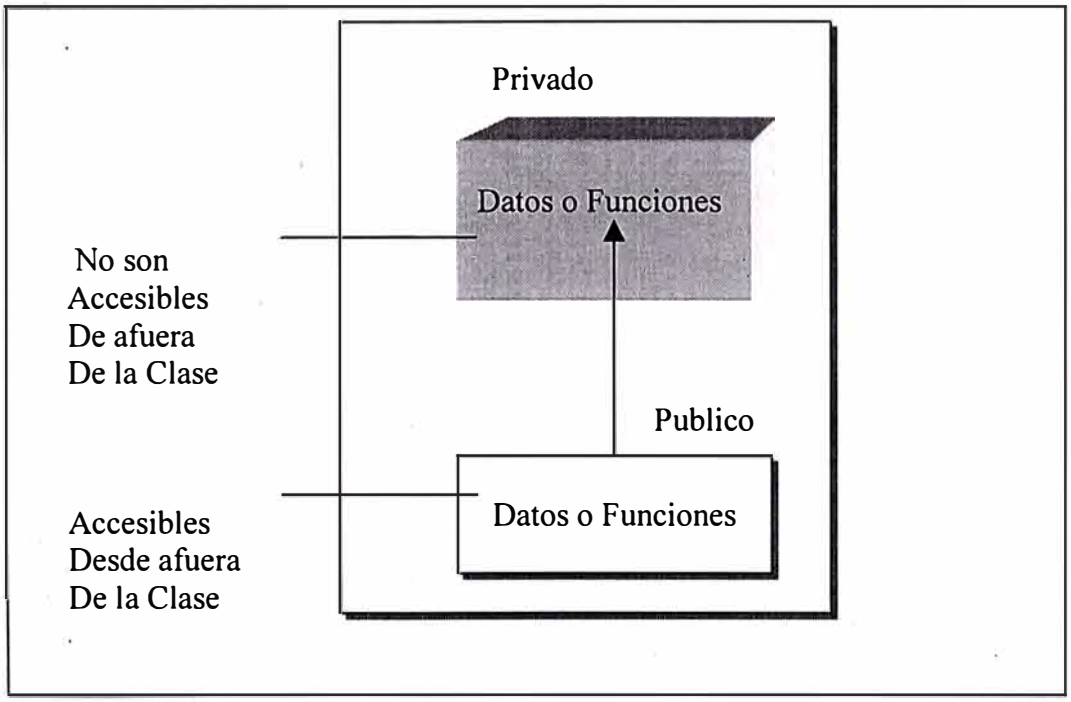


Figura 6-5. Privado y Público

6.3.3.2.5 Constructores

Son funciones miembro especializadas, las cuales tienen por tarea inicializar los objetos automáticamente en el momento de su creación, lo que traerá como beneficio eliminar los errores debidos al uso de datos no inicializados. Las clases pueden declarar uno o más constructores dependiendo de la naturaleza a modelarse. Los constructores no son llamados por las declaraciones del programa, sino que son automáticamente ejecutados por el compilador.

6.3.3.2.6 Destructores

Son funciones miembro las cuales son llamadas automáticamente para efectuar una serie de operaciones necesarias cuando el objeto ya no se necesita. Normalmente los destructores ejecutan las operaciones inversas de los constructores. Cada clase solo admite un destructor y al igual que los constructores no pueden ser llamados directamente por declaraciones dentro del programa, sino que son llamados automáticamente por el compilador.

6.3.3.2.7 Creación de Nuevos Tipos de Datos

Uno de los beneficios de los objetos es que estos dan al desarrollador una manera conveniente de construir nuevos tipos de datos. Supongamos que trabajamos con posiciones en tres dimensiones (tal como coordenadas x, y, z, o latitud, longitud, altitud) en un programa, de seguro que nos gustaría expresar valores posicionales con operaciones aritméticas normales, tale como:

Pos1 = pos2 + origen

Donde las variables pos1, pos2, y origen representan respectivamente una terna de cantidades numéricamente independientes. Creando una clase que incorpore estos tres valores, y declarando pos1, pos2, y origen como objetos de esta clase, podemos crear un nuevo tipo de dato. Muchas características de los lenguajes orientados al objeto están dedicadas a facilitar la creación de nuevos tipos de datos en esta manera.

6.3.3.2.8 Overloading (Sobrecarga)

Este término es una clase de polimorfismo, y esta referido al hecho de que a un operador existente, tal como + o = se le de la capacidad de operar en un nuevo tipo de datos. En el caso anterior se dice que el operador ha sido overloaded (sobrecargado).

6.3.3.2.9 Overriding (Superar)

Este término se refiere al hecho de que una función miembro puede ser implementada de diferente manera con el mismo nombre en las clases derivadas. El objetivo es que las llamadas a las funciones sean idénticas para objetos creados con la clase base y la derivada, pero el comportamiento de acuerdo a la implantación respectiva.

6.3.3.3 Herencia

La idea de clases nos lleva a la idea de *herencia*. En nuestra vida diaria, usamos el concepto de clases como divididas en subclases. Conocemos que la clase de animales esta dividida en mamíferos, anfibios, insectos, aves etc. La clase de vehículos esta dividida en carros, camiones, ómnibuses y motocicletas.

El principio de esta clase de división es que cada subclase comparte características comunes con la clase desde la cual es derivada. Carros, camiones, ómnibuses y motocicletas todos tienen ruedas y un motor; estas son las características definidoras de los vehículos. Adicional a las características compartidas con otros miembros de la clase cada subclase también tiene sus propias características particulares: los ómnibuses por ejemplo tienen asientos para mucha gente, los camiones tienen espacio para carga.

De manera similar, una clase en la programación orientada al objeto puede ser dividida en subclases. En C++ la clase original es llamada la *clase base;* y otras clases que comparten estas características pueden ser definidas, pero adicionan otras características y refinamientos propias de ellas. Estas clases son llamadas *clases derivadas.*

No se confunda la relación de objetos a clases con la relación de una clase base a una derivada. Los objetos, los cuales existen en la memoria de la computadora, cada una encarna características de su propia clase, las cuales sirven como una plantilla. Las clases derivadas heredan las características de su clase base, pero adicionan nuevas propias de ellas.

La herencia es probablemente la característica más poderosa de la programación orientada al objeto después de las clases, y permite la reusabilidad del código. Una vez que una clase ha sido escrita y depurada, no necesitamos tocarla nuevamente, pero puede sin embargo ser adaptada a trabajar en diferentes situaciones. Reusando el código existente ahorra dinero y tiempo e incrementa la confiabilidad de un programa. La herencia puede también ayudar en la conceptualización original de un problema de programación, y en el diseño global del programa.

Un importante resultado de la reusabilidad es la facilidad en la distribución de las librerías de clases.

Un desarrollador puede usar una clase creada por otra persona o compañía, y sin modificar esta, derivar otras clases desde esta adecuadamente para una situación particular. La idea es mostrada en la figura 6-6.

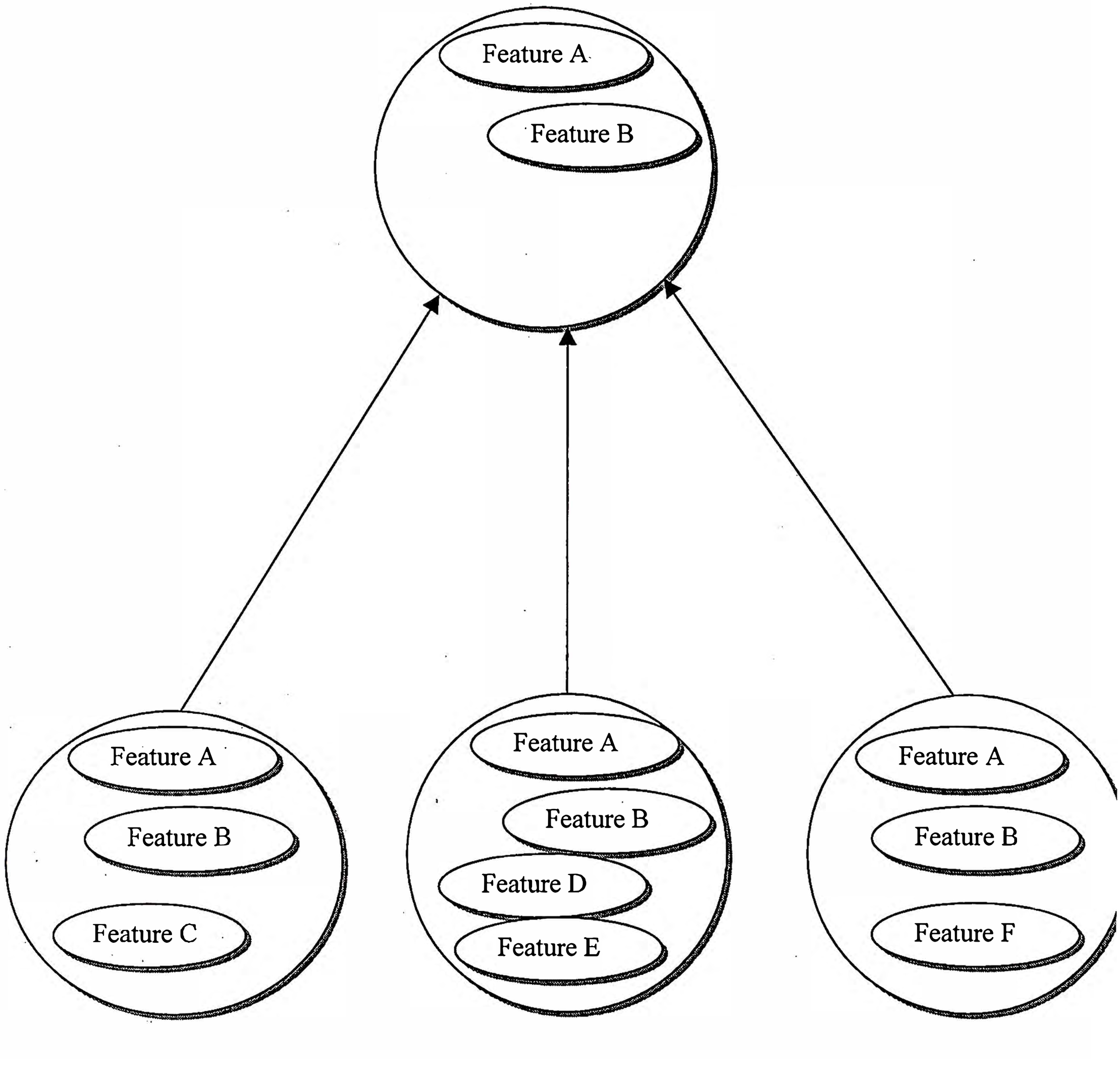
6.3.3.3.1 La Herencia y el Desarrollo de Programas

El proceso de desarrollo de software como se ha estado practicando por décadas por los desarrolladores, esta siendo fundamentalmente alterado por la programación orientada al objeto. Esto no es debido únicamente al uso de clases, sino también al uso de herencia como veremos en la siguiente ilustración.

El programador A crea una clase X. El programador B piensa que la clase X podría ser mejorada, entonces deriva una nueva clase Y a partir de la clase X, y adiciona algunas extensiones.

Los programadores C, y D luego escriben aplicaciones que usan la clase Y.

El programador B puede que no tenga acceso al código fuente de las funciones miembros de la clase X, y los programadores C, y D pueden tampoco tener acceso al código fuente de la clase Y. Pero debido a la característica de reusabilidad de software en los lenguajes orientados al objeto, B puede modificar y extender el trabajo de A, C y D pueden crear aplicaciones.



Clases Derivadas

Figura 6-6 La Herencia

Notemos que la distinción entre desarrolladores de herramientas de software y desarrolladores de aplicaciones es borrosa o imprecisa. El programador A crea una herramienta de programación de propósito general, la clase X. El programador B crea la clase Y, que es una versión especializada de la clase X. Los programadores C, y D crean aplicaciones. A es un desarrollador de herramientas, C y D son desarrolladores de aplicaciones. B esta en algún punto intermedio.

De cualquier manera la programación orientada al objeto esta haciendo el escenario de la programación más flexible y a la vez más compleja.

6.3.3.4 Polimorfismo

Este término prestado de la biología describe que organismos relacionados pueden asumir una variedad de formas. En la programación orientada al objeto el polimorfismo, esta referido al uso de operadores y funciones de diferentes maneras, dependiendo sobre que tipo de datos están siendo usados. Así podemos ver que un operador o función tiene distintas formas de comportarse, y que es el tipo de objeto sobre el cual se actúa el determinante del comportamiento.

6.3.3.5 La Reusabilidad en la Orientación al Objeto

Una vez que una clase ha sido escrita, creada y depurada, esta puede ser distribuida a otros programadores para uso en sus propios programas. Esto es llamado *Reusabilidad.* Esto es similar a la manera en que una librería de un lenguaje procedural puede ser incorporada dentro de diferentes programas.

Sin embargo, en la programación orientada al objeto, el concepto de herencia provee una importante extensión a la idea de la reusabilidad. Un programador puede tomar una clase existente y sin modificar esta adicionarle más características y funcionalidad. La nueva clase heredara la funcionalidad de la vieja, pero esta es libre de adicionar nuevas características propias.

La facilidad con la cual el software existente puede ser rehusado, es uno de los principales beneficios, posiblemente el principal beneficio de la programación orientada al objeto. Muchas compañías encuentran que el reuso de las clases en un segundo proyecto provee una utilidad mayor en la inversión original.

6.3.3.6 Instantiación o Creación de Objetos

Una clase no es una entidad física en almacenamiento. Siendo similar a un tipo de datos, el almacenamiento solo es separado cuando la clase es usada para crear un objeto. Este proceso de creación es llamado instantiación. El término es referido al hecho de hacer una instancia (ocurrencia) o entidad física de una clase.

Como resultado de este uso de la clase, tenemos un objeto con las mismas características de la clase usada, la cual ha sido usada como plantilla.

6.3.3.7 Clases Objetos y Memoria

En la programación orientada al objeto, la memoria es utilizada de una manera eficiente.

Todos los objetos de una clase dada usan las mismas funciones. Las funciones miembros son creadas y puestas en memoria sólo una vez, cuando son definidas en la especificación de la clase. Debido a que las funciones para cada objeto son idénticas, no hay sentido en duplicar las mismas cada vez que se cree un objeto de la misma clase. Los datos miembros, sin embargo mantendrán diferentes valores, por lo que debe haber una instancia de cada dato miembro en cada objeto.

Los datos miembros son por esto puestos en memoria cuando cada objeto es definido, luego tenemos un juego de datos por cada objeto.

La figura 6-7 muestra las funciones y datos en memoria.

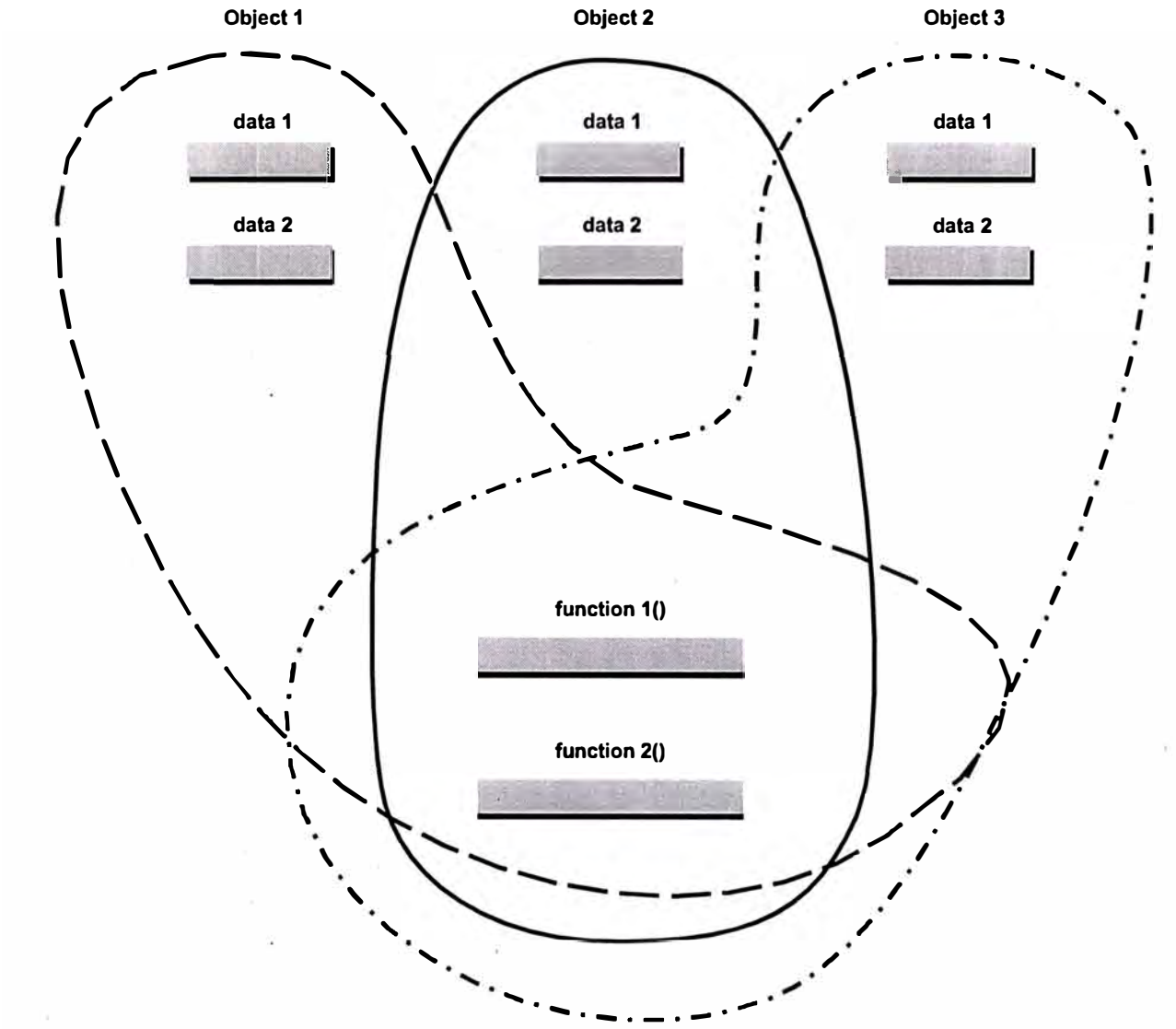


Figura 6-7 Las funciones y los datos en la memoria

6.3.3.8 El Viejo C y el Nuevo C++

El lenguaje C++ es derivado del lenguaje C. Hablando estrictamente C++ es un superconjunto de C: Casi cada una de las declaraciones correctas en C, es también una declaración correcta en C++, aunque lo inverso no es cierto. Los elementos más importantes adicionados a C para crear C++ están relacionados a clases, objetos, y la programación orientada al objeto. (C++ fue originalmente llamado "C con clases.") Sin embargo, C++ tiene muchas otras características nuevas. La figura 6-8 muestra la relación entre C y C++.

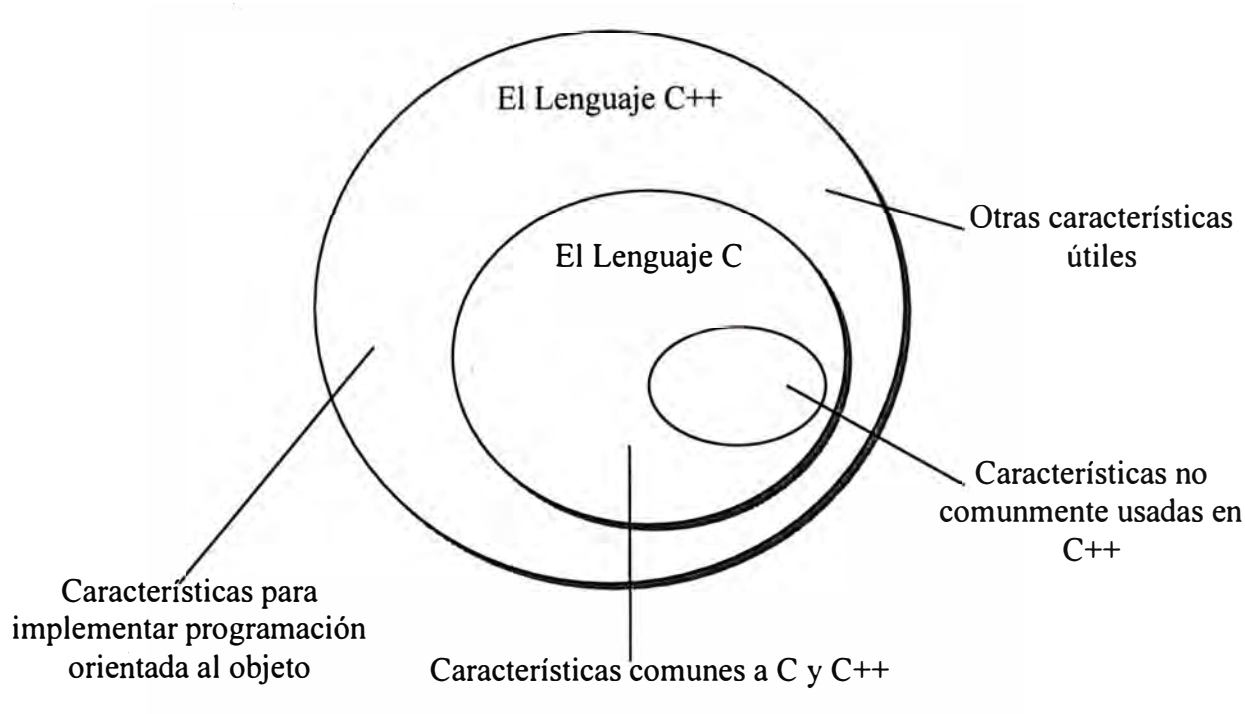


Figura 6-8 La relación entre C y C++

6.3.3.9 ¿Por qué BC++?

Dentro de la gama LPV los más sobresalientes en promover mecanismos para un buen diseño, altamente frameworks, compactos, con código bien definido, y enmarcados dentro del rigor de un lenguaje de programación orientado al objeto se encuentran Delphi y BC++. Mas aún estos LPVs corrigen más de las deficiencias encontradas en otros LPVs como Visual Basic, Power Builder etc. Tal vez la principal diferencia este dada por el lenguaje de soporte detraes del IDE, por un lado tenemos Object Pascal (Delphi) y por otro C++, los cuales son ya conocidos en cuanto a su rapidez en el medio, y así mismo cuentan con extensiones para soportar buenas prácticas de programación y generación de código más eficiente (conseguir más con menos líneas de código). Dichos lenguajes detrás del IDE son verdaderos lenguajes de programación orientados al objeto.

También cabe mencionar que el software entregable o output generado por Delphi o BC++, es realmente una aplicación ejecutable, a diferencia por ejemplo de VB el cual se asemeja a una macro grande o a un archivo scrip. En VB un programa llamado VBRUN300.DLL debe acompañar el archivo. EXE para correr dicha macro. Cualquier error cometido contra este archivo acompañante como olvidarlo o borrarlo redundara en que la aplicación no pueda ser corrida. Por lo expuesto anteriormente es fácil deducir que aquella aplicación verdaderamente ejecutable será mas clara, fácil de distribuir y mantener.

Concluyentemente BC++ provee un compilador que esta optimizado para entregar una aplicación rápida, sin que el desarrollador tenga que trabajar duro para optimizar el programa.

Así BC++ es un entorno visual de programación, orientada al objeto para el desarrollo rápido de aplicaciones de propósito general, y cliente/servidor que corren bajo Windows 95/98 y NT. Usando BC++ se pueden crear Aplicaciones-Windows altamente confiables y eficientes.

BC++ provee una exhaustiva librería de componentes reusables y un juego de diseño de herramientas RAD, incluyendo aplicaciones y patrones de formas, y expertos de programación.

El entorno visual de programación (IDE) incluye todas las herramientas necesarias para diseñar, desarrollar, probar, y depurar una aplicación.

En resumen se citan algunas de las bondades de BC++

- Poderoso manejo de tipos de datos
- Poderosas habilidades para la estructuración de los datos
- Buen soporte para modularidad
- Poderosas facilidades de depuración
- Poderosas facilidades de edición
- Poderosa habilidad para llamar rutinas de otros lenguajes y para ser llamados por rutinas escritas en otros lenguajes
- Poderosas facilidades de soporte de trabajo grupal, incluyendo herramientas de control de código fuente.

6.3.4 Intefaces del Entorno Visual

Al comienzo, la vasta mayoría de programas Windows fueron escritos en C. De hecho, la Interface de Programación de Aplicaciones Windows (API por sus siglas en inglés de Aplicación Programming Interface) es justamente una colección gigantesca de funciones C, hay mas de mil. El API esta compuesta de tres principales DLLs (por sus siglas en inglés de Dynamic Link Library) los cuales son verdaderos programas de 32-bits: User, Kernel, y GDI (por sus siglas en inglés de Graphics Device Interface). El User DLL manipula el sistema de ventanas y controles. El Kernel manipula las tareas del núcleo tales como la asignación de memoria, intercambio de tareas, y gestión de procesos. El GDI manipula los gráficos y dibujos.

Cada uno de estos DLLs esta compuesto de númerosas subrutinas que las aplicaciones deben llamar desde un lenguaje de alto nivel, que provea una adecuada interface.

Desafortunadamente, el API no es fácil de usar, y esta repleta de crípticas declaraciones de funciones que son difíciles de descifrar. Lo peor de todo es que el API no provee prácticamente información acerca de cuando es apropiado llamar a especificas funciones. Algunos lo asemejan a un mapa con gran cantidad de destinos importantes pero sin instrucciones de como llegar a estas.

La API podría ser dividida como sigue:

User: contiene todas las funciones para manipular las ventanas, incluyendo controles, etc. Entre los grupos de funciones en el User DLL tenemos:

- Funciones de creación y gestión de ventanas
- Funciones de menús
- Funciones de Mensajes
- Funciones de Diálogos
- Funciones del Clipboard

Kernel: contiene funciones que tratan con el sistema mismo. Aquí tenemos algunos grupos de funciones:

- Funciones de Asignación de Memoria
- Funciones de Procesos
- Funciones Thread
- Funciones de Archivos

GDI: contiene funciones que tratan con gráficos e impresión, tales como:

- Funciones Bitmap
- Funciones de Dibujo
- Funciones de Impresión
- Funciones Metafiles
- Funciones de Fonts
- Funciones de Help

Alguien penso que debería haber un modo más fácil de llevar a cabo las tareas de desarrollo. Así fue aparente que la programación en Windows estaba bien adecuada para el lenguaje C++, y viceversa. Creando clases que encapsularán tareas comunes de la programación en Windows, un programador podría ser mucho más productivo.

Una vez que la clase fuera creada para encapsular las varias tareas de una ventana, por consiguiente esa clase podía ser usada una y otra vez y así la revolución de Frameworks empezó.

6.3.4.1 Frameworks

Un Framework es una colección de clases que simplifican la programación en Windows, encapsulando técnicas de programación comúnmente usadas. Los frameworks son también llamados librerías de clases.

Los frameworks populares tienen clases que encapsulan ventanas, controles de edición, cajas de listas, operaciones de gráficos, bitmaps, scrollbars, cajas de diálogos, etc.

Entonces el punto principal es que los frameworks hacen la programación en Windows mucho más fácil de lo que esta sería si se usara simplemente C. En otras palabras los frameworks ocultan detalles al desarrollador, los cuales no se necesitan conocer. Todo lo que el desarrollador hace es tomar los objetos que forman el framework y los hacen trabajar en los programas. Un buen framework toma completa ventaja de la programación orientada al objeto. Algunos lo hacen mejor que otros. La Object Windows Library y la Visual Component Library de Microsoft son ejemplos grandiosos de programación orientada al objeto. Estos proveen la propia abstracción necesaria para desarrollar cualquier tipo de aplicación en el negocio de la programación de una manera seria y profesional.

6.3.4.1.1 Costo por usar frameworks

Generalmente podría pensarse que los programas desarrollados con frameworks son más grandes y lentos que la contraparte escrita en C. Esto es parcialmente correcto. Las aplicaciones escritas con frameworks no necesariamente tienen que ser más lentas que los programas escritos en C. Hay alguna sobrecarga adicional inherente en el lenguaje C++, pero no es notoria en un programa típico Windows.

Los programas escritos en C++ tienden a ser más largos que su contraparte C, digamos que el peor caso es de tres a uno, lo cual es una diferencia significante. La diferencia de tamaño en programas finales entre una aplicación C y C++ desarrollada con frameworks será más notoria en programas muy pequeños. A medida que el programa crezca en tamaño y este sea más sofisticado, las diferencias en tamaño serán menos notoria.

Una de las diferencias es simplemente la diferencia entre C y C++. C++ tiene una sobrecarga debido a sus características tales como el manipuleo de excepciones, información en tiempo de ejecución (RTTI), y otros adicionales. En la opinión de expertos, la diferencia en el tamaño del código es aceptable debido a las características que el lenguaje C++ provee. Yo soy un creyente de que debemos desarrollar software con la menor cantidad de líneas de código como sea posible, con las herramientas que tenemos disponibles. Pero yo también soy realista y comprendo que el tiempo hoy en día es un ingrediente definitivo en la industria del software. Yo soy siempre partidario de intercambiar algún tamaño de código por la fuerza que nos da el lenguaje C++ combinado con una aplicación framework.

Los frameworks necesitan ser separados dentro de dos categorías: frameworks C++ y VCL. Primero discutiré los frameworks C++ y luego VCL. Hay realmente dos frameworks C++ viables, y ellos son OWL (por sus siglas en inglés Object Windows Library) de Borland y MFC (por sus siglas en inglés Microsoft foundatión Class Library) de Microsoft.

6.3.4.1.2 Librería de Objetos Windows (OWL) de Borland

Borland tomó la delantera en la carrera de frameworks con OWL hace unos pocos años atrás. Primero fue OWL 1.0. Esta primera versión de OWL fue un producto separado vendido por Borland para ser usado con el compilador C++ 3.0 (el mismo OWL que fue escrito para turbo Pascal y después convertido a C++).

Después de OWL 1 vino OWL2. OWL 2 fue una pieza maestra la cual incluía muchas de las características del lenguaje C++. Lo mejor de todo fue que OWL 2 se incluía como parte del compilador C++ 4.0. A partir de aquí Borland incluiría OWL como parte de su paquete C++.

Luego Borland reviso OWL 2.0 y emitió una nueva versión OWL 2.5. OWL 2.5 fue una versión mayor la cual adicionaba OLE (por sus siglas en inglés de Object Linking and Embedding) en un nuevo set de clases llamado el Framework de Objetos Componentes (OCF por sus siglas en inglés Object Component Framework). OCF no es técnicamente parte de OWL. Este trabaja muy bien con OWL, pero al mismo tiempo puede ser usado independientemente de OWL.

La ultima y extraordinaria versión es OWL 5.0. OWL 5.0 representa mejoras significantes a OWL 2.5. Los cambios primarios vienen en unas clases nuevas que encapsulan los nuevos Win32 Custom Controls. También OCF fue actualizado en la versión OWL 5.0.

OWL es un LPOO bastante amigable y cumple todas las reglas de la POO. Su nivel de abstracción esta balanceado entre la facilidad de uso y su poder. OWL ha hecho un gran trabajo al encapsular el entorno de Windows y eso es realmente una fortaleza. Parte del problema con ese nivel de encapsulación es que OWL es complejo, y algunas veces difíciles de comprender sobre todo cuando recién se esta aprendiendo. La complejidad de OWL, es considerada por algunos su debilidad. Toma algún tiempo conocer bastante bien esta herramienta, pero una vez que se logra conocerla, el desarrollo de programas Windows llega a ser muy eficiente.

6.3.4.1.3 La Librería de Fundación de Clases (MFC) de Microsoft

En un punto entre OWL1 y OWL2, nació MFC. MFC es incluida como parte del compilador Visual C++ de Microsoft. Actualmente versiones de MFC son incluidas con compiladores de Symantec, Watcom, también Borland y otros.

Podría decirse que MFC es una librería de clases diferente que OWL. MFC es menos abstracta y esta bastante cercana al API de Windows. Las fortalezas de MFC llacen en tres áreas primarias. Primero, es relativamente fácil de aprender. (Comprenda que ningún C++ framework de la mano con programación Windows es fácil de aprender, pero MFC es un poco más fácil de captar que la competencia). Es más fácil de aprender primeramente debido a que es menos abstracto en algunas áreas. Si alguien es nuevo a la

programación Windows, probablemente encontrara OWL y MFC del mismo grado de dificultad para aprender. Pero si ese alguien viene de programar en C y conoce API Windows, entonces MFC será más fácil de aprender que OWL.

Otra fortaleza de MFC, es que de acuerdo a algunos este framework es una delgada envoltura de API Windows. Otra vez para programadores Windows quienes se están moviendo del lenguaje C al C++ con MFC, encontraran una gran ventaja. Ellos pueden empezar usando MFC y sentir que nada ha cambiado.

Finalmente, MFC tiene el distintivo de pertenecer a Microsoft. La ventaja es que como una nueva característica y tecnología viene todo junto, entonces MFC puede ser el primero en implementar estos.

Microsoft puede emitir una nueva tecnología, y MFC puede ya tener soporte para esa tecnología cuando esta es anunciada.

MFC también tiene su debilidad. Primero esta es una envoltura débil de API. Aún cuando líneas arriba asegure que esto era una fortaleza, también constituye una debilidad. La idea detrás de una librería de clases es aislar al usuario de lo que no necesita saber o usar. MFC falla en esta prueba en muchos casos. Además MFC no es un LPOO amigable. Algunas veces aparenta ser una implementación de colección de clases que no trabajan y sin embargo funciona bien.

¿Quién Gana?

Sin temor a preguntarnos, MFC es más ampliamente usada que OWL. Parte de la razón es que MFC y el compilador Visual C++ llevan el nombre de Microsoft. No es un secreto que Microsoft es el rey en la industria de software de PCs. Tampoco es un secreto la potencia que Microsoft tiene en el mercado, la misma que otras compañías solas pueden soñar.

Hay quienes habiendo usado OWL y MFC extensivamente consideran que OWL es un mejor framework. Pero MFC es hoy en día indudablemente el framework de elección. Hay muchas razones, algunas de las cuales he tocado. Otras razones incluyen la percibida falta de dirección de Borland en los años recientes. Algunos decisores prefieren jugar seguro y compran productos producidos por Microsoft sin importar el mérito técnico.

6.3.4.2 La Librería de Componentes Visuales

En 1995 Borland introdujo un nuevo producto llamado Delphi, el cual fue un éxito instantáneo.

Delphi ofreció RAD usando algo llamado componentes.
Los componentes son objetos que pueden ser tirados en una forma y manipulados vía propiedades, métodos y eventos, todo esto dentro de un entorno de programación visual.

El concepto de programación basada en formas fue primero popularizada por Visual Basic de Microsoft.

A diferencia de Visual Basic, Delphi uso una derivación de Pascal como su lenguaje de programación.

Este nuevo lenguaje llamado Object Pascal introdujo POO al lenguaje Pascal. En ese sentido Object Pascal es a Pascal lo que C++ es a C. Delphi y Object Pascal representaron el matrimonio de programación orientada al objeto y programación basada en formas. Adicionalmente Delphi produciría verdaderos ejecutables, Programas Reales. Programas que no requieren un programa DLL (por sus siglas en inglés Dynamic Link Library) en el tiempo de ejecución para poder correr, programas que fueron compilados no interpretados, programas que corrían decenas de veces más rápido que programas desarrollados en Visual Basic. El mundo de la programación fue impresionado.

Delphi también introdujo la VCL, la misma que es un framework para desarrollo de aplicaciones Windows en Object Pascal. Pero realmente VCL no es comparable a OWL y MFC. Esta es un framework, pero el núcleo es muy diferente. Es diferente primeramente debido a que fue desarrollada con los conceptos de propiedades, métodos y eventos.

Como la tecnología de objetos promueve reusabilidad, la misma VCL que esta en el corazón de Delphi esta también en el corazón de BC++.

Las clases de la VCL encapsulan la API de Windows, con lo cual se simplifica el accesos a las funciones API, y ayuda al programador a usar esas funciones efectivamente. La VCL da la interface orientada al objeto Win32 API. Una aplicación compilada en BC++ trabaja como cualquier otra aplicación que llama funciones de API, pero el trabajo del desarrollador es tremendamente simplificado usando componentes y objetos, los cuales

manipulan las llamadas a las funciones por él, en el orden apropiado y con cualquier requerida estructura de datos.

En conclusión la VCL permite a los programadores construir aplicaciones visualmente arrastrando objetos (instantiación de objetos) sobre formas, para luego suministrar funciones de manipulación de eventos (métodos) para controlar lo que el objeto hace.

En vez de llamar funciones API, los programadores construyen objetos, establecen propiedades y llaman a los miembros de las clases.

CAPITULO 7

METODOLOGIA DE DESARROLLO

A continuación se describe la metodología de desarrollo para aplicaciones de base de datos cliente/servidor y se introducen los conceptos importantes en la construcción de bases de datos y aplicaciones.

7.1 Consideraciones orientadas al usuario

Para nadie envuelto en el campo del desarrollo de software es hoy novedad que el cliente o usuario es el protagonista principal, alrededor del cual gira toda la aplicación a desplegarse, buscándose siempre su satisfacción para el éxito de la misma.

Es el usuario como primer elemento o entidad, el que tiene la ultima palabra en cuanto a la aceptación exitosa de todo el esfuerzo puesto a través del desarrollo. Por eso es importante tener en consideración algunos puntos que señalare como producto de la experiencia propia y de otros, que considero se deben poner en práctica a lo largo del proyecto.

7.1.1 El Usuario en Control

Tengamos presente que el usuario debe siempre sentir que él esta en control de lo que pasa en la pantalla. Los usuarios deberían siempre sentir que son ellos los que siempre inician las acciones, antes que reaccionar a los caprichos de la computadora. Si vamos a proveer un alto nivel de automatización en la aplicación, asegurémonos de que el usuario tiene el control sobre el proceso de automatización.

Recordemos que los usuarios son individuos, y ellos tienen preferencias y necesidades. Es importante proporcionar una manera de personalizar la aplicación. Por ejemplo veamos cuan fácil es personalizar la interface Windows 95/98/NT. La habilidad para cambiar fonts, colores, iconos, etc. nos hace sentir a Windows como una experiencia personal. Muchos de estos atributos dentro de Windows están disponibles al desarrollador. Tomemos ventaja de estos y habilitemos a la aplicación para que siga los esquemas de colores y selecciones de fonts del resto del sistema. Si no lo hacemos, la aplicación aparentara ser rígida e inflexible.

Hagamos a la aplicación comunicativa. Una buena aplicación informa que esta haciendo, o en que estado o modo se encuentra. Por ejemplo en Microsoft Word, cuando estamos en modo overwrite las letras OVR aparecen en la barra de estatus.

Nuestras aplicaciones deberían ser los más interactivas posibles, estas deberían ser sensibles y no dejar al usuario preguntándose que esta pasando.

7.1.2 Dirección

Habilitemos al usuario a manipular los objetos en su entorno. La frase "Un dibujo es mejor que mil palabras" es más verdadera cada día. Dejemos los tecnicismos de lado, es más fácil recordar la apariencia de algo que la sintaxis del comando de ese algo. Diseñemos el software intuitivamente visible. Dejemos que los usuarios vean como las acciones que ellos hacen afectan los objetos en la pantalla.

Uno de las maneras más directas como los usuarios pueden interactuar con una computadora es a través del uso de metáforas. Son las metáforas las que han hecho a la Macintosh una computadora popular y parcialmente responsable de su éxito. Por ejemplo el concepto de folder tiene mas sentido que un directorio o archivo. Para nosotros que estamos en el mundo del negocio podemos comprender un "gabinete de archivos", "archivos" dentro de este, y "documentos" dentro de los folders. Esto tiene perfecto sentido para nosotros y hace la transferencia de estos conceptos a la computadora más fácil. Las metáforas soportan el concepto de reconocimiento-del-usuario en vez de recolección-del-usuario. Los usuarios pueden recordar usualmente los significados asociados con un objeto mas fácilmente que con un comando.

7.1.3 Consistencia

Este es uno de los aspectos más importantes del desarrollo de una aplicación visual. La consistencia es una de las principales razones de aceptación del producto final. Si todas las aplicaciones son consistentes en la manera como actúan con el usuario y como presentan o capturan los datos, el usuario puede dedicar mas tiempo a la ejecución de sus tareas y no en aprender las diferencias en la manera que la aplicación interactúa con él.

Esta consistencia se expande a diversas áreas que deberíamos tener en cuenta:

- Asegurémonos que la aplicación actúe de manera similar al sistema operativo Windows. El usuario entonces puede fácilmente transferir las habilidades que ha aprendido en Windows a la aplicación
- Asegurémonos que el producto es consistente con sigo mismo. Si por ejemplo la aplicación soporta Ctrl+C para cierta operación en una pantalla, no usemos un paradigma diferente en otra pantalla (tal como Ctrl+D).
- Asegurémonos que nuestras metáforas sean consistentes. Usemos iconos que el usuario pueda diferenciar con facilidad o que pueda asociar para llevar a cabo operaciones relacionadas o similares.

Usemos consistencia en todo momento cuando usemos menús, barra de herramientas, y otros controles comunes.

7.1.4 Arrepentimiento

Dejemos que el usuario explore la aplicación, y captemos todas las acciones para registrar todas aquellas que necesitan ser reversibles, corregidas o canceladas. Los usuarios necesitan saber de antemano que las acciones que están a punto de efectuar son destructivas. La gente es propensa a cometer errores, la aplicación tiene que pedir una confirmación de todas las acciones destructivas, usando el teclado o el mouse, en caso de que estas fueran iniciadas por error.

Asimismo la aplicación debería ser capaz de habilitar al usuario a desactivar algunas de las confirmaciones para ciertas acciones, una vez que este se familiariza con la aplicación.

7.1.5 Retroalimentación

La cosa que más molesta a un usuario es ver que la computadora esta frente de el y que aparentemente no esta haciendo nada. El usuario no sabe lo que la computadora esta haciendo, y la computadora no indica nada. La regla es permitirle al usuario saber que esta pasando, proveyéndole retroalimentación basados en tiempo. Podemos usar una combinación de elementos visuales así como auditivos, para hacer que el usuario conozca que la computadora esta procesando algo.

Es importante que la retroalimentación este cerca al punto donde el usuario este trabajando. Si por ejemplo ellos están ingresando datos en la parte superior de la pantalla, no presentemos un mensaje de error en la parte baja (a no ser que la aplicación tenga una barra de estatus). Podemos también cambiar la forma del cursor para indicar una condición (como el reloj de arena). Los usuarios no toleraran una computadora muerta por mas de un par de segundos.

7.1.6 Estética

La aplicación debe también ser visualmente agradable al usuario. Esto significa varias cosas. Además de usar los colores para las pantallas (de tal manera que la aplicación se combine con el entorno), el diseño de la pantalla misma es muy importante. La posición de los objetos determina cuan usable es la pantalla, así como la cantidad de elementos en la pantalla misma. Cuando sea posible usemos la regla del siete. Demos al usuario solo siete posibilidades (mas o menos dos). Este número de cinco a nueve elecciones viene de investigaciones de la cantidad de cosas que el cerebro humano puede manipular al mismo tiempo. Con mas de nueve cosas la gente tiende a confundirse y a sufrir de sobrecarga cerebral.

7.1.7 Simplicidad

Este es el último principio de diseño. La aplicación deberá ser muy fácil de aprender, de usar y muy amigable. Debemos balancear dos cosas que son contrarias: una es el acceso a toda la funcionalidad e información en la aplicación; la segunda, mantener simple la interface y el uso del producto. Una buena aplicación balancea estos dos principios y se sitúa en la parte media.

No llenemos las pantallas de palabras. Cuando usemos las etiquetas en los campos de entrada de datos pongamos "Nombre" y no "Nombre del Cliente". Tratemos de usar el menor número de palabras que comuniquen el significado correcto. Usemos el principio de muestra progresiva, el cual consiste en presentar los datos según se necesiten. Por ejemplo en un programa de directorio telefónico, se podría mostrar el nombre y el número de la persona en la pantalla inicial y el usuario tendrá que presionar un botón que muestra la información restante de la persona.

También tengamos presente que al momento del despliegue es necesario desplegar un producto que sea fácil de instalar, flexible y amigable, de tal manera que el proceso se complete como una consecuencia de acciones ejecutadas a través de pantallas sucesivas, pidiendo información e informando los pasos que se están efectuando.

7.2 Consideraciones orientadas a la Aplicación Cliente/Servidor

El desarrollo de bases de datos y aplicaciones cliente/servidor es similar en muchas maneras al desarrollo de otros tipos de software, pero hay algunas distinciones importantes y retos que se deben considerar. La metodología presentada en esta

sección debería ser usada como una guía que pueda ser adaptada para satisfacer las necesidades especificas del negocio.

Una base de datos por si misma es difícilmente de utilidad; es la aplicación la cual usualmente provee los mecanismos e interfaces para que los usuarios finales accesen dicha base de datos. Una aplicación de base de datos cliente/servidor consiste de elementos cliente y elementos servidor.

En la actualidad los sistemas de administración de bases de datos relacionales (RDBMS, por sus siglas en inglés) son los que tienen el dominio de aceptación en el mercado, y estos proveen un proceso rápido de transacciones y compartimiento de datos en un entorno sea usuario único o multi-usuario. En su núcleo, tenemos la tecnología servidor que ofrece soporte transparente a través de redes heterogéneas. Estos RDBMS corren en Windows 95/98, Windows NT, Novell NetWare, y muchas implantaciones del sistema operativo Unix.

Antes de discutir el proceso de desarrollo, es importante definir alguna terminología. Los términos "cliente" y "servidor" son usados conceptualmente y no físicamente. En la práctica, el cliente es una computadora personal corriendo bajo Windows y el servidor es una estación de trabajo o una plataforma de alta performance corriendo un sistema operativo multiusuario tal como Unix o Windows NT. Sin embargo esto no es necesariamente el caso. Algunas bases de datos permiten a una plataforma ser cliente y servidor. En la mayoría de los casos, será más eficiente en términos de costos correr elementos clientes en computadoras personales de bajo costo y correr los elementos servidor en maquinas más sofisticadas.

Los elementos server incluyen el motor de la base de datos, la base de datos, y procesos auxiliares que se ejecutan en el servidor tales como funciones definidas del usuario (UDFs, por sus siglas en inglés), filtros BLOBS, triggers, y procedimientos

almacenados. Todos estos elementos servidor son parte de la base de datos o están estrechamente relacionadas a esta, y corren en la plataforma del servidor.

Los elementos cliente corren en el cliente, e incluyen comandos SQL y DSQL anidados o aplicaciones API que proveen la interface de usuario (con el sistema relacionado de presentación GUI, tal como Windows), compartición media tales como motores de bases de datos o drivers ODBC, drivers de bajo-nivel de red para protocolos de comunicación tales como TCP/IP. Otros elementos cliente incluyen herramientas tales como Windows ISQL (para definición de datos) y Servidor Manager (para la administración de la base de datos), y herramientas de desarrollo de aplicaciones.

7.3 Desarrollo de la Base-de-Datos/Aplicación

El término "desarrollo de la base-de-datos/aplicación" se refiere a ambos, construir la base de datos con los elementos servidor asociados y construir la aplicación con los asociados elementos clientes.

El objetivo del desarrollo de una Base-de-Datos/Aplicación cliente/servidor es construir un artefacto el cual satisface las necesidades a largo plazo de los usuarios. Mientras el objetivo es obvio, es importante no apartar la vista del mismo a lo largo de las complejidades y comúnmente conflictivas demandas del proceso de desarrollo. Las tres fases primarias del desarrollo Base-de-Datos/Aplicación son:

- Diseño
- Implantación
- Despliegue y Mantenimiento

En cada una de estas fases hay tareas a realizar para la base de datos y para la aplicación, como se ilustra en la figura 7-1.

CICLO DEL DESARROLLO DE LA BASE DE DATOS Y LA APLICACIÓN

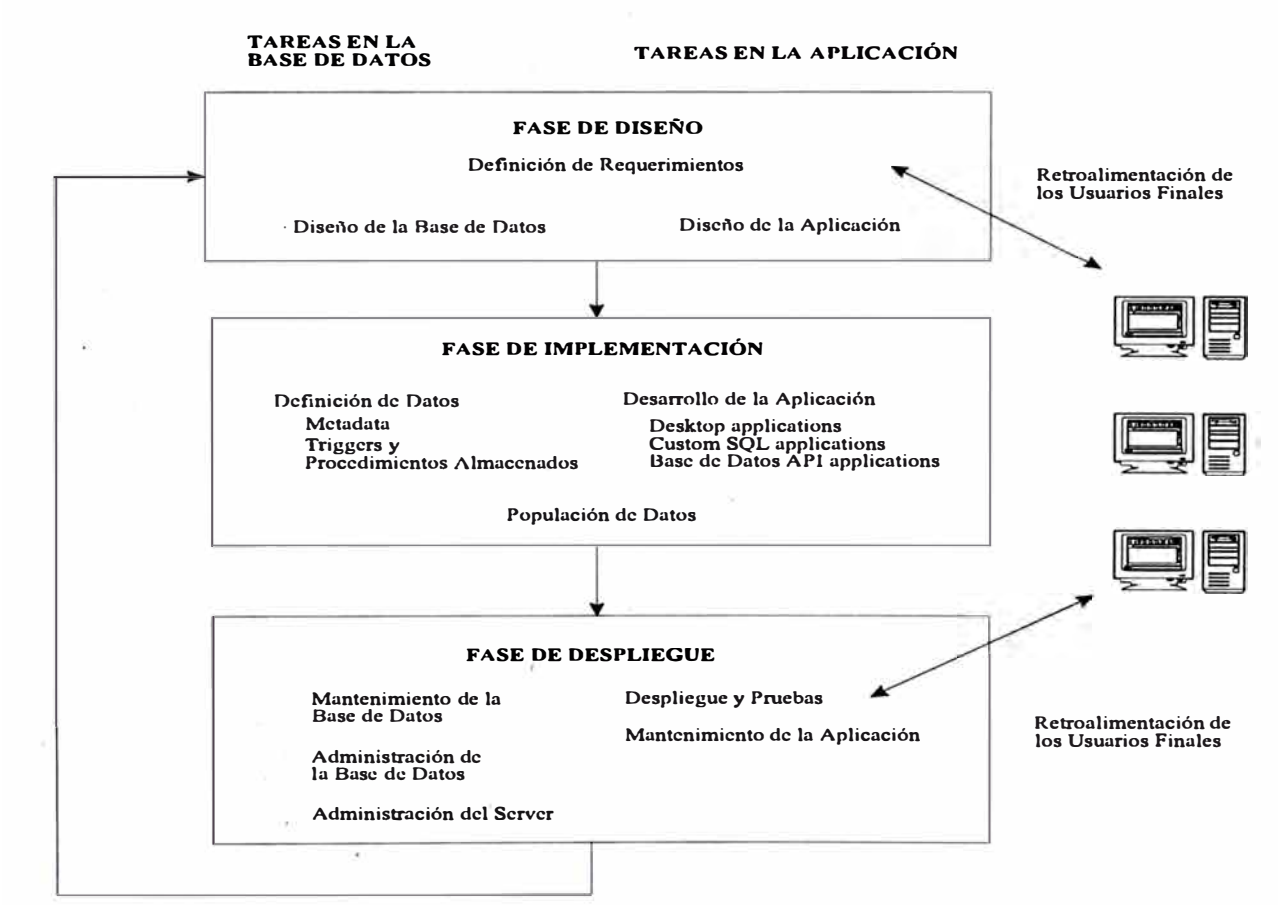


Figura 7-1.Desarrollo de la Base de Datos Aplicación

Dependiendo del tamaño y alcance del proyecto de desarrollo, las tareas de la base de datos y la aplicación pueden ser llevadas a cabo por diferentes individuos o por el mismo individuo. Comúnmente, un individuo o equipo será responsable por las tareas de la base de datos, y otro individuo o equipo será responsable por las tareas de la aplicación.

Cuando las responsabilidades del desarrollo son así divididas, es importante delinear claramente en la fase de diseño que funciones serán ejecutadas por la base de datos (típicamente en el servidor) y cuáles por la aplicación (típicamente en el cliente). Usualmente, las líneas funcionales están claramente definidas. Pero procesos de la base de datos tales como UDFs y procedimientos almacenados pueden algunas veces realizar funciones que también pueden ser realizadas por la aplicación cliente. Dependiendo en el despliegue de la configuración expectada, en la fase de diseño se puede asignar tales funciones a cualquiera cliente o servidor.

Es también importante comprender que el desarrollo Base-de_datos/Aplicación es por su naturaleza un proceso iterativo. Los usuarios pueden no comprender completamente sus propias necesidades, o pueden definir necesidades adicionales según se efectúe el desarrollo. También, cambiando las necesidades del negocio cambiara los requerimientos a lo largo del tiempo.

7.3.1 Fase de Diseño

La fase de diseño empieza con la definición de requerimientos. En consulta con usuarios finales conocedores, se define las especificaciones funcionales para la base de datos y para las aplicaciones.

Algunas funciones pueden ser realizadas por cualquiera la base de datos o por las aplicaciones; por ejemplo una función matemática de transformación podría ser realizada por cualquiera por un modulo del programa de

aplicación cliente o por una función definida para el usuario en el servidor. Se debe tratar de determinar cual seria la más conveniente configuración de despliegue y asignar la función adecuadamente. Por ejemplo, si se espera que las plataformas del cliente sean computadoras personales de bajo nivel, y se espera que la plataforma del servidor sea una estación de trabajo de nivel alto, considérese las funciones intensivas de calculación en el servidor con UDFs o procedimientos almacenados. Si la configuración de hardware cambia, entonces es posible mover las funciones entre el cliente y el servidor en una iteración posterior.

El diseño de la base de datos es una tarea compleja que empieza con el modelamiento de datos. El modelamiento de datos envuelve el análisis de los datos y procesos organizacionales y trasladar estos a componentes de la base de datos. Para el modelamiento complejo de bases de datos, se puede usar una herramienta CASE para ser asistido en la construcción de los modelos entidad-relación. Los detalles de modelamiento y diseño de bases de datos están basados en la teoría relacional y están mas allá del alcance de este estudio.

El diseño de la aplicación esta basado en los requerimientos definidos previamente. A partir de allí este se convierte en una larga tarea conceptual de trasladar los requerimientos a las herramientas especificas de desarrollo de la aplicación. Si las aplicaciones están siendo mantenidas (upsizing) entonces mucho del diseño de las aplicaciones puede ya estar hecho, o se puede requerir solo modificaciones menores.

7.3.2 Fase de Implantación

En la fase de implantación, se construye la base de datos y las aplicaciones concebidas en la fase de diseño.

Para la base de datos esta fase envuelve la creación de la base de datos en sí misma y sus objetos constituyentes: tablas, vistas, columnas, triggers, procedimientos almacenados, etc.

Generalmente para crear una base de datos y sus componentes los RDBMS usan una implantación de SQL. La construcción de la base de datos envuelve la definición de los datos, por lo cual los RDBMS proveen una set de declaraciones llamadas DDL (por sus siglas en ingles de Data Definition Languaje).

Una base de datos consiste de una variedad de objetos propios de la base de datos, tales como tablas, vistas, dominios, procedimientos almacenados, triggers, etc. Los objetos de la base de datos contienen toda la información acerca de la estructura de la base de datos y los datos. Debido a que estas encapsulan la información referente a los datos, los objetos de la base de datos son algunas veces llamados "metadata".

Usualmente una base de datos es un archivo simple el cual comprende todos la "metadata", y los datos en la base de datos, como se muestra en figura 7-2 en la siguiente página.

7.3.3 Fase de despliegue

Esta es la fase en la cual los usuarios tienen la oportunidad de interactuar con la ultima versión creada del artefacto en la última iteración. En la fase de despliegue el prototipo de la Base-de-Datos/Aplicación es puesta a prueba. El producto creado en la fase de implantación es entregado a los usuarios. Inevitablemente, habrá sugerencias y requisiciones de los usuarios para modificaciones o adicionales características. Según estas características sean

implementadas en iteraciones subsiguientes, la Base-de-datos/aplicación se acerca cada vez más a ser completada. El despliegue del prototipo base-de-datos/aplicación nos ayuda a dirigir subsiguiente análisis y desarrollo.

Durante esta fase, los desarrolladores deben depurar los programas, respondiendo a las sugerencias de los usuarios, y adicionar nuevas características. Desde que los jueces que tienen la ultima palabra de la eficacia de una aplicación son los usuarios, los desarrolladores de la aplicación deben estar preparados para incorporar cambios a las aplicaciones debido a la dinámica de las necesidades del mundo real, así como a mejoras en general. Comúnmente los cambios en las aplicaciones requerirán cambios en la base de datos, y así mismo, cambios en la base de datos pueden requerir cambios en la aplicación.

En algún punto, una aplicación desplegada pasa de ser una base para prueba a una base-de-datos/aplicación en producción, la cual luego necesitara ser mantenida.

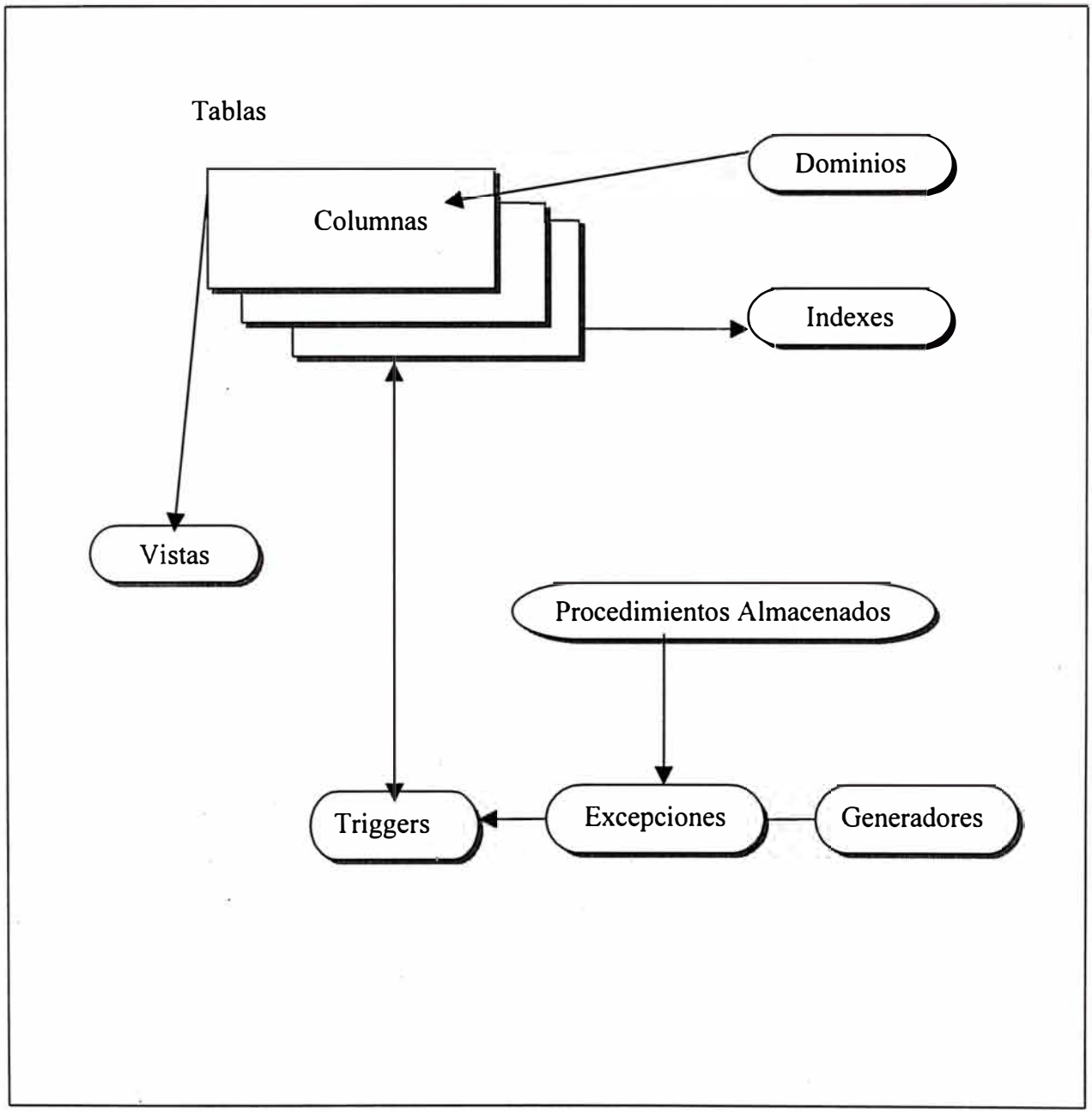


Figura 7-2.Objetos en la base de datos

CASO PRACTICO

Presentación

El prototipo que a continuación se presenta tiene por finalidad mostrar las bondades del enfoque de desarrollo usado por el autor. Dicho prototipo esta desarrollado para la plataforma Windows 95/98/NT, y ha sido construido integramente en C++. La base de datos relacional usada en la modalidad Stand-Alone es Paradox a traves del driver estandar ODBC correspondiente, y en la modalidad Cliente/Servidor la implementación esta hecha para la base de datos Interbase de Borland.

El prototipo es la respuesta dada para satisfacer la necesidad de captación, registro y consultas de ordenes de compra (emitidas por los clientes) de una manera confiable, agil y amigable tanto para los usuarios de ventas así como para los clientes externos e internos que ellos sirven.

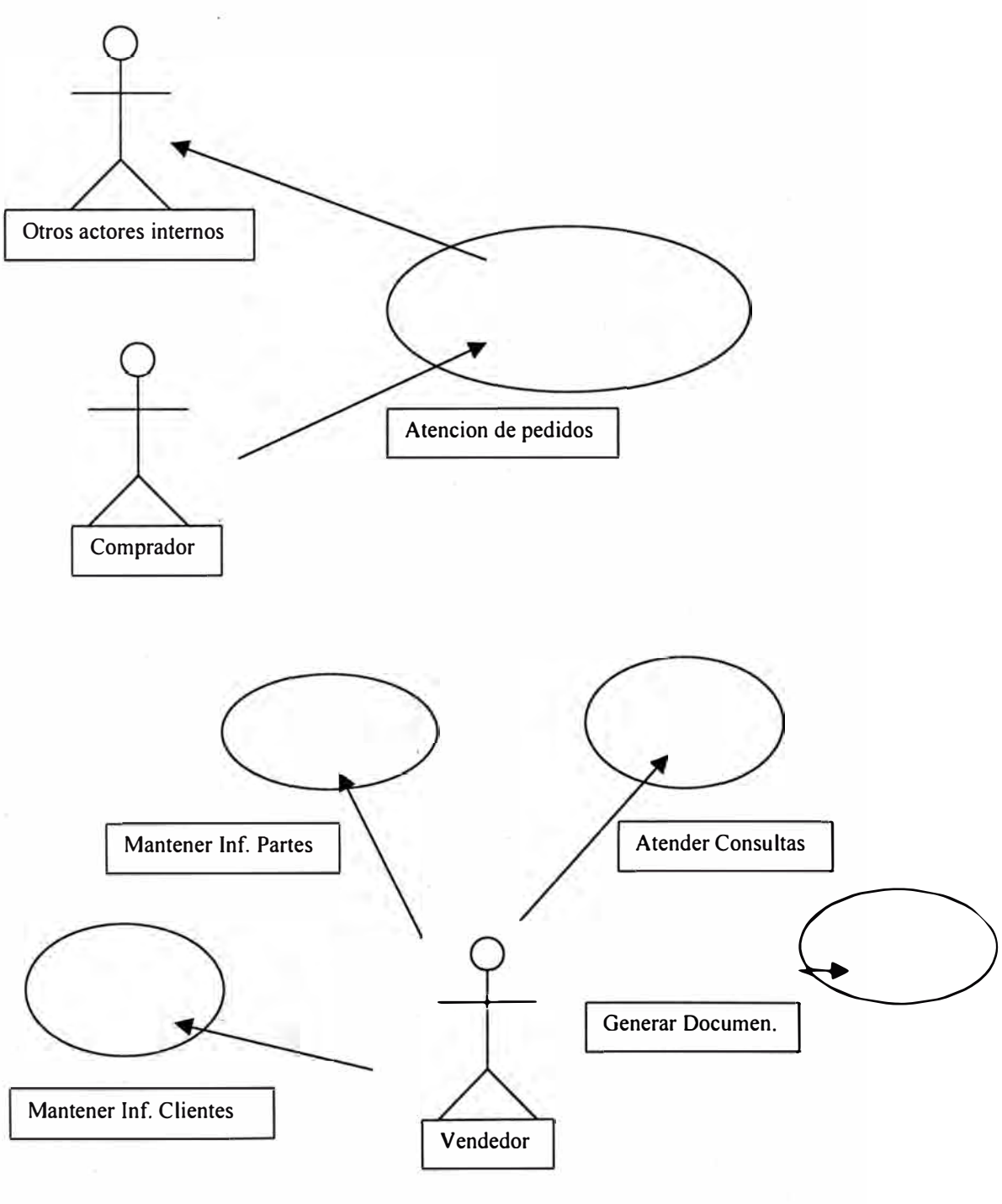
Puntualizamos, que actualmente existe redundancia de esfuerzos por parte del personal de nuestra compañia en el afan de satisfacer consultas en linea, solicitadas por los clientes despues que ellos han emitido y se ha aceptado una orden de compra.

Dentro de las consultas por parte de los clientes estan aquellas relacionadas con Ordenes de compra y facturación. Generalmente los clientes quieren tener en su poder una copia de dichos documentos para sus registros o para ir procesando sus compras internamente, por lo que se hace necesario generar copias de los mismos para ser enviados via fax.

La necesidad del servicio expuesto, incluye tanto a los clientes que emiten ordenes telefonicamente, por correo, via fax, o tratando personalmente con representantes del departamento de ventas en las instalaciones de la compañia.

Lo declarado anteriormente, nos lleva a la necesidad de desarrollar un sistema para dar soporte al sistema de ventas en la red computarizada de la empresa. Dicho sistema ha sido bautizado con el nombre sistema de Manipuleo Rápido de Ordenes de Compra (QOEH por sus siglas en inglés Quick Order Entry Handling).

DIAGRAMA DE CASOS DE USO



CASO DE USO 17	Vender Mercaderias
Objetivo dentro del Contexto	Cliente expide requerimiento a nuestra empresa, espera envio de mercaderias, y acepta que sean cargadas a su cuenta o facturadas
Alcance & Nivel	Empresa y Resumen
Precondiciones	Tenemos registrada toda la informacion del cliente
Condicion Final del Exito	Cliente tiene su mercaderias y nosotros tenemos el valor monetario de las mercaderias.
Condicion Final de Falla	Nosotros no enviamos las mercaderias, el cliente no tiene que pagar su valor
Actor Primario	Cliente, cualquier agente actuando por el cliente
Disparador O Trigger	Llega un requerimiento de compra

8	Cliente paga la factura
Paso	**Accion de Division o Bifurcacion**
3a	Compañia no dispone de uno de los items ordenados: 3a.1 Renegociar orden
4a	Cliente paga directamente con tarjeta de credito 4a.1 Aceptar pago por tarjeta de credito, continuar (CU 23)
4b	Cliente desiste de la compra 4b.1 Se cancela el proceso de venta
6a	Cliente cambia los requerimientos 6a.1 Cambio de algun item, continuar (CU 40) 6a.2 Anulacion de orden, continuar (CU 41)
8a	Comprador devuelve la mercaderia 8.a1 Manejar mercaderia devuelta, continuar (CU 25)

INFORMACION RELACIONADA	<< Vender Mercaderias >>
Prioridad	Alta
Performance	4 minutos por orden,
Frecuencia	600 / dia
Canal a Actor Primario	Telefono, Fax, Interactivo
Actores Secundarios	Cia. Tarjeta de credito, Banco, Servico envios
Canales a Actores Secundarios	

ASUNTOS ABIERTOS	Que pasaria si la informacion de la tarjeta esta errada? Que pasaria si el cliente paga de diversas modalidades?
Release	1,0
Caso de uso Superordinado	Transacciones de cliente (caso de uso 5)
Caso de uso subordinado	Crear orden (caso de uso 15) Tomar pago con tarjeta de credito (caso de uso 23) Manejar mercaderias devueltas (caso de uso 25)

CASO DE USO	<< Vender mercaderias >>	
VARIACIONES	Paso	**Accion**
	1	Cliente puede utilizar: * Entrada telefonica * Fax, o * Por correo * Personalmente
	8	Cliente puede pagar: * Contado * Cheque *Tarjeta de crédito * Contra entrega

CLASES IDENTIFICADAS DENTRO DEL SISTEMA

- CLASE CLIENTE
- CLASE ORDEN
- CLASE PROVEEDOR
- CLASE ITEM
- CLASE PARTE
- CLASE VENDEDOR

CLASES Y SUS ESTRUCTURAS

CLIENTE
Codigo
Compañia
Direccion 1
Direccion 2
Ciudad
Estado
Apartado Postal
Pais
Telefono
Fax
Impuesto
Contacto
Fecha ultima factura

ORDEN
OrdenNº
ClienteNº
FechaVenta
FechaDespacho
EmpleadoNº
EnvioAlContacto
EnviaADirecc1
EnviaADirecc2
EnviaACiudad
EnviaAEstado
EnviaAPais
EnviaATelefono
MetodoEnvio
OrdenCompraNº
Terminos
MetodoPago
TotalItems
Impuesto
Flete
MontoPagado

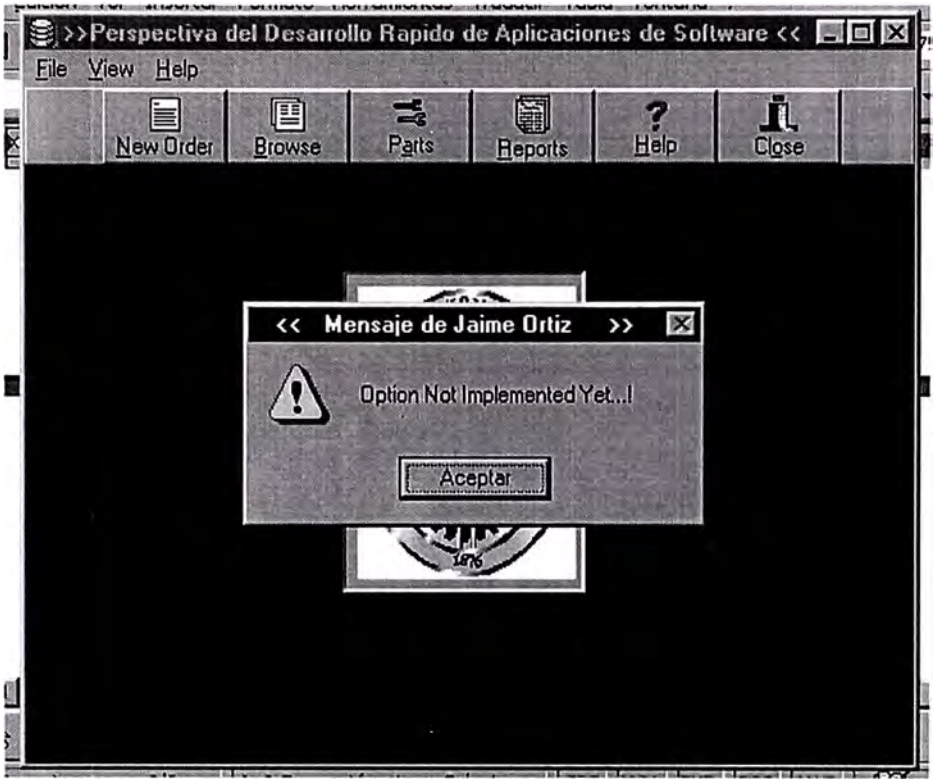
PROVEEDOR
Codigo
Nombre
Direccion 1
Direccion 2
Ciudad
Estado
Apartado Postal
Pais
Telefono
Fax
Preferencia
Contacto

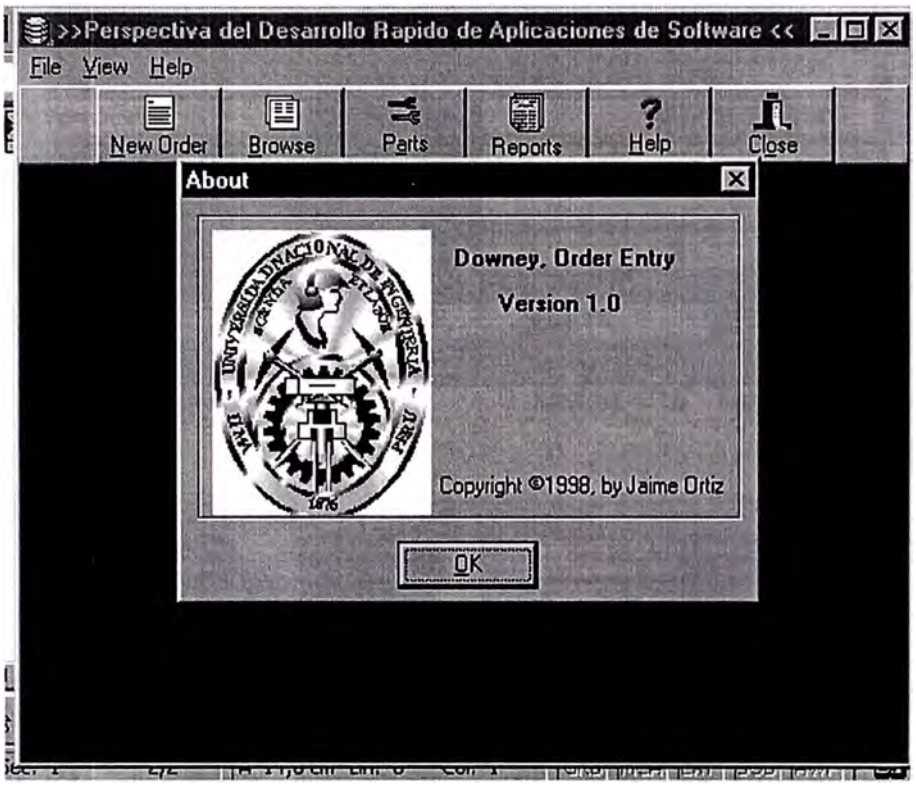
PARTE
ParteNº
VendedorNº
Descripcion
Disponibles
EnOrdenes
Costo
PrecioLista

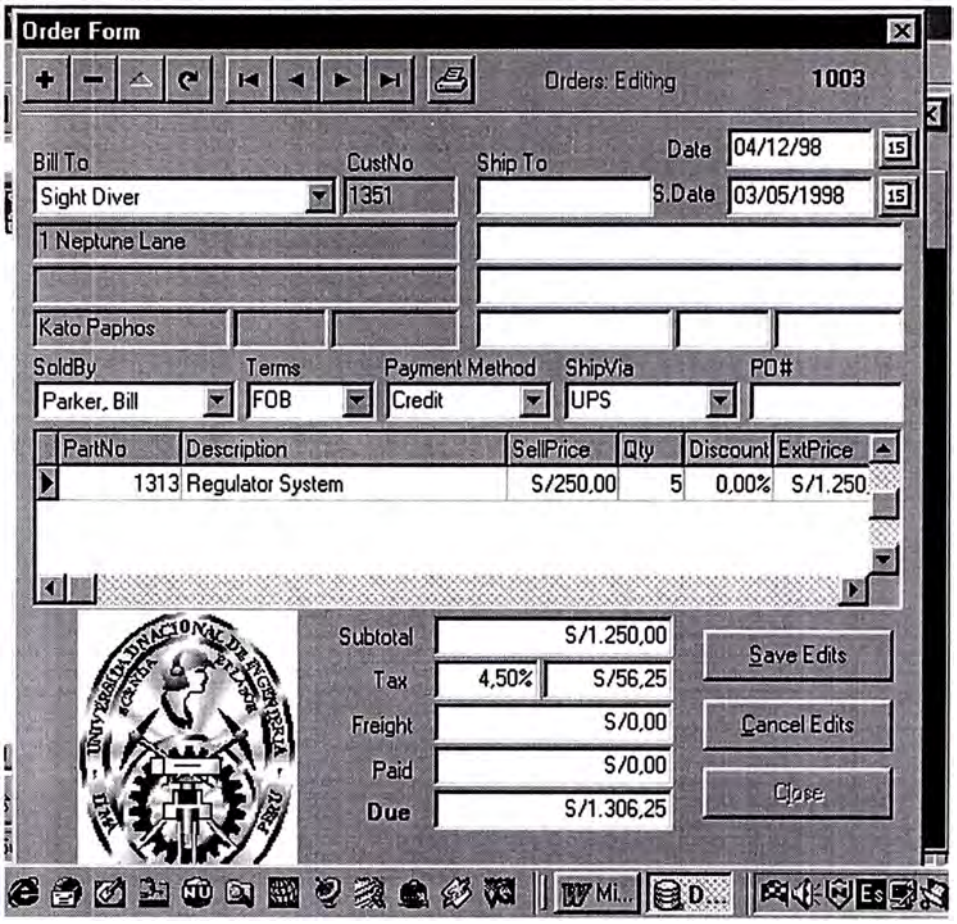
ITEM
OrdenNº
ItemNº
ParteNº
Cantidad
Descuento

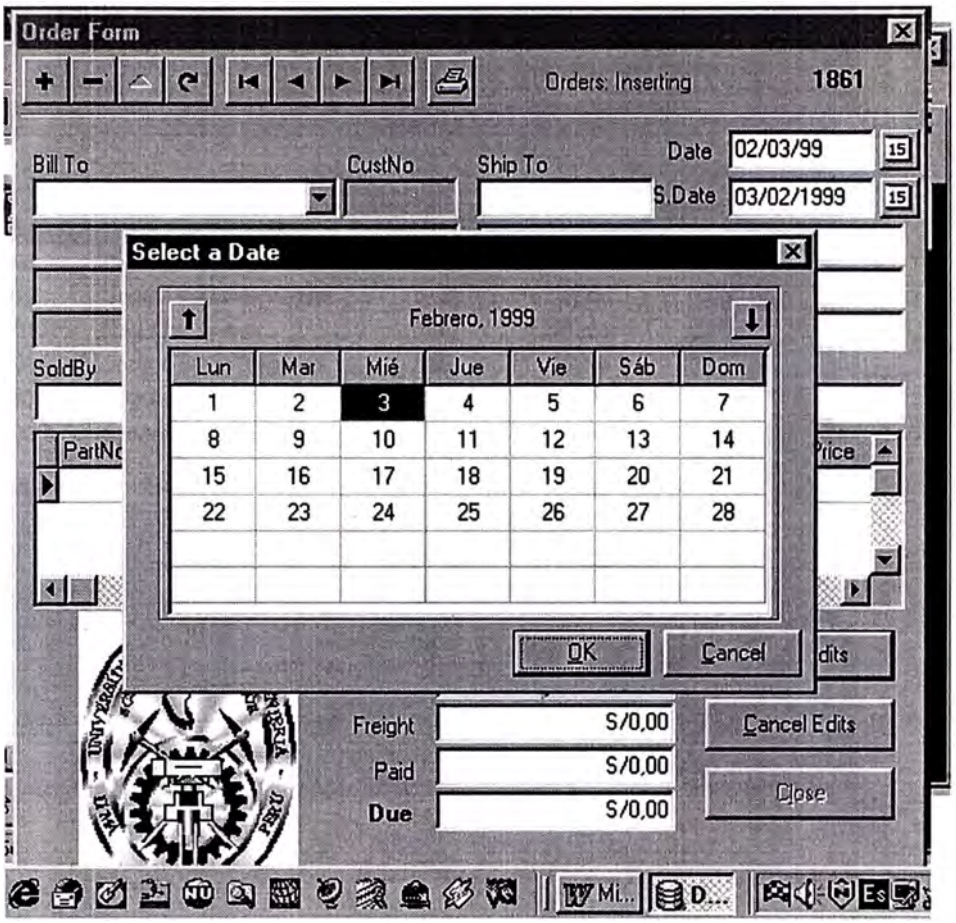
VENDEDOR
EmpNº
ApellidoPaterno
PrimerNombre
ExtTelefonica
FechaIngreso
Salario

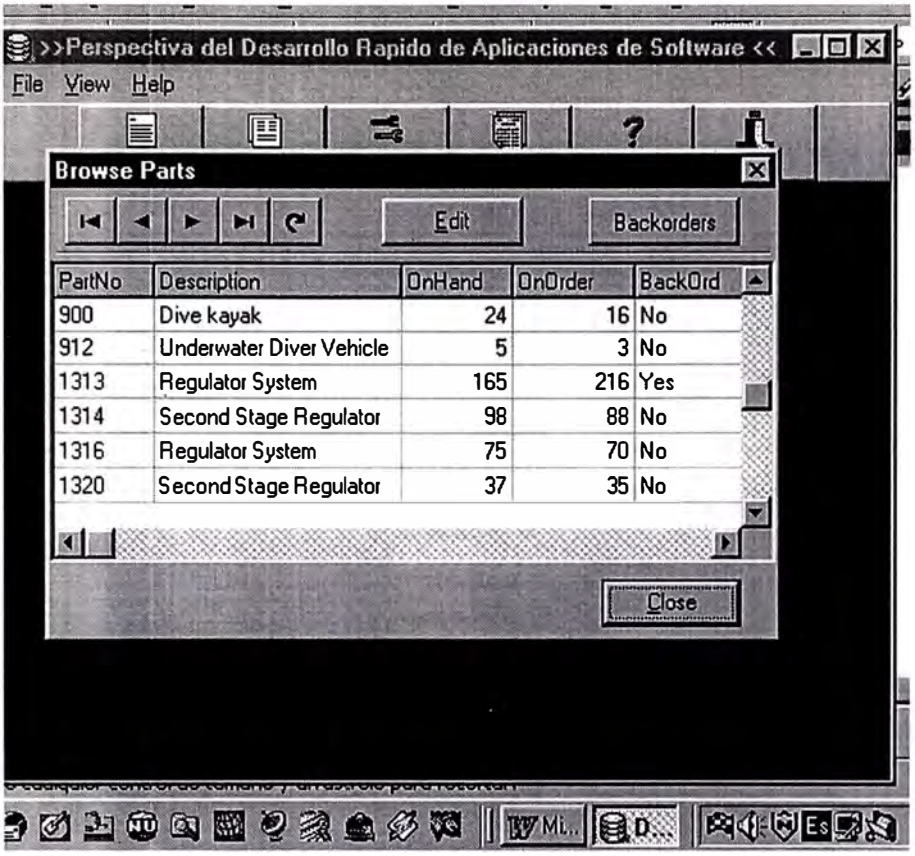

Mi PC
Bandeja de entrada
Internet Explorer
>>Perspectiva del Desarrollo Rapido de Aplicaciones de Software <<
File View Help
New Order
Browse
Parts
Reports
Help
Close
Inicio
Downey G...
07:40 PM

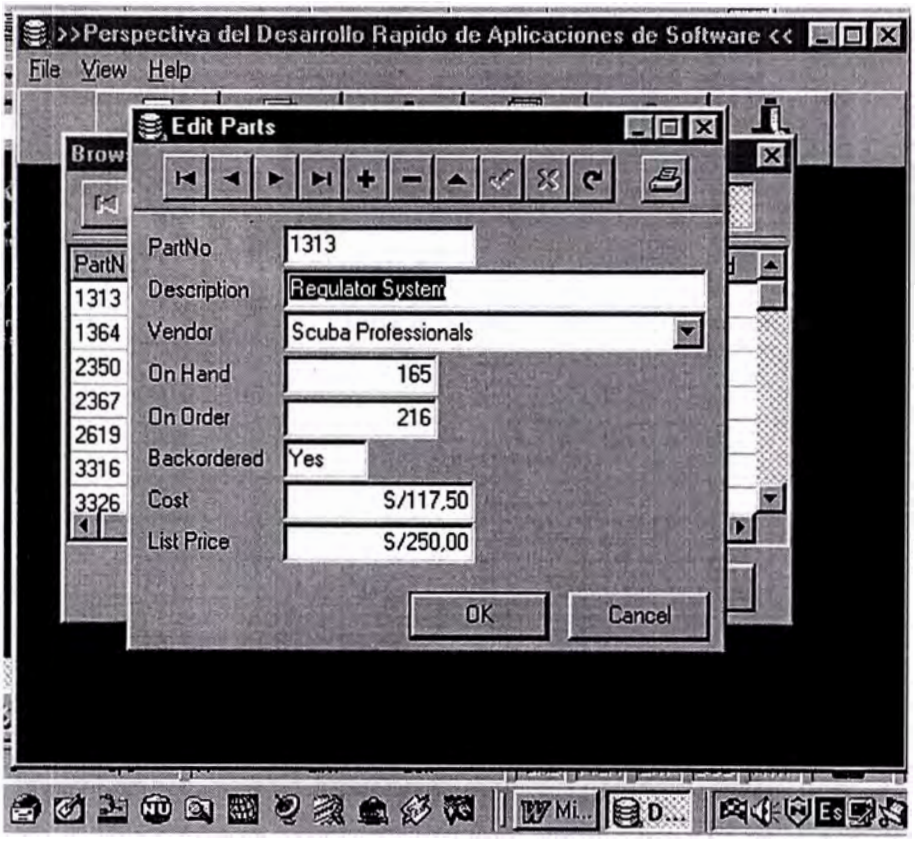

>>Perspectiva del Desarrollo Rapido de Aplicaciones de Software <<
File View Help
New Order
Browse
Parts
Reports
Help
Close
<< Mensaje de Jaime Ortiz >>
Option Not Implemented Yet...!
Aceptar

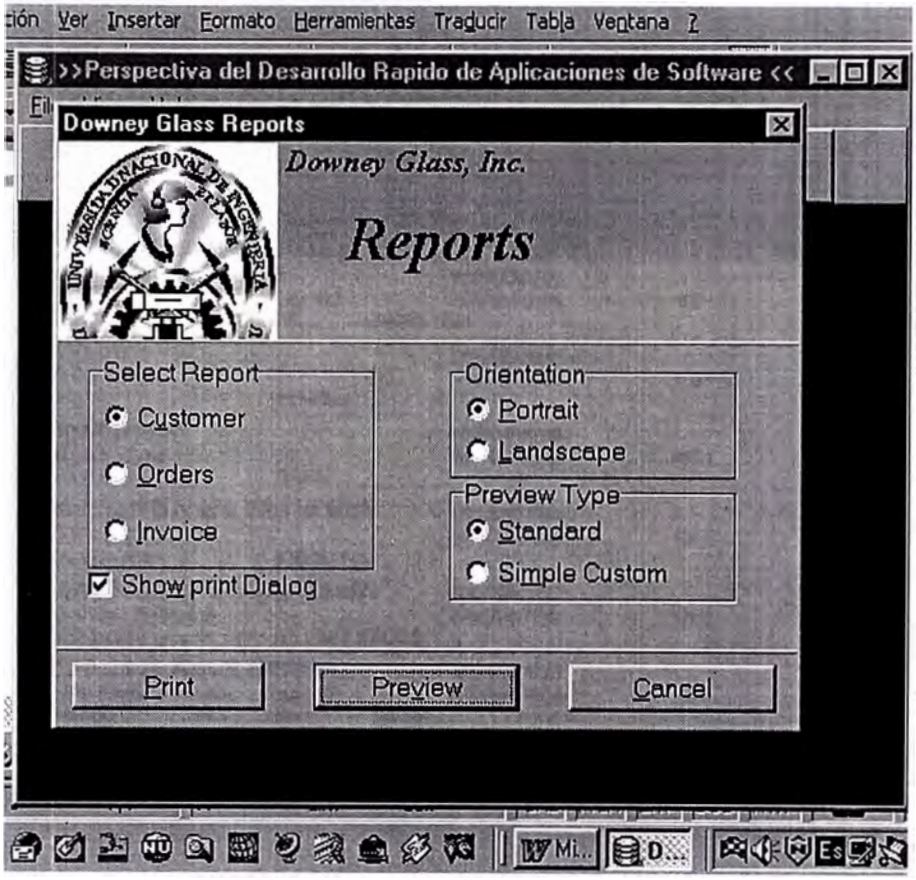

>>Perspectiva del Desarrollo Rapido de Aplicaciones de Software <<
File View Help
New Order
Browse
Parts
Reports
Help
Close
About
Downey, Order Entry
Version 1.0
Copyright ©1998, by Jaime Ortiz
OK

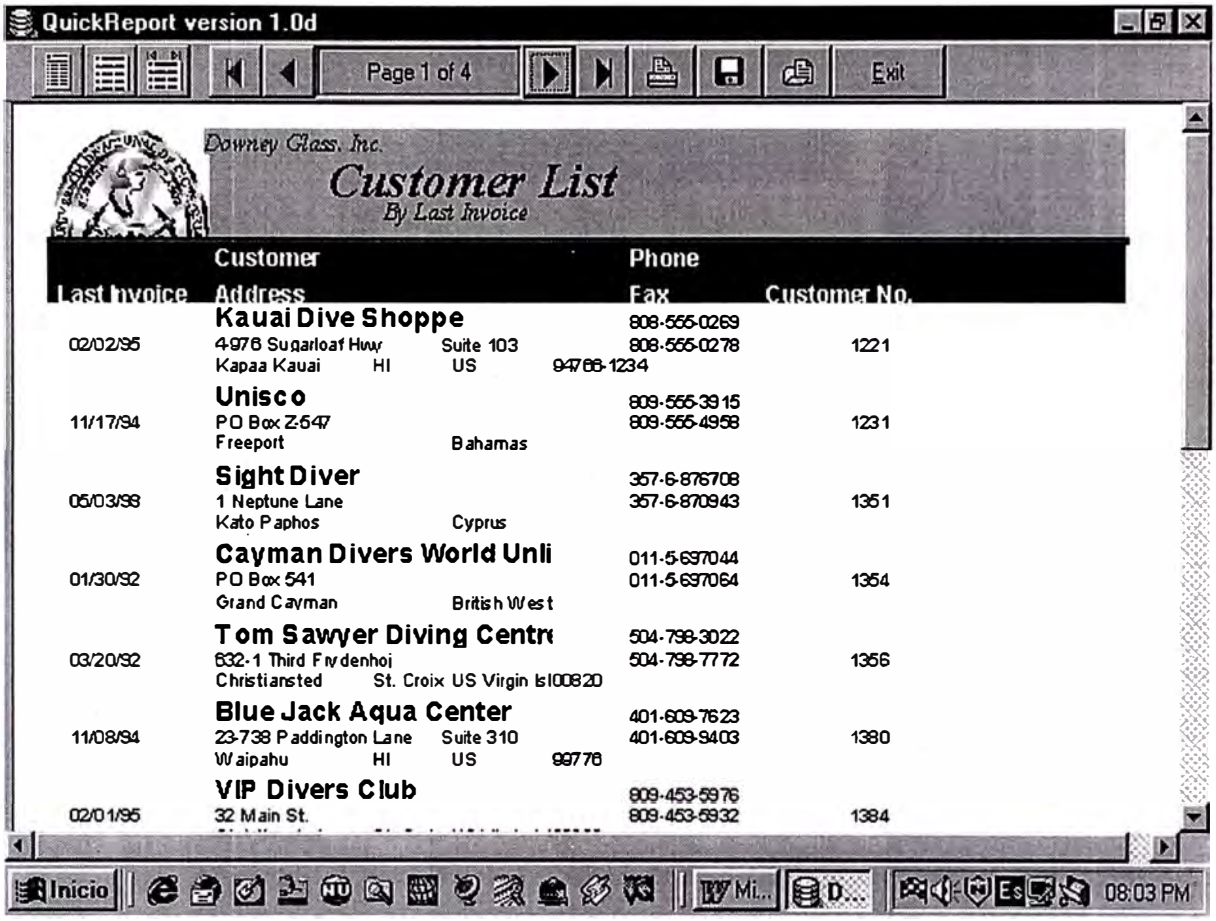

Order Form
Orders: Editing
1003
Bill To
CustNo
Ship To
Date
04/12/98
Sight Diver
1351
S.Date
03/05/1998
1 Neptune Lane
Kato Paphos
SoldBy
Terms
Payment Method
ShipVia
PO#
Parker, Bill
FOB
Credit
UPS
PartNo
Description
SellPrice
Qty
Discount
ExtPrice
1313
Regulator System
S/250,00
5
0,00%
S/1.250,
Subtotal
S/1.250,00
Tax
4,50%
S/56,25
Freight
S/0,00
Paid
S/0,00
Due
S/1.306,25
Save Edits
Cancel Edits
Close

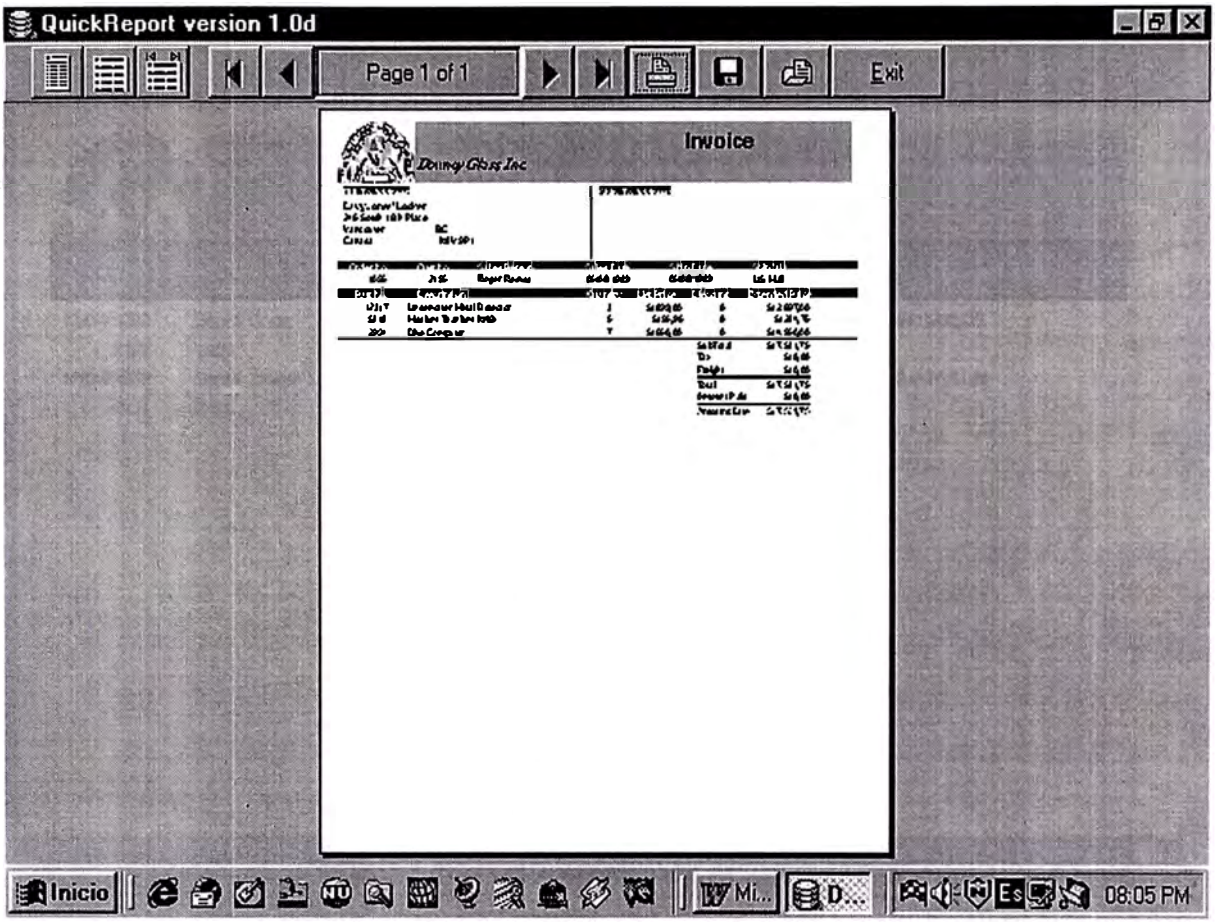

Order Form
Orders: Inserting
1861
Bill To
CustNo
Ship To
Date
02/03/99
S.Date
03/02/1999
Select a Date
Febrero, 1999
Lun Mar Mié Jue Vie Sáb Dom
1 2 3 4 5 6 7
8 9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
OK
Cancel
SoldBy
Freight
S/0,00
Paid
S/0,00
Due
S/0,00
Cancel Edits
Close

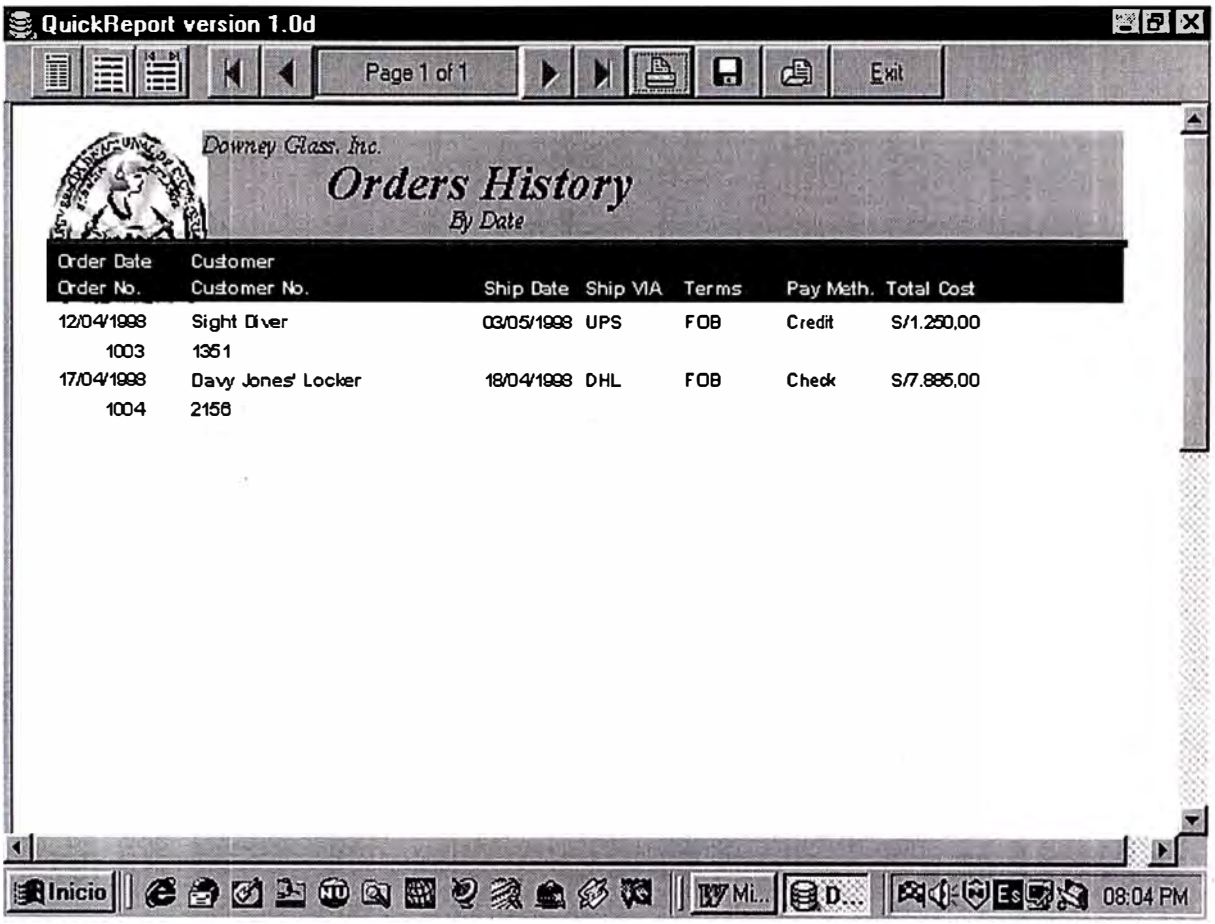

>>Perspectiva del Desarrollo Rapido de Aplicaciones de Software <<
File View Help
Browse Parts
Edit
Backorders
PartNo Description OnHand OnOrder BackOrd
900 Dive kayak 24 16 No
912 Underwater Diver Vehicle 5 3 No
1313 Regulator System 165 216 Yes
1314 Second Stage Regulator 98 88 No
1316 Regulator System 75 70 No
1320 Second Stage Regulator 37 35 No
Close


>>Perspectiva del Desarrollo Rapido de Aplicaciones de Software <<
File View Help
Edit Parts
PartNo 1313
Description Regulator System
Vendor Scuba Professionals
On Hand 165
On Order 216
Backordered Yes
Cost S/117,50
List Price S/250,00
OK
Cancel
PartN
1313
1364
2350
2367
2619
3316
3326


ción Ver Insertar Formato Herramientas Traducir Tabla Ventana ?
>>Perspectiva del Desarrollo Rapido de Aplicaciones de Software <<
Downey Glass Reports
Downey Glass, Inc.
Reports
Select Report
Customer
Orders
Invoice
Show print Dialog
Orientation
Portrait
Landscape
Preview Type
Standard
Simple Custom
Print
Preview
Cancel


QuickReport version 1.0d
Page 1 of 4
Exit
Downey Glass, Inc.
Customer List
By Last Invoice
Customer
Phone
Last Invoice
Address
Fax
Customer No.
Kauai Dive Shoppe
808-555-0269
02/02/95
4-976 Sugarloaf Hwy
Suite 103
808-555-0278
1221
Kapaa Kauai
HI
US
94766-1234
Unisco
809-555-3915
11/17/94
PO Box Z-547
809-555-4958
1231
Freeport
Bahamas
Sight Diver
357-6-876708
05/03/93
1 Neptune Lane
357-6-870943
1351
Kato Paphos
Cyprus
Cayman Divers World Unli
011-5-697044
01/30/92
PO Box 541
011-5-697064
1354
Grand Cayman
British West
Tom Sawyer Diving Centre
504-798-3022
03/20/92
632-1 Third Frydenhoj
504-798-7772
1356
Christiansted
St. Croix US Virgin Is
00820
Blue Jack Aqua Center
401-609-7623
11/08/94
23-738 Paddington Lane
Suite 310
401-609-9403
1380
Waipahu
HI
US
99776
VIP Divers Club
809-453-5976
02/01/95
32 Main St.
809-453-5932
1384
Inicio
08:03 PM


QuickReport version 1.0d
Page 1 of 1
Exit
Invoice
Inicio
08:05 PM


QuickReport version 1.0d
Page 1 of 1
Exit
Downey Glass, Inc.
Orders History
By Date
Order Date
Order No.
Customer
Customer No.
Ship Date
Ship VIA
Terms
Pay Meth.
Total Cost
12/04/1998
1003
Sight Diver
1351
03/05/1998
UPS
FOB
Credit
S/1.250,00
17/04/1998
1004
Davy Jones' Locker
2156
18/04/1998
DHL
FOB
Check
S/7.885,00
Inicio
08:04 PM

CONCLUSIONES

1. Beneficios y Costos del Desarrollo Rápido de Aplicaciones

1.1. Beneficios

Los beneficios que trae consigo el desarrollo rápido de aplicaciones están ligados a las consecuencias (benéficas), de cumplir con el cronograma establecido de acuerdo al planeamiento del proyecto. A continuación una revisión de los beneficios más saltantes:

- En el desarrollo rápido no se prevé como única prioridad la(s) fecha(s) de despliegue, si no que se pone extremo cuidado en la performance, uso y mantenimiento del artefacto de software, con lo cual se mejora la calidad del mismo.

- El cumplimiento del cronograma desde la perspectiva del desarrollo rápido, lleva inherente el cumplimiento de los objetivos en cuanto a la calidad, costos, satisfacción del cliente, etc. según estos hayan sido programados, o en el peor de los casos a que estén muy cercanos de lo programado.

- El desarrollo rápido se pone en practica como una filosofía que lleva intrínseca la potencialización de la productividad y la elevación de los estándares a niveles mejores de los existentes.

- El desarrollo rápido nos desembaraza de los malestares propios de las practicas de desarrollo lentos, los cuales pueden ser cuantificables como perdidas monetarias y otras veces no como fracasos, perdida de confianza, perdida de reputación, perdida de clientes etc.

- La moral de los desarrolladores, gerencia, clientes y usuarios finales mejora notablemente a través de todo el proceso del proyecto, debido básicamente a la visibilidad del progreso y al cumplimiento del cronograma.

- El desarrollo rápido contribuye a la calidad del producto a través del ciclo de vida del mismo, debido a que las practica previstas por el desarrollo rápido incluyen el uso de tecnologías de soporte. Dentro de tales tecnologías tenemos por ejemplo la orientada al objeto, la cual es una herramienta poderosa de enfrentar la complejidad al momento de conceptualizar el problema, así mismo facilita y flexibiliza el diseño, para en la fase de construcción, construir el artefacto a partir de componentes reusables. Lo anteriormente expuesto trae como consecuencia artefactos más robustos, y acompañados de mejor documentación.

- Cuando un artefacto de software es producto de un proyecto de desarrollo rápido, la atención a los requerimientos en la fase de mantenimiento se da como una iteración más.

1.2. Problemas con el desarrollo rápido.

A continuación algunos problemas a ser abordados en el desarrollo rápido:

- Las mejoras en la velocidad del desarrollo no pueden darse instantáneamente, ni pueden obtenerse desempaquetando nuevos productos, o utilizando una nueva herramienta o método, lo que trae descontento y desanimo poco después de haberse optado por el desarrollo rápido. El desarrollo rápido requiere de un tiempo de maduración.

- Debido a que el desarrollo rápido no produce resultados en el corto plazo, ya que los gerentes o decisores en la implementación de nuevas practicas están enmarcados en el corto plazo es que se tiende a rechazar dicha implementación.

- La implementación del desarrollo rápido debe darse como una política a nivel corporativo, lo que implica que gerentes, desarrolladores, y clientes estén comprometidos en un alto grado, lo cual puede ser difícil de lograr.

- El compromiso de toda la estructura jerárquica envuelta en los proyectos de desarrollo, desde los gerentes de mas alto nivel hasta los desarrolladores de más bajo nivel esta en aceptar cambios radicales en su filosofía.

- Hay que enfrentar costos asociados con el entrenamiento, reeducación y a veces software.

- Generalmente el personal técnico es reacio a cambios, lo cual dificulta la implementación de nuevas practicas.

- La reusabilidad en la que se basa el desarrollo rápido, así como la confección de los componentes de software requieren de un gran esfuerzo.

RECOMENDACIONES

A continuación algunas recomendaciones:

- Al igual que con cualquier otra practica de desarrollo, en el desarrollo rápido el énfasis estará puesto en captar fundamentalmente las necesidades del negocio, a partir de donde nos abocaremos al sistema de información que mejor represente dichas necesidades, para luego dedicarnos al respectivo artefacto de software.

- El problema de desarrollo lento ha sido una constante a lo largo de la historia del desarrollo de software En la actualidad existe la necesidad por explorar nuevas tecnologías, así como combinaciones de las mismas, por lo que la evolución en la confección de artefactos de software deberá contemplar esfuerzos de desarrollo rápido.

- Las nuevas tendencias y complejidades de los actuales proyectos de software imponen que hagamos un alto en nuestras rutinarias tareas para abordar practicas que nos hagan más competitivos en la industria del desarrollo de software. El desarrollo rápido deberá ser visto como una opción, que adicione servicio a los artefactos de software desplegados en la forma de participación y satisfacción del cliente.

- Las diferentes practicas incluidas dentro del planeamiento para tener proyectos de desarrollo rápido dependerá específicamente del medio ambiente donde se efectué el desarrollo. Se deberá hacer una selección de las tecnologías y usarlas de acuerdo a la aplicación.

- El abordaje del desarrollo rápido es un reto que debe tomarse con la mejor predisposición por parte de las personas envueltas en este. Será necesario primero concientizar a las altas gerencias para luego difundir esta practica hacia abajo en la jerarquía organizacional. Lo anterior deberá ser hecho por etapas y tomarse siempre como un proceso de maduración.

BIBLIOGRAFIA

Karten, Naomi. *Managing Expectatións*. New York: Dorset House, 1994.

Whitaker, Ken. *Managing Software Maniacs*. New York: John Wiley & Sons 1994

Gordon, V. Scott, and James M. Bieman. "*Rapid Prototyping: Lessons Learned*", IEEE Software, (Enero 1995): 85-95.

Heckel, Paul. *The Elements of Friendly Software Design*. New York: Warner Books. 1991.

Mc Carthy, Jim. *Dynamics of Software Development*. Redmond, Wash.: Microsoft Press. 1995.

Van Genuchten, Michiel. "*Why is Software Late*?. IEEE Transactions on Software Engineering, vol 17, no. 6 (Junio-1991): 582-590.

Robert, Lafore. "*Object Oriented Programming in C++*". California: The White Group Press. 1995.

Dan Osier, Steve Grobman, Steve Batson. 1996. *Delphi 2*. Indianapolis: Sams Publishing.

Ken Reisdorph, Ken Henderson. 1997. *C++ Builder. Indianapolis*: Sams Publishing.

Brooks Frederick P. Jr. *The Mythical Man-Month*, Anniversary Edition. Reading, Mass.: Addison-Wesley. 1995.

Schultheis, Summer. *Management Information Systems: The Manager's View*. Arizona: IRWIN. 1994

James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Lorenzen. *Object-Oriented Modeling and Design*. New Yersey: Prentice-Hall Inc. 1991.

ANEXOS

A LOS CLÁSICOS 30 ERRORES ENUMERADOS

Efecto de los Errores en el Cronograma de Desarrollo

El uso de cualesquiera de las mejores prácticas es necesario pero no suficiente para alcanzar la máxima velocidad de desarrollo. Aún si hacemos algunas cosas correctamente, tal como usar extensivamente algunas de las prácticas modernas de programación, nosotros podríamos estar aún cometiendo algún error que anula la productividad ganada.

No se piense que cuando se habla de desarrollo rápido, todo lo que tenemos que hacer es identificar las raíces que son causa del desarrollo lento, eliminar estas y luego tendremos desarrollo rápido. El problema es que no sólo hay un puñado de raíces causas del desarrollo lento, y al final tratar de encontrar esas raíces no es muy útil.

El camino del desarrollo de software esta minado, y las minas con las que nos encontremos parcialmente determinaran cuan rápido o lento desarrollamos software.

En la industria del software una manzana podrida puede afectar a las demás. Para caer dentro del desarrollo lento, todo lo que necesitamos es cometer un error grande; y para alcanzar desarrollo rápido debemos evitar cualquier error grande.

Los Errores Clásicos

Los "errores clásicos" son aquellas prácticas inefectivas de software con predecibles malos resultados, que han sido elegidas comúnmente por mucha gente. La gran parte de estos errores tienen una apariencia seductiva.

- ¿Necesitamos rescatar un cronograma retrasado?
  Adicionemos mas personal.
- ¿Queremos reducir el cronograma?
  Seamos más agresivos con el cronograma.
- ¿Hay algún contribuyente clave del proyecto agravando al resto del equipo?
  Esperemos hasta que finalice el proyecto para despedirlo.
- ¿Tenemos que entregar un proyecto urgente?
  Incluyamos a los desarrolladores que tenemos disponibles ahora, y empecemos tan rápido como sea posible.

Los desarrolladores, administradores, y clientes usualmente tienen buenas razones para tomar las decisiones que ellos toman, y la seductiva atracción de los errores clásicos es parte de la razón de por que estos errores se cometen con mucha frecuencia. Pero debido a que estos se han cometido muchas veces, sus consecuencias se han convertido en fácilmente predecibles.
Los errores clásicos rara vez producen los resultados que se espera de ellos.

A continuación enúmero tres docenas de errores clásicos. El común denominador es que necesariamente no conseguiremos desarrollo rápido si evitamos estos errores, pero definitivamente caeremos en desarrollo lento sino los evitamos.

Una vez que se comprenda el efecto de estos errores en la velocidad del desarrollo, podremos usar una lista de ayuda en el planeamiento del proyecto y la gestión del riesgo.

Algunos errores clásicos relacionados al personal

A continuación algunos de los errores clásicos relacionados al personal.

1: **Motivación Socabada**. Estudios tras estudios han mostrado que la motivación tiene un gran efecto en la productividad y calidad que ningún otro factor. La gerencia debería tomar en cuenta pasos que eleven la moral del equipo de desarrollo a través del proyecto y eliminar aquellos que socaban la misma.

2: **Personal Débil**. Después de la motivación, las habilidades individuales de los miembros del equipo y/o las relaciones entre ellos como equipo, probablemente tienen las más grandes influencias en la productividad. La contratación de personal realizada de una manera no concienzuda basada por ejemplo en la disponibilidad del desarrollador, o de alguien que aparenta ser idóneo para el proyecto, en vez de ser hecha en base a quien puede completar el mayor trabajo sobre la vida del proyecto, amenaza todo tipo de esfuerzo puesto en el desarrollo rápido. La práctica mencionada nos permite un rápido comienzo del proyecto pero no establece las bases adecuadas para un desarrollo rápido.

3: **Problemas Incontrolados del Personal**. El no acertar en el trato con problemas del personal también amenaza la velocidad del desarrollo. El no tomar acciones para enfrentar los problemas de personal es el reclamo más común que los miembros de un equipo tienen con sus lideres. En infinidad de ocasiones la manzana podrida es conocida dentro del equipo, pero el líder del equipo no hace nada para resolver la situación.

4: **Estoicismo**. Algunos desarrolladores de software ponen un gran énfasis en el estoicismo dentro del proyecto, pensando que este puede ser beneficioso. Lo real es que el estoicismo es más malo que bueno. Un énfasis en estoicismo hace que el proyecto sea propenso a riesgos extremos y desanima la cooperación entre los demás

responsables en el proceso de desarrollo de software. En estos casos generalmente los jefes no saben que ellos necesitan tomar acciones correctivas sino hasta que el daño ha sido hecho. En este caso el énfasis es puesto en las actitudes de los desarrolladores en vez de concentrarse en progresos uniformes, consistentes y significativos.

5: **Adicionar Personal a un Proyecto Retrasado**. Posiblemente este sea el error más clásico de los clásicos. Cuando el proyecto esta retrasado, el adicionar personal puede disminuir la productividad de los miembros existentes del equipo antes que aumentarla. Hay quienes se refieren a este error como echar gasolina al fuego.

6: **Medio Ambiente no Aparente**. Los desarrolladores (trabajadores en general) que ocupan oficinas privadas, sin bullicio, convenientemente orientadas, ventiladas etc. tienen una mejor performance que si trabajaran en cubículos o en oficinas apiñadas o con ruido, lo que trae como consecuencia que el cronograma de desarrollo se alargue.

7: **Fricción entre Desarrolladores y Clientes.** La fricción entre desarrolladores y clientes puede darse de diversas maneras. Los clientes pueden sentir que los desarrolladores no cooperan cuando estos rechazan los acortes en el cronograma o cuando estos fallan en desplegar en la fecha prometida. Los desarrolladores pueden sentir que los clientes insisten sin razón acerca de los cronogramas, o cambios de requerimientos después que los requerimientos han sido establecidos. También podría haber simplemente conflictos de personalidad entre los dos grupos.

Los efectos primarios de esta fricción es la comunicación pobre, y los efectos secundarios de la comunicación pobre incluye los requerimientos pobremente comprendidos, diseño pobre de la interface del usuario, y en el peor caso los clientes rehusándose a aceptar el producto completo. En promedio la fricción entre clientes y desarrolladores puede ser tan severa que ambas partes consideren la cancelación del proyecto. Estas fricciones toman tiempo para ser superadas y esto distrae a ambos clientes y desarrolladores del trabajo real del proyecto.

8: **Expectativas Irreales.** Una de las causas más comunes de fricción entre los desarrolladores y sus clientes son las expectativas irreales. Muchos de los problemas en el desarrollo de software, especialmente en lo concerniente a la velocidad de desarrollo tienen su origen en espectativas irreales no declaradas. Es nuestro interés el tratar de establecer las expectativas explícitamente de tal manera que demos luz a cualquiera de las expectativas irreales que nuestros clientes puedan tener acerca del cronograma o de los entregables. Aunque las expectaciones irreales no alargan por si mismas el cronograma de desarrollo, estas contribuyen a la percepción de que el cronograma es demasiado largo y eso puede casi siempre ser malo.

9: **Falta de Respaldo Efectivo en el Proyecto.** Es necesario un alto nivel de respaldo para soportar muchos aspectos del desarrollo rápido, incluyendo un planeamiento realista, control de cambios y la introducción de nuevas prácticas de desarrollo. Sin un respaldo ejecutivo efectivo, otros funcionarios de alto nivel en la organización pueden forzar para que aceptemos fechas de entrega irreales, o efectuar cambios que debiliten el proyecto.

10: **Falta de Responsables Comprometidos.** Todos los responsables en el esfuerzo de desarrollo de software deben estar altamente comprometidos con el proyecto. Esto incluye el aval ejecutivo, el líder del equipo, los miembros del equipo, el staff de marketing, usuarios finales, clientes, y cualquier otro que tenga responsabilidad en el proyecto. La cooperación estrecha que ocurre solo cuando tenemos un compromiso de todos los responsables nos permite una precisa coordinación de esfuerzos para el desarrollo rápido que es imposible obtener sin un compromiso de buen nivel.

11: **Falta de Interacción del Usuario.** Para todos es bien sabido que una de las razones de por que los proyectos de sistemas de información tienen éxito, es debido a la interacción del usuario. Los proyectos sin una interacción temprana del usuario ponen en riesgo la buena comprensión de los requerimientos y son vulnerables a cambios futuros en el proyecto lo cual consume recursos y tiempo.

12: Pensamientos Positivos. Hay una gran número de problemas que caen dentro de esta especificación. Cuantas veces hemos escuchado declaraciones como las que siguen de diferentes personas:

"Ninguno de los miembros del equipo realmente creyeron que podrían completar el proyecto de acuerdo al cronograma dado, pero ellos pensaban que si trabajaban duro, y todo iba bien ellos podrían tener éxito."

"Nuestro equipo no ha hecho mucho esfuerzo para coordinar las interfaces entre las diferentes partes del producto, pero hemos estado comunicándonos bien en cuanto a otros productos, y las interfaces son relativamente simples, así que probablemente nos tomara uno o dos días para depurar todo."

"Sabemos que el contratista del subsistema de la base de datos iba a tener dificultades para completar el trabajo con los niveles del staff que ellos especificaron en la propuesta. Ellos no tenían tanta experiencia como alguno de los otros contratistas, pero pueda ser que compensen con energía su falta de experiencia. Probablemente ellos desplegaran el artefacto a tiempo."

"No necesitamos mostrar al cliente la ronda final de cambios del prototipo. Estoy seguro que conocemos lo que ellos quieren."

"El equipo dice que se tendrá que hacer un esfuerzo extraordinario para cumplir con la fecha fijada, ellos no cumplieron con la primera parte por unos pocos días, pero pienso que pueden acabar a tiempo."

Pensamiento positivo no es justamente optimismo. Es cerrar los ojos y esperar que algo trabaje cuando no tenemos un fundamento razonable para pensar que esto sucederá. El pensamiento positivo en el comienzo de un proyecto nos lleva a graves problemas al

final del mismo. Este socaba el planeamiento y puede ser la raíz de un gran número de problemas difíciles de enfrentar.

Algunos errores clásicos relacionados a los Procesos

Los errores relacionados a los procesos retrasan los proyectos debido a que estos socaban los talentos y esfuerzos de los desarrolladores. A continuación algunos de los peores errores relacionados a los procesos.

13. **Insuficiente Gestión del Riesgo.** Algunos errores no son suficientemente comunes para ser considerados clásicos. Esos son llamados "riesgos". Como con los errores clásicos, si no hacemos una gestión adecuada de los riesgos, solo una cosa tiene que estar mal para cambiar el proyecto de un desarrollo rápido a uno lento: "la falla de gestionar el riesgo de tal único error es un error clásico".

14. **Optimismo Excesivo en los Cronogramas.** Los retos enfrentados por alguien construyendo una aplicación de 3 meses, son completamente diferentes a los retos enfrentados por alguien construyendo una aplicación de un año de duración. Estableciendo un cronograma excesivamente optimista pone en riesgo el proyecto sometiendo su alcance, mellando el planeamiento efectivo y abreviando las actividades criticas de comienzo del proyecto, tales como el análisis de requerimientos y el diseño. Esto también pone demasiada presión en los desarrolladores, lo cual afecta la moral y productividad de los desarrolladores en el largo plazo.

15. **Falla del Contratista.** Las compañías algunas veces contratan piezas de un proyecto cuando están muy apuradas para hacer el trabajo internamente. Pero los contratistas frecuentemente despliegan productos tarde, de inaceptable baja calidad, o que no cumplen con las especificaciones. Los riesgos tales como requerimientos inestables, o

interfaces mal definidas pueden ser magnificadas cuando se incluye un contratista en la escena. Si las relaciones con el contratista no son manejadas cuidadosamente, el uso de contratistas puede enlentar un proyecto en vez de hacerlo más veloz.

16. **Planeamiento Insuficiente.** Si no se planea para alcanzar desarrollo rápido, no podemos esperar alcanzarlo.

17. **Abandono del Planeamiento Bajo Presión.** Los equipos del proyecto hacen planes y luego rutinariamente los abandonan cuando se ven en problemas con el cronograma. El problema no reside en el abandono del plan, sino en fallar en la creación de un plan alternativo, para luego caer en simplemente codificar. Generalmente los equipos abandonan los planes después que fallan con una fecha de entrega. La consecuencia de esto es la continuación del proyecto de una manera no coordinada.

18. **Desperdicio de Tiempo Durante el Inicio Difuso.** El "Inicio Difuso" es el período previo al comienzo del proyecto. Es el tiempo normalmente empleado en los procesos de aprobación y presupuestación. No es raro para un proyecto demorar meses o años en el inicio difuso para luego darle certificado de nacimiento con un cronograma agresivo. Es mucho más fácil, barato, y menos riesgoso ahorrar unas pocas semanas o meses al inicio difuso, que comprimir el cronograma de desarrollo por la misma cantidad de tiempo.

19. **Recortes en las Actividades Iniciales.** Los proyectos que están en apuros son sensibles al recorte de sus actividades no esenciales, y desde que el análisis de requerimientos, arquitectura, y diseño no producen directamente código estos son blancos fáciles. El líder del equipo me dijo, "que no teníamos tiempo para hacer el diseño".

    Los resultados de este error también conocido como "saltar a codificar" son todos predecibles. Estudios muestran que los proyectos que obvian las actividades iniciales,

comúnmente tienen que hacer el mismo trabajo en las postrimerías del proyecto, a un costo comprendido entre 10 a 100 veces el haberlo hecho apropiadamente la primera vez. Si no podemos dedicar 5 horas para hacer lo correcto la primera vez, ¿podremos más tarde dedicar 50 horas para hacer el trabajo de una manera apropiada?

20. **Diseño Inadecuado.** Un caso especial de recortes en las actividades iniciales es el diseño inadecuado. Los proyectos que son urgentes mellan el diseño no asignando tiempo suficiente para esta actividad, lo cual crea un ambiente de presión que dificulta las concienzudas consideraciones alternativas de diseño. El énfasis en el diseño se da en la conveniencia antes que en la calidad, por lo que a la larga nos veremos en la necesidad de enfrentar muchos ciclos de diseño antes de que podamos finalizar el sistema, lo cual es un desperdicio de tiempo.

21. **Recortes en Quality Assurance.** Los proyectos urgentes comúnmente tratan de obtener ahorros eliminando las revisiones de diseño y código, eliminando el planeamiento de las pruebas, y efectuando solo pruebas superficiales. Generalmente las revisiones se acortan para lograr ventajas en el cronograma, pero a la larga tales ventajas no compensan la dedicación que debe hacerse para conseguir el sistema completamente depurado. Este resultado es típico. Acortando un día de Quality Assurance en los comienzos del proyecto, probablemente nos costara de entre 3 a 10 días de actividad en las postrimerías del proyecto. Este recorte socaba la velocidad de desarrollo.

22. **Frecuente Convergencia Prematura o Excesiva.** Justo antes de que un producto es cronogramado para ser desplegado, hay presión para preparar el producto para su emisión, desactivando las ayudas de depuración dentro del código, afinar y mejorar la performance, removiendo o desactivando las características parciales que no pueden ser completadas al tiempo del despliegue, implantar versiones rápidas de características que absolutamente deben ser completadas al tiempo de despliegue, imprimir la documentación final, incorporar el sistema final de ayuda, pulir el programa de

instalación, etc. En proyectos urgentes existe la tendencia a forzar la convergencia temprano. Desde que no es posible forzar el producto a converger cuando se quiera, algunos proyectos de desarrollo rápido tratan de forzar la convergencia media docena de veces o más antes que ellos finalmente alcancen el éxito.

23. **Planeamiento Para Recuperarse mas Adelante.** Una clase de reestimación es responder erróneamente a un desliz en el cronograma. Si estamos trabajando en un proyecto de 6 meses, y nos toma 3 meses para alcanzar el hito de 2 meses, Que hacemos?. Muchos proyectos simplemente planean recuperarse mas tarde, pero nunca lo hacen. Es natural aprender mas del producto conforme lo construimos, incluyendo el aprender más acerca de lo que se tomará para construirlo. Ese aprendizaje necesita ser reflejado en el cronograma reestimado.

    Otra clase de errores de reestimación tienen su fuente en los cambios del producto. Si el producto que estamos construyendo cambia, el monto de tiempo que necesitamos para construirlo también cambia. Muchas veces hay cambios de requerimientos sustanciales entre la propuesta original y el inicio del proyecto sin una reestimación correspondiente del cronograma y recursos. Apilando nuevas características sin ajustar el cronograma garantiza que no desplegaremos el producto a tiempo.

Algunos errores clásicos relacionados al Producto

A continuación algunos errores clásicos relacionados al Producto.

24. **Requerimientos Dorados y Plateados.** Algunos proyectos tienen mas requerimientos de lo que estos necesitan desde el mismo comienzo. Con frecuencia la performance es declarada como un requerimiento mas comúnmente de la que esta necesita ser, y eso puede innecesariamente alargar el cronograma de desarrollo. Los usuarios tienden a estar menos interesados en las características complejas que los desarrolladores, y tales complejidades adicionan desproporción a los cronogramas.

25. **Cambios Repentinos.** Aún si tenemos éxito evitando requerimientos dorados y plateados, los proyectos en promedio experimentan cerca de un 25 por ciento de cambios en requerimientos a lo largo del desarrollo. Tales cambios pueden producir al menos un 25 por ciento de adición al cronograma de software, lo cual puede ser fatal para un proyecto de desarrollo rápido.

26. **Desarrolladores Dorados y Plateados.** Los desarrolladores están fascinados por la nueva tecnología y algunas veces están ansiosos de probar nuevas características del lenguaje, ambiente, o crear una implementación propia de alguna característica que han visto en otro producto, sea este requerido o no en su producto. El esfuerzo requerido para diseñar, implementar, probar, documentar y dar soporte a características que no son requeridas, alargan el cronograma.

27. **Negociación Sacar y Poner.** A veces ocurre que después de negociar con la gerencia, y conseguir la aprobación de un desliz en el cronograma de un proyecto que se esta desarrollando uniformente mas lento de lo esperado, se adiciona completamente nuevas tareas después que se ha cambiado el cronograma. La razón para que esto se dé es difícil de saber, debido a que el gerente quien aprueba el desliz en el cronograma esta implícitamente reconociendo que el cronograma estuvo errado. Pero una vez que el cronograma ha sido corregido, la misma persona toma acciones explícitas para hacerlo otra vez incorrecto. Esto no puede ayudar si no que mella el cronograma.

28. **Desarrollo Orientado a la Investigación.** Investigaciones en proyectos de desarrollo de software han dado como resultado alertas en cuanto a las cercanías de los límites de la ciencia de la computación. Si el proyecto requiere la creación de nuevos algoritmos o nuevas prácticas de computación, caeremos en el campo de la investigación del software antes que en el campo del desarrollo del mismo. Los cronogramas de desarrollo de software son razonablemente predecibles; los cronogramas en investigación de software ni siquiera son teóricamente predecibles.

Si tenemos como metas del producto usar sofisticados algoritmos, alta velocidad en el uso de la memoria, etc. deberíamos asumir que el cronograma es altamente especulativo. Si vamos a hacer uso de sofisticadas técnicas y tenemos alguna debilidad en el proyecto como falta de personal, debilidad en el personal, requerimientos vagos, interfaces no estables con contratistas terceros; debemos desistir de predecir el cronograma.

Algunos errores clásicos relacionados a la Tecnología

Los siguientes errores clásicos están relacionados al uso y mal uso de tecnología moderna.

29. **El Síndrome de la Bala de Plata (Panacea).** Comúnmente se pone demasiada confianza en los beneficios declarados de tecnologías no usadas (generadores de reportes, lenguajes de quinta generación, tecnologías orientadas al objeto, etc.) y en la poca información acerca de cuan bien estos trabajaran en el ambiente particular de desarrollo. Cuando los equipos se avocan hacia una única nueva práctica, nueva tecnología, o proceso rígido y esperan que esto solucione los problemas de sus cronogramas ellos son inevitablemente defraudados.

30. **Cambio de Herramientas a la Mitad del Proyecto.** Esta es un viejo recurso que difícilmente alguna vez trabaja. Algunas veces puede tener sentido mejorar incrementalmente de la versión 3.0 a la 3.1 o algunas veces aún a la versión 4. Pero la curva de aprendizaje, el rehacer nuevamente, y los errores inevitables hechos con una herramienta completamente nueva, usualmente cancela cualquier beneficio cuando se esta en la mitad del proyecto.

31. **Sobreestimados Ahorros en Nuevas Herramientas o Métodos.** Las organizaciones rara vez mejoran sus productividades a pasos gigantes, no importa cuantas herramientas nuevas o métodos estas adopten, o cuan buenos sean dichas herramientas o métodos. Los beneficios de nuevas prácticas son parcialmente desplazadas por las curvas de aprendizaje asociadas a dichas prácticas, y aprender a usar nuevas prácticas para obtener la máxima ventaja toma tiempo. También las nuevas prácticas llevan consigo riesgos nuevos, los cuales probablemente serán descubiertos solo usando tales prácticas. Probablemente experimentaremos una uniforme y lenta mejora en el orden de un despreciable porcentaje por proyecto en vez de experimentar dramáticas ganancias. En muchos casos hay que ser conservativos y esperar a los mas un 5 por ciento de ganancia en productividad, en vez de asumir que doblaremos la productividad.

32. **Falta de Control Automatizado del Código Fuente.** La falla a usar el control automatizado de código fuente expone al proyecto a riesgos innecesarios. Sin este, los desarrolladores tienen que coordinar su trabajo manualmente. Ellos pueden convenir en poner la ultima versión de cada archivo dentro de un directorio maestro y asegurarse cada uno con los demás antes de copiar archivos dentro del directorio. Pero alguno podría olvidar copiar la ultima versión, o alguno podría reescribir el trabajo de algún otro. Otros desarrolladores crean nuevo código a partir de interfaces desactualizadas, y luego tienen que rediseñar el código cuando descubren que usaron la versión errada de la interface. Los usuarios reportan defectos que no podemos reproducir debido a que no tenemos manera de reprodrucir la versión que ellos estuvieron usando.

B PRÁCTICAS DE QUALITY ASSURANCE

Quality Assurance tiene dos propósitos principales. El primer propósito es asegurarse que el producto desplegado tiene un nivel aceptable de calidad. Aunque ese es un propósito importante, este esta fuera del alcance del presente estudio. La segunda función de quality assurance es detectar errores en el escenario cuando estos consumen menos tiempo (y será menos costoso) para ser corregidos. Esto casi siempre significa detectar errores lo más cercano posible al momento cuando son introducidos. Cuando más tiempo un error permanece en el producto, se consumirá mas tiempo (y será más costoso) para removerlo. Quality assurance es por lo tanto una parte indispensable de cualquier programa serio de desarrollo rápido, debido a que provee soporte critico para la velocidad máxima de desarrollo.

Cuando un producto de software tiene demasiados defectos, los desarrolladores ocupan mas tiempo arreglando el programa que lo que ellos emplearon escribiendo este. Muchas organizaciones han encontrado que es mejor no instalar los defectos desde el principio. La clave para no instalar defectos es poner atención a los fundamentos de quality-assurance desde el primer día.

Algunos proyectos tratan de ahorrar tiempo reduciendo el tiempo empleado en las prácticas de quality-assurance tales como revisión del diseño y el código. Otros proyectos que están retrasados tratan de actualizarse comprimiendo el cronograma de pruebas, el cual es vulnerable a reducciones debido a que este es usualmente el camino critico al final del proyecto. Estas son algunas de las peores decisiones que puede hacer una persona para maximizar la velocidad de desarrollo.

Proyectos que son urgentes debido a que están retrasados son particularmente vulnerables a recortes en quality assurance al nivel del desarrollador individual. Cuando estamos en apuros digamos a "solo 30 días de desplegar", en vez de escribir un modulo de impresión completamente nuevo y separado de seguro que lo haremos será apiñar código en el modulo de consulta por pantalla. De hecho sabemos que eso es un mal diseño, que es difícil de extender o mantener, pero no tenemos tiempo para la confección de un modulo de la manera correcta. Estamos bajo presión para conseguir el producto a tiempo, así que nos sentimos obligados a tomar el atajo.

Tres meses mas tarde, el producto aún no ha sido desplegado, y esos atajos regresan a incomodarnos. A este punto encontramos que los usuarios no están felices con la impresión, y que la única manera de satisfacer sus requerimientos es extender significantemente la funcionalidad de la impresión. Desafortunamente la funcionalidad de la impresión y la funcionalidad de consultas por pantalla se han entrelazado entre sí. El rediseño de la impresión y la separación de esta de la consulta por pantalla, es en este momento una operación difícil, que consume tiempo, y propensa a error.

El punto principal es que un proyecto que fue supuesto a poner un fuerte énfasis en alcanzar el cronograma mas corto posible, esta desperdiciando tiempo de la siguientes maneras:

- El tiempo original empleado en el diseño e implementación del código de impresión dentro del modulo de consulta fue completamente desperdiciado debido a que gran parte de este código será desechado. De la misma manera el tiempo empleado en pruebas de la unidad, y depuración de dicho modulo modificado ha sido también desperdiciado.
- Tiempo adicional debe ser empleado para limpiar el código especifico de la impresión en el módulo de la consulta.

- Tiempo adicional en pruebas y depuración en el módulo modificado de consulta debe ser empleado para asegurarse que el código modificado aún trabaja, después de que el código para impresión haya sido limpiado.
- El nuevo modulo de impresión, el cual debió haber sido diseñado como una parte integral del sistema desde el principio, tiene ahora que ser diseñado alrededor y dentro del sistema existente, el cual no fue diseñado con esto en mente.

Todo esto sucede, cuando el único costo necesario si es que se hubiera tomado la decisión correcta desde un principio, hubiera sido diseñar e implementar una versión del modulo de impresión.

Este ejemplo no es raro. Los proyectos que están en problemas con el cronograma comúnmente están obsesionados con trabajar mas duro en vez de trabaja de manera más inteligente. En estos casos la atención a la calidad es vista como un lujo. El resultado es que el proyecto sufre de altibajos, con lo cual el problema del cronograma se agudiza.

Una decisión temprana en el proyecto de no enfocar la detección de defectos, o una decisión de posponer la detección de defectos para después en el proyecto cuando esto será mas caro y un excesivo consumo de tiempo es una decisión irracional en el desarrollo rápido.

Si podemos prevenir defectos, o detectarlos y removerlos lo mas temprano posible, nos dará un beneficio significante en el cronograma. Como regla general estudios muestran que cada hora dedicada en la prevención de defectos, reducirá el tiempo de reparación en el rango de 3 a 10 horas. En el peor de los casos, el reparar problemas de requerimientos una vez que el software esta operativo típicamente costara de 50 a 200 veces lo que se hubiera gastado en reparación en la fase de requerimientos. Dado que cerca del 50 por ciento de todos los defectos usualmente existen en la fase de diseño, se pueden ahorrar enormes montos de tiempo detectando defectos mas temprano que en la etapa de pruebas.

Módulos Propensos a Error

Un aspecto de quality assurance que es particularmente importante al desarrollo rápido es la existencia de módulos propensos a error. Un modulo propenso a error es un modulo que es responsable por un desproporcionado número de defectos. Por ejemplo IBM ha encontrado en sus proyectos de desarrollo que en promedio el 57 por ciento de los errores estaban concretados en el 7 por ciento de los módulos. Otros estudios han mostrado que cerca del 20 por ciento de los módulos en un programa son típicamente responsables por cerca del 80 por ciento de los errores.

Los módulos propensos a error tienden a ser más complejos que otros módulos en el sistema, menos estructurados, y usualmente largos. Comúnmente estos módulos son desarrollados bajo una presión excesiva del cronograma y no son completamente probados.

Para el desarrollo rápido es necesario hacer una práctica de alta prioridad el identificar y rediseñar los módulos propensos a error. Por ejemplo si la tasa de errores de un modulo es cerca de 10 defectos por cada 1000 líneas de código, revisémoslo para determinar si es que este debería ser rediseñado o reimplementado. Si dicho modulo esta pobremente estructurado, excesivamente complejo o excesivamente largo, rediseñemos el modulo y reimplementémoslo desde cero, con lo cual ahorraremos tiempo y mejoraremos la calidad del producto.

Pruebas

La práctica más común de quality assurance es indudablemente la prueba de ejecución, encontrar errores ejecutando un programa y observando lo que este hace. Las dos clases básicas de pruebas de ejecución son pruebas de unidades, en las que el desarrollador verifica que su código trabaje correctamente, y pruebas del sistema, en el que un especialista independiente prueba el sistema para ver si este funciona como se esperaba.

La efectividad de las pruebas varıa enormemente. Las pruebas de unidades pueden encontrar de entre 10 a 50 por ciento de defectos en un programa. Las pruebas del sistema pueden encontrar de entre 20 a 60 por ciento de los defectos de un programa. Conjuntamente la tasa de detección de errores es comúnmente menos del 60 por ciento. El resto de los errores son encontrados por técnicas de detección de errores tales como las revisiones o por los usuarios finales después que el software ha sido puesto en producción.

Las pruebas son la oveja negra de las prácticas QA en lo que concierne a la velocidad de desarrollo. Estas pueden hacerse de una manera tan burda que enlenta el cronograma de desarrollo, pero comúnmente su efecto en el cronograma es solo indirecto. Son las pruebas las que descubren que la calidad del producto no es de buen nivel para ser desplegado, y que el producto tiene que ser retrasado hasta que este sea mejorado. De esta manera las pruebas se convierten en el mensajero con malas noticias que afectan el cronograma.

La mejor manera de mejorar las pruebas desde un enfoque de desarrollo rápido es planear por adelantado por malas noticias, y establecer las pruebas de tal manera que si riesgos para desplegar el producto los especialistas puedan hacerlo notar tan rápido como sea posible.

Revisiones Técnicas

Las revisiones técnicas incluyen todas las clases de revisiones que son usadas para detectar defectos en requerimientos, diseño, código, etc. Las revisiones varían en el nivel de formalidad y en efectividad, y juegan un rol mas critico en la velocidad de desarrollo que las pruebas. Las revisiones técnicas son útiles e importantes suplementos a las pruebas. Las revisiones tienden a encontrar diferentes tipos de errores que las pruebas no pueden. Estas detectan errores mas temprano, lo que es saludable para el cronograma. Las revisiones son más efectivas en cuanto a costo en una base de defecto encontrado, debido a que estas detectan ambos el síntoma del defecto y la causa detrás de este al mismo tiempo. Las pruebas detectan solo los sintamos del defecto, y el desarrollador aún tiene que encontrar la causa para depurarlo. Las revisiones tienden a encontrar un porcentaje mas alto de defectos. Las revisiones proveen un fórum para que los desarrolladores compartan su conocimiento de las mejores prácticas, lo cual mejora sus habilidades de desarrollo rápido en el tiempo. Las revisiones técnicas son un componente critico de cualquier esfuerzo de desarrollo que esta tratando de alcanzar el cronograma más corto posible.

A continuación se tratan las clases más comunes de revisiones.

Ensayos de Guión

Esta práctica se refiere a cualquier reunión en la cual dos o más desarrolladores revisan el trabajo técnico con el propósito de mejorar su calidad. Estas revisiones son útiles al desarrollo rápido debido a que las podemos usar para detectar defectos mucho antes de la etapa de pruebas. Lo más temprano que las pruebas pueden detectar un defecto de requerimientos es por ejemplo, después que el requerimiento ha sido especificado, diseñado, y codificado. Un ensayo de guión puede detectar un defecto de requerimientos al momento de la especificación antes de que cualquier trabajo de diseño o codificación haya sido hecho.

Los ensayos de guión pueden encontrar entre 30 a 70 por ciento de los errores en un programa.

Lecturas de Código

Las lecturas de código son de alguna manera procesos de revisión más formal que los ensayos de guión pero se aplica nominalmente al código. En la lectura del código, el autor del código proporciona código fuente a dos o más especialistas. Los especialistas leen el código y reportan cualquier error al autor del código. Un estudio en el Laboratorio de Ingeniería de Software de la NASA encontró que las lecturas de código detectaron cerca de el doble de defectos por hora de esfuerzo que las pruebas. Lo anterior sugiere que en un proyecto de desarrollo rápido, alguna combinación de lectura de código y pruebas seria más efectivo para el cronograma que el uso de únicamente pruebas.

Inspecciones

Las inspecciones son una clase de revisiones técnicas formales que se han mostrado extremadamente efectivas en la detección de defectos a través de un proyecto. Con las inspecciones los desarrolladores reciben entrenamiento especial en inspecciones y juegan específicos roles durante la inspección. El "moderador" proporciona el trabajo a ser inspeccionado antes de la reunión de inspección. Los "inspectores" examinan el trabajo antes de la reunión y usan listas de chequeo que estimulen sus revisiones.

Durante la reunión de inspección, el "autor" usualmente parafrasea el material en inspección, los inspectores identifican errores, y el "escribano" registra los errores. Después de la reunión el moderador produce un reporte de inspección que describe cada defecto e indica que se hará en cada caso. A través del proceso de inspección se recopila datos acerca de defectos, horas empleadas corrigiendo defectos de tal manera que se puede analizar la efectividad de los procesos de desarrollo de software y mejorarlos.

Al igual que con los ensayos de guión se pueden usar inspecciones para detectar defectos más temprano que con las pruebas. Se pueden usar para detectar errores en requerimientos, prototipos de interface de usuario, diseño, código, etc. Las inspecciones encuentran de entre 60 a 90 por ciento de los defectos en un programa lo que es considerablemente mejor que con ensayos de guión o pruebas. Debido a que las inspecciones pueden usarse temprano en el ciclo de desarrollo, estas producen ahorros netos en el cronograma de entre 10 a 30 por ciento.